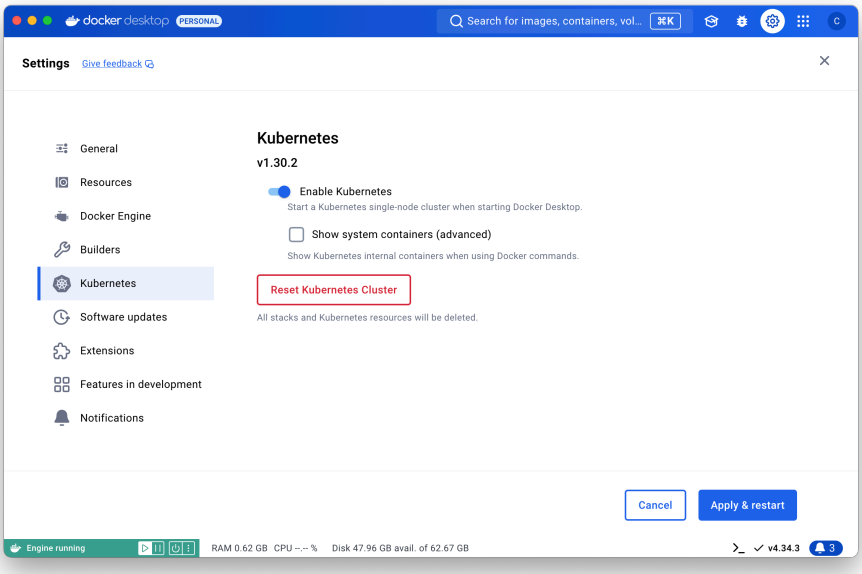
CS 548—Spring 2025
Enterprise Software Architecture and Design
Assignment Nine—Kubernetes

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. You will need to enable Kubernetes in Docker Desktop in Settings[1]:



The `kubectl` Kubernetes client is installed with Docker Desktop (at `/usr/local/bin/kubectl` on MacOS). 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 get-contexts
$ kubectl config use-context docker-desktop
```

Kubernetes for Docker Desktop does not come with the dashboard[2] enabled, you will have to install it for yourself if you wish, and it requires the installation of helm. But this is not required for the assignment.

---

[1] https://docs.docker.com/desktop/kubernetes/.
[2] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-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 `clinic-database-deploy.yaml` for Kubernetes[3]:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cs548db
  labels:
    app: cs548db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cs548db
  template:
    metadata:
      labels:
        app: cs548db
    spec:
      restartPolicy: Always
      containers:
      - name: cs548db
        image: cs548/clinic-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 clinic-database-deploy.yaml
$ kubectl get pods
$ kubectl describe pod pod-name
```

Now create a YAML configuration file `clinic-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

[3] 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: https://kubernetes.io/docs/tasks/inject-data-application/. 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.

from the "node" (i.e., your laptop) with `psql` or the Intellij Database tool, 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 (your laptop) 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 service port. Accesses on the port of the service are forwarded to the *target port* 5432 in the pod (if not specified, this target port defaults to be the same as port). This target port is the port that the server in the pod should be listening on. Start this service up in Kubernetes[4]:

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

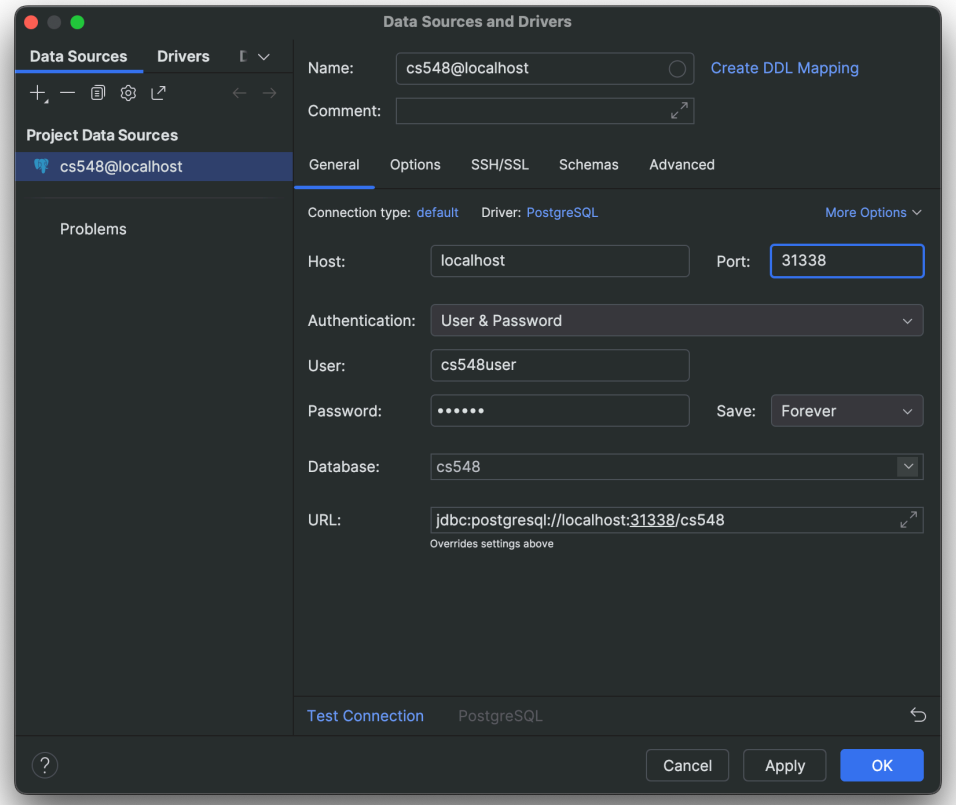
Here is an example output from the last step:

```
Name:                     cs548db
Namespace:                default
Labels:                   app=cs548db
Annotations:              <none>
Selector:                 app=cs548db
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.107.176.144
IPs:                      10.107.176.144
Port:                     jdbc  5432/TCP
TargetPort:               5432/TCP
NodePort:                 jdbc  31338/TCP
Endpoints:                10.1.0.8:5432
```

