IMPROVING THE CODEBASE OF SIGNBANK

Joshua Goddard

Bachelor of Engineering
Software Engineering

Department of Computing
Macquarie University

September 5, 2016

Supervisor: A/P Steve Cassidy

ACKNOWLEDGMENTS

I would like to acknowledge my supervisor, Steve Cassidy, for proposing this project, for explaining it to me and for helping me with queiries and problems I have had. I would also like to acknowledge the various signbank developers for showing enthusasim for this project and for lending their experience with Signbank to me whenever I have needed it.

STATEMENT OF CANDIDATE

I, Joshua Goddard, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Computing, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment an any academic institution.

Student's Name: Joshua Goddard

Student's Signature: Joshua Goddard (electronic)

Date: September 5, 2016

ABSTRACT

The purpose of this project is to improve the codebase of Signbank, a web-application written in Django that provides information relating to sign languages. The codebase is suffering from a lack of tests, from being out of date (with respect to the latest versions of Python and Django), from lacking documentation, and from containing commponents that should be independent from one another but are not. By the end of this project, the Signbank codebase was improved by writing tests for it, by making components that should be independent from one another independent from one another, by writing documentation, and by upgrading the codebase to the latest versions of Django and Python, versions 1.10 , and 3.5, respectively. The project did fall short in some areas (as is to be expected from a large software project). Some advice is given to future developers on how they can tie up the lose ends left by this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Auslan.org.au and bslsignbank.uc.ac.uk are websites that provide information relating to *Sign Languages*. The principal feature of these websites is a dictionary of *signs*.

Briefly, a *sign* is a combination of hand movements, facial expressions, and arm movements that, together, convey meaning. [1]

Sign language, like its oral counterpart, comes in many varieties. Two such varieties are Australian Sign Language (Auslan) and British Sign Language (BSL). Auslan.org.au provides information relating to Auslan, and bslsignbank.uc.ac.uk provides information relating to BSL.

Auslan.org.au was designed and implemented first, and bslsignbank.uc.ac.uk forked it sometime later. Since this time, the two websites have diverged greatly (with separate developers working on each). In addition to the British and Australian websites, there are a Finnish one and a Dutch one. Both the Finish and the Dutch versions are based on copies of the Australian version's source code. It is estimated that the copies took place two years ago.

The supervisor of this project, Steve Cassidy, wrote the Auslan website in Django, a web-framework written in Python. When Steve wrote Auslan, he used the then most recent version of Django, version 1.6. Since this time, however, Django has moved on to version 1.10. The website has not changed with Django, and as a result is deficient in some aspects. Broadly, the goal of this project is to ameliorate these deficiencies.

The codebase of the Auslan website shall be referred to as the *Signbank codebase* for the remainder of this document.

1.1 Overview of features

The following list contains the features available on the Auslan website:

- A dictionary, with an entry for each sign in Auslan. Each entry in the dictionary contains a definition of the sign, a video of a person signing the sign, linguistic information about the sign, and a link to submit feedback on the sign.

Figure 1.1: The front page of Auslan

Figure 1.2: The front page of BSL

- The dictionary is searchable.
- Information on the deafness, and on the Auslan-speaking community of Australia.
- Resources for learning finger spelling.
- An interface for the administrator of the website to view feedback, and to manage the content of the website.
- A user-login system.

The BSL website, being derived from Auslan, contains similar features.

1.2 Goals of the project

The current state of the Signbank codebase is not good. It uses a version of Django that is several years old, it contains *sections* that should be independent from one another but are not, it lacks documentation, and it lacks tests. The goal of this project, then, is to fix these problems, and thus improve the codebase of Signbank.

The *sections* referred to in the previous paragraph are

- Video
- Dictionary
- Feedback
- Registration
- Pages
- Attachments

These sections will be important for understanding the goals of the project.
What follows is an overview of the goals of the project.

1.2.1 Independent apps

From the perspective of someone who uses the Auslan website, each of the items in the above list can be thought of as a feature. From the Signbank developer's perspective, however, each item should, ideally, be thought of as a pluggable Django application as independent from the other items in the list as is possible. One of the problems with Signbank is that some of the items in the list are tied to other items in the list beyond what is reasonable. The first goal, then, of this project is to take each item as it exists now in the Auslan codebase and to turn it into an independent Django application.

It should then be possible to recreate Signbank by combining all of these indepednent applications together, into a single Django Project. (For an explanation on the terminology *Django application* and *Django project*, see chapter 2.)

1.2.2 Use the latest Django and Python versions

Sigbank was written in the 1.6 version of Django. This version is several years old.

The second goal of this project, then, is to upgrade the Signbank codebase to version 1.10 of Django, the latest version; and to use the latest version of Python, version 3.5; and to try, where reasonable, to maintain compatibility with the older version of Django and Python.

1.2.3 Tests

The Auslan codebase lacks tests. The third goal, then, of this project is to write tests. Each Django application should have its own suite of tests. In addition, tests which span the entire Signbank codebase should be written.

A code coverage of at least 85 percent is desired.

1.2.4 Documentation

The Auslan codebase lacks documentation. The fourth goal, then, of this project is to write documentation. Each Django application should have its own documentation.

1.3 Project planning

Each of the following subsections deals with an aspect of project planning.

Task Name	Start Date	End Date	Duration
Gain initial, broad understanding of the project	25/07/2016	1/08/2016	7
Identify requirements	1/08/2016	5/08/2016	4
Learn Django	25/07/2016	1/11/2016	99
Write the progress report	10/08/2016	5/09/2016	26
Set up Travis CI (continuous integration)	5/08/2016	6/08/2016	1
Set up cookiecutter-django-package	5/08/2016	6/08/2016	1
Set up Codecov (test coverage)	5/08/2016	6/08/2016	1
Modularise the feedback app	5/08/2016	2/09/2016	28
Modularise the video app	2/09/2016	30/09/2016	28
Modularise the registration app	30/09/2016	3/10/2016	3
Modularise the dictionary app	3/10/2016	31/10/2016	28
Modularise the Pages app	31/10/2016	7/11/2016	7
Modularise the Attachments app	31/10/2016	7/11/2016	7
Intergrate all of the apps	5/08/2016	7/11/ 2016	94
Write the final report	15/10/2016	7/11/2016	23
Give presentation	14/11/2016	15/11/2016	1

Table 1.1: What must be done

1.3.1 Scope

The major goals of the project were given in section 1.2. Presented in the table above are the concrete tasks that, if completed successfully, will realise these goals. Each row of the table corresponds to a task. The tasks that have the word *modularise* in them require a few words of explanation.

By *modularise*, the following things are meant:

- Make each app independent from the other apps (to the extent that it it reasonable to do so).
- Write tests for each app.
- Write documentation for each app.
- Upgrade the cosebase so that it adheres to the best-practices for Django 1.10, and Python 3.5.

As can be gleaned from the table, there is a lot to do, perhaps too much. In the end, the Pages and Attachments apps were not successfully modularised. That said, these two apps are, according to Steve Cassidy, very broken and therefore not worth salvaging. Research into what could replace these apps was undertaken; *wagtail*, a Content Management System (CMS) for Django, looks promising. Future developers, picking up

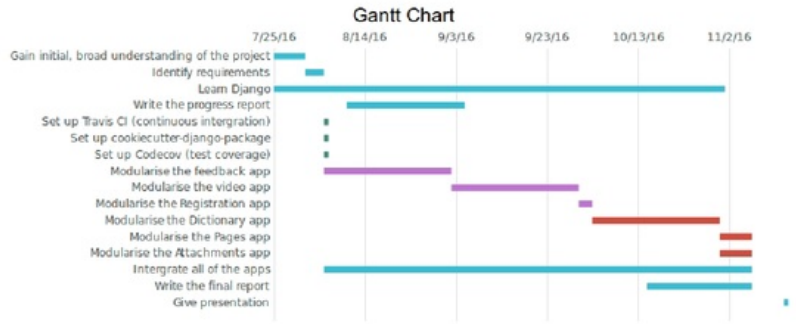


Figure 1.3: Gant chart

from where this project ended, should consider that app as a replacement for the Pages and Attachments apps.

1.3.2 Costs

No formal budget was allocated to this project.

The need to make a purchase never arose, perhaps because the project consists entirely of free software.

1.3.3 Time

The gantt chart, figure 1.3, gives a visual overview of the time allocated to each task. In the end, it took longer to modualrise the dictionary app, the feeback app, and the video app than was anticipated.

Chapter 2

Literature Review

The goals outlined in section 1.2 draw upon three fields of knowledge:

- Django
- Testing in Django
- Documentation

The literature on each of these different fields is surveyed in this chapter. Because this is a software project, some of the literature is written in the form of a blog post.

2.1 Django

Django is a web-framework written in python. It was created in 2003 by developers working for the newspaper *Lawrence Journal-World*. [2]

2.1.1 Web-Framework

A web-framework is a software framework for writing a web-application. It provides the generic functionality required by a web-application, leaving the user of the framework to provide the functionality specific to his web-application. [3] There are many benefits to using a web-framework. In all but a very few cases, the benefits outweigh the costs. Using a web-framework saves time; it lowers the barriers to entry for writing a web-application; it provides functionality that is well-tested for bugs, and that is efficient in terms of speed and memory usage.

Django is one of many web-frameworks. It happens to be the 5th most popular one as of the September, 2016. [4]. Due to its popularity, there are a plethora of resources on it. The following list contains each resource that is referenced in this literature review:

- The official Django documentation [5]

- Test Driven Development with Python [6]
- Test-Driven Development with Django [7]
- The Django Book [2]
- Two Scoops of Django [8]

2.1.2 Django and MVC

The three most important words in Django development are *models*, *views* and *urls*. These words appear everywhere, and for good reason: they promote a well-established software architectural pattern known as Model View Controller (MVC). [2]

The MVC pattern is a famous architectural pattern in software. It provides a template for solving a commonly occuring problem in software: the display and manipulation of data. An application that uses the MVC pattern is split into three independent components: [2]

- The *model* – this component of the software handles how the data are stored in the databases. [2]
- The *view* – this component of the software handles how the data are displayed (to the user). [2]
- The *controller* – this component of the software handles user input, mapping it to an appropriate action in the system, which may involve manipulating data and/or displaying data back to the user. [2]

Because each component is independent from the other components , a change can be made in one component without affecting the other components. For example, the database, defined in the *models* component, could be changed from Sqlite3 to PostgreSQL without the need to change code in the *views* and *controller* components of the application. As it turns out, this occurrs commonly in Django applications.

Django uses the MVC pattern, though with a slight variation: Django lacks a controller component. Due to this variation, the question of whether Django *really* uses MVC commonly comes up. The official django documentation has this to say:

"...In our interpretation of MVC, the view describes the data that gets presented to the user. Its not necessarily how the data looks, but which data is presented. The view describes which data you see, not how you see it. Its a subtle distinction. [...] Where does the controller fit in, then? In Djangos case, its probably the framework itself: the machinery that sends a request to the appropriate view, according to the Django URL configuration." [5]

So, Django itself acts as the controller, with the urls defined in the *urls* component of a Django application mapping user input to an appropriate action in the Django application. The quote also mentions that the *views* component of a Django application is

responsible for *what* data are displayed, not *how* the data look. This, too, is another way in which Django deviates from the MVC pattern. In Django, how the data are displayed is controlled by *templates*. *templates*, *views*, *models* and *urls* are explained in the next section.

