MAY 2, 2017

# DJANGO TESTING DOCUMENT

CAPSTONE: AWESOME USAGE MONITOR

JESSICA BLASCH
PORTLAND STATE UNIVERSIITY
Portland, OR, U.S.A.

Version 1.0

Version History

| Version # | Revision Date | Responsible Party | Reason |
|---|---|---|---|
| 1.0 | 02 May, 2017 | Jessica Blasch | Initial creation of source document |

## Table of Contents

# Introduction

## Read Me

I have written this document assuming the reader has little or no prior experience with Django. I, for one, am a beginner at using the Python language having written only one program in it prior. I had no prior experience using Django (or anything like it), before the start of this project, and absolutely zero experience writing and running tests.

If you, like myself, have little or no prior experience with Python, Django, and/or testing, it can be extremely daunting how to even begin. Django documentation has tutorials to get you acquainted with the Django environment and they have a tutorial on testing. I think the tutorials are okay in so much that they let you touch Django and get some preliminary fighting out of the way, but that's about all you do in them; they have you robot through instructions without sufficient explanation about the set up and code they have you copy/pasting.

All that being said, I highly recommend getting the Django documentation tutorials out of the way and moving on to bigger and better things. YouTube has a lot of Django testing tutorials that really walk you through the process. They, of course, all use very trivial examples, but, I think, are more helpful. One tutorial in particular that I am fond of is:

The Django Test Driven Development Cookbook: https://www.youtube.com/watch?v=41ek3VNx_6Q

This one really helped me "get out the gate". I went ahead and put the tutorial into document format transcribing and adding images of what the host said and did as I worked through it. That document will accompany this document as a resource for anyone working on this project in the future. That way, you can go through the tutorial and later reference the accompanying document instead of scrolling through video trying to find something. As stated before, the examples used in many tutorials are trivial, so don't expect that they will just hand you answers, but they can set you on the right path.

I have tried to differentiate between "instructions" and "my side commentary" by using grayed text as commentary. You may see me use "django-admin" and "django-admin.py" for different instances. The reason for this is that *they both work*. Last, but not least, when a new test is created run it, edit as needed, and run it again until it passes. This will save you time trying to track down when and where things went south.

6/13/17: The headings starting with "Testing Views", at this time, are incomplete. To date, the only testable file available in the git repository is the models.py file. I believe views, etc. are being created and will be uploaded soon, but not in time to test. I have one suggestion under "Testing Views". The rest, I am commenting, but would highly recommend viewing the tutorial I cited and the source document I created for it as it relates to the additional headings.

6/15/17: Many files in the aum project have been updated. I have added the settings.py, models.py, and managers.py files to the test environment. Tests still pass. Project hand-off to the sponsor happens today.

## Integration of Product and Test Environments

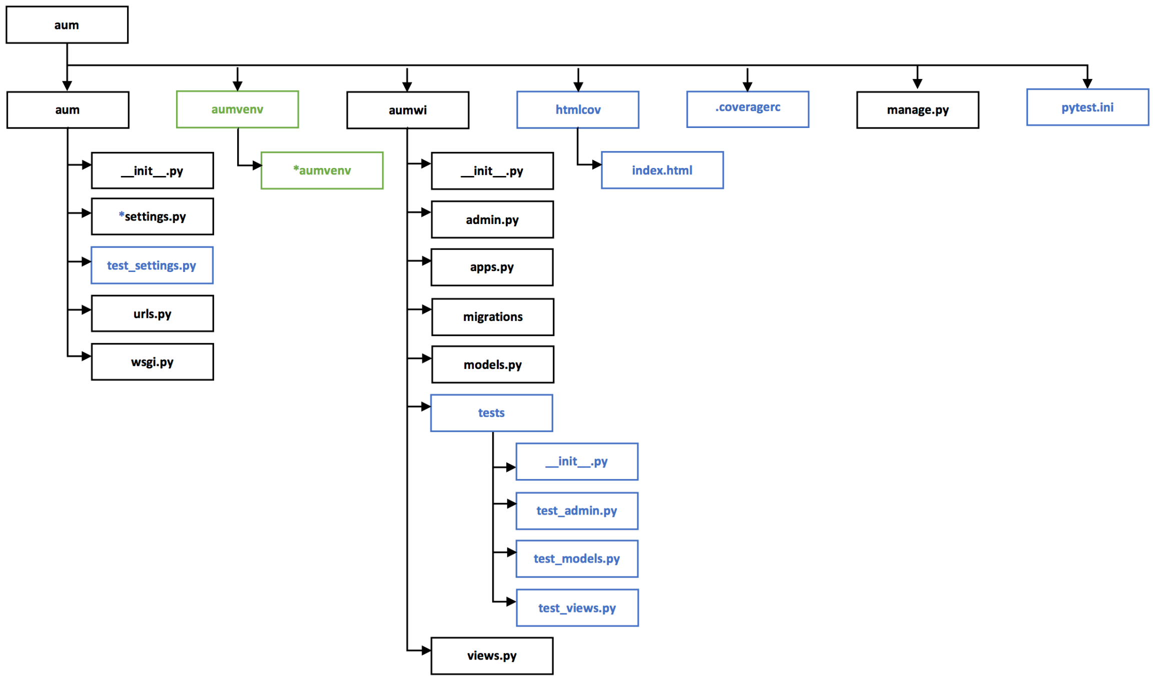
For the initial project, there were multiple "teams" within our team; one person did circuit design and the programming of it, one person did the framework, and one person did the testing. This document covers what was done in the test environment. Integrating the test environment into the product package should be able to be accomplished by familiarizing oneself with this document's procedures and then applying them inside the virtual environment for the AUM product. Some steps like creating a virtual environment and adding the app to the settings file may be skipped, since they will have already been done in the development of the AUM product.

## OS and Considerations

I am using a 2016 MacBook Pro with macOS Sierra version 10.12.4. Please note, the documentation for creating a virtual environment is what I found to work on my machine. It will be different, if you are using a machine that does not use OS X. You can do a general search engine query on how to set up a virtual environment, but I found that Django Girls has a pretty comprehensive list of instructions for several operating systems: https://tutorial.djangogirls.org/en/django_installation/ . Everything else should be the same.

## General Overview of Test Setup

### File and Folder Map



### File and Folder: Functions and Relations

**aum** (top level)**:** This is the root folder for your project. Django documentation will instruct you to first create a folder to house your Django project. There are a couple ways to determine the final folder set up.

