

# Cloud Computing and SE

## Assignment 4

Dr. Daniel Yellin

THESE SLIDES ARE THE PROPERTY OF DANIEL YELLIN  
THEY ARE ONLY FOR USE BY STUDENTS OF THE CLASS  
THERE IS NO PERMISSION TO DISTRIBUTE OR POST  
THESE SLIDES TO OTHERS

# Due date

Assignment is due Feb 03, 2026 at 23:55  
03/02/2026 at 23:55

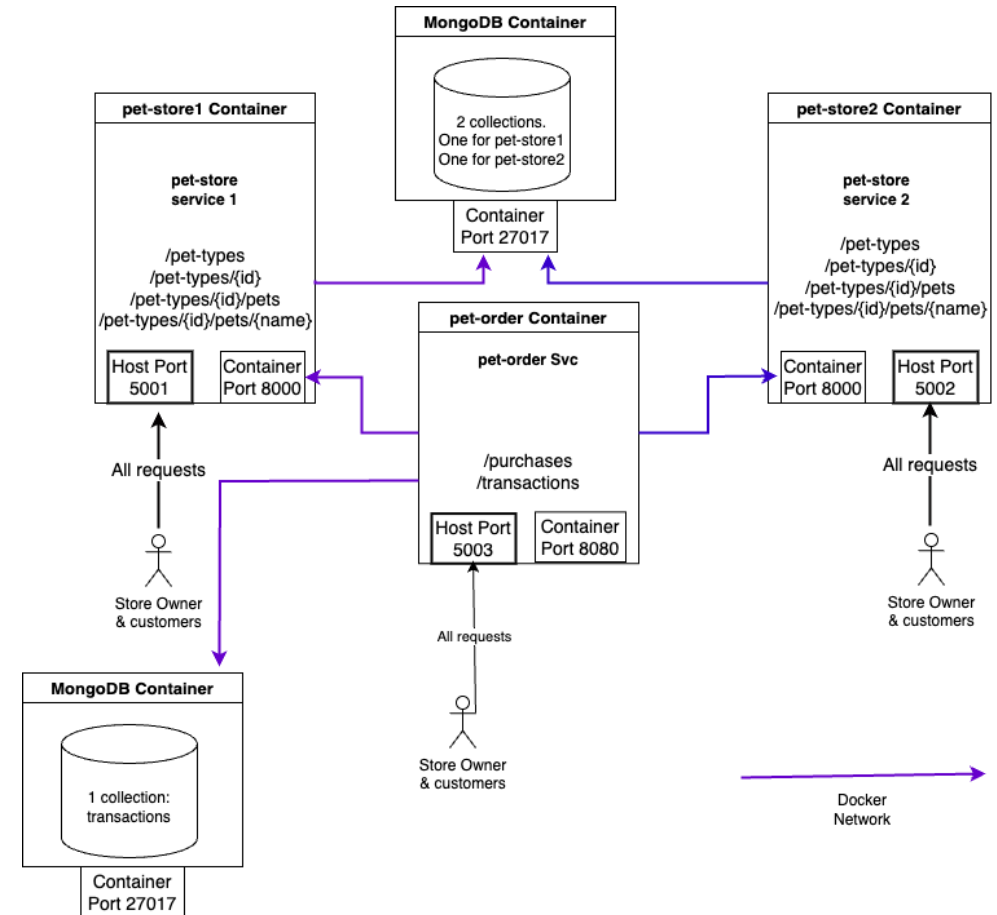
# Assignment #4: CI/CD for the Pet Store app

- In assignment #4, you will build a GitHub Actions CI/CD pipeline for the Pet Store application you built in assignment #2. You will use 2 instances of the pet-store service, the pet-order service and 2 instances of MongoDB.
- For this assignment you will **not** need: NGINX, load balancing, the Docker Compose “restart” command, nor the special “kill” request.
- The pet-store and pet-order services you use for this assignment should be the same as the ones required for assignment #2.
  - If you had any mistakes in assignment #2, fix them before using that code for assignment #4.