2.1.3 The typical components of a Django application

```
mysite
├── manage.py
├── myapp
│   ├── admin.py
│   ├── tests.py
│   ├── models.py
│   ├── views.py
│   └── templates
│       └── myapp
│           └── index.html
└── mysite
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

The directory structure above is typical of a Django application. [8]. Before going any further, a few points on Django terminology are necessary. In Django terminology, *mysite* in the directory structure above is called a *project*, not an *app*. The word *app* is reserved for *myapp* in the directory above. An *app* in django is a "small library designed to represent a single aspect of a project." [8]. So, in the case of Signbank, Signbank is a project, and it contains various apps, each of which represents a single aspect of it. The various apps of which Signbank consists were listed earlier. Ideally, these apps should be independent from one another, a topic which will be addressed later in the literature review.

The following sub-subsections give a (very) brief overview of the key components of a django application

Models

In a Django app, the file *models.py* is the *mode*l component of the app. This file defines the data of the app. [5]. A typical *models.py* file looks like this

```
1 from django.db import models
2
3
4 class Person(models.Model):
5     first_name = models.CharField(max_length=30)
6     last_name = models.CharField(max_length=30)
```

[5]

The class `Person` defines a database called *Person*; the class variables are the fields of the database. It is very straighforward, and one of the strengths of Django. The data stored in the database can be queried (retrieved) though Django's *Object Relational Mapping* (ORM).

Urls

In a Django app, the file *urls.py* is part of the *controller* component of the app. This file defines each valid URL in the app, and it defines which view , defined in *views.py* and explained in the next sub-subsection, should be called for a given URL. [5]

```
1  from django.conf.urls import url
2
3  from feedback import views
4
5
6  app_name = "feedback"
7  urlpatterns = [
8      # ex: /
9      url(r"^$", views.index, name="index"),
10     # ex: generalfeedback/
11     url(r"^generalfeedback/$", views.GeneralFeedbackCreate.as_view(),
12         name="generalfeedback"),
13     # ex: show/
14     url(r'^show', views.showfeedback, name = 'showfeedback'),
15     # ex: general/delete/1/
16     url(r'^(?P<kind>general|sign|missingsign)/delete/(?P<id>\d+)/$',
17     views.delete, name = 'delete'),
18     # ex: missingsign/
19     url(r'^missingsign/$', views.missingsign,
20         name='missingsign'),
21     # ex: abscond-1/
22     url(r'^sign/(?P<keyword>.+)-(?P<n>\d+)/$', views.signfeedback,
23         name = 'signfeedback'),
24 ]
```

The code snippet above is from the feedback app of Signbank. It defines, using regexes, each valid url in the app. If a URL entered by a user matches one of the definitions in the code snippet above, then the function in *views.py* corresponding to the URL definiton is called.

Views

In a Django app, the file *views.py* is the *view* component of the app. This file defines what action should be taken for a given URL. [5].

```
1 from django.shortcuts import render, get_object_or_404, redirect
2 from django.conf import settings
3
4
5 def index(request):
6     return render(request, "feedback/index.html",
7         {"language": settings.LANGUAGE_NAME,
8          "country": settings.COUNTRY_NAME,})
```

The above code snippet comes from the feedback app of Signbank. The index() function is called when a user enters the URL that corresponds to it. The function calls the render() function. This function generates HTML per a template, explained next, and then that HTML is returned by the *index()* function, ultimatley making its way back to the user who requested it.

Templates

In a Django app, templates serve the purpose of a *view* in the MVC pattern. Templates control how the data look to the user. This is done through Django's templating language, though other language can be subsituted if desired. *jinja* is a popular substitute. [2]

2.1.4 Independent apps

To conclude the discussion on Django, the concept of an independent app is given a brief overview. This concept is very important for the project.

As has been mentioned previously, a Django project consists of apps, each of which , ideally, represents one aspect of the project. The authors of Two Scoops of Django offer this advice:
"In essence, each app should be tightly focused on its task. If an app cant be explained in a single sentence of moderate length, or you need to say and more than once, it probably means the app is too big and should be broken up."

In the extreme case, the app is so decoupled from the project that it can be used in an arbitrary Django project. That is the ideal that this project strives for. Both the official Django documentation and Two Scoops of Django offer advice on how to achieve this, and on how to package the app. The authors of Two Scoops of Django provide a convenient program called ccookiecutte-django-package, which is capable of generating the template for an independent Django app. [8] [9]

2.2 Testing in Django

"Testing is a pillar of professional software development" [7]; "Automated testing is an extremely useful bug-killing tool for the modern Web developer. You can use a collection of tests a test suite to solve, or avoid, a number of problems..." [5] All of the literature reviewed had only good things to say about testing. The official Django documentation gives two benefits to testing: [5]

- "When youre writing new code, you can use tests to validate your code works as expected."
- " When youre refactoring or modifying old code, you can use tests to ensure your changes havent affected your applications behavior unexpectedly."

2.2.1 Two kinds of testing

There are myriad categories of tests. [7] In this literatue review, two kinds will be considered:

- Functional tests
- Unit tests

Unit tests test the correctness of source code functions, while functional tests test the correctness of a feature (or features) of the app. A feature could span multiple source code functions. [7] . Unit tests should be short and self-contained. Functional tests, on the other hand, can be large and can span multiple components of an app. [7] Both kinds of tests should be written [8] as each offers unique benefits.

Django makes it very easy to write unit tests and functional tests. [5]

Unit tests

Django adds features ontop of python's unittest module to make writing unit tests for django apps very easy. To ensure that unit tests are self-contained, Django creates a clean database at the beginning of each test and tears it down once the test has finished. [5]

Django provides two different classes for writing unit tests:

- RequestFactory
- Client

The literature is mixed on which to use. *Test Driven Development with Python* [6] does not touch upon RequestFactory, but *Test-Driven Development with Django* [7] does. For writing unit tests, Kevin Harvey, the author of *Test-Driven Development with Django*, makes a convincing case for why RequestFactory is superior to Client. RequestFactory

is closer to the ideal unit test because it is more self-contained than the Client. To give a concrete example, in order to test a view function with the Client, the url must be correct. This adds a dependency which has nothing to do with the view function under test. RequestFactory does not suffer from this problem because it makes no assumptions about the url.

RequestFactory's benefits come at a cost, however. It is harder to write tests using the RequestFactory class, largely due to lacking documentation. [8]

Functional tests

Django does not provide a native functional test library(though some argue that the Client class is *really* a functional test library [10]). This literature review considers two third-party libraries

- Selenium [11]
- Web-Test [12]

These two libraries can test a django app from the perspective of the user, right down to simulating a button press. Selenium differs from web-test in that selenium runs in the browser. [13] Web-test, on the other hand, does not. It uses the WSGI protocol, a protocol for python applications to communicate with web-servers, to test the app. [**?**] For this reason, it only works for python apps. For the purposes of this project, that web-test only works for python apps is not a problem.

For the purposes of this project, web-test is superior. The only reason one would use selenium over web-test is to test java script running in the browser. [13] Signbank does not appear to have much complicated java script, so selenium is not worth the cost of being slow compared to web-test. [13].

2.2.2 Continuous Integration

Two scoops of django recommends Jenkins, but Jenkins is outdated. [8] Steve Cassidy has recommended Travis Cl.

Travis CI is a cloud-based continuous integration system. It will build your app , run the app's tests, and then report whether the build was successful. It integrates seamlessly with GitHub; it detects when a new commit is pushed to an app's GitHub page, and then it runs the buld process described previously.

Travis CI has extensive documentation that will be useful for constructing the `.travis.yml` files, the file that configures the build process.

2.2.3 Test coverage

Both the official Django documentation and Two Scoops recommend [8] [5] *coverage.py* as a way to measure how much of the app is covered by the tests.

To supplement Coverage, a cloud-based code coverage tool called Codecov will be used. This tool enables one to see the lines of an app's code that are not being covered by the app's tests.

2.3 Documentation

Kevin Harvey identified documentation as a pillar of great software development. [7]. In the field of documentation writing, Sphinx is the undisputed champion. [8]

Chapter 3

Approach

This chapter describes how the tasks identified in section 1.3.1 were tackled. Note that not all of the tasks identified in 1.3.1 are described here; the tasks not described here are too trivial to be worthy of an explanation.

It's also worth mentioning the system on which the app was developed; this is descibed in the next section.

3.1 Overview of the development environment

The Signbank codebase is primarily written in Django; this means that it should be able to run on Windows, Ubuntu and other Linux distributions, and Mac. Because Ubuntu is easiest to work on, it was chosen as the development environment for this project; specifically, version 16.04 of Ubuntu was used. Django's own web-server was used as the HTTP server for development. So that was the development environment, but there was also the production envionment.

In general the production environment isn't necessarily the same as the development environmen for an app. For development, one wouldn't use Django's own web-server, for example. Instead, one would use *nginx* or *apache*. Worrying about *production* issues was beyond the scope of this project, however. Still, it's worth mentioning because the difference between production and development can be confusing.

3.2 Overview on the structure of the project

Signbank is, in the Django vernacualr, a *Django project* made up of *Djano apps*.

The video app, feedback app, and dictionary app have their own GitHub repositories that were created at various stages thougout this project. The registration app also has a GitHub repository, though one created and owned by a third party (all of this will be explained later when the registration app is explained.)

The purpose of mentioning that was to clarify, for a third time, the difference between a *Django app* and a *Django package*. This distinction will be important for understanding

the next three sections, which describe how cookiecutter-django-package, Travis CI, and Codecov were set up. Note that each of these three tasks, setting up cookicutter-django-package, setting up Travis CI, and setting up Codecov, had to be done for each app (except for the registration app).

3.3 Set up cookiecutter-django-package

Cookiecutter-django-package is a template for reusable Django applications. (For a background on reusable Django applications, see chapter 2.) The video app, the feedback app, and the dictionary app are based on the template provided by cookiecutter-django-package.

What follows are the steps that were followed to set up cookiecutter-django-package.

1. Install a python program called `cookiecutter`.
2. Run the program. The progarm will ask you questions, such as, 'What is your name?', 'What software license do you want to use?', and, 'What is the name of your app?' It uses your answers to these questions to configure the template for your particular circumstances.
3. When the program finishes quizzing you, it will create the directory structure of your app.

The directory structure is important. It contains all of the files that constitute your app; cookiecutter-django-package organises them cleanly for you, and fills them with initial data to get you started.

