System Test Document

Performing Ecological Momentary Assessments (EMA) via Picroft or Mobile App

CA400
Santiago de Arribas de Renedo • 16449272
Pedro Marques Canes • 16770821

Supervisor: Dr. Tomas Ward

Date Complete: 16/05/2020

Table of Contents

Table of Contents **2**

Abstract **3**

Overview **3**

API and Database **3**

Unit Testing the API 3

Ad-Hoc Testing 7

Mycroft **7**

Unit Testing 7

Testing the Skill 8

Ad-Hoc Testing Mycroft 9

Android App **9**

Unit Testing 9

Functional Testing 10

Logs 11

Abstract

Testing is a vital aspect of building any reliable software. A system that has multiple components requires extensive testing to make sure that it continues to function as intended as each new feature is added. Not only do the features themselves need to be tested to ensure they fundamentally work but once integrated they need to be tested with the rest of the system to make sure that everything still works as before.

Overview

Our system consists of many different components that need to work together. The backend of the system features a custom API that allows other elements of the system to communicate to the database. The mobile app launches on an Android platform while the open source Mycroft AI is built on a raspberry pi to be used as the home assistant. We've designed a custom Mycroft skill to be used by the user to interact with our system which uses additional scripts to pull GPS information from the pi. All these various components work differently and as such need appropriate testing. Unit testing and functional testing as well as integration testing are necessary to ensure the robustness and reliability of these components and their connected functionality.

API and Database

The API consists of 8 different resources which allow the app or mycroft skill to communicate with the 8 tables in the database. These resources use HTTP request methods that are responsible for performing Create, Read, Update and Delete (CRUD) operations on the database.

Unit Testing the API

The API is tested using unit testing. This is primarily used to check the HTTP request methods of each of the resources. If we look at an example of a unit test for a specific resource - /*user* we can see a breakdown of this test:

```python
import unittest
import json

from manage import app
from app.main import db

class UserTest(unittest.TestCase):

    def setUp(self):
        self.app = app.test_client()
        self.db = db.get_db()

    def test_post(self): #Post a new user
        payload = json.dumps({
            "id": 1000,
            "username": "user1000",
            "pair_pin": "9999"
        })

        response = self.app.post('/user', headers={"Content-Type": "application/json"}, data=payload)

        self.assertEqual(int, type(response.json['id']))
        self.assertEqual(200, response.status_code)

        payload = json.dumps({
        "id": 1001,
        "username": "user1001",
        "pair_pin": "0000000000"
        })

        response = self.app.post('/user', headers={"Content-Type": "application/json"}, data=payload)
        #This should result in  a500 error as there cannot be a pair pin longer than 9 characters

        self.assertEqual(500, response.status_code)
```

We begin by using a *setUp* method to start our unit tests. A *test_post* method tests a POST request. It checks to make sure that a pin over 9 characters cannot be added.

```python
    def test_get_all(self): #Get all users
        payload = json.dumps({
            "id": 1000,
            "username": "user1000",
            "pair_pin": "9999"
        })

        self.app.post('/user', headers={"Content-Type": "application/json"}, data=payload)
        response = self.app.get('/user', headers={"Content-Type": "application/json"}, data=payload)

        self.assertEqual(int, type(response.json['id']))
        self.assertEqual(200, response.status_code)
```

Next, this *test_get_all* method checks that our GET request functions as expected.

```
    def test_get(self): #Get user 1000
        payload = json.dumps({
            "id": 1000,
            "username": "user1000",
            "pair_pin": "9999"
        })

        self.app.post('/user', headers={"Content-Type": "application/json"}, data=payload)
        response = self.app.get('/user/1000', headers={"Content-Type": "application/json"}, data=payload)

        self.assertEqual(int, type(response.json['id']))
        self.assertEqual(200, response.status_code)

        response = self.app.get('/user/1002', headers={"Content-Type": "application/json"}, data=payload)

        self.assertEqual(int, type(response.json['id']))
        self.assertEqual(404, response.status_code)
        #This should result in a 404 error as user 1002 does not exist
```

