

Journal:

**Day 1: Foundational Flask Development and Local Database Integration** The primary goal of Day 1 was to build the core local functionality of the Flask web application, including user-facing forms and backend API endpoints capable of interacting with a PostgreSQL database running locally in Docker.

- **Task 1: Enhance Flask Frontend with HTML/CSS**
- **Actions:**
  1. **Created templates/ and static/ directories:** Standard Flask convention for organizing HTML templates and static assets (CSS, JavaScript, images) respectively. This promotes a clean project structure.
  2. **Developed templates/book.html:** This HTML file provides the user interface for booking an appointment. It includes a form with fields for patient name, doctor selection, and appointment time.
    - *Reason:* To allow users to visually interact with the system and submit appointment requests.
  3. **Created static/style.css:** Basic CSS rules were added to improve the visual appearance and usability of the book.html form.
    - *Reason:* To make the user interface more presentable and user-friendly.
  4. **Updated app.py to render book.html:** The /book route in app.py was modified to use Flask's `render_template('book.html')` function.
    - *Reason:* To connect the URL endpoint (/book) to the actual HTML page, so users visiting that URL see the booking form.
  5. **Initial Testing and Troubleshooting:**
    - Addressed a `ModuleNotFoundError: No module named 'flask'`. This occurred because the Flask library was not installed in the Python environment where app.py was being run.
    - *Reason & Solution:* Python couldn't locate the Flask package. Resolved by installing Flask using `pip install Flask`.
    - Added print statements to app.py to display accessible URLs in the terminal upon startup.

- *Reason:* User request for clearer feedback during development, making it easier to know which URLs to access for testing.

- **Task 2: Develop Backend API Endpoints & Local Database Setup**

- **Actions:**

1. **Installed psycopg2-binary:** This package is the PostgreSQL database adapter for Python.

- *Reason:* To enable the Flask application to communicate with (send queries to and receive results from) the PostgreSQL database.

2. **Configured Database Connection in app.py:**

- Connection parameters (DB\_NAME, DB\_USER, DB\_PASS, DB\_HOST, DB\_PORT) were defined. The password was initially read from db\_password.txt to avoid hardcoding it directly in the script.

- *Reason:* To provide the necessary credentials and location details for psycopg2 to connect to the database.

- A get\_db\_connection() helper function was created to encapsulate the logic for establishing a database connection.

- *Reason:* To promote reusable code and centralize connection management.

3. **Developed API Endpoints in app.py:**

- **POST /book-appointment:** This endpoint handles the submission of the booking form. It receives form data, connects to the PostgreSQL database, and executes an SQL INSERT statement to store the new appointment.

- *Reason:* To provide the core functionality of creating and saving new appointments in the system.

- **GET /appointments:** This endpoint retrieves all existing appointments from the database using an SQL SELECT statement and returns them, typically as JSON.

- *Reason:* To allow for viewing of booked appointments, useful for administration or patient history features.

4. **PostgreSQL Database Setup and Troubleshooting:**

- A postgres Docker container (named postgres-db) was used to host the database. A db\_setup.sql file was created to define the healthcare database and the appointments table structure.
- **Major Challenge: FATAL: database "healthcare" does not exist:** This was a persistent issue. The Flask application (running directly on the host at this stage) could not connect to the healthcare database within the postgres-db Docker container, even though the container itself was running.
- *Root Cause Investigation:* Extensive troubleshooting involved checking container logs, app.py connection strings, and the database initialization process. The key issue was related to **Docker volume persistence and initialization**. The PostgreSQL Docker image has a mechanism to run initialization scripts (like creating a database specified by POSTGRES\_DB or running scripts in /docker-entrypoint-initdb.d/) *only when a data volume is initialized for the first time*. If the named volume (pgdata\_healthcarebooking) existed from previous (potentially failed) attempts and wasn't correctly initialized with the healthcare database, subsequent container starts wouldn't recreate it if the image logic deemed the volume already "initialized."
- *Resolution Steps:*
  1. The postgres-db container was stopped and removed.
  2. Crucially, the Docker named volume (pgdata\_healthcarebooking) was explicitly removed (docker volume rm pgdata\_healthcarebooking) to ensure a completely fresh start.
  3. The postgres-db container was restarted with the -e POSTGRES\_DB=healthcare environment variable (which instructs the official image to create this database if it doesn't exist on fresh volume initialization) and the volume mount.
  4. After confirming the healthcare database was created, the db\_setup.sql script (which creates the appointments table) was manually executed within the container using psql.
  5. The DB\_HOST in app.py was confirmed to be 127.0.0.1 (or localhost) because, at this point, the Flask app was running directly on the host, and the postgres-db container had its port 5432 mapped to the host.
- Successful booking and retrieval of appointments via Postman confirmed the fix.

**Day 2: Containerization and Kubernetes Deployment for Flask** The focus of Day 2 was to containerize the now-functional Flask application, deploy it to a local Kubernetes cluster (via Docker Desktop), and implement autoscaling.

- **Task 1: Create Dockerfile and Docker Image for Flask App**

- **Actions:**

1. **Created deployment/Dockerfile:** This file contains instructions to build a Docker image for the Flask application. It specifies the base Python image, sets up the working directory, copies application code and dependencies, installs Python packages from requirements.txt, exposes port 5000, and defines the command to run the Flask app.
    - *Reason:* To package the Flask application and all its dependencies into a standardized, portable, and reproducible unit (a Docker image).
  2. **Created deployment/requirements.txt:** This file lists the Python dependencies (Flask, psycopg2-binary).
    - *Reason:* To explicitly declare the application's Python package dependencies, ensuring they are installed consistently when the Docker image is built.
  3. **Built the Docker Image:** The command `docker build -t healthcarebooking_app:v1 -f deployment/Dockerfile .` was used.
    - *Reason:* To create the runnable Docker image named `healthcarebooking_app` with tag `v1` from the Dockerfile specifications.
  4. **Ran the Containerized Flask App and Tested DB Connection:** The Flask app container was run.
    - **Networking Challenge: DB\_HOST for Container-to-Container:** When the Flask app ran *inside its own container*, `DB_HOST = "127.0.0.1"` in `app.py` failed because `127.0.0.1` now referred to the Flask container itself.
    - *Reason & Solution:* Containers on the same Docker network can resolve each other by their container names. `DB_HOST` in `app.py` was changed to `"postgres-db"` (the name of the running PostgreSQL container). The Flask app Docker image was rebuilt with this change.
    - Successful booking confirmed the Flask container could connect to the `postgres-db` container.
- **Task 2: Create Kubernetes YAML Files and Deploy Flask App**

- **Actions:**

1. **Pushed Docker Image to Docker Hub:** docker push hayden2310/healthcarebooking\_app:v1.
  - *Reason:* To make the application image accessible to the Kubernetes cluster. Kubernetes pulls images from a registry (like Docker Hub) to run them.
2. **Created deployment/flask-deployment.yaml:** This Kubernetes manifest defines a Deployment object.
  - *Reason:* To declare the desired state of the Flask application in Kubernetes, including which Docker image to use, the number of replicas (pods), port configurations, labels for organization, environment variables, resource requests/limits for scheduling, and readiness/liveness probes for health checking.
3. **Created deployment/flask-service.yaml:** This manifest defines a Service object of type ClusterIP.
  - *Reason:* To provide a stable internal network endpoint (a consistent IP address and DNS name within the cluster) to access the Flask application pods. The Service load balances traffic across the pods managed by the Deployment.
4. **Applied YAMLs to Kubernetes:** kubectl apply -f ....
  - *Reason:* To instruct Kubernetes to create or update resources as defined in the YAML files.
5. **Verified and Accessed:** Used kubectl get pods,svc,deployment to check status and `kubectl port-forward service/flask-app-service 5000:5000` to access the application running in Kubernetes from the local machine.

