Testing Policy

FOR

Ninshiki

Version 0.1.0 - Demo #2 Draft

A detailed guide for the testing of the software

Compiled By

The Software Sharks Team

Coetzer MJ
Bester TJ
Orrie O
Bekker L
Lew J
Matodzi MC

Department of Computer Science
The University of Pretoria

For

Bramhope International School of Inovation

Department of Computer Science, University of Pretoria

https://github.com/OrishaOrrie/SoftwareSharks

This document was drafted under the supervision of involved lecturers according to the assessment guidelines of the final year Computer Science module: COS 301 - Software Engineering, presented by the Department of Computer Science in the faculty of Engineering, Built Environment and Information Technology at the University of Pretoria during the first semester of the year 2018

First release, April 2018

Contents

1 Introduction

This document describes the testing procedures used for the different subsystems of Ninshiki ("the system") from Software Sharks. The different subsystems are namely:

- Mobile App (Android)
- Web App (Angular)
- Backend (NodeJS and Python)

Following this, the specific unit tests written for each subsystem will be described for the respective subsystem, as well as the links to the location of test reports.

2 Testing procedures

2.1 General Testing Procedure

The development of the system follows the principles of Test-Driven Development. The general testing process is as follows:

1. For each use case, a test function is written that acts as a user completing a task.
2. Initially, all test functions fail. If they don't, they are rewritten so that they fail.
3. Starting from the highest priority use cases, the functional classes and programs that solve each use case are implemented, so that the respective test functions succeed.
4. This is repeated until all functions are implemented, all use cases are solved, and all test functions pass successfully.

This procedure is followed when writing test functions and test cases for each subsystem.

2.2 Android specific testing

The Android testing procedure arranges the tests into three categories:

1. Small Tests: These comprise 70% of the overall tests and test each major component's functionality.
2. Medium Tests: These comprise 20% of the overall tests and integrate several components. They are run on real devices or emulators.
3. Large Tests: These comprise 10% of the overall tests and test integration and UI by running a UI workflow.

Small Tests are conducted in the development environment (Android Studio) and are done while functions are developed to ensure that certain functions work after others are added.

Medium Tests are conducted in the emulator, to make sure that different components work and interface together as expected. Different screen sizes and OS versions are emulated to ensure functionality across different devices.

Large Tests are conducted also in the emulator or a real device to ensure that the UI workflow is optimal and that the use cases are met. Unit tests are written using Espresso.
Android Studio provides all the tools for each category of testing.

2.3 Angular Testing Procedure

There are two tools used in the Angular testing procedure: Jasmine and Karma.

Jasmine is a testing framework that allows for test functions to be written in Javascript that supports Test Driven Development. Jasmine allows for functions to be written in a low-level, easy-to-understand format. Multiple Javascript files can be run by a single Jasmine HTML file.

Karma integrates with Angular and Jasmine in that it runs a separate browser with all Jasmine tests running. This browser automatically updates after changes are made in the development environment, re-running the Jasmine tests.

These two tools, when used with Angular make automated testing simple and efficient.

2.4 NodeJS Testing Procedure

The framework used for NodeJS testing is Mocha. Mocha allows test functions to be written in JavaScript and run using a command.

Similar to Karma, nodemon is a tool that ensures as changes are made to the code, the Mocha tests are run again automatically.

2.5 Python Testing Procedure

Python by default includes a test module that allows for test functions to be written in the same file as the implementation. This is sufficient for function testing.

However for test cases and more complex tests, “pytest” is a standard tool that is full-featured and simple to use. Test files are written and then run using a command. Note that this is process.

A manual process is used in the system for the reason that Python is used to develop the image processing and classification aspect of the system. Testing and running this process is resource-heavy, especially when training the model. For this reason, testing is done manually after a certain milestone is reached instead of automatically after code is changed.

3 Unit Tests

This section describes the general format of unit tests and test reports for the respective subsystems.

3.1 Android Unit Tests

This section will be completed at a later date.

3.2 Angular Unit Tests

These unit tests ensure that the functionality and UI workflow of the web app is working.

The Angular subsystem is made up of components which each serve a function or provide a feature. For example, the Home component serves as a landing page of the web app, allowing users to quickly access the app's main pages.

For each component, a set of test functions are written and specified in the component's spec file. For example, the home component's spec file is named: home.component.spec.ts.

When 'ng test' is run, Angular uses Karma to run all spec files (for all the components) and displays all the results in a new browser window. The page in this browser window is saved as an HTML page called "Karma.html" in the "tests" directory, located here:

`https://github.com/OrishaOrrie/SoftwareSharks/tree/web/Application%20Source/tests`

The following use cases and descriptions of their respective test functions are shown below:

1. Use Case: Capture an Image.
   (a) Test Function: If the "Webcam" button is clicked, then check if the webcam is opened.
2. Use Case: Select an Image from Storage
   (a) Test Function: If the "Upload" button is clicked, then check if the file explorer opens.
   (b) Test Function: Check that an image file is selected after the file explorer closes.
3. Use Case: Send Request
   (a) Test Function: Once the image is uploaded, check that a loading bar is displayed indicating a wait for response.

(b) Test Function: Once the image is uploaded, check that a "Upload Successful" message is displayed.

4. Use Case: Receive Response

   (a) Test Function: Check that once an HTTP OK Status response is received, a list of classification results are returned

5. Use Case: Weight Analysis

   (a) Test Function: When a field is touched or dirtied, check that the "Item Weight" text is updated dynamically.

   (b) Test Function: Check that only numerical values can be entered in each field.

   (c) Test Function: Use mock values (float, integer, large, small) to test if the "Item Weight" calculation is correct.

   (d) Test Function: Check that negative values are not submittable.

6. Use Case: Contact Company

   (a) Test Function: After the email field is touched or dirtied, check that the validation function is run.

   (b) Test Function: When the submit button is clicked, check if the email send function is run.

   (c) Test Function: When the email function is run, check if an Http request is sent.

3.3 NodeJS Unit Tests

These unit tests ensure that the backend interface with the web app, the mobile app, and the Python image classification program as well as the implementation of the email service and image processing is working correctly.

Mocha test files are written testing the functionality of each written Javascript file used for the server. The command 'npm test' is used to run each test file, with the test results being displayed and saved in the same "tests" directory.