



IoT HOMECARE SYSTEM TESTING

Hristian Vitrychenko
Nikki Constancon
Juan du Preez
Gregory Austin
Marthinus Richter

August 10, 2017

1 Introduction

The testing report is designed to describe all tests that have been done on the system and future tests to be done on the system.

1.1 Purpose

The purpose of this document will be to list and describe all of the tests for the ReVA system, how they are carried out, what their purpose is and what subsystem and related module(s) they are related to.

1.2 Structure of the document

Each subsystem of ReVA will be addressed individually with the specific tests for each module concerned with that subsystem. The subsystems are:

- Real-Time Subsystem
This includes the modules: Data Collection, Pub/Sub Server, Data Pull, and User Interface
- Data Storage Subsystem
This includes the modules: Data Storage, Pub/Sub Server
- History/Statistics Subsystem
This includes the modules: Data Storage, Statistics, Pub/Sub Server, and User Interface
- User Management Subsystem
This includes the modules: User Management, User Interface
- Notification Subsystem
This includes the modules: Pub/Sub Server, Notification, and User Interface
- Advice Subsystem
This includes the modules: Advice and User Interface

1.3 Definitions, Acronyms, and Abbreviations

1.3.1 Acronyms

- **UI (*User Interface*)**

The means by which the user and a computer system interact, in particular, the use of input devices and software.

1.3.2 Definitions

- **Unit Test**

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinised for proper operation.

- **Integration Test**

Find out if the units if integrated together will work without errors. For example, argument passing and data updation etc.

- **Functionality Test**

Tests all functionalities of the software against the requirement.

- **Performance Test**

This test proves how efficient the software is. It tests the effectiveness and average time taken by the software to do desired task. Performance testing is done by means of load testing and stress testing where the software is put under high user and data load under various environment conditions.

- **Alpha Testing**

The team of developer themselves perform alpha testing by using the system as if it is being used in work environment. They try to find

out how user would react to some action in software and how the system should respond to inputs.

1.4 Additional Information

The code being tested is written by different developers, thus this document serves as a way in order to review the functionality and tests done, and to provide information about future tests on future or current functionality.

2 Real-time subsystem

Modules involved: Data Collection, Pub/Sub Server, Data Pull, and User Interface

Implementation status: implemented

Primary Actors: All users

Functionality: View real time data on patient(s)

2.0.1 Description of current tests

The data streaming consists of Raspberry pi's streaming to servers, and servers streaming that data out and the application displaying those data streams. We do this using the Nodejs package Zetta (which is already been tested) for the Data Collection and the Pub/Sub Server, and an open source Zetta-Starter-Android-Application developed to interface with the Zetta API to receive the streams (DataPull) that's generated by the server which has also been tested. These have all been tested individually already, thus to create our own unit tests would be redundant.

The current tests are Integration Tests to see that data is in fact streaming from and to devices. You can see the tests in the three figures below.

2.0.2 Future tests:

Future tests will include Functionality Testing to confirm it adheres to requirements, Performance testing (seeing capacity and application performance) and some Alpha Testing (testing it ourselves to potentially find problems).

2.0.3 Data Collection

Comment: At the moment the Pi collects data from mock devices. This data generated and streamed using the zetta architecture and therefore has already been tested with Unit tests thoroughly. (Zetta is an open source Nodejs project purposed for IoT)

2.0.4 Pub/Sub Server

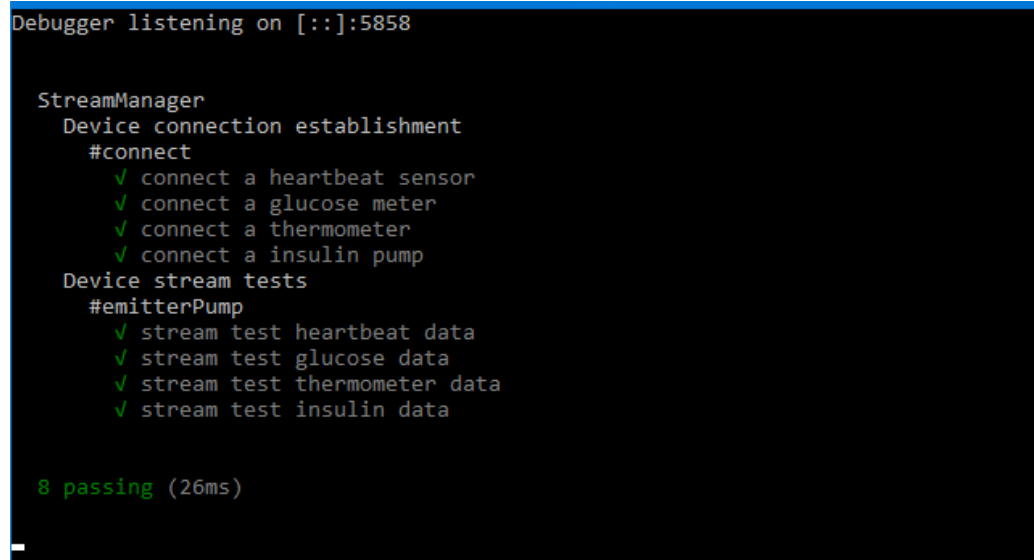
Comment: The Pub/Sub server uses zetta architecture as well and that's how it communicates with the pi (receives streams being published) and any requesting devices looking to subscribe. Zetta already is established and has tests.