1. Use the folder you created for the project as the root folder and create set up within it.

   From the terminal, cd into the folder and do an admin call to start the project within it.
   e.g. $ django-admin startproject aum .

   Notes: *Preferred method.*
   The period " ." is crucial in the call. Using this method, the folder you created to house your project becomes the root folder and houses your settings and other folders/files directly inside it. This is the method I will use in the documentation.

2. Create a root folder and set up within the folder you created to house your project.

   From the terminal, cd into the folder and do an admin call to start the project within it.
   e.g. $ django-admin.py startproject aum

   Notes: *I don't care for this method.*
   The reason being that now there is yet another folder (not pictured) encompassing the project root and settings folders. It seems an unnecessary complication.

When working in the terminal, this is, generally speaking, the only folder you need to cd in to in order to work on the project. From this folder, you can create/activate your virtual environment and make changes to all other files and folders inside this folder without having to cd into them.

Notes: Name does not matter, but must be the name of the project initially, if you use method 2 above. Otherwise, it's whatever you call it and is changeable.

**aum** (2nd level): This is the settings folder for the project that is automatically created when you start your Django project.

Notes: *You cannot change the name*, because it's in the settings. It's crucial this folder name matches between all people working on the project. It will be generated for you at the time you start your Django project.

**aumvenv:** This is the virtual environment folder that you create for the project. It's not necessary, but it is *highly recommended*, because it allows you to create settings for the project (e.g. Python and Django versions) and isolate it so it will not affect other projects you may be working on.

Notes: Name does not matter. It's whatever you call it, but it is static after.

**aumwi**: This is the app folder for the project that you create after you start your Django project.
e.g. $ django-admin.py startapp aumwi

Notes: It's crucial this folder name matches the name being used by all people working on the project.

**manage.py**: The manage.py file is automatically created in each Django project. manage.py does the same thing django-admin does with a few extras (Django's command-line utility for administrative tasks).

- - - The following items in the diagram denoted in **blue** have to do with the testing setup. - - -

Please note: Some items below are reliant upon doing pip installs in order to function. That is covered later in this document.

***aum/settings.py**: The name of your app needs to be added to this file.

**test_settings.py**: You need to create this file. This is where the settings for the test database and testing email settings live.

**tests**: You need to create this folder and its contents. This folder replaces the "**tests.py**" file that Django automatically creates, when you start a new app. This is where your tests live. You need to create an "**__init__.py**" file for it. Models, views, forms, etc. and any other files you want to test will have their own test files you will need to also create within this folder.

**htmlcov**: This folder is created for you when you start running your tests. It houses a callable index file to show your test coverage.

**.coveragerc**: You need to create this file. This is a hidden file. It allows you to omit certain things from test coverage reports in order to keep them clean of information you don't care about (e.g. items that are not testing related).

**pytest.ini**: You need to create this file. It tells **pytest** where your test settings live, allows you to stop migrations during testing and set the format the output of your coverage report.

## Code Editor

I have Canopy and Anaconda installed on my machine as well as Python 2.6, 2.7, and 3.6. However, Canopy is geared to use and interpret Python 2 (legacy). For this project, I will be using Atom, but here are a few different options.

PyCharm: https://www.jetbrains.com/pycharm/download/#section=mac ← Extremely powerful. Less suitable for a beginner.

Gedit: https://wiki.gnome.org/Apps/Gedit#Download ← Open-source. Available for all operating systems.

Sublime Text 3: https://www.sublimetext.com/3 ← Free evaluation period. For all operating systems.

Atom: https://atom.io/ ← Open-source. Created by GitHub. Available for Windows, OS X, and Linux.

## Set up Virtual Environment

The following assumes setup is done on a Mac under the home directory e.g. /Users/Name (where "Name" is the name of your login). If you are using a different machine, Django Girls has a good tutorial that covers multiple systems and potential problems under their Django installation page: https://tutorial.djangogirls.org/en/django_installation/

Further reading
Virtual environments: https://docs.python.org/3/tutorial/venv.html
http://python-guide-pt-br.readthedocs.io/en/latest/dev/virtualenvs/
Virtualenvwrapper: https://virtualenvwrapper.readthedocs.io/en/latest/

1. Install virtualenv and virtualenvwrapper

   Command line:
   ```
   $ pip3 install virtualenv
   $ pip3 install virtualenvwrapper
   ```
   ← I was unable to get virtual environment wrapper to work.

2. Set up a folder for the Django project (Do not call it "Django") and create the virtual environment

   Command line:
   ```
   $ cd ~
   $ mkdir test_aum
   $ cd test_aum
   $ python3 -m venv aumvenv
   ```
   ← You may call it whatever you like. I used "aumvenv".

3. Start virtual environment

   Command line:
   ```
   $ source aumvenv/bin/activate
   ```

   You will know the virtual environment has been activated when the beginning of the command line displays (aumenv) at the beginning.

4. Make sure you have the latest version of pip

   Command line:
   ```
   (aumvenv) ~$ pip install --upgrade pip
   ```

5. Install Django

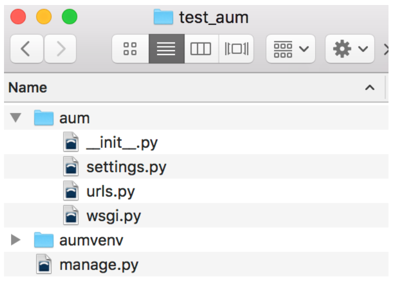
   Command line:
   ```
   (aumvenv) ~$ pip install django~=1.11.2
   ```
   ← This is the latest version used for initial testing. Use the most current version by omitting "~=1.11.2", but make sure the tests still run.

