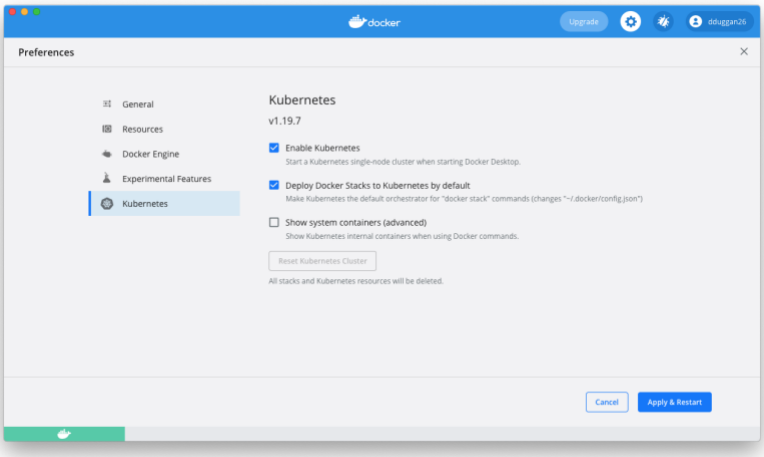
CS 548—Fall 2022
Enterprise Software Architecture and Design
Assignment Nine—Kubernetes

In this assignment, you will deploy the microservices that you developed in previous assignments into a Kubernetes cluster.

Step 1: Install Kubernetes

I will assume that you have already installed Docker Desktop, if you are running this on a Windows or MacOS machine[1]. See here for more information on how to use Kubernetes with Docker Desktop. You will need to enable Kubernetes in Docker Desktop in Preferences:



The `kubectl` Kubernetes client is installed with Docker Desktop, at `/usr/local/bin/kubectl`. You may need to use the following command to set your Kubernetes context to that for Docker Desktop (if for example you have installed minikube):

```
$ kubectl config use-context docker-desktop
```

Kubernetes for Docker Desktop does not come with the dashboard enabled, you will have to do this:

```
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml
```

The following command will run a proxy server or application-level gateway between localhost and the Kubernetes API server, so you can access the dashboard from your Web browser:

[1] If you have a Linux laptop, you may want to use minikube instead. See here for more information about installing minikube.

```
$ kubectl proxy
```

Then access the dashboard at this URL:

```
http://localhost:8001/api/v1/namespaces/kubernetes-
dashboard/services/https:kubernetes-dashboard:/proxy/
```

You will need to authenticate as an administrative user to access the dashboard, see here for how to do this. Create a file called `dashboard-adminuser.yaml` with this content:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
```

Then create the administrative user for the dashboard as follows:

