Tardigrades

Jason Boulware
Austin Hollis
Jeffrey Williams
Rowen Loucks

Table of Contents

Chapter 1 (Deliverable 1) **2**

Chapter 2 (Deliverable 2) **3**

Chapter 3 (Deliverable 3) **4**

Chapter 4 (Deliverable 4) **6**

Chapter 5 (Deliverable 5) **7**

Chapter 1 (Deliverable 1)

Our team has decided to use Amara for our Test Driven Development project for our Software Engineering (CSCI 362) class. We chose this assignment because of the accessibility factor and the amount of potential it has to help millions of people. We also chose this project because it is written in Python, a language that most of our team members know fluently, which makes it easier to read, test and debug the code. In the beginning we had issues getting the code to simply build due to the poor guidelines in the Amara ReadMe.txt file.

One of the biggest flaws we have seen is the fact that the source code has repositories inside of a repository. The next thing we discovered which has been a pain to get through has been the myriad of errors we have encountered while trying to build it. The ReadMe.txt file does not state what version of Python is needed so we chose to go with Python 2.7. On top of not knowing exactly which Python version is needed the ReadMe.txt file did not state that pip is needed and it did not state that django is also needed. If you encounter any problems while trying to get the project built try executing 'which pip' and then use the full path of pip (ours happened to be in /usr/bin/pip) to install django. The project involves docker-compose which is annoying in and of itself to understand and get working properly but the ReadMe.txt file has a link to the docker-compose web-page to follow. It is also worth mentioning that we had to change the docker-compose.yml file at the top from "3" to "2" to get it to work as well.

After we fixed all the errors we were encountering and had all the necessary components we used the /bin/dev build to build the environment and docker container. That same file also has a test command that will run their test cases which happened to be over one thousand tests with only 3 being skipped and the program failing one test with the message "9 != 13".

```
                                    FAILURES
SubtitleTypesTest.test_subtitle_list self =
<tests.subtitles.test_types.SubtitleTypesTest testMethod=test_subtitle_list>
def test_subtitle_list(self):l = SubtitleFormatListClass(ParserList,
GeneratorList)>
self.assertEqual(len(l), 13 E
AssertionError: 9 != 13
/var/run/amara/tests/subtitles/test_types.py:32: AssertionError
1 failed, 1121 passed, 3 skipped in 243.29 seconds
```

In short, the README file could use with some updating and noting the specific requirements of Python, Django, Docker, etc. to make building and getting the program in a working state a lot more simple for others to help assist and improve the program.

Chapter 2 (Deliverable 2)

Testing Process

We want to begin by testing some of the most common components of Amara. Since Amara allows users to provide captions, subtitles, and translated videos, we felt some of the basic tests should include each of these features. We will begin with simpler tests and eventually tackle more difficult portions later.

Test Recording Procedures

We have decided to have our script run and compare then post whether or not the test case passed or failed in an HTML file that will automatically launch once the script is done running.

Hardware and Software Requirements

The hardware requirement includes having a computer and having Amara installed and built on the machine. The software requirements include having the following installed onto the computer Python 2.7, pip, django, docker-compose, the bleach Python library (bleach), and the lxml Python library (lxml).

Constraints

Some of the constraints affecting the testing process include verifying data we are testing is as close to being "realistic" as possible. We must assume our test cases will be accurate.

Unit Tests

· *Check Appropriate Video File*

Given a filename plus an appropriate video file extension, test & see if it is appropriate.

· *Check Appropriate Audio File*

Given a filename plus an appropriate audio file extension, test & see if it is appropriate.

· *Check Appropriate MIME Type File*

Given a filename plus an appropriate MIME file extension, test & see if it is appropriate.

· *Check User is Authorized to View Video*

Given two names, test & see if it the name that created the video is the same as the person trying to view the video currently.

· *Escape Ampersands*

Giving text that has '&', replace the '&' with '&' for compatibility with HTML format

Chapter 3 (Deliverable 3)

Introduction:

Our team has begun work on building a testing framework for the Amara project we selected. We've had a fair number of setbacks, from the project's constant updates causing our virtual machine to fail to build the project to the poor documentation for basically any part of the project, but we've found a decent number of methods that should be easily testable given the current state of the project.

