eDocent Testing Documentation

Web Application

For testing the web application, much use of manual testing was done. We eventually found it to be quicker to add some basic regression testing to make sure that new changes would not break old code. We wanted to make sure that the admin site was still function and able to add new objects. The testing was not completely thorough, but enough to catch any major issues. The test below checks to see that we can log into the admin site and that the necessary fields are there.

```
def test_can_create_new_museum_via_admin_site(self):
    # Gertrude opens her web browser, and goes to the admin page
    self.browser.get(self.live_server_url + '/admin/')

    # She sees the familiar 'Django administration' heading
    body = self.browser.find_element_by_tag_name('body')
    self.assertIn('Django administration', body.text)

    username_field = self.browser.find_element_by_name('username')
    username_field.send_keys('admin')

    password_field = self.browser.find_element_by_name('password')
    password_field.send_keys('admin')
    password_field.send_keys(Keys.RETURN)

    body = self.browser.find_element_by_tag_name('body')
    self.assertIn('Site administration', body.text)

    museum_links = self.browser.find_elements_by_link_text('Museums')

    self.assertEquals(len(museum_links), 1)

    museum_links[0].click()

    body = self.browser.find_element_by_tag_name('body')
    self.assertIn('0 museums', body.text)

    # She sees a link to 'add' a new poll, so she clicks it
    new_museum_link = self.browser.find_element_by_link_text('Add museum')
    new_museum_link.click()

    body = self.browser.find_element_by_tag_name('body')
    self.assertIn('Name:', body.text)
    self.assertIn('City', body.text)
    self.assertIn('State',body.text)
```

Another example of this create an assortment of a museum, artist, collection and art object to make sure the has-a and has-many relationships are functional in how the models are defined. By creating the different objects and connecting them in their correct relationships we are testing that the objects are being saved and are able to reference each other.

```
class MuseumModelTest(TestCase):

    def test_creating_a_new_objects_and_saving_it_to_the_database(self):
        # start by creating a new Poll object with its "question" set
        m = Museum()
        m.name = "Museum Test"
        m.streetAddress = "123 Easy St"
        m.city = "Boston"
        m.state = "MA"
        m.zipCode = "12345"
        m.openingHours_M = "9-5"
        m.openingHours_T = "9-5"
        m.openingHours_W = "9-5"
        m.openingHours_R = "9-5"
        m.openingHours_F = "9-5"
        m.openingHours_ST = "9-5"
        m.openingHours_SN = "9-5"

        m.events = "None"
        #m.image
        m.website = "www.mfa.org"
        m.description = "museum description"
        m.parking = "Yes"
        m.ticket_prices = "$25.00"
        m.visitor_info = "vistor information"
        m.membership = "yes"


        # check we can save it to the database
        m.save()

        # now check we can find it in the database again
        all_museums_in_database = Museum.objects.all()
        self.assertEquals(len(all_museums_in_database), 1)
        only_museum_in_database = all_museums_in_database[0]
        self.assertEquals(only_museum_in_database, m)

        # and check that it's saved its two attributes: question and pub_date
        self.assertEquals(only_museum_in_database.name, "Museum Test")
        self.assertEquals(only_museum_in_database.city, "Boston")

        # making a test collection
        c = Collection()
        c.title = "Collection 1"
        c.description = "test collection"
        c.museum = m
        c.save()

        # making a test artist
        a = Artist()
        a.name = "Mark"
        a.nationality = "Dutch"
        a.biography = "lorem ipsum prada"
        a.dateOfBirth = "2014-12-02"
        a.image = "image/test.jpg"
        a.save()

        # making a test Art object
        art = Art()
        art.title ="Art #1"
        art.artist = a;
```

Android Application

For testing the Android Application, we used a lot of manual testing while we were coding. This meant making sure certain methods were doing what we wanted and retrieving the information we needed. Another example of manual testing would be of variables as well. An example of manual testing from the DocentSplashPageActivity.java file is,

```
Log.d("Just City!!!!!!!!!!!!!!!!!!!",cities.getJSONObject(i).getString("city"));
```

Here, we are checking if we are receiving a list of the cities in order to dynamically fill the City spinner once the user selected a state. These type of manual tests were done all throughout our coding and are still in the code.

We also used the Android test automation framework Robotium. This allowed us to test all of the functionality of our Android Application. We created a test android application, which targets our eDocent Project in Eclipse. We test the splash page using the code,

```
 public void testSplashPage()  {
  solo.assertCurrentActivity("testing splash", EDocentSplashActivity.class);
  //selects MA
    solo.pressSpinnerItem(0, 0);
    //select Boston
    solo.pressSpinnerItem(1, 0);
    //select MFA
    solo.pressSpinnerItem(2, 0);
    //click on the button
    solo.clickOnButton("Look For A Museum!");
  }
```

We run this test project and if a test passes, then a green bar appears. Otherwise, if a test fails, then a red bar appears along with a trace to show where the test failed.This is also what the results of a JUnit test looked like. We wrote tests that would definitely fail to make sure that our test methods were being executed. An example of a failed test case is,

```
public void testSplashPage()  {
  solo.assertCurrentActivity("testing splash", EDocentSplashActivity.class);
  fail();
}
```

Therefore, we know testSplashPage is actually being executed. To view our Robotium tests, go to our GitHub project page (https://github.com/tab262/cs673), go to the branch “android-testing.” Click on the folder “VirtualMuseumCuratorTest.” Then, click on “src,” “com,” “bucs,” “virtualmuseumcurator,” “test” and then the EDocentSplashActivityTest.java file.

Finally, we also did unit testing by using Eclipse ADT and the provided JUnit framework. We had to create a JUnit test application. Methods needed a @Test tag before them so the compiler understood it was a test. An example for our museum database is,

```
@Test
public void getCategories() {
        CategoryType[] categories = database.getCategories();
        //22 categories (name, address, description, etc.)
        assertTrue(categories.length == 22);
        assertTrue(categories[0].getCategoryName().equals("name"));
        assertTrue(categories[1].getCategoryName().equals("streetaddress"));
        assertTrue(categories[2].getCategoryName().equals("city"));
        assertTrue(categories[3].getCategoryName().equals("state"));
        assertTrue(categories[4].getCategoryName().equals("zipcode"));
}
```

To view what this database looks like, go to http://edocent.herokuapp.com/curator/ .

Initially while trying new features in an individual android project, we also tested few features like audio using Robotium:

```
public void testAudio()
   {
        solo.assertCurrentActivity("Check on first activity", ArtInfoActivity.class);
        solo.clickOnButton("Play");
        solo.assertCurrentActivity("I just Played the song", ArtInfoActivity.class);
        solo.clickOnButton("Stop");
        solo.assertCurrentActivity("I just Stopped the song", ArtInfoActivity.class);
   }
```