```
$ kubectl apply -f dashboard-adminuser.yaml
```

Similarly, you need to create a role binding giving this user privilege to access the dashboard (obviously something you want to be very careful about!). Create a file `dashboard-authorization.yaml` with this content:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

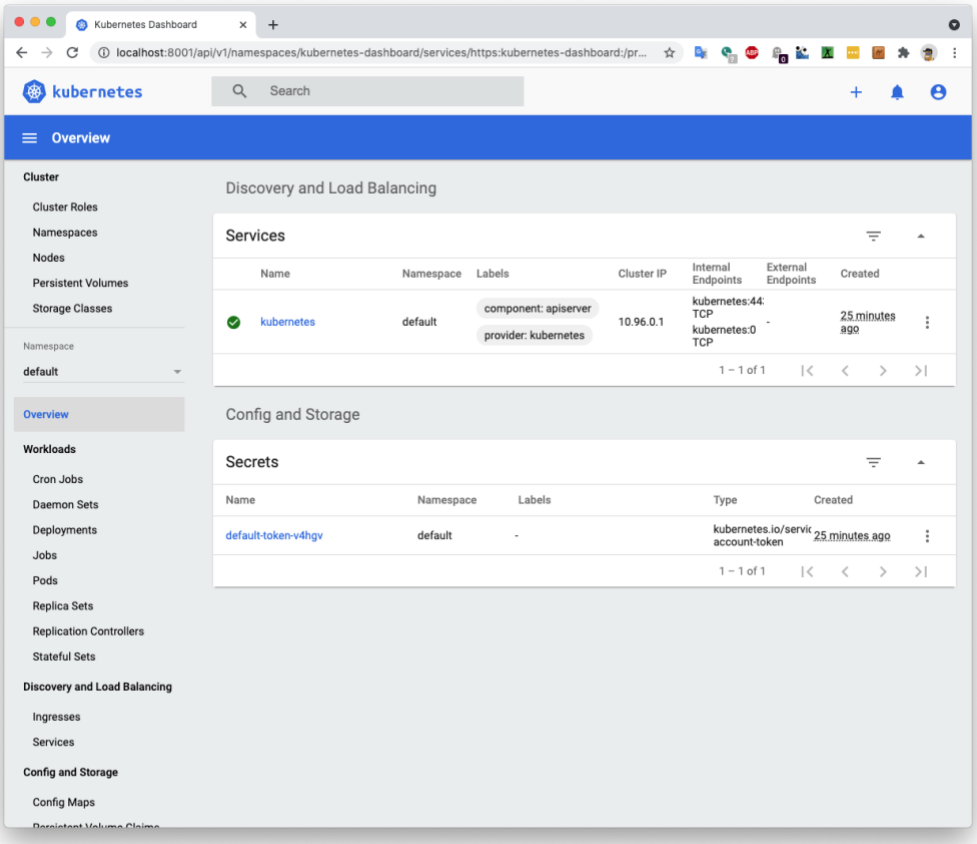
Then grant the administrative user permission to manage the dashboard cluster as follows:

```
$ kubectl apply -f dashboard-authorization.yaml
```

The Web page for the dashboard requires a bearer token, basically a login token, that authenticates you as the administrative user that you have just authorized to access the dashboard:

```
$ kubectl -n kubernetes-dashboard get secret $(kubectl -n kubernetes-dashboard
get sa/admin-user -o jsonpath="{.secrets[0].name}") -o go-
template="{{.data.token | base64decode}}"
```

Copy and paste the bearer token into the dashboard "login screen," granting you access to the dashboard:



Step 2: Deploy the Database Server

We will deploy the database server in Kubernetes. In the previous assignment, you created a Docker image that initialized the database when the server was run. To now deploy this in Kubernetes, create a YAML configuration file `cs548-database-deploy.yaml` for Kubernetes[2]:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cs548db
  labels:
    app: cs548db
spec:
  replicas: 1
  selector:
```

[2] Passing passwords through environment variables set in configuration scripts should be considered an anti-pattern. A better approach is to use docker secrets: If the superuser password is available in a file in /run/secrets in the container, then specify the location of this file in the environment variable `POSTGRES_PASSWORD_FILE`. See here for more information. Also, in a production deployment, the database should be mounted on an external volume, as you did in EC2, otherwise it will be deleted with the container when the deployment ends.

```
    matchLabels:
      app: cs548db
  template:
    metadata:
      labels:
        app: cs548db
    spec:
      restartPolicy: Always
      containers:
      - name: cs548db
        image: cs548/database
        env:
          - name: POSTGRES_PASSWORD
            value: XXXXXX
          - name: DATABASE_PASSWORD
            value: YYYYYY
        imagePullPolicy: Never
```

The `matchLabels` field specifies how this deployment matches the pods that it may manage. The `imagePullPolicy` ensures that Kubernetes does not attempt to pull the docker image from the global docker repository, but only pulls it from the local repository. Start this database server up as a pod in Kubernetes:

```
$ kubectl apply -f cs548-database-deploy.yaml
$ kubectl get pods
$ kubectl describe pod pod-name
```

