

Wishify pytest and Status Page

Overview

Every 30 minutes, pytest is run on our server. The json-report plugin is used to record the results of the tests in a file. Then, a python script is run to create a processed version of this report file, and the previous report names are rotated. Currently, only the last 48 test runs are saved.

wishify.ca/status shows the results of the latest test run, as well as a summary of the last 48 tests based on their pass percentage. This frontend page makes a GET request to api.wishify.ca/status. This API request returns the report in which the “api_status_report” symlink points to on the server. This symlink is changed when a new report is generated in such a way to avoid possible race conditions.

File Structure

```
backend/
├── pytest/
│   ├── reports/
│   │   ├── processed_report_#
│   │   ├── recent_report
│   │   └── recent_report_history
│   ├── tests/
│   │   ├── conftest.py
│   │   ├── test_auth.py
│   │   ├── test_events.py
│   │   ├── test_users.py
│   │   └── test_wishlists.py
│   ├── api_status_report
│   ├── process_test_report.py
│   └── run_API_Tests.sh
```

Cron and run_API_Tests.sh

Every 30 minutes on the hour (XX:00) and half hour (XX:30), the script run_API_Tests.sh will be run on the server by cron.

```

backend > pytest > $ run_API_Tests.sh
1  #!/bin/bash
2
3  REPORT_DIR="reports/"
4  REPORT_FILE="recent_report"
5  PROCESSED_REPORT="processed_report"
6  MAX_HISTORY_SIZE=48
7
8  cd /var/www/4P02-course-project/backend/pytest
9
10 /home/ubuntu/.local/bin/pytest --json-report --json-report-omit keywords streams root --json-report-verbosity 2
11
12 /usr/bin/python3 process_test_report.py "$REPORT_DIR" "$REPORT_FILE" "$MAX_HISTORY_SIZE"
13
14 /usr/bin/ln -sft "${REPORT_DIR}${PROCESSED_REPORT}_0" api_status_report
15
16 for ((i="$MAX_HISTORY_SIZE"; i>=1; i--))
17 do
18     if [ -f "${REPORT_DIR}${PROCESSED_REPORT}_${i}" ]; then
19         END=$((i+1))
20         /usr/bin/mv "${REPORT_DIR}${PROCESSED_REPORT}_${i}" "${REPORT_DIR}${PROCESSED_REPORT}_${END}"
21     fi
22 done
23
24 END=$((MAX_HISTORY_SIZE+1))
25
26 if [ -f "${REPORT_DIR}${PROCESSED_REPORT}_${END}" ]; then
27     /usr/bin/rm "${REPORT_DIR}${PROCESSED_REPORT}_${END}"
28 fi
29
30
31 /usr/bin/cp "${REPORT_DIR}${PROCESSED_REPORT}_0" "${REPORT_DIR}${PROCESSED_REPORT}_1"
32
33 /usr/bin/ln -sft "${REPORT_DIR}${PROCESSED_REPORT}_1" api_status_report