# The Pet Store application for assignment #4

The Pet Store application is the same as assignment #2 except NGINX not required:

- 2 instances of the pet-store service (host ports 5001/5002)
- Both use the same instance of MongoDB
- 1 instance of the pet-order service (port 5003)
- The pet-store services use one instance of Mongo and the pet-order service uses a separate instance of Mongo
- All customer and owner requests to the pet-store and pet-order services are addressed to the host port.
- Requests from one service to another are through the container ports.



# Overview of assignment #4

- You need to create a **new GitHub repository** in which to put your code, your Dockerfiles, Docker-compose.yml (if you use it), any other needed artifacts for your application, your pytest tests, and your GitHub Actions workflow.
- The workflow is **triggered by a push event** to the repository.
- The workflow must have **3 different jobs**:
  1. The first job builds the images for your pet-store and pet-order service. It is called the *build* job. If successful, it proceeds to the second job.
  2. The second job is called the *test* job. It uses the images from the first job and a MongoDB image (pulled from DockerHub) to run the Pet Store application in containers. It runs a pytest script to test the application. If the tests are successful, it proceeds to the third job.
  3. The third job, called the *query* job, also runs the Pet Store app in containers. This will dynamically determine which tests to run, as will be explained shortly.

# Image and ports and workflow name

- The pet-store #1 (#2) service must listen on **localhost port 5001 (5002)** and the pet-order service must listen on **localhost port 5003**.
- This is important because the tester will assume he can issue requests to those ports.
- The workflow needs to be stored in the file “assignment4.yml” in the subdirectory `/.github/workflows` of your repository
- The name of the workflow is “assignment4”

# Output files

When the workflow terminates, the following artifacts should be available on GitHub:

1. A log file.
2. The results of running the pytest tests in the test job.
3. An output file that gives the results for specific requests specified in the 3rd job.

You should use the `actions/upload-artifact@v4` demonstrated in class to make these artifacts available after the workflow terminates.

# Log file (available on workflow termination)

The log file is a text file. It should have the name “log.txt”. It contains the following lines (each line is terminated with the ‘\n’ character):

- Line 1: The time the workflow starts executing. The format of the date should be the result of the command `bash command: “date -lminutes”` at the start of the workflow. (after the “-” symbol is a capital “i”).
- Line 2: The name(s) of the submitter(s) of this assignment, each name separated by a comma.
- Line 3: If the image for service X (X is one of ‘pet-store’ or ‘pet-order’) is successfully built in job 1, then “image X successfully built”. Otherwise (the image not successfully built) “image X not able to be built”
- Line 4: If container Y (Y is one of ‘pet-store #1’, ‘pet-store #2’, or ‘pet-order’) is successfully started in job 2 then “Container Y up and running”. Otherwise (the container not started successfully) “Container Y failed to run”.
- Line 5: If the tests (in job 2) are successful then “All pytestes succeeded”. Otherwise (at least one test failed) then “pytestes failed”.

# Using pytest for testing

- The second job will run tests on your images. It must use the tool `pytest` (a python testing framework).
  - Even if you build your service in another language, the testing should be done with `pytest`.
- Your repository should have a sub-directory named “tests”.
- The file containing the pytests to be run should be in the subdirectory “tests” and should be named “`assn4_tests.py`”. If your pytests use any auxiliary files, they should also be in that directory.
- The workflow should execute the tests using the “-v” (verbose) option. E.g., “`pytest -v assn4_tests.py`”

# Test results (available on workflow termination)

- pytest produces output listing the test results. This output from pytest must be stored in a file named “asn4\_test\_results.txt”.
- This file should always be uploaded to GH whenever job 2 executes, even if some of the tests fail (and the workflow does not complete).
- The בודק (tester) will run the workflow an additional time, with a different set of tests. He will do so by uploading a new version asn4\_tests.py. Hence your workflow must invoke pytest specifying that the tests are in the file asn4\_tests.py.