Now create a YAML configuration file `cs548-database-service.yaml` to provide access to this deployment. It would make sense to just make this available within the cluster by specifying a type of `ClusterIP`, but you may want the option of being able to connect to it from the "node" (i.e., your laptop) with `psql` or the Eclipse Dali plugin, so we expose it outside the cluster using `NodePort`:

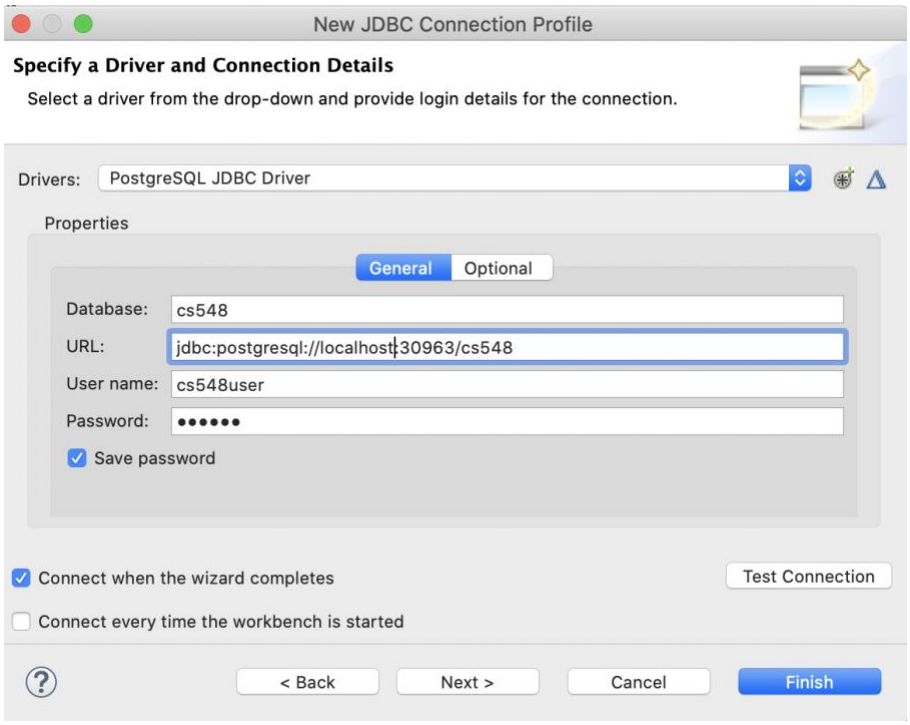
```
apiVersion: v1
kind: Service
metadata:
  name: cs548db
  labels:
    app: cs548db
spec:
  type: NodePort
  ports:
  - name: jdbc
    port: 5432
    targetPort: 5432
  selector:
    app: cs548db
```

The node port is the port on the node on which this service is exposed outside the cluster. Since it is not specified here, it will be generated automatically. The port is internal to the cluster and is that on which other pods in the cluster can access this service (with "virtual host name" `cs548db`), and accesses on the node port are forwarded to this port. Accesses on

this port are forwarded to the target port 5432 in the pod (if not specified, this target port defaults to be the same as port), this is the port that the server in the pod should be listening on. Start this service up in Kubernetes[3]:

```
$ kubectl apply -f cs548-database-service.yaml
$ kubectl get service cs548db
$ kubectl describe service cs548db
```

From the node port, you can get the port on localhost that you can use to connect to the database server. Communications to this port are forwarded to port 5432 on the container in the pod. You can test this out by setting up a connection from Eclipse, with host name localhost and port number given by the node port:



Step 3: Deploy the Microservice

It is best practice to separate the pods where the frontend applications and the domain microservice run. If they both run in the same JVM, then restarting one requires restarting the other. Create a deployment configuration `clinic-domain-deployment.yaml` as before:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: clinic-domain
```

[3] You can also use the YAML file to undeploy an application, e.g.,

