quickQuote

A web application to investigate the use of video in the newsroom

UCL Msc CS Project Report

COMPGC99G Individual Project

Pietro Passarelli

01-09-2015

Student ID

947720

Internal Supervisor:

Dr. Graham Roberts, UCL

External Supervisor:

Ben Whitelaw, Times & Sunday Times

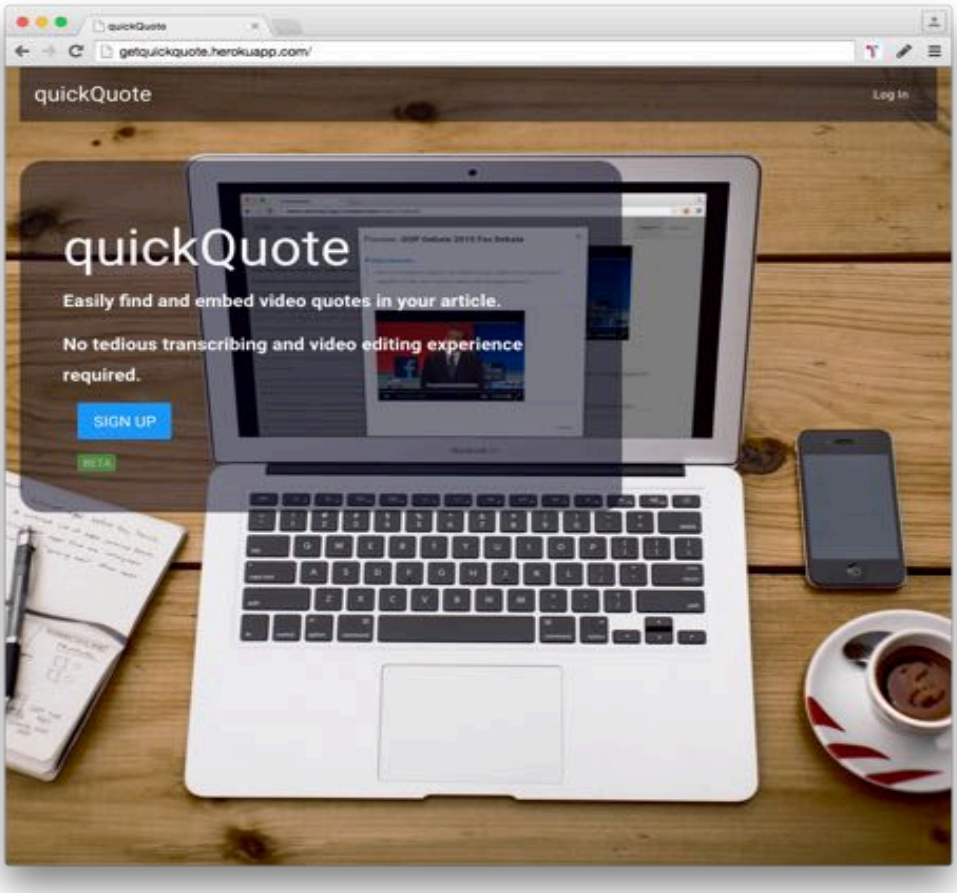
This report is submitted as part requirement for the MSc Computer Science degree at UCL. It is substantially the result of my own work except where explicitly indicated in the text.

Abstract

For journalists working on a news article for the web, embedding a video quote is a time-consuming activity and this often leads to the video not being included in the article altogether. The aim of this project is to produce a web application that can facilitate this process.

An investigation into the problem domain of working with video in a newsroom was carried out and a number of rapid prototypes were built using an iterative approach. Where speech-to-text, NLP APIs and ways to model a hyper-transcript were explored.

The project resulted in a working application that provides a comprehensive solution for journalists to be able to quickly find and export a video quote to embed in a news article.



Project page: http://times.github.io/quickQuote

Candidate Number or Name: 947720 – Pietro Passarelli

This student is registered as dyslexic / dyspraxic with UCL Student Disability Services.

Please refer to marking guidelines at:
http://www.ucl.ac.uk/disability/information-for-staff/dyslexiamarkingguidelines

CONTENTS

1. INTRODUCTION

1.1. Outline the problem

For a journalist working on a news article for the web, embedding a video quote is a time-consuming activity and this often leads to the video not being included in the article altogether.

There are many reasons for wanting to add a video clip of the quote to an article. This can add elements of non-verbal communication to the narrative of the written piece, in addition to giving depth to the quote and the tone of the speaker. The video can provide the emotional charge behind the words and contribute to a vastly richer user experience. This gives the reader a full understanding of every dimension of the quote, ie, the context, tone, emotion, humour etc. Without this, the viewer reads only the words and the full depth of the quote is not fully communicated. Essentially, adding video clips to quote lines ensures that no depth of meaning and content of the words behind the quote is missed by the user. For example, a written piece about quotes from a public figure such as Donald Trump, there is only so much a viewer can grasp by reading a quote such as:

> *"A five billion dollar web site, I have so many websites, I have them all over the place. I hire people they do a web site, it costs me three dollars." (CNN 2015)*

This quote was in criticism of a healthcare website and was said in a tongue-in-cheek tone with elements of humour and outrage. These elements are not immediately obvious to the user without seeing the video clip and how Donald Trump expressed himself.

In the current work-flow, once the journalist has the video file, they would have to find the quote, which often entails manually scrubbing through the video in search of that one usable sound bite, which can be something like finding a needle in a haystack, particularly if the video is lengthy. Additionally, transcriptions are not a viable option either, as often there is no time to wait for them or no budget, and without time-codes, transcriptions do not help to speed up the process (Passarelli 2015).

In a fast paced newsroom often there is no time for such a time-consuming process.

The aim of this project is to produce a web application that can maximise not only the depth of content but also the speed in which the content is produced.

1.2. Area of Interest

One reason why this project is interesting is because it utilises the new possibilities that have opened up with the introduction of the HTML5 video tag. As well as the

opportunities that have arisen in combining this with the increasing quality of speech to text technologies.

Traditionally video and audio have been like black boxes on the web, often having to recur to flash to provide video capabilities to pages.

With the introduction of the HTML5 video tag, Javascript libraries like videos make it easier to manipulate the video, treating it as a Javascript object. However this turns the video into “a ‘black box’ we can do something with, such as triggering events at defined timecodes. It does not allow us to directly obtain the content of the video in a programmatic way and to manipulate the result. An example would be to take from the video the content of the quote and then analyse this to identify keywords and key topics. Another example would be to search what has been said in the video, find the quote and trim the video segment accordingly.

I believe it is by combining video with its corresponding timecoded transcription that we can provide a direct programmatic solution.

1.3. The Challenges

There were two main challenges for this project. The first was centered around investigating the problem of how working with video and transcriptions in the newsroom could lead to an increase of time-consuming multimedia production. The second main challenge was then identifying an appropriate solution.

1.3.1. Investigation of the Problem

Investigating and defining the problem was one of the first challenges. The initial problem-hypothesis was formulated around researching ways to make working with video in the newsroom easier and faster. Identifying the set-backs journalists have was difficult because they themselves did not fully know what they were. This is discussed further in the investigation chapter, which outlines how through a series of proof-of-concept prototypes and discussions with journalists and newsroom stake holders, it emerged that a the lack of a system to quickly extract video quotes was the problem and it was the solution to this problem that was the second stage of the challenge.

1.3.2. A Comprehensive Solution

One of the challenges around this project is providing a comprehensive work-flow for the journalist and the user that is clear and easy to follow through from beginning to end, without requiring any training.

1.3.3. Video Transcriptions

The fastest way to be able to quickly select a caption from a video, is to search a transcription of that video. Therefore a challenge was to research and identify the most appropriate system to get the transcription of any given video. Human generated transcriptions were not an option because of time and resource constraints. Automated transcriptions needed to meet a certain threshold of acceptance from the user, as well as have accurate timecodes in order to be searchable and in sync with the video for quick feedback.

1.3.4. The Output

The output of the video quote selection system will be a text quote with the corresponding video clip associated with it. This needs to be something journalists can easily add to their news article with minimal or no configuration. The system also needs to be able to cut the original video source to trim it down to the video quote selection as a HTML5 H264 MP4 video.

1.4. Aims and Goals

1.4.1. Aims

My aim for this project is to use an R&D approach to identify an appropriate solution to an every day problem; that of using video and transcriptions in the context of newsrooms to facilitate and increase their multimedia production for the web, as well as develop a working web application proof of concept.

Additionally, throughout the process, an aspect of the project became to learn the following technologies:

- Javascript/JQuery to manipulate the DOM to implement the front end logic of the application.
- JSON and AJAX
- MVC design pattern of the Rails framework.
- Working with APIs, such as
  - –user authentication with google auth
  - –speech to text API
- Ruby Mine IDE
- How to model the domain of the application into back end classes, rails models, efficiently.
- VideoJs and HTML5 Video
- Testing
  - –Acceptance test - Selenium
  - –Unit Test - RSpec
- FFMPEG to trim and transcode video.
- Deployment on Deis / Heroku

- Refactoring components from open-source projects that were using hyper-transcript, with videojs and JQuery.

1.4.2. Goals

The main goal is to build a web application that makes extracting video quotes and publishing them to news articles much easier.

Goals for this project are to;

- Investigate the problem in the newsroom around working with video and prototype a compelling solution.
- build a web application that;
- given a video returns a searchable transcription in sync with the video.
- allows the user to select quotes
  - –allows the journalist user to preview their quote selection before exporting
  - –cut the video in the background once a quote selection is being exported.
  - –export the video quote clip as HTML5 MP4 H264
- produce a set of unit and acceptance tests to make the application as robust as possible.
- deliver a system manual
- deliver a user manual
- provide code documentation
- open-source the project.

1.5. Overview of the Project

My personal preference for developing applications is to use a lean agile approach (Eric Ries 2011) to software development, with an initial emphasis on researching and identifying a problem solution, working closely with users and stake holders through proof of concept paper sketches prototypes and research on existing user practices and work-flows. As done during the course of the UCL Msc for GC02 industry project working with the Audava Start-up (Passarelli Pietro et al. 2015) and for the Entrepreneurship module developing an autoEdit application (Passarelli Pietro 2015b).

However in the context of newsroom development at the Times and Sunday Times, this was not always possible, as it was required to produce an interactive prototype to demonstrate proof of concepts hypothesis to carry out what would be the equivalent of the “customer development” phase in the lean approach (Maurya 2012).

Working within these constraints over the whole project took an iterative approach with three main overarching phases:

1.5.1. 1.Preliminary Research

Identifying, researching and learning the technology stack. This consisted of becoming familiarised with the Times & Sunday Times development team's current technology stack.

1.5.2. 2.Hypothesis and prototype

Investigating the problem domain of the application. Making hypotheses, prototyping and investigating problems and solutions. Confront results with users and stake holders.

In this phase, a bottom up approach was used to make a prototype and ensure the core aspects of each component fully worked and integrated as a whole.

Once the appropriate problem and solution was identified, this was followed by a rapid prototype phase, to confirm such solution with the users and stake holders, in order to agree on user requirements, before moving on to implementation.

1.5.3. 3.Implementation

Followed by the final phase of re-factoring and implementation of the web application.

In this phase a top down approach was used, with more care put on re-factoring code into components.

A more detailed breakdown of the investigation carried out in this iterative process can be found in the investigation chapter.

See appendix for outline of phases and internship timeline.

1.6. Overview of Report

Chapter 1, Introduction The first chapter outlines the problem investigated, the area of interest, aims and goals and gives an overview of the project and the rest of the report.

Chapter 2, Context The second chapter is concerned with providing context and background research to the project, as well as sources, and similar solutions. It also introduces the software.

Chapter 3, Investigation As mentioned, the project was built with an iterative approach, investigating the optimal solution for a system that would allow journalists to work more efficiently with video in their news article publications. This section expands on the different phases of the R&D investigation.

Chapter 4, Requirements This chapter outlines the requirements for the final iteration of the video quote extractor application, the problem statement, list of requirements, use cases, and result of requirements analyses.

Chapter 5, Implementation The implementation of the application is discussed in this chapter. Design, architecture, components design, database and implementation details.

Chapter 6, Testing An overview of the testing strategy, the use of unit test and acceptance testing is presented in this chapter and a summary of test results.

Chapter 7, Conclusion A summary of what the project has achieved, as well as critical evaluation and recommendation for future work.

2. CHAPTER 2 CONTEXT

2.1. Background information

2.1.1. Newsroom Challenges

To best understand the challenges in developing software in a news environment, the leaked New York Times Innovation report was referred to.

The biggest issue highlighted in the report was that flagship projects such as *Snow Fall* (NY Times 2015, p36) were time consuming and a lot of effort went into the making of a one-off piece with a relatively short life span.

> *"We have a tendency to pour resources into big one-time projects and work through the one-time fixes needed to create them and overlook the less glamorous work of creating tools, templates and permanent fixes that cumulatively can have a bigger impact by saving our digital journalists time and elevating the whole report. We greatly undervalue replicability." (NY Times 2015, p36)*

The report also points out that the digital publication Quartz has a different approach:

> *"We are focused on building tools to create snow falls every day, and getting them as close to reporters as possible. I'd rather have a snow-fall builder then a snowfall."*
>
> *Kevin Delaney, editor of Quartz (NY Times 2015, p36)*

The NY Times competitors such as BuzzFeed, were instead able to separate form from content. The best example is a dialect quiz by the NY Times, that was widely popular and BuzzFeed, inspired by the format made a "Quizz builder" and after the first release, published 20 variations with minimal effort (NY Times 2015, p36). This consideration of building a system rather then a one-off editorial piece was the initial inspiration behind the making of the Interactive Debate Prototype at the Times Build The News Hackathon, discussed in more detail in the investigation section.

2.1.2. V.A.R.K.

The other consideration that guided the project came from VARK, the idea of the variety of learning styles (Neil Flemming 2015). Simply put, different people learn in different ways. Five main learning styles are identified - visual, auditory, reading and writing, kinaesthetic and multi-modal. The core concept is that each individual has one (or more) preferred learning style. Additionally but also more crucially, the same information can be delivered in different ways to meet the varying learning styles. Applying this consideration to the publishing of news articles pushes us to reconsider

the use of multimedia to engage a wider audience, taking into account the variety of learning styles.

This consideration guided many of the assumptions through the investigation up to adding the corresponding video segment to a text quote in the final web application.

2.2. Related Work and Similar Solutions

2.2.1. Similar solutions

The following paragraphs consider similar solutions and different approaches to working with video transcriptions on the web. These will be divided into tools and editorial. Tools are those projects that allow the user to create something such as providing a video and returning transcriptions. Editorials are projects that focus more on the content and the delivery of information. Beginning with the editorial project, it will soon become apparent in this context that they are often a starting point for developing tools.

Editorial

Aljazeera Debate Obama - Romney

Aljazeera Obama-Romney Debate - Mark Boas

In the ‘Aljazeera Debate Obama - Romney’ (Mark Boas 2012) the word-accurate hyper-transcript was done manually.

What is interesting about this project is that it gives a way into the video, allowing the user to search the text and provide some basic infographic through a pie chart. Then, on the search terms, the user can see how many times the respective candidates have mentioned a certain word.

The project is open sourced on github (Marb Boas 2013).

The code from this open source project was researched both at *Build the News Hackathon* and at *BBC Times News Hackathon*, both of which are discussed in the subsequent chapter of investigation. The major difference between the implementation of the hyper-transcript in this project from the implementation in quickQuote is that this project uses the popcornJs (Mozilla 2010) library and JPlayer (Happyworm LTD 2009), while quickQuote uses videoJs (Brightcove Inc 2010) and JQuery (jQuery Team 2006). This is discussed in more detail in further chapters.