This *test_get* method checks that a specific user is in the database. It also checks the validity of a non-existent user: **1002.**

```
    def test_put(self): #Update pair pin
       payload = json.dumps({
         "id": 1000,
         "username": "user1000",
         "pair_pin": "9999"
    })

        self.app.post('/user', headers={"Content-Type": "application/json"}, data=payload)

        payload = json.dumps({
            "pair_pin": "0000"
        })

        response = self.app.put('/user/1000', headers={"Content-Type": "application/json"}, data=payload)

        self.assertEqual(int, type(response.json['id']))
        self.assertEqual(200, response.status_code)

        response = self.app.get('/user/1002', headers={"Content-Type": "application/json"}, data=payload)

        self.assertEqual(int, type(response.json['id']))
        self.assertEqual(404, response.status_code)
        #This should result in a 404 error as user 1002 does not exist
```

This *test_put* method tests the PUT request of our API.

```
    def test_delete(self): #Delete user
       payload = json.dumps({
            "id": 1000,
            "username": "user1000",
            "pair_pin": "9999"
        })

        self.app.post('/user', headers={"Content-Type": "application/json"}, data=payload)
        response = self.app.delete('/user/1000', headers={"Content-Type": "application/json"}, data=payload)

        self.assertEqual(int, type(response.json['id']))
        self.assertEqual(200, response.status_code)

        response = self.app.get('/user/1002', headers={"Content-Type": "application/json"}, data=payload)

        self.assertEqual(int, type(response.json['id']))
        self.assertEqual(404, response.status_code)
        #This should result in a 404 error as user 1002 does not exist

    def tearDown(self):
        # Delete Database collections after the test is complete
        for collection in self.db.list_collection_names():
            self.db.drop_collection(collection)
```

Finally we test a DELETE method and use *tearDown* to finish our unit tests.

These tests can run by changing the environment used by *manage.py* from *prod* to *test.*

```
app = create_app('prod')
app.app_context().push()
manager = Manager(app)
migrate = Migrate(app, db)
manager.add_command('db', MigrateCommand)
app = create_app('prod')
app.register_blueprint(blueprint)
app.app_context().push()

@manager.command
def run():
    app.run()

@manager.command
def test():
    """Runs the unit tests."""
    tests = unittest.TestLoader().discover('app/test', pattern='test*.py')
    result = unittest.TextTestRunner(verbosity=2).run(tests)
    if result.wasSuccessful():
        return 0
    return 1

if __name__ == '__main__':
    manager.run()
```

The *test* method defines using the unittest test runner to run any tests in the test folder with *test* in its name.

```
class DevelopmentConfig(Config):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = postgres_prod_base
    SQLALCHEMY_TRACK_MODIFICATIONS = False

class TestingConfig(Config):
    DEBUG = True
    TESTING = True
    SQLALCHEMY_DATABASE_URI = postgres_test_base
    PRESERVE_CONTEXT_ON_EXCEPTION = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False

class ProductionConfig(Config):
    DEBUG = True
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SQLALCHEMY_DATABASE_URI = postgres_prod_base

config_by_name = dict(
    dev=DevelopmentConfig,
    test=TestingConfig,
    prod=ProductionConfig
)
```

The different environments are set in the *config.py* file.

Ad-Hoc Testing

Ad-Hoc testing was also used to continuously test the API and database by continuously running API calls and viewing the Heroku logs. This allowed us to see if any resources had problems while the API was deployed.

Mycroft

Just as important as the API, it is important to test the Daisy skill so that we know it is performing certain actions as expected. In particular, we checked to make sure the *update_gps.py* and *get_questions.py* were functioning properly. We used unit testing for this also.

