Fundamentals of container clouds with Rahti

Exercises

1. Authorizing client session and creating a project

Create a project named `course-training-<number>` using the `oc` command line interface (CLI) tool.

Locate the project in the web console and verify that it exists using the `oc` tool.

2. Execute a container in a pod

Pre-requisites: *Exercise 1*

The Docker's hello-world

- Run container image `hello-world` in a pod called hello-pod.
- Get the standard output of the container.
- Verify that the container is crash-looping. Fix it so that it isn't. Replace the container with fixed pod specification.

Custom "Hello, world"

- Run command `echo Hello, world!` inside container from image `alpine:edge`. Name the pod as `custom-hello-pod`.
- Verify that the standard output of the container really is "Hello, world!".

Sleeping pod

- Create a pod based on `alpine:edge` image that sleeps for 7200 seconds and then exits. Name it `sleeping-pod`.

Cleanup

- Remove the pods named `hello-pod` and `custom-hello-pod`. Let the sleeping pod sleep.

3. Navigate Rahti Console

Pre-requisites: *Exercise 1, 2*

Terminal session

Open terminal session to your sleeping pod using the Rahti GUI console.

- What is the user id? (`id -u`)
- What is the group id? (`id -g`)
- Try creating file `/hello` containing string `hello`. Did it work? (`echo hello > /hello`)
- Try that with `/tmp/hello`
- Kill the process that sleeps. Did the pod disappear?

Cleanup

- Remove the pod `sleeping-pod` using console GUI

4. Python application in Rahti

Pre-requisites: 1

Create a Python 3.6 application using the service catalog template named "Python".

For sources code of the application, make a fork of the code located at `https://github.com/cscfi/rahti-flask-hello` under your own github account.

Place the application in the project you created in Exercise 1.

Name the application `hello-flask-<number>`, where `<number>` is the number in your training account.

Once the application is running, find the URL where it is available in the internet.

5. Liveness and Readiness probes

Pre-requisites: *Exercise 1*

The following pod waits for 30 seconds, then creates the file `/tmp/alive` inside the container, then waits again for 30 seconds and deletes the file.

- Edit the specification so that the container is "ready" and "live" only if the file `/tmp/alive` exists.
- Wait for 30 seconds before starting to check if the container is live.
- Try what happens if the liveness probing starts immediately when the container is started?
- **Cleanup**: Delete the pod from the cluster afterwards with `oc delete pod probe-tests`.

Tip #1: Go to Monitoring page on the Application Console to see the status of your pod.

Tip #2: You can monitor the pod with `watch oc describe pod probe-tests` or `watch oc get pod probe-tests`

Tip #3: The GNU coreutils tool *`cat <filename>`* returns 0 if the exists and everything went fine.

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: probe-tests
spec:
  containers:
  - name: probe-test-container
    image: centos:7
    command:
    - sh
    - -c
    - >
      echo "Waiting for 30 seconds to go live" &&
      for i in {1..30}; do echo "."; sleep 1; done &&
      touch /tmp/alive &&
      echo "Now waiting for 30 seconds to go die" &&
      for i in {1..30}; do echo "."; sleep 1; done &&
      rm /tmp/alive &&
      echo "Going to sleep mode" &&
      sleep inf
```

6. OOM killer

Create pod yaml-spec that runs the image `docker-registry.default.svc:5000/rahti-course-2019/oom-killer:1`. That image contains a python code, listed in `app/app.py` file, that reserves 10MB more memory every second.

Make sure that the container gets killed when it reserves over 50MB of memory.

7. Hello world web server with DeploymentConfig

Pre-requisites: *Exercise 1*

In this exercise, write (or copy-paste from slides) all the API objects in YAML plaintext and submit them with `oc create -f ...`.

DeploymentConfig

- Create a DeploymentConfig that will spawn a pod running image `openshift/hello-openshift`. Name it `hello-openshift` and label it `app: hello-openshift`. Apply the same label `app: hello-openshift` to the pods to be spawned as well.
- What is the name of the pod that appeared? *Hint:* `oc get pod -l app=hello-openshift`.
- Delete the pod.
- What is the name of the pod that appeared?
- List all objects that have metadata label `app: hello-openshift`. What *Kinds* of objects are listed?

Service

- Create a Service object that will redirect traffic internally to the pod.

Route

- Expose the Service to internet at 'hello-rahti-##.rahtiapp.fi', where ## is the number of the training account you are holding.
- Secure the route with TLS edge termination policy and redirect insecure traffic to the secure one.

Cleanup: DeploymentConfig

- If you remove DeploymentConfig, what will happen to the corresponding ReplicationController and Pods?
- Remove DeploymentConfigs. Did you guess correctly?
- *Bonus*: Why did it happen?

Cleanup: The rest

- Remove the Route and the Service objects.

8. Application data on persistent volume

Pre-requisites: *Exercise 4*

In the following exercises you are expected to modify the YAML specification directly with the Rahti Application Console. At the Application Console page, the specification can be found by clicking "Applications → Deployments", then the name of the Deployment and finally clicking the upper right corner "Actions → Edit YAML".

This exercise comprises of three parts:

1. Create and mount a persistent volume with the picture https://rahti-course-nov-2019.a3s.fi/kitten.jpg to the application `hello-flask-#` made in exercise 4 at `/opt/app-root/src/static`.

   Navigate to `http://hello-flask-#-course-training-#.rahtiapp.fi/kitten` and you should see a kitten.

   Note: It doesn't matter in which order you mount and copy, as long as you end up with pods with the file `/opt/app-root/src/static/kitten.jpg`.

   Tip #1: Storage can be added and mounted to the DeploymentConfig with the web console.

   Tip #2: `oc cp kitten.jpg <podname>:...` or `oc rsh dc/hello-flask-#` and `curl -L -O <url>`

2. Create a configmap from file `custom.json` with the following contents: "'json { "greeting": "Custom Hello from custom.json" }

   Tip #3: `oc create configmap ...` "'

3. Create a secret and use that secret to bring environment variable "PASSWORD=secretPassword" to the application.

   You may change the password above to your liking.

   Tip #4: `oc create secret generic ...`

Loading urls http://hello-flask-#-course-training-#.rahtiapp.fi/kitten/ and http://hello-flask-#-course-training-#.rahtiapp.fi/secret-kitten/secretPassword (with # being the training account number entered earlier) should now display kittens.

9. Webhook to hello-flask

Pre-requisites: *Exercise 4*.

You can now configure GitHub webhook for your hello-flask application, this will trigger your application builds automatically when there are push events in your GitHub code.

Configure a GitHub webhook to the fork repository of rahti-flask-hello you made earlier in Exercise 4.

- To find out the secret in the webhook payload look at the BuildConfig of the application: `oc get bc hello-flask-# -o yaml` and look for element `github.secret` in the array `spec.triggers`, where `#` is your training account number.
- ***Tip #1***: Get the payload URI w/o the secret with `oc describe bc rahti-flask-#`.
- Edit app.py: Change line 6 to `DefaultTitle="Application from Student #"` where # is your training account number.
- Commit changes, verify that new build for your application is triggered in Rahti.

10. BuildConfigs and Triggers

Pre-requisites: *Exercise 1*.

Inlining Dockerfiles for the base image

Create a BuildConfig inlining the following `Dockerfile` (don't be tempted to uncomment the `#RUN ...` lines yet)