## Project Test Environment Setup

1. Start new Django project

Command line:

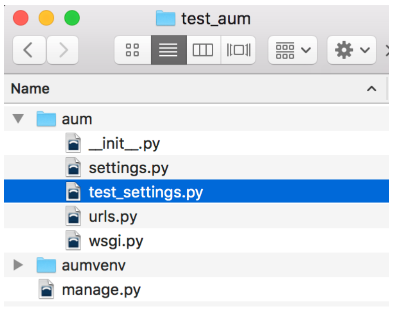
```
(aumvenv) ~$ django-admin startproject aum .
```



2. Create a "**test_settings.py**" file in the settings folder test_aum/aum

Command line:

```
(aumvenv) ~$ touch aum/test_settings.py
```

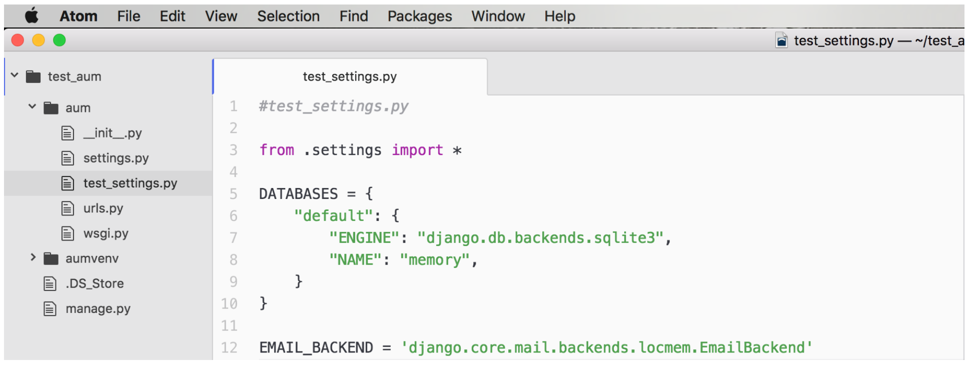


Add the following code:

```
#test_settings.py

from .settings import *

DATABASES = {
  "default": {
    "ENGINE": "django.db.backends.sqlite3",
    "NAME": "memory",
  }
}

EMAIL_BACKEND = 'django.core.mail.backends.locmem.EmailBackend'
```

Instead of using a Postgres or MySQL db, we are using an "in memory" sqlite db which makes tests fast. Every test will destroy the db and create a new one. If you do that with a db that needs I/O operations on a hard drive, it's going to be slow. In-memory is a good way around that.

Email backend is set to local memory, so that you don't send real emails while running tests.



```
#test_settings.py

from .settings import *

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": "memory",
    }
}

EMAIL_BACKEND = 'django.core.mail.backends.locmem.EmailBackend'
```

3. Install pytest & plugins and create “**pytest.ini**” in the root folder capstone_aum

Install a few Python libraries:

Command line:

```
(aumvenv) ~$ pip install pytest
(aumvenv) ~$ pip install pytest-django
(aumvenv) ~$ pip install pdbpp
(aumvenv) ~$ pip install pytest-cov
(aumvenv) ~$ deactivate
(aumvenv) ~$ source aumvenv/bin/activate
```

pytest has lots of plugins. You can use them by just installing them.
https://pypi.python.org/pypi/pytest/3.0.7

pytest-django is a plugin in pytest
https://github.com/pytest-dev/pytest-django

pdb++ is a drop-in replacement for pdb. It’s for setting breakpoints into your tests and then you’ll be able to use the pdb++ debugger which is nicer than the original, because it has colors and code completion.
https://pypi.python.org/pypi/pdbpp/

pytest coverage helps you generate a coverage report. When we run our tests, it generates a bunch of html files. You can see all your code files and the percentage of lines in the file that have been hit by your tests.
https://pypi.python.org/pypi/pytest-cov/2.4.0

100% test coverage doesn’t mean you have good tests. However, without 100% test coverage it’s easy to get in a situation where sloppy fixes add up over time and your test coverage goes down.

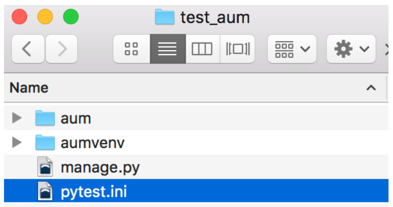
* If you get an error after installing pdbpp install it again. I didn’t end up using it, but it could prove very helpful later.

After you install all the libraries, you need to deactivate your virtual environment and then reactivate it (as shown). Otherwise, it will throw weird errors.

This file needs to go next to the manage.py file:

Command line:

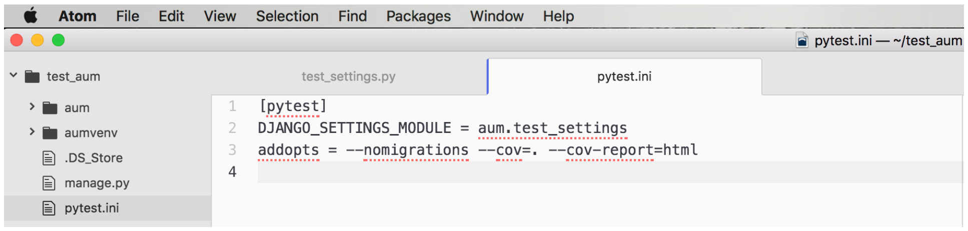
```
(aumvenv) ~$ touch pytest.ini
```



Add the following code:

```
[pytest]
DJANGO_SETTINGS_MODULE = aum.test_settings
addopts = --nomigrations --cov=. --cov-report=html
```

← Tell it where the test settings file is
← Add more arguments to the command argument

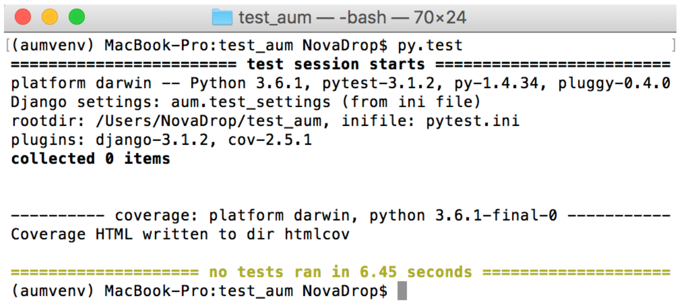


```
[pytest]
DJANGO_SETTINGS_MODULE = aum.test_settings
addopts = --nomigrations --cov=. --cov-report=html
```

Now try running it:

Command line:

```
(aumvenv) ~$ py.test
```



```
(aumvenv) MacBook-Pro:test_aum NovaDrop$ py.test
========================= test session starts =========================
platform darwin -- Python 3.6.1, pytest-3.1.2, py-1.4.34, pluggy-0.4.0
Django settings: aum.test_settings (from ini file)
rootdir: /Users/NovaDrop/test_aum, inifile: pytest.ini
plugins: django-3.1.2, cov-2.5.1
collected 0 items


---------- coverage: platform darwin, python 3.6.1-final-0 -----------
Coverage HTML written to dir htmlcov

==================== no tests ran in 6.45 seconds ====================
(aumvenv) MacBook-Pro:test_aum NovaDrop$
```

4. Create a “**.coveragerc**” file

Command line:

```
(aumvenv) ~$ touch .coveragerc
```

**This is a hidden file**, so it’s easy to forget it exists. It will not show up in normal file searches. However, without it, the coverage report will look messed up. The html file will be a huge list where half of the files don’t even interest you. Because you don’t test files added to this list it means the coverage will be 0%, which will lower your total coverage and you want your total coverage to be 100%. So, whenever you have a file you don’t care about, you need to add it to this list so that it gets removed from the coverage report.

