Automated Testing Framework for Enketo

Final Report

By : KMP
Kyle Sheeley
Matendo Reguma
Patrick McCardle

CSCI 362 Software Engineering
Dr. Jim Bowring
College of Charleston

Table of Contents

Chapter 0

Introduction

This is a final report produced by the team KMP's experience while building an automated testing framework for the open source H/FOSS project . KMP is a group of Computer Science juniors and seniors at The College of Charleston. The team KMP is composed of three persons, one junior Kyle Sheeley and two seniors, Patrick McCardle and Matendo Reguma.

We were given the task of creating a twenty-five test case automatic testing framework for an open-source humanitarianism project of our choosing. A list of possible open source project candidates was given to us by Dr Bowring. Also, prior to choosing our project, there were few things that we had to learn. For example, we had minimal experience in writing scripts as well as Git commands. In addition, we were to record our progress along the way and break each record up into chapters that specify what exactly were our goals for the project at the time.

There were several free open source project candidates from which to choose. However, the team KMP picked three open source project such as OpenMRS, Sahana Foundation and Martus. These three open source projects had dependency issues. Therefore, we ultimately decided on Enketo as our final project. This report is organized in several chapters which are in chronological order, starting from picking a project all the way up through injecting faults in a fully automated testing framework.

Chapter 1

Picking a project

We chose a humanitarian project that stemmed from the KoboToolbox project, which some of you may be aware of. KoBoToolbox is a free open-source tool for mobile data collection, available to all. It allows you to collect data in the field using mobile devices such as mobile phones or tablets, as well as with paper or computers.

It is being continuously improved and optimised particularly for the use of humanitarian actors in emergencies and difficult field environments, in support of needs assessments, monitoring and other data collection activities.

What makes Enketo better than it's competition?

Surveys deployed with Enketo:

- work offline
- have beautiful themes and widgets
- are printer-friendly
- can use very powerful skip and validation logic
- run on any device, mobile or desktop, as long as it has a fairly modern browser.

Chapter 2

Testing Progress

Since none of us had ever dealt with a project this big before, we were a little lost as to where to look to find built-in tests, or even what the difference between component and unit testing was. We began diving into the code and looking for provided test cases. It is a good thing that enketo is so organized and came with nice comments, as it didn't take us long to locate the tests. We discovered that since the project is written in NodeJS, there are many components to running the built-in tests. The first is Grunt.

Grunt

Built on top of Node.js, Grunt is a task-based command-line tool that speeds up workflows by reducing the effort required to prepare assets for production. It does this by wrapping up jobs into tasks that are compiled automatically as you go along. Basically, you can use Grunt on most tasks that you consider to be Grunt work and would normally have to manually configure and run yourself. Grunt can handle most things you want to have within your JavaScript workflow, from minifying to concatenating JavaScript as well as running Mocha tests automatically what is pretty handy for our use case.

Mocha

Mocha is a feature-rich JavaScript test framework running on Node.js and the browser, making asynchronous testing simple and fun. It does just about everything a JavaScript developer needs, and yet remains customisable enough to support both behaviour-driven and test-driven development styles. Mocha tests run serially, allowing for flexible and accurate reporting, with the ability to map uncaught exceptions to the correct test cases.