In this project, it is also interesting that the developer inserted an "Easter egg" that would let the user play only the sentences with a certain word in it, adding the query in the URL ?k=economy&t=1000. Where "economy" could be any keyword, and t is for the time interval, we wish to assign to each sentence.

http://www.aljazeera.com/indepth/interactive/2012/10/201210179222591398O.html?k=economy&t=1000

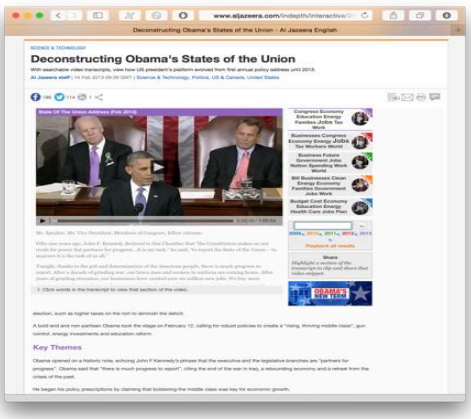
(it seems to work best on Firefox)

However this was not made available as a function in the actual GUI of the application.

This feature is similar to the video grep project discussed below.

Aljazeera Obama State of the Union Archive

This project (Al Jazeera staff 2013) generalises the previous project at an archive level. However, again, as far as I am aware, no automation was used to deal with the transcriptions or the different video input.



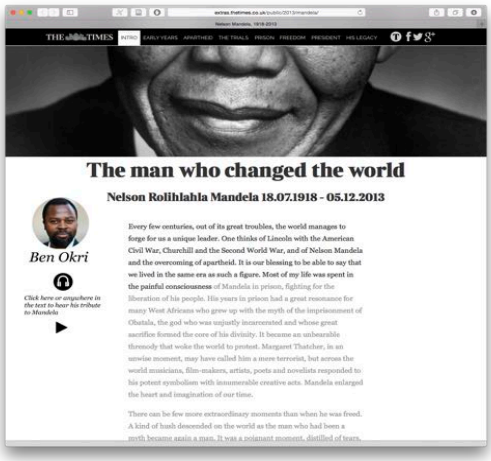
Aljazeera Obama state of the Union - Mark Boas

This is where the idea for a system that could take a video and perform some kind of analysis was formulated.

Another interesting aspect of this project, is that there is a section providing a summary for each video.

Mandela Speech

The Times & Sunday Times published (Ændrew Rininsland 2013) a piece on a Mandela speech, using the hyper audio pad (Mark Boas 2013a) on the back of Aljazeera hyper-debates. However the transcription was done manually using the programs VLC and Notepad and subsequently a software to make subtitles, adding the text with in and out points, to then get an srt to convert to a hyper-transcript using the Boas (Mark Boas 2013b) hyper-audio converter.



Mandela Hyper-transcript – Ændrew Rininsland

The piece used audio only. What is interesting about it from a user point of view, is that the text changes colour as the audio is playing, which keeps the correspondence between the order of the audio and the order of the text.

This interactive was built using the Doctop (Times Digital 2013) library to use a Google doc as a back-end and Bootstrap (Twitter 2012) was used for the front-end.

However the project relying on Google doc as a backend was then affected by Google changes in the API, which had the ability to break the project. A better solution to making the most of using Google docs as a CMS would be to keep Google docs in sync with a database that updates the application. This way, if the connection between the database and Google docs were to drop due to changes in the API, the front end of the application would not be effected.

Tools

F5 Transcription

F5 is a transcription tool (Thorsten Pehl, Thorsten Dresing 2010), made by German academics. It enables the user to transcribe a video manually, with a convenient interface which allows the application to provide timecode stamps on new line input.

F5 Transcriptions

F4 Analysis

In subsequent versions of F5 they released F4 Analysis (Thorsten Pehl, Thorsten Dresing 2013), which takes those same transcriptions and adds human tagging to it for qualitative analyses.

What is notable about the idea of providing insight into text of transcription coding is the content with tags and adding summary. This, however, is done manually by the user. Something that arose during the investigation was an interest in automating this part.

oTranscribe

Made by newsroom developer at WSJ (Elliot Bentley 2013), oTranscribe is very popular with journalists.

It facilitates transcriptions of video and audio, with intuitive interface and a few keyboards short-cuts.

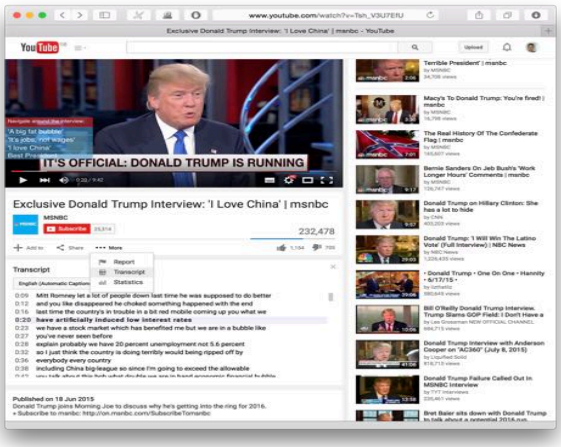
oTranscribe - WSJ

However, there is no automation of the transcription process.

YouTube Captioning

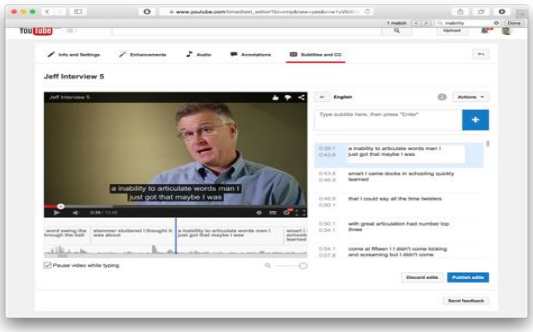
YouTube has a captioning service and clickable transcriptions. However their transcription is mostly aimed at subtitles and even in the transcription view, it is not

word accurate, it is line accurate, perhaps because it uses the subtitle file generated during the captioning to model the hyper-transcript.



YouTube Captioning Transcriptions

The advantage of YouTube automated captioning is that it is constantly improving and becoming more and more accurate.



YouTube editor to edit captions

YouTube also provides an API (Google YouTube 2015b) to retrieve captions although it is not clear how the user would go about retrieving automatically generated captions through the API, as it requires a language to be set before the caption is generated.

However, most importantly, the disadvantage is that in version two, Google announced it was going to discontinue access to the captioning service through the API, (Google YouTube 2015a) even though this is still available through version 3 of the API (Google YouTube 2015b). Automatically generated transcriptions are a big component of this project. Because of the rate at which speech to text technologies are improving, it is difficult to predict which speech-to-text API is going to be the best option long term. It was therefore decided to define an interface for the speech-to-text API in order to be able to change the speech-to-text provider in the future, should the

service discontinue or a better solution is found. This concept is explored in more depth in further chapters. Furthermore, the choice of not relying on Google/YouTube is there, for there is no commitment on their part to maintaining this service as part of the API long term.

Spoken Data

Despite YouTube captioning being used for a great part of the prototyping to get the srt file of the transcription as an input to the application before integrating the speech-to-text API component, when it came to implementing such components, it became apparent that Spoken Data (Igor Szoke 2015) was a better fit.

Spoken Data is mostly tailored for small teams of professional transcribers, but also individuals requiring transcriptions.

With it being a smaller company, Spoken Data was more responsive to making changes and improvements. For example, a user needed to be able to delete a video uploaded to their system through the API, (to avoid issues of duplicates in their dashboard if the user uploads, deletes and uploads again the same video in an application) and they were very quick at implementing that feature.

They also provided other functionalities such as a view to edit the transcription, that allows users to edit recording subtitles through a tokenized access URL. That is available through the API.

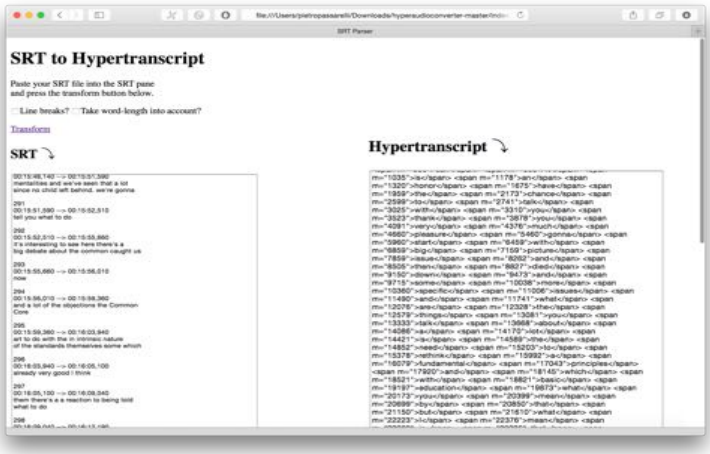
http://spokendata.com/transcription/846/e6881695913bb807b81cf87ce79f4bfd3c84d0c4

(from Spoken Data demo account)

This could be useful to give the users of this project's app a way to edit automated transcriptions without having to implement it from scratch.

Hyper-Transcript Converter

Eventually Mark Boas (Mark Boas 2013b) automated the process of passing srt files into a hyper-transcript with a hyper-transcript converter written in JavaScript. This library was developed to support the hyperaudio pad project, (Mark Boas 2013a) but it is all client side Javascript.



Hyper-audio converter - Mark Boas

From the line-accurate time code of the srt file it works out a word accurate hyper-transcript, with each word in a span tag with a relative time attribute.

An analysis of this algorithm can be found in the appendix, as this was converted from Javascript to Ruby to move this conversion logic in the backend of the application.

Video Grep

Video grep (Sam Lavigne 2014b), as the name suggests, is a command-line-utility to search the audio in a video file and return with cut segment as a new file.

Video Grep is a Python library for editing videos that provides the user with a search tool to search and return all the clips of sentences where a specific word is mentioned (Sam Lavigne 2014c). For an example, the developers used this same tool to extract all the parts in a speech where the White House press secretary says ‘what I can tell’ (saaaam 2014).

This uses offline speech to text library pocket sphinx (The Sphinx group Carnegie Mellon University 2015b), a C based lightweight version of the Java based CMU sphinx library (The Sphinx group Carnegie Mellon University 2015a), to transcribe the video and generate a subtitle file.

The subtitle file associated with the video is used as a starting point to find the words and then cuts the corresponding video using the Python library moviepy (Zulko 2014), which is built on top of FFMPEG (FFmpeg team 2000).

A Mac app version (Sam Lavigne 2014b) (using electron (Kevin Sawicki 2015) of Video Grep was recently being released providing a GUI to the command-line utility.

Video Grep - mac app using electron

No level of granularity control on selection, or hyper-transcript was used.

This project is reminiscent of Cassetteboy (cassetteboy 2006) - a well-known video artist, who utilised video editing mash ups, generally of politician's speeches for parody.

However in this interview he mentions he uses a manual and time-consuming workflow to search and identify the relevant video segments for his edit.

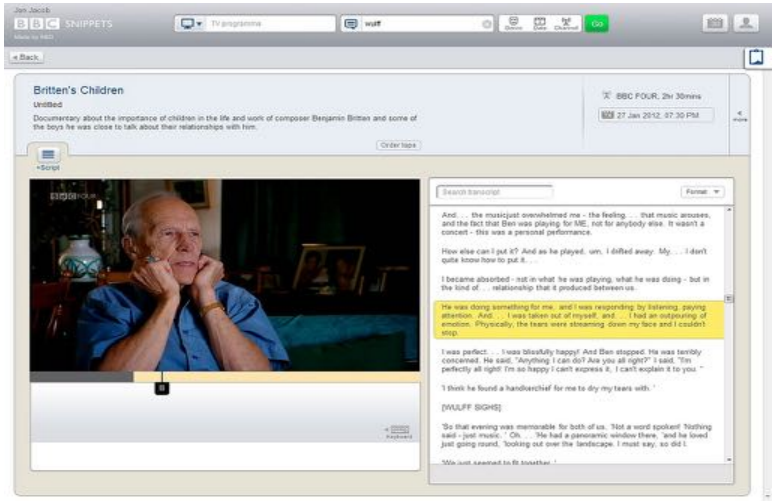
> *Making these videos takes a lot longer than many people might assume, doesn't it…*
>
> *"It is a long time. The original Apprentice video, we were working on it on and off for a couple of months. This new one we've done is well over a month. If you think about just watching the material, say a boardroom scene lasts half an hour, it will actually take an hour, an hour and half to get through it the first time around. You just keep stopping it, taking samples of bits that might be useful, filing them and then categorising them. So you can only do a few episodes per day."*
>
> *"You also start to go a bit mad. It's quite a strange way of working and watching a show. You are listening, but you are listening to individual words rather than sentences. You try to ignore as much as possible the actual meaning and focus on what you twist it to. That takes quite a lot of concentration and you couldn't watch more than 6 or 7 episodes a day, because you'd drive yourself absolutely crazy." (Alex Fletcher 2014)*
>
> *There is also an audio grep version (Sam Lavigne 2014a).*

BBC Snippets

This application (BBC R&D 2011a) allows BBC employees to search for programs in the BBC Archive (only within the last 5 years), searching across the text of the

subtitles. It uses a form of hyper-transcript to keep audio and video in sync, allowing the user to cut video segments from past videos.

However the subtitles seem to be generated using OCR from onscreen subtitles rather then using speech to text technology (BBC R&D 2011b) .



BBC Snipets

The hyper-transcript seems to be more sentence-accurate rather then word-accurate.

As far as I am aware there is no real system to deal with un-subtitled videos and transcriptions generation is not in place.

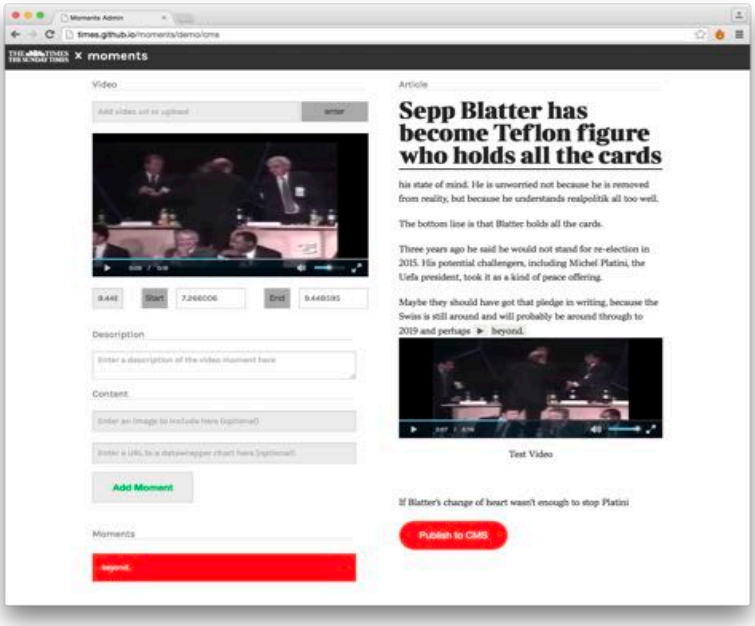
No form of analysis on transcription, auto-tagging or summarisation was provided.

Most importantly, the cutting of the video segment is done at the level of the video timeline and not of the text.

Moments - Prototype

Moments is a web application proof of concept that allows a user to add an inline drop-down video snippet to a text article. It uses a CMS for the journalist and has a published view for the viewer. It does not use transcriptions.

From this project I initially got the idea for an inline text dropdown of a video element.



Moments

It was prototyped by the Times Digital Team and is the winner of the 2014 Editors Lab at GENSummit2014. However as mentioned this is a proof of concept and not a fully working application.

Hyper Audio Pad

Hyper Audio Pad by Boas (Mark Boas 2011) allows the user to make remixes of videos based on the text of their transcriptions/subtitles.

It uses Boas hyper-audio converter component (Mark Boas 2013b).

prEdit

This is a similar solution to hyper-transcript but is desktop based and provides integration with editing software. The interface is not very intuitive which limits user interaction.