Test Cases:

Test ID: 01
Description: Checks for appropriate file extension for torrent files.
Path to file: /project/unisubs/libs/vidscraper/
Component being tested: filetypes.py
Method being tested: isTorrentFilename(file)
Test input: "Torrent.what"
Expected Outcome: False

Test ID: 02
Description: Guesses Extension Type
Path to file: /project/unisubs/libs/vidscraper/
Component being tested: filetypes.py
Method being tested: guessExtension(file_description)
Test input: "video/quicktime"
Expected outcome: .mov

Test ID: 03
Description: Guesses Extension Type
Path to file: /project/unisubs/libs/vidscraper/
Component being tested: filetypes.py
Method being tested: guessExtension(video_description)
Test input: "video/x-matroska"
Expected outcome: .mkv

Test ID: 04
Description: Guesses Mime Type
Path to file: /project/unisubs/libs/vidscraper/
Component being tested: filetypes.py

Method being tested: guessMimeType(filename)
Test input: “test.mov”
Expected outcome: video/quicktime

Test ID: 05
Description: Guesses Mime Type
Path to file: /project/unisubs/libs/vidscraper/
Component being tested: filetypes.py
Method being tested: guessMimeType(filename)
Test input: “test.avi”
Expected outcome: video/x-msvideo

Report:

To start our testing, we are mainly focusing on filetypes.py in order to determine whether a filename and the extension is appropriate for an audio, video and torrent file. Overall, our tests are relatively simple given the scope of the project we selected, but such tests are still useful for big projects like this. These smaller methods are incredibly important to the overall function and security of the complete project.

Chapter 4 (Deliverable 4)

Introductions:

For this week's deliverable, our team had to create 25 test cases and have our script read the test cases files, compare the test results and have the outcome of the comparison open in a HTML file.

Process:

For our framework, we had trouble finding functions to test for since the Amara framework had a myriad of import statements and circular dependencies which became quite an issue when trying to test single methods in the python files of the project code. However, there were two files in particular that had methods for testing for different file types that could be submitted as well as audio file types. We tested those thoroughly since our project deals with accessibility and subtitles for videos. Inside the filetypes folder we tested guessExtension(), guessMimeType(), isTorrent(), isVideoFilename(), isAudioFilename(), isTorrentFilename(), and a few other methods to get our 25 test cases.

For the runAllTests script we ran into some trouble trying to get the python file to run to get the actual results. However, we did make a big improvement from the last deliverable because now we are recording our results with bash and then outputting the results to an HTML file to make it easier to view. The HTML file also automatically launches once the test cases are complete.

Report:

Finding usable test cases has been the hardest challenge for this deliverable as most of the files have a heavy dependency on django and docker imports which have been difficult to get set up to work properly, but we feel confident about getting it complete before the deadline. Scripting has not been as big of an issue as we had originally thought which is a huge positive for us.

Chapter 5 (Deliverable 5)

Introductions:

For this week's deliverable, our team had to do 5 fault injections in the code and test how those fault injections affect our test cases.

Process:

For our fault injections, we used the following methods: fraction_to_milliseconds(str_milli), centiseconds_to_milliseconds(centi), isAllowedFilename(filename) and can_view_activity(user, viewing_user). Inside of these methods were if statements and we would insert a 'not' to get the opposite if it was a single if statement without an else statement. We would change an 'and' to an 'or' and vice versa.

Original	def fraction_to_milliseconds(str_milli):
	  if not str_milli:
	  return 0
	  ...
	  ...

Fault Injection	def fraction_to_milliseconds(str_milli):
	  if not not str_milli:
	  return 0
	  ...
	  ...

Original	def centiseconds_to_milliseconds(centi): return int(centi) * 10 if centi else 0
Fault Injection	def centiseconds_to_milliseconds(centi): return int(centi) * 10 if 0 else centi

Original	def isAllowedFilename(filename):
	  return (isVideoFilename(filename) or
	  isAudioFilename(filename) or
	  isTorrentFilename(filename))

Fault Injection	def isAllowedFilename(filename):
	  return (isVideoFilename(filename) and
	  isAudioFilename(filename) or
	  isTorrentFilename(filename))

Original	def isAllowedFilename(filename):
	  return (isVideoFilename(filename) or
	  isAudioFilename(filename) or
	  isTorrentFilename(filename))

Fault Injection	def isAllowedFilename(filename):
	  return (isVideoFilename(filename) and
	  isAudioFilename(filename) and
	  isTorrentFilename(filename))

Original	def can_view_activity(user, viewing_user):
	  return (user == viewing_user or
	  viewing_user.is_superuser or
	  user.created_by == viewing_user)

Fault Injection	def can_view_activity(user, viewing_user):
	  return (user == viewing_user and
	  viewing_user.is_superuser or
	  user.created_by == viewing_user)

Report:

Upon testing the fault injections it turned out that whichever method we changed the code in would fail while the others would still pass. Injecting those 5 fault injections caused an obvious turn of events in the amount of test cases that passed and the ones that did not pass.