Chai

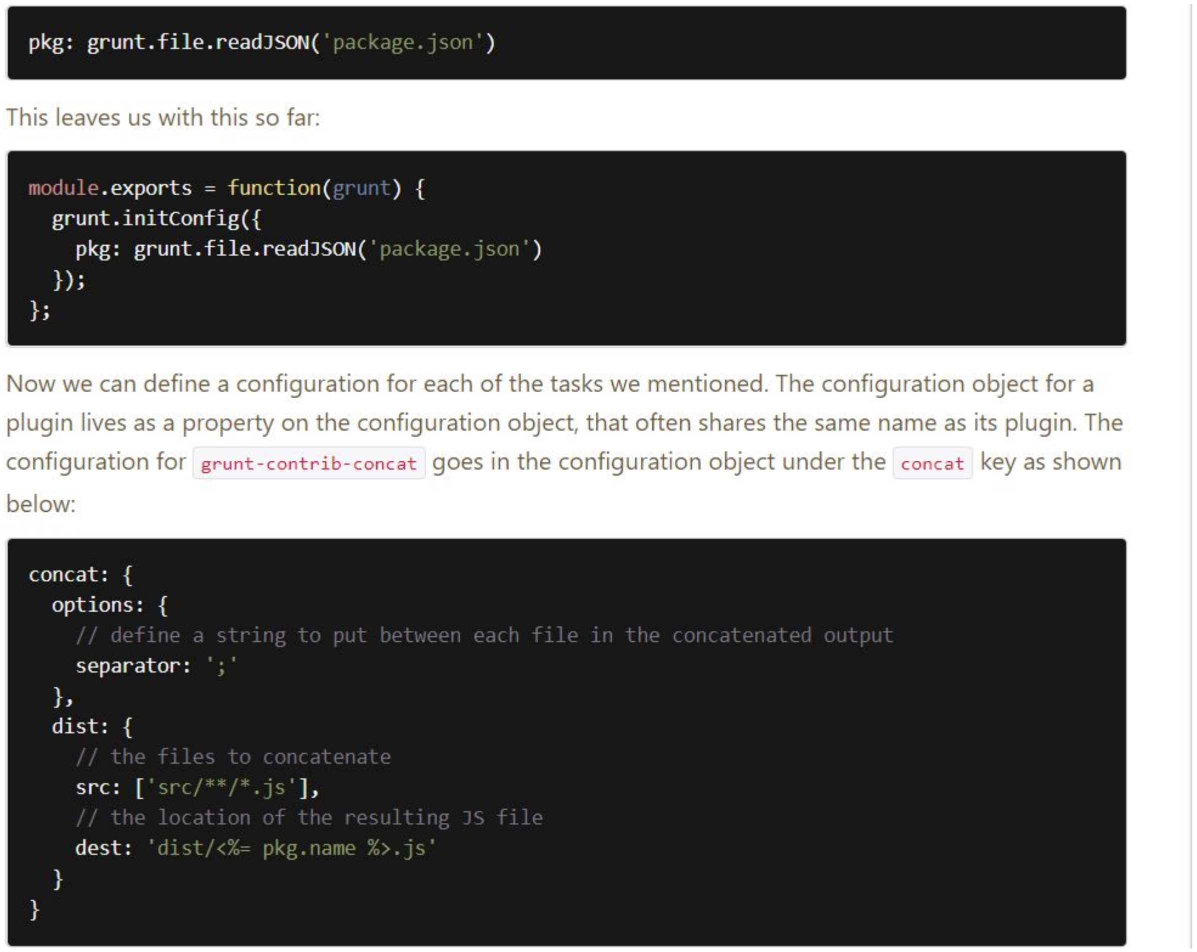
Chai is a BDD / TDD assertion library for node and the browser that can be delightfully paired with any javascript testing framework.

Chai aims to be an expressive and easily approachable way to write assertions for JavaScript project testing. Developers have already started expanding on Chai’s available language through plugins such as spies, mocks/stubs, and jQuery support.

Interesting is that the motivation for Chai came about with the release of Mocha. At the time, there was no apparent assertion library to pair with it that would allow for the same assertions to be used on both server and client with the inherent simplicity that Mocha provides.

The project uses the automated task runner ‘Grunt”, that instantiates tasks within a Gruntfile that is located in the project root. From here, we can store the project settings from the `package.json` file into the `pkg` property. This allows us to refer to the values of properties within our `package.json` file.

Short grunt example :



```
pkg: grunt.file.readJSON('package.json')
```

This leaves us with this so far:

```
module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json')
  });
};
```

Now we can define a configuration for each of the tasks we mentioned. The configuration object for a plugin lives as a property on the configuration object, that often shares the same name as its plugin. The configuration for `grunt-contrib-concat` goes in the configuration object under the `concat` key as shown below:

```
concat: {
  options: {
    // define a string to put between each file in the concatenated output
    separator: ';'
  },
  dist: {
    // the files to concatenate
    src: ['src/**/*.js'],
    // the location of the resulting JS file
    dest: 'dist/<%= pkg.name %>.js'
  }
}
```

Mocha example :

We wrote a script to run the grunt tests from the project root, and got results from almost 350 provided tests. 281 passing, and 68 failing tests was the most successful-rated fully implemented test results we could come with. After looking up the paths to the tests the first notion I got was to look in package.json, which directed me to the /app and /test directories that did not have clear tests that we could model from.