- **Task 3: Configure Horizontal Pod Autoscaling (HPA)**

- **Actions:**

1. **Created deployment/flask-hpa.yaml:** This manifest defines a HorizontalPodAutoscaler object.
  - *Reason:* To enable Kubernetes to automatically increase or decrease the number of Flask application pods based on observed CPU utilization (or other metrics), allowing the application to handle variable loads efficiently.
2. **Applied HPA YAML.**

### 3. Troubleshooting HPA:

- **Issue 1: HPA Metrics <unknown>/50%:** The HPA couldn't fetch CPU utilization metrics.
- *Reason & Solution:* The pods targeted by the HPA (defined in flask-deployment.yaml) must have CPU resource requests specified in their container definition. Without this, Kubernetes cannot calculate CPU utilization as a percentage of the request. Added resources: {requests: {cpu: "..."}} to the deployment.
- **Issue 2: Pods Pending due to Insufficient cpu during HPA Scale-Up:** When HPA tried to create new pods, they couldn't be scheduled.
- *Reason & Solution:* The sum of CPU requests from all pods (existing + new ones HPA wanted to create) exceeded the allocatable CPU of the single node in Docker Desktop. Resolved by:
  1. Reducing the requests.cpu per pod in flask-deployment.yaml (e.g., to 50m).
  2. Adjusting the replicas in flask-deployment.yaml to 1 (initial state before HPA scales).
  3. Setting minReplicas: 1 and maxReplicas: 3 (a realistic maximum for the local node) in flask-hpa.yaml.
- 4. **Load Testing and Observation:** Generated HTTP traffic to the Flask service and used kubectl get hpa -w and kubectl get pods -w to observe the HPA scaling the deployment up to 3 pods under load, and then scaling back down to 1 pod when the load was removed. This confirmed the HPA was working correctly.

**Day 3, Task 1: Deployment of PostgreSQL as a StatefulSet in Kubernetes****Objective:** To establish a persistent and reliably networked PostgreSQL database instance within the Kubernetes cluster, capable of serving the Flask application backend. This involved transitioning from a Docker-based PostgreSQL setup to a Kubernetes-native StatefulSet deployment.**Procedures and Rationale:**

1. **Persistent Storage Configuration (postgres-pvc.yaml - Note: This initial separate PVC was later superseded by volumeClaimTemplates but was part of the initial thought process based on the plan):**
  - **Action:** A PersistentVolumeClaim (PVC) manifest (postgres-pvc.yaml) was initially drafted as per the project plan to request persistent storage for PostgreSQL.

- **Rationale:** Stateful applications like databases require their data to persist across pod restarts, rescheduling, or node failures. PVCs provide a mechanism for pods to request and bind to available PersistentVolume (PV) resources, ensuring data durability.

## 2. StatefulSet Definition (postgres-statefulset.yaml):

- **Action:** A StatefulSet manifest (postgres-statefulset.yaml) was created to manage the PostgreSQL deployment. This included:
  - Specifying the postgres:16 Docker image.
  - Defining environment variables for database initialization (POSTGRES\_DB=healthcare, POSTGRES\_USER=postgres, POSTGRES\_PASSWORD=qelol669).
  - Crucially, using volumeClaimTemplates to dynamically provision a unique PersistentVolumeClaim for each PostgreSQL pod (e.g., postgres-data-postgres-0 for the postgres-0 pod). This template requested 1Gi of storage with ReadWriteOnce access mode.
  - Mounting the claim at /var/lib/postgresql/data within the container.
- **Rationale:** StatefulSets are the Kubernetes workload API object designed for stateful applications. They provide stable, unique network identifiers (e.g., pod-name.service-name), stable persistent storage linked to each pod ordinal, and ordered, graceful deployment and scaling. This is essential for database systems.

## 3. Headless Service Definition (postgres-service.yaml):

- **Action:** A headless Service manifest (postgres-service.yaml) named postgres was created (with clusterIP: None).
- **Rationale:** For StatefulSets, a headless service is used to control the network domain of the pods and provide stable DNS entries for each pod (e.g., postgres-0.postgres.default.svc.cluster.local). This allows other applications within the cluster (like the Flask app) to connect to specific database instances reliably, rather than a load-balanced IP that could point to any replica if a regular ClusterIP service were used with multiple database replicas (though for a single replica, a regular service would also work, headless is standard for StatefulSets).

## 4. Deployment and Initial Troubleshooting:

- **Action:** The YAML manifests for the service and StatefulSet were applied to the Kubernetes cluster using `kubectl apply`.
- **Issue Encountered:** The `postgres-0` pod entered a `CrashLoopBackOff` state. Log analysis (`kubectl logs postgres-0`) revealed an `initdb` error: directory `/var/lib/postgresql/data` exists but is not empty, due to the presence of a lost+found directory in the root of the mounted persistent volume.
- **Resolution:** The `postgres-statefulset.yaml` was modified to include the `PGDATA` environment variable (`PGDATA=/var/lib/postgresql/data/pgdata`). This directed the PostgreSQL `initdb` process to use a subdirectory within the mounted volume, thus avoiding conflict with the pre-existing lost+found directory. The pod was then deleted to allow the StatefulSet controller to recreate it with the corrected configuration, leading to successful startup.

## 5. Flask Application Reconfiguration and Testing:

- **Action:** The Flask application's configuration (`app.py`) was updated: `DB_HOST` was changed from `"postgres-db"` (used for Docker container-to-container networking) to `"postgres"` (the name of the Kubernetes headless service for PostgreSQL).
- The Flask application's Docker image (`hayden2310/healthcarebooking_app:v2`) was rebuilt and pushed to Docker Hub with this change.
- The `flask-app-deployment.yaml` was updated to use the new image version, and the deployment was rolled out.
- **Issue Encountered:** Upon testing, the Flask application connected to the PostgreSQL service but returned an error: relation `"appointments"` does not exist.
- **Resolution:** The PostgreSQL database, though initialized with the healthcare database by the StatefulSet's environment variables, did not automatically contain the appointments table. This table was created manually by connecting to the `postgres-0` pod via `kubectl exec -it postgres-0 -- psql -U postgres -d healthcare` and executing the `CREATE TABLE appointments (...)` SQL statement.
- **Final Verification:** Subsequent API calls (POST to `/book-appointment`, GET to `/appointments`) via Postman (accessed through `kubectl port-forward`) confirmed successful data persistence and retrieval, validating the end-to-end connectivity between the Flask application and the PostgreSQL StatefulSet within Kubernetes.

**Day 3, Task 2: Set Up Logging and Monitoring with Kubernetes Dashboard**  
**Objective:** To implement real-time monitoring and basic logging for the Healthcare Appointment Booking



System deployed on Kubernetes, thereby enabling visibility into application health, resource utilization, and autoscaling behavior. **Procedures and Rationale:**

### 1. Deployment of Kubernetes Dashboard:

- **Action:** The official Kubernetes Dashboard was deployed to the cluster using the recommended manifest:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
```

- **Rationale:** The Kubernetes Dashboard provides a web-based user interface for managing and monitoring cluster resources, including Deployments, StatefulSets, Pods, Services, and Horizontal Pod Autoscalers (HPA). It is a standard tool for visualizing the state and health of workloads in real time.

### 2. Access Configuration and Authentication:

- **Action:** A secure access method was established by running `kubectl proxy`, which exposes the Dashboard locally at `http://localhost:8001/...`. To enable administrative access, a ServiceAccount (`admin-user`) was created in the `kubernetes-dashboard` namespace, and a ClusterRoleBinding was applied to grant it cluster-admin privileges. A login token was then generated using:

