ITI8801 · Cloud Architectures and DevOps

Practice 6 · Assignment 6: your service on the course Kubernetes cluster

Siim Rebane · TalTech · Autumn 2026

What you build

Your Assignment 5 service, running in your own namespace on the course cluster, reachable from the internet at your own host name:

```
http://<your student code in lower case>.k8s.digivader.com/
```

Part	Written by you
the service	your A5 image, unchanged, from its public registry
its database	PostgreSQL, with its data on a volume that outlives the pod
the Kubernetes objects	manifests in YAML, applied with `kubectl apply`

No VM, no Terraform, no Compose this week: the cluster is provided, and everything you create lives in your namespace.

Your namespace

	You have
namespace	your student code in lower case; the examples use the test namespace `000007test`
credential	a kubeconfig file, only for you: `iti8801.digivader.com/kubeconfig/`, with your student code and the check code you submitted for Lab 2 in Moodle
what it allows	create, change and delete the usual application objects in your namespace; nothing outside it
quota	requests: 0.5 CPU, 2 GiB memory; limits: 6 CPU, 6 GiB; 20 pods; 4 volume claims, 4 GiB of storage in total; no Service of type `NodePort` or `LoadBalancer`
defaults	a container without resources gets requests 50m / 64Mi and limits 500m / 256Mi
network	connections to your pods only from your own namespace and from the ingress controller

Check it works:

```
kubectl auth can-i create pods            # yes
kubectl describe quota
```

What must exist in your namespace

Object	Requirements
a **Deployment** of your service	your A5 image: the same repository as in your A5 check, any tag; **2 replicas**; a readiness probe and a liveness probe; CPU and memory requests, a memory limit
a **Service** for it	type `ClusterIP`
an **Ingress**	host `<your code in lower case>.k8s.digivader.com`, path `/` to your Service
a **StatefulSet** for PostgreSQL	one replica; its data in a volume from `volumeClaimTemplates`; a readiness probe
a headless **Service** for the database	its name is the StatefulSet's `serviceName`; your service's database address uses that name, e.g. `@db:5432`
a **Secret** with the database password	created with `kubectl create secret`, never from a file in git; both the database and your service read it with `secretKeyRef`

No Service of type `NodePort` or `LoadBalancer`: the Ingress is the only way in, and your quota allows none.

The end state, checked

- `GET /health` and the whole A5 API answer through your host name, as in Assignment 5
- two pods of your service are ready
- the API keeps answering while one of your service's pods is deleted
- an item created through the API is still there after the database pod is deleted and comes back
- the password appears in no manifest: the Secret exists in the cluster only

The validator

`https://iti8801.digivader.com/lab6-validator/` : your student code, nothing else.

Box	Green when
API	every step of the A5 API test answers through your Ingress
Service	your Ingress leads to a `ClusterIP` Service and from it to a Deployment with 2 ready pods, readiness and liveness probes, requests and a memory limit, the password from a Secret, your A5 image
Database	a PostgreSQL StatefulSet, ready, its volume claim Bound, a readiness probe, the password from a Secret; your Deployment's database address names its Service
Survives	checked once the first three are green: while the validator deletes one of your service pods, at most 3 requests fail and the last ones succeed; an item created before it deletes your database pod is there afterwards
Manifests	your upload is accepted

The validator reads your namespace directly, as you do with `kubectl get`, and checks the API from outside.

What the upload must contain

- every manifest you applied: `.yaml` or `.yml` files

Refused: a manifest of `kind: Secret`, a kubeconfig, any file with a token or a private key in it, files that are not YAML. A refused upload lists every reason and stores nothing.

Accepted when the files together contain a Deployment, a Service, an Ingress and a StatefulSet.

What to submit, and when

- The validator's code of a check that is **PASSED**. Paste it into Moodle.
- Deadline: Sunday next week, 23:59, as for every assignment.
- Your objects may stay: the cluster is the course's, not yours to pay for. The quota is shared by everything you run in your namespace.

Walkthrough

The steps in order: what to write, the commands, and what each should print. The YAML is given; replace what is yours (`<…>`). The outputs come from the test namespace `000007test`, whose service is the A5 reference image, port 8000, database user and database `app`.