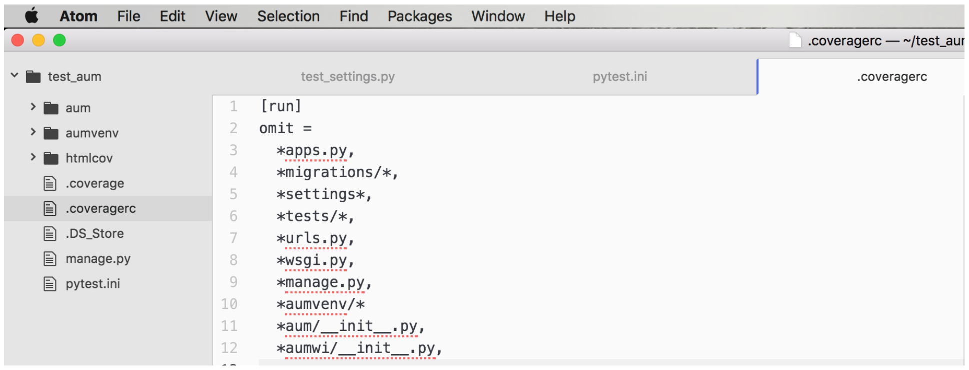
Add the following code:

```
[run]
omit =
  *apps.py,
  *migrations/*,
  *settings*,
  *tests/*,
  *urls.py,
  *wsgi.py,
  *manage.py,
  *aumvenv/*
  *aum/__init__.py,
  *aumwi/__init__.py,
```

← What this code means is that in certain files you don’t care about the test coverage.

e.g.

Don’t test migrations, because they’re auto-generated. Don’t test the test files, because that’s over kill. Don’t test virtual environment files, because it’s not your actual project, etc.



```
[run]
omit =
  *apps.py,
  *migrations/*,
  *settings*,
  *tests/*,
  *urls.py,
  *wsgi.py,
  *manage.py,
  *aumvenv/*
  *aum/__init__.py,
  *aumwi/__init__.py,
```

5. Install mixer

Command line:

```
(aumvenv) ~$ pip install mixer
```

**mixer** is a tool for helping you create test fixtures.

e.g.

Say you have a view that shows your user profile and you want to test it. That view can only be called by giving the primary key of that user. There needs to be something in the database so this view can even be called, because the first thing this view is going to try to do is fetch that model from the database. If it’s empty it will throw a 404 error.

If the profile model has 20 fields, mixer will put random values into each of these fields. Your tests may pass and then one day fail because mixer put something into your tests you never thought about. So, it can help you find test cases you never thought about.

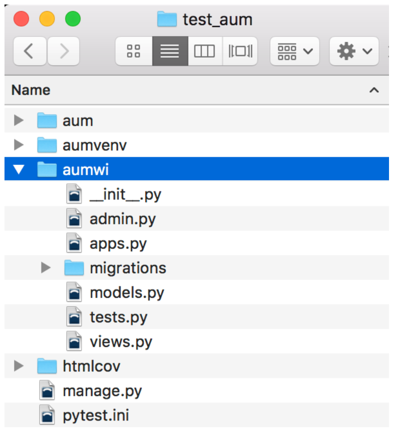
6. Create the folder set up for writing tests
   - Remove the “**tests.py**” file and create a “**tests**” folder instead
   - Each Django app will have a “**tests**” folder
   - Each code file will have a tests file e.g “**test_models.py**”

Command line:

| | |
|---|---|
| `(aumvenv) ~$ django-admin startapp aumwi` | ← Create “**aumwi**” app in the root folder |
| `(aumvenv) ~$ rm aumwi/tests.py` | ← Delete the “tests.py” file in the “**aumwi**” folder |
| `(aumvenv) ~$ mkdir aumwi/tests` | ← Create “**tests**” folder in the “**aumwi**” folder |
| `(aumvenv) ~$ touch aumwi/tests/__init__.py` | ← Create an ‘**__init__.py**’ “**aumwi/tests**” folder |

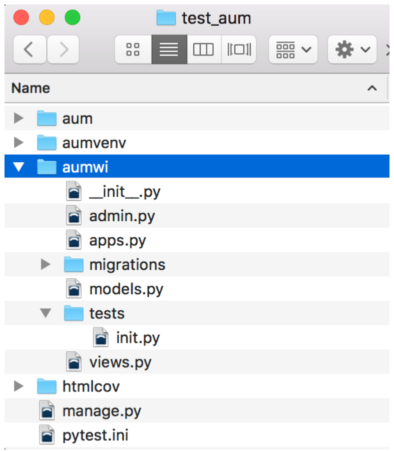
To ensure we’re on the same page, here is a view of the aumwi folder before and after alterations:

aumwi folder before alterations:



```
test_aum
Name
▶ aum
▶ aumvenv
▼ aumwi
    __init__.py
    admin.py
    apps.py
  ▶ migrations
    models.py
    tests.py
    views.py
▶ htmlcov
  manage.py
  pytest.ini
```

aumwi folder after alterations:

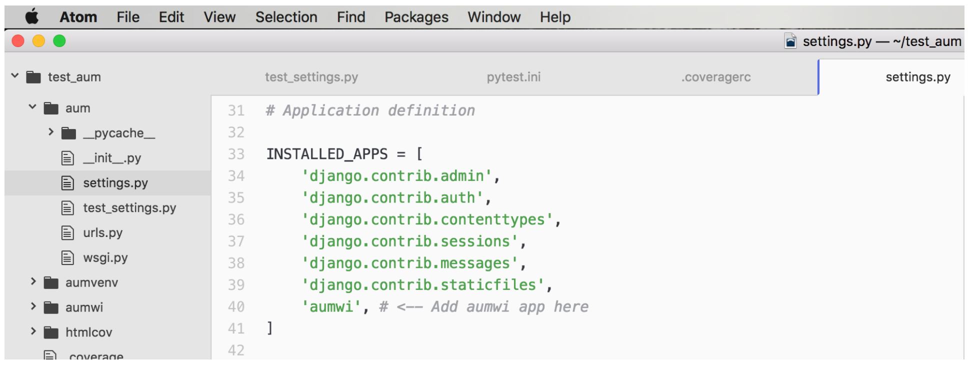


```
test_aum
Name
▶ aum
▶ aumvenv
▼ aumwi
    __init__.py
    admin.py
    apps.py
  ▶ migrations
    models.py
  ▼ tests
      init.py
    views.py
▶ htmlcov
  manage.py
  pytest.ini
```