```
kubectl -n kubernetes-dashboard create token admin-user
```

- **Rationale:** Kubernetes Dashboard requires authentication for security. Creating a dedicated admin ServiceAccount with appropriate permissions ensures secure and controlled access, while the token-based login is a best practice for local development and demonstration environments.

### 3. Monitoring Workloads and Resources:

- **Action:** Upon logging into the Dashboard, the “Workloads” section was used to monitor the status of all major components:
- **Deployments:** The flask-app-deployment was observed for pod health, replica count, and image version.
- **StatefulSets:** The postgres StatefulSet was monitored to ensure the database pod was running and healthy.

- **Pods:** Real-time CPU and memory usage metrics for both the Flask and PostgreSQL pods were reviewed, confirming that resource requests and limits were being enforced and that the application was operating within expected parameters.
- **Horizontal Pod Autoscaler:** The HPA resource (flask-app-hpa) was inspected to verify that it was tracking CPU utilization and adjusting the number of Flask app replicas in response to load.
- **Rationale:** Visual monitoring of these resources provides immediate feedback on application health, resource consumption, and autoscaling behavior. It also aids in troubleshooting and performance optimization.

#### 4. Evidence Collection:

- **Action:** Screenshots were taken of the Dashboard's "Workloads" page, showing Deployments, StatefulSets, and Pods with their respective resource usage bars. Additional screenshots were captured from the HPA section and detailed pod metrics views.
- **Rationale:** These screenshots serve as concrete evidence of successful monitoring setup and provide visual proof of the system's operational state, resource utilization, and autoscaling in action.

**Outcome:** The Kubernetes Dashboard was successfully deployed and configured, providing a comprehensive, real-time view of the Healthcare Appointment Booking System's components and their health. This setup enables ongoing monitoring, supports troubleshooting, and demonstrates advanced Kubernetes operational practices.

#### Day 4 Summary: Testing, Optimization, and GKE Deployment

**PreparationDate:** 29/05/2025 (Simulated) **Objective:** To conduct comprehensive testing of the deployed application, optimize its performance and resource utilization, and prepare the environment for deployment to Google Kubernetes Engine (GKE). **Activities**

#### Undertaken and Outcomes:

##### 1. End-to-End System Testing:

- Successfully accessed the healthcare appointment booking system's frontend via the Kubernetes service (flask-app-service) after enabling port forwarding (kubectl port-forward service/flask-app-service 5000:5000).
- A test appointment booking was submitted through the /book interface.

- The submitted appointment data was successfully persisted and verified in the PostgreSQL database (running as pod/postgres-0) using kubectl exec.
- The GET /appointments API endpoint was tested using Postman, successfully retrieving all booked appointments.
- Load was simulated on the Flask application, and the Horizontal Pod Autoscaler (HPA - flask-app-hpa) was observed. The HPA demonstrated scaling behavior, increasing pod replicas in response to CPU load, although it was noted that the third pod sometimes remained in a pending state due to CPU resource constraints in the local Minikube environment.
- Evidence of these tests was captured for the e2e\_test.png deliverable.

## 2. **Application and Database Optimization:**

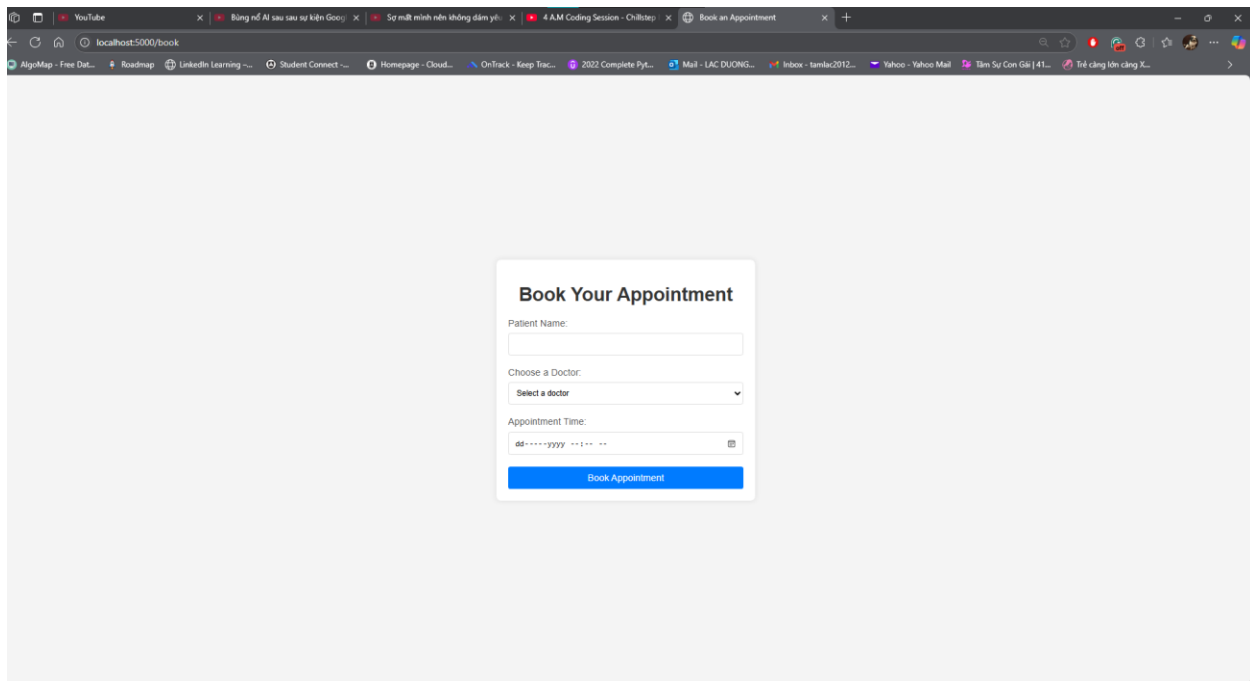
- **Database Indexing:** An SQL index (idx\_appointment\_time) was created on the appointment\_time column of the appointments table to improve query performance. The optimization.sql script containing the CREATE INDEX statement was created and successfully applied to the PostgreSQL database.
- **Query Performance Verification:** The EXPLAIN ANALYZE command was executed on a sample query. While the database utilized a sequential scan due to the small dataset size, the index creation was successful, and a baseline for query performance was established.
- **Resource Management:** Resource requests and limits for CPU and memory were defined in the deployment/flask-deployment.yaml file for the Flask application pods.
- Initial values from the plan were: CPU request "200m", memory request "256Mi", CPU limit "500m", memory limit "512Mi".
- Following observations about potential CPU scheduling issues in the local environment, the CPU request was revised to "50m" to ensure pod stability, while other planned values were maintained.
- The updated flask-deployment.yaml was applied to the Kubernetes cluster.
- Resource usage (CPU and Memory) was monitored using the Kubernetes Dashboard, observing the impact of the new resource settings.
- Evidence of query analysis and resource monitoring was compiled for the optimization.png deliverable.

### 3. Preparation for Google Kubernetes Engine (GKE) Deployment:

- A new Kubernetes cluster (healthcarebooking-cluster-standard) was successfully provisioned on Google Kubernetes Engine (GKE) in the australia-southeast2 region.
- kubectl was configured to connect to the newly created GKE cluster using the gcloud container clusters get-credentials command.
- Connectivity to the GKE cluster was verified by successfully listing the cluster nodes using kubectl get nodes. The output confirmed the presence of GKE nodes (e.g., gke-healthcarebooking-cl-default-pool-30f65d0f-f626).
- The GKE cluster status and node information were captured for the gke\_cluster.png deliverable.

## Appendices

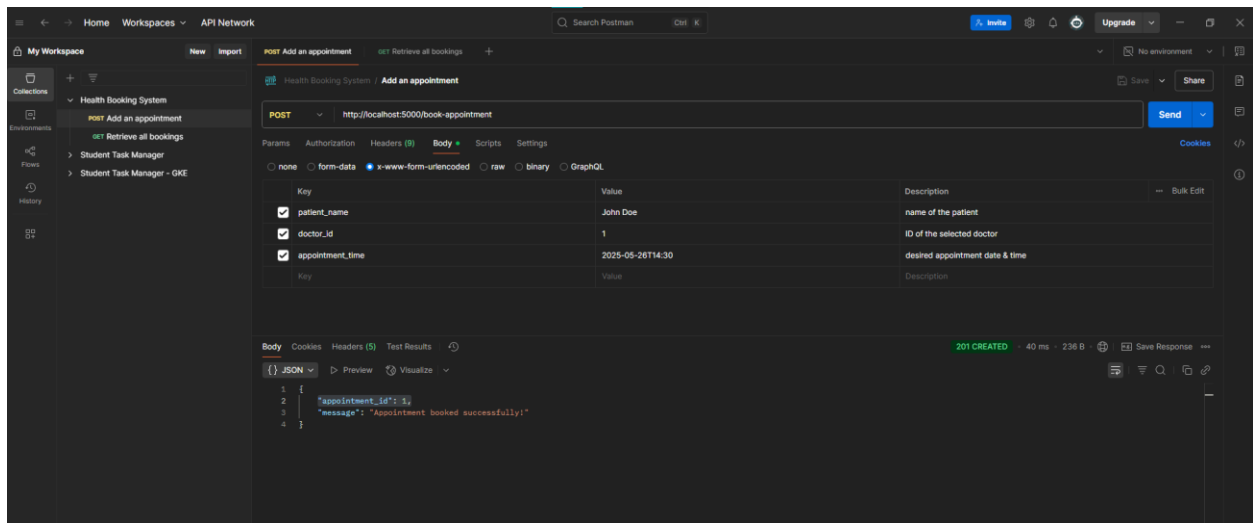
“flash\_booking\_form.png”



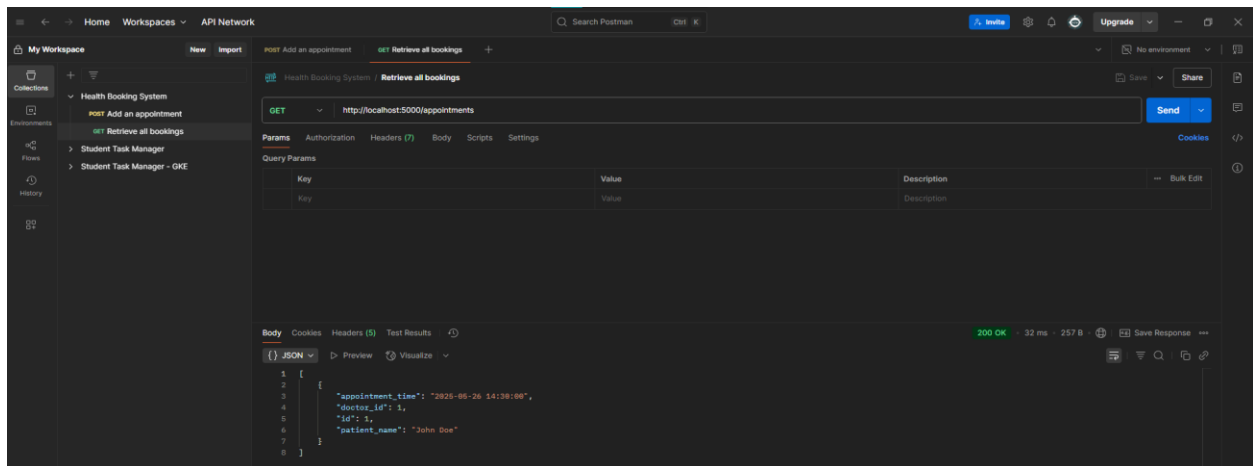
The screenshot shows a web browser window with multiple tabs. The active tab is titled 'Book an Appointment' and shows a form titled 'Book Your Appointment'. The form has the following fields:

- Patient Name:** A text input field.
- Choose a Doctor:** A dropdown menu with the text 'Select a doctor'.
- Appointment Time:** A date and time picker field showing 'dd-mm-yyyy --:--'.
- Book Appointment:** A blue button at the bottom of the form.

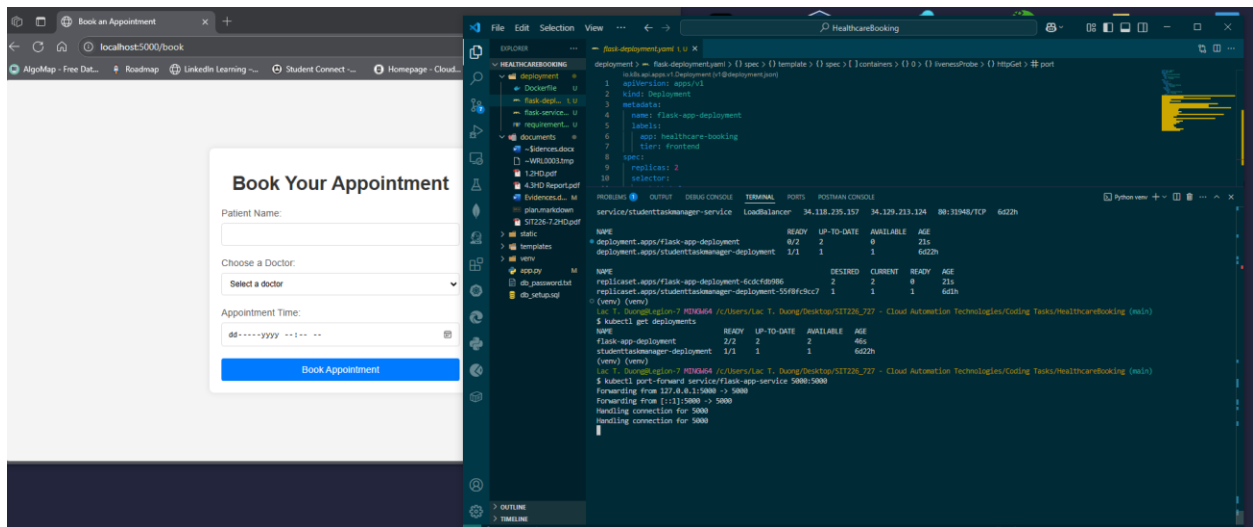
“api\_post.png”



“api\_get.png”



“flask\_k8s.png”



“hpa\_scaling.png”