```
signbank-video
├── .github
├── docs
├── tests ..................................... This folder contains the tests.
├── video. .................... This folder contains the views, models, etc of the app.
├── editorconfig
├── .gitignore
├── .travis.yml
├── AUTHORS.rst
├── CONTRIBUTORS.RST
├── HISTORY.rst
├── LICENSE
├── MANIFEST.in
├── Makefile
├── README.rst
├── requirements.txt
└── requirements_dev.txt
```

```
├── requirements_test.txt
├── runtests.py. ..... This file is the settings.py file, and test runner, combined.
├── setup.cfg
├── setup.py
└── tox.ini
```

The directory structure above is the directory structure of the video app. This directory structure was created by running `cookiecutter`. Some of the files created by `cookiecutter` were not used; the most important files and directories are listed below:

- tests
- app name (in the directory structure above, this is called `video`.
- .travis.yml
- runtests.py
- setup.py
- the requirements files (`requirements.txt`, et etc)

3.3.1 tests

The tests directory contains all of the tests for an app, as well as all of the data that the tests require to run.

Due to a name-spacing issue, it was necessary to create a `urls.py` file inside the tests directory, and to make this `urls.py` the `ROOT_URLCONF`, defined in `runtests.py`.

3.3.2 app name

This directory contains the logic, data definitions, and static assets of the app. In Django vernacular, this directory contains the models, views, urls, and templates, and static images, videos, JS code, and CSS code, of the app.

3.3.3 .travis.yml

More will be said about this file in section 3.5.

3.3.4 runtests.py

This file runs the tests defined in the tests directory. It also acts acts as the `settings.py` file for the app when the tests are being run.

The tests can be run with the command `python3 runtests.py`. That command will run all of the tests inside the tests directory, and the output of the tests will be sent to the terminal.

3.3.5 setup.py

This file installs the app on the computer. The command `python3 setup.py install` will install the app on the computer, making it available to a Django project, or to another Django app.

3.3.6 Requirements files

The three files `requirements.txt`, `requirements_dev.txt`, and `requirements_test.txt` define the python apps (not necessarily Django apps) that an app depends on. This project only eneded up using `requirements.txt`, and `requirements_text.txt` The difference between these two files is that the the latter contains only the apps required for the tests to run. More will be said on these two files in each app's section of this document.

3.4 Set up Travis CI

Travis Ci (Travis) is a cloud-based continuous integration sytem. (For more of a beackground on it , see chapter 2.)

Each app's GitHub repository was connected to Travis. This was done through a nify interface on Travis' website. Travis was then set up to run an app's test suite whenever a new commit was pushed to the app's GitHub repository. Pass or failure of the test suite was conveniently indicated on the app's GItHub page by a *badge*, as in figure 3.2. When the tests fail, that badge turns red, and the text on it reads, 'Failing.' Travis also keeps a history of test results on its website.

So that was a high-level overview of how Travis was used in this project. What follows is a more detailed description.

When Travis detects a new new commit on an app's GitHub repository, it creates a virtual Ubuntu 12.04 environment and builds the app in it. Travis' build instructions come from a file called `.travis.yml`, located in the root directory of the app. The cookiecutter-django package conveniently creates this file for you.

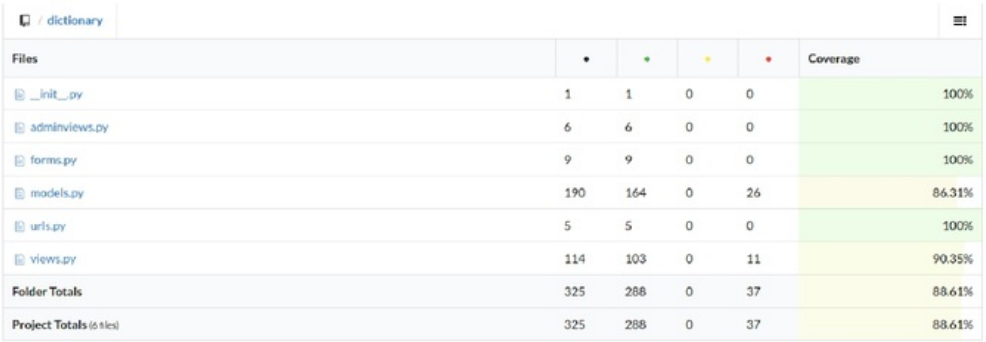
```
1  # Config file for automatic testing at travis-ci.org
2
3  language: python
4
5  python:
6    - "3.5"
7    - "3.4"
8    - "2.7"
9
10 before_install:
11   - pip install codecov
12   - sudo apt-get -qq update
13   - sudo apt-get install -y ffmpeg
```

Figure 3.1: Travis badge

```
14
15 install: pip install -r requirements_test.txt
16
17 script: coverage run --source video --omit=*/fields.py
18
19 after_success:
20   - codecov
```

The listing above is the `.travis.yml` file of the video app. It will serve as an example for explaining how the file works in general.

- The third line of the file tells Travis that the app is written in Python.
- The 5th to 8th lines tell travis that it should use versions 3.5, 3.4, and 2.7 of Python. Travis repeats the build process for each of these versions of Python; this ensures that your app is compatiable across a range of Python versions. Interestingly, it was found that all of the apps failed to pass their test suites for version 3.3 of Python. It was decided that it wouldn't be worth the effort of making the apps compatible with Python 3.3. That's why Python 3.3 is not in the the `.travis.yml` file.
- The 10th to 15th lines install all programs that are external to the app.
- Line 17 runs the tests, defined in the app's tests directory. It also measures the code coverage achieved by the tests; that's what the `coverage` command is for.



/ dictionary

Files	•	•	•	•	Coverage
__init__.py	1	1	0	0	100%
adminviews.py	6	6	0	0	100%
forms.py	9	9	0	0	100%
models.py	190	164	0	26	86.31%
urls.py	5	5	0	0	100%
views.py	114	103	0	11	90.35%
Folder Totals	325	288	0	37	88.61%
Project Totals (6 files)	325	288	0	37	88.61%

Figure 3.2: Code coverage for the dictionary app

- Line 19 sends the coverage data off to Codecov, a cloud-based code coverage service that will be described in a later section.

3.5 Set up Codecov

Codecov is a cloud-based code coverage visualisation tool. (For more of a background on it, see chapter 2.)

cookiecutter-django-package sets up the `.travis.yml` file to integrate with Codecov. That happens on line 19 of the `.travis.yml` file above. One manual setp was required, however, for the integration to work: Codecov has to be connected with the app being tested. This was done through Codecov's website.

Figure 3.2 shows one of the visualisations that Codecov can produce. That picture is showing the percentage of lines in each of the dictionary app's source code files that are covered by the tests.

More on this will be said in the sections that describe the approach taken for modularising each app.

3.6 Modularise the feedback app

The feedback app handles the user-submitted feedback. Here is the GItHub page of the feedback app created in this project: https://github.com/hujosh/signbank-feedback

3.6.1 Overview of the feedback app

The feedback app allows users to submit five kinds of feedback:

- Feedback on a sign – This is feedback on a word in the dictionary.

- Feedback on a missing sign – If a user feels like a sign is missing from the dictionary, then he can make that known via this kind of feedback.
- Feedback on a gloss – This is for feedback on the sign itself.
- Interpreter feedback – this is for feedback on a sign that is submitted by an interpreter.
- General feedback – This is for feedback that doesn't fit into any of the above categories.

The feedback app also handles showing the feedback to the administrator of the website, and allowing the administrator to delete feedback.

3.6.2 Making the app independent

The `models.py` component of the feedback app had some unnecessary dependencies with the dictionary app.

```
1 class SignFeedback(models.Model):
2     """Store feedback on a particular sign"""
3
4     user = models.ForeignKey(authmodels.User, editable=False)
5     date = models.DateTimeField(auto_now_add=True)
6
7     translation = models.ForeignKey(Translation, editable=False)
```

The code above comes from the `models.py` file of the feedback app. Line seven contains a foreign key to a row in the translation databas, which is defined in the dictionary app. Now, what exactly a `translation` is will be explained later, in the dictionary app section; the point to be made here is that, from the perspective of the feedback app, it doesn't matter what a translation is.

Before removing the dependency, it was necessary to understand why it was there in the first place. The reason was traced to the part of the feedback app that was responsible for showing feedback to the administrator. The administrator needed a way to view, in the dictionary, the sign (gloss), or word that the feedback was about. The translation foreign key was used to get the URL, and thus the location in the dictionary, of the sign (gloss), or word.

Knowing why the dependency was there enabled it to removed without breaking the aforementioned functionality required by the administrator. Here's how that was achieved.

All words in the dictionary had the following URL form `/dictionary/words/keyword-n`, where *keyword* is the word itself, and n is a number used to differentiate between words that have the same word (this can occur, for example, when two words are homonyms). So, if all that was required to locate a word in the dictionary was its URL, why not just store the URL in the feedback app rather than the translation foreign key? So that was how the problem was solved. The same solution was applied to the gloss feedback.

Removing that dependency brought the feedback closer to the ideal of being independent. Unfortunately, it was not possible to remove all dependencies. The missing sign feedback allowed users to upload a video, and the uploaded video depended on functions defined in the video app that could not be removed.

3.6.3 Upgrade the cosebase

Upgrading the codebase was quite a lot of work, but in the end it paid off. The *views.py* file, where the most extensive code upgrading took place, went from 300 lines of code to 140 lines of code. This large reduction in lines of code was achieved by converting the forms of the feedback app to *ModelForms*.

Compare the same function that in the first code listing using a *ModelForm*, and in the second code listing does not use a *ModelForm*.