7. Add the “**aumwi**” app to the aum/settings.py file

   Import the settings.py file from Capstone-LID_Tracking/aumwi/models.py (last version available: 5/15/17)

   → This should be a simple cut/paste over the code auto-generated in the test environment settings.py file.
   → If unable to do this, at a minimum, add the “aumwi” app manually like below:



```
test_aum
  aum
    __pycache__
    __init__.py
    settings.py
    test_settings.py
    urls.py
    wsgi.py
  aumvenv
  aumwi
  htmlcov
  coverage
```

```python
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'aumwi', # <-- Add aumwi app here
]
```

At this point, we are set up to begin testing

**!!! Important !!!**

py.test will:
- Find all files called “**test_.py**”
- Execute all functions called “**test_()***” on all classes that start with “**Test***”

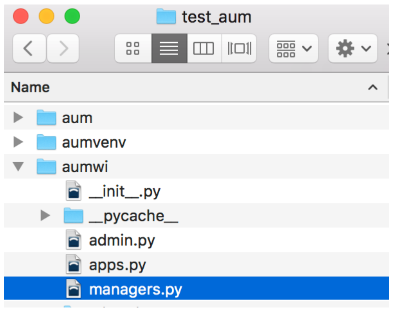
## Testing Models

1. Import models.py from Capstone-LID_Tracking/aumwi/models.py (last version available: 5/15/17)

   This is a simple cut/paste.

2. Create “**managers.py**” file and import managers from Capstone-LID_Tracking/aumwi/managers.py

   Command line:
   ```
   (aumvenv) ~$ touch aumwi/managers.py
   ```



   Then cut/paste the managers.py file from Capstone-LID_Tracking/aumwi/managers.py into the file just created.

3. Create “**test_models.py**” file in aumwi/tests folder

   Command line:
   ```
   (aumvenv) ~$ touch aumwi/tests/test_models.py
   ```

   Add the following code:

```
# test_models.py

import pytest  # This is the test interpreter. Must install pytest to use.
pytestmark = pytest.mark.django_db  # Use pytest marks to tell pytest-django your test needs database access.
                        # Must install pytest-django to use.
                        # http://pytest-django.readthedocs.io/en/latest/database.html
from aumwi.models import User, ModuleList, AuthUser, UsageLog # mixer doesn't need this. For use with e.g. users = User.objects.all()
from django.test import TestCase # Allows use of assertEqual
from mixer.backend.django import mixer # Test fixture creator. Must install mixer.
from django.db import IntegrityError, transaction

class TestUser(TestCase):
  def test_init(self):
    obj = mixer.blend('aumwi.User')
    users = User.objects.all()
    self.assertEqual(len(users), 1), 'Should create a User instance.'

  def test_unique_fields(self): #Exception raised when the relational integrity of the database is affected
    with self.assertRaises(IntegrityError):
      # roll back transaction to a clean state before letting the IntegrityError bubble up
      with transaction.atomic():
        # This code executes inside a transaction.
        user1 = mixer.blend('aumwi.User', id = '5dad0428-1da1-4d6d-8cd2-ca88ad23a6f6')
        user2 = User(
        id = '5dad0428-1da1-4d6d-8cd2-ca88ad23a6f6',
        username = 'tinyrobotarmy',
        full_name = 'Your Mom',
        email = 'dontpantsme@gmail.com',
        rfid = 9223372036854775807,
        )
        user2.save(force_insert = True), 'Should raise IntegrityError: User.id'
    with self.assertRaises(IntegrityError):
      with transaction.atomic():
        user1 = mixer.blend('aumwi.User', username = 'SilverwareGolem')
        user2 = mixer.blend('aumwi.User', username = 'SilverwareGolem'),'Should raise IntegrityError: User.username'
    with self.assertRaises(IntegrityError):
      with transaction.atomic():
        user1 = mixer.blend('aumwi.User', email = 'dontpantsme@gmail.com')
```

```
            user2 = mixer.blend('aumwi.User', email = 'dontpantsme@gmail.com'),'Should raise IntegrityError: User.email'
        with self.assertRaises(IntegrityError):
            with transaction.atomic():
                user1 = mixer.blend('aumwi.User', rfid = '922337203685477580')
                user2 = mixer.blend('aumwi.User', rfid = '922337203685477580'), 'Should raise IntegrityError: User.rfid'

    def test_empty_fields(self): #Exception raised when the relational integrity of the database is affected
        '''
        I am not testing User.id here. This test does not work for the id,
        I think because the system is designed to assign one at the time of saving.
        '''
        with self.assertRaises(IntegrityError):
            # roll back transaction to a clean state before letting the IntegrityError bubble up
            with transaction.atomic():
                # This code executes inside a transaction.
                obj = mixer.blend('aumwi.User', username = None),'Should raise IntegrityError: User.username'
        with self.assertRaises(IntegrityError):
            with transaction.atomic():
                obj = mixer.blend('aumwi.User', email = None),'Should raise IntegrityError: User.email'
        with self.assertRaises(IntegrityError):
            with transaction.atomic():
                obj = mixer.blend('aumwi.User', rfid = None), 'Should raise IntegrityError: User.rfid'

class TestModuleList(TestCase):
    def test_init(self):
        obj = mixer.blend('aumwi.ModuleList')
        modules = ModuleList.objects.all()
        self.assertEqual(len(modules), 1), 'Should create a ModuleList instance.'

    def test_unique_fields(self): #Exception raised when the relational integrity of the database is affected
        with self.assertRaises(IntegrityError):
            # roll back transaction to a clean state before letting the IntegrityError bubble up
            with transaction.atomic():
                # This code executes inside a transaction.
                mod1 = mixer.blend('aumwi.ModuleList', id = 'b099ce7e-fa35-4a1c-a418-c0be75c677f9' )
                mod2 = ModuleList(
                id = 'b099ce7e-fa35-4a1c-a418-c0be75c677f9',
                smid = 9223372036854775800,
                name = 'Soldering Iron: Hunka Hunka Burning Metal'
                )
                mod2.save(force_insert = True), 'Should raise IntegrityError: ModuleList.id'

    def test_empty_fields(self): #Exception raised when the relational integrity of the database is affected
        '''
        I am not testing Module.id here. This test does not work for the id,
        I think because the system is designed to assign one at the time of saving.
        '''
        with self.assertRaises(IntegrityError):
            # roll back transaction to a clean state before letting the IntegrityError bubble up
            with transaction.atomic():
                # This code executes inside a transaction.
                obj = mixer.blend('aumwi.ModuleList', smid = None ), 'Should raise IntegrityError: ModuleList.smid'
        with self.assertRaises(IntegrityError):
            with transaction.atomic():
                obj = mixer.blend('aumwi.ModuleList', name = None ), 'Should raise IntegrityError: ModuleList.name'

class TestAuthUser(TestCase): # Must explicitly be given inputs
    def test_init(self):
        obj = mixer.blend('aumwi.AuthUser',
        user_id = 'f92f4747-7cd4-4afd-990b-8d0185370038',
        smid_id = 'a9147eda-ed55-4b9c-bff0-192a6d580a6b')
        a_user = AuthUser.objects.all()
        self.assertEqual(len(a_user), 1), 'Should create an AuthUser instance.'

class TestUsageLog(TestCase): # Must explicitly be given inputs
    def test_init(self):
        obj = mixer.blend('aumwi.UsageLog',
        user_id = '5dad0428-1da1-4d6d-8cd2-ca88ad23a6f6',
        smid_id = '087e4535-15dd-4381-a2f5-a2ba938ff880',
        )
        u_log = UsageLog.objects.all()
        self.assertEqual(len(u_log), 1), 'Should create a UsageLog instance.'
```