2.2.2. Related Work

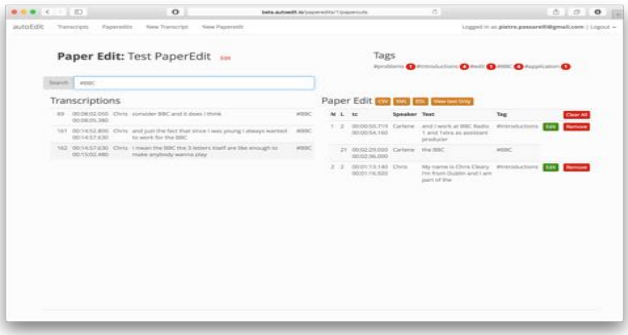
With a background in documentary production, the author has a long standing interest for working with transcriptions effectively, so here are some previous projects that influenced the current one.

Examples of previous projects undertaken based around an interest in transcriptions are listed below:

AutoEdit - for Entrepreneurship Module

autoEdit - ‘Digital paper editing’ (Passarelli Pietro 2015a) is an application that enables the user to make a script from a selection of lines from the transcriptions of interviews. The user can then export the file as an EDL. This can be opened with the editing software of the user’s choice to reconnect the corresponding video into a sequence, and finish working on the rough-cut assembly (Passarelli Pietro 2015c).

autoEdit had a component for parsing srt/sbv files. Initially, during the quickQuote project, the idea of utilising a similar parsing component occurred. However, a line accurate rather than a word accurate hyper-transcript would be more beneficial.



autoEdit

Additionally, autoEdit does not support uploading video and cutting video segment, but instead uploads srt files then exports EDL file.

Build the News Proof of Concept

Interactive debate - Analysing speech in Video, is a system that generates a video transcription, identifies the different speakers, provides a summary of the main topics and keywords as well as the emotional charge of the speaker. Winner of the Times ‘Build The News Hackathon’ (Passarelli Pietro 2015d).

This is further discussed in the Investigation section.

Difference

Video quote extractor is different from previously mentioned projects in that it;

- leverages speech to text API to automate transcriptions
- has an easy-to-use interface to make selections of quotes, and cut the video automatically
- provides embed code for embedding
- gives more control to the user in selecting a quote through text selection

- makes the most of HTML5 video tag using videojs and doesn't need to use popcorn Javascript, flash or JPlayer
- is an MVC web application with distinction between back end, and front end and has the database to preserve state.

2.3. Research done

Research was also done on the following topics:

- Speech-to-text solutions
  - –YouTube captioning
  - –SpokenData API
  - –Dragon Naturally speaking
  - –CMU Sphinx (offline Speech To Text)
- Hyper-transcripts
  - –analyse algorithm of hyper-transcript converter and refactoring in Ruby, *see appendix*
  - –video Javascript, clickable, searchable transcriptions components
- knowledge domain of the application.

2.4. Tools and Software selected for use in the project

2.4.1. Intro to software

An overview of Rails and a description of how Rails uses AJAX to handle client side interactivity can be found in the appendix.

Ruby on Rails vs Laravel PHP

Both Ruby on Rails and Laravel PHP are well supported with Jet Brains IDE, which was used in this project to increase productivity.

After an initial study of the Laravel PHP framework, to minimise the learning curve of new language/framework for the server side back end and to concentrate on the investigation and the implementation of the solution, it was decided to opt for Rails. Because of previous experience in Rails, it was a good chance to consolidate a number of components (including an sbv/srt parser) that could be modified from a previous project, and other Ruby libraries such as a Timecode one (Julik Tarkhanov 2009).

Front End Client Side Interactivity

JQuery vs Angular vs React

The framework Angular and React were studied, but the conclusion was that it was a bit over-complex for this project as well as having too steep a learning curve. Instead, a better option was to use JQuery, to ensure cross browser compatibility and to try and keep code well organised within the Rails framework. First of all, Javascript had to be learned, to better understand the workings of JQuery, especially for manipulating elements on the DOM.

Mobile First Responsive Bootstrap

Despite there being no requirement of the use of an application that works on a mobile or tablet, it was decided to keep the option open, using Bootstrap's responsive, mobile first framework (Twitter 2012) and to use Bootswatch (Thomas Park 2014) for the theme. This makes it easy to change between Bootswatch themes with minimal effort, but also if in the future there is the need to create a bespoke theme, the Bootswatch classes and IDs provide an interface where it is easy to modify the style.

Hyper-Transcript - Video

The hyper audio converter (Mark Boas 2013b) and Boas Al Jazeera interactive debate projects (Mark Boas 2012), discussed in the previous chapter, were studied closely to understand the implementation of the hyper-transcript, especially looking at the interactivity and the two way sync between text and video.

At the BBC Hackathon, discussed in the Investigation section, there was a chance to prototype an alternative way to do hyper-transcripts using Video JS and JQuery instead of popcorn JS and JPlayer used by Boas (Mark Boas 2013a).

Because of the nature of this project, it was decided that it was more logical to parse the srt file in the back end, in the model class of the Rails MVC, which also would make it easier to test it. This is also consistent with the MVC design pattern where views are less code-heavy than models.

It was also decided to store the transcript in the database in a way that would make it easier to retrieve as a hyper-transcript. This is discussed in later sections.

In the appendix, there is an analysis of the javascript hyper-audio converter by Boas and this project's re-factoring implementation in Ruby.

Using the hyper-audio converter JavaScript library/open source project (Mark Boas 2013b), it was decided to convert that into MVC Ruby language as a method in the model to refactor it as a component and to more efficiently enable testing (as this way could be refracted as a Ruby gem to be used in subsequent projects). This is discussed in greater detail in following chapters.

Re-writing hyper audio converter from Javascript to Ruby meant that it could be abstracted and packaged as a gem component to create hyper-transcripts from srt.

Video JS

Video JS allows us to make the most of the HTML5 video tag, instantiating it as a Javascript object (Brightcove Inc 2010). In particular for this project it was useful to be able to play, pause, retrieve and set current time. These functions were at the core of the hyper-transcript interactivity, realised with JQuery. Each word was encapsulated in the HTML element span tag with timecode attribute-associated in seconds which made this possible.

APIs

Implementation of Ruby SDK for Spoken Data

The choice of using spoken data rather then YouTube captioning API was discussed in an earlier section. In order to facilitate a more modular component, a Ruby SDK was made to create an interface for the text-to-speech API that would make it possible to be able to change text to speech component, without undergoing major refactoring of the rest of the application. See appendix for the code implementation.

Google Authenticator

Google Authenticator was used to handle user authentication and integrate with the current system in place at the Times.

2.4.2. Deployment

The application was deployed on the Times Amazon Cluster using Deis. A Heroku-style deployment was chosen amongst the options, for its widely-known convention, for when the project will be open sourced. A bash script was written to automate the deployment process.

2.5. Tools

Ruby Mine IDE, Git, and Github and deployment Heroku style were all used. Balsamiq Mockup 3 was used for drawing and presenting proof of concept prototypes to the users and stakeholders. The use of VM was considered an option, such as Vagrant but to reduce the overhead of learning curves, the decision to not use it was reached. Instead, it was opted to keep logs of how to setup various dependencies.

2.6. Summary

This chapter has provided context and background to the project, as well as explored sources, similar solutions and introduced the software.

3. INVESTIGATION

The first step for creating a tool for journalists in the newsroom required gaining a good understanding of the problem domain.

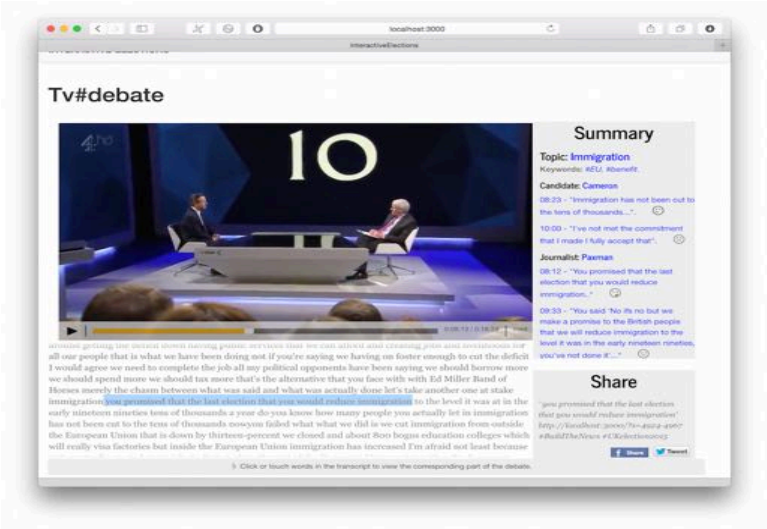
> *A problem domain is the context in which a particular problem exists. For example, the problem domain in which a specific route plan exists is that of maps, route planning, travelling and strategies for moving around. Critically, the problem domain is relatively stable, changing only slowly, while specific problems to be solved are transient and change regularly. (Winder et al. 2006, p354)*

This chapter outlines the most significant part of the investigation into the problem domain.

> *If you are able to capture the problem domain as the core of the design of your program, then the program code is likely to be more stable, more reusable and more easily adaptable to specific problems as they come and go. If you only capture a specific problem as the core of your design then your program code is only good for solving one problem and will, at best, require significant modification to solve a different problem. (Winder et al. 2006, p354)*

3.1. Background: The Times Build the News Hackathon

Prior to the quickQuote project, on 28th - 29th March 2015, there was the Times Build the News Hackathon, of which the author of this dissertation took part and was on the winning team. There, a demo proof of the author's concept for the idea **given a video would provide an insight into its content** was given, by generating transcription, identifying key speakers, main topics, keywords and a summary.



An R&D Approach was used for the development of this application, testing components in isolation and manually passing output between each to verify that they would integrate and build a dummy-interactive demo to demonstrate proof of concept (Madalina Ciobanu 2015).

Drawing form a number of open-source projects and components:

- Open-source project Interactive Aljazeera Obama-Romney debate, mentioned in the previous section (Mark Boas 2012)
- Choosing a suitable case study, ie TV interview for elections.
- Identify components, and manually test in and out before implementing the system. Making API calls and send results manually
- YouTube captioning - YouTube API v3 (Google YouTube 2015b)
- Speaker diarization (and cross referencing timecodes)
- Hyper-audio Convert open source project in JS(subsequently converted into Ruby and moved in model of MVC rails) Boas (Mark Boas 2014) expressed a preference for using HTML as a data-structure. Here, it was flagged that it did not seem to be a sound soft engineering choice. JSON was suggested as an interchange format to preserve the state of transcript.

This led to being taken on for an internship with the Times and Sunday Times digital team and this is where the quickQuote project began.

3.2. Research of the Technology Stack

The internship was to start in July and last until mid-August.

The month of June was spent researching the technology stack and possibilities for the implementation of such systems.

Laravel PHP was initially considered as MVC framework, as it is the one used by the Times. However after careful consideration, Ruby on Rails was considered the best choice, as the author's previous experience with it would make the prototyping a lot faster.

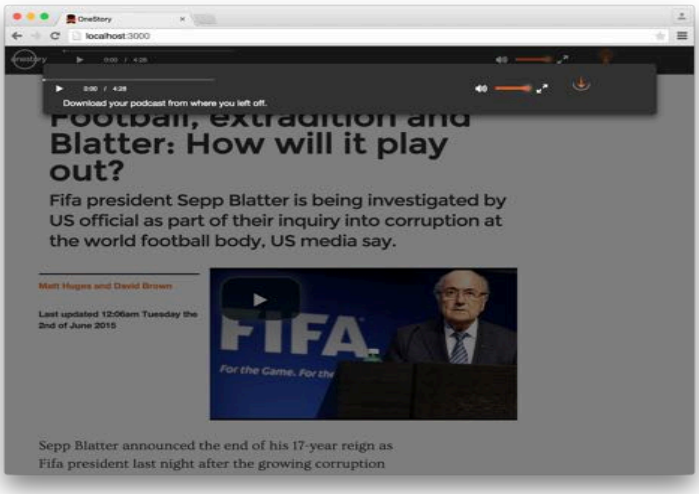
For front end, AngularJS was considered for the same reason, but considering the steep learning curve, JQuery was used instead.

3.3. BBC Hackathon NewsHack

On June 3rd and 4th, the author took part in the BBC NewsHack Hackathon as developer in a team with a journalist and designer. This was won with one of the author's ideas.

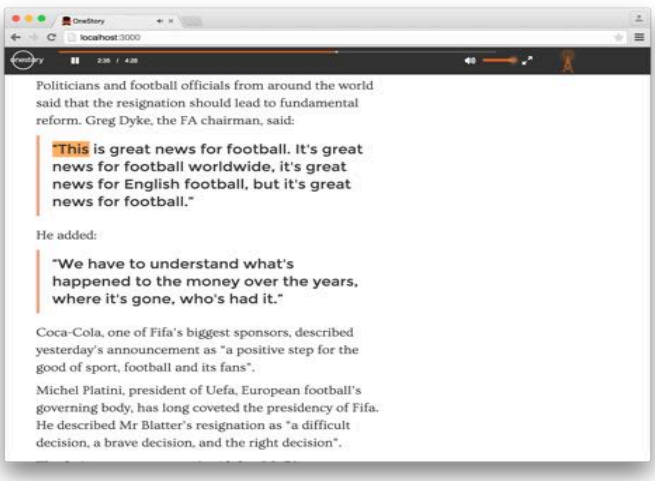
This was a good chance to test an idea for a possible summer project. The system would be creating a story editor CMS that would allow journalists to create a multimedia story once (with audio, video, stills and text) and produce 3 distinct outputs: a news article, a podcast, and a video. The viewer could then switch between these three outputs seamlessly.

Because of time constraints and to narrow down the scope of the project, it was decided to focus on building a proof of concept demo that would allow switching between the article and podcast.



BBC News Hack - Proof of concept Demo

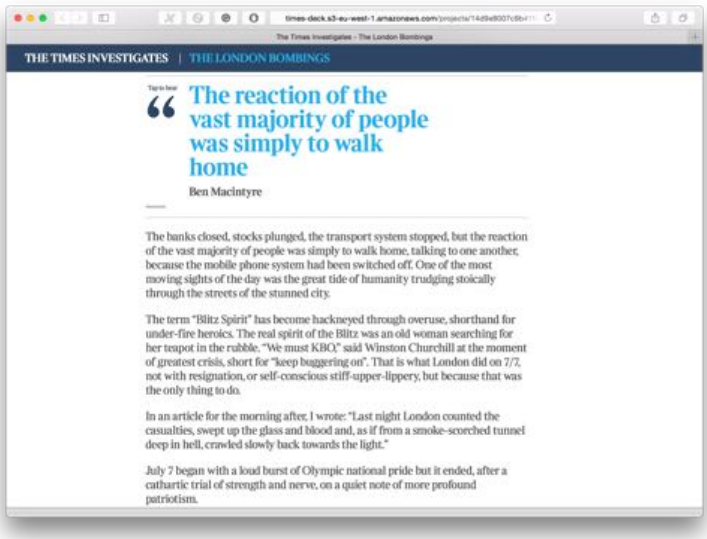
The system also allowed the user to click onto a quote and hear an associated sound byte.



BBC News Hack - Proof of concept Demo - Audio Quote

Despite the success of the project, it was decided that for the summer project it was more useful for the Times to build a tool that would integrate with their existing CMS tool and would have a higher use rate, as these type of multimedia productions are quite labour-intensive at present as well as infrequent.

As a result of the proof of concept, one of the developers at the Times developed a “card component” to embed “audio quotes” in their existing CMS, for a long form article about the 7/7 bombings.



Times Audio Quote Component

However this audio quote component takes the input of the text of the quote and the corresponding video file. It does not help with finding the quote and trimming the file.