```
1  @login_required
2  def missingsign(request):
3      if request.method == "POST":
4          form = MissingSignFeedbackForm(request.POST, request.FILES)
5          if form.is_valid():
6              form_to_save = form.save(commit=False)
7              form_to_save.user = request.user
8              form_to_save.save()
9              success_message = 'Thank you for your feedback...'
10             messages.success(request, success_message)
11             return HttpResponseRedirect(reverse('feedback:missingsign'))
12     # Any other kind of request goes here
13     else:
14         form = MissingSignFeedbackForm()
15     return render(request, 'feedback/missingsign_form.html',
16                                 {
17                                  'language': settings.LANGUAGE_NAME,
18                                  'country': settings.COUNTRY_NAME,
19                                  'title':"Report a Missing Sign",
20                                  'form': form
21                                  })
```

```
1  @login_required
2  def missingsign(request):
3      posted = False # was the feedback posted?
4      if request.method == "POST":
5          fb = MissingSignFeedback()
6          fb.user = request.user
7          form = MissingSignFeedbackForm(request.POST, request.FILES)
8          if form.is_valid():
```

```
 9             # either we get video of the new sign or we get the
10             # description via the form
11              if form.cleaned_data.has_key('video') and
12               form.cleaned_data['video'] != None:
13               fb.video = form.cleaned_data['video']
14              else:
15               # get sign details from the form
16                fb.handform = form.cleaned_data['handform']
17                fb.handshape = form.cleaned_data['handshape']
18                fb.althandshape = form.cleaned_data['althandshape']
19                fb.location = form.cleaned_data['location']
20                fb.relativelocation = form.cleaned_data['relativelocation
21                fb.handbodycontact = form.cleaned_data['handbodycontact']
22                fb.handinteraction = form.cleaned_data['handinteraction']
23                fb.direction = form.cleaned_data['direction']
24                fb.movementtype = form.cleaned_data['movementtype']
25                fb.smallmovement = form.cleaned_data['smallmovement']
26                fb.repetition = form.cleaned_data['repetition']
27              # these last two are required either way (video or not)
28              fb.meaning = form.cleaned_data['meaning']
29              fb.comments = form.cleaned_data['comments']
30              fb.save()
31              posted = True
32     else:
33         form = MissingSignFeedbackForm()
34     return render_to_response('feedback/missingsign.html',
35                               {
36                               'language': settings.LANGUAGE_NAME,
37                               'country': settings.COUNTRY_NAME,
38                                'title':"Report a Missing Sign",
39                                'posted': posted,
40                                'form': form
41                                },
42                              context_instance=RequestContext(request))
```

The first code listing is much shorter, and much more readable, and subsequently much more maintainable.

Another improvement made to the codebase was the way in which success messages were handled. In the code listing directly above, on line 31 a variable, `posted`, is being set to true if the feedback is successfully posted. This variable is then sent to the template `feedback/missingsign.html`, where it is used thus:

```
1 {% if posted %}
2 <div id="feedbackmessage">
```

```
3     <p class='alert alert-info'>
4         Thank you for your feedback.
5         Note that addressing your feedback may take some time
6         depending on the level of requests.
7     </p>
8 </div>
```

In the improved code listing, lines 9 and 10 replace the `posted` variable with Django's messages framework [14]. Line 10 puts the success message, defined on line 9, into the Response object, and thus it can be accessed in the template without having to pass in a variable. This improvement makes the code easer to read, and also easier to maintain, because adding a new success message , or error message, is as simple as

```
1  messages.success(request, success_message)
```

or

```
1 messages.error(request, error_message)
```

Upgrading the codebase from Django 1.6 to 1.10 was not a very big job. The only incompatability was the code on line 41 of the above code listing. It was removed , and nothing was required to replace it.

Upgrading the codebase from Python 2.7 to Python 3.5 was also not a big job. The `has_key` function of the dictionary (Python's dictionary class) class does not exist in Python 3.5, and so the scattered instances of it had to be replaced with the more *pythonic*

```
1 if 'key' in dic
```

syntax

3.6.4 Writing documentation

Sphinx was used to write the documentation. Sphinx is so established as a documentation writing tool that the cookiecutter-django-package started the app off with a README file written in the Sphinx (RestructedText) format.

The feedback app wasn't very complicated, so extensive documentation wasn't required; a few words on how to run the tests, how to intall the feedback app, and what variables in `settings.py` the feedback app requires, were documented. The README is displayed prominently on the feedback app's GItHub page, for all to see.

3.6.5 Writing tests

The test writing was a success. A code coverage of 86 percent (which exceeds the 85 percent goal) was achieved. That said, code coverage is only a start; testing the same lines of code under different data – especially data relating to edge cases – matters too, something that code coverage cannot detect.

In total 44 test cases were created, spanning unit tests and functional tests. The tests were split across four different files:

- `test_views.py` – For testing the functions in `views.py`
- `test_urls.py` – For testing the URL regexes in `urls.py`
- `test_functional.py` – For the functional tests
- `test_models.py` – For testing functions defined in `models.py`

First, a discussion on `test_views.py`

The decision to call the file `test_views.py` wasn't arbitrary. The test runner, invoked by the command `python3 runtests.py`, looks for files inside the tests directory with the word `test` prepended to them. Developers who take up where this project left off should keep this in mind when creating new test files.

Now, a look at the internals of `test_views.py`. Each function in `views.py` corresponds to a class in test_views.py. For example:

```
1  class IndexView(TestCase):
2      def setUp(self):
3          self.factory = RequestFactory()
4          # the url is irrelevant when RequestFactory is used...
5          self.url = '/'
6
7      def test_index_view_renders_right_template(self):
8          '''
9          'The index' view should render the 'feedback/index.html'
10         template
11         '''
12         request = self.factory.get(self.url)
13         with self.assertTemplateUsed('feedback/index.html'):
14             response = index(request)
15
16     def test_index_view_returns_200_response_code(self):
17         '''
18         The response code reutrned by the index view
19         should be 200.
20         '''
21         request = self.factory.get(self.url)
22         response = index(request)
23         self.assertEqual(response.status_code, 200)
24
25     def test_right_sign_language_is_rendered(self):
26         '''
27         The sign language rendered in the template
28         'feedback/index.html' should equal the
```

```
29          'LANGUAGE_NAME' variable in 'settings.py'.
30          '''
31          LANGUAGE_NAME = settings.LANGUAGE_NAME
32          request = self.factory.get(self.url)
33          response = index(request)
34          self.assertContains(response, LANGUAGE_NAME)
```

The code listsing above defines all of the unit tests that test the `index` function, which is defined in the app's `views.py` file. The `index` function happenes to be quite a simple function – it merely displays a static page – so the tests for it are not very complicated. That said, all of the other tests defined in test_views,py are structured similarily to the tests in the above code listing.

Each test class has a *SetUp* method. This method defines the data that all of the test cases in the test class have in common. This turned out to be useful because there were many occassions where it was required that a variable referenced in each test case be changed. With the variable defined in just one place, however, it only needs to be changed in one place.

The name of a test case describes what the purpose of the test is. For example, the test `test_index_view_returns_200_response_code` leaves nothing to the imagination. And to make the purpose of the test even clearer, it has a docstring defining, in prose, what the purpose of the test is. All of the tests defined in `test_view.py` have this self-explanatory nature. This comes in very handy when the tests are run and fail, because one can see and understand immediately what went wrong without having to search through the test code.

Most of the test classes test, at a minimum, that the right response code is returned, and that the right template is rendered. Tests for these two conditions can be seen in the code listing directly above. The reason for why these two cases were split across two tests, instead of testing them both in one test, is that one can be true while the other false. So the failure of the `test_index_view_returns_right_template` method, for example, contains no ambiguity as to what failed.

Due to the decision to use the `RequestFactory` class instead of the `Client` class (see chapter 2 for background on this), it was necessary to create two auxillary functions for running the tests in `test_views.py`.

```
1  def create_request(url = ''', method, data=None, permission=None):
2      '''
3      This function creates one of various requests. The type
4      of request that this function creates depends on the parametres
5      of the function.
6
7      Call this function in a test case, and use the returned
8      request object as an argument to a view.
9      '''
10     factory = RequestFactory()
```

```
11      # Set up the user...
12      user = create_user(permission)
13      if 'GET' in method.upper():
14          request = factory.get(url)
15      elif 'POST' in method.upper():
16          request = factory.post(url, data)
17      else:
18          raise ValueError("%s is an unrecognised method.'"%(method))
19      setattr(request, 'session', 'session')
20      messages = FallbackStorage(request)
21      setattr(request, '_messages', messages)
22      request.user = user
23      return request
```

```
1  def create_user(permission=None):
2      users = User.objects.all()
3      nusers = len(users)
4      # If a user doesn't exist already...
5      if nusers != 1:
6          user = User.objects.create_user(
7              username='Jacob', email='jacob@  ',
8              password='top_secret', first_name = "Jacob",
9              last_name = "smith")
10     else:
11         # If the user has already been created, use it
12         user = users[0]
13     if permission is not None:
14         permission = Permission.objects.get(name=permission)
15         user.user_permissions.add(permission)
16     return user
```

The reason why the `create_request` function was necessary was that the Request object returned by the `RequestFactory` constructor is very bare – something that makes it good for unit testing, though at the cost of extra work by the programmer writing the tests. The `create_function`, and its accompanying `create_user` function, were the extra work required. That said, it paid off because the tests were closer than they otherwise would have been to the idea unit test.

The bareness of the Request object returned by RequestFactory is that it lacks the middleware for handling sessions and the message framework. These two things are manually added into the Request object on lines 19 to 23 of the `create_request` function.

For the benefit of future developers who pick up where this project left off, a brief explanation on how to use these two functions is given.

The function `create-request` should be called in a test case, and the returned Request object should be passed to the view being tested, like this:

```
def test_index_view_returns_200_response_code(self):
        '''
        The response code reutrned by the index view
        should be 200.
        '''
        request = self.factory.get()
        response = index(request)
        self.assertEqual(response.status_code, 200)
```

In the code listing above, the `create_request` function is called with no arguments. This means that the default Request object is returned. The default Request object is a get request, with a user who has no peremissions logged in, and with no data in the query string.

The create_user function doesn't need to be called directly by test wrter. Since submitting the feedback required one to be logged in, it was necessary to create a logged in user. That is the purpose of the function.

So it's relatively simple to use, and brings the unit tests closer to the ideal unit test. That said, there were some difficulties in using it instead of the `Client` object. The `assertTemplateUsed(response, template_name)` method doesn't work for Requets created by the RequestFactory. Instead, one must do this:

```
def test_index_view_renders_right_template(self):
        '''
        'The index' view should render the 'feedback/index.html'
        template
        '''
        request = self.factory.get(self.url)
        with self.assertTemplateUsed('feedback/index.html'):
            response = index(request)
```

This solution took a long time to discover.

The last thing that will be said about the tests in `test_views.py` is how they helped to uncover an insidious bug that existed in the original `views.py` code.

