

# Cloud Computing and SE

## Assignment 3

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

# Assignment #3: CI/CD for dishes/meals service

- In assignment #3, you will build a GitHub Actions CI/CD pipeline for the dishes and meals service you built in assignment #1.
- The dishes and meals service you use for this assignment should be **exactly** the same as the one required for assignment #1:
  - If you had any mistakes in assignment #1, fix them before using that code for assignment #3.
  - If you have any questions about assignment #1, look at the instructions for assignment #1 given on-line, and look at the homework forum for clarifications.
  - NOTE: We changed some of the details of the dishes and meals services in assignment #2. For instance, in assignment #2 the GET /dishes and GET /meals requests returned a JSON array instead of a JSON object (containing embedded dish/meal objects) as it did in assignment #1. For this assignment we will revert back and use **exactly the same** requirements from assignment #1 and the service should return a JSON **object** with embedded JSON objects in the case of GET /dishes and GET /meals.
- This assignment can be done alone or in a group of two/three students together (no more than 3 in one group).

# Overview of assignment #3

- You need to create a **new repository** in which to put your code, your Dockerfile, 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**:
  - The first job builds the image for your service. It is called the *build* job. If successful, it proceeds to the second job.
  - The second job uses the image from the first job to run it in a container and uses pytest to test the service. It is called the *test* job. If successful, it proceeds to the third job.
  - The third job also runs the image in a container. It will issue specific requests to the service and record the results in a file. It is called the *query* job.

# Image and ports and workflow name

- The image you build must listen on **localhost port 8000**. (The same as in assignment #1)
- This is important because the tester will assume he can issue requests to the image at host port 8000.
- The workflow needs to be stored in the file “assignment3.yml” in the subdirectory `/.github/workflows` of your repository
- The name of the workflow is “assignment3”

# 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.
3. An output file that gives the results for specific queries specified in the 3rd job.

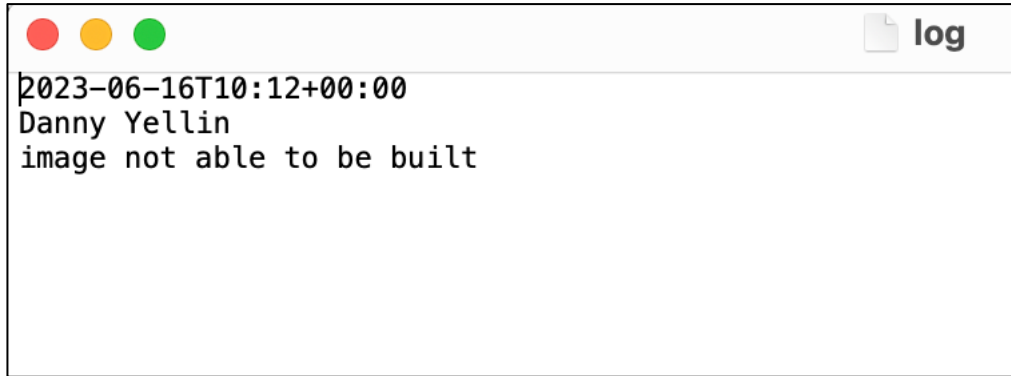
You should use the `actions/upload-artifact@v3` demonstrated in class to make these artifacts available after the workflow terminates.