Step 0 · kubectl

System	Command
macOS	`brew install kubectl`
Windows	`winget install -e --id Kubernetes.kubectl`
Linux	the package from `pkgs.k8s.io`, or the binary from `dl.k8s.io`: kubernetes.io/docs/tasks/tools

```
$ kubectl version --client
Client Version: v1.37.1
```

The cluster runs Kubernetes 1.36; a `kubectl` at most one minor version older or newer works with it (1.35 to 1.37).

Step 0 · your kubeconfig

Download `iti8801-<code>.yaml` from `iti8801.digivader.com/kubeconfig/` (your student code plus the check code you submitted for Lab 2 in Moodle), save it as `~/.kube/iti8801.yaml`, then either:

```
export KUBECONFIG=~/.kube/iti8801.yaml          # macOS, Linux: for this terminal
$env:KUBECONFIG="$HOME\.kube\iti8801.yaml"      # Windows PowerShell
```

or copy it to `~/.kube/config` if you have no other cluster. Then:

```
$ kubectl config get-contexts
CURRENT   NAME      CLUSTER   AUTHINFO     NAMESPACE
*         iti8801   iti8801   000007test   000007test
$ kubectl auth can-i create pods
yes
$ kubectl describe quota
Resource                Used  Hard
limits.cpu              0     4
requests.memory         0     2Gi
…
```

The context sets your namespace: `kubectl get pods` works without `-n`. The file is your password to the cluster: not in git, not in a screenshot, not in a chat.

Step 1 · the Secret

The password for the database, made once, kept only in the cluster:

```
$ kubectl create secret generic db --from-literal=password="$(openssl rand -hex 16)"
secret/db created
$ kubectl get secret db
NAME   TYPE     DATA   AGE
db     Opaque   1      0s
```

Part	Means
`create secret generic db`	a Secret named `db`, of the general type
`--from-literal=password=…`	one key, `password`, with the value given
`$(openssl rand -hex 16)`	a random 32-character password, generated by your shell; you never need to see it
`DATA 1`	one key stored

Both the database and your service read the same key, with `secretKeyRef` in their pod templates. Changing the Secret later does not change the password inside an existing database volume.

Step 2 · the database: write `db.yaml`, part 1

Two objects in one file, separated by `---`: the headless Service, then the StatefulSet. Replace `<user>` and `<database>` with the names your service's database address uses:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  clusterIP: None
  selector:
    app: db
  ports:
  - port: 5432
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: db
spec:
  serviceName: db
  replicas: 1
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
      - name: postgres
        image: postgres:18-alpine
```

Step 2 · the database: write `db.yaml`, part 2

The rest of the container (same indentation, straight after `image:`), then the claim template at the StatefulSet's `spec` level:

```
      env:
      - name: POSTGRES_USER
        value: <user>
      - name: POSTGRES_DB
        value: <database>
      - name: POSTGRES_PASSWORD
        valueFrom:
          secretKeyRef: {name: db, key: password}
      ports:
      - containerPort: 5432
      readinessProbe:
        exec:
          command: ["pg_isready", "-U", "<user>", "-d", "<database>"]
        periodSeconds: 5
      resources:
        requests: {cpu: 100m, memory: 128Mi}
        limits: {cpu: 500m, memory: 512Mi}
      volumeMounts:
      - name: data
        mountPath: /var/lib/postgresql
volumeClaimTemplates:
- metadata:
    name: data
  spec:
    accessModes: ["ReadWriteOnce"]
    resources:
      requests:
        storage: 1Gi
```

Step 2 · the database: apply and check

```
$ kubectl apply -f db.yaml
service/db created
statefulset.apps/db created
$ kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
db-0   0/1     Pending   0          0s
$ kubectl get pods                                  # 12 s later
NAME   READY   STATUS    RESTARTS   AGE
db-0   1/1     Running   0          12s
$ kubectl get pvc
NAME        STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
data-db-0   Bound    pvc-1ab3d9bc-fdbb-4167-a431-a003e7b28c66   1Gi        RWO            local-path
$ kubectl exec db-0 -- pg_isready -U <user>
/var/run/postgresql:5432 - accepting connections
```

In the output	Means
`Pending` for a few seconds	the volume is being created and a node chosen; longer than a minute: `kubectl describe pod db-0`
`data-db-0 Bound`	the claim from `volumeClaimTemplates`, named `<template>-<pod>`, has its volume
`pg_isready`	Postgres accepts connections; `-U` is your database user

Step 3 · your service: write `deployment.yaml`

Your A5 image, its port, the database address with `db` as the host, the password from the Secret through `$(DB_PASSWORD)`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 2
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: service
        image: <your A5 image, e.g. docker.io/you/a5:1>
        env:
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef: {name: db, key: password}
        - name: DATABASE_URL
          value: postgresql://<user>:$(DB_PASSWORD)@db:5432/<database>
        ports:
        - containerPort: <port>
        readinessProbe:
          httpGet: {path: /health, port: <port>}
          periodSeconds: 5
        livenessProbe:
          httpGet: {path: /health, port: <port>}
          periodSeconds: 10
        resources:
          requests: {cpu: 50m, memory: 96Mi}
          limits: {cpu: 500m, memory: 256Mi}
```