# The pytest tests to execute (1)

1. Execute 3 *POST /pet-types* requests to **pet store #1** with payloads PET\_TYPE1, PET\_TYPE2, and PET\_TYPE3.
2. Execute 3 *POST /pet-types* requests to **pet store #2** with payloads PET\_TYPE1, PET\_TYPE2, and PET\_TYPE4.

These tests are successful\* if (i) all requests to a specific store returns unique ids, (ii) the return status code from each POST request is 201, and (iii) the “family” and “genus” fields match the values given in PET\_TYPEX\_VAL. (These definitions are provided in the following slides.)

Let id\_1, id\_2,..., id\_6 be the ids returned from each of the 6 POST requests above.

3. Execute *POST /pet-types/{id\_1}/pets* to **pet-store #1** with payload PET1\_TYPE1 and another with payload PET2\_TYPE1. 2 POSTs in total.
4. Execute *POST /pet-types/{id\_3}/pets* to **pet-store #1** with payload PET5\_TYPE3 and another with payload PET6\_TYPE3. 2 POSTs in total.

These tests are successful if all requests return status code 201.

\* “The test is successful” means that your pytest tests for these conditions and all the tests pass.

# The pytest tests to execute (1)

5. Execute *POST /pet-types/{id\_4}/pets* to **pet-store #2** with payload PET3\_TYPE1,
6. Execute *POST /pet-types/{id\_5}/pets* to **pet-store #2** with payload PET4\_TYPE2.
7. Execute *POST /pet-types/{id\_6}/pets* to **pet-store #2** with payload PET7\_TYPE4 and another with payload PET8\_TYPE4. 2 POSTs in total.

These tests are successful if all requests return status code 201.

8. Execute *GET /pet-types/{id2}* to **pet-store #1**. The test is successful if (i) the JSON returned matches all the fields given in PET\_TYPE2\_VAL and (ii) the return status code from the request is 200.
9. Execute a *GET /pet-types/{id6}/pets* to **pet-store #2**. The test is successful if (i) the returned value is a JSON array containing JSON pet objects with the fields given in PET7\_TYPE4, and PET8\_TYPE4, and (ii) the return status code from the GET request is 200.

# *Pet Types*

```
PET_TYPE1 = {  
    "type": "Golden Retriever"  
}
```

```
PET_TYPE1_VAL = {  
    "type": "Golden Retriever",  
    "family": "Canidae",  
    "genus": "Canis",  
    "attributes": [],  
    "lifespan": 12  
}
```

```
PET_TYPE2 = {  
    "type": "Australian Shepherd"  
}
```

```
PET_TYPE2_VAL = {  
    "type": "Australian Shepherd",  
    "family": "Canidae",  
    "genus": "Canis",  
    "attributes": ["Loyal", "outgoing", "and", "friendly"],  
    "lifespan": 15  
}
```

```
PET_TYPE3 = {  
    "type": "Abyssinian"  
}
```

```
PET_TYPE3_VAL = {  
    "type": "Abyssinian",  
    "family": "Felidae",  
    "genus": "Felis",  
    "attributes": ["Intelligent", "and", "curious"],  
    "lifespan": 13  
}
```

```
PET_TYPE4 = {  
    "type": "bulldog"  
}
```

```
PET_TYPE4_VAL = {  
    "type": "bulldog",  
    "family": "Canidae",  
    "genus": "Canis",  
    "attributes": ["Gentle", "calm", "and", "affectionate"],  
    "lifespan": None  
}
```