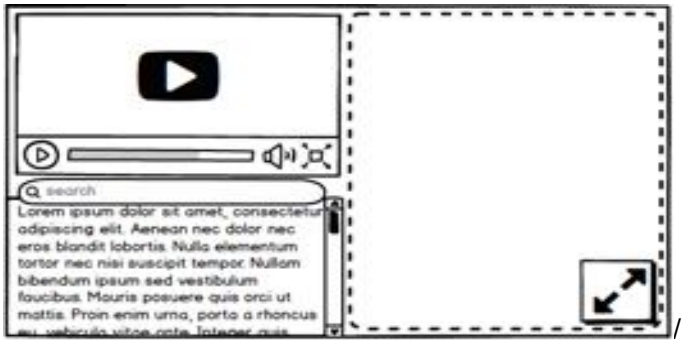
3.4. Video Analyser Prototype

To investigate the use for a system that would provide some sort of insight into videos, I started researching and prototyping a “video analyser” web application.

3.4.1. Baseline Project

The baseline for this project was a web application where, given a video by the user and a subtitle file transcription of that video, would display the video with a clickable, searchable version of the transcriptions, and some insight into the video, such as the most used keywords and their significance.

YouTube captioning was used to generate an srt subtitle file of the transcription.



Baseline project

The application is composed of an upload, a dashboard and a published view.

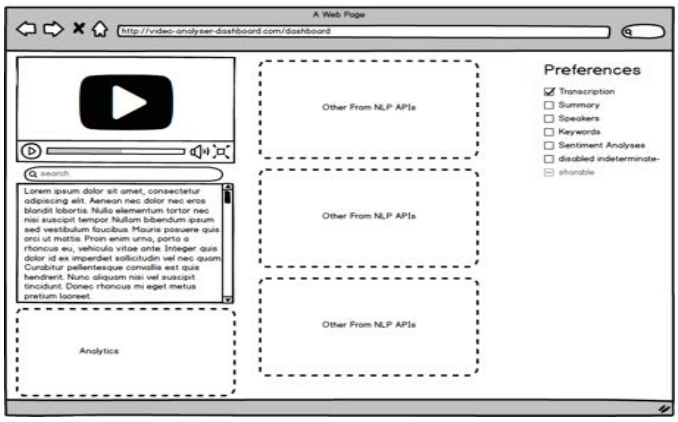
The dashboard is where the user can search and interact with the transcriptions through a hyper-transcript synced with the video. From the dashboard the user can then publish.

At this stage the published view consists of a link that displays the video and the hyper-transcript.

3.4.2. NLP Analyses

Because the nature of the R&D investigation was to explore what was feasible, and whether there was any real use for using NLP to gain an insight into the video, this phase was prioritised over the API to automatically generate transcriptions. The method of sampling out what was “manually generated” from the speech to text API was used as the starting point of this NLP analyses component (AlchemyAPI Inc 2015b).

As this involved researching NLP API solutions, the choice narrowed down to Alchemy API for identifying keywords, key concept, entities and sentiment analyses and “smmry” for summarization (Elmaani LLC 2009).

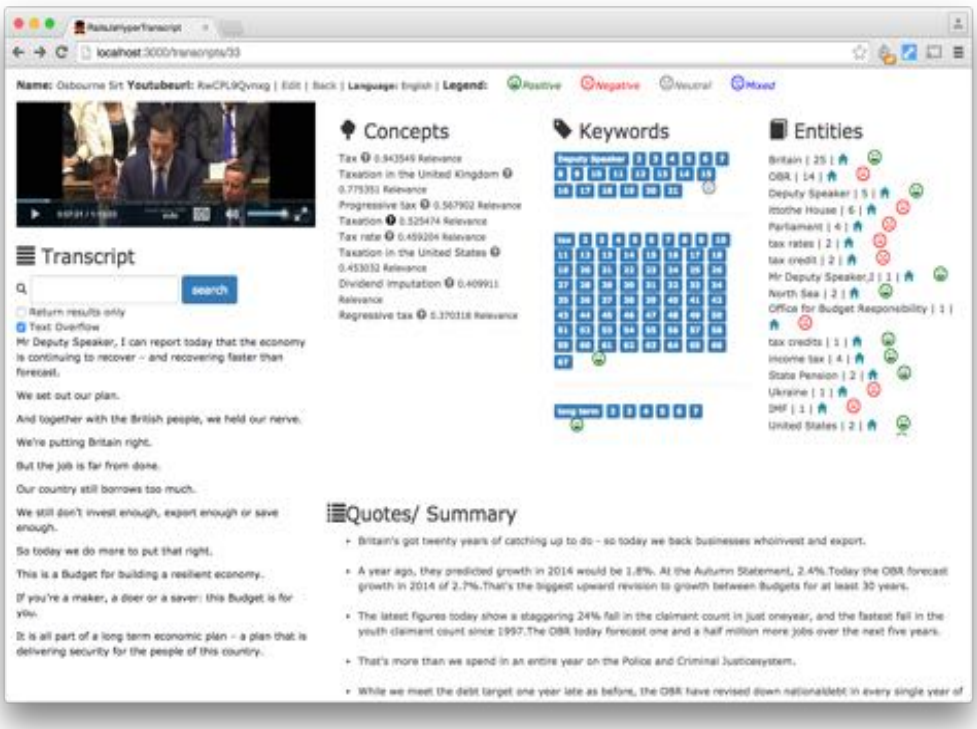


Video Analyser - NLP - Dashboard

3.4.3. NLP Analyses Use Case

To try to narrow down what could be useful information and/or interpolation that could be done with the data returned by the NLP API calls, it was decided to try to apply a use case.

The George Osborne 2015 parliamentary Budget speech (HM Treasury and The Rt Hon George Osborne MP 2015).



Video Analyser - NLP - Use Case

This was done working with the Data Team at the Times. The system identified keywords, concepts, Entities, provided sentiment analyses, and with summary API provided suggestions on most significant quotes from the speech.

However the specific use case demonstrates that without further interpolation and analyses of the NLP results, the insight provided by the system was not particularly useful to the journalists, and the user's feedback did not provide any clues on how that could be improved.

This is also when it became apparent that summarisation API algorithms need transcription to work. In the case study of the George Osborne budget speech, the

transcription had punctuation, because it was a written speech and the original could be found on the .gov website (HM Treasury and The Rt Hon George Osborne MP 2015). This was then added to YouTube captioning, that lined it up with the video. However the issue with this was that anything said by the speaker outside of what was written for the speech was not included in the transcription, and unscripted remarks were left out.

The modeling of the transcription in the database to circumvent this issue is discussed in the Requirements Chapter.

3.5. quickQuote - A Video Quote Extractor

Having explored the NLP analyses option, a revisit to the results of the Hackathon was useful. Here, the implementation of the audio quote component for the Times CMS was considered, looking into carrying out something equivalent for video. The audio component assumes that the user already has a trimmed audio clip and text quote. Whilst the video quote improvement (aside from supporting video) would also have been helpful in identifying the quote.

3.5.1. The Hypothesis

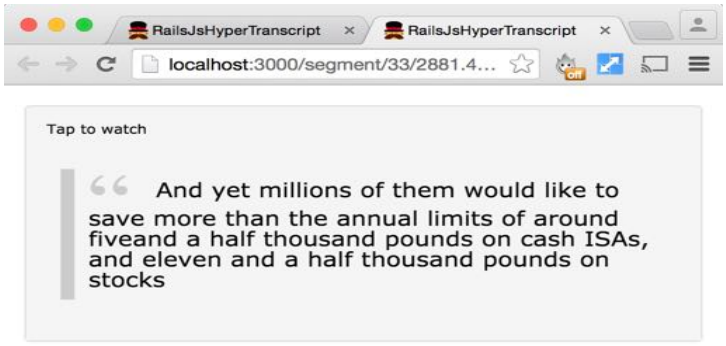
After discussions with the journalist who wrote the article that incorporated the audio quote component, the hypothesis was that a system to extract video quotes would have saved him a considerable amount of time (estimated to be more then five hours) when selecting a video clip for that article.

3.5.2. The Proof of Concept Prototype

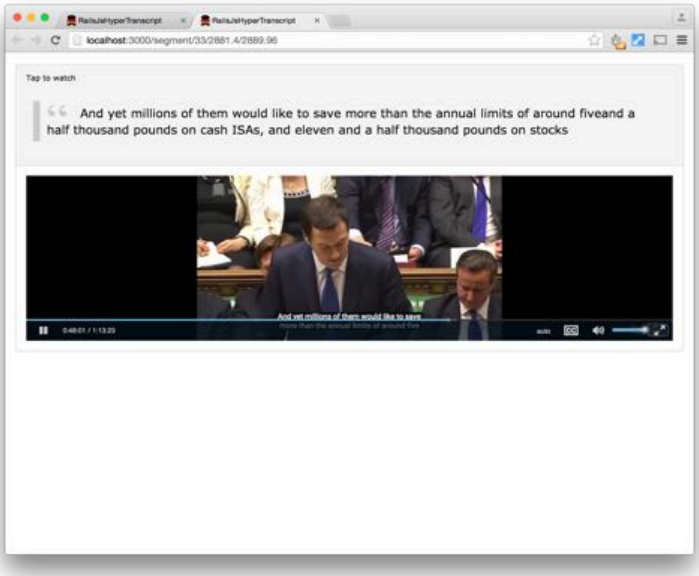
Components from previous project such as the baseline project with hyper-transcript were used.

Using the hyper-transcript from the previous video analyser system *NLP Analyses Use Case* I added the feature to demonstrate the functionality.

The user selects text and a link is then generated and added to the page. When clicked, it takes the user to a new page with the quote and a drop-down of the video.



proof of concept quote

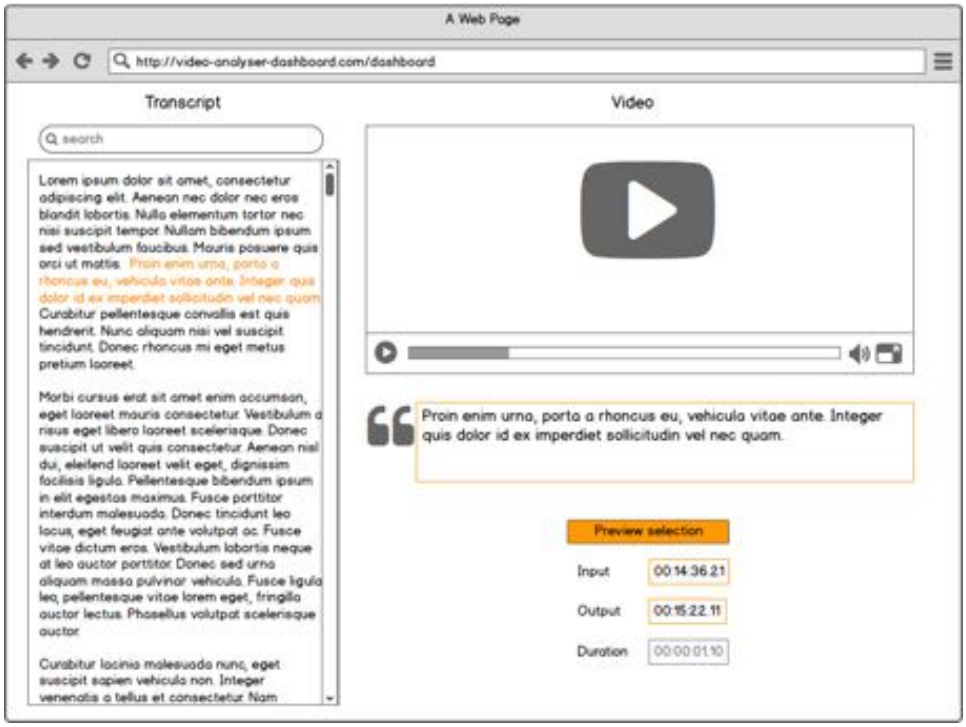


proof of concept dropdown

3.5.3. User Stake Holder Feedback

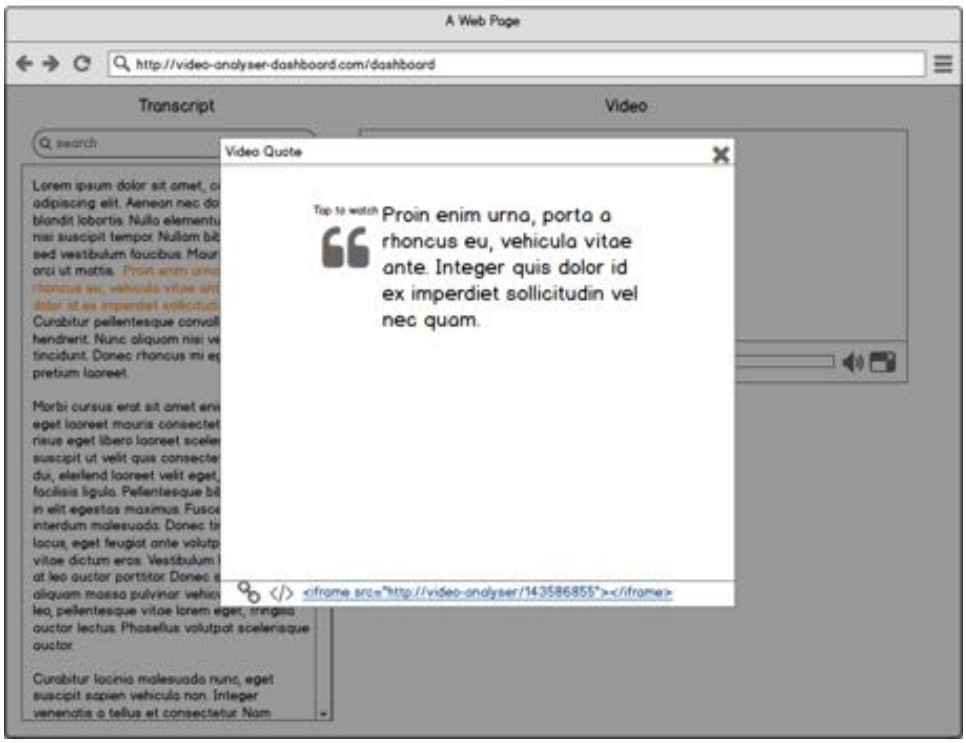
This was presented and a discussion to agree on a minimum set of requirements for the project arose.

Presenting a proof of concept where, when the user selects a quote, it populates a text area of where the selection can be edited.



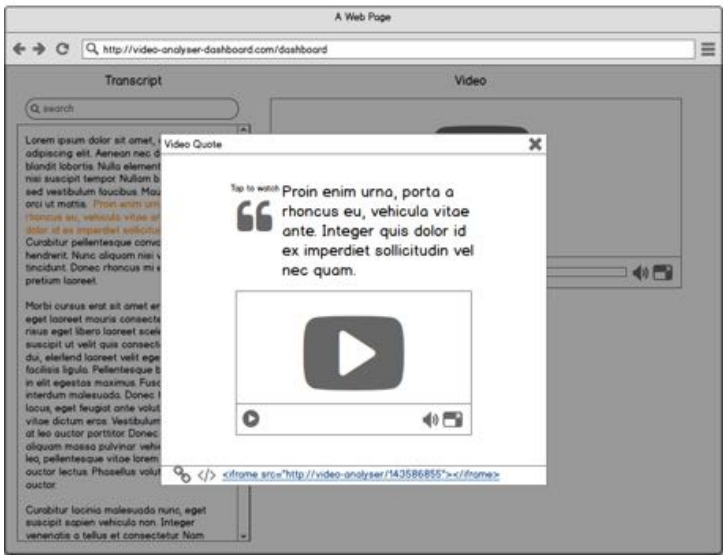
Select a quote

The quote can also be previewed.