```
 else:
       # get sign details from the form
       fb.handform = form.cleaned_data['handform']
       fb.handshape = form.cleaned_data['handshape']
       fb.althandshape = form.cleaned_data['althandshape']
       fb.location = form.cleaned_data['location']
       fb.relativelocation = form.cleaned_data['relativelocation']
       fb.handbodycontact = form.cleaned_data['handbodycontact']
       fb.handinteraction = form.cleaned_data['handinteraction']
       fb.direction = form.cleaned_data['direction']
       fb.movementtype = form.cleaned_data['movementtype']
       fb.smallmovement = form.cleaned_data['smallmovement']
```

```
13          fb.repetition = form.cleaned_data['repetition']
```

The code listing above is part of the `missingsign` function, defined in views.py. The code is setting the fields of an instance of the `SignFeedback` model with values obtained from a submitted form; this is all in preparation for saving the tSignFeedback model instance to the database. Each of the fields in the code listing above is optional; when no data for a field are submitted by a user, the field is supposed to default to the number (integer) 0.

When the tests were ran on this bit of code, however, a database error kept occuring, saying that `''` was not a valid integer. Somehow, the default value of 0 was not being set. In the end, it was worked out that in the code listing above, each of the fields that did not have values submitted for them were being set to `''''`. Since Django does not consider `'''` to be blank, or empty, the default value was not set, hence `''` was making its way into the database for fields that had be be integers.

When the `ModelForm` upgrade, described in subsection 3.6.3 took place, this error was resolved. Indeed, one of the reasons for upgrading the code to use `ModelForm` was to resolve this issue.

The second test file, `test_urls.py`, will now be discussed.

This file is responsible for testing that the URL regexes are correct, and that an entry in `urls.py` points to the right function in `views.py`. (For more backround on `urls.py`, see chapter 2.

Here is a test defined in test_urls.py

```
1 def test_root_url_resolves_to_index_view(self):
2         '''
3         '/' should be routed to the index function
4         in 'views.py'
5         '''
6         found = resolve('/')
7         self.assertEqual(found.func, index)
```

Like the test cases defined in `test_views.py`, this test case is self-explanatory. It's testing that the index url – \ – points, or routes, to the `index` function in `views.py`. The `resolve` function, on line 6 of the above code listing, searches through the urls defined in `urls.py` for the url that matches the string passed in as the first argument to `resolve`. If no urls match, an exception is raised; if a url matches, then the next line of code checks that the matched url points to the correct function in `views.py`.

The test cases themselves are relatively simple, but there was one complication: what if the url being tested points to not a function but a class based view? [15]. This occurred for the general feedback view, which was implemented as a class based view. Here's how that was dealt with in `test_urls.py`

```
1 def test_general_feedback_url_resolves_to_general_feedback_view(self):
2     '''
3     '/generalfeedback/' should be routed to the GeneralFeedbackCreate
4   class in 'views.py'
```

```
5       '''
6       found = resolve('/generalfeedback/')
7       self.assertEqual(found.func.__name__,
8         GeneralFeedbackCreate.__name__)
```

Instead of passing a function object to `assertEqual`, the name of the class based view is passed, and compared to the name of the class based view that the url points to.

The test in `test_models.py` will now be discussed.

This file is a small one, containing only one test. The purpose of this file is to test, directly, the functions defined in `models.py`. The `models.py` of the feedback app wasn't very complicated, so it didn't require extensive unit testing.

The tests in `test_functional.py` will now be discussed.

This file contains functional tests; a test defined in here spans multiple function across multiple files of the feedback app. (For a bacground on functional testing and unit testing, see chapter 2.)

These tests, apart from spanning multiple functions across multiple files, enable html elements, defined in the templates, to be tested. For example, does the right outcome occur upon clicking a button?

Clearly the *RequestFactory* stlye tests, used for the aforementioned unit tests, are not a good fit for functional testing. Instead, a testing framework called Django-webtest was used. (For a background on this, see chapter 2).

Basically, these tests mirrow how a user would use the feedback app. Here is an example test case to illustrate how this works:

```
1  class MissingSignFeedbackPage(WebTest):
2
3   def test_submit_missing_sign_feedback_without_required_field(self):
4           response = self.app.get(self.url, user=self.user)
5           # The user submits the form without entering the required field
6           response = response.form.submit()
7           # He should not have been redirected
8           self.assertIn('200', response.status)
9           # There should be no success message
10          self.assertNotIn(self.success_message, response)
11          # He should be back on the general feedback form
12          self.assertTemplateUsed(response, 'feedback/missingsign_form.html')
13          # He should see the error message on the form
14          required_field = 'meaning'
15          error_string = "This field is required."
16          self.assertFormError(response, 'form', required_field, error_string)
17          # Make sure that no other fields are displaying errors.
18          # Each field displays two error messages, actually. That's
```

```
19          # why I passed 2 as a parameter...
20          # The reason for this is that both bootstrap3
21          # and django's own form code display an error message.
22          self.assertContains(response, error_string, 2)
23
24          # Make sure that no feedback was saved to the database
25          feedback = MissingSignFeedback.objects.all()
26          self.assertEqual(len(feedback), 0)
```

The first thing to notice about the code listing above is that the class, *MissingSignFeedbackPage*, does not inherit from django.test.TestCase. That is due to the fact that Django-webtest was used, a third party app.

The test, in the above code listing, tests that the right outcomes occur upon submitting missing sign feedback but without the required fields of the missing sign feedback form being submitted. Once again, the name of the test is self-explanatory. It was opted not to use a docstring for the functional tests, because as can be seen, inline – or *hashtag* – comments explain what is happening for every line requiring an explanation. This style of wrtiing comments in functional tests was picked up from Harry J Percival's *TestDriven Development with Pyton* [6].

Each line of the test tells a story of the user submitting missing sign feedback without a required field.

Interestingly, this test found a bug that, due to a time constraint, could not be fixed; future developers, beginning from where this project left off, should fix this. The bug is simple: error messages for missing, required data are being displayed twice. This is due to the fact that both *bootstrap3*, a django app for integrating the bootstrap framework with a Django app, and Django, render an error message, and thus two error message are rendered.

3.6.6 Conclusion

In conclusion, the goals that were set for the feedback app – modularising it, writing tess for it, writing documentation for it, and updating its codebase to the latest versions of Django and Python, were achieved. On the modularisation front, a foreign key tying the feedback app to the dictionary app was removed; on the testing front code coverage of 86 percent was achieved, exceeding the requirement of 85 percent code coverage; on the documentation front, a simple but informative README, produced with Sphinx, was created; and on the upgrade front, the code was upgraded to the latest versions of Django and Python, resulting in fewer lines of code that will be easier to maintain and debug.

There are, however, still a few issues left unresolved. These will be picked up in a later section on unresolved issues. That said, the feedback app, as it is now, can be integrated with the other apps successfully.

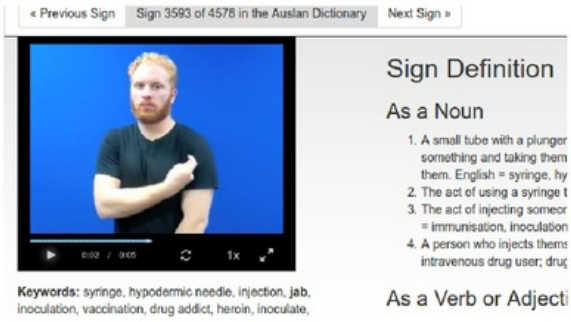


Figure 3.3: The word jab, as it appears in the dictionary. Its accompanying video is on the left

3.7 Modularise the video app

The video app handles the uploading, storage, retrival, and deletion of videos; both videos submitted by users as feedback, and the videos of the signs that are displayed in the dictionary, are handled by the video app.

Here is the GItHub page of the video app created in this project: https://github.com/hujosh/signbank-video

3.7.1 Overview of the video app

Each word in the dictionary has an accompanying video of a person *signing* the sign of the word. This can be seen in figure 3.3. What can't be seen in figure 3.3, however, are the options that an administrator has for deleting the video, and uploading a new video; the video app is responsible for all of these things.

The video app is also responsible for handling videos submitted by users as feedback. Unfortunately, due to time constraints, this aspect of the video app was not dealt with. Developers, picking up from where this project left off, should look into this. More will be said on this in a later section on unresolved issues.

3.7.2 Making the app independent

There were unnecessary dependencies scattered all across `views.py`. These all had to do with getting the *gloss* (this will be explained in the section on the dictionary app) associated with a video.

Here is an example, from the function `addvideo`:

```
1 def addvideo(request):
2     """View to present a video upload form and process
3     the upload"""
4
```

```
5       if request.method == 'POST':
6
7           form = VideoUploadForGlossForm(request.POST, request.FILES)
8           if form.is_valid():
9
10              gloss_id = form.cleaned_data['gloss_id']
11              gloss = get_object_or_404(Gloss, pk=gloss_id)
12
13              vfile = form.cleaned_data['videofile']
14
15              # construct a filename for the video, use sn
16              # if present, otherwise use idgloss+gloss id
17              if gloss.sn != None:
18                vfile.name = str(gloss.sn) + ".mp4"
19              else:
20                vfile.name = gloss.idgloss + "-" + str(gloss.pk) + ".mp4"
21
22              redirect_url = form.cleaned_data['redirect']
23
24              # deal with any existing video for this sign
25              oldvids = GlossVideo.objects.filter(gloss=gloss)
26              for v in oldvids:
27                  v.reversion()
28
29              video = GlossVideo(videofile=vfile, gloss=gloss)
30              video.save()
31
32              # TODO: provide some feedback that it worked (if
33              # immediate display of video isn't working)
34              return redirect(redirect_url)
35
36      # if we can't process the form, just redirect back to the
37      # referring page, should just be the case of hitting
38      # Upload without choosing a file but could be
39      # a malicious request, if no referrer, go back to root
40      if request.META.has_key('HTTP_REFERER'):
41          url = request.META['HTTP_REFERER']
42      else:
43          url = '/'
44      return redirect(url)
```

On line 11 of the code listing above, the gloss_id submitted in the gloss_id field of the `VideoUploadGlossForm` is used to retrieve the gloss_id's corresponding gloss from the `Gloss` database, which is defined in the dictionary app. With this gloss retrieved from

the `Gloss` database, the video is saved, on line 29 and 30, to the GlossVideo database of the video app.

While it is true that a video is, logically, associated with a gloss, there's no need for a video to store the foreign key of its associated gloss; instead, the gloss_id of the associated gloss can be stored as a string of characters.

Here is how the `GlossVideo` model was changed so that it stored the `gloss_id` as a character string:

```
1  class GlossVideo(models.Model, VideoPosterMixin):
2      """A video that represents a particular idgloss"""
3      videofile = models.FileField("video file",
4          upload_to=settings.GLOSS_VIDEO_DIRECTORY, storage=storage)
5      #gloss = models.ForeignKey(Gloss)
6      gloss_id = models.CharField(max_length=50)
7      ## video version, version = 0 is always the one that will be displayed
8      # we will increment the version (via reversion) if a new video is added
9      # for this gloss
10     version = models.IntegerField("Version", default=0)
```

The code listing above is part of the `GlossVideo` model in `models.py`. On line 6, it can be seen that the gloss_id is being stored as a character string, and that no foreig key is therefore required.

There was another dependency in views.py that was related to glosses. Here is the `iframe` function, the function in which the dependency occurred.