Unit Testing

Unit tests were written for the *update_gps.py* and *get_questions.py* programs. These are expected to update the home assistants GPS information, perform context detection correctly, save questions in the correct JSON format and prompt the Daisy skill.

```
import unittest
import json
import requests
import gps
import get_questions

class GPSTest(unittest.TestCase):
    def test_gps_type(self):
        gps_type = update_gps.get_gps()
        self.assertEqual(tup, type(gps_type)) #Should be in type tuple initially
        gps_str = update_gps.convert_tup_to_str(gps_type)
        self.assertEqual(str, type(gps_str)) #Should be in type string

if __name__ == '__main__':
    unittest.main()
```

This simple gps unittest asserts that the type of the final GPS result is in string format

```
import unittest
import json
import requests
import get_questions

class TestQuestion(unittest.TestCase):
    def test_user_details(self):
        payload = json.dumps({
            "id": 1000,
            "username": "user1000",
            "pair_pin": "9999"
        })

        requests.post('/user', headers={"Content-Type": "application/json"}, data=payload) #First create new user
```

The *TestQuestion* method creates all necessary objects to add a user details object.

```
        payload = json.dumps({
                "id": "1003",
                "user_ID": "1000",
                "question_ID": "1002",
                "ask_question": True,
                "user_available": "True",
                "device_to_use": "null"
            })

        requests.post('/user-details', headers={"Content-Type": "application/json"}, data=payload) #Create a new user-details object
        self.assertEqual(200, response.status_code) #Check if this was successful

    def test_check_user_available(self):
        check_user = check_user(1000)
        self.assertEqual("True", check_user) #check that user is available

if __name__ == '__main__':
    unittest.main()
```

This object is populated with dummy data. The *test_check_user_available* method checks to make sure the *user_available* field has been set correctly.

These tests are run using:

```
python -m unittest <test_file>
```

Testing the Skill

Skills are tested using an automated testing framework. When skills are updated or uploaded to the marketplace, they are tested to see if they have any conflicts with other skills. However, additional tests can also be written to test intents. A *test/intent* folder can be created with filenames containing a *.intent.json* suffix.

```
{
    "utterance": "hi daisy",
    "expected_response": "hi, have you registered with the daisy app"
}
```

An utterance test to make sure Daisy responds appropriately when a user first interacts with the skill.

These tests can be run by using the testing scripts available in the *mycroft-core* repository. The scripts available to test the Mycroft skill are as follows:

discover_test.py	Pytest script that runs all tests in all the skills directories.	Activate by running ***pytest discover_test.py*** which will find all available tests for every skill.
single_test.py	This is used to run tests for a single skill.	Activate by running ***python single_test.py***

		/path/to/skill which will find all available tests for that skill.
skill_developers_testrunner.py	This is a script that script that can be copied to a skill folder and run any tests for that skill	Activate by running ***python skill_developers_testrunner.py*** in the skills folder.

Ad-Hoc Testing Mycroft

This type of manual testing was also used to test the Daisy skill and see if there were any problems with the skill. By continuously running through the commands, we could see how the Mycroft performed when having an interaction with the user. Through this we were able to determine that at certain points, Mycroft had difficulty understanding utterances and by implementing *LOG.info() outputs - essentially* Python **print** statements we could see what info Mycroft was receiving. However, this type of testing was extremely tedious and not recommended for a more complex skill.

The *update_gps.py* file also updated a log file each time it was accessed. This was to check that the Daisy skill was running this file as requested.

```
logs = root_dir + "/access_logs.txt"
myFile = open(logs, "a")
myFile.write("\nUPDATE_GPS.PY: Accessed on " + str(datetime.now()))
```

Updating *access_logs.txt.* Checking this file indicates if the program is running at the right time.