Preview a quote

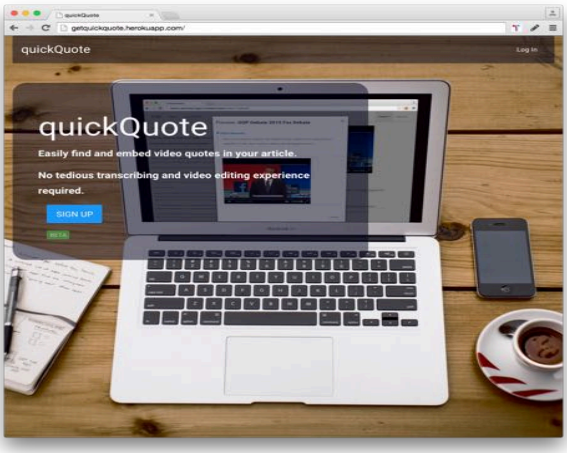
When clicking on the quote, the video drops down.



Preview a quote

3.5.4. Refactoring and Implementation

This was followed by refactoring, implementation and deployment, which is discussed in more detail in the following chapters.



Home Page

3.6. Summary

This chapter has given an overview of the iterative nature of the investigation that was carried out to better understand the problem domain, frame hypothesis and build prototypes.

4. REQUIREMENTS AND ANALYSIS

The initial focus of the investigation was on capturing the problem domain of working with video and transcription in the newsroom to give a way into adding more multimedia elements to news articles, aiming to reduce the cost traditionally associated with working on multimedia.

> *Object-oriented analysis and design is based on a process of first identifying the abstractions needed by a program and modelling them as classes. (Winder et al. 2006)*

This was the approach taken during the investigation to identify a comprehensive solution. In this section an overview will be given of the requirements gathered in the last stage when the final solution was identified, and what the program is expected to achieve will be considered.

4.1. Problem Statement

The problem statement was introduced at the beginning of the introduction, following the investigation section. What follows revisits of the problem statement.

The aim is to develop a web application that allows the user to upload a video, select a quote (from an automatic generated transcription of the video) and obtain a HTML code to embed a video quote into a news article. This should have an intuitive user interface.

4.2. List of requirements

> *The first goal of requirement gathering is to generate a list of what the program is expected to do. Following an agile approach, each requirement is called a 'story', therefore, the requirements are a collection of stories (Winder et al. 2006, p359).*
>
> *Each story is a short written description of some aspect or behaviour the program should have. (Winder et al. 2006, p359)*

Because of the iterative nature of the investigation, by the time the requirements of the final solution were identified, the user stories were already elaborated into scenarios and geared towards use cases.

The user requirements will now be considered. The user should be able to:

- Authenticate into the application, using their Google login
- Upload a video
- Automatically generate a transcription of the uploaded video
- Search the transcriptions for occurrences of specific words

- Click on the transcription and have the play of the video move to the corresponding point in the video
- When playing the video, easily identify the corresponding text in the transcription
- Select text from the transcription to identify a segment of video quote to export
- Preview their selection
- Cut the selected video segment, by exporting the quote
- Export a video quote, as a HTML embed code.

4.3. User Journeys

Use case 1, **Signing Up**: The user registers and signs into the application through their Google account.

Use case 2, Uploading a Video

Pre condition: the user has logged in. The user navigates to the upload page, and uploads a video into the application.

Use case 3, **Analysing** a Transcription

Pre condition: the user has logged in to the application. The user navigates to the list of videos, clicks on one, searches the transcription, clicks on hyper-transcript, plays and pauses the video.

Use Case 4, Selecting and Previewing a Quote Selection

Pre condition: the user has logged into the application, the user has navigated to a specific video.

The user selects the text of the hyper-transcript, it is then given a preview of the text selected and the timecode in, out and the duration.

Clicking on preview selection takes the user to a preview modal, where the text of the quote appears styled as a blockquote tag element and when clicking on it, the video drops down, previewing what it would look like when exported.

The user can then play the video, and only the video corresponding to the text selection will play.

Use Case 5, **Exporting** a Video Quote Selection

Pre condition: the user has logged in to the application, the user has navigated to a specific, video the user has selected a quote.

The user clicks on the export button, and receives feedback that the application is processing the request.

When the processing is complete, a modal returns the HTML code with the code to embed the video quote in a news article, which can then be copied and pasted elsewhere.

The video tag in the HTML code contains a trimmed version of the video that matches the selection of the text.

4.4. Result of Requirements Analyses

To build a model of the program, Brain-storming and common sense was used, combined with identifying the requirements and use cases (Winder et al. 2006).

The overall modeling of the application was fairly straightforward given the gathering of the requirements. If a user has several videos, those videos are going to have transcriptions and each video will have several quotes.

However it was through the iterations of the investigation that shortcomings of the initial modeling of the program emerged. The biggest issue was around modeling the transcriptions which will be considered in more detail in the next section.

4.4.1. Modeling Transcriptions

One of the most challenging parts was deciding how to model the transcriptions in the database, as getting this right would save a considerable amount of refactoring, should the application change in the future.

There were two major areas to decide upon, one was whether to make the transcription word or line accurate and the other was around the lack of punctuation associated with the transcriptions, which would have implications when manipulating elements in the DOM.

For instance when implementing search of hyper-transcript in JQuery, if the user wishes to highlight and match words in the transcript, it poses no issues but if the user wishes to narrow down results and show only sentences that have the specific word being searched for, the user would need to calculate the boundaries of each sentence, requiring extra parsing on the client side.

Raising the question of how the user can identify sentences: neither option has a representation of sentences, as YouTube generated lines from the subtitles tend to be arbitrary separations of the text. At the moment adding </p> when a word or line has a full stop in it, could work, but automatic generated transcriptions do not have punctuation.

Option One - Model subtitle file

The first option considered was to model over the subtitle file srt file.

This is a sample srt file:

```
1
00:00:10,120 --> 00:00:11,070
Hello world
test

2
00:00:12,020 --> 00:00:12,220
Another sentence
...
```

and this would be the corresponding Lines table

id	tc_in	tc_out	text	transcript_id
1	“00:00:10,120”	“00:00:11,070”	Hello world test	2
2	“00:00:12,020”	“00:00:12,220”	Another sentence	2
…	…	…	…	…

Where ID is not the line number of the ‘srt’ file, but rather a unique identifier for the line, consistent with the Rails convention.

while tc_in and tc_out are timecodes from the .srt file.

Here, storing the timecode as a string, is the method used, utilising the ruby timecode gem to parse the timecode, or do timecode operations if needed.

Alternatively the Timecode gem could be set as a custom type in the Rails active record ORM.

The text is self explanatory, although it is worth noting that in the srt and sbv at times the text can be split across two lines.

The transcript_id is the foreign of the transcript the lines belong to.

The issue with the line accurate option is that, if made with Youtube, the lines are arbitrarily defined by Youtube captioning, and do not necessarily correspond to sentence breaks.

Option 2 - Hyper-transcript

The second option was to model the transcription over the hyper-transcript that will be used to represent it in the view.

Using the open-source hyper audio converter from github (Mark Boas 2013b), this converts the srt file from line accurate transcriptions to word accurate ones. Re-factoring this code that parses an srt file into a hyper-transcript in Ruby, enables the parsing in the model and makes it easier to test, rather than doing the parsing in the front end.

The result would be that the “hyper-transcript” would appear like this:

Hello world test Another sentence

Where m="10120" is the corresponding time for that word in the video in seconds.

The corresponding representation in the database would be:

id	tc	Word	transcript_id
1	10.48	Hello	2
2	11.38	world	2
3	12.08	test	2
4	13.08	Another	2
5	12.28	sentence	2
…	…	…	…

The main difference is that this time it is word accurate and we do not have a tc_out but instead just a timecode in as tc.

Aside from these details, the main issue to resolve was whether to model the transcript as word accurate or line accurate.

Lack of Punctuation in Transcription

One of the issues around transcriptions that were affecting the modeling included the fact that the automatically generated transcriptions do not have punctuation but could have punctuation if the user were to manually edit them, for instance through the spoken data transcript editor interface.

The lack of transcriptions punctuation is an issue for two reasons. On one hand without punctuation, the hyper-transcript would be displayed as one continuos block of text without paragraph breaks as any break point (or opening and closing of a P-tag) would be fairly arbitrary at that point. The other issue would be to create a

summarisation feature at some point in the future, which would enable automatically suggested quotes, which is something that was tested during the investigation.

Final Modeling of the Transcriptions and Database

For the final modeling of the transcriptions it was decided to store the lines information of the srt file automatically generated by the speech to text APIs but use the words as the atomic element of the transcription.

If the user were to edit and add punctuation to the transcription at a later stage, it would be easy to write a method to update the lines table.

As mentioned, Rails uses active record ORM to map the tables of the database, to the classes of the model of the MVC. This is discussed in further detail in next chapter.

4.5. Summary

This chapter gave an overview of the project requirements, and discussed the most interesting problems in this area.

5. DESIGN AND IMPLEMENTATION

5.1. Rails Model Classes

As mentioned in the previous chapter, Rails uses ORM Active Record, to map the model classes to a database table. Therefore the data modeling corresponds to the class modeling, in an object oriented fashion.

The relationship between classes will now be considered in greater detail, also exploring the attributes of each.

A user has many videos.

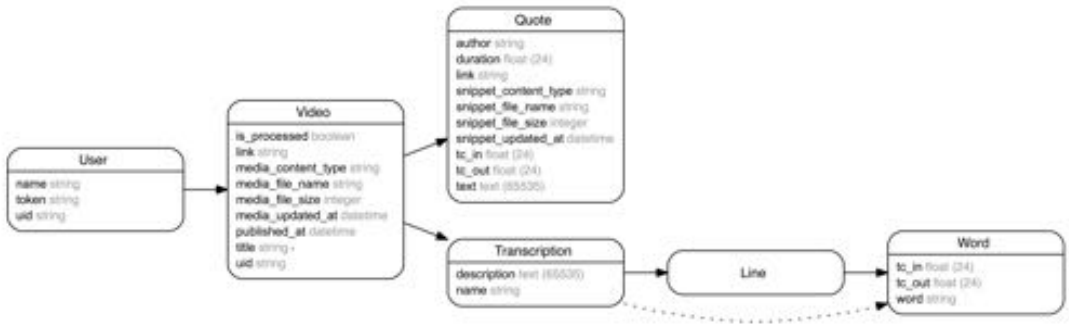
A video belongs to a user, and has many transcriptions.

A transcription belongs to a video, has many lines, and has many words through lines.

A line belongs to a transcription and has many words

A word belongs to a line and has one transcription through a line.

This diagram gives an overview of the classes in the application and their attributes.



Implementaiton diagram

5.1.1. User Class

Google Authentication was used to handle authentication, the sole purpose of the user class is to keep track of a session while the user is logged in.

5.1.2. Video Class

Video File

The paperclip gem (thoughtbot 2008) was used to handle video file assets. This is very convenient, as it simplifies operations such as adding or deleting a file associated with a video instance. For example you can delete the video file associated with the video instance as follows:

```
@video.media = nil
```

As used in the video controller under the destroy method, so that when the user deletes a video object, it deletes the associated file as well.

Paperclip automatically handles other attributes associated with the file, such as content type, file name, file size and the date of upload.

is_processed

The Boolean attribute is_processed was used to handle the speech to text API (spoken data) delay in processing transcription.

In the database this is set as false, represented as zero, when the user first uploads it. The video is then sent to SpokenData. Then when the user views the index videos page/show page, the controller (when serving the page) checks the is_processed to evaluate if it is necessary to make the API call.

UID

The UID attribute is produced by SpokenData to retrieve transcriptions in subsequent API calls.

Link

The link attribute stores a link to the url of the amazon s3 bucket where the video files are stored.

5.1.3. Transcription Class

This only has description and name, because the main function is to group the lines.

5.1.4. Line Class

Lines have no attributes, but only a transcription ID foreign key, as their sole purpose is to group words, which is a solution to the lack of punctuation as discussed in previous chapters.

5.1.5. Word Class

Line_ID

Words have a line foreign key line_id to group them into lines as previously discussed.

TC_IN and TC_OUT

The timecode attributes TC_IN and TC_OUT are stored as float and they represent seconds.

These are generated using an algorithm to make a word accurate hyper-transcript data structure, discussed in the following sections.

When parsing the srt file the time is converted from this format hh:mm:ss,ms (ie "00:00:10,120") to seconds using the Timecode gem (Julik Tarkhanov 2009) before saving.

Initially, the method of storing as string in the notation used by the srt file was thought of, followed by a choice of seconds because both FFMPEG and Videojs work in seconds, and could use Timecode Gem to convert to other timecode formats should a need for it arise.

Word

The word attribute itself stored as a string.

5.1.6. Quote Class

Video File

Similarly to the video class, the quote class uses the paperclip gem to handle the video file. The attribute for the video file is called snippet.

During the project, a snippet of a quote was defined has a video segment being cut from a video using FFMPEG.

Link

Similarly to the video table, link also stores a link to Amazon S3 URL for the trimmed video segment.

TC_In and TC_Out

Timecode in and timecode out and the duration stored as floats, represent seconds for the video segment of the quote and are relative to the parent video.

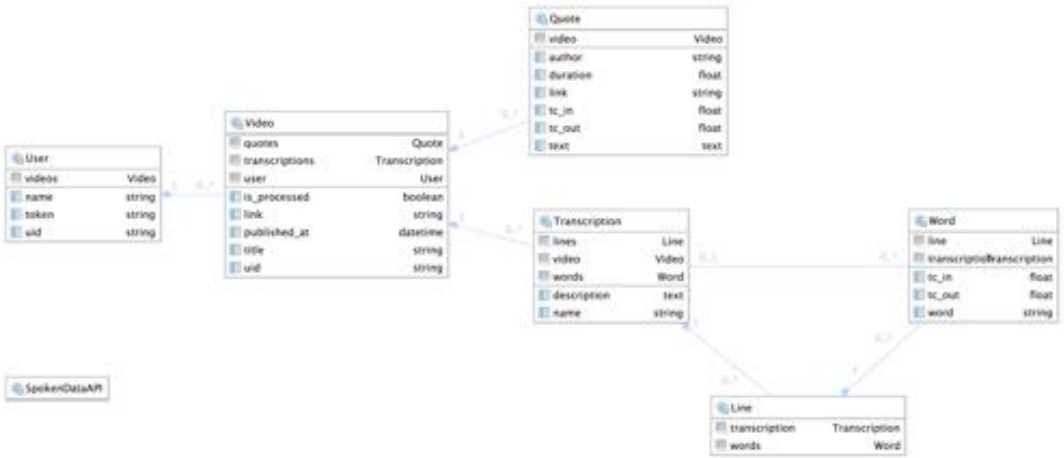
These are computed in the front end using JQuery initially when then selection is made, and then sent through to be saved in the database when the selection is exported using AJAX. This is discussed further in later sections.

Author, and Text

Last but not least there are attributes to string and text attributes to save the details of the quote such as author and text.

5.2. Database

As mentioned, Rails uses ORM, object relational manager, so the model classes discussed above are automatically mapped to a database table through migrations.



Implementation diagram

5.3. Components

The following text will now consider the implementation of some of the most relevant and interesting components.

5.3.1. Third Party Components

Firstly, the third party components used should be considered. The main ones were Video JS, Bootstrap and Bootswatch.

Bootstrap

Bootstrap was mostly used to handle responsiveness of the website, but also from the bootstrap library, the modal and drop down JS components were used, for preview and export of video quote, and video quote drop down.

Bootswatch

Bootswatch was used for theme, both to provide a quick intuitive design to the front end but also to provide a set of interface CSS classes and IDs if in the future bespoke theme would need to be made, to make it easier to swap and plug.

VideoJS

VideoJS was used to make the most of HTML 5 video tag and treat video as a JS object. Combined with the functionality of the HTML5 video tag was possible to use methods such as GetCurrent time and SetCurrent time that were crucial not only to make the hyper-transcript work effectively but also to make it possible, combined with JQuery, to select, preview and export a video quote.