```
1  def iframe(request, videoid):
2      """Generate an iframe with a player for this video"""
3      try:
4          gloss = Gloss.objects.get(pk=videoid)
5          glossvideo = gloss.get_video()
6
7          if django_mobile.get_flavour(request) == 'mobile':
8              videourl = glossvideo.get_mobile_url()
9          else:
10             videourl = glossvideo.get_absolute_url()
11
12         posterurl = glossvideo.poster_url()
13     except:
14         gloss = None
15         glossvideo = None
16         videourl = None
17         posterurl = None
18     return render_to_response("iframe.html",
19                 {'videourl': videourl,
20                  'posterurl': posterurl,
```

```
21                  'aspectRatio': settings.VIDEO_ASPECT_RATIO,
22                   },
23                  context_instance=RequestContext(request))
```

What this code above is doing is the following: some signs are *homophones*. That is, words of different meanings sharing the same sign. So for example, the words *dog* and *biscuit* , if they shared the same sign, would be classed as homophones of one another.

The problem then arises that the video of a sign will be stored muliple times if that sign is shared by multiple homophones. Due to the waste of space that would result from this, Signbank only stores a video for one of the homophones in a set of homophones. This means that, in order for a word to find its associated video, it must first determine whether it has any homophones, and then it must determine for which one of the homophones the video is stored. The logic that searched for homophones is inside of the dictionary app, however, the search for the homophone is initiated in the video app, on line 5 of the above code listing. It would be better if the search for homohpones was initiated in the dictionary app.

Unfortunately, due to time constraints, this could not be implemented. Developers picking up from where this project left off should look into this. More will be said on it in the unreseolved issues section of this document.

There was one final dependency issue that will be mentioned here. The program `ffmpeg` is used by the video app to extract a frame from a video. This frame serves as the thumbnail for the video. Steve, the supervisor of this project, raised whether it would be possible to replace this with something that was written purely in Python rather than depend on the external `ffmpeg` program. Unfourtunately, no purely Python solutions were found.

3.7.3 Upgrade the cosebase

The code for the video app was successfully upgraded to Django version 1.10 and Python version 3.5. Compatability was also maintained with Python Version 2.7.

As was done with the feedback app, the way that a success message was displayed to the user was changed. Now, when the adding of a video is sucessful, the user gets redirected to a view called `successpage`, which displays a sucess message to the user. Here is the new function:

```
1  def successpage(request):
2      # If there is a success message to display
3      if messages.get_messages(request):
4          return render(request, "video/success_page.html")
5      else:
6      # If not go back to the index page
7          return redirect('/')
```

On line 3, the Request object is searched for sucesss messages. If one is found, then the `success_page.html` template is rendered back to the user; otherwise, the user is

redirected back to the index page of the website.

On second thought, it would have perhaps been better to redirect back to the dictionary entry for which the video was uploaded, and to display a succes message there. Again, developers picking up from where this project left off could look into this. The reason why the alternative solution was not done in the first place was that I did not have a strong enough knowledge of Django at the time of writing the video app to know how to do it. That's why the gantt chart, figure 1.3, shows *leaning django* as a task that went on for the life of the project.

3.7.4 Writing documentation

Like for the feedback app, the cookiecutter-django-package created a README that contained some initial documentation written in the Sphinx (Restructed text) format. To this initial documentation, provided by cookiecurret-django-package, was added an explanation on how to set up the ffmpeg program to work with the video app. What one must do is install the `ffmpeg` program that is compatible with one's computer system. Then, in `settings.py`, one must set the variable `FFMPEG_PROGRAM` equal to the path to the `ffmpeg` program. On Ubuntu, `ffmpeg` can be installed with the command `sudo apt-get install -y ffmpeg`, and it's usually, put into the `/usr/bin` directory. Of course, for Mac and Windows, it will be totally different; that's why it's worth explaining in the documentation.

Three other variables, `VIDEO_UPLOAD_LOCATION`, `GLOSS_VIDEO_DIRECTORY`, and `MEDIA_ROOT` were also documented because they can be confusing. The `MEDIA_ROOT` variable should be set equal to the root directory in which all uploaded videos are to be stored.

The variables `VIDEO_UPLOAD_LOCATION`, and `GLOSS_VIDEO_DIRECTORY`, are sub-directories of the `MEDIA_ROOT` directory; `VIDEO_UPLOAD_LOCATION` stroes the videos that were uploaded by users as feedback, and `GLOSS_VIDEO_DIRECTORY` stoes all the videos of signs.

3.7.5 Writing tests

The test writing for the video app was exceptionally successful. A code coverage of 91 percent was achieved, greatly exceeding the requirement of 85 percent.

In total, 31 tests were written, all of them unit tests. (Functional tests were not written because there was no interface defined within the video app with which users could interact.)

The tests were split across three different files:

- `test_views.py` – For testing the functions in `views.py`
- `test_urls.py` – For testing the URL regexes in `urls.py`
- `test_models.py` – For testing functions defined in `models.py`

The test set up for the video app was different from the feedback app in some respects. One reason for that was that the video app required test data – a test video. The test

video, used in the tests, was stored inside of the tests directory, and further inside of the testmedia directory.

First, the tests inside of `test_views.py` will be discussed.

The tests in `test_views.py` test the functions defined in `views.py` As with the the tests in the feedback app's `test_views.py`, the the tests of the video app's `test_views.py` use the `RequestFactory` rather than the simpler `Client` object, and they use the `creaet_request`, and `create_user` methods, already discussed in section 3.6. There is a substantial difference, however. Observe the following except from the `test_views.py` of the video app:

```
1  class AddVideoTests(BaseTest):
2      def setUp(self):
3          BaseTest.setUp(self)
4          self.factory = RequestFactory()
5          # the url is irrelevant when RequestFactory is used...
6          self.url = '/addvideo/'
7          self.success_url = '/success/'
8          self.data = {"gloss_id" : "3", "videofile" : self.videofile}
9
10     def test_add_video_view_redirects_to_success_page_after_successful_
11         '''
12         The add video view should redirect
13         to the success view on succecssful upload of a video.
14         '''
15         request = create_request(url=self.url,
16             method='post', data=self.data)
17         response = addvideo(request)
18         self.assertEqual(response.status_code, 302)
19         self.assertEqual(reverse('video:successpage'),response.url)
20
21     def test_add_view_redirects_to_index_if_no_video_uploaded(self):
22         '''
23         The add video view should redirect to the index view if
24         no video is uploaded.
25         '''
26         request = create_request(url=self.url, method='post')
27         response = addvideo(request)
28         self.assertEqual(response.status_code, 302)
29         self.assertEqual('/',response.url)
30
31     def test_add_view_redirects_to_referer_if_referer_present_and_error
32         '''
33         If the referer attribute in the request is present,
```

```
34          then the view should redirect there on error.
35          '''
36          request = create_request(url=self.url, method='post')
37          test_referer = 'test/test'
38          request.META['HTTP_REFERER'] = test_referer
39          response = addvideo(request)
40          self.assertEqual(response.status_code, 302)
41          self.assertEqual(test_referer, response.url)
42
43      def test_add_video_calls_reversion_on_existing_videos(self):
44       '''If addvideo is called for a gloss_id that already contains
45            a video, then that video should go through reversion
46       '''
47       request = create_request(url=self.url, method='post', data=self.data)
48        # First, create the video
49        gloss_id = 3
50        vid = GlossVideo.objects.create(videofile=self.videofile, gloss_id=gloss_
51        # Now, add a video via addvideo
52        response = addvideo(request)
53        print(response)
54        # There should now be two videos for the gloss_id
55        videos = GlossVideo.objects.filter(gloss_id=gloss_id).order_by('version')
56        self.assertEqual(len(videos), 2)
57        # The first video in videos has version 0
58        self.assertEqual(videos[0].version, 0)
59        # The second video in videos has version 1
60        self.assertEqual(videos[1].version ,1)
61        # The second video's name should end in.bak
62        self.assertIn('.bak', videos[1].videofile.name)
```

This except of code contains the tests that test the `addvideo` function. As can be seen, they are written in a similar style to that of the feedback app's test – self-explanatory test case names, docstring, etc. However, there is a substantial difference. Note that on line 1, the `AddVideoTests` class inherits from `BaseTest`, and not `django.tests.TestCase`, as was done with the feedback app's tests.

`BaseTest` is a simple class, defined in the tests directory, that performs two useful functions:

- Before each test is run, it sets up the test video, stored in the testmedia directory, that is to be used in the test.
- After each test, it deletes the contents of the `VIDEO_UPLOAD_LOCATION`, and `GLOSS_VIDEO_DIRECTORY` directories. Since the tests manipulate these two directories while they are running, it's necessary to restore them to their initial state after each test finishes.

Here is the code of the `BaseTest` class:

```
1  import shutil
2  import os
3
4  from django.test import TestCase
5  from django.core.files import File
6  from django.conf import settings
7
8
9  class BaseTest(TestCase):
10     '''
11     This class simply defines the setUp() and
12     tearDown() functions common to all the tests.
13     '''
14     def setUp(self):
15         # this is the path to the video used by the tests
16         self.vidfilename = os.path.join(settings.MEDIA_ROOT,'video.mp4'
17         self.videofile = File(open(self.vidfilename, "rb"),
18              "12345.mp4")
19
20     def tearDown(self):
21         # After each test, delete any left over files that
22         # were created by the test
23         shutil.rmtree(os.path.join(settings.MEDIA_ROOT,
24             settings.VIDEO_UPLOAD_LOCATION),
25             ignore_errors=True)
26
27         shutil.rmtree(os.path.join(settings.MEDIA_ROOT,
28             settings.GLOSS_VIDEO_DIRECTORY),
29              ignore_errors=True)
```

As can be seen on line 9, `BaseTest` inherits from `django.test.TestCase`. This means that, when the classes in `test_views.py` inherit from `BaseTest`, they are also inheriting from `django.tests.TestCase`. So none of that functionality, or behaviour, is lost.

The `SetUp` function, on line 14, runs, automatically, before each test case defined in a class that inherits from `BaseTest`. First, `SetUp` locates the test video in the file system, on line 16, and then, on line 17, `Setup` turns th located video into a `File` object, which renders it ready to be used by the test case waiting for `SetUp` to finish.

`TearDown` automatically runs after each test case. It deletes the contents of the `VIDEO_UPLOAD_LOCATION`, and `GLOSS_VIDEO_DIRECTORY` directories using the `shutil.rmtree` function. Now the next test that runs can start out with clean directories, and no lingering data can thus affect it.

So BaseTest was a useful class that ensured that each test case was independent from

the others, thus brining the unit tests of `test_views.py` closer to the ideal unit test.

The tests in `test_urls.py` will now be explained.

The tests in `test_urls.py` serve the same purpose as the tests in the feedback app's `test_urls.py`. See section 3.6 for a thorough explanation on the workings of these tests; the exlapnation won't be repeated here.

The tests in `test_models.py` will now be explained.

The tests in `test_models.py` test the functions defined in `models.py`. Unlike the feedback app's `test_models.py`, the video app's `test_models.py` is substantially larger. This is due the fact that the video's `models.py` contains more functions than the feedback's `models.py`.

The original video app actually had a couple of tests in `test_models.py` already. These tests were changed slightly, however. They were changed so that they inherited from `BaseTest` rather than from `django.tests.TestCase`, and they were distilled somehwat. What is meant by *distilled* will be illustrated by two code listings; the first is of a test in the original `test_methods.py`, and the second shows the test after it was changed.