**About the code:**

I've tried to comment the code to explain why certain things are used and what results should be for tests, but feel a bit more clarification might be helpful.

mixer:

As stated earlier in this document, mixer is a test fixture creator.

You could use standard object implementation…

```
obj = User(
id = '5dad0428-1da1-4d6d-8cd2-ca88ad23a6f6',
username = 'tinyrobotarmy',
full_name = 'Yo Momma binShoppin',
short_name = 'Yo Mom',
email = 'dontpantsme@gmail.com',
rfid = 9223372036854775807,
.
.
.
# And the other 20 fields for this model
)
```

Or use mixer and save many lines of coding

```
obj = mixer.blend('aumwi.User')
```

You can also set fields like so: `obj = mixer.blend('aumwi.User', username = 'tinyrobotarmy')`

When testing unique fields, mixer alone will not work when testing duplicate primary key values, because the interpreter thinks you are just updating an existing entry, so you have to explicitly feed information for the duplicate entry and force an insert/save.

Testing creating model instances:

Theoretically, one could use the primary key to test if a model instance was created

```
assert obj.pk == 1, 'Should create a User instance'
```

This did not work for me, so I used the following "work around" to test the length of the entries

```
self.assertEqual(len(users), 1), 'Should create a User instance.'
```

Testing unique and/or non-empty fields:

When testing for things like uniqueness, what you're actually testing for is to see if an error is thrown for violating said uniqueness. This requires some research to determine the type of error one can expect from a specific violation and then testing it. (-or putting things in you know shouldn't work and seeing the error thrown in the test results)

In the case of uniqueness, for the models have fields that require such, I grouped tests for each field in one model test for efficiency. The problem with that is that once the first error is thrown the others won't be caught. To get around this, I ran each field test to execute within its own transaction.

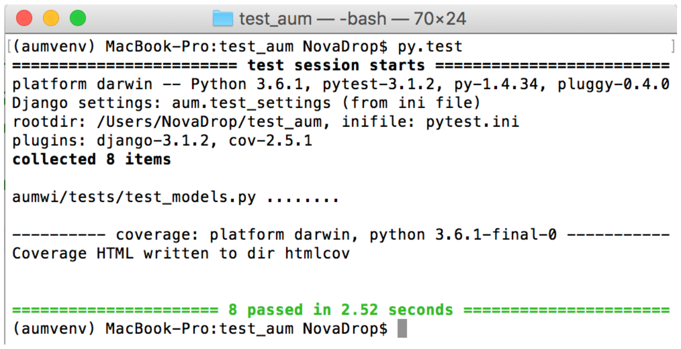
Concerns:

I've noticed that fields for the User model, e.g. name and username, have need for added restriction on minimum character input. At this time, one can type a space and the fields will save.

4. Run the tests (Should be done after each new test is created)

Command line:

```
(aumvenv) ~$ py.test
```



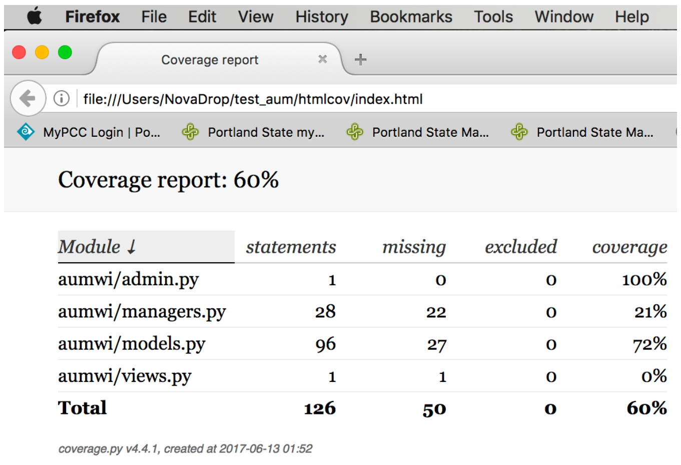
```
(aumvenv) MacBook-Pro:test_aum NovaDrop$ py.test
======================== test session starts =========================
platform darwin -- Python 3.6.1, pytest-3.1.2, py-1.4.34, pluggy-0.4.0
Django settings: aum.test_settings (from ini file)
rootdir: /Users/NovaDrop/test_aum, inifile: pytest.ini
plugins: django-3.1.2, cov-2.5.1
collected 8 items

aumwi/tests/test_models.py ........

---------- coverage: platform darwin, python 3.6.1-final-0 -----------
Coverage HTML written to dir htmlcov


====================== 8 passed in 2.52 seconds ======================
(aumvenv) MacBook-Pro:test_aum NovaDrop$
```

5. Look at the test coverage report

Command line:

```
(aumvenv) ~$ open htmlcov/index.html
```

This is the coverage report main page that will pull up in your browser when you run the coverage report.