# Pets

```
PET1_TYPE1 = {  
  "name": "Lander",  
  "birthdate": "05-14-2020"  
}
```

```
PET2_TYPE1 = {  
  "name": "Lanky"  
}
```

```
PET3_TYPE1 = {  
  "name": "Shelly",  
  "birthdate": "07-07-2019"  
}
```

```
PET4_TYPE2 = {  
  "name": "Felicity",  
  "birthdate": "27-11-2011"  
}
```

```
PET5_TYPE3 = {  
  "name": "Muscles"  
}
```

```
PET6_TYPE3 = {  
  "name": "Junior"  
}
```

```
PET7_TYPE4 = {  
  "name": "Lazy",  
  "birthdate": "07-08-2018"  
}
```

```
PET8_TYPE4 = {  
  "name": "Lemon",  
  "birthdate": "27-03-2020"  
}
```

# The query job and its output file

The query job will read a file named **query.txt** in your repository.

- The tester will upload this file.
- Each entry of query.txt file will be of the form: <query> or <purchase>, described on the next slide.
- The query job needs to create a new text file called **response.txt**
- You can test your job by providing your own query.txt file in the format to be described below.

# <query> and <purchase> formats

If the entry is a <query>, it will be of the format: “**query:** <pet-store-num>,<query-string>;”.

- <pet-store-num> is either “1”, or “2”.
- <query-string> will be of the form “<field>=<value>”, where field is one of {‘type’, ‘family’, ‘genus’, ‘lifespan’}. This is a query-string for a GET request to the pet-store for resource /pet-types.

If the entry is a <purchase>, it will be of the format: “**purchase:** <JSON-purchase>;” for a POST request to the pet-order service for resource /purchases. <JSON-purchase> is a JSON purchase object defined in HW#2:

```
{  
  "purchaser": string,  
  "pet-type": string,  
  "store": number,      // the store field is optional. If supplied, it equals 1 or 2.  
  "pet-name": string,   // pet-name is optional. Can only be supplied if store is supplied.  
  "purchase-id": string  
}
```

# Executing the query job (1)

1. Run the pet store application as in job #2 (the test job).
2. Execute the 6 POST */pet-types* and 8 POST */pet-types/{id\_x}/pets* as given in pytest steps 1-7.
3. Read the query.txt file in the root of your repository.
4. For each <query> entry execute the GET request on */pet-types* to the specified store with the given query string. In the response.txt file, insert the entry  
“<status-code> <nl> <payload>” where:
  - <status-code> is the status returned from the request
  - <nl> is a new line
  - <payload> is “NONE” if the status code is not 200
  - <payload> is the returned JSON payload if the status code is 200. (it can take up multiple lines)

## Executing the query job (2)

5. For each <purchase> entry execute the POST request on */purchases* with the given payload. In the response.txt file, insert the entry “<status-code><nl> <payload>” where:
  - <status-code> is the status returned from the request
  - <nl> is a new line
  - <payload> is “NONE” if the status code is not 201
  - <payload> is the returned JSON payload if the status code is 201. (it can take up multiple lines)
6. After each entry in the response.txt file, insert a “;” on a newline
7. The entries in response.txt should be in the same order as they are given in query.txt

## Executing the query job (3)

Hence the will generate a file **response.txt** of the following form:

<status-code>

<payload>

;

<status-code>

<payload>

;

...

<status-code>

<payload>

;

The job needs to upload to GitHub the file **response.txt** when it finishes recording the results.

# How we will grade this assignment

- You must give the tester (בודק) access to your repository as a collaborator. His GitHub username is: **NirltaCohen**. You should only give him permission close to the submission date (as invitations only last 1 week).
- The tester will:
  1. Upload a file "query.txt" to be used in job 3 of your workflow.
  2. Test that your workflow executes successfully on a push event.
  3. Will make changes to the files in your directory to see how they affect the workflow
    - He may change the Dockerfile so that the build fails
    - He may change code so that some of your tests fails
  4. Will replace assn4\_tests.py with a different set of pytest and check the results
  5. Will check that the 3 outputs from your workflow are correct.

# Submitting your assignment

When submitting your assignment, please **attach a document listing the team members**. This should be done even if submitting alone.