As the screenshot shows along with the failed tests comes a description containing the error and the path to the test. We began looking into the node_modules folder which pointed us to a library containing "supertest" directory. When we cat some of the tests they looked a little foreign at first but we are able to pick out methods and find keywords that point us in the right direction. We will compose 25 tests on at least 5 different components. We will test components such as server/client connection, user authentication, GET/SET requests, and the cached instances. At the top level of the tests directory we will have a bash script that will run all tests.

Tests

Test number : 1

- Requirement being tested : new server/client connection
- Component : new submission
- Method : isNew() check
- Inputs : undefined
- Expected outcome : rejected

Test number : 2

- Requirement being tested : new server/client connection
- Component : new submission
- Method : isNew() check
- Inputs : null
- Expected outcome : rejected

Test number : 3

- Requirement being tested : new server/client connection
- Component : new submission
- Method : isNew() check
- Inputs : false
- Expected outcome : rejected

Test number : 4

- Requirement being tested : new server/client connection
- Component : new submission
- Method : isNew() check
- Inputs : -1

- Expected outcome : rejected

Test number : 5

- Requirement being tested : new server/client connection
- Component : new submission
- Method : isNew() check
- Inputs : 0
- Expected outcome : rejected

Test Recording Procedures

The testing driver for our testing suite will save the results of every run, including the pass/fail responses. The test file will include the date it was run, and an ID for the test. The results will be put together and displayed in an HTML web browser.

Requirements Traceability

Validation test that system requirements are met

Requirement test (sending in a node test to make sure node.js is installed properly)

Hardware and Software

Ubuntu 16.04 or later is required for this Docker container

Chapter 3

Testing Framework procedure

We began looking in the package.json file to understand what is used to test the built-in tests, we came across the script that shows when "test" or "grunt test" is entered, to run the default tests.

Package.json

To run tests:

```
"scripts": { "start": "node .", "test": "grunt test", "postinstall": "grunt"
```

```
npm run test -> grunt test
```

Moving into the Gruntfile.js we got an idea of how the tests were set up, and found out what exactly the Grunt tasks do, where they are located, and how the tasks find the information they need to perform the tests.

Original grunt test

Grunt has task instantiations, with the name of task being the first argument, and the taskList being the second

This example alias task defines a "default" task whereby the "jshint", "qunit", "concat" and "uglify" tasks are run automatically if Grunt is executed without specifying any tasks:

EXAMPLE

```
grunt.registerTask('default', ['jshint', 'qunit', 'concat', 'uglify'])
```

Original from project

```
grunt.registerTask( 'test-and-build', [ 'env:test', 'mochaTest:all', 'karma:headless',
'env:production', 'default' ] );
```

NEW

```
grunt.registerTask( 'test-and-build-kmp', [ 'env:test', 'mochaTest:kmp', 'karma:headless',
'env:production', 'default' ] );
```

we will be focusing on the mochaTest task for these tests, as mocha is what is used to test the server connection. Below is an example of the what the mochaTest task looks like in the Gruntfile, and without getting too in depth into mocha, we are able to replicate and build a new framework using the old one as a model.

Original mochaTest from project

```
// test server JS
mochaTest: {
    all: {
        options: {
            reporter: 'dot'
        },
        src: [ 'test/server/**/*.spec.js' ]
    },
    account: {
        src: [ 'test/server/account-*.spec.js' ]
    }
},
```

UPDATED

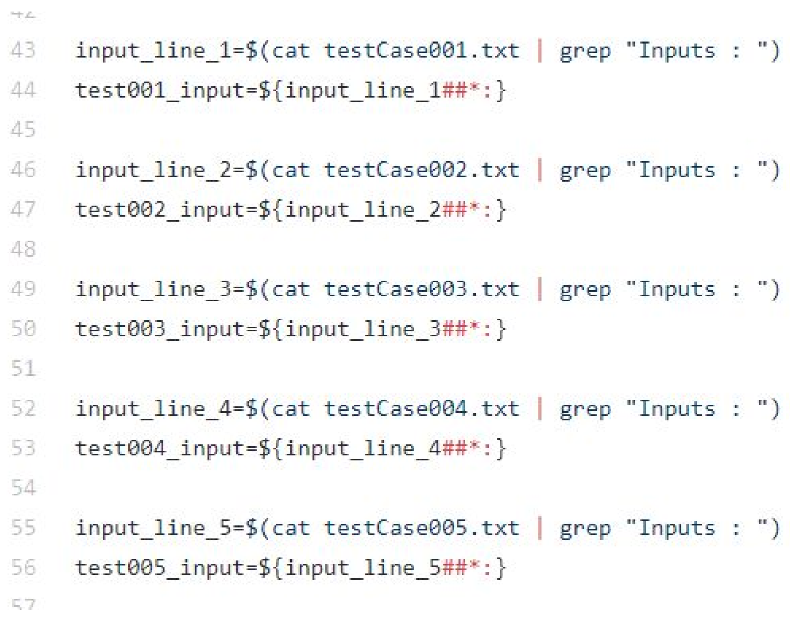
```
// test server JS
mochaTest: {
    all: {
        options: {
            reporter: 'dot'
        },
        src: [ 'test/server/**/*.spec.js' ]
    },
    account: {
        src: [ 'test/server/account-*.spec.js' ]
    },
    kmp: {
        src: [ 'test/server/kmp-*.spec.js' ]
    }
},
```