Android App

Unit Testing

Unit tests were written for all activities in the App. These tests can be found in the test folder of the App package. Unit testing in the App was made using JUnit. These tests have the aim to assure that all activities are launched properly. This is done by running the activity and finding all the elements present in the respective XML layout file.

```
@Rule
public ActivityTestRule<Register> mActivityTestRule = new ActivityTestRule<>(Register.class);

private Register mActivity = null;

Instrumentation.ActivityMonitor monitor = getInstrumentation().addMonitor(Login.class.getName(),  result: null,  block: false);
Instrumentation.ActivityMonitor userMonitor = getInstrumentation().addMonitor(MainActivity.class.getName(),  result: null,  block: false);
Instrumentation.ActivityMonitor researcherMonitor = getInstrumentation().addMonitor(Researcher.class.getName(),  result: null,  block: false);

@Before
public void setUp() throws Exception {
    mActivity = mActivityTestRule.getActivity();
}

@Test
public void testLaunch() {
    View appNameView = mActivity.findViewById(R.id.app_name);
    View appCreateAccountInfoView = mActivity.findViewById(R.id.app_create_account_info);
    View fullNameView = mActivity.findViewById(R.id.fullName);
    View emailView = mActivity.findViewById(R.id.email);
    View passwordView = mActivity.findViewById(R.id.password);
    View userTypeView = mActivity.findViewById(R.id.userType);
    View registerBtnView = mActivity.findViewById(R.id.registerBtn);
    View loginTextView = mActivity.findViewById(R.id.loginText);
    View progressBarView = mActivity.findViewById(R.id.progressBar);

    assertNotNull(appNameView);
    assertNotNull(appCreateAccountInfoView);
    assertNotNull(fullNameView);
    assertNotNull(emailView);
    assertNotNull(passwordView);
    assertNotNull(userTypeView);
    assertNotNull(registerBtnView);
    assertNotNull(loginTextView);
    assertNotNull(progressBarView);
}
```

Sample of unit testing code from Register activity. Tests are making sure that all elements of the Register screen are launching properly.

Functional Testing

Functional tests are found in the same file as the previously mentioned unit tests. Each activity has its own test file. Functional tests are used for the activities to make sure activities transition properly between each other. The functional tests were written using JUnit and Espresso.

```
@Test
public void testLaunchOfLoginOnClick() {
    assertNotNull(mActivity.findViewById(R.id.loginText));

    onView(withId(R.id.loginText)).perform(click());
    Activity loginActivity = getInstrumentation().waitForMonitorWithTimeout(monitor,  timeOut: 5000);

    assertNotNull(loginActivity);

}

@Test
public void testLaunchOfUserOnRegister() {
    assertNotNull(mActivity.findViewById(R.id.registerBtn));

    onView(withId(R.id.registerBtn)).perform(click());
    Activity userActivity = getInstrumentation().waitForMonitorWithTimeout(userMonitor,  timeOut: 5000);

    assertNull(userActivity);
}

@Test
public void testLaunchOfResearcherOnRegister() {
    assertNotNull(mActivity.findViewById(R.id.registerBtn));

    onView(withId(R.id.registerBtn)).perform(click());
    Activity resActivity = getInstrumentation().waitForMonitorWithTimeout(researcherMonitor,  timeOut: 5000);

    assertNull(resActivity);
}
```

Sample of the functional test code for the Register screen. These tests are making sure that the activities move properly from one to the other.

Logs

Throughout the development of the project, logs were written to test the different features. These were done using Logcat.

```
private void createUser(String userName) {
    DBUser dbUser = new DBUser(userID, userName,  pair_pin: "null");

    Log.d( tag: "Register",  msg: "ID: " + dbUser.getId());
    Log.d( tag: "Register",  msg: "USERNAME:  " + dbUser.getUsername());
    Log.d( tag: "Register",  msg: "PAIR PIN:  " + dbUser.getPair_pin());

    Call<DBUser> call = daisyApi.createUser(dbUser);

    call.enqueue(new Callback<DBUser>() {
        @Override
        public void onResponse(Call<DBUser> call, Response<DBUser> response) {

            if (!response.isSuccessful()) {
                Log.d( tag: "RegisterError",  msg: "ERROR CREATING USER: " + response.code());
                Log.d( tag: "RegisterError",  msg: "ERROR CREATING USER:  " + response.message());
                Log.d( tag: "RegisterError",  msg: "ERROR CREATING USER:  " + response.raw().toString());
                return;
            }

            DBUser userResponse = response.body();
            Log.d( tag: "RegisterSuccess",  msg: "USER CREATED SUCCESSFULLY CODE: " + response.code() + response.message());
            Log.d( tag: "RegisterSuccess",  msg: "ID RESPONSE: " + userResponse.getUser().getId());
            Log.d( tag: "RegisterSuccess",  msg: "PAIR_PIN RESPONSE:  " + userResponse.getUser().getPair_pin());
            Log.d( tag: "RegisterSuccess",  msg: "USERNAME RESPONSE:  " + userResponse.getUser().getUsername());


            final Handler handler = new Handler();
            handler.postDelayed(new Runnable() {
                @Override
                public void run() {
                    //Do something after 2000ms
                    createPhone();
                }
            },  delayMillis: 500);

        }
```

