Exercise: Flask API, GitHub Actions, and Persistent Data

Juan David Guarnizo Hernandez, PhD
juan.guarnizo@tec.mx

TC2007B - AD2025

Goals

By the end of this comprehensive exercise you will be able to:

1. Create a minimal Flask API using Python.
2. Build and publish a containerized version of the API to DockerHub.
3. Manually publish an image to the GitHub Container Registry (GHCR).
4. Configure a GitHub Actions workflow to automatically build and publish container images.
5. Deploy and test the containerized API, including exchanging images with a partner.
6. Expand a Flask API to include full CRUD (Create, Read, Update, Delete) functionality.
7. Capture and analyze live HTTP traffic to your API using `tcpdump` and Wireshark.

This exercise assumes you are running a Kali Linux virtual machine on your personal device. We encourage you to work in pairs, as you will be publishing and testing images with a partner.

---

0) Software and Files Required

- **Kali Linux:** A running instance (Guest OS in a VM is recommended).
- **Docker:** To be installed (see installation note below).
- **Docker Compose:** To be installed (often included with Docker).
- **Accounts in:**
  - Github - https://github.com
  - DockerHub - https://hub.docker.com

System Installation

You must install Docker on your Kali Linux system before starting. Please follow the official Kali documentation to install `docker.io`.

- **Installation Guide:** https://www.kali.org/docs/containers/installing-docker-on-kali/

After installation, be sure to start and enable the Docker service:

```
sudo systemctl start docker
sudo systemctl enable docker
```

This exercise also uses `jq` to format JSON output, which you can install with:

```
sudo apt update
sudo apt install -y jq
```

You will also need to create accounts in `Github` and `DockerHub` if you do not have them already.

Part 1: Docker Clean and Minimalist Flask API

In our last session, we interacted with Docker, but you probably ended up with many unused containers and images. Let's clean up our environment.

1.1) Clean Up Docker Environment

These commands stop and remove all containers. Be careful if you have other work running.

```
docker stop $(docker ps -a -q) 2>/dev/null || true
docker rm $(docker ps -a -q) 2>/dev/null || true
```

Let us also delete all images in our system. First, list them:

```
docker image ls
```

Then, remove any images you no longer need using their ID or REPOSITORY:TAG.

```
docker image rm -f $DOCKER_IMAGE_ID_OR_NAME
```

1.2) Flask Super Minimalistic API

Let's create a container image for an API written in Flask. This means we will use `python:latest` as our base image.

Create a new folder for your project, and inside it, create two files: `app.py` and `Dockerfile`.

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/")
def index():
    # Change the message to something unique
    return jsonify({"message": "it works! Student XXX"})

if __name__ == "__main__":
    app.run(threaded=True, host='0.0.0.0', port=3000)
```

app.py

```
FROM python:latest

WORKDIR /app

COPY . .
RUN pip install Flask

EXPOSE 3000

CMD ["python", "app.py"]
```

Dockerfile

Task: Build, Push to DockerHub, and Test

- Build this image: `docker build -t my-flask-api:1.0 .`
- Push it to your DockerHub account (you will need to `docker login` first, then `docker tag` it with your username, e.g., `your-user/my-flask-api:1.0`, and then `docker push`).
- Can you deploy and run your new image? (Remember to use `-p 3000:3000`)
- Access the service from your laptop's browser at `http://<VM_IP>:3000`.
- Ask your partner for their image from DockerHub and deploy it.

Part 2: Create GitHub Repo & Manually Publish to GHCR

Now, let's get our code into GitHub and our initial image into the GitHub Container Registry (GHCR).

2.1) Create GitHub Repository

Create a new Git repository on GitHub. Once created, commit and push your project files (`app.py` and `Dockerfile`) to the repository.

One you commit and push, your directory structure should look like this:

```
my-api/
 |-- app.py
 |-- Dockerfile
```

System Installation

Do not include any secrets or passwords in your code or repository. This is an exercise that required public repositories.

Part 3: Configure GitHub Actions for Automation

Now that our repository and initial package are on GitHub, we will set up GitHub Actions to automate all future builds and pushes.

3.1) Set Up GitHub Actions

GitHub Actions is a tool that allows you to execute processes when changes are pushed to your repository. We will use it to publish our image to the GitHub Container Registry (GHCR).

1. In your GitHub repository, go to the **Actions** tab.
2. You will see suggestions. Search for the template **Simple Workflow**, and then click on **Configure** to set it up.
3. This will create a new file at `.github/workflows/blank.yml`. **Commit changes** to save this default file.
4. Go back to the **Actions** tab. You will see your "Simple workflow" has run.
5. Make a small change to your `app.py` (e.g., change the message), then commit and push the change. Observe in the **Actions** tab how this new push triggers the workflow again.

3.2) Modify Workflow to Publish to GHCR

Now we will modify the `.github/workflows/blank.yml` file to build and publish our Docker image. Edit the file to contain the following:

```yaml
name: CI

on:
  push:
    branches: [ "main" ]

# New Content: define permissions and environment variables
permissions:
  contents: read
  packages: write

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ghcr.io/${{ github.repository }}/my-api

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      # ... previous steps remain the same ...