2.0.5 Data Pull

Comment: The way the data is pulled is by using the architecture of the open source Zetta-Starter-Android-Application developed to interface with the Zetta API that's generated by the server. This has also been tested, so testing is redundant.

2.0.6 User Interface

Comment: The user interface only displays data, and therefore it is easy to verify what is being displayed and that it is working as expected.



```
Debugger listening on [::]:5858

StreamManager
  Device connection establishment
    #connect
      ✓ connect a heartbeat sensor
      ✓ connect a glucose meter
      ✓ connect a thermometer
      ✓ connect a insulin pump
    Device stream tests
      #emitterPump
        ✓ stream test heartbeat data
        ✓ stream test glucose data
        ✓ stream test thermometer data
        ✓ stream test insulin data

8 passing (26ms)
```

Figure 1: Mocha output of successful mock device streams


```

StreamManager
  Device connection establishment
    #connect
      1) connect a heartbeat sensor
      2) connect a glucose meter
      3) connect a thermometer
      4) connect a insulin pump
    Device stream tests
      #emitterPump
        5) stream test heartbeat data successful
        6) stream test glucose data successful
        7) stream test thermometer data successful
        8) stream test insulin data successful

0 passing (49ms)
8 failing

1) StreamManager Device connection establishment #connect connect a heartbeat
sensor:

    AssertionError: expected 'timeout' to equal 'device-link-heartbeat'
    + expected - actual

    -timeout
    +device-link-heartbeat

    at Context.<anonymous> (test\test_zettaStreams.js:17:50)

2) StreamManager Device connection establishment #connect connect a glucose me
ter:

    AssertionError: expected 'timeout' to equal 'device-link-glucosemeter'
    + expected - actual

    -timeout
    +device-link-glucosemeter

    at Context.<anonymous> (test\test_zettaStreams.js:20:50)

3) StreamManager Device connection establishment #connect connect a thermomete
r:

```

Figure 2: Mocha output (1 of 2) for unsuccessful mock device streams (due to being programmatic disabled)

3 Data Storage Subsystem

Modules involved: Data Storage, Pub/Sub Server

Implementation status: unimplemented

```
5) StreamManager Device stream tests #emitterPump stream test heartbeat data:

  AssertionError: expected 'null' to equal '[0,1,2,3,4,5,6,7,8,9]'
  + expected - actual

  -null
  +[0,1,2,3,4,5,6,7,8,9]

  at Context.<anonymous> (test\test_zettaStreams.js:34:48)

6) StreamManager Device stream tests #emitterPump stream test glucose data:

  AssertionError: expected 'null' to equal '[0,1,2,3,4,5,6,7,8,9]'
  + expected - actual

  -null
  +[0,1,2,3,4,5,6,7,8,9]

  at Context.<anonymous> (test\test_zettaStreams.js:37:48)

7) StreamManager Device stream tests #emitterPump stream test thermometer data
:

  AssertionError: expected 'null' to equal '[0,1,2,3,4,5,6,7,8,9]'
  + expected - actual

  -null
  +[0,1,2,3,4,5,6,7,8,9]

  at Context.<anonymous> (test\test_zettaStreams.js:40:48)

8) StreamManager Device stream tests #emitterPump stream test insulin data:

  AssertionError: expected 'null' to equal '[0,1,2,3,4,5,6,7,8,9]'
  + expected - actual

  -null
  +[0,1,2,3,4,5,6,7,8,9]

  at Context.<anonymous> (test\test_zettaStreams.js:43:48)
```

Figure 3: Mocha output (2 of 2) for unsuccessful mock device streams (due to being programmatic disabled)

3.0.1 Future tests:

Unit tests for individual modules, Data Storage in particular because the Pub/Sub server is zetta. There will also be Integration Testing to make sure that data storage and the pub/sub server are passing messages as expected.