```
Lac T. Duong@Legion-7 MINGW64 /c/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/Health
o $ kubectl get hpa flask-app-hpa -w
NAME          REFERENCE          TARGETS          MINPODS  MAXPODS  REPLICAS  AGE
flask-app-hpa Deployment/flask-app-deployment  cpu: 2%/50%      1         3         1         35s
flask-app-hpa Deployment/flask-app-deployment  cpu: 48%/50%     1         3         1         75s
flask-app-hpa Deployment/flask-app-deployment  cpu: 76%/50%     1         3         1        105s
flask-app-hpa Deployment/flask-app-deployment  cpu: 76%/50%     1         3         2         2m
flask-app-hpa Deployment/flask-app-deployment  cpu: 39%/50%     1         3         2        2m45s
flask-app-hpa Deployment/flask-app-deployment  cpu: 40%/50%     1         3         2        3m46s
flask-app-hpa Deployment/flask-app-deployment  cpu: 47%/50%     1         3         2        4m46s
flask-app-hpa Deployment/flask-app-deployment  cpu: 68%/50%     1         3         2        5m16s
flask-app-hpa Deployment/flask-app-deployment  cpu: 68%/50%     1         3         3        5m30s
flask-app-hpa Deployment/flask-app-deployment  cpu: 69%/50%     1         3         3        6m16s
flask-app-hpa Deployment/flask-app-deployment  cpu: 39%/50%     1         3         3        7m16s
flask-app-hpa Deployment/flask-app-deployment  cpu: 9%/50%      1         3         3        7m46s
flask-app-hpa Deployment/flask-app-deployment  cpu: 2%/50%      1         3         3        8m46s
flask-app-hpa Deployment/flask-app-deployment  cpu: 2%/50%      1         3         3        12m
flask-app-hpa Deployment/flask-app-deployment  cpu: 2%/50%      1         3         2        12m
flask-app-hpa Deployment/flask-app-deployment  cpu: 2%/50%      1         3         2        12m
flask-app-hpa Deployment/flask-app-deployment  cpu: 2%/50%      1         3         1        13m
flask-app-hpa Deployment/flask-app-deployment  cpu: 2%/50%      1         3         1        13m
[]

Lac T. Duong@Legion-7 MINGW64 /c/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/Health
o $ kubectl get pods -w
NAME                                READY   STATUS    RESTARTS   AGE
flask-app-deployment-6bc66c7bb6-qhdm8  1/1     Running   0           55m
flask-app-deployment-6bc66c7bb6-rgxqh  1/1     Running   0           8m27s
flask-app-deployment-6bc66c7bb6-wrnrx   0/1     Pending   0           4m57s
studenttaskmanager-deployment-55f8fc9cc7-7cphg  1/1     Running   0           40h
flask-app-deployment-6bc66c7bb6-wrnrx   0/1     Terminating 0           7m
flask-app-deployment-6bc66c7bb6-wrnrx   0/1     Terminating 0           7m
[]
```

```
Lac T. Duong@Legion-7 MINGW64 /c/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl describe pod flask-app-deployment-6bc66c7bb6-wmrnx
Name: flask-app-deployment-6bc66c7bb6-wmrnx
Namespace: default
Priority: 0
Service Account: default
Node: <none>
Labels: app=healthcare-booking
pod-template-hash=6bc66c7bb6
tier=frontend
Annotations: cloud.google.com/cluster_autoscaler_unhelpable_since: 2025-05-26T12:06:42+0000
cloud.google.com/cluster_autoscaler_unhelpable_until: Inf
Status: Pending
IP:
IPs: <none>
Controlled By: ReplicaSet/flask-app-deployment-6bc66c7bb6
Containers:
  flask-app-container:
    Image: hayden2310/healthcarebooking_app:v1
    Port: 5000/TCP
    Host Port: 0/TCP
    Limits:
      cpu: 500m
      memory: 256Mi
    Requests:
      cpu: 50m
      memory: 128Mi
    Liveness: http-get http://:5000/ delay=15s timeout=1s period=20s #success=1 #failure=3
    Readiness: http-get http://:5000/ delay=5s timeout=1s period=10s #success=1 #failure=3
    Environment:
      FLASK_APP: app.py
      FLASK_RUN_HOST: 0.0.0.0
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-sqrrn (ro)
Conditions:
  Type Status
  PodScheduled False
Volumes:
  kube-api-access-sqrrn:
    Type: Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds: 3607
    ConfigMapName: kube-root-ca.crt
    ConfigMapOptional: <nil>
    DownwardAPI: true
QoS Class: Burstable
Node-Selectors: <none>
Tolerations: node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type Reason Age From Message
  ----
Warning FailedScheduling 73s default-scheduler 0/1 nodes are available: 1 Insufficient cpu, preemption: 0/1 nodes are available: 1 No preemption victims found for incoming pod.
Normal NotTriggerScaleUp 74s cluster-autoscaler pod didn't trigger scale-up:
```

## “postgres\_statefulset.png”