```
1  def test_GlossVideo_create(self):
2    """We can create a GlossVideo object"""
3    vid = GlossVideo.objects.create(videofile=self.videofile)
4    self.assertTrue(os.path.exists(vid.videofile.path),
5        "vidfile doesn't exist at %s" % (vid.videofile.path,))
6    vid.delete_files()
7    self.assertFalse(os.path.exists(vid.videofile.path),
8        "vidfile still exists after delete at %s" % (vid.videofile.path,))
```

```
1    def test_GlossVideo_create(self):
2          '''
3          An instance of the GlossVideo model should
4          correspond to an actual video in the filesystem,
5          sroted in the correct location.
6          '''
7          vid = GlossVideo.objects.create(videofile=self.videofile)
8          self.assertEquals(os.path.dirname(os.path.dirname(vid.videofile.name ))
9              settings.GLOSS_VIDEO_DIRECTORY)
10         self.assertTrue(os.path.exists(vid.videofile.path),
11             "vidfile doesn't exist at %s" % (vid.videofile.path,))
12
13     def test_GlossVideo_delete(self):
14         '''
15         An instance of the GlossVideo model should
```

```
16          be deletable. The file which it represents
17          should be removed from the file system.
18          '''
19          vid = GlossVideo.objects.create(videofile=self.videofile)
20          vid.delete_files()
21          self.assertFalse(os.path.exists(vid.videofile.path),
22              "vidfile still exists after delete at %s" % (vid.videofile.
23          # the poster should also be deleted
24          self.assertEqual(vid.poster_path(create=False),None)
```

In the first code listing, the test `test_GLossVideoCreate` tests both the creation and deletion of the video. To bring it closer to the ideal unit test, the creation of a video, and the deletion of a video, were separated into two different test cases, as can be seen in the second code listing.

Perhaps the reason for why the original code did not separate between the testing of the creatio, and deletion of a video was due to the fact that the video had to be deleted to esnure that the next test that was to run started out with clean directories. This concern was removed in the second code listing by the useful `BaseTest` class, which cleans the directories after each test.

The last thing that must be said about the tests of the video app is how Coverage was set up. Because the video app contains two files of Python source code that are not used anywhere in the video app, they had to be excluded from the Coverage calculation. This was achieved in the `.travis.yml` thus:

```
1  language: python
2
3  python:
4    - "3.5"
5    - "3.4"
6    - "2.7"
7
8  before_install:
9    - pip install codecov
10   - sudo apt-get -qq update
11   - sudo apt-get install -y ffmpeg
12
13 install: pip install -r requirements_test.txt
14
15 script: coverage run --source video --omit=*/fields.py
16
17 after_success:
18   - codecov
```

On line 15 of the above code listing, the `fields.py` file is excluded from the the Coverage calculation because it's not used anywhere in the Video app. To exclude another

file, another file can be appended to line 15, preceded by a comma.

3.7.6 Conclusion

In conclusion, the modularising of the video app was a success, although there still remains work to do. A code coverage of 91 percent was achieved; a dependency with the dictionary app was removed; documentation was written up; and the code of the video app was upgraded to Python 3.5, and Django 1.10.

The `iframe` function of `views.py`, will, however, require more work. This will be discussed further in the section on future work.

3.8 Modularise the registration app

Signbank has user accounts. These accounts are used to control access to various resources. For example, the admin, and only the admin, is allowed to edit the dictionary, and view, and delete feedback.

The registration component of the original signbank code is, according to the supervisor of this project, Steve, in a bad shape, and therefore not worth refactoring or salvaging. Instead, it was decided that a third party registration app be used.

A registration app that could be easily integrated with the other Signbank apps, and that provided social login – the ability to login via, for example, Facebook – was sought. In the end, a Django app called `allauth` was decided upon to provide the registration component.

`allauth` is recommended by the creators of Django [16]. Here is its GItHub Page. https://github.com/pennersr/django-allauth.

Because it's a third party app, no tests, or documentation had to be written for it; `allauth` has its own suite of tests, and it has its own documentation, hosted here: https://django-allauth.readthedocs.io/en/latest/.

Integrating the app with the apps of signbank is very simple. One need only install `allauth`, and then add it to the `INSTALLED_APS` variable in `settings.py`. Other variables must also be defined in `settings.py`; all of this is explained in the documentation of `allauth`.

3.9 Modularise the dictionary app

The dictionary app defines the data that are stored in the dictionay; it handles the searching for these data, the viewing of these data, and the editing of these data. The dictionary app is subsequently the most critical component of Signbank.

Unfortunately, due to time constraints, only the viewing and searching of data in the dictionary were implemented. The editing of the data in the dictionary, done by the administrator, will have to be dealt with by future developers, picking up from where this project left off.

Figure 3.4: The sign for the keyword dog

Here is the GItHub page of the dictionary app created in this project: https://github.com/hujosh/signbank-dictionary

3.9.1 Overview of the dictionary app

The structure of the data defined in the dictionary app is rather complicated. It was mostly devised decades ago, long before the Signbnak website existed. [17]

Let's begin with the most familiar concept: a word. In the `models.py` file of the dictionary app, there is a class called `Keyword`. This class represents an english word that is associatd with a sign.

```
1 class Keyword(models.Model):
2     """
3     An english keyword that will be a translation of a sign
4     """
5     def __str__(self):
6         return self.text
7     text = models.CharField(max_length=100, unique=True)
```

For example, say that the `text` field of the `Keyword` class, in the above code listing, is *dog*. That keyword will have an association with a sign, as in figure 3.4. Note that a sign can be associated with multiple keywords, as *beg*, and *dog* are in 3.4.

A keyword of the `Keyword` class is associated with a sign through a table called `Translation`. The `Translation` table is defined thus:

```
1 class Translation(models.Model):
2     """
3     An English translations of Auslan glosses
4     """
5     gloss = models.ForeignKey("Gloss")
6     translation = models.ForeignKey("Keyword")
7     index = models.IntegerField("Index")
```

A `Translation` instance has an associated keyword, an associated sign (the gloss), and an index. Several `Translation` instances can be associated with a single `Keyword` instance. This occurs when a keyword has multiple translations into Auslan. Such a thing occurs in figure 3.4, where, in the top right of the figure, 3 alternative translations of the keyword *dog* are visible (next to the text *Matches for the word dog*). The `Translation` class uses its index field to differentiate between the `Translation` instances that have the same keyword. In figure 3.4, the `Translation` instance associated with the keyword dog has an `index` of 1. The other translations associated with the keyword dog can be viewed by clicking on the three buttons at the top right of the image, respectively

So that leaves the definition of the sign itself. This is handled by the `Gloss` class of `models.py`. The `Gloss` class is very complicated, with dozens of fields and cross-references. It sufficies to say that an instance of the `Gloss` class is a sign. The two most important fields of the `Gloss` class are the `gloss_id` field, and the `sn` field. The former uniquely identifies an instance of the `Gloss` class, and the latter also uniquely identifies an instance of the `Gloss` class. However, the `sn` field is used not for identification but for ordering the signs (glosses) in the dictionary.

The dozens of other fields in the `Gloss` class record linguistic information, not really of interest to the goals of this project.

So that was a high-level overview of the structure of the data in the dictionary app. There are many other classes in dictionary's `models.py`, though they are not of interest to the goals of this project; they would be of interest to a linguist , however. Gaining an understanding of the structure of these data took a long time. Developers picking up from where this project left off will hopefully find the explanation given here useful. For a thorough tutorial on the structure of the data, see [17].

3.9.2 Making the app independent

The dictionary app has many dependencies with the other apps. This is inevitable given its pivotal role in the Signbank codebase. It wasn't appropriate, or safe, to start playing around with its dependencies; it was decided to leave it as it is. Perhaps future developers, picking up from where this project left off, could look into this.

3.9.3 Upgrade the cosebase

The codebase was successfully updated to the latest versions of Django and Python.

Several new functions were defined in `views.py`. They were `remove_crude_words`, `remove_words_not_belonging_to_category`, `paginate`, and `get_gloss_position` These functions were created to condense repeated blocks of code that occurred throughout `views.py` into self-contained functions.

3.9.4 Writing documentation

The app has its own documentation, created automatically for it by cookiecutter-django-package. Since the dictionary app required the definition of many variables in `settings.py`, writing documentation proved useful in keeping track of them, and will prove useful to new developers trying to understand the dictionary app.

3.9.5 Writing tests

The test writing for the dictionary app was a sucess. 89 percent code coverage was achieved, exceeding the requirement of 85 percent.

In total, 29 test cases were written. Unfortunately, due to time constraints, only unit tests were written.

The following test files were created:

- `test_views.py` – For testing the functions in `views.py`
- `test_urls.py` – For testing the URL regexes in `urls.py`
- `test_models.py` – For testing functions defined in `models.py`

These tests follow the same pattern that has already been described in section 3.6, and section 3.7. So explanations of the files won't be repeated here.

However, there are a few things worth mentioning.

Due to the complexity of the data of the dictionary app, it was not feasible to create test data in the test code. Instead, a Django test fixture was used. [18]

A test fixture is a `json` file that defines data to be loaded into the database. Here is an excerpted example of the `json` file used in the dictionary app

```
1  [
2   {
3           "pk": 1,
4           "model": "dictionary.keyword",
5           "fields": {
6               "text": "Aborigine"
7           }
8       },
9       {
10          "pk": 2,
11          "model": "dictionary.keyword",
12          "fields": {
13              "text": "Abraham"
14          }
15      },
16
17  ]
```

The `json` listing above defines two `Keyword` instances.

The fixture is used in a test file like so:

```
1  class SearchView(TestCase):
2      # Django will populate the database with the data
3      # in the file 'test_data.json'.
4      # You should read the django docs to understand how this works.
5      fixtures = ["test_data.json"]
```

In the above code listing, the fixture is loaded on line 5. For each test case of the `SearchView` class, the data defined in the fixture will be loaded into the database; after the test finishes, the data are remoed, thus ensuring clean, independent test cases.

The final thing to be said about the tests is an interesting but that they uncovered. In the original `views.py`, a decorator called `login_required_config` was defined. The purpose of this decorator was to control whether a view required one to be logged in to access it depending on the value of the `LOGIN_REQUIRED` variable in `settings.py`.

Here is the decorator:

```
1  def login_required_config(f):
2      """like @login_required if the ALWAYS_REQUIRE_LOGIN setting is True"""
3
4      if settings.ALWAYS_REQUIRE_LOGIN:
5          return login_required(f)
6      else:
7          return f
```

It makes the function require login if the `REQUIRED_LOGIN` variable is set to true. It seems like it should work. Two test cases were written to test the correctness of this. One, however, kept failing. Eventually, it was woked out that the `login_required_config` decorator only decorates the function when the function is defined, not necessarily after the `LOGIN_REQUIRED` variable is changed. With the cause of the bug known, it became possible to fix it. The following mofixied `login_required_config` decorator solves the problem.