```
FROM nginx:mainline-alpine

#RUN chmod g+rwX /var/cache/nginx /var/run /var/log/nginx
#RUN chgrp -R root /var/cache/nginx

COPY cfg/default.conf /etc/nginx/conf.d/default.conf
COPY cfg/nginx.conf /etc/nginx/nginx.conf
COPY cfg/index.html /usr/share/nginx/html/index.html
#RUN chmod -R g+rX /etc/nginx/

RUN chmod -R g+rX /usr/share/nginx/html
EXPOSE 8080
```

- Use the ConfigMap listed in file `nginx-config.yaml` to provide the files `cfg/default.conf`, `cfg/nginx.conf` and `cfg/index.html` to the build context.
- **Bonus**: Create the nginx-config.yaml file by yourself from files under `cfg/` directory.
  - ***Tip #1***: `oc create configmap <name> --from-file=<file1> --from-file=<file2> ... --dry-run -o yaml`
- Create an ImageStream object named `my-openshift-nginx` and set the output of your BuildConfig to ImageStreamTag `my-openshift-nginx:latest`.
- ***Tip #2***: Create skeleton code of the objects with `oc new-build my-bc --dry-run -o yaml --allow-missing-images -D - > bcs.yaml` and copy-paste the Dockerfile above and press Control-D.

Actual website from Github

Fork `github.com/cscfi/rahti-httpd-ex` in GitHub. Add the following Dockerfile to your fork:

```
FROM my-openshift-nginx:latest
COPY ./ /usr/share/nginx/html/
RUN rm /usr/share/nginx/html/Dockerfile
RUN chmod -R g+rX /usr/share/nginx/html
```

Create a scaffolding for a new application with the following command

```
$ oc --name new-app my-static-home https://github.com/<youraccount>/rahti-httpd-ex \
    --dry-run -o yaml > scaffolding.yaml
```

The file `scaffolding.yaml` will now contain multiple API objects in a *List* object.

The container `my-static-home` should be in a crash-loop right now. The lines starting with `#RUN ...` in the first Dockerfile should not have be commented out in the first place after all the correct Dockerfile looks like be:

```
FROM nginx:mainline-alpine

RUN chmod g+rwX /var/cache/nginx /var/run /var/log/nginx
RUN chgrp -R root /var/cache/nginx

COPY cfg/default.conf /etc/nginx/conf.d/default.conf
COPY cfg/nginx.conf /etc/nginx/nginx.conf
COPY cfg/index.html /usr/share/nginx/html/index.html
RUN chmod -R g+rX /etc/nginx/

RUN chmod -R g+rX /usr/share/nginx/html
EXPOSE 8080
```

Modify the BuildConfig housing Dockerfile defined in the first part in and rebuild the image in ImageStreamTag `my-openshift-nginx:latest`. ***Tip #3***: You can modify the Dockerfile directly from the web console: Locate the BuildConfig and click Actions → Edit.

Once build has been completed, the final application should work and you can make a Route to it.