```

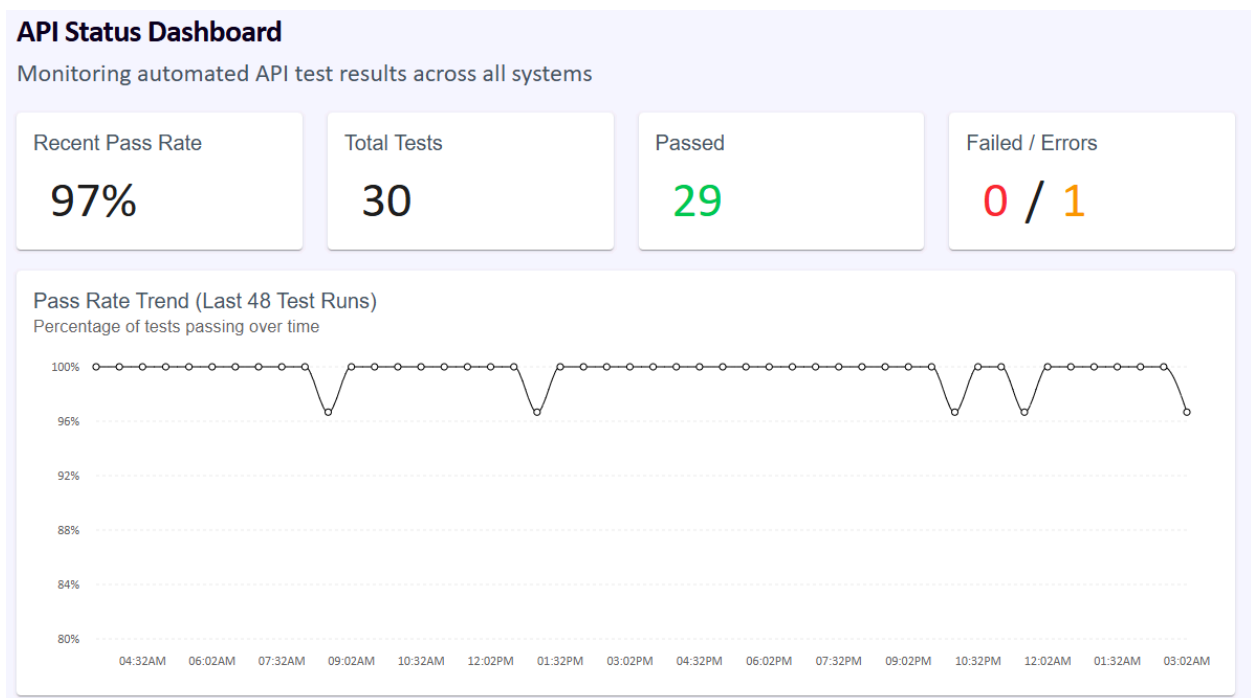
- The script will run **pytest** (line 10) and then run the **process_test_report.py** script (line 12).
- The **pytest** command will output the raw report as the **recent_report** file in the **reports/** directory.
- The **process_test_report.py** script will create a processed version of the raw output. The processed output will be in a file called **processed_report_0**
- The symlink, **api_status_report** will temporarily point at this new **processed_report_0** file, while the others are renamed. The older the file, the higher the number. With the current parameters, if the file ends in **_49** it will be deleted.
- The symlink **api_status_report** is set back to **processed_report_1** after it is copied from **processed_report_0**. **processed_report_0** is not deleted to avoid any possible race conditions, like fulfilling a request that started when the symlink still pointed at **processed_report_0**, and the request finished after the symlink was changed.

Processing Reports

There is a lot of information in the report created by the **pytest json-report** plugin that is not needed, so the **process_test_report.py** script takes this raw output file and creates a cleaner version for frontend use. This script captures the test results of every module/file, and makes the strings palatable for the frontend (e.g. extracts test name from the function name, and extracts category name from the file name)..














The history of past test results are stored in **recent_report_history**. This file simply stores an array of past tests, which includes the time they happened, and their passed percentage, with the most recent entries at the beginning of the array. When **process_test_report.py** runs, the file is read, the new entry is added, then the file is overwritten. This updated history array is then put inside the output **processed_report_0** file.

Status Page



The summary statistics of the latest test are shown at the top of the page, and the graph shows the history of the last 48 tests (last 24 hours since tests run every 30 minutes).

The time of the reports are based on the server time, and is **not** translated to the browser's timezone

Test Categories	
Expand each category to see individual test results	
Auth	 ^
Create Account	
Create Account Duplicate Email	
Create Account Missing Data	
Login Account	
Login Missing Credentials	
Login Account Incorrect Credentials	
Logout	
Auth Me	
Auth Me Token Errors	
Events	 v
Users	 v
Wishlists	 v

The Test Categories section shows the tests that were run and their result.

- **Green** indicates the test passed
- **Yellow** indicates there was an issue with either setting up before the test, or cleaning up after. Yellow does not mean the test failed, but rather an error occurred within the process of conducting the test.
- **Red** indicates the test failed.

The categories are created based on the name of the test file. For example, **test_auth.py** contains all of the tests within the **Auth** category.

The test names shown on the frontend are taken from the name of the test function. For example, **test_create_account_duplicate_email** is the test **Create Account Duplicate Email** shown on the frontend.