5.3.2. SpokenData API SDK

The SpokenData API provides a set of RESTful URLs as an interface to access their service. This was abstracted as an SDK component and an interface was written that could allow for the future swapping out of YouTube captioning API, other third party API, pr using CMUSphinx for offline speech to text, provided that the appropriate interface is implemented.

The SDK interface was inspired by Alchemy API Ruby SDK (AlchemyAPI Inc 2015a) and where an “API class” contains methods encapsulating Restful API calls, as well as aiding the API Key authentication.

See the appendix to view the code.

Additionally, SpokenData supports sending a video through a post request specifying the URL of the video, which makes it easier to send a video that might be stored on an Amazon S3 Bucket.

5.3.3. Hyper-transcript

To make a Hyper-transcript after considering various implementations mentioned in the context chapter, in the context of the Times Build the News Hackathon the Mark Boas implementation of Hyper-transcript was used in the Obama Romney Al Jazeera debate (Mark Boas 2012) which used popcorn JS and WJPlayer.

It was during the BBC NewsHack Hackathon that the decision to attempt to make a simplified hyper-transcript using video JS and JQuery was made.

This proved to be a simple solution with more straightforward code.

So what is a Hyper-transcript?

First and foremost it could be argued that it is easier to think of it as a data structure. It should be a data structure that stores the text transcription of a video and has timecode information to map the text to the corresponding point in the video. The granularity of this text-video correspondence should be word accurate.

On the front end it should provide functionality such as searching for specific words, live colour highlight to keep correspondence between text and video, and scroll sync with the video.

Choosing an Appropriate Data Structure

Boas (Mark Boas 2014) argues for the choice of using HTML as a data structure to preserve the state of the transcription.

However with the increasingly flexible use of JSON as a data structure and frameworks such as Angular and React, it could be argued that if the hyper-transcript component used JSON representation for the data structure and had a method to parse srt files and a method to render it as HTML, that would be a far better and more future-proof implementation on the server side, which would also make it easier to do things such as saving it onto a relational database and integrate with front end frameworks such as Angular or React, should there be a need for it.

Then on the front end client side it would be represented as HTML, using JS (or some other JS-based library or framework) for client side interactivity.

Parsing Srt File

To create a hyper-transcript from srt file input, retrieved from speech to text API. It was then decided to refactor the Boas hyper-audio converter JS opensource project (Mark Boas 2013a), as mentioned in the context chapter.

One reason for refactoring was that this could be extrapolated as a standalone component - Ruby Gem. The other was that to be consistent with the Ruby on Rails convention over configuration idea, it was best to move this business logic into the model, rather than having this processing done with client side JS.

Another reason was that this way, the method could take into account the content of an srt file as a string and return a hyper-transcript data structure making it more of a self sufficient component.

A full breakdown of the analysis of the hyper-audio converter by Boas can be found in the appendix.

Saving the Hyper-Transcript Data Structure Into DB

As mentioned in the previous section, the method called convert_srt_to_word_accurate_hypertranscript which is in the video model, takes a srt file as a string and outputs hyper-transcript hash data structure (which is equivalent to a JSON)

Once the srt had been turned into a hyper-transcript data structure, then the result was saved into the database using Rails Active Record ORM that, as discussed in previous sections, was modeled around the structure of the hyper-transcript.

The parse_srt_and_save_into_db method takes in the video UID which is stored in the database and is originally returned by SpokenData on successful submission of a video.

It then initialises a SpokenData API object and uses the retrieve_subtitles_srt implemented in the Ruby SpokenData SDK to check if a srt transcription is available for the SpokenData API. If it is, it is then converted from a string containing the content of the srt file to a hyper-transcript Ruby hash/JSON data structure using the convert_srt_to_word_accurate_hypertranscript.

This hyper-transcript data structure is then saved into the database using the save_to_db method, which will now be considered in further detail.

It creates a new transcription instance, iterated over the lines in the hyper-transcript data structure and creates words associated with lines using Rails ORM syntax @line.words.create.

Serving Hyper-Transcript Data Structure to Views

To render the hyper-transcript to the views in Rails, it was decided to follow the Boas (Mark Boas 2013b) way of surrounding the text of each word into a span tag and to add other information as attributes.

By rendering HTML using Ruby ERB templating language in the Rails view, app/views/videos/show.erb.html

```
 <% @video.transcriptions.last.lines.each do |line| %>
      <p class="line">
     <% line.words.each do |word| %>
          <span class="word not-read" tcIn="<%= word.tc_in %>" tcOut="<%= word.t
c_out %>" line="<%= word.line.id %>" transcription="<%= word.transcription.id %>">
<%= word.word %></span>
        <% end %>
```

```
    </p>
  <% end %>
```

This way each line is inside a paragraph tag and for each line all the words are enclosed into a span tag with the corresponding attributes.

@video.transcriptions.last.lines Rails ORM syntax was used, to retrieve the lines from the last transcription of the current video.

This is because a video is allowed to have multiple transcriptions. In the current implementation it is only going to have one, but just in case this changes and the transcription is updated, the option of retrieving the latest version is always wanted, as it will be the most up-to-date.

This could also allow a swap out of transcription to show something such as a translation in a different language.

Ruby ERB renders as HTML span tag on the client side.

```
<span class="word not-read" tcin="482.708" tcout="483.749" line="237" transcription
="6">web</span>
<span class="word not-read" tcin="483.055" tcout="484.443" line="237" transcription
="6">site</span>
```

The interactivity to make the span tag clickable is then added using JQuery.

A more step by step explanation of the code can be found in the in line commenting below.

```
//make text spans clickable
  clickableSpans();

//Function to make the span tags clickable, the user clicks the text and is then taken t
o the corresponding part in the video.

  function clickableSpans() {
      //grabs the video element with `video_player` as id. and does play pause on it to
get current time to zero.
      videojs('video_player').play();
      videojs('video_player').pause();
      $("span").click(function(){
          //get the `tcin` attribute of clicked span tag, which is the corresponding time for
that word in seconds
          var time = $(this).attr('tcin');
          //sets current time of player
          videojs('video_player').currentTime(time);
          //sets play to current player
          videojs('video_player').play();
```

```
    }); //end span JQuery $("span").click
  }// end of clickable Spans function.
```

Colour Highlight

To make the correspondence between the video and text of the hyper-transript clear, it was useful for the text to change colour while the video was playing and show the current word.

This was done by iterating through all of the words, and comparing the current time of the video player with the timecode in time attribute of each word. If the words 'time attribute' is less than the currentTimeInSeconds the CSS Class is changed from not-read to read, otherwise the opposite, which respectively changes the colour of the text from grey to black.

Scroll Sync

It was also useful to make the correspondence between the video and text of the hyper-transcript clear, to have auto scroll sync of the hyper-transcript to the current part of the text while the video was playing.

The player on time update function was considered by noting that the implementation of colour highlight is triggered constantly while the video is playing. Within that scope the scrollText function was also called.

This works by finding the span tag of the current word by comparing the current time against the time attribute of the word.

For accuracy and to generate a better match, decimal places have been removed.

The function .position() is used to to obtain the position attribute, while .scrollTop() can take argument of where to scroll it to, while 'without argument' returns the user to the current position.

Once there is a match, we are using position().top and adding that as an off-set to the current position of the transcription using .scrollTop(). This moves the hyper-transcript while the video is playing.

Word Search

JQuery was used to be able to search for a specific word, highlighting whole lines in yellow, and making the word within it bold.

Keypress is used to detect when the user types into the input field for the search.

Keyup is used to reset the search as the user deletes characters to enter another search.

Val is used to obtain the character values in the search-criteria input box.

var txt = $('#search-criteria').val();

and contains is used to identify a match, both at line and word level.

.line:contains('"'+txt+'"')

In the identified matches, CSS properties are changed to a yellow background for the lines and to bold font weight for the words to highlight a match.

5.3.4. Selectable Quotes

Before considering how JQuery has been used to select quotes, the HTML elements used to preview the selection should be considered.

There is a text area for the text of the quote and four form input fields, respectively for the author of the quote, the time code in, the time code out and the duration of the selection.

Let us now consider how JQuery method works to get a mouse selection. It goes and uses window selection method with different options for cross browser compatibility and creates a range.

First part of the code, uses mouse up to detect when a selection is made.

Then an if else is used to create a range in different ways that takes account of browser differences.

The code below adds the selected text to the text area for the user to be able to edit it.

```
quote = selection.toString();
$("#textQuote").val(quote);
```

A Focus node and an Anchor node are used to obtain the extremity of the selection. They are respectively the first and last element of the selection.

A range object is then created, and a number of base cases are being considered, such as whether the selection starts from a paragraph tag for a line or a span tag for a word. The closest word element is grabbed and associated with the selection_out and selection_in var.

The two values are compared, because if the selection is done from bottom to top, then timecode in and time code out would be inverted resulting in a negative duration and causing error in the subsequent exporting of the video segment.

The input fields for timecode are then populated.

5.3.5. Preview Selection

To preview the selection of a quote, the Bootstrap model was used to preview selection, grab text from text area, create a video instance with Video JS, and set play and stop using TC in, TC out and the current time method.

5.3.6. Export Selection

This component is composed of another component to cut video and another to generate the HTML code. These could be examined individually but because of the flow of the data it was more logical to consider it as a whole, discussing it sequentially.

The first part using AJAX, when clicking on an export button, grabs from HTML elements that hold text of the quote selected in the text area and other details in various form input fields such as and timecode in and out, duration and the author's name.

This is then string interpolated into a JSON and sent to the routes at the url path save_quote as a post request with JSON data associated to it.

```
$("#export").click(function(){
// read select quote input fields of form, and makes ajax request to save quotes
// ajax to save quotes
   var $btn = $(this).button('loading');

   $TCIN =  $("#input").val();
   $TCOUT= $("#output").val();;
   $duration =$("#duration").val() ;
   $author = $("#author").val();
   $textQuote =$("#textQuote").val();
   $dataToSend = {video_id: <%= @video.id %> , tc_in: $TCIN , tc_out: $TCOUT, du
ration: $duration, text: $textQuote, author: $author} ;

   var currentUrl =  window.location.origin;

    $.ajax({
       type: 'POST',
       url: currentUrl +'/save_quote/',
       data: { "quoteDetails" : JSON.stringify($dataToSend)},
       dataType: 'json',
       async: true
    }).done(function(data){
   ...
```

in rails routes config/routes.rb, the http post request for save_quote is forwarded to the controller quotes and the method save_quote_with_video_snippet invoked.

```
post 'save_quote' =>'quotes#save_quote_with_video_snippet'
```

Where the details of the quote are saved. See appendix for a breakdown of the save_quote_with_video_snippet controller method.

FFMPEG Trimming Video

Within that method, while processing, a system call request is made using backtick `. A system call is equivalent to having the possibility of running a terminal command line command within the Ruby language. In this case we are making a call to FFMPEG, which is a library written in C and installed on the deployment server through a build pack, to trim the video.

String interpolation to cut video with FFMPEG providing a url of the original video on Amazon S3 is used. In FFMPEG -i stands for input, -ss specifies the trimming and takes the start time, timeocde in, -t specifies the duration for the cut and -vcodec libx264 is used to specify the H264 MP4 HTML5 compliant video codec. Some more codecs and preferences are also used and finally the path of where to save the file to.

```
`ffmpeg -i 'http://#{ENV['AMAZON_S3_BUCKET']}.s3.amazonaws.com/videos/media/
#{@video.id}/#{@video.media_file_name}' -ss #{@quote.tc_in} -t #{@quote.duration}
 -vcodec libx264 -acodec aac -async 1 -strict -2 #{@path_to_file}`
```

The video is then saved into the public folder, to save the video onto Amazon S3, using the paper clip gem needed to open the video as a file, loop through it and assign it to the @quote.snippet as would be the case with a temporary file being uploaded.

```
File.open(@path_to_file, "r") do |f|
  @quote.snippet = f
end #
@quote.save
```

The Amazon Bucket URL is then saved in the database for convenience of retrieval, using string interpolation and the snippet is removed from the root of the application, since it has been moved to the Amazon Bucket.

```
@quote.link = "http://#{ENV['AMAZON_S3_BUCKET']}.s3.amazonaws.com/videos/q
uotes/#{@quote.id}/#{@snippet_file_name}"

@quote.save

`rm -rf #{@path_to_file}`
```

There may be a way to cut the video directly onto the bucket but this was the most straightforward option for now. Another option could be to look into third party API for server side video trimming, however this was not considered at this time as it could incur higher costs.

Generating HTML Export Code

When AJAX receives data from the call back from controller a text area is used to provide the HTML top part of the code. It contains CSS style, JS script to enable a drop down of the video, without using the Bootstrap drop down component to ensure wider compatibility when clicking on the quote.

The top part of the HTML code is generated using the code in the invisible text area textAreaExport. This was done for simplicity and because it makes it easier in a more human friendly format to change and customise the export code as opposed to if it was one long string. It also contains the CSS to style the quote.

```
<!--Place holder textarea for styling of block quote on export, not visible on the page--
>
<textarea name="hide" id="topExport" style="display:none;">
  <style>
    /*styling of the video quote*/
  </style>
  <script>
     /*script for dropdown on video*/
  </script>
<div class='video-quote' id='videoQuoteText-id-n'>
  <span class='glyphicon glyphicon-play-circle'></span>
  <small>&#9658;  Watch Video </small>
</textarea>
<!--end -->
```

The callback receives the url of the recently cut snippet, which we need to include in the HTML of the export.

In the second part of the AJAX request when the callback is received the rest of HTML code is constructed using string concatenation and string interpolation.

The export text area in the export HTML modal is populated using string interpolation.

5.4. Server Side Deployment

Since the project is going to be open-sourced, it was decided to use Heroku style deployment as this is the most popular. This works with the Times infrastructure as they use Deis on top of their Amazon cloud services.

This uses a build-pack to add Ruby to the server, as well as a multipack setup to install FFMPEG onto the server.

See appendix for further details.

5.5. Summary

This chapter considered the design and implementation choices, concentrating on the most interesting solutions.

6. TESTING

The application was extensively tested to ensure that it fully worked, and the tests proved successful.

The testing strategy consisted in identifying the most important components and use a mixture of manual and automated testing to ensure they were working correctly. Given more time, more automated testing would have been put in place. Edge cases and data validation was also addressed.

External libraries and APIs were not tested as deemed outside of the scope of a good testing strategy, however SDK component for spoken data API developed by the author, and srt to hyper-transcript parser were identified as key component of the system and tested extensively.

Before considering the various type of tests, the structure of the application was revisited to identify the most important components that were deemed necessary testing.

6.1. Identifying Most Important Components

6.1.1. Overview of System

Here, an overview of the system following the flow of data will be considered.

A user signs up and authenticates using Google authentication.

Initially a video is uploaded into the application, by an authenticated user.

The video and its attachment is saved and sent to the speech-to-text API for processing. When the video is ready, a transcription of the video can be requested by the application. Once the application receives the transcription it parses it and saves it onto the database.

The transcription is displayed in the view as a hyper-transcript with the corresponding video. The hyper-transcript and the video have a certain degree of client side interactivity, to allow the user to select, preview and export a quote.

When exporting a quote, the video is trimmed and a HTML embed code is generated allowing the video quote to be embedded elsewhere.

6.1.2. Identifying the Most Important Components

From this analysis, it emerged that the most important aspect to test was the integration testing. The **user journeys** presented in the requirements chapter, can be

seen as a sequence of requests, that correspond to the minimum level of integration testing, that would guarantee a working application.

However because of the delay in processing and returning transcription by the text-to-speech API, not all user journeys could be tested within a given timeframe, without having to wait for the API response. The API response when processing a video file can take anything from fifteen minutes to a couple of hours depending on a number of factors.