# 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):

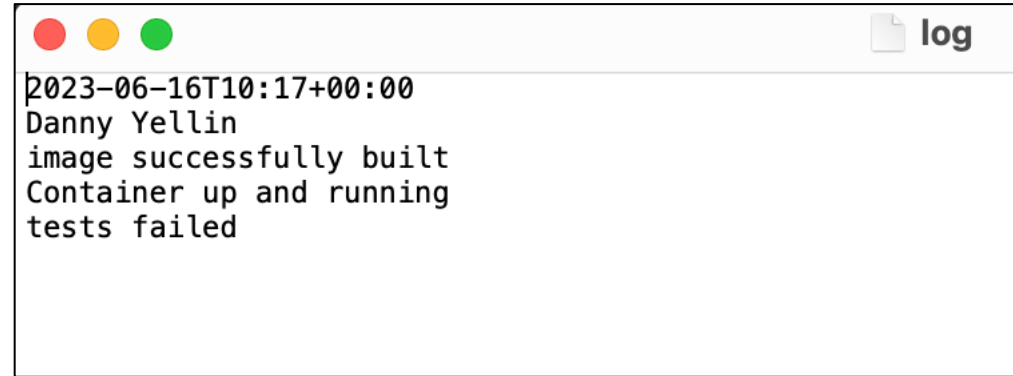
- Line 1: The time the workflow starts executing. The format of the date should be the result of the command bash command: “date –lminutes” at the start of the workflow.
- Line 2: The name(s) of the submitter(s) of this assignment
- Line 3: If the image is successfully built, then “image successfully built”. Otherwise (the image not successfully built) “image not able to be built”
- Line 4: If the container is successfully started then “Container up and running”. Otherwise (the container not started successfully) “Container failed to run”.
- Line 5: If the tests are successful then “tests succeeded”. Otherwise (at least one test failed) then “tests failed”.

# Log file (cont)

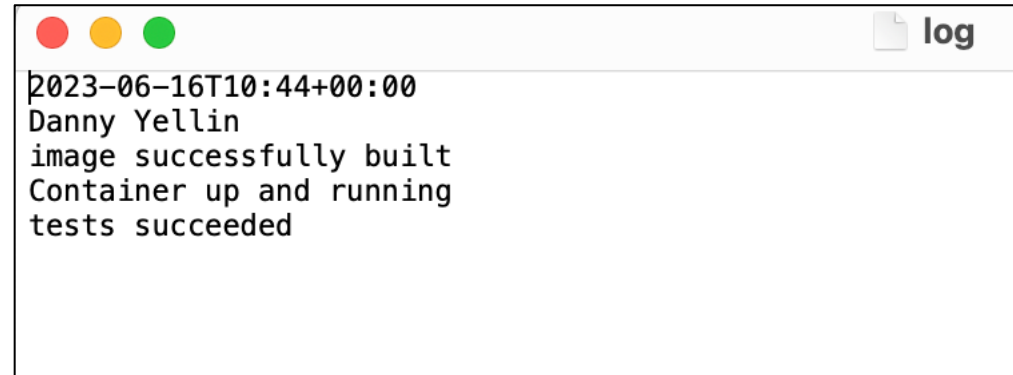


Example log when the step to build the image fails

If the workflow terminates because of an error in a step, then only the previous logs and the log relating to the error step (if there is one) needs to be included. For example, if the image is not successfully built, then lines 1-3 would be in the log but not line 4. Similarly, if the container fails to start, then lines 1-4 would be in the log, but not line 5.



Example log when the testing step fails



Example log when the workflow completes successfully

# Using pytest for testing

- The second job will run tests on your image. 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 “`assn3_tests.py`”. Depending on how you build your tests, you may want additional files to help testing. If so, they should also be in that directory.
- The workflow should execute the tests using the “-v” (verbose) option. E.g., “`pytest -v assn3_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 “asn3\_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 tester will run the workflow an additional time, with a different set of tests. He will do so by uploading a new version asn3\_tests.py. Hence your workflow must invoke pytest specifying that the tests are in the file asn3\_tests.py.

# The tests to execute

There are 8 tests to be executed in pytest against your image:

1. Execute three *POST /dishes* requests using the dishes, “orange”, “spaghetti”, and “apple pie”. The test is successful\* if (i) all 3 requests return unique IDs (none of the IDs are the same), and (ii) the return status code from each POST request is 201.
2. Execute a *GET dishes/<orange-ID>* request, using the ID of the orange dish. The test is successful if (i) the sodium field of the return JSON object is between .9 and 1.1 and (ii) the return status code from the request is 200.
3. Execute a *GET /dishes* request. The test is successful if (i) the returned JSON object has 3 embedded JSON objects (dishes), and (ii) the return status code from the GET request is 200.
4. Execute a *POST /dishes* request supplying the dish name “blah”. The test is successful if (i) the return value is -3, and (ii) the return code is 404 or 400 or 422.