```
Lac T. Duong@Legion-7 MINGW64 /c/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl get all
NAME                                     READY   STATUS    RESTARTS   AGE
pod/flask-app-deployment-566cb5c67-9h665 1/1     Running   0           15m
pod/postgres-0                            1/1     Running   0           43m
pod/studenttaskmanager-deployment-55f8fc9cc7-7cphg 1/1     Running   0           2d6h

NAME                                     TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
service/flask-app-service                ClusterIP      34.118.225.88    <none>            5000/TCP          15h
service/kubernetes                       ClusterIP      34.118.224.1     <none>            443/TCP           7d14h
service/postgres                          ClusterIP      None             <none>            5432/TCP          60m
service/studenttaskmanager-service        LoadBalancer  34.118.235.157   34.129.213.124   80:31948/TCP     7d13h

NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/flask-app-deployment      1/1     1             1           15h
deployment.apps/studenttaskmanager-deployment 1/1     1             1           7d13h

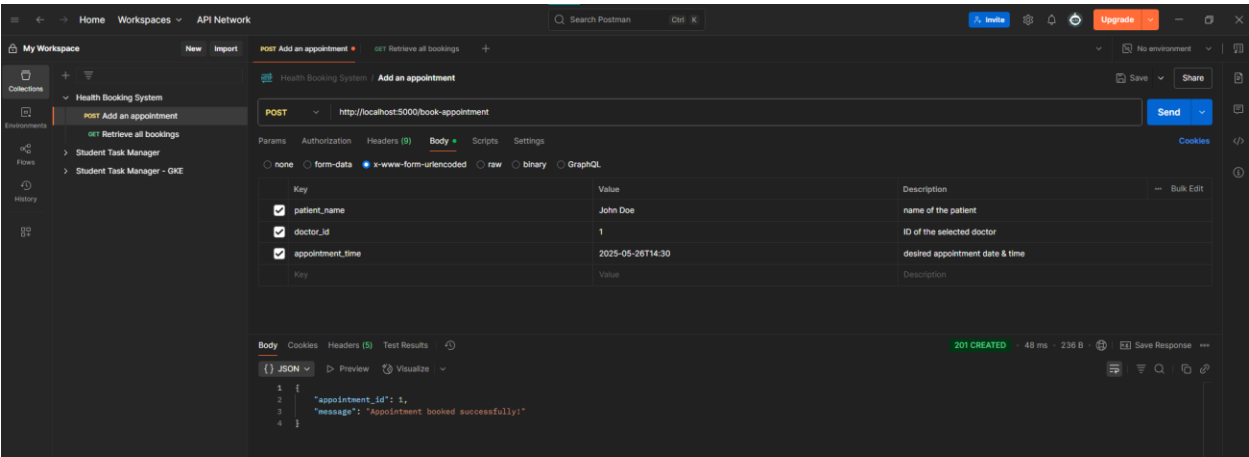
NAME                                     DESIRED   CURRENT   READY   AGE
replicaset.apps/flask-app-deployment-566cb5c67 1         1         1       15m
replicaset.apps/studenttaskmanager-deployment-55f8fc9cc7 1         1         1       6d16h

NAME                                     READY   AGE
statefulset.apps/postgres                1/1     59m