Step 3 · your service: apply and check

```
$ kubectl apply -f deployment.yaml
deployment.apps/api created
$ kubectl get deployment api                          # 10 s later
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
api    2/2     2            2           10s
$ kubectl logs deployment/api | tail -3
Found 2 pods, using pod/api-544c998ffd-vnwgg
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     10.42.5.1:53584 - "GET /health HTTP/1.1" 200 OK
$ kubectl describe pod api-544c998ffd-nnrtk | tail -2
  Normal   Started    9s               kubelet   spec.containers{service}: Container started
  Warning  Unhealthy  8s (x2 over 9s)  kubelet   spec.containers{service}: Readiness probe failed: Get "http://10.42.3.16:8000/health": … connection refused
```

In the output	Means
`READY 2/2`	both pods pass their readiness probe
`logs deployment/<name>`	the output of one of its pods; `kubectl logs <pod>` for a particular one
`Unhealthy` in the first seconds	the probe ran before the program listened: normal. If it keeps coming, the probe's path or port is wrong
`GET /health … 200` in the log	the kubelet's probes, every few seconds

Step 4 · the Service: write `service.yaml`, apply, test

```yaml
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  selector:
    app: api
  ports:
  - port: 80
    targetPort: <port>
```

```
$ kubectl apply -f service.yaml
service/api created
$ kubectl get service api
NAME   TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
api    ClusterIP   10.43.185.1   <none>        80/TCP    0s
$ kubectl port-forward service/api 8080:80
Forwarding from 127.0.0.1:8080 -> 8000
$ curl -s localhost:8080/health                    # in a second terminal
{"status":"ok"}
```

Part	Means
`port: 80`, `targetPort: <port>`	the Service answers on 80 and sends to your container's port
`port-forward service/api 8080:80`	your machine's port 8080 to the Service, through the API server; runs until Ctrl-C

When this works and the Ingress does not, the problem is in the Ingress: its host name, or the Service name and port it points to.

Step 5 · the Ingress: write `ingress.yaml`, apply, test

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api
spec:
  rules:
  - host: <your code in lower case>.k8s.digivader.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: api
            port:
              number: 80
```

```
$ kubectl apply -f ingress.yaml
ingress.networking.k8s.io/api created
$ kubectl get ingress
NAME   CLASS     HOSTS                           ADDRESS                              PORTS   AGE
api    traefik   000007test.k8s.digivader.com   193.40.157.103,193.40.157.158,…   80      3s
$ curl -s http://000007test.k8s.digivader.com/health
{"status":"ok"}
$ curl -s -o /dev/null -w "%{http_code}\n" http://193.40.157.249/
404
```

`CLASS traefik`: the cluster's ingress controller carries your rule; `ADDRESS`: the nodes it listens on. `404` for a request without your host name: the host name in the request is what selects your Service.

Step 6 · it survives

What the validator does, by hand. An item created, a service pod deleted, the database pod deleted:

