# Testing Policy Document

## AiPi

**Regan Shen**

**Contents**

-Tools used
-Procedure for Testing
-Unit Testing
-Integration Testing
-Automated Testing

## Tools Used:

All the backend code was coded in the language of python, therefore all test (Unit and integration) was coded in Python. Python offers many tools and packages to achieve ones testing goals. To do the integration testing, the pytest tool was used, included in the pytest package is the unittest tool, the unittest tool was also used in order to complete the integration testing. To do the unit testing a few tools were needed, these are as follows, just like the integration testing pytest was needed. In addition, the mock tool from the unittest package was used in order to mock data. One of the most important tools for the unit testing was the nose package. From the nose package we were able to use tools such as assert_true, assert_equal and many more to get precise tests.

- PyTest
- unitTest
- mock
- nose

## Procedure for Testing:

There are 4 steps that need to be completed in order for the entire system to be tested completely and ready for deployment. The first testing that needs to be completed is the Unit testing. After the unit testing is completed and passed the next testing is the integration testing. After both unit and integration testing is completed and passed we move on to End-to-End testing. The final step in the testing process is to complete and pass the user acceptance criteria test. When the tester starts running the manual tests he will report in failed tests to the developer of the code, fortunately in the AiPi team, the tester and backend developer was the same person and so any issues that were discovered during manual testing was fixed almost immediately during the developing and testing phase. I will briefly discuss below how the AiPi team followed the procedure for testing.

## Unit Testing:

The first testing that needs to be done is the unit testing. Unit testing is the process of testing individual coding components or units of code. This will be done each time a new unit of code is added to the GitHub repository as well as when the tester runs test to purposely find any problems in the code. Due to the nature of our project all our methods are very dependent on third party API's in order to retrieve the necessary data to create ETF's. And so when coding and running the unit tests it was very important that we can mock data and therefore not be connected to any API or database. So the Tester used the mock tool in order to mock requests, either get or post, that would have retrieved data from the API. In this essence we were able to perform unit testing and test if the functions performed the necessary tasks when given the mock data, in some tests the mock data had errors in them in order to test each units error handling. In order to run the unit tests they

were ran in the terminal by using the "python -m nose --verbosity=2 unitTesting.py" command. This ran all tests and returned the time taken and if any failures it would return the necessary message. In a total 70 unit tests were run and all 70 tests passed.