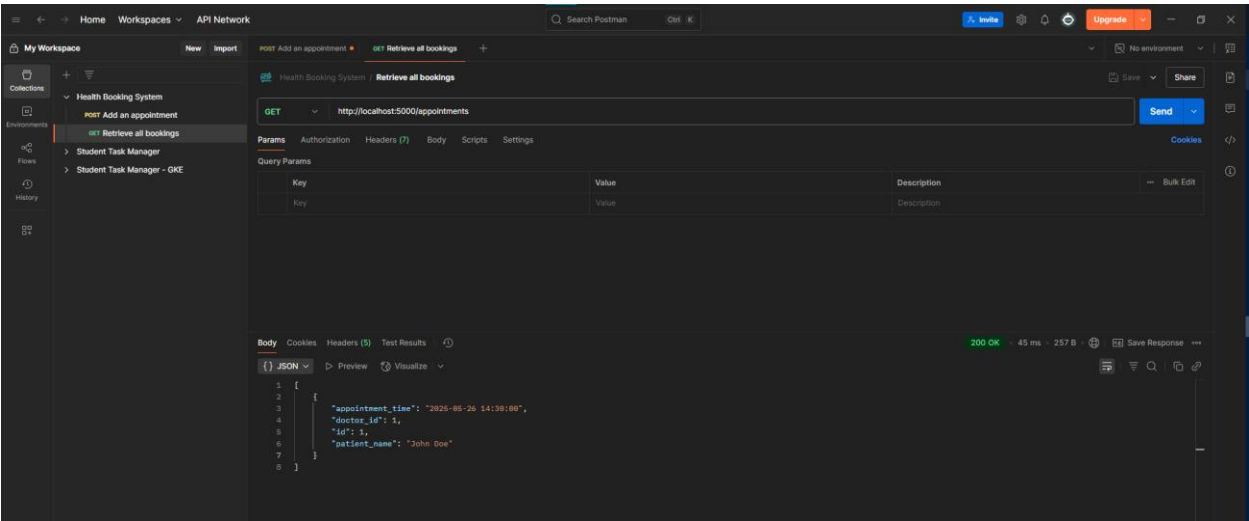
NAME                                     REFERENCE                                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/flask-app-hpa  Deployment/flask-app-deployment  cpu: 2%/50%  1         3         1           13h
(venv)
Lac T. Duong@Legion-7 MINGW64 /c/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl get pvc postgres-data-postgres-0
NAME                                     STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
postgres-data-postgres-0                Bound    pvc-dbl328b-b921-4496-aacc-56f7e1f245ef  1Gi        RWO             standard-rwo    <unset>                  63m
(venv)
```

Successful API Interaction (Proving Flask App Connection to k8s Postgres)

flask\_k8s\_db\_connection\_post.png

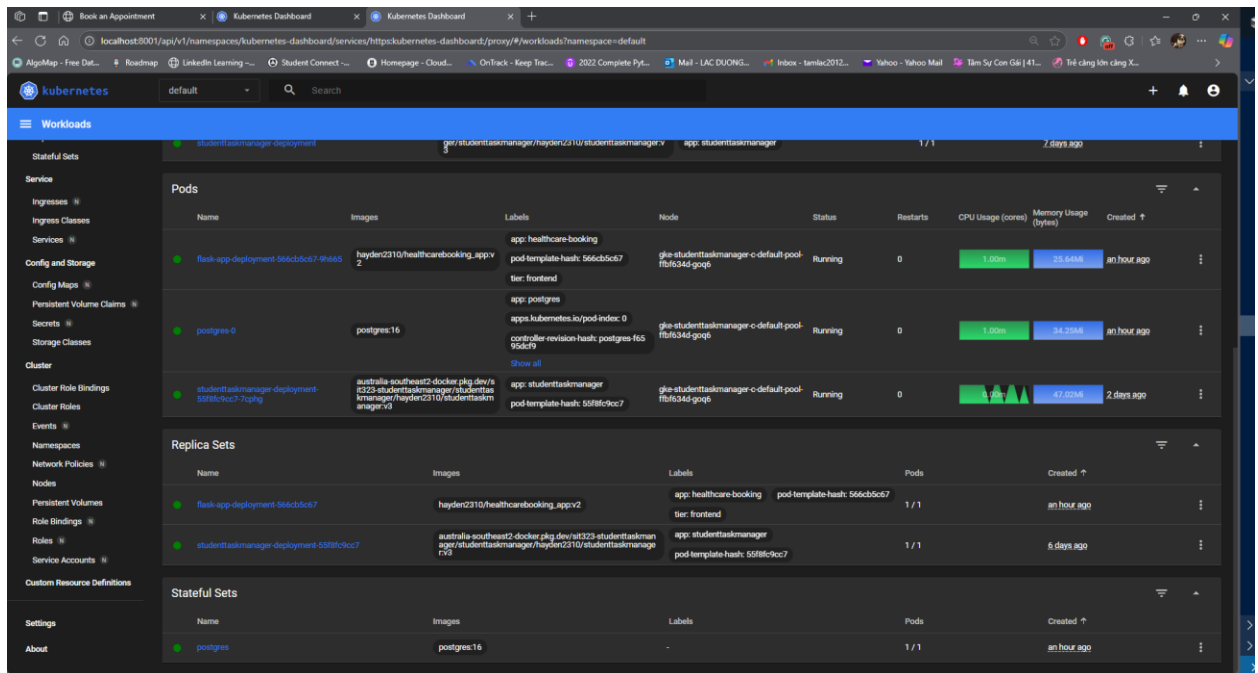
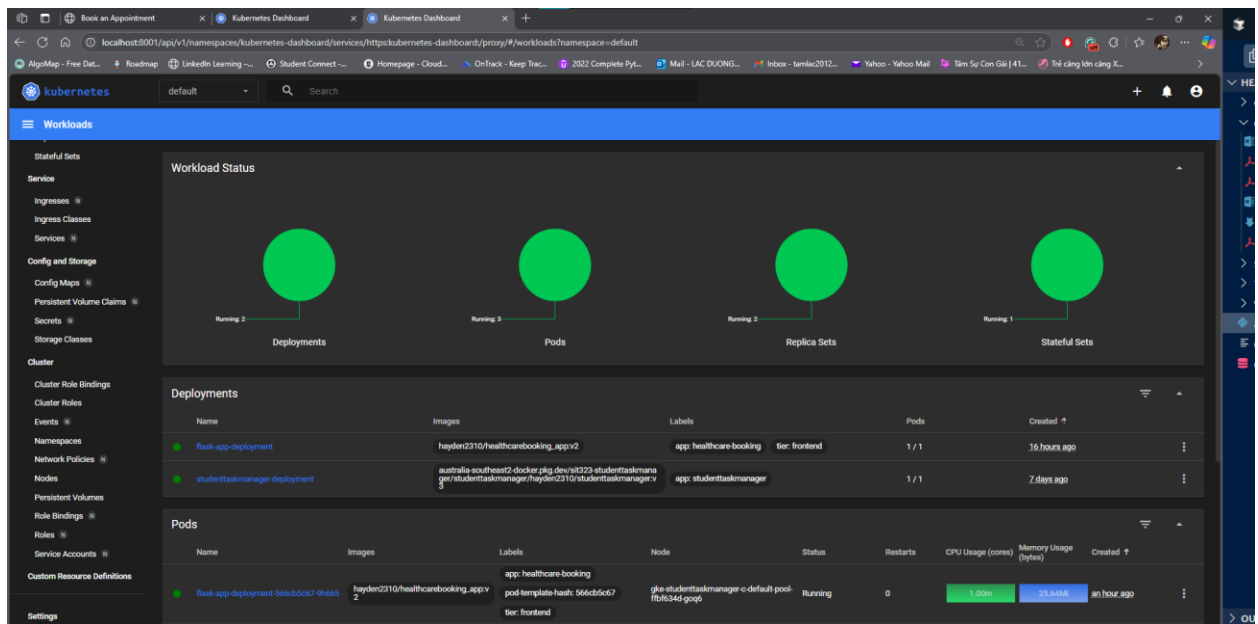


flask\_k8s\_db\_connection\_get.png

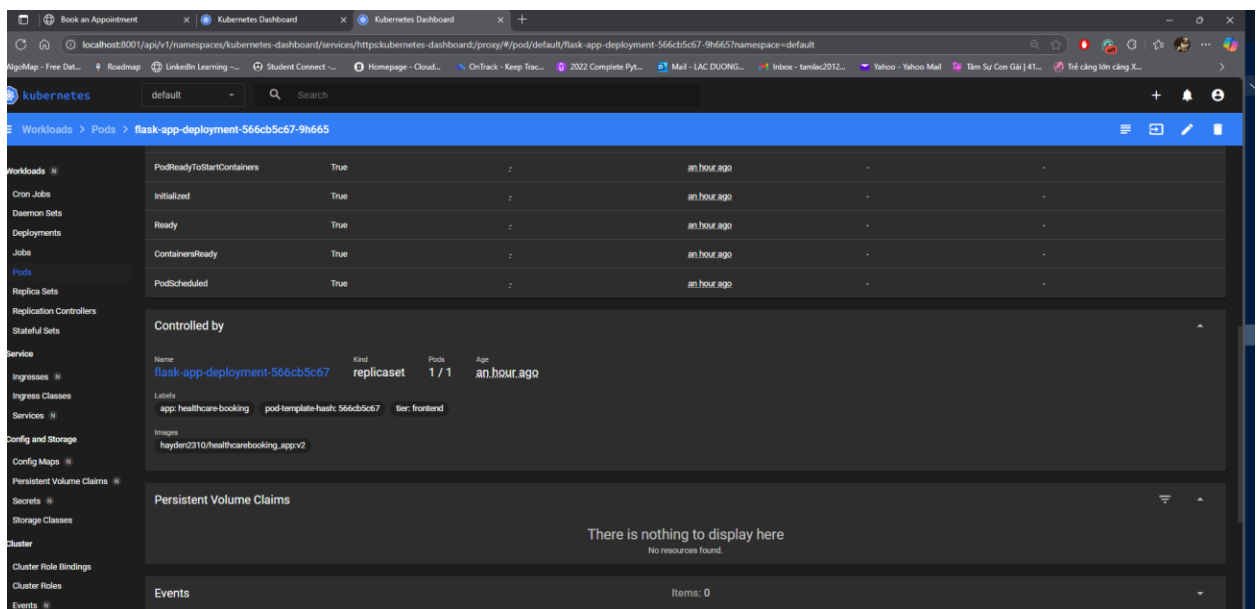
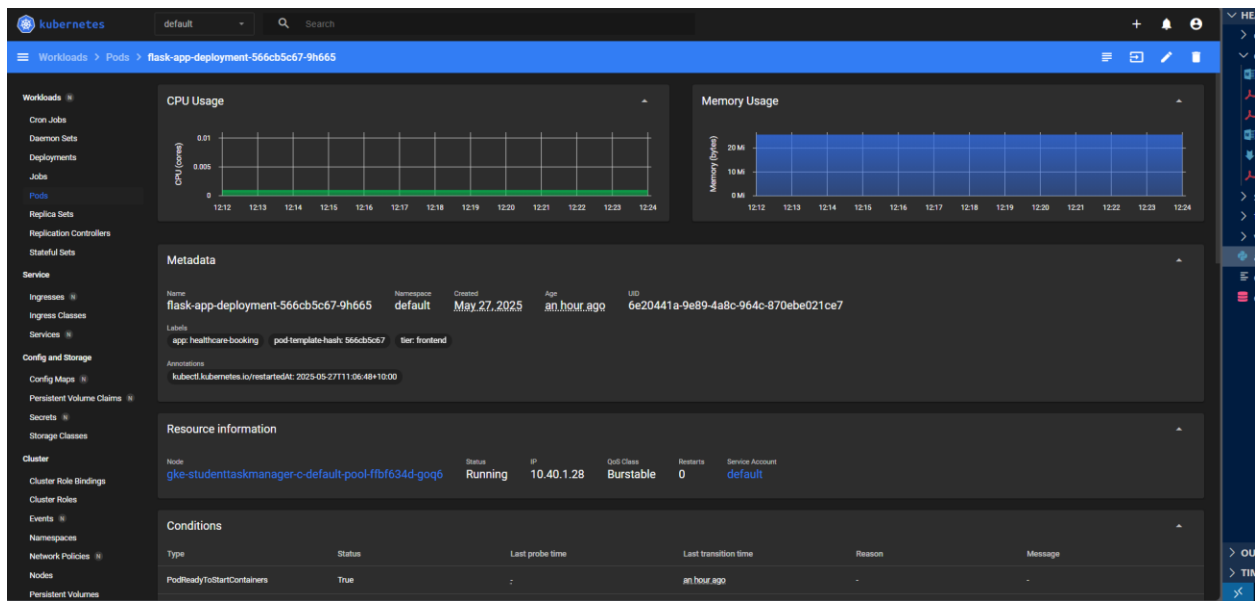


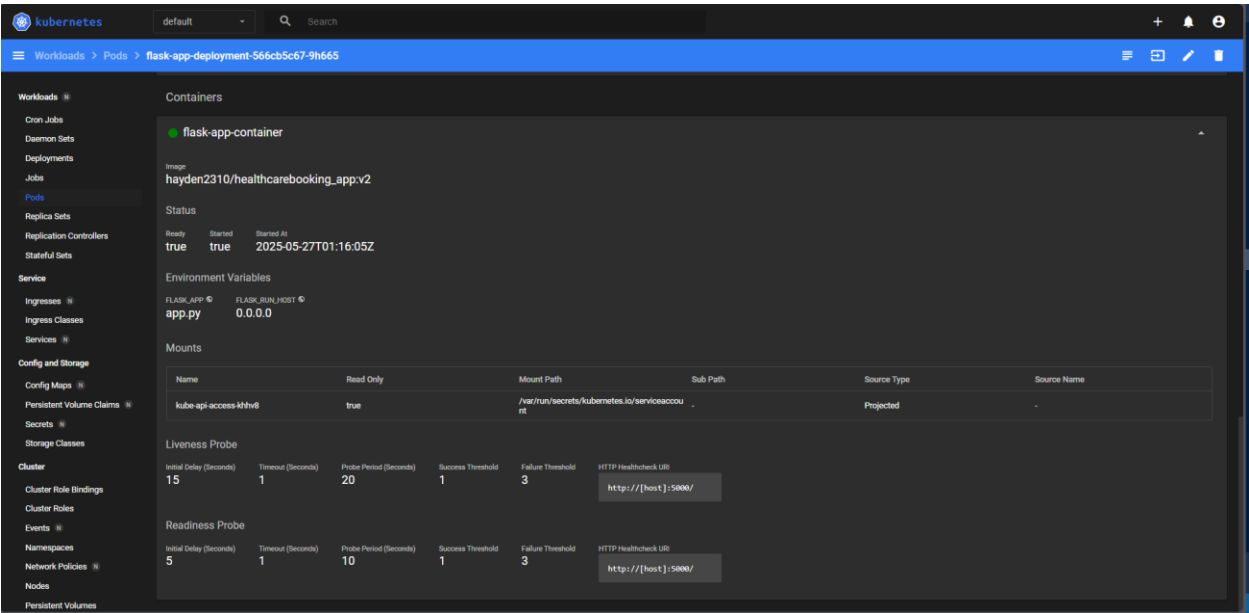
“k8s dashboard-metric.png”



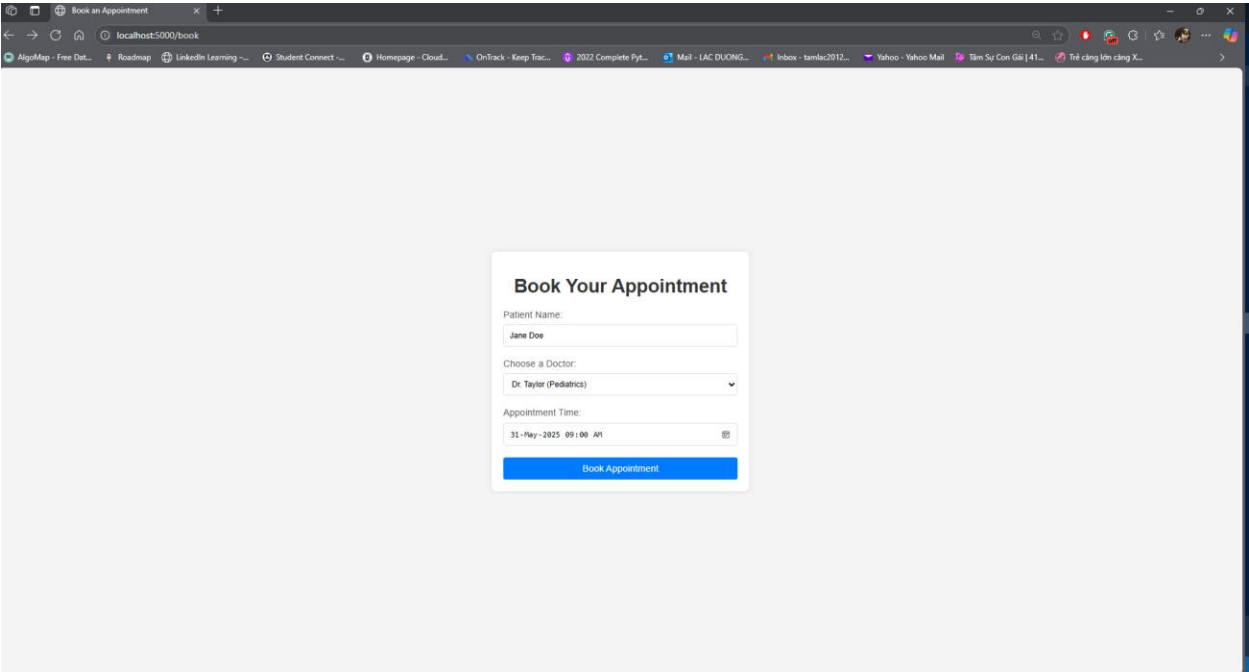


“flask-app-deployment-566cb5c67-9h665-metric-detail.png”





“e2e\_test.png”