4 History/Statistics subsystem

Modules involved: Data Storage, Statistics, Pub/Sub Server, and User Interface **Implementation status:** unimplemented

4.0.1 Future tests:

Unit tests for data-storage, statistics and the user interface. For data-storage this will include unit tests to verify statistical/historical data is stored and timestamped correctly. For statistics this will include unit testing for verification of getting the right data from queries, as well as any mathematics done by the statistics module, e.g., min, max, avg, mean etc. The user-interface will need unit-testing to make sure that graphs are being shown as specified, or statistics about the patient are indeed correct. Other tests will include integration testing in the future to make sure everything is communicating as expected. There will also be functionality testing to be sure that the History/Statistics subsystem meets the SRS accurately.

5 User Management Subsystem

Modules involved User Management, User Interface **Implementation status:** partly-implemented

5.0.1 User Management

Implementation status: partly-implemented

Primary Actors: users, admin users

Functionality Create, Remove, Update or Delete items related to data

storage, e.g., user info, users, user relationships etc.

5.0.2 Description of current tests

Precondition: The user manager is tested to ensure that arguments passed to the database manager are correctly formatted, i.e., that the json conforms to the database model.

Postcondition: Ensure that intended parameters are added to the database after successful query.

Invariant: Ensure that only intended parameters change within the related managers and database.

Example of when the CRUD test passes

```
Debugger listening on [::]:5858

  UserManager
    database CRUD
      #addUser
        ✓ adds a user to the db (1022ms)
        ✓ passes on caught error due to invalid primary key
      #getUser
        ✓ gets an existing user from the db
        ✓ attempts to get a nonexistent user from the db
      #updateUser
        ✓ update a existing user
        ✓ attempts to updata a non existing user

  6 passing (1s)
```

Figure 4: Mocha output of successful *User Manager* test

User Manager CRUD fail example, due to Cassandra database being down. Note error reports on the console is also piped into a rotating file. Thus systems users will be able to track down errors after they occur.

```

Debugger listening on [::]:5858
warn:   databaseManager initialized in testing mode, using keyspace [test]

  UserManager
    database CRUD
      #addUser
error:   #DatabaseManager: lsAll host(s) tried for query failed. First host tried, 127.0.0.21:904
2: Error: connect ECONNREFUSED 127.0.0.21:9042. See innerErrors.
      1) adds a user to the db
      2) passes on caught error due to invalid primary key
      #getUser
      3) gets an existing user from the db
      4) attempts to get an nonexistent user from the db
      #updateUser
      5) update a existing user
error:   #databaseManager#try max fails encountered. TODO: notify tech support
error:   #databaseManager#try failed, global fail count: 1
      6) attempts to updata a non existing user

  0 passing (12s)
  6 failing

  1) UserManager database CRUD #addUser adds a user to the db:
     Error: Timeout of 2000ms exceeded. For async tests and hooks, ensure "done()" is called; if
returning a Promise, ensure it resolves.

  2) UserManager database CRUD #addUser passes on caught error due to invalid primary key:
     Error: Timeout of 2000ms exceeded. For async tests and hooks, ensure "done()" is called; if
returning a Promise, ensure it resolves.

  3) UserManager database CRUD #getUser gets an existing user from the db:
     Error: Timeout of 2000ms exceeded. For async tests and hooks, ensure "done()" is called; if
returning a Promise, ensure it resolves.

  4) UserManager database CRUD #getUser attempts to get an nonexistent user from the db:
     Error: Timeout of 2000ms exceeded. For async tests and hooks, ensure "done()" is called; if
returning a Promise, ensure it resolves.

```

Figure 5: Mocha output of unsuccessful *User Manager* test, due to database connection failure

5.0.3 User Interface

Implementation status: partly-implemented

Primary Actors: non-users

Functionality Type in details and register to become a new user. There is only local validation and forms currently implemented. Client-server communication for registration and user management is not yet implemented.

5.0.4 Description of current tests

There are unit tests for each input that test the validation of those inputs. So there are tests for email, passwords, usernames etc. All to make sure that the validation is correct.