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Build and push
        uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ${{ env.IMAGE_NAME }}:latest
          cache-from: type=gha
```

```
          cache-to: type=gha,mode=max
```

.github/workflows/docker-publish.yml

Commit and push this new workflow file.

Task: Deploy from GHCR

How can you deploy this newly created `latest` image from GHCR into your VM environment?

Part 4: Expanding the Flask API (In-Memory)

Now that our automation is working, let's expand our API to be a "news" service with full CRUD functionality. We will implement these endpoints:

- `GET /`: returns a list of available endpoints
- `GET /news`: returns a list of news articles
- `POST /news`: creates a news article from JSON
- `PUT /news/<int:id>`: updates an existing article
- `DELETE /news/<int:id>`: deletes an existing article

For simplicity, we'll use an in-memory Python dictionary (data resets on app restart).

4.1) Update app.py

Replace the content of your `app.py` with this skeleton:

```python
from flask import Flask, jsonify, request, abort

app = Flask(__name__)

# In-memory store
news = [
    {"id": 1, "title": "Initial News", "content": "This is the first article."}
]
# We use a global variable for the counter
global next_id
next_id = 2 # simple auto-increment for IDs

@app.route("/", methods=["GET"])
def index():
    return jsonify({
        "message": "Welcome to the News API!",
        "endpoints": {
            "list_all_news": "GET /news",
            "create_news": "POST /news",
            "update_news": "PUT /news/<id>",
            "delete_news": "DELETE /news/<id>"
        }
    })

@app.route("/news", methods=["GET"])
def list_news():
    return jsonify({"count": len(news), "items": news})

```

```python
@app.route("/news", methods=["POST"])
def create_news():
    global next_id
    if not request.json or not 'title' in request.json:
        abort(400) # Bad request

    new_item = {
        'id': next_id,
        'title': request.json['title'],
        'content': request.json.get('content', "")
    }
    news.append(new_item)
    next_id += 1
    return jsonify(new_item), 201 # Created

# Helper function to find an item
def find_news_item(item_id):
    for item in news:
        if item['id'] == item_id:
            return item
    return None

@app.route("/news/<int:item_id>", methods=["PUT"])
def update_news(item_id: int):
    item = find_news_item(item_id)
    if not item:
        abort(404) # Not found
    if not request.json:
        abort(400)

    # Update fields
    if 'title' in request.json:
        item['title'] = request.json['title']
    if 'content' in request.json:
        item['content'] = request.json['content']

    return jsonify(item)

@app.route("/news/<int:item_id>", methods=["DELETE"])
def delete_news(item_id: int):
    item = find_news_item(item_id)
    if not item:
        abort(404)

    news.remove(item)
    return jsonify({"status": "deleted", "id": item_id})

if __name__ == "__main__":
    app.run(threaded=True, host='0.0.0.0', port=3000)
```

app.py (In-Memory Version)

Task: Commit, Push, & Smoke Test

Now, commit and push this change for `app.py` to your repository.

- Go to your GitHub **Actions** tab and watch your workflow run.
- Once it finishes, pull the new `latest` image from GHCR and run it.
- Quick smoke tests with `curl` (you may need to `apt install jq`):

```
# Test 1: POST a new item
curl -X POST http://localhost:3000/news -H "Content-Type: application/json"
    -d '{"title": "Network capture day", "content": "Bring Wireshark!"}' |
    jq

# Test 2: List all items
curl "http://localhost:3000/news" | jq
```

Part 5: Capturing & Analyzing HTTP Traffic

Let's capture the unencrypted traffic to our API. You'll run `tcpdump` on the VM while your partner generates requests from their laptop or phone.

5.1) Find your VM's IP and Interface

```
ip addr show
# or
ip -br a
```

Identify the primary interface (e.g., `eth0`, `ens3`) and the VM IP address. Share that IP with your partner so they can access `http://<VM_IP>:3000`.

5.2) Start a Focused Capture

Run `tcpdump` to capture packets on port 3000 and write them to a file.

```
# Replace eth0 with your interface if different
sudo tcpdump -i eth0 -w news-api.pcapng -s 0 tcp port 3000
```

5.3) Generate Traffic

Let `tcpdump` run. Ask your partner to generate traffic from their device (replace `<VM_IP>` with your IP):

```
# Create a few items
curl -X POST http://<VM_IP>:3000/news -H "Content-Type: application/json" \
 -d '{"title": "From phone", "content": "Posting from my phone"}' | jq

# List items
curl http://<VM_IP>:3000/news | jq

# Update item with id 1 (if exists)
curl -X PUT http://<VM_IP>:3000/news/1 -H "Content-Type: application/json" \
-d '{"content": "Edited from laptop"}' | jq

# Delete item with id 1 (if exists)
curl -X DELETE http://<VM_IP>:3000/news/1 | jq
```

5.4) Stop Capture and Transfer

Stop `tcpdump` with `Ctrl+C`. Copy `news-api.pcapng` to your laptop (e.g., using `scp` or `WinSCP`).

```
# From your laptop:
scp student@<VM_IP>:~/news-api.pcapng ./
```

5.5) Analyze in Wireshark

Open `news-api.pcapng` in Wireshark.

- Use the display filter `http` to see only HTTP traffic.
- Click on a `POST` request. In the packet details pane, expand `Hypertext Transfer Protocol`. You will see the headers.
- Expand the `JavaScript Object Notation` section to see the JSON payload in clear text.
- Right-click a packet and choose **Follow > TCP Stream** to see the raw request and response.