```

Lac T. Duong@legion-7 MINGW64 /c:/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl exec -it postgres-0 -- psql -U postgres -d healthcare -c "EXPLAIN ANALYZE SELECT * FROM appointments WHERE appointment_time > '2025-05-01';"
QUERY PLAN
-----
Seq Scan on appointments (cost=0.00..1.02 rows=1 width=234) (actual time=0.011..0.012 rows=2 loops=1)
  Filter: (appointment_time > '2025-05-01 00:00:00'::timestamp without time zone)
Planning Time: 0.485 ms
Execution Time: 0.049 ms
(4 rows)

```

After applied optimization.sql

```

Lac T. Duong@legion-7 MINGW64 /c:/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl exec -it postgres-0 -- psql -U postgres -d healthcare -c "EXPLAIN ANALYZE SELECT * FROM appointments WHERE appointment_time > '2025-05-01';"
QUERY PLAN
-----
Seq Scan on appointments (cost=0.00..1.02 rows=1 width=234) (actual time=0.007..0.008 rows=2 loops=1)
  Filter: (appointment_time > '2025-05-01 00:00:00'::timestamp without time zone)
Planning Time: 0.330 ms
Execution Time: 0.042 ms
(4 rows)

```

“gke\_cluster.png”

The screenshot shows the Google Cloud console interface for the project 'SIT226-HealthCareBooking'. The 'Kubernetes engine / Clusters' page is active, displaying a table with one cluster: 'healthcarebooking-cluster-standard' in the 'Running' state. A terminal window is overlaid on the console, showing the following commands and output:

```

C:\Users\Lac T. Duong\AppData\Local\Google\Cloud SDK>gcloud container clusters get-credentials healthcarebooking-cluster-standard --region australia-southeast2 --project sit226-healthcarebooking
ERROR: (gcloud.container) Invalid choice: 'cluster'.
Maybe you meant:
gcloud container attached
gcloud container aws
gcloud container azure
gcloud container clusters
gcloud healthcare consent-stores
gcloud healthcare datasets
gcloud healthcare dicom-stores
gcloud healthcare fair-stores
gcloud healthcare h17v2-stores
gcloud healthcare operations
To search the help text of gcloud commands, run:
gcloud help -- SEARCH_TERMS
C:\Users\Lac T. Duong\AppData\Local\Google\Cloud SDK>gcloud container clusters get-credentials healthcarebooking-cluster-standard --region australia-southeast2 --project sit226-healthcarebooking
Getting cluster endpoint and auth data.
kubeconfig entry generated for healthcarebooking-cluster-standard.
C:\Users\Lac T. Duong\AppData\Local\Google\Cloud SDK>

```

Below the terminal window, a terminal snippet shows the output of 'kubectl get nodes' and 'kubectl config current-context'.

```

Lac T. Duong@legion-7 MINGW64 /c:/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl get nodes
NAME                                STATUS    ROLES    AGE    VERSION
gke-healthcarebooking-cl-default-pool-3bf5c501-h52s  Ready    <none>   4m11s  v1.32.3-gke.1927009
gke-healthcarebooking-cl-default-pool-7c2584b7-29f7  Ready    <none>   4m5s   v1.32.3-gke.1927009
gke-healthcarebooking-cl-default-pool-eddccb40-tz1k  Ready    <none>   4m11s  v1.32.3-gke.1927009
Lac T. Duong@legion-7 MINGW64 /c:/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl config current-context
gke_sit226-healthcarebooking_australia-southeast2_healthcarebooking-cluster-standard

```

“gke\_flask.png”