Following is the code for each unit test to show what is tested.

```
@Test
public void enterAgeTest() throws Exception
{
    int[] ages = {-3, 0, 6, 23, 65, 150, 4, 12, -4, -5};
    for(int i = 0; i < ages.length; i++)
    {
        if(ages[i] < 0 || ages[i] > 140)
        {
            throw new Exception();
        }
    }
}
```

```
@Test
public void verificationTestLoginDetails() throws Exception
{
    String[] emails = {"Bob@gmail.com", "Carl@yahoo.co.za", "lisa.com", "mina@hotmail", "tanish", "eric123@yknott.com", "trish@mail.c1", ".co.za@charles", "@.co.za", "co"};
    String[] passwords = {"123", "hello", "youaretheone", "IamGr00t", "Ihiiamatest", "IshouldnWork", "Ishouldnot", "ISHOULDNOWORK2", "ThisIs123", " "};

    for(int i = 0; i < emails.length; i++)
    {
        verificationTestLoginDetails(emails[i], passwords[i]);
    }
}
```

5.0.5 Future tests:

Integration tests that verify client-server communication is happening correctly. Functionality Tests to confirm the User Management subsystem adheres to the requirements.

```

public void verificationTestLoginDetails(String email, String password) throws Exception
{
    String mail = email, pass = password;

    if(email.length() < 1 || password.length() < 6)
    {
        throw new Exception();
    }
    else if(!email.contains("@") || (!password.contains("1") && !password.contains("2") && !password.contains("3") && !password.contains("4") && !password.contains("5")
        && !password.contains("6") && !password.contains("7") && !password.contains("8") && !password.contains("9") && !password.contains("0")))
    {
        throw new Exception();
    }
    else if(!email.contains(".co.za") && !email.contains(".com") || password.toLowerCase().equals(password) || password.toUpperCase().equals(password))
    {
        throw new Exception();
    }
}

```

```

@Test
public void verificationTestRegPatient1() throws Exception
{
    String[] emails = {"Bob@gmail.com", "Carl@yahoo.co.za", "lisa.com", "mina@hotmail", "tanith", "eric123@yknott.com", "trish@mail.cl", ".co.za@charles", "8.co.za", "co"};
    String[] passwords = {"123", "hello", "youaretheone", "IamGr00t", "lhiianatest", "IshouldW0rk", "Ishouldnot", "ISHOULDN0TW0RK2", "ThisIs123", " "};
    String[] confirmPass = {"123", "hello", "youaretheone", "IamGr00t", "lhiianatest", "IshouldW0rk", "Ishouldnot", "ISHOULDN0TW0RK2", "ThisIs124", " "};

    for(int i = 0; i < 10; i++)
    {
        verificationTestLoginDetails(emails[i], passwords[i]);

        if(!passwords[i].equals(confirmPass[i]))
        {
            throw new Exception();
        }
    }
}

```

6 Notification Subsystem

Modules involved: Pub/Sub Server, Notification, and User Interface

Implementation status: unimplemented

6.0.1 Future tests:

Unit tests for each individual module as well as integration tests to be sure messages are being passed properly between modules.


```

@Test
public void pickUserNameTest() throws Exception
{
    String[] existing = {"carl123", "bob6", "Juan du Eggz", "Jamie Bob", "George", "Sally", "Mick", "John", "1Address", "Bob"};
    String[] tested = {"carl13", "bb6", "Juan du Eggz", "Jamie Bob", "George", "Sally", "Mik", "John", "1Address", "Bo"};

    for(int i = 0; i < existing.length; i++)
    {
        for(int j = 0; j < tested.length; j++)
        {
            if(existing[i].equals(tested[j]))
            {
                throw new Exception();
            }
        }
    }
}

```

7 Advice Subsystem

Modules involved: Advice and User Interface

Implementation status: unimplemented

7.0.1 Future tests:

Unit tests for each individual module as well as integration tests to be sure messages are being passed properly between modules. This is made easier because both of these modules are local to the application.

8 Other future tests based on non-functional requirements

8.1 Usability test

A full usability test will be conducted with at least 6 people to determine how usable the UI on the app is. Questionnaires and surveys will be used to determine where the fault lies and then improvements can be made.

8.2 Accessibility test

Determining if the application gives affordance to people with disabilities is important, especially considering who the application is purposed for, the bedridden or very sick patients. It also allows a greater number of people to be comfortable with the app allowing more users.

9 Platform compatibility test (future test)

Platform compatibility will be tested, so that we know devices of different screen sizes and different android versions work correctly with our application.