How To Add More Tests

Brief pytest Breakdown

The file **conftest.py** contains the fixtures available to all of the test functions. **Fixtures** are functions that can be called by a test function to perform some setup or cleanup. The tests should be atomic, meaning all resources needed for a test should be created before the it, and then destroyed after.

As an example, we will look at the **test_create_account_duplicate_email** function.

```
def test_create_account_duplicate_email(setup_test_account, cleanup_test_account):
    sleep(sleepTime)
    res = req.post(
        domain+"/auth/register",
        json={
            "email":email,
            "password":password,
            "displayName":"Automated Test Account"
        }
    )

    assert res.status_code == 409
```

This function tests if a new account can be created if an account already exists with the given email address. For this test, we need a test account with a known email, so we call **setup_test_account** and **cleanup_test_account** in the function parameters. **pytest** will run these fixtures for us when the test is run.

Reminder: these fixtures are declared in the **conftest.py** file

```
@pytest.fixture
def setup_test_account(request):
    sleep(sleepTime)
    res = req.post(
        domain+"/auth/register",
        json={
            "email":email,
            "password":password,
            "displayName":"Automated Test Account"
        }
    )
```

This setup fixture is short and sweet, it simply creates the test account.

```

@pytest.fixture
def cleanup_test_account(request):
    token = None

    def _method(tok):
        nonlocal token
        token = tok

    yield _method
    sleep(sleepTime)

    if token is None:
        res = req.post(
            domain+"/auth/login",
            json={
                "email":email,
                "password":password
            }
        )
        token = res.json()["token"]

    sleep(sleepTime)
    res = req.delete(
        domain+"/users",
        headers={"Authorization": f"Bearer {token}"},
        json={
            "password":password
        }
    )

```

When you define a fixture for cleanup, you must use the **yield** keyword. The code before **yield** will run before the test, and the fixture will continue with the code after the **yield** after the test. When you call **yield**, you can return a value or function. In the case of **cleanup_test_account** above, it defines a function called **_method()**, and provides this function when it yields. This function can then be used by a test to pass data to the cleanup fixture. This is especially important when the cleanup requires an ID in order to delete something, or in this case, a token if it has already been fetched prior in the execution of the test. The variables used in **_method()** must be declared **nonlocal** so the right scope is used.

```
def test_login_account(setup_test_account, cleanup_test_account):
    sleep(sleepTime)
    res = req.post(
        domain+"/auth/login",
        json={
            "email":email,
            "password":password,
        }
    )

    cleanup_test_account(res.json()["token"])

    assert res.status_code == 200
    assert res.json()["token"]
```

To pass the data using the yielded function, you use the fixture's name. In the above screenshot, `cleanup_test_account(res.json()["token"])` is passing `res.json()["token"]` to the `_method()` we saw yielded by the `cleanup_test_account` fixture.

```
def test_put_wishlist(setup_test_account, setup_test_wishlist, cleanup_test_account, cleanup_test_wishlist):
    token, wishlist_id = setup_test_wishlist

    rename = "My renamed Wishlist"

    sleep(sleepTime)
    res = req.put(
        domain+f"/wishlists/{wishlist_id}",
        headers={"Authorization": f"Bearer {token}"},
        json={"name":rename}
    )
```

When using multiple cleanup fixtures, they will execute right to left, so make sure `cleanup_test_account` is the leftmost cleanup fixture. (imagine the fixtures being executed as a stack).

Adding a Test Category

To add a test category, create a new file in the tests/ directory.

- The file name must begin with **test_**.

Adding a Test

To add a test, simply add a test function to a file.

- The function name must begin with **test_**.

Important Notes

- Every file has an **email** and **password** specified at the top. This is not confidential information, this is simply the credentials used by the test account that is created for the tests. Since these credentials are used for many requests back-to-back, every computer/host running these tests should use different credentials, so there are no issues with the account already existing when two hosts run the tests at the same time.
 - TLDR: Change the **email** at the top of every page to be different than the **main branch**, but ensure all of your files contain the same credentials (maybe someone can investigate a way to only have these credentials in one place, instead of in every file).
- Please use the `time.sleep()` function before every request so you are not spamming the backend with all of your requests back-to-back. If the requests are too fast, you will start to see many tests error out.