```

Lac T. Duong@legion-7 MINGW64 /c:/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl config current-context
gke_sit226-healthcarebooking_australia-southeast2_healthcarebooking-cluster-standard
Lac T. Duong@legion-7 MINGW64 /c:/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl get nodes
NAME                                STATUS    ROLES    AGE    VERSION
gke-healthcarebooking-cl-default-pool-3bf5c501-h52s  Ready    <none>   3d3h   v1.32.3-gke.1927009
gke-healthcarebooking-cl-default-pool-7c2584b7-29f7  Ready    <none>   3d3h   v1.32.3-gke.1927009
gke-healthcarebooking-cl-default-pool-eddccb40-tz1k  Ready    <none>   3d3h   v1.32.3-gke.1927009
Lac T. Duong@legion-7 MINGW64 /c:/Users/Lac T. Duong/Desktop/SIT226_727 - Cloud Automation Technologies/Coding Tasks/HealthcareBooking (main)
$ kubectl get pods -o wide
NAME                                READY    STATUS    RESTARTS   AGE    IP              NODE                                NOMINATED NODE    READINESS GATES
flask-app-deployment-7d994d895c-x6pxg  1/1      Running    0           11m    10.56.0.10      gke-healthcarebooking-cl-default-pool-7c2584b7-29f7  <none>             <none>
postgres-0                             1/1      Running    0           12m    10.56.0.9       gke-healthcarebooking-cl-default-pool-7c2584b7-29f7  <none>             <none>

```

Encountered errors:

A/ 2 Postgresql servers are running at the same time:

have **TWO** PostgreSQL servers running:

1. **The Docker Container (postgres-db):** This is the one running Debian, where you've been using docker exec to create the healthcare database. Its logs showed version 17.4 earlier.
2. **A Native Windows PostgreSQL Installation:** This is running directly on your Windows host, was compiled with Visual C++, and is version 16.2. This is the server that your test\_db.py and Flask app are actually connecting to on 127.0.0.1:5432. This native Windows PostgreSQL instance *does not* have the healthcare database.

When you run python app.py or python test\_db.py on your Windows machine and connect to 127.0.0.1:5432 (or localhost:5432), it's connecting to the PostgreSQL server that "answers" on that port *first* from the perspective of your Windows host. It seems your native Windows PostgreSQL installation is taking precedence or is the one configured to listen on that specific IP/port combination in a way that psycopg2 resolves to it.

Solutions:

- Open "Services" in Windows (search for services.msc).
- Look for a service related to PostgreSQL (e.g., postgresql-x64-16)
- Stop the Native Windows PostgreSQL Service.
  - In the Services window, right-click the PostgreSQL service and select "Stop".
  - To prevent it from starting on boot, right-click -> Properties -> Startup type: "Manual" or "Disabled".

## B/ Inter-Container Network Communication in Docker Environment

- **Issue Encountered:** An initial challenge arose when containerizing the Flask application and its PostgreSQL database. The Flask application, running in its Docker container, was unable to establish a connection with the PostgreSQL container. The application's database host configuration was initially set to localhost or 127.0.0.1, leading to connection errors such as psycopg2.OperationalError: connection refused.
- **Root Cause Analysis:** This issue stemmed from the network isolation inherent in Docker containers. Within a container's isolated network namespace, localhost refers to the container itself, not the host machine or other sibling containers. Consequently, the Flask application was attempting to connect

to a PostgreSQL instance it presumed to be within its own network boundaries, which was incorrect.

- **Resolution Implemented:** The problem was rectified by leveraging Docker's internal DNS resolution capabilities for containers sharing a common Docker network. The database host (DB\_HOST) parameter within the Flask application's configuration (app.py) was modified from localhost to the service name of the PostgreSQL container (i.e., postgres-db). This change enabled the Flask container to correctly resolve and connect to the PostgreSQL container using its designated service name, thereby establishing successful database communication. In addition, a network called healthcare-network through command: docker network create healthcare-network.

### **C/ Kubernetes Pod Scheduling and Resource Allocation for Horizontal Pod Autoscaling (HPA)**

- **Issue Encountered:** Upon deploying the Flask application to the Kubernetes cluster (via Docker Desktop), difficulties were observed with pod scheduling and the functionality of the Horizontal Pod Autoscaler (HPA). New pods, whether created by the initial deployment or by HPA scaling events, frequently entered a Pending state. Examination of pod events using kubectl describe pod revealed Warning FailedScheduling messages, citing "Insufficient CPU" as the cause, even when the node appeared to have available capacity. Furthermore, the HPA initially displayed <unknown> for CPU utilization targets, indicating an inability to retrieve necessary metrics.
- **Root Cause Analysis:** These issues were traced to aspects of Kubernetes' resource management and HPA configuration:
- **Resource Requests and Limits:** Kubernetes schedules pods based on their declared resource requests (e.g., CPU, memory). If the sum of the requests of all pods on a node (or attempting to be scheduled on it) exceeded the node's allocatable capacity, new pods could not be scheduled. The initial CPU request per pod might have been set too high for the single-node Docker Desktop Kubernetes environment, particularly when HPA attempted to scale out.
- **HPA Metrics Dependency:** For the HPA to effectively monitor and scale based on CPU utilization, the target pods' container specifications must include defined requests for CPU. An absence or misconfiguration of resources.requests.cpu in the Deployment YAML prevented the Metrics Server from reporting, and thus the HPA from acting upon, the pods' CPU usage.

- **Resolution Implemented:** A multi-faceted approach was adopted to resolve these Kubernetes-specific challenges:
- **Resource Request Optimization:** The cpu and memory requests within the flask-deployment.yaml for the application container were carefully reviewed and adjusted to more modest values (e.g., requests.cpu reduced to 50m). This ensured that the cumulative resource requests, even during HPA scale-out events, remained within the allocatable capacity of the local Kubernetes node.
- **HPA Configuration Tuning:** The flask-hpa.yaml was modified to set a minReplicas of 1 and a maxReplicas of 3, aligning the scaling behavior with the resource capacity of the test environment. The initial replicas in the Deployment was also set to 1. These changes provided a more stable baseline and prevented premature resource exhaustion during scaling.
- **Ensuring Metric Availability:** The explicit definition of resources.requests.cpu in the Deployment manifest was confirmed, enabling the Kubernetes Metrics Server to collect and provide the necessary data for the HPA to function correctly, which was subsequently reflected in the HPA status showing actual CPU utilization percentages.
- **Issue Encountered:**

During the deployment of PostgreSQL using a Kubernetes StatefulSet with a PersistentVolumeClaim (PVC) managed by volumeClaimTemplates, the postgres-0 pod failed to reach a Ready state, repeatedly entering a CrashLoopBackOff cycle. Examination of the pod's logs (kubectl logs postgres-0 -c postgres) revealed the following critical error message from the initdb process:

```
initdb: error: directory "/var/lib/postgresql/data" exists but is not empty

initdb: detail: It contains a lost+found directory, perhaps due to it being a mount point.

initdb: hint: Using a mount point directly as the data directory is not recommended. Create a subdirectory under the mount point.
```

- **Root Cause Analysis:**

The official PostgreSQL Docker image's entrypoint script executes initdb to initialize a new database cluster if the designated data directory is empty. In Kubernetes, when a PersistentVolume (PV) is provisioned (often by a cloud provider's storage class) and bound to a PVC, the underlying filesystem on the PV might not be completely devoid of files or directories. Specifically, it is common for new filesystems to contain



a lost+found directory at their root. When this PV was mounted into the postgres-0 pod at the default PostgreSQL data path /var/lib/postgresql/data, the initdb script detected the presence of the lost+found directory. As a safety precaution against data corruption or misconfiguration, initdb refuses to initialize a database cluster in a non-empty directory. This caused the PostgreSQL server to fail its startup sequence, leading to container termination and the observed CrashLoopBackOff behavior.

- **Resolution Implemented:**

To resolve this initialization conflict, the PostgreSQL data directory within the container was configured to use a dedicated subdirectory inside the mounted persistent volume. This was achieved by modifying the postgres-statefulset.yaml manifest to include the PGDATA environment variable for the PostgreSQL container:

```
# In deployment/postgres-statefulset.yaml
# ...
spec:
  containers:
    - name: postgres
      image: postgres:16
      # ...
      env:
        # ... (other environment variables like POSTGRES_DB, POSTGRES_USER, POSTGRES_PASSWORD)
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata # Instructs PostgreSQL to use this subdirectory
      volumeMounts:
        - name: postgres-data
          mountPath: /var/lib/postgresql/data # The PVC is mounted here
      # ...
```

By setting PGDATA to /var/lib/postgresql/data/pgdata, the PostgreSQL entrypoint script was instructed to perform the initdb process within this specified subdirectory. Since

this pgdata subdirectory would be empty upon its first creation within the mounted volume, initdb could proceed without error.

The lost+found directory at the root of the mounted volume (/var/lib/postgresql/data/lost+found) no longer interfered with the database initialization. After applying this configuration change to the StatefulSet and allowing Kubernetes to recreate the postgres-0 pod, the PostgreSQL container started successfully, and the pod achieved a Running and Ready state. This confirmed that the database was correctly initialized and operational within its persistent storage.