Therefore Unit testing the **Video model**, for parsing and saving of the transcription was also deemed to be an important part.

Testing the **Spoken Data API SDK** implemented by the author was also considered to be an important part, as well as testing the **method to parse the srt file into a hyper-transcript** data structure.

The video model was also tested for validating the input of the **video upload form**.

How the various parts were tested during the overall testing strategy will now be considered in more detail.

6.2. Testing User Journeys

Integration testing in terms of user journeys, as a sequence of requests, was tested both manually and automatically.

6.2.1. Automatic Testing - Selenium

Automated testing using the application Selenium covered **use case one and two**, from the requirements chapter, i.e, a user signing up and uploading a video.

Selenium works by initialising a web driver that opens an instance of the Firefox web browser. The driver can then be set to navigate to a defined url.

For this, a test Gmail account (without step 2 factor authentication that is enabled by default for The Times & Sunday Times staff) was used to speed up the process. Corresponding environment variables were added to the project and used in the script to make it easier for other developers to use once the project is open-sourced.

Below are two extracts demonstrating how login and video upload were scripted, while an overview of how Selenium works with the code of the test can be found in the appendix.

Testing the Login - Selenium

Here is an example with inline comments of how the login functionality was tested.

In Selenium the find_element method is used to grab HTML elements on the page, by class or ID. The send_keys element is used to fill in the text of an input field and the click method does what the name suggests. A combination of find element, send keys and click can be used to navigate between pages.

```
# getting email input field element
email = driver.find_element(:id => 'Email')
# adding text, email address, to the input field
email.send_keys ENV['TEST_EMAIL']
# getting submit/next form button
next_btn = driver.find_element(:id => 'next')
# clicking on button to submit form
next_btn.click
```

Testing Video Upload - Selenium

It was also possible to test the upload of a video file, by passing the file path using send_keys, after which when the user clicks on the button, this uploads the video.

```
video_media_file =driver.find_element(:id => "video_media")
video_media_file.send_keys Rails.root.join('test', 'selenium', 'test_video.mp4').to_s
# saving
save_btn = driver.find_element(:id => "saveBtn")
save_btn.click
```

Once the Selenium script was written this proved to be a quick and easy way to test whether the sign in and video upload worked as expected.

6.2.2. Manual Testing

The remainder of the journey that had not been tested with Selenium was tested manually. This involved testing components richer in client side interactivity and therefore harder to test programmatically.

Use case 3, Analysing a Transcription

Once navigated to the video show.html.erb page, the client side interactivity of the transcription was tested, which is mostly Javascript and JQuery specific code for the video show page.

Test ID	Hyper-transcript Component	Description	Expected Result
1	Search	search the transcription	matched words become bold, and the text is highlighted

2	Click	clicks on hyper-transcript	Video starts playing at corresponding point of the transcription.
3	play	plays and pauses the video.	when clicking pause it pauses, when clicking play it plays.
4	auto hi-light	words change colour depending on video playhead position, while video playing	words before the one clicked become back, the word following becomes grey and the colour changes as video in playing.
5	auto hi-light	words change colour depending on video playhead position	Scrubbing video played back and forth to see text colour change, to corresponding point in transcription.
6	Sync Scroll on playing	Scroll sync between transcript and video	While video is playing Transcript scrolls to keep words corresponding to current playhead point on screen
7	Sync Scroll on playhead change	Scroll sync between transcript and video	Scrubbing video played back and forth moves transcript scroll sync position
8	Sync Scroll on word click	Scroll sync between transcript and video	clicking to another word moves transcript scroll sync position

Use Case 4, Selecting and Previewing a Quote Selection

Test ID	Preview Component	Description	Expected Result
8	Select	selecting text of transcription	automatically prepopulate select form quote text area, and input fields for duration, input and output.
9	Edit selection	edit text of quote in text area	textarea for quote selection is editable
10	Add Author	Add author's name	input field for author is editable

		to selection	
11	Preview	click on preview button	opens pre-populated preview modal with quote and no video visible (at this stage)
12	Preview - Dropdown	Clicking on quote in preview modal	Video drops down and starts playing from input, stops at output of selection and resets playhead to input.
13	Preview - Dropdown	clicking on quote while video is playing	video pauses whilst playing and disappears, if clicking again video re-appears and starts playing again.

Use Case 5, Exporting a Video Quote Selection

Test ID	Export Component	Description	Expected Result
13	Export button	click on export button	the button gives visual feedback that the application is processing the request
14	Export Modal	When processing is complete modal returns results	Text area with HTML embed code containing, Style tag with CSS style for blockquote, script tag with Javascript code for dropdown, blockquote HTML element, video element with URL with word snippet in the name.
15	trimmed video	To check the video trimmed is a valid video copy url of video in video tag element in new browser window and check if the video plays	video plays and duration match expected.

6.3. Unit Testing

To test the video modal, unit testing was used and key methods within that modal.

6.3.1. Testing the Video Model

Because of the way the application has been implemented the correct working of the video model is crucial for the rest of the application to work as expected and it was therefore necessary to put extra care into testing its functionality.

How this was done both manually and with automated testing will now be considered.

Automated

The model was unit tested using the Rails built in test suite. *See appendix for a full breakdown of tests results.*

Main areas that were tested were around validating presence of title and video file when creating a new video from the upload form. This was tested both in success and failure cases, such as the edge case where a user might try to submit the form with empty fields.

The method to parse a string containing the content of an srt file was tested. Because the content of an srt file changes every time a new video is uploaded, a more flexible way of testing the consistency of the format was needed. A regular expression to describe the structure of a hyper-transcript was therefore used, as a spectrum of arrays containing words. Each word is a Ruby hash that has key value pairs for time in, time out and text. An array of words is a line from the transcription. The outer array is the transcription and contains all the lines.

See below an example of a hyper-transcript data structure.


```
[
        [
{"tc_in"=>0.049, "tc_out"=>1.1656666666666666, "word"=>"God"},
 {"tc_in"=>0.4212222222222222, "tc_out"=>3.399, "word"=>"maintain"},
 {"tc_in"=>0.7934444444444445, "tc_out"=>1.1656666666666666, "word"=>"I"},
 {"tc_in"=>1.1656666666666666, "tc_out"=>2.2823333333333333, "word"=>"and"},
 {"tc_in"=>1.5378888888888889, "tc_out"=>3.026777777777778, "word"=>"Kyle"},
{"tc_in"=>1.910111111111111, "tc_out"=>3.399, "word"=>"show"},
 {"tc_in"=>2.2823333333333333, "tc_out"=>3.7712222222222223, "word"=>"from"},
 {"tc_in"=>2.6545555555555556, "tc_out"=>6.004555555555555, "word"=>"Lewistow
n"},
```

{"tc_in"=>3.026777777777778, "tc_out"=>5.632333333333333, "word"=>"Montana"}

],

[

{"tc_in"=>3.399, "tc_out"=>6.5, "word"=>"atomic"}, {"tc_in"=>3.9158333333333335, "tc_out"=>7.533666666666667, "word"=>"Western"}, {"tc_in"=>4.432666666666667, "tc_out"=>7.533666666666667, "word"=>"called"}, {"tc_in"=>4.9495000000000005, "tc_out"=>8.567333333333334, "word"=>"Bochner"}, {"tc_in"=>5.466333333333333, "tc_out"=>8.0505, "word"=>"since"}, {"tc_in"=>5.983166666666667, "tc_out"=>8.050500000000001, "word"=>"2008"}

],

...

]

A regular expression was used to test the validity of the hyper-transcript data structure returned by the method convert_srt_to_word_accurate_hypertranscript that was parsing an srt file and returning the hyper-transcript data structure. A sample srt file was used for this purpose.

The method to save the hyper-transcript data structure into the database save_to_db was tested as well.

6.3.2. Testing the Spoken Data API SDK

The spoken data API SDK was tested by initialising it as an object and testing the most important methods send_by_video_url and retrieve_subtitles_srt by running the script and examining the results.

```
##################### How to use, example in 3 steps
# 1.Creates a spoken data api instance initialising it with `user_id` and API key.
spokenDataAPIObject = SpokenDataAPI.new(ENV['SPOKEN_DATA_USER_ID'],ENV['SPOKEN_DATA_API_TOKEN'])

# 2. send video by URL
# sending a video by url will return you the spoken data api assigned video ID
uid = spokenDataAPIObject.send_by_video_url("https://youtu.be/e6GFEfJtlnc")
puts "UID: "+uid

# 3. retrieve captions
# use recording uid to retrieve subtitles, when ready.
# returns srt file if ready, boolean false if not
```

```
# as a first run when uploading you should expect false as your video is still being pro
cessed.
response= spokenDataAPIObject.retrieve_subtitles_srt(uid)
puts "Response: "+response.to_s
#####################
```

To test that the SDK was retrieving the UID correctly from the response, it was necessary to wait until the video had been processed, or to test the method with the UID of the video already in the system.

6.4. Summary

This chapter has considered the testing strategy used to ensure a working application was delivered and has explored how this was done with a mixture of manual and automated testing. Focusing on both the integration testing on user journeys gathered in the requirements stage and the use of unit testing for key methods of the video modal.

7. CONCLUSIONS AND PROJECT EVALUATION

This chapter will give a summary of the achievements of the project, critically evaluate major parts of the project and discuss suggestions for future work.

7.1. Summary of what the project has achieved

The project has achieved the building of a working web application that meets the defined user requirements as follows:

- The application returns a searchable hyper-transcript in sync with the video
- Allows the user to select a quote
- Allows the user to preview their quote selection before exporting
- Cut the video in the background once a quote selection is being exported, using FFMPEG.
- Export the video quote clip as HTML5 mp4 H264
- Generates HTML code to embed the video quote elsewhere

Additionally, the project also:

- Investigated and established a solution to the problems that arise when working with video in the newsroom
- Built a Ruby SDK for spoken data speech-to-text API
- Refactored hyper-transcript and hyper-audio converter in Ruby
- Tested the application manually and with Selenium
- Deployed on Heroku / Deis - delivered a system manual
- Delivered a user manual - delivered code documentation
- Released the project as an open-source project

7.2. Critical Evaluation of the Project

Four main challenges were identified around the successful realisation of the project; investigating the problem, providing a comprehensive solution, identifying the best system to generate video transcriptions and last but not least provide a suitable output.

These challenges will now be considered individually.

7.2.1. Investigation of the Problem

The problem of optimising the use of video in the newsroom was explored with an iterative approach and a series of rapid prototypes, presented to and discussed regularly with the users.

Use of NLP APIs

The use of natural language processing to give an insight into the transcriptions, was an interesting experiment. However, further research into how to correlate and interpret the results returned from the NLP API is needed.

Use of Summarisation APIs

The lack of punctuation proved to be a problem when using summarisation APIs, because they work by ranking sentences delimited by full stops. However an efficient solution was identified in considering the timecode line breaks of the subtitle file as sentences to gain a useful approximation. With further research, it was discovered that the automatic suggestion of quotes was not a necessary requirement for journalists in this context. But rather to be able to find quotes, implementing a word search was a sufficient solution.

7.2.2. A comprehensive solution

Because of the iterative nature of the investigation the resulting application, grounded on solid research and insight, has proved to be a successful comprehensive solution to a real newsroom problem.

7.2.3. Video Transcriptions

With an iterative approach and the use of rapid prototyping, the project successfully investigated the possibilities around facilitating and optimising the use of video in the newsroom. The initial hypothesis that transcription would have been useful in giving a way into the content, was in fact correct.

SpokenData was chosen as the most appropriate solution due to it’s ease of use and flexibility discussed in previous chapters. However, to ensure a modular component based design, the RESTful API URLS of the speech to text API were implemented as a separate component through an interface that would allow a change in the speech-to-text API in the future should a future need arise.

7.2.4. Output

Great care was put into delivering an output of the video quote as something that could easily be integrated with the existing system with minimal effort, hence the decision to trim the video and generate an HTML code that the user can embed elsewhere in their news article.

Particular attention was also put into using FFMPEG to create a H264 mp4 video file compliant with HTML5 standards, to ensure greater cross browser compatibility.

7.3. Future Work

The project so far has satisfied the current project goals. However some suggestions will now be considered for future work, should time allow.

7.3.1. Add Support for Audio File

FFMPEG and spoken data already support audio so it is simply a matter of adding this for the video and to quote Rails' model paperclip settings for file attachment of the audio file type.

7.3.2. HTML5 Video, OGG and WebM

The system exports as MP4 H264. HTML5 supports a number of video sources, which means that the future implementation of the system could also transcode the video into ogg and WebM to ensure greater cross browser compatibility.

7.3.3. Twitter export

With the very recent introduction of support by the Twitter API for tweets with video attachments, it could be an interesting output option for quickQuote. A set of constrains a round characters count, video file size and length would need to be taken into account.

7.3.4. Restrict Access to @times.co.uk Addresses

Despite the project residing on an internal domain known only to Times employees, it would be recommended to implement a restriction on the user login that checks against a valid @times.co.uk address.

7.3.5. More Tests

Given more time, a more comprehensive set of unit and integration tests could be written making use of the built-in facilities provided by the Rails framework.

7.4. Wrap-up

Delivering an application that provides a comprehensive solution for journalists working with video in the newsroom is a challenging task. However grounded on solid research, an iterative approach that made ample use of rapid prototyping and following a short feedback loop with the users, the project has not only investigated an optimal solution but also produced a working application that satisfies all of the requirements.

Furthermore the author believes the insight from this investigation could provide a starting point for more efficient ways of working with video through the use of transcriptions.

8. SYSTEM MANUAL

Demo and details of the application can be found here - times.github.io/quickQuote

This is a step by step plan on how to install quickQuote. It will get you to a point of having a deployed instance on Heroku.

8.1. Prerequisite

You will need to have the following installed

- Ruby
- Ruby on rails
- Git
    - Git installed locally
    - Github
- Heroku
    - Heroku account
    - Heroku installed locally
- ffmpeg (for running development version locally)
    - MySql
    - MySql - Sequel Pro

Showing how to install these is outside of the scope of this system manual.

8.2. System dependencies

a list of most relevant system dependencies

- ruby version 2.0.0
- rails version 4.2.0
- ffmpeg version 2.7.1
    - –-enable-shared
    - –-enable-pthreads
    - –-enable-gpl
    - –-enable-version3
    - –-enable-hardcoded-tables
    - –-enable-avresample
    - –-cc=clang
    - –-host-cflags=
    - –-host-ldflags=
    - –-enable-libx264
    - –-enable-libmp3lame
    - –-enable-libvo-aacenc
    - –-enable-libxvid
    - –-enable-libvorbis

    –-enable-libvpx
    –-enable-vda
- Video js 4.12.8 *cdn*
- jquery 3.0.0-alpha1 *cdn*
- bootswatch paper bootstrap.min.css 3.3.5 *cdn*
- ajax jquery 1.11.2 *cdn*
- bootstrap js 3.3.5 *cdn*

see Gemfile for list of rails gem dependencies.

8.3. Configuration

8.3.1. API keys

You'll need to make the following accounts to get the API keys

- Amazon S3
    –production
    –development
- spoken data API
    –production
    –development
- Google API
    –production
    –development
- local mysql database for testing - *optional*

If not using Heroku, but deploying Heroku style, for instance on Deis, you'll also need a separate database.

optional - amazon RDS

- production
- development

spoken data

Getting two API keys for spoken data requires two email addresses to make two distinct account(development and testing can use the same address), it is possible to just use one account for production and development, but I'd advise against as it can get very confusing very quickly if you try to distinguish on their dashboard which videos you uploaded in development and which ones are your users'.

Google API key

In the Google developer console, create a new project, and get the client id and client secret.

You'll also need to **enable Google+**.

