# Testing Report

# *Dekū*

**Client**
Shawn Squire

**Team 6**
Boris Boiko
Raymond Chan
Gilbert Kuo
Andrew Naviasky
Jeremy Neal

4/30/2014

# Table of Contents

## 1. Introduction

### 1.1 Purpose of This Document

The purpose of this document is to document the testing format and practices used during the development of the Dekū application. The audience for this document is the engineering team developing Dekū**,** any future maintainers of the project, and the customer. Shawn Squire.

In this document, the format for testing, as well as the process in which they are run, are detailed. This is documented separately for each context in which testing is required.

### 1.2 References

1. System Requirements Specification
2. Behavior-Driven Development: http://en.wikipedia.org/wiki/Behavior-driven_development

## 2. Testing Process

### 2.1 Description

For the development of Dekū, the engineering team is employing a behavior-driven development (BDD) cycle. A full description of this process is available on Wikipedia. This practice allows the developers to ensure that code performs as expectedly immediately, and saves time by not requiring the writing of tests after code is written, which can often result in incomplete and/or incorrect tests.

For this process, the team is using the use cases in the SRS as a model for defining our methods. Once we've determined the structure of our function, as well as the inputs and outputs, we define our tests, and use those tests to build our functions.

There are two systems used for testing in this application. Code is written in JavaScript and Python, and the systems for testing are different.

For JavaScript, we will be using Jasmine (http://jasmine.github.io/2.0/introduction.html) as our testing framework, with Grunt (http://gruntjs.com) as a test runner. Jasmine is a behavior-driven test framework, where tests are written to define a unit, the unit is then written, and the test is used to verify.

For Python, we will be using the built-in unittests module, with nose (https://nose.readthedocs.org/en/latest/) as a test runner. This is a less opinionated test framework, but will work for our needs.

HTML and CSS does not require testing. Any errors in those technologies will be covered in the code inspection process.

While we had the plan to use BDD from the start, that did not happen. Much of the initial code was written with an incomplete understanding of the mechanisms at work, especially with BackboneJS, one of the main components of the interface. Additionally, the API was not unit tested initially. Tests have been written to verify API endpoints, as well as to verify model instantiation in the Backbone interface.

Testing the interface did not begin in earnest until much of the basic interface was already implemented. This is due largely to a lack of data being used. The majority of the design work was in managing the application state, without a user or any data being passed. However, methods involving data creation, user authentication, etc. were tested with Jasmine.

Due to issues in the way Backbone verifies templates, our views could not be tested using Jasmine. However, these views are tested manually and shown to work as expected.

### 2.2 Testing Sessions

Discrete testing sessions were not used. Tests were written in either Python or JavaScript depending on what needed to be tested. Otherwise, manual testing via the interface or cURL was used to validate code prior to commits.

### 2.3 Impressions of the Process

Test driven development, when used properly and efficiently, is a very powerful methodology. The ability to know your code works as designed, and to know when and why it is not, is immensely helpful. Feedback is far more productive and meaningful, as opposed to errant print statements and best guesses.

There are downsides. While TDD prevents programmers from writing poor tests, it also prevents them from implementing features. In many cases, the code for this application was very straightforward. Tests were useful for verifying proper return types, but the structure of the application is very opinionated, and getting the wrong return values was very unlikely given this rigidity.

Overall, our process has been below the expectations that we've set. As previously stated, we had an incomplete understanding of how to structure the application down to a low level. This made testing difficult, as the values for which we needed to test were very specific in most cases. However, the testing that we have done, regardless of the process involved, has made troubleshooting much easier. The quality of the program is much improved, with code being cleaner and easier to read. It also forces a certain level of convention in API calls. They follow a fairly rigid conditional structure, which is guided by the expected return values under certain conditions.

## 3. Test Results

**Appendix A - Team Review Sign-off**

This section of the document verifies that each team member has reviewed and agreed to the terms set forth in this document. This includes the testing frameworks used, the process in which they are to be written and run, and any issue thereof.

Boris Boiko Signature _____ Date: ___/___/___

Comments:

Raymond Chan Signature _____ Date: ___/___/___

Comments:

Gilbert Kuo Signature _____ Date: ___/___/___

Comments:

Andrew Naviaski Signature _____ Date: ___/___/___

Comments:

Jeremy Neal Signature _____ Date: ___/___/___

Comments:

## Appendix B - Document Contributions

Jeremy Neal - Initial skeleton of document, including sections 1.1, 2.1, Appendix A