Advanced Software Engineering
Exception Handlers: Team Assignment 8
Gabrielle Taylor: gat2118
Jackie Lin: jl4162
Leon Song: ls3233
Aaron Ong: ao2591

Continuous Integration and Coverage

Updated taskboard: https://trello.com/b/6olyGinw/task-board-2
Continuous Integration Dashboard: https://travis-ci.org/gabriellet/thepriceisright

Continuous Integration Tool

For continuous integration, we used Travis CI, which was set up directly from our github repository. We can see from the output of the build and run scripts that while our tests and coverage reports run successfully, flake8, which is a python style checker that is PEP8 compatible does not. However, all the issues it finds are cosmetic (e.g. using tabs instead of spaces, lines being too long) and will be fixed at some point in the future.

Test Coverage Tool

- In order to test using the coverage tool, install coverage.py by running '`pip install coverage`'.
- To access the coverage, navigate to coverage-report.txt in the top level directory, or look at the build status on Travis (link above).
- To manually run the coverage tool and see its report, run the following two commands:

```
coverage run --source='.' manage.py test stocks

coverage report
```

Test Coverage Output

Coverage Report Issued by Travis, our CI tool:

```
$ coverage report
Name                                    Stmts   Miss  Cover
```

```
----------------------------------------------------------------------
authentication/__init__.py                          0      0   100%
authentication/admin.py                             1      1     0%
authentication/apps.py                              3      3     0%
authentication/forms.py                             6      6     0%
authentication/migrations/__init__.py               0      0   100%
authentication/models.py                            1      1     0%
authentication/tests.py                             1      1     0%
authentication/urls.py                              3      3     0%
authentication/views.py                            11     11     0%
manage.py                                          13      6    54%
priceIsRight/__init__.py                            0      0   100%
priceIsRight/settings.py                           21      0   100%
priceIsRight/urls.py                                7      7     0%
priceIsRight/wsgi.py                                4      4     0%
stocks/__init__.py                                  0      0   100%
stocks/admin.py                                     1      0   100%
stocks/apps.py                                      3      0   100%
stocks/forms.py                                    21     21     0%
stocks/migrations/0001_initial.py                   8      0   100%
stocks/migrations/0002_auto_20161026_0353.py        5      0   100%
stocks/migrations/0003_auto_20161109_1143.py        5      0   100%
stocks/migrations/0004_parentorder_progress.py      5      0   100%
stocks/migrations/0005_auto_20161130_2058.py        5      0   100%
stocks/migrations/__init__.py                       0      0   100%
stocks/models.py                                  158    105    34%
stocks/tests.py                                    38     17    55%
stocks/urls.py                                      3      3     0%
stocks/views.py                                    74     74     0%
----------------------------------------------------------------------
TOTAL                                             397    263    34%
```

Brief writeup on coverage report

UI Testing was the most significant challenge in extending our test coverage. A large portion of our code is dedicated to updating the progress of a trade as it executes throughout the day, and displaying its progress in the UI (e.g. progress bar changes red when an order fails, blue when cancelled.) While some automated UI testing tools like selenium can simulate user behavior, its challenging to implement a test to verify that “the progress bar turns red on failure” or an error popup displays when a user gives bad input.

A related issue that made testing even harder was that our code base consists of several languages — Python, Ajax, Javascript, html etc. The django framework provides robust testing tools to test backend models, but testing interactions between back and front end is another issue (e.g. if the status of a ParentOrder model changes, verify that Ajax updates the UI to reflect the change.)

Finally, our trade algorithm depends on tracking several local variables e.g. counter for number of attempts to sell at $x, or remainder quantity left to sell. These objects are defined within the trade() method, and are not associated with any backend models. This means that we cannot create mock objects to track the data as it runs through the algorithm, so those variables cannot be covered by unit tests.