Logs underlined in orange.

```
2020-05-16 16:47:41.187 29575-29575/com.example.daisyandroidapp D/Register: ID: RWRbIfmcxweYv4hCszfXdIx44Ub2
2020-05-16 16:47:41.187 29575-29575/com.example.daisyandroidapp D/Register: USERNAME: mana
2020-05-16 16:47:41.187 29575-29575/com.example.daisyandroidapp D/Register: PAIR PIN: null
2020-05-16 16:47:41.621 29575-29575/com.example.daisyandroidapp D/RegisterSuccess: USER CREATED SUCCESSFULLY CODE: 2000K
2020-05-16 16:47:41.621 29575-29575/com.example.daisyandroidapp D/RegisterSuccess: ID RESPONSE: RWRbIfmcxweYv4hCszfXdIx44Ub2
2020-05-16 16:47:41.621 29575-29575/com.example.daisyandroidapp D/RegisterSuccess: PAIR_PIN RESPONSE: null
2020-05-16 16:47:41.621 29575-29575/com.example.daisyandroidapp D/RegisterSuccess: USERNAME RESPONSE: mana
2020-05-16 16:47:41.767 29575-29575/com.example.daisyandroidapp W/RegisterActivity: Collection usernames updated successfully
2020-05-16 16:47:41.777 242-999/? I/BufferQueueProducer: [com.example.daisyandroidapp/com.example.daisyandroidapp.Register](this:0xa93d6000,id:666,api:1,p:29575,c:242) queueBuffer: fps=60.42 dur=1009.59 max=27.62 min=9.34
2020-05-16 16:47:42.615 29575-29575/com.example.daisyandroidapp D/RegisterSuccess: PHONE CREATED SUCCESSFULLY CODE: 2000K
2020-05-16 16:47:42.615 29575-29575/com.example.daisyandroidapp D/RegisterSuccess: ID RESPONSE: null
2020-05-16 16:47:42.615 29575-29575/com.example.daisyandroidapp D/RegisterSuccess: SERIALKEY RESPONSE: null
2020-05-16 16:47:42.615 29575-29575/com.example.daisyandroidapp D/RegisterSuccess: USERID RESPONSE: null
2020-05-16 16:47:42.781 242-998/? I/BufferQueueProducer: [com.example.daisyandroidapp/com.example.daisyandroidapp.Register](this:0xa93d6000,id:666,api:1,p:29575,c:242) queueBuffer: fps=60.78 dur=1003.59 max=30.02 min=7.29
2020-05-16 16:47:42.926 29575-29575/com.example.daisyandroidapp D/RegisterSuccess: PHONE ID SUCCESSFULLY ADDED TO FIRESTORE
```

Logs when the user is registering.

```java
private void updatePhone(String latLong) {
    Call<DBPutResponse> call = daisyApi.updatePhoneCoords(phoneId, latLong);
    call.enqueue(new Callback<DBPutResponse>() {
        @Override
        public void onResponse(Call<DBPutResponse> call, Response<DBPutResponse> response) {
            if (!response.isSuccessful()){
                //response not successful
                Log.d( tag: "TrackServiceAPIError",  msg: "Response not successful");
                Log.d( tag: "TrackServiceAPIError",  msg: "CODE: " + response.code() + response.message());
                return;
            }

            //response successful
            Log.d( tag: "TrackServiceSuccess",  msg: "Database updated coords successfully");
            Log.d( tag: "TrackServiceSuccess",  msg: "LatLong: " + latLong);
            Log.d( tag: "TrackServiceSuccess",  msg: "CODE: " + response.code());
        }

        @Override
        public void onFailure(Call<DBPutResponse> call, Throwable t) {
            Log.d( tag: "TrackServiceAPIError",  msg: "PUT REQUEST ERROR: " + t.getMessage());
        }
    });
}
```