Below is the start and the end of the unit testing output.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\regan\Documents\GitHub\TradeSim\apps\Backend> python -m nose --verbosity=2 unitTesting.py
unitTesting.test_AiFactor_code000Value ... ok
unitTesting.test_AiFactor_code001Value ... ok
unitTesting.test_AiFactor_code002Value ... ok
unitTesting.test_AiFactor_code003Value ... ok
unitTesting.test_AiFactor_code011Value ... ok
unitTesting.test_AiFactor_code012Value ... ok
unitTesting.test_AiFactor_code013Value ... ok
unitTesting.test_AiFactor_code101Value ... ok
unitTesting.test_AiFactor_code102Value ... ok
unitTesting.test_AiFactor_code103Value ... ok
unitTesting.test_AiFactor_code104Value ... ok
unitTesting.test_AiFactor_code105Value ... ok
unitTesting.test_AiFactor_code106Value ... ok
unitTesting.test_AiFactor_fitnessFunction_None ... ok
unitTesting.test_AiFactor_fitnessFunction_Value ... ok
unitTesting.test_ETF_code000 ... ok
unitTesting.test_ETF_code001_002_notRemoved ... ok
unitTesting.test_ETF_code001_002_removeStock ... ok
unitTesting.test_ETF_code003_None ... ok
unitTesting.test_ETF_code003_removeStock ... ok
unitTesting.test_ETF_code011_012_013_ForRule011_BetweenMinAndMax ... ok
unitTesting.test_ETF_code011_012_013_ForRule011_greaterThanMax ... ok
unitTesting.test_ETF_code011_012_013_ForRule011_lessThanMin ... ok
unitTesting.test_ETF_code011_012_013_ForRule012_BetweenMinAndMax ... ok
unitTesting.test_ETF_code011_012_013_ForRule012_greaterThanMax ... ok
unitTesting.test_ETF_code011_012_013_ForRule012_lessThanMin ... ok
unitTesting.test_ETF_code011_012_013_ForRule013_BetweenMinAndMax ... ok
unitTesting.test_ETF_code011_012_013_ForRule013_greaterThanMax ... ok
unitTesting.test_ETF_code011_012_013_ForRule013_lessThanMin ... ok
```

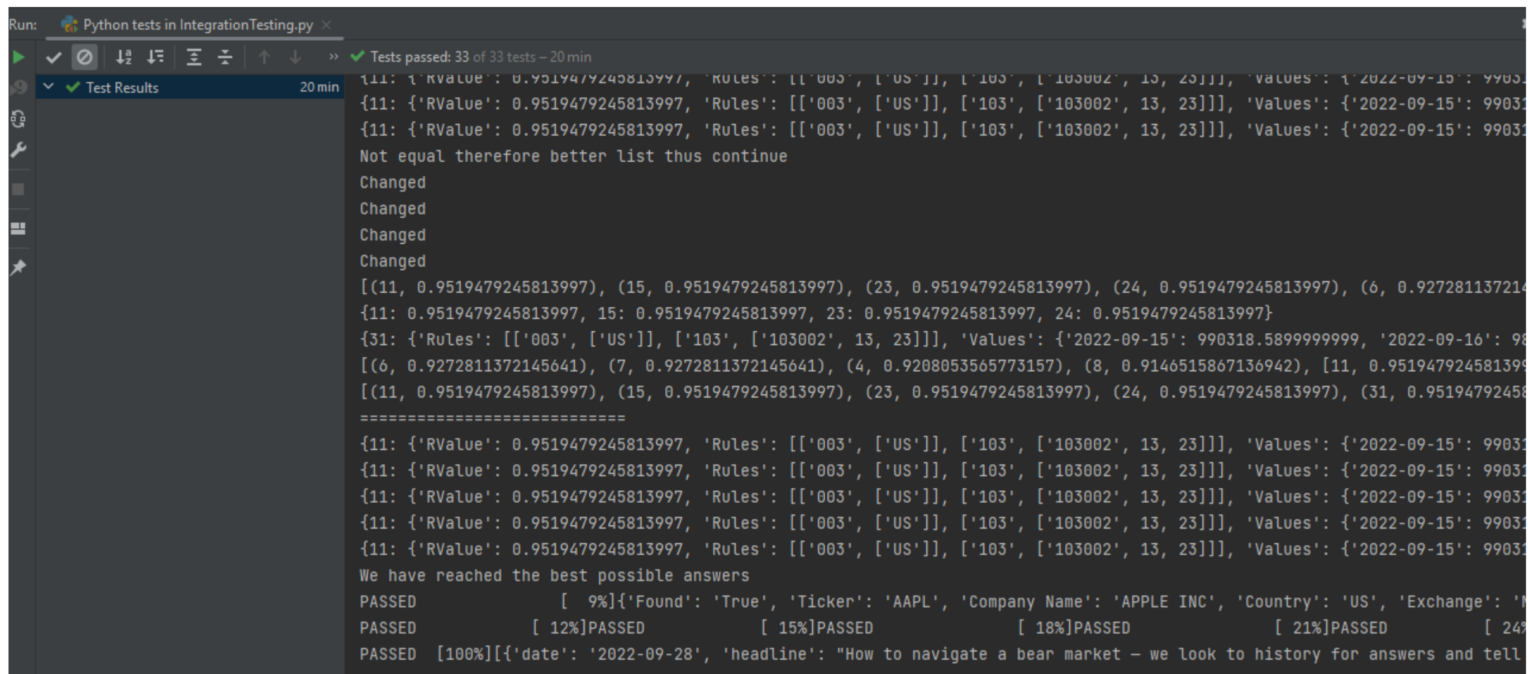
```
unitTesting.test_listAllCompanies_Value ... ok
unitTesting.test_newsInformation_returnValueNews ... ok
unitTesting.test_newsInformation_returnValueNone ... ok
unitTesting.test_stockInformationUnitTest_returnValueNone ... ok
unitTesting.test_stockInformationUnitTest_returnValueStockValue ... ok