```
$ kubectl delete -f cs548-database-service.yaml
$ kubectl delete -f cs548-database-deploy.yaml
```

```
  labels:
    app: clinic-domain
spec:
  replicas: 1
  selector:
    matchLabels:
      app: clinic-domain
  template:
    metadata:
      labels:
        app: clinic-domain
    spec:
      restartPolicy: Always
      containers:
      - name: clinic-domain
        image: cs548/clinic-domain
        env:
        - name: DATABASE_PASSWORD
          value: YYYYYY
        imagePullPolicy: Never
```

And create a service configuration `clinic-domain-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: clinic-domain
  labels:
    app: clinic-domain
spec:
  type: NodePort
  ports:
  - name: http
    port: 8080
  - name: https
    port: 8181
  selector:
    app: clinic-domain
```

Once the service is running, you can test some of the CRUD operations at this URL, using the node port (which you will have to look up with `kubectl describe service`):

```
http://localhost:31435/api
```

Step 3: Deploy the applications

Each application will be a client of the microservice. For the frontend Web application, define the deployment descriptor:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: clinic-webapp
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: clinic-webapp
  template:
    metadata:
      labels:
        app: clinic-webapp
    spec:
      restartPolicy: Always
      containers:
      - name: clinic-webapp
        image: cs548/clinic-webapp
        imagePullPolicy: Never
```

And also, the service descriptor:

```
apiVersion: v1
kind: Service
metadata:
  name: clinic-webapp
  labels:
    app: clinic-webapp
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 8080
  - name: https
    Port: 8181
  selector:
    app: clinic-webapp
```

This will expose the application to clients as a cloud application, with client requests routed through a load balancer that distributes the load among a replica set of pods. The endpoint for the service will now be exposed by the public port number 8080 outside the cluster, and you can view the state of the domain via a Web browser at this URL:

```
http://localhost:8080/clinic-webapp/clinic
```

Use a similar strategy to deploy the frontend Web service, `clinic-rest`. Define YAML files describing its deployment and service and use these to launch it in your Kubernetes cluster. You can use a Web browser to test some of the query operations for this Web service, and use the REST client from a previous assignment to upload data to the Web service. Use `kubectl logs` to show via logs that both the Web application and the Web service are using the backend microservice to access the domian.

Once your apps are deployed, you can get a list of the running pods with:

```
$ kubectl get pods
```

You can get the details of a particular pod with:

```
$ kubectl describe pod pod-name
```

And you can see the logs at a pod with:

```
$ kubectl logs pod-name
```

Finally, you can use the Kubernetes dashboard to review your setup in your Kubernetes cluster.

For your submission, you should provide your dockerfiles (and their inputs, including WAR files) and YAML configuration files. You should also provide a video that demonstrates:

1. Creating docker images for the services that you deploy.
2. Launching your services, and using `kubectl` (e.g. `kubectl describe service`) to show the endpoint information for these services.
3. Demonstrate your assignment working, as in previous assignments: use the REST client CLI to invoke your frontend Web service and show the results in the Web app, and show with the logs that they are invoking the domain microservice in the background.
4. Show the database tables via Eclipse JPA perspective, connecting to the database server node port, and querying the database using the `clinic-rest` and `clinic-domain` Web services.

Make sure that your name appears at the beginning of the video. For example, display the contents of a file that provides your name. *Do not provide private information such as your email or cwid in the video*. Be careful of any "free" apps that you download to do the recording, there are known cases of such apps containing Trojan horses, including key loggers. **Your video must be MP4 format!**

Your submission should be uploaded via the Canvas classroom, as a zip file. Your solution should consist of a zip archive with one folder, identified by your name. This folder should contain the files and subfolders with your submission.

It is important that you provide your dockerfiles (and their inputs) and YAML files. You should also provide a video demonstrating setting up and running your services in Kubernetes. It is sufficient to demonstrate Kubernetes running on your own machine. You should also provide a completed rubric.