\* “The test is successful” means that your pytest needs to test for these conditions.

# The tests to execute

5. Perform a *POST dishes* request with the dish name “orange”. The test is successful if (i) the return value is -2 (same dish name as existing dish), and (ii) the return status code is 400 or 404 or 422.
6. Perform a *POST /meals* request specifying that the meal name is “delicious”, and that it contains an “orange” for the appetizer, “spaghetti” for the main, and “apple pie” for the dessert (note you will need to use their dish IDs). The test is successful if (i) the returned ID > 0 and (ii) the return status code is 201.
7. Perform a *GET /meals* request. The test is successful if (i) the returned JSON object has 1 meal, (ii) the calories of that meal is between 400 and 500, and (iii) the return status code from the GET request is 200.
8. Perform a *POST /meals* request as in test 6 with the same meal name (and courses can be the same or different). The test is successful if (i) the code is -2 (same meal name as existing meal) and, and (ii) the return status code from the request is 400 or 422.

# 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.
- You can test your job by providing your own query.txt file in the format to be described below.
- The query job needs to create a new text file called **response.txt**
- Each line of queries.txt file will be of the form: <food-item>. I.e., it will be the name of a “dish”. Each line is terminated by a ‘\n’ character. Example of <food-item>s include:

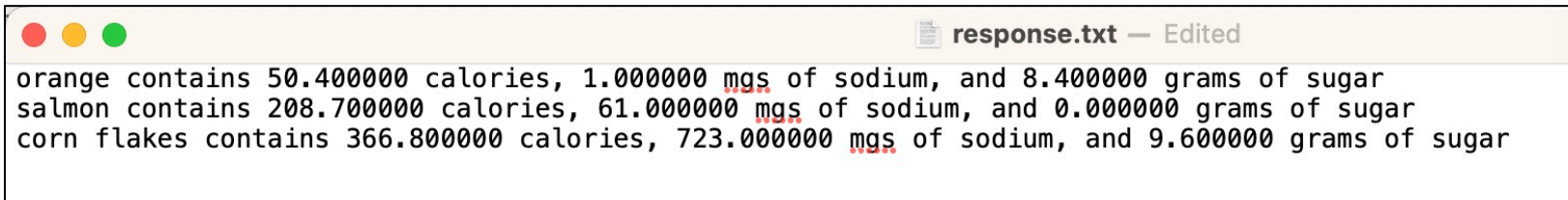
```
orange
salmon
corn flakes
```

Example query.txt file

# The query job and output file (cont)

Job 3 needs to:

1. Once again run your image in a container. Read each line of the “query.txt” file and using POST and GET requests to add and then retrieve the <food-item>, add a line to the **response** file of the following form:
  - “<food-item> contains X calories, Y mgs of sodium, and Z grams of sugar”
    - where X is the number of calories returned for <food-item>, Y is the number of mgs of sodium returned for <food-item>, and Z is the number of grams of sugar returned for <food-item>.
  - If line n in query.txt contains <food-item-n>, then line n of response.txt contains the response for <food-item-n>.
2. The job needs to upload to GitHub the file **response.txt** when it finishes recording the results.



```
orange contains 50.400000 calories, 1.000000 mgs of sodium, and 8.400000 grams of sugar
salmon contains 208.700000 calories, 61.000000 mgs of sodium, and 0.000000 grams of sugar
corn flakes contains 366.800000 calories, 723.000000 mgs of sodium, and 9.600000 grams of sugar
```

Example response file

# How we will grade this assignment

- You must give the tester (בודק) access to your repository as a **collaborator**
- The tester will:
  1. Upload a file "queries.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 fail
  4. Will replace assn3\_tests.py with a different set of pytests and check the results
  5. Will check that the 3 outputs from your workflow are correct.