```
$ curl -s -X POST http://000007test.k8s.digivader.com/api/boardgames -H 'Content-Type: application/json' -d '{"name":"Catan","players":4}'
{"id":1,"name":"Catan","players":4}
$ kubectl delete pod api-544c998ffd-nnrtk
pod "api-544c998ffd-nnrtk" deleted from 000007test namespace
$ kubectl get pods -l app=api
NAME                   READY   STATUS    RESTARTS   AGE
api-544c998ffd-vnwgg   1/1     Running   0          18s
api-544c998ffd-z2gz9   0/1     Running   0          2s
$ curl -s http://000007test.k8s.digivader.com/api/boardgames/1
{"id":1,"name":"Catan","players":4}
$ kubectl delete pod db-0
pod "db-0" deleted from 000007test namespace
$ kubectl get pods                                  # 8 s later
NAME                   READY   STATUS    RESTARTS   AGE
api-544c998ffd-vnwgg   1/1     Running   0          28s
api-544c998ffd-z2gz9   0/1     Running   0          12s
db-0                   1/1     Running   0          9s
$ curl -s http://000007test.k8s.digivader.com/api/boardgames/1
{"id":1,"name":"Catan","players":4}
```

In the output	Means
a new pod `0/1 Running` next to the old one `1/1`	the ReplicaSet replaced the deleted pod; the other one answered meanwhile
`db-0` back in 9 s, the item still there	the new `db-0` mounted the same claim
a service pod `0/1` after the database restart	its readiness probe fails while the database is away. If `/health` checks the database, both pods can be out for a few seconds and the Ingress answers `no available server`; then they come back

Step 7 · the validator

- `https://iti8801.digivader.com/lab6-validator/` : your student code, nothing else. Four boxes, then the upload.
- Upload the four files: `db.yaml` , `deployment.yaml` , `service.yaml` , `ingress.yaml` . Never the Secret, never the kubeconfig.
- Paste the code of the **PASSED** check into Moodle.

```
$ kubectl get all
NAME                          READY   STATUS    RESTARTS   AGE
pod/api-544c998ffd-vnwgg      1/1     Running   0          48s
pod/api-544c998ffd-z2gz9      1/1     Running   0          32s
pod/db-0                      1/1     Running   0          29s

NAME          TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
service/api   ClusterIP   10.43.185.1   <none>        80/TCP     37s
service/db    ClusterIP   None          <none>        5432/TCP   61s

NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/api   2/2     2            2           48s

NAME                  READY   AGE
statefulset.apps/db   1/1     61s
```

Commands you will use

Command	Does
`kubectl apply -f file.yaml`, `-f .`	create or update the objects in a file, or in every file of a directory
`kubectl get pods`, `… -o wide`, `… -l app=api`	list; with node and IP; only those with a label
`kubectl get all`, `kubectl get pvc`, `kubectl get ingress`	the application objects; the volume claims; the Ingresses
`kubectl describe <type> <name>`	one object with its state and events: the first place to look when something is not `Running`
`kubectl logs <pod>`, `-f`, `--previous`	output; followed; of the run before the last restart
`kubectl exec <pod> -- <command>`, `-it … -- sh`	a command in the container; an interactive shell, if the image has one
`kubectl delete -f file.yaml`, `kubectl delete pod <name>`	delete what a file describes; one object
`kubectl rollout status / undo deployment/<name>`	follow a rollout; go back one
`kubectl port-forward service/<name> 8080:<port>`	the Service on your machine's port 8080
`kubectl describe quota`, `kubectl top pods`	what your namespace reserves; what the pods use now

When something does not work

You see	Look at
`Error from server (Forbidden): … exceeded quota`	`kubectl describe quota`: your requests and limits, and what already runs
`Pending`	`kubectl describe pod`: the scheduler's reason at the bottom
`ImagePullBackOff`	the image name and tag; the repository public; the image has a `linux/amd64` variant
restarts climbing	`kubectl logs <pod> --previous`: the output of the run that ended
`CreateContainerConfigError`	`kubectl describe pod`: a Secret or key the pod reads does not exist
`0/1` ready, `Running`	the readiness probe's path and port; `kubectl describe pod` shows the probe's answers
`404` from the Ingress	the host name in the Ingress against the one in your request
`502`, `503`, `no available server` from the Ingress	the Service's selector against the pods' labels; its port against the container's port; no ready pods

The cluster's names for your objects

From	Your service is
another pod in your namespace	`http://api:80`
a pod in another namespace	not reachable: the network policy
the internet	`http://<your code in lower case>.k8s.digivader.com/`, through the Ingress

Your service's database address, with the headless Service named `db`:

```
postgresql://<user>:<password>@db:5432/<database>
```

`db` resolves to the database pod's own address. The password part comes from the Secret: an environment variable from `secretKeyRef`, then `$(DB_PASSWORD)` inside the next variable.



Questions