System Tests

1. Users can track the behavior of an order throughout the day
2. Log into the system and place several orders
3. Navigate to the home page, the UI should display all active and completed orders

4. Additionally, each Order ID should be clickable, and will lead you to a detailed page of each Parent Order
5. Verify that all child orders associated with each parent order are displayed, even if failed or not
6. To do this, log into the database shell (open terminal and cd into the app folder, then type ‘python manage.py shell’)
7. Issue the command: ‘print ChildOrder.objects.filter(ParentOrder__pk=<id>)’ and check that all ChildOrders are displayed in the UI
8. Users must be able to pause, cancel, and resume trade orders.
    a. Log into the system
9. Click “Place Order” on top right of UI, to navigate to order page
10. Key in an order (anything integer between 1 and overflow is supported)
11. Navigate to home page, all active orders should be listed and their statuses displayed
12. Click on an order, the UI should display an option to Pause or Cancel
13. Try clicking Pause: The UI should now give an option to resume an order
14. Verify that the order stops selling, the UI should show that no new ChildOrders are created, not even failed ones
15. Try resuming, the UI should now show that our server is trying to execute more trades and create new ChildOrders
16. Repeat the same for Cancel Order, except that there should be no option to Resume a cancelled order.
17. Order must weight each trade attempt according to how long the market is open (i.e. if the market is open for 8 hours, take our time to sell. If market only open for 3 more hours, algorithm must sell more aggressively).
18. Log into the system when the exchange just opens
19. Place an order
20. Verify initial behavior is to try selling ChildOrders which are 10% quantity of the parent.
21. Pause the order
22. Resume the order at a later time (try various times, e.g. 2 hours later, 3 hours later, 6 hours later)
23. Each time, the algorithm should try to sell more aggressively and split ChilderOrders into larger amounts, or sell ChildOrders more frequently.

Showcase Demo Script

Features to be presented

1. Show how to place an order and demonstrate it flowing through the pipeline
    a. Navigate to “place order” page and place an order
    b. Order input handles erroneous input (e.g. non-numeric input, non-positive input, non-integer input, overflow)
    c. Navigate back to home page, show off all progressing/completed orders.
    d. Click on a single order. Show off that we create all these child orders

2. Talk about the behavior of the trading algorithm e.g. when fail to sell then what happens, when success then what happens.
    a. The trading algorithm breaks each order up into smaller orders which are spread out evenly over a time period in order to lessen its market impact. (TWAP)
    b. When an order is entered, the trading algorithm first tries to execute a smaller order at 10% of the parent order’s volume.
    c. When an order fails multiple times, the algorithm will decrease the order size.
    d. When an order succeeds multiple times, the algorithm increases the order size.
    e. Successful orders are stored in the db.
    f. Orders can be paused/resumed/cancelled.
3. Show off UI features
    a. Order progress bar updates asynchronously without page refresh through AJAX
    b. Order detail page updates asynchronously without page refresh through AJAX
    c. Can pause/resume and cancel orders

What we plan to say (in prose)

1. Order placing
    a. “So here is the order placing page”
    b. “We display friendly UI messages for bad input e.g. integer overflow, negative numbers”

    c. “We issue a popup to confirm a user’s order, just in case she gives the wrong input”
    d. “Issuing an order leads us back to the index page, where a user can observe all his current/past orders”
2. Behavior of algorithm
    a. After an order is placed, the trading algorithm will break the order into smaller chunks which are sold over the course of the day. The algorithm first attempts to execute a child order which is 10% of the parent order by volume. Observe that as the child orders fails to execute repeatedly, the algorithm will adjust to the market’s supply by attempting to sell smaller and smaller child orders until it succeeds. Repeated successes also signal to the algorithm that the market’s supply is sufficient, and it will start attempting to sell larger and larger child orders.
3. Cool UI features
    a. The user can view the status of an order through asynchronously updating progress bars and transaction tables.
    b. The user can pause and resume or even cancel an ongoing order.

Order/flow of presentation

1. Hi everyone, our goal was to implement an app that executes trading orders according to a T.W.A.P algorithm → demonstrate app
2. Demonstrate placing several orders → (e.g 10000, 200, **3000**, 400)
    a. Does not accept bad input: overflow, negative orders, 0.01
3. Navigate to home page and show that our system is tracking multiple orders
    a. Table views
4. Show that progress bars are updating asynchronously for each order
5. Click on an order and show each child order being generated
    a. This is where Aaron will talk about how the trade algorithm behaves
    b. E.g. when a child order fails, the system will try to downsize the next child order
    c. Also point out that progress bar and transaction table updates asynchronously
6. Demo Pause/Resume and Cancel features
    a. Show that child orders stop being created when we pause
    b. And that they continue being created when we resume

    c. Show that order changes status when canceled
7. Show that each user profile can only view their own orders and that orders still continue to sell even when the user is logged out.
    a. Log into another account to show that it sees a different list of orders and that orders placed by this account do