Logs underlined in orange.

```
2020-05-16 16:47:46.189 29575-29575/com.example.daisyandroidapp D/TrackServiceSuccess: Database updated coords successfully
2020-05-16 16:47:46.190 29575-29575/com.example.daisyandroidapp D/TrackServiceSuccess: LatLong: 53.283072,-6.2584865
2020-05-16 16:47:46.190 29575-29575/com.example.daisyandroidapp D/TrackServiceSuccess: CODE: 200
```

Logs when the location is being updated.

```
@Override
public void onError(int i) {
    Log.d(TAG, msg: "ERROR: " + i);
    if ((i == SpeechRecognizer.ERROR_NO_MATCH)
            || (i == SpeechRecognizer.ERROR_SPEECH_TIMEOUT))
    {
        Log.d(TAG, msg: "didn't recognize anything");
        userResults += "";
        // keep going
        if (listening) {
            Log.d(TAG, msg: "Listening again");
            mSpeechRecognizer.destroy();
            listen();
        }
    }
}

@Override
public void onResults(Bundle resultsBundle) {
    Log.d(TAG, msg: "onResults: " + resultsBundle.toString());
    ArrayList<String> matches = resultsBundle.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION);
    if (matches.get(0) != "null")
        userResults += " " + matches.get(0);
    else
        userResults += "";
    if (matches != null)
        Log.d(TAG, msg: "matches: " + matches.get(0));
    Log.d(TAG, msg: "I have listen to: " + userResults);
    if (listening) {
        Log.d(TAG, msg: "Listening again");
        mSpeechRecognizer.startListening(mSpeechRecognizerIntent);
    }
}
```

Logs underlined in orange.

```
2020-05-16 16:47:48.120 29575-29575/com.example.daisyandroidapp D/MicrophoneService: onResults: Bundle[mParcelledData.dataSize=312]
2020-05-16 16:47:48.120 29575-29575/com.example.daisyandroidapp D/MicrophoneService: matches: hello words
2020-05-16 16:47:48.120 29575-29575/com.example.daisyandroidapp D/MicrophoneService: I have listen to:  hello hello words
2020-05-16 16:47:48.121 29575-29575/com.example.daisyandroidapp D/MicrophoneService: Listening again
2020-05-16 16:47:48.124 29575-29575/com.example.daisyandroidapp D/MicrophoneService: onResults: Bundle[mParcelledData.dataSize=312]
2020-05-16 16:47:48.125 29575-29575/com.example.daisyandroidapp D/MicrophoneService: matches: hello words
2020-05-16 16:47:48.125 29575-29575/com.example.daisyandroidapp D/MicrophoneService: I have listen to:  hello hello words hello words
2020-05-16 16:47:48.125 29575-29575/com.example.daisyandroidapp D/MicrophoneService: Listening again
2020-05-16 16:47:48.595 29575-29575/com.example.daisyandroidapp D/MicrophoneService: ready service
2020-05-16 16:47:49.521 29575-29575/com.example.daisyandroidapp D/MicrophoneService: ERROR: 6
2020-05-16 16:47:49.521 29575-29575/com.example.daisyandroidapp D/MicrophoneService: didn't recognize anything
2020-05-16 16:47:49.521 29575-29575/com.example.daisyandroidapp D/MicrophoneService: Listening again
2020-05-16 16:47:49.525 29575-29575/com.example.daisyandroidapp D/MicrophoneService:  true
2020-05-16 16:47:49.526 29575-29575/com.example.daisyandroidapp D/MicrophoneService: onBeginingOfSpeech
2020-05-16 16:47:49.526 29575-29575/com.example.daisyandroidapp D/MicrophoneService: Listening now
2020-05-16 16:47:49.579 29575-29575/com.example.daisyandroidapp D/MicrophoneService: ready service
```

Logs of when microphone is listening.