**Continuous Integration:** The practice of integrating our project often (merging through github), coupled with automated testing.

**Automated Testing Tools**

**Backend**

* [Django has many testing related tools built-in](https://docs.djangoproject.com/en/1.10/topics/testing/tools/), which are ran through bash scripts
  + **Database Testing**
    - [Write code to test the database by importing TestCase class](https://docs.djangoproject.com/en/1.10/topics/testing/overview/)
    - This class is also used to create dummy data
    - Tests are all done on a dummy database
  + **Rest API Testing**
    - Write code to simulate a client that makes http requests by importing Client class
  + **Unit Testing**

Django’s unit tests use a Python standard library module: [**unit-test**](https://docs.python.org/3/library/unittest.html#module-unittest)**.** This module defines tests using a class-based approach.

Creating and Running tests in Django:<https://docs.djangoproject.com/en/1.10/topics/testing/overview/>

* + **Functional Testing** 
    - Use Django’s test client to establish that the correct template is being rendered and that the template is passed the correct context data.
    - Use in-browser frameworks like [Selenium](http://seleniumhq.org/) to test *rendered* HTML and the *behavior* of Web pages, namely JavaScript functionality. Django also provides special support for those frameworks; see the section on [LiveServerTestCase](https://docs.djangoproject.com/en/1.10/topics/testing/tools/#django.test.LiveServerTestCase) for more details.
    - Django Testing tools<https://docs.djangoproject.com/en/1.10/topics/testing/tools/>
    - Selenium

<http://www.seleniumhq.org/>

* + **Code Coverage**
    - Django’s Integration with **Coverage.py**: [**http://coverage.readthedocs.io/en/latest/**](http://coverage.readthedocs.io/en/latest/)

Coverage.py is a tool for measuring code coverage of Python programs. It monitors your program, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not. Coverage measurement is typically used to gauge the effectiveness of tests. It can show which parts of your code are being exercised by tests, and which are not.

The latest version is coverage.py 4.2, released July 26th 2016. It is supported on:

* Python versions 2.6, 2.7, 3.3, 3.4, 3.5, and 3.6.
* PyPy 4.0 and 5.1.
* PyPy3 2.4 and 5.2
  + **Integration Testing**

**This is based on question posted to StackOverflow; I will continue to research this area.**

* + - **I would recommend to use some BDD framework (Behave, Lettuce) and run BDD tests from a CI server (TravisCI or Jenkins) against external server (staging environment for example). So, the process could be:**
* Push **changes to GitHub**
* GitHub **launches build on CI server**
* CI server runs unit tests
* CI server deploys to integration environment (or staging, if you don't have integration)
* CI server runs integration end to end tests against the new deployed code
* If all succeeds, this build will be promoted to "can be deploy to production" or something like that
  + **Third-party packages**
    - Use the following packages and libraries to assist with writing and running your test suite:
* [django-webtest](https://pypi.python.org/pypi/django-webtest): makes it much easier to write functional tests and assertions that match the end user’s experience. Couple these tests with Selenium tests for full coverage on templates and views.
* [coverage](http://nedbatchelder.com/code/coverage/): is used for measuring the effectiveness of tests, showing the percentage of your codebase covered by tests. If you are just starting to set up unit tests, coverage can help offer suggestions on what should be tested. Coverage can also be used to turn testing into a game: I try to increase the percent of code covered by tests each day, for example.
* [django-discover-runner](https://github.com/jezdez/django-discover-runner): helps locate tests if you organize them in a different way (e.g, outside of *tests.py*). So if you organize your tests into separate folders, like in the example above, you can use discover-runner to locate the tests.
* [factory\_boy](https://github.com/rbarrois/factory_boy), [model\_mommy](https://github.com/vandersonmota/model_mommy), and [mock](https://pypi.python.org/pypi/mock): all are used in place of [fixtures](https://docs.djangoproject.com/en/dev/howto/initial-data/) or the ORM for populating needed data for testing. Both fixtures and the ORM can be slow and need to be updated whenever your model changes.

* + **Additional information**
    - **A Guide to Testing in Django:**<http://toastdriven.com/blog/2011/apr/10/guide-to-testing-in-django/>
    - **Integration Testing With pyVows and Django:** <https://realpython.com/blog/python/integration-testing-with-pyvows-and-django/>
    - [**How A Django Developer Can Write Speedy Unit Tests, Pt. 1: The Basics** http://blog.celerity.com/how-to-write-speedy-unit-tests-in-django-part-1-the-basics](http://blog.celerity.com/how-to-write-speedy-unit-tests-in-django-part-1-the-basics)
    - **Best practices**

<https://realpython.com/blog/python/testing-in-django-part-1-best-practices-and-examples/>

**Frontend**

* **Unit Testing**
* Jest
  + used for unit testing. The most recent release plays nicely with react Native
  + <http://facebook.github.io/jest/docs/tutorial-react-native.html>\
* Integration/feature Testing
* Appium
  + Same foundation as selenium
  + Seems to be popular as a testing tool for React native apps.
  + Compatible with both iOS and android