```
1  def login_required_config(function):
2      '''
3      Like @login_required if the ALWAYS_REQUIRE_LOGIN setting is True.
4      '''
5      def wrapper(*args, **kwargs):
6          if settings.ALWAYS_REQUIRE_LOGIN:
7              decorated_function = login_required(function)
8              return decorated_function(*args, **kwargs)
9          else:
10             return function(*args, **kwargs)
11     return wrapper
```

This decorator modifies the function evey time that it is called, so it's able to pick up changes to the `LOGIN_REQUIRED` variable in `settings.py`. The idea for this was obtained from the book *Learning Python*, by Mark Lutz. [19]

3.9.6 Conclusion

Although there is still work to be done, modularising the dictionary app was a success. The desired code coverage was reached, documentation was written, and the codebase was upgraded to the latest versions of Django and Python. Improvements to the structure of the code were also made.

There is still a substantial amount of future work to do on the dictionary app. Unfourtunately, time constraints were the cause of this. Future work will be described in the future work section of this document.

3.10 Modularise the pages app

The pages app enables the administrator of the website to add content to the website, such as articles or blogs.

Unfortunately, due to time constraints, the modularisation of the pages app was not completed. That said, the pages app was, according to the supervisor of this project, Steve, unsalvagable. The future work section will touch on where to go with this going forward.

3.11 Modularise the attachments app

Unfortunately, due to time constraints, the modularisation of the attachments app was not completed. That said, the attachments app was, according to the supervisor of this project, Steve, unsalvagable. The future work section will touch on where to go with this going forward.

3.12 Integrate the apps

With the dictionary, feedback, video and registration apps modularised, it should be a very si

mple matter to combine the apps together into a unified Django project. An experimental Signbank project, combining all of the apps together, was successfully completed; however, due to time constraints, a GitHub page of the unified project has not been created.

This will be adddressed further in the future work section.

Overall, the project went reasonably well. Time constraints, notwithstanding. Much about Django and web-development was leaned, and that was, perhaps, the most useful

thing to come from this project.

Chapter 4

Conclusion

Here it shall be seen whether the tasks identified in section 1.3.1 were successfully completed.

The first three concrete tasks:

- Set up Travis CI
- Set up cookiecutter-django-package
- Set up Codecov

were successfully completed for the dictionary, video and feedback apps. However, due to time constraints, these three tasks were not completed for the pages app or the attachments app.

The next four tasks:

- Modularise the feedback app
- Modularise the video app
- Modularise the registration app
- Modularise the dictionary app

were successfully completed, though there does remain some work left to do in the video and dictionary apps.

The pages and attachments apps were not successfully modularised.

A unified Django project integrating the feedback, video, registraion and dictionary apps was created; however, it was experimental. Where this can go from here will be discussed in the future work section.

Chapter 5

Future work

This section lists, for each app, things which are left to do for future developers. Note that the registration app is not listed here. This is because it's a third party app, beyond the control of future Signbank developers.

Also note that the original codebase of Auslan Signbank is located here: https://github.com/Signbank/Auslan-signbank In order to complete most of the future work tasks identified in this section, it will be necessary to take the code that didn't make it into this project from that GitHub page, and to try to fit it into the various apps created for this project. Each app's GitHub page is given in each app's section of this document. After applying a change to an app, future developers can run the app's test suite, ensuring, to a reasonable degree, that nothing was broken by the change.

5.1 Future work for the feedback app

The feedback app was mostly completed. There remains only two things that are not finished.

5.1.1 Interpreter feedback

The interpreter feedback form is a form for the submission of feedback by interpreters. An interpreter is one who translates between, for example, Auslan and English.

Future developers will need to add the `InterpreterFeedback` model to feedback's `models.py`. That shouldn't be too difficult, though there is a foreign key, dependent on the dictionary app. The form `InterpreterFeedbackForm` will also have to be defined, in the feedback app's `forms.py` file. Lastly, a future developer will need to think of a how he will show interpreter feedback inside of the `show.html` template of the feedback app.

5.1.2 User uploaded video

Uses can upload a video when they submit feedback on a missing sign, and on general feedback. This functionality didn't make it into the feedback app created in this project

due to time constraints. The problem was that uploading a video in the feedback app had a dependency on the video app. Future developers will have to think about how they can bridge the two apps, whist maintaining reasonable independence.

5.2 Future work for the video app

The video app is mostly completed. There was only one thing that was not finished.

5.3 The iframe function

The `iframe` function, in the video app's `views.py`, was not completed. The reason for this was that it had a tricky dependency on the dictionary app. This dependency was discussed at length in section 3.7. Future developers should refer to the discussion in section 3.7 for guidance on how to proceed here.

The `iframe` function, responsible for the display of a video in the dictionary, is rather important, so it's a shame that it wasn't finished.

5.4 Future work for the dictionary app

The dictionary app is, unfortunately, only about half complete. The functionality that handles the searching, and viewing of a gloss, and word is complete; it's the functionality that handles the creation and editing of a gloss by the administrator that was not completed.

This functionality can be found in the `adminviews.py`, and `update.py` view of the original Signbank dictionary app.

Also, quite a few class methods of the `Gloss` class in `models.py` did not make it into the dictionary app of this project.

5.5 Future work for the attachments app and pages app

Due to time constraints, these two apps were not completed. In discussion with the supervisor of this project, Steve, it was said that these two apps are not worth refactoring; instead, a third party Content Managemenet System app (CMS) should be sought.

Future developers should look at the CMS `wagtail`. It's recommened by the creators of Django [16].

Finally, a few words will be said about what future developers can do to integrate all of the apps together

5.6 Future work for the integration

The integration won't be fully complete until all of the apps that constitute the integration are fully complete; so this is still a long way off. Once this is complete, however, integrating the apps should be as simple as installing each of the apps, and then adding them to the Django project's `INSTALLED_APPS` variable. Each of the apps require their own variables to be defined in the Django project's `settings.py` file. Each app's README explains what these variables are.

Finally, cookiecutter-django (https://github.com/pydanny/cookiecutter-django) is recommended as a template for the integrated Django project. Note that cookiecutter-django is not the same thing as cookiecutter-django-package; cookiecutter-django-package is a template for resuable apps, while cookiecutter-django is a template for a Django Project.

Chapter 6

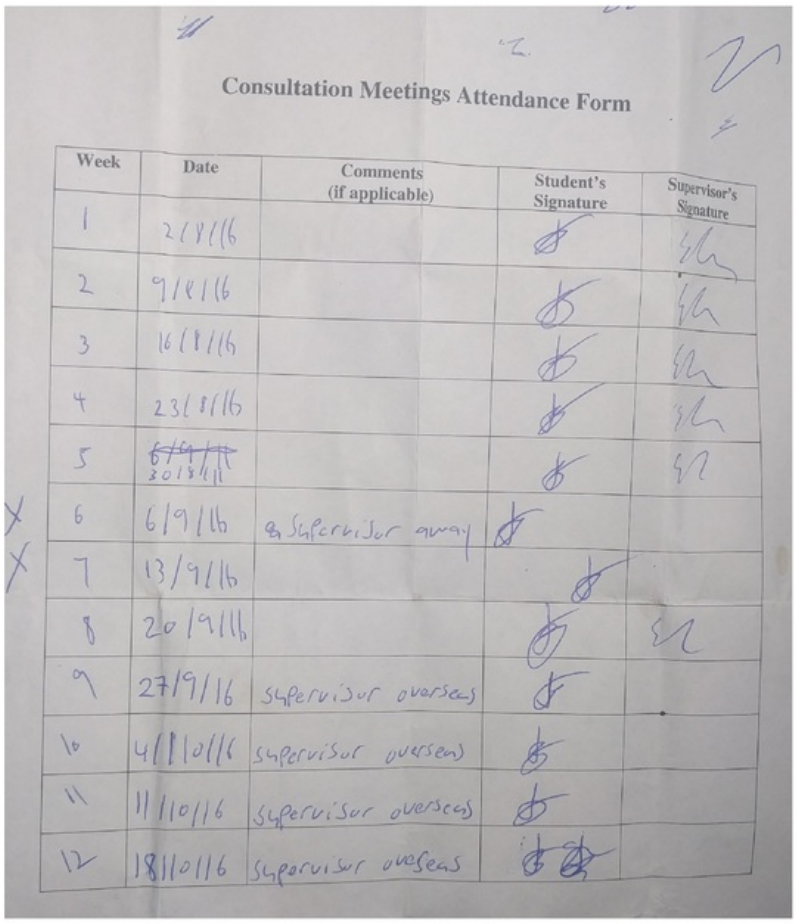
Abbreviations

MVC – Model View Controller

Appendix A

Attendance slip

Attached is the attendance slip. Note that my supervisor was overseas, in America, for the last several weeks. We communicated by email.



Consultation Meetings Attendance Form

Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
1	2/9/16			
2	9/9/16			
3	16/9/16			
4	23/9/16			
5	~~6/9/16~~ 30/9/16			
6	6/9/16	& Supervisor away		
7	13/9/16			
8	20/9/16			
9	27/9/16	supervisor overseas		
10	4/10/16	supervisor overseas		
11	11/10/16	supervisor overseas		
12	18/10/16	supervisor overseas		

Figure A.1: Attendance slip

Bibliography

[1] "Sign language," *Encyclopedia Britannica*, 2016.

[2] A. Holovaty and J. Kaplan-Moss, *The Django Book*. New York, New York: Apress, 2009.

[3] J. Knupp, "What is a web framework?" [Online]. Available: https://jeffknupp.com/blog/2014/03/03/what-is-a-web-framework/

[4] "Find your new favorite web framework." [Online]. Available: http://hotframeworks.com/

[5] "Django documentation," 2016. [Online]. Available: https://docs.djangoproject.com/en/1.10/

[6] H. J. Percival, *Test Driven Development with Python*. Sebastopol, California: O'Reilly Media, 2014.

[7] K. Harvey, *Test-Driven Development with Django*. Birmingham, UK: Packt Publishing, 2015.

[8] D. R. Greenfield and A. R. Greenfield, *Two Scoops of Django*. Los Angeles, California: Two Scoops Press, 2015.

[9] Pydanny, "cookiecutter-djangopackage." [Online]. Available: https://github.com/pydanny/cookiecutter-djangopackage

[10] I. Lewis, "Testing django views without using the test client," July 2015. [Online]. Available: https://www.ianlewis.org/en/testing-django-views-without-using-test-client

[11] *Selenium documentation*, 2016. [Online]. Available: http://www.seleniumhq.org/docs/01_introducing_selenium.jsp

[12] I. Bicking, *Testing Applications with WebTest*, 2016. [Online]. Available: http://webtest.pythonpaste.org/en/latest/

[13] JoshMock and idanzalz, "Difference between django-webtest and selenium," 2012. [Online]. Available: http://stackoverflow.com/questions/12448878/difference-between-django-webtest-and-selenium

[14] D. S. Foundation, "The messages framework." [Online]. Available: https://docs.djangoproject.com/en/1.10/ref/contrib/messages/

[15] Django, "Class-based views." [Online]. Available: https://docs.djangoproject.com/en/1.10/topics/class-based-views/

[16] J. Triplett, "5 favorite open source django packages," December 2015. [Online]. Available: https://opensource.com/business/15/12/5-favorite-open-source-django-packages

[17] T. Johnston, "The lexical database of auslan (australian sign language)," 1999. [Online]. Available: https://www.sign-lang.uni-hamburg.de/intersign/workshop1/johnston/

[18] D. S. Foundation, "Fixture loading." [Online]. Available: https://docs.djangoproject.com/en/1.10/topics/testing/tools/#django.test.TransactionTestCase.fixtures

[19] M. Lutz, *Learning Python*, R. Roumeliotis, Ed. O'Reilly Media, 2013.