----------------------------------------------------------------------
Ran 70 tests in 0.107s

OK
PS C:\Users\regan\Documents\GitHub\TradeSim\apps\Backend>
```

## Integration Testing:

Integration testing is only started once a few unit tests have been run and completed. This is the process of combining at different units of code that have completed their testing and then running tests with the integrated units of code. Integration testing is also the process of now testing different units while using the third party API's as well as the local databases. And thus we did not need to use the mock tool when doing integration testing, however because we used API's and did not mock any data the integration testing took very long to complete. The integration testing was completed using the built in pytest package. The integration testing was run at the end of the development and tested all objects, classes and methods in order to test if the system is reliable. Within the integration tests we also tested all methods that use a third party API directly. And fixed any error handling if the API was down.  In total we had 33 test for integration testing, this may not seem as a lot, however because of the way the back-end has been coded when running one test it will run almost all the methods and functions, we also tested for extreme inputs. Below is an image showing the passing of all 33 tests and you can see it took 20 mins to complete.



```
C:\Python39\python.exe "C:\Program Files\JetBrains\PyCharm 2022.1\plugins\python\helpers\pycharm\_jb_pytest_runner.py" --
Testing started at 21:32 ...
Launching pytest with arguments C:/Users/regan/Documents/GitHub/TradeSim/apps/Backend/IntegrationTesting.py --no-header -

============================ test session starts ============================
collecting ... collected 33 items

IntegrationTesting.py::Tests::test_AddEndDate
IntegrationTesting.py::Tests::test_AiFactor
IntegrationTesting.py::Tests::test_StockInformation
IntegrationTesting.py::Tests::test_companiesByExchange
IntegrationTesting.py::Tests::test_companiesByIndustry
IntegrationTesting.py::Tests::test_companiesRevenue
IntegrationTesting.py::Tests::test_createEtf_noRule
IntegrationTesting.py::Tests::test_createEtf_testRule000
IntegrationTesting.py::Tests::test_createEtf_testRule001
IntegrationTesting.py::Tests::test_createEtf_testRule002
IntegrationTesting.py::Tests::test_createEtf_testRule003
IntegrationTesting.py::Tests::test_createEtf_testRule011
IntegrationTesting.py::Tests::test_createEtf_testRule012
IntegrationTesting.py::Tests::test_createEtf_testRule013
IntegrationTesting.py::Tests::test_createEtf_testRule101
IntegrationTesting.py::Tests::test_createEtf_testRule102
IntegrationTesting.py::Tests::test_createEtf_testRule103
```

## Automated Testing:

When it came to the Automated Testing we knew we couldn't do any integration testing as this would take to long to run and complete the checks. So for our CICD we used the unit testing that tests almost all the functions with mock data and in this essence we can test the functionality of the code with out the need of the actual data before pushing something onto the repo. If any of the test failed during the automated testing the necessary message would be displayed showing where the error has risen and the pull request will be blocked. Below is an image that shows the Actions and results.

Workflows New workflow

All workflows
Build Client
Build python engine
Requirements

Build python engine
BuildBE.yml

Filter workflow runs

411 workflow runs — Event — Status — Branch — Actor

This workflow has a `workflow_dispatch` event trigger. Run workflow

| Run | Branch | Time |
| --- | --- | --- |
| Merge pull request #194 from ReganShen/main<br>Build python engine #440: Commit 352f9a7 pushed by MichaelViljoen | main | 1 hour ago<br>25s |
| (Backend) : Added Testing and debugging<br>Build python engine #439: Pull request #194 opened by ReganShen | ReganShen:main | 1 hour ago<br>27s |
| Update addETF.html<br>Build python engine #438: Commit 393b8a1 pushed by siphoxnkosi | main | 2 hours ago<br>29s |
| Update compare.js<br>Build python engine #437: Commit af6fb9f pushed by siphoxnkosi | main | 2 hours ago<br>30s |
| Update home.js<br>Build python engine #436: Commit 78d2061 pushed by MichaelViljoen | main | 3 hours ago<br>1m 19s |