Firefox File Edit View History Bookmarks Tools Window Help
Coverage report
file:///Users/NovaDrop/test_aum/htmlcov/index.html
MyPCC Login | Po...
Portland State my...
Portland State Ma...
Portland State Ma...

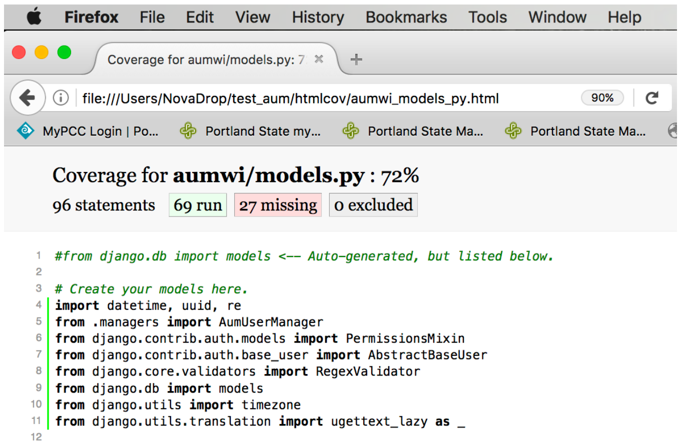
Coverage report: 60%

| Module ↓ | statements | missing | excluded | coverage |
|---|---|---|---|---|
| aumwi/admin.py | 1 | 0 | 0 | 100% |
| aumwi/managers.py | 28 | 22 | 0 | 21% |
| aumwi/models.py | 96 | 27 | 0 | 72% |
| aumwi/views.py | 1 | 1 | 0 | 0% |
| **Total** | **126** | **50** | **0** | **60%** |

*coverage.py v4.4.1, created at 2017-06-13 01:52*

e.g.

If you click on "**aumwi/models.py**" in the coverage report, this is what you'll see at the top of the page.

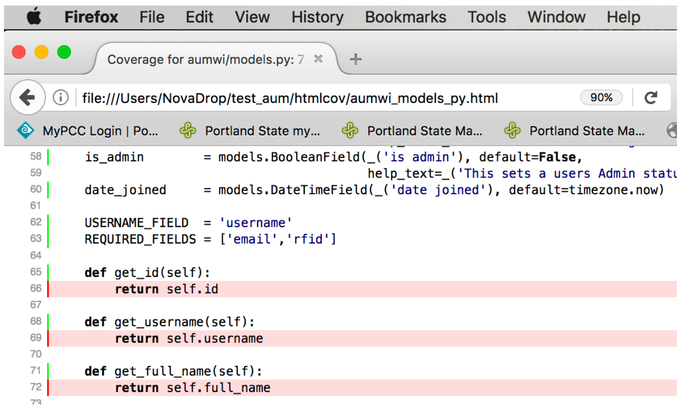

Firefox File Edit View History Bookmarks Tools Window Help
Coverage for aumwi/models.py: 7
file:///Users/NovaDrop/test_aum/htmlcov/aumwi_models_py.html
90%
MyPCC Login | Po...
Portland State my...
Portland State Ma...
Portland State Ma...

Coverage for **aumwi/models.py** : 72%

96 statements 69 run 27 missing 0 excluded

```
#from django.db import models <-- Auto-generated, but listed below.

# Create your models here.
import datetime, uuid, re
from .managers import AumUserManager
from django.contrib.auth.models import PermissionsMixin
from django.contrib.auth.base_user import AbstractBaseUser
from django.core.validators import RegexValidator
from django.db import models
from django.utils import timezone
from django.utils.translation import ugettext_lazy as _

```

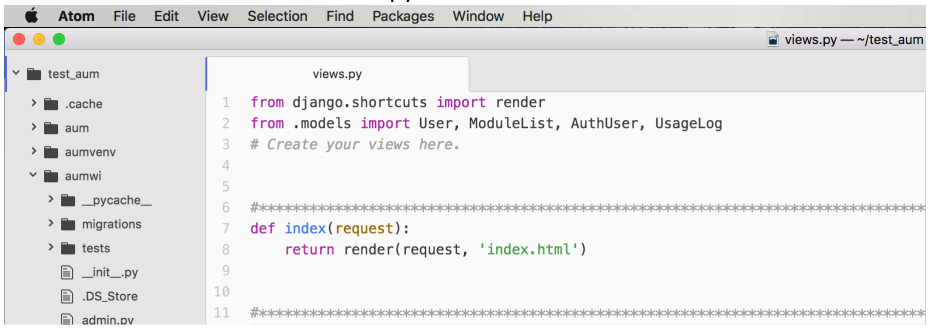
If you scroll down, you'll see that code that has been "hit" by the tests has a green line next to it and code that hasn't has a red line and is highlighted pink.


Firefox File Edit View History Bookmarks Tools Window Help
Coverage for aumwi/models.py: 7
file:///Users/NovaDrop/test_aum/htmlcov/aumwi_models_py.html
90%
MyPCC Login | Po...
Portland State my...
Portland State Ma...
Portland State Ma...

```
    is_admin        = models.BooleanField(_('is admin'), default=False,
                                          help_text=_('This sets a users Admin statu
    date_joined     = models.DateTimeField(_('date joined'), default=timezone.now)

    USERNAME_FIELD  = 'username'
    REQUIRED_FIELDS = ['email','rfid']

    def get_id(self):
        return self.id

    def get_username(self):
        return self.username

    def get_full_name(self):
        return self.full_name
```