Chapter 4

Testing framework

As specified in the instructions, we have all twenty five of our test cases written in in the testCases folder that have the component being tested, the inputs to the test, the method being tested, and the expected output. As well as a working script that reads through the test cases and generates and runs the test case, pipes the output into an html file and displays the test results in a web browser.

In our testcase.txt file, we can see a model of what the method being tested will look like to give someone who is unfamiliar with the project a better understanding of how to write a test case if they wish to. From this testcase.txt file, we found a good way to extract information using the ‘grep’ command provided to us by linux. In the simplest terms, grep (global regular expression print) will search input files for a search string, and print the lines that match it. Beginning at the first line in the file, grep copies a line into a buffer, compares it against the search string, and if the comparison passes, prints the line to the screen. Grep will repeat this process until the file runs out of lines. Notice that nowhere in this process does grep store lines or change lines.

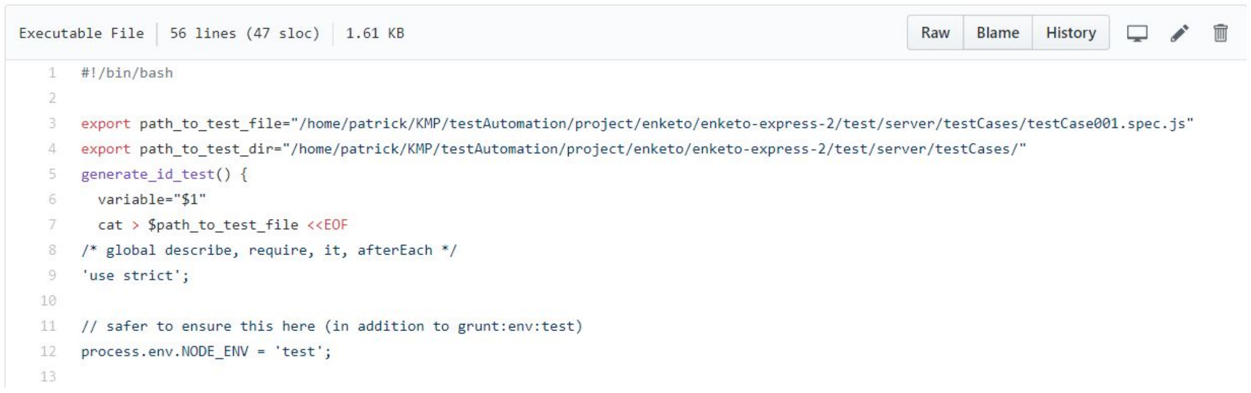


```
input_line_1=$(cat testCase001.txt | grep "Inputs : ")
test001_input=${input_line_1##*:}

input_line_2=$(cat testCase002.txt | grep "Inputs : ")
test002_input=${input_line_2##*:}

input_line_3=$(cat testCase003.txt | grep "Inputs : ")
test003_input=${input_line_3##*:}

input_line_4=$(cat testCase004.txt | grep "Inputs : ")
test004_input=${input_line_4##*:}

input_line_5=$(cat testCase005.txt | grep "Inputs : ")
test005_input=${input_line_5##*:}
```

From the picture above, you can see how we used grep to get the inputs from the testCase.txt files, we categorized our tests into groups based on what method/component was being tested. For our first 5 tests, they are our “id” tests, so we store them in an “id test” array. The syntax for creating an array in bash is simply array=(item1 item2 item3 etc…)

After all the test inputs are parsed and grouped a for loop iterates through the specified array, calls a function ‘get_new_x_path’ at the beginning of each iteration. This self-created function gives a brand new path to where the eventual test cases will be located. The function takes an already declared variable ‘path_to_test_dir’ and creates a unique file/path that is named after the test input when called, this makes it easy to separate tests and see what the input is without having to actually go into the file. Once each input has it’s own unique file path we are ready to generate the tests, this is done by calling a function available to us in our “something.sh” script that we source into at the beginning of this script.