[4] 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
```

The node port gives you the port on the local machine (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 Intellij IDEA, 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
  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_USERNAME
        value: cs548user
      - name: DATABASE_PASSWORD
        value: YYYYYY
      - name: DATABASE
        value: cs548
      - name: DATABASE_HOST
        value: cs548db
      - name: MEMORY_THRESHOLD
        value: "10485760"
      imagePullPolicy: Never
```

The `matchLabels` field specifies how this deployment matches the pods that it may manage. The `imagePullPolicy` ensures that Kubernetes should **not** pull the docker image to the local repository if it is not already present. Since we are running Kubernetes locally, and we will have pushed the docker image to the local repository already, this is not necessary here, but this will change if we deploy in a Kubernetes cluster in the cloud. The environment variable bindings for the container should match those from the previous assignment. Start this microservice up as a pod in EKS:

```
$ kubectl apply -f clinic-domain-deployment.yaml
$ kubectl get pods
$ kubectl describe pod pod-name
$ kubectl logs pod-name
```

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
  selector:
    app: clinic-domain
```

We don't specify a HTTPS port because Payara Micro Community Edition does not support it. Start this service up in Kubernetes[5]:

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

```
$ kubectl apply -f clinic-domain-service.yaml
$ kubectl get service clinic-domain
$ kubectl describe service 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 again have to look up with `kubectl describe service`):

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

Step 4: 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
  selector:
    app: clinic-webapp
```

```
$ kubectl delete -f clinic-domain-service.yaml
$ kubectl delete -f clinic-domain-deployment.yaml
```

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
```

Use a similar strategy to deploy the frontend Web service, `clinic-rest`. Define a YAML file describing its deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: clinic-rest
spec:
  replicas: 1
  selector:
    matchLabels:
      app: clinic-rest
  template:
    metadata:
      labels:
        app: clinic-rest
    spec:
      restartPolicy: Always
      containers:
      - name: clinic-rest
        image: cs548/clinic-rest
        imagePullPolicy: Never
```

Next define a YAML file describing its service, where we specify a different port from the target port because the webapp has already bound to port 8080 on your host machine:

```
apiVersion: v1
kind: Service
metadata:
  name: clinic-rest
  labels:
    app: clinic-rest
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 9090
    targetPort: 8080
  selector:
    app: clinic-rest
```

The web service can be accessed at this URL:

```
http://localhost:9090/api
```

You need to specify different ports for the webapp and web service, because both must be accessible outside the cluster on your laptop. This was not necessary for the microservice

because it was never exposed outside the cluster (except via the randomly assigned nodeport). Within the cluster, it is accessed on its own pod via a DNS lookup that resolves to the pod IP address. The combination of pod IP address and port are unique within the cluster.

You can use a web browser to test some of the query operations for the microservice and 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 webapp and the web service are using the backend microservice to access the domain.

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:
    a. Use a web browser to query the microservice (e.g., for lists of patients and providers).
    b. Use a web browser to query the REST web service (e.g., for a patient).
    c. Use the REST client CLI to invoke your frontend web service.
    d. Use a web browser to show the results in the webapp.
    e. Show with the logs that the webapp and REST web service are invoking the domain microservice in the background.
4. Show the database tables via Intellij Database tool, 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.