## Testing Views

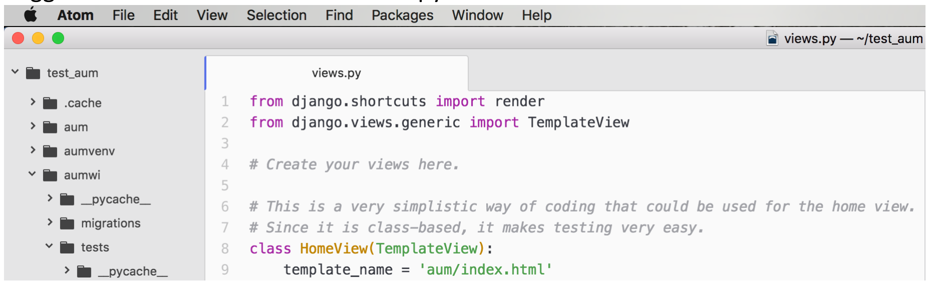
Current code for home view: views.py



```
from django.shortcuts import render
from .models import User, ModuleList, AuthUser, UsageLog
# Create your views here.


#*****************************************************************************
def index(request):
    return render(request, 'index.html')


#*****************************************************************************
```

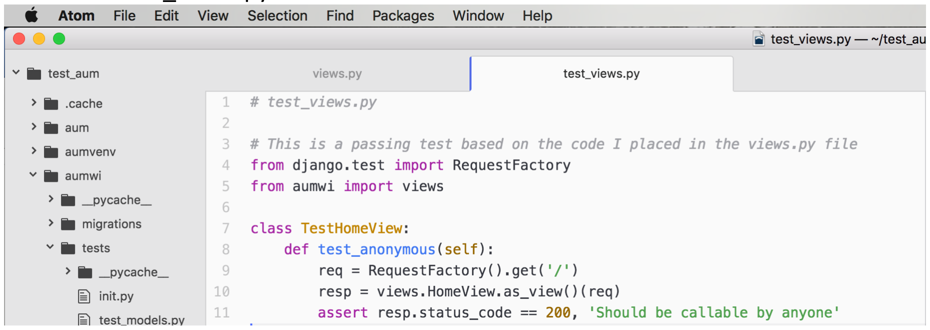
Issue: This code may work (I don't know), but it is not class-based, which makes it hard to test.

Suggested code for home view: views.py



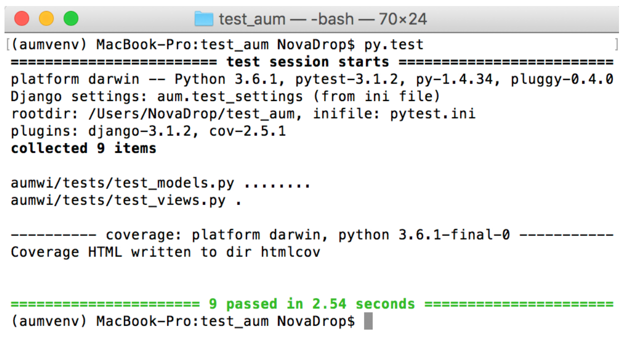
```
from django.shortcuts import render
from django.views.generic import TemplateView

# Create your views here.

# This is a very simplistic way of coding that could be used for the home view.
# Since it is class-based, it makes testing very easy.
class HomeView(TemplateView):
    template_name = 'aum/index.html'
```

Test code: test_views.py



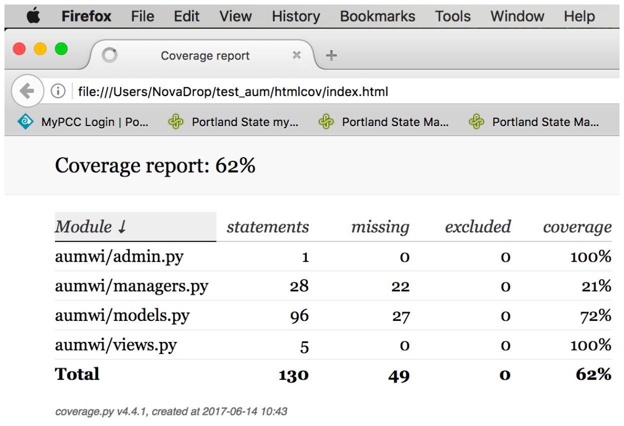
```
# test_views.py

# This is a passing test based on the code I placed in the views.py file
from django.test import RequestFactory
from aumwi import views

class TestHomeView:
    def test_anonymous(self):
        req = RequestFactory().get('/')
        resp = views.HomeView.as_view()(req)
        assert resp.status_code == 200, 'Should be callable by anyone'
```

Test results:
(This would make the 9th test)



```
(aumvenv) MacBook-Pro:test_aum NovaDrop$ py.test
======================== test session starts ========================
platform darwin -- Python 3.6.1, pytest-3.1.2, py-1.4.34, pluggy-0.4.0
Django settings: aum.test_settings (from ini file)
rootdir: /Users/NovaDrop/test_aum, inifile: pytest.ini
plugins: django-3.1.2, cov-2.5.1
collected 9 items

aumwi/tests/test_models.py ........
aumwi/tests/test_views.py .

---------- coverage: platform darwin, python 3.6.1-final-0 -----------
Coverage HTML written to dir htmlcov


====================== 9 passed in 2.54 seconds ======================
(aumvenv) MacBook-Pro:test_aum NovaDrop$
```

Coverage Report:


Firefox File Edit View History Bookmarks Tools Window Help
Coverage report
file:///Users/NovaDrop/test_aum/htmlcov/index.html
MyPCC Login | Po...
Portland State my...
Portland State Ma...
Portland State Ma...

Coverage report: 62%

| Module ↓ | statements | missing | excluded | coverage |
|---|---|---|---|---|
| aumwi/admin.py | 1 | 0 | 0 | 100% |
| aumwi/managers.py | 28 | 22 | 0 | 21% |
| aumwi/models.py | 96 | 27 | 0 | 72% |
| aumwi/views.py | 5 | 0 | 0 | 100% |
| **Total** | **130** | **49** | **0** | **62%** |

coverage.py v4.4.1, created at 2017-06-14 10:43

I have left these code changes, for the home view, and the accompanying test in the testing files for the AUM project.

## Testing Admins

This is for tests which have to do with views in admin list views.

e.g. an abridged list view of AUM user data.

## Testing Update User/Module Requests

This is for tests which have to do with views that use forms to update existing entries in the database.

## Testing Authentication

This is for tests which have to do with views that can only be accessed by super users etc.  and not just anyone.

## Testing Forms

This is for tests which have to do with entering new data into a form and field entry restrictions etc. –Not testing templates here.

## Testing 404 Errors

This is for tests which have to do with catching 404 exceptions, so they do not bubble up into your tests causing them to crash.

Again, see the tutorial and reference document I’ve included with the testing documentation for the AUM project to see where testing would be going were there more time.

Thank you,
~Jessica Blasch

END.