The ‘generate_id_test’ “$i” line is the bread and butter of this framework. Each component being tested requires a whole slew of (different) global variables, assertions, requirements and such. Since these are necessary to keep in the testing files we found a very useful use of the ‘cat’ command here, by using cat > name_of_file <<EOF. Doing this allows for everything underneath the cat command (and before the 2nd EOF) to be copied and pasted into a new file in a completely different directory. This made it easy to create new test files with all the required global variables and such. The ‘generate_id_test’ “$i” creates the new file and inside the method being tested, replaces a $variable parameter, with the “$i” parameter that is passed in from the id_inputs array.



```
#!/bin/bash

export path_to_test_file="/home/patrick/KMP/testAutomation/project/enketo/enketo-express-2/test/server/testCases/testCase001.spec.js"
export path_to_test_dir="/home/patrick/KMP/testAutomation/project/enketo/enketo-express-2/test/server/testCases/"
generate_id_test() {
  variable="$1"
  cat > $path_to_test_file <<EOF
/* global describe, require, it, afterEach */
'use strict';

// safer to ensure this here (in addition to grunt:env:test)
process.env.NODE_ENV = 'test';

```

After the cat > $path_to_test_file <<EOF line, everything below it is copied into a new unique file until the 2nd EOF is reached, at which point the new unique file is ended, and you may go about scripting or end the file if you choose.

Here you can see that the method in the “id” testCase.txt files matches the method created in the file, with the $variable in place of the “$i” which is in place of the id_test inputs array. The 2nd EOF stop the creation of the new file. The new/overwritten test files will sit patiently after being created, until a variable named TEST_RESULTS will call the command to the run the newly created tests, and store the output of the test results into a time-stamped html file.

This procedure of ....

1) parse Inputs line of testCase.txt
2) collect inputs and store into groups (arrays)
3) Iterate through the groups, and with each iteration it creates a new path for the given input by name, and runs the ‘generate_x_tests’ script that creates the new files and puts them in the correct path.
4) run the new tests, and store the results in a time-stamped html file …

is repeated throughout the script for each method being tested. With each new ‘generate_x_tests’ script containing the correct global variables and requirements, it is easy to keep all requirements for the different tests and it is easy to create different test files within the same script.

Chapter 5

Fault Injections

For our fault injections, we were trying to see if we could get at least 5 test cases to fail without having ALL of the test cases fail.

An example of how we caused a test to fail without breaking the code. Here we sent in the wrong error code, it was expecting a 400 error but instead we injected a 404 error and caused the test to fail.

Results:

We were able to successfully inject five different faults into the code in order to tests the robustness of the system to handle error capabilities. The aim was to simulate a scenario in which tests were constantly run by another developer to find any weaknesses or bugs in the overall system or program. Some of the tests failed and this demonstrated the importance of methodically testing a software product to catch any hidden minor or major bugs.

Chapter 6

Experiences

Throughout the semester we felt that this project had some ups and downs. We had been learning in class all along how hard it is to guess how long a task or deliverable will take in software engineering, and we believe we got first hand experience of exactly what that feels like. There were times when we would knockout tasks or fix bugs and felt like we were ahead of the game, only to realize something doesn't work later on, and we were then behind trying to catch up to our deadline.

We do believe though, that team projects are just naturally tougher because of the requirement to meet up, work together and discuss and come up with ideas in a collaborative manner. Still we feel that this semester-long project has taught us many things. None of us had dealt with a project like this before, so it took us a while to just be comfortable with the file structure of the project.

We were able to learn about many things that are very real-world applicable, like docker, grunt, and mocha. We learned about containers which we talked about in class, and saw examples of good software engineering, i.e a lot of tests, good comments and clear directions. We probably learned the most about bash scripting, as the most major component of the project was the script.