You'll also also setup a call back URI, such as

http://name_of_your_application.herokuapp.com/auth/google_oauth2/callback

You'll need a URI for the deployed application and one for development. Google does not support adding localhost, so you'll need to setup a custom local url. explained below

Local domain name for use in development

For use during development it is required to setup a local DNS for your local host address that maps to your localhost.

In terminal

$ **sudo** nano /etc/hosts

in the hosts file

127.0.0.1 localhost **your local domain name**.com

You can use same details for development and testing. This needs to be added to the Google Console API as the redirect URL. so that in config/application.yml you can set the environment variable to be:

REDIRECT_URIS: 'http://your_local_domain_name.com:3000/auth/google_oauth2/callback'

testing optional

It can be useful to get a dummy gmail address without step two authentication for use with selenium for testing.

These need to be added to the config/application.yml file that contains the project environment variables

```
# Dummy Gmail Logins for testing
LOCAL_DOMAIN: 'http://your_local_domain_name.com:3000'
TEST_EMAIL: 'whatever@gmail.com'
TEST_EMAIL_PASSWORD: 'password'
```

more details on how to make application.yml below

8.3.2. Add API Keys to project

we are using the figaro gem to deploy our environment variables onto Heroku. add API keys to a file config/application.yml

dividing by production and development.

$ bundle exec figaro install

creates the config/application.yml and adds it to .gitingore.

you can use the following template to fill in the details in config/application.yml

```
GOGLE_PROJECT_ID: ''

test:
  #Google Project ID, OAuth 2.0 client ID:
  GOOGLE_CLIENT_ID_DEVELOPMENT: ''
  GOOGLE_CLIENT_SECRET_DEVELOPMENT: ''
  REDIRECT_URIS: 'http://your_local_domain.com:3000/auth/google_oauth2/callbac
k'
  JAVASCRIPT_ORIGINS:  'https://www.example.com'
  #MysQL db - Local
  DB_TEST_USERNAME: 'root'
  DB_TEST_PASSWORD: ''
  DB_TEST_DB: ''
  DB_TEST_END_POINT: '127.0.0.1'
  DB_TEST_PORT: '3306'
  #S3 Bucket
  AMAZON_S3_ACCESS_KEY_ID: ''
  AMAZON_S3_SECRET_ACCESS_KEY: ''
  AMAZON_S3_BUCKET: ''
  #Spoken Data API
  SPOKEN_DATA_USER_ID: ''
  SPOKEN_DATA_API_TOKEN: ''

development:
  #Google Project ID, OAuth 2.0 client ID:
  GOOGLE_CLIENT_ID_DEVELOPMENT: ''
  GOOGLE_CLIENT_SECRET_DEVELOPMENT: ''
  REDIRECT_URIS: 'http://your_local_domain.com:3000/auth/google_oauth2/callbac
k'
  JAVASCRIPT_ORIGINS:  'https://www.example.com'
  #MysQL db - Amazon RDS
  RDS_USERNAME_DEVELOPMENT: ''
  RDS_PASSWORD_DEVELOPMENT: ''
  RDS_DB_DEVELOPMENT: ''
  RDS_END_POINT_DEVELOPMENT: ''
```

```
  RDS_PORT_DEVELOPMENT: ''
  #S3 Bucket
  AMAZON_S3_ACCESS_KEY_ID: ''
  AMAZON_S3_SECRET_ACCESS_KEY: ''
  AMAZON_S3_BUCKET: ''
  #Spoken Data API
  SPOKEN_DATA_USER_ID: ''
  SPOKEN_DATA_API_TOKEN: ''

production:
  #Google Project ID, OAuth 2.0 client ID:
  GOOGLE_CLIENT_ID: ''
  GOOGLE_CLIENT_SECRET: ''
  REDIRECT_URIS: 'http://the_name_of_your_app.herokuapp.com/auth/google_oaut
h2/callback'
  JAVASCRIPT_ORIGINS:  ''
  #MysQL db - Amazon RDS
  RDS_USERNAME: ''
  RDS_PASSWORD: ''
  RDS_DB: ''
  RDS_END_POINT: ''
  RDS_PORT: ''
  #S3 Bucket
  AMAZON_S3_ACCESS_KEY_ID: ''
  AMAZON_S3_SECRET_ACCESS_KEY: ''
  AMAZON_S3_BUCKET: ''
  #Spoken Data API
  SPOKEN_DATA_USER_ID: ''
  SPOKEN_DATA_API_TOKEN: ''
```

8.4. How to run the test suite

There is a Selenium script to programmatically test login and file upload.

To run it from root of the application use the following command.

$ ruby /test/selenium/selenium_test_video_upload.rb

For this to work the Rails server needs to be running. rails s in terminal from root of application.

To unit test the Video model.

$ **rake** test test/models/video_test.rb

8.5. Database creation - for local development and testing.

This can be used for testing as well.

- Install mysql locally *os x*
- To start the server System preferences > MySQL > Start MySQL Server
- Use Squel Pro (or equivalent) to connect to the local MySQL db server.

```
#MysQL db - Local
DB_TEST_USERNAME: 'root'
DB_TEST_PASSWORD: ''
DB_TEST_DB: ''
DB_TEST_END_POINT: '127.0.0.1'
DB_TEST_PORT: '3306'
```

For production, the database is created as part of the deployment script.

8.6. Deployment instructions

To deploy onto heroku cd into the application root folder, login into heroku.

$ heroku login

Then run one of the deployment scripts. Before running the script, inspect the script to customise the deployment to your needs.

When deploying the application for the first time run.

$ **./first_time_deploy.sh**

For subsequent deployments you can use

$ **./deploy.sh**

If after the first deployment you want to deploy as new application run

$ **new_deploy.sh**

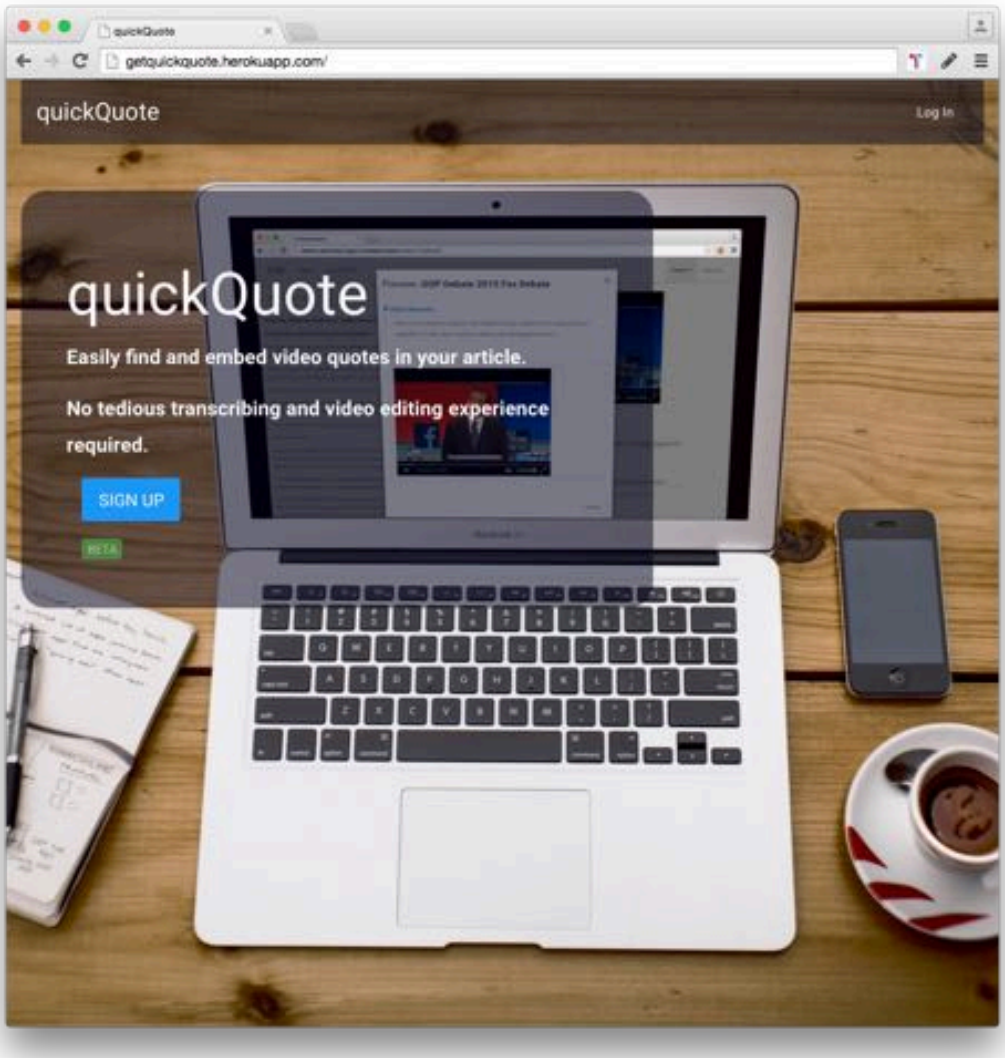
Note that this last one, does not delete the previous application, it simply removes the git remote. if you wish to delete it, you'd need to run heroku apps:destroy --app $app --confirm $app where $app is the name of the app you want to delete. Alternatively you can log in to your Heroku account and delete it from there.

9. USER MANUAL

“quickQuote” is a web application to select video quotes from a video, to embed in an article. It uses Spoken Data API to generate a transcription of the video. The user can then search, and select a quote. This can be exported, and the application trims the video, and generates the HTML code to embed it with the corresponding part of the video associated as a dropdown.

9.1. Login

Authentication is done through Google Auth, no need to register, just login with user Google credentials.

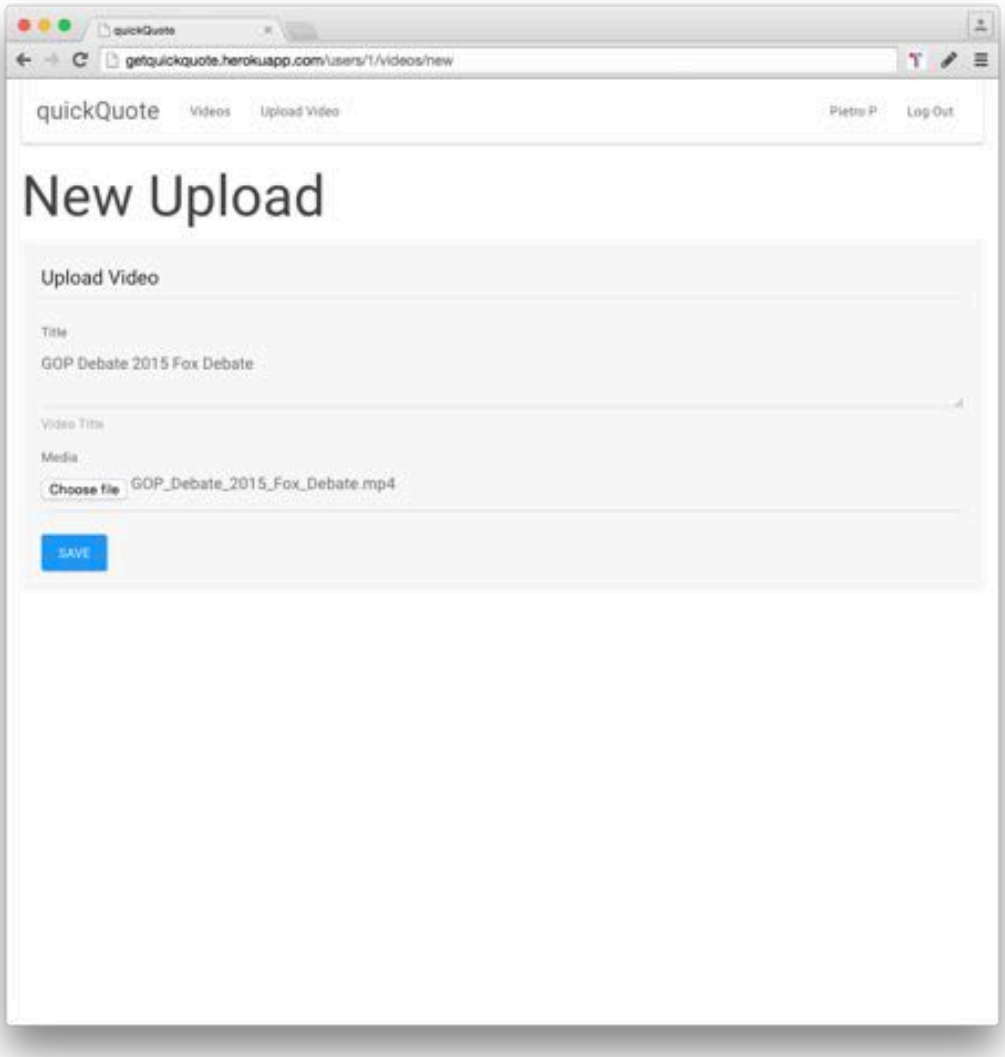


Login

9.2. Uploading a video

Once logged in the navigation bar displays two new menus Videos and Upload videos.

Click on Upload videos to get started.

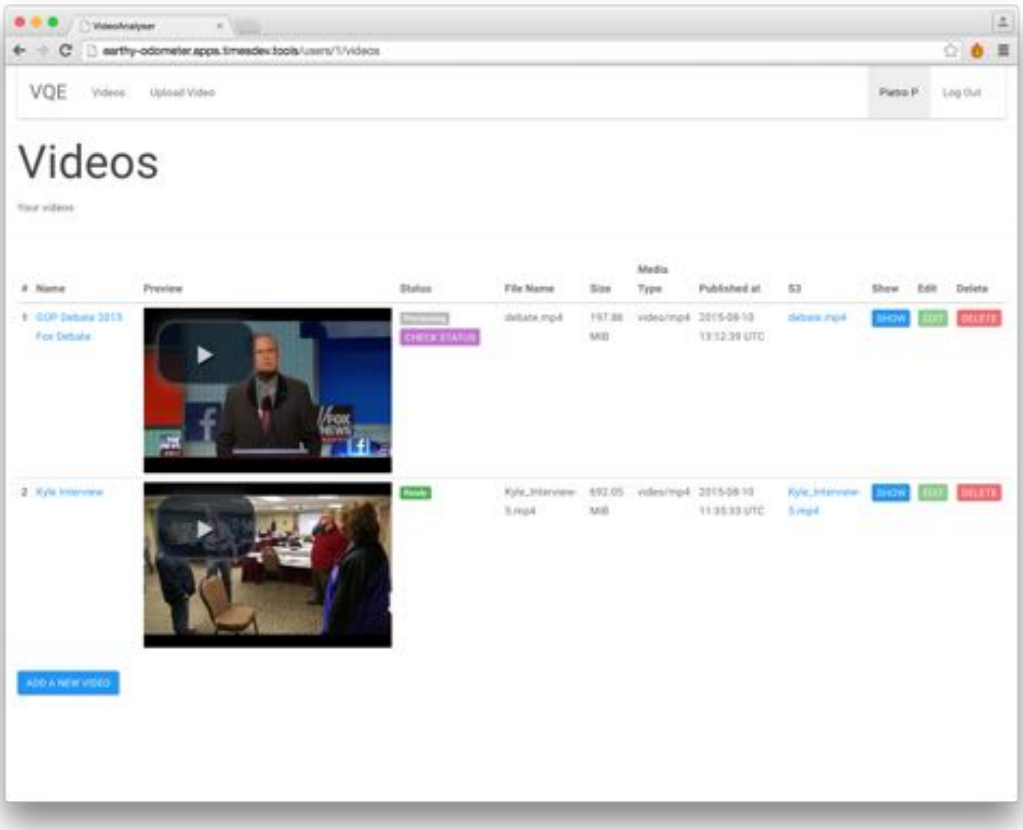


Upload

9.3. Viewing videos

You can check the status of a video by clicking on Check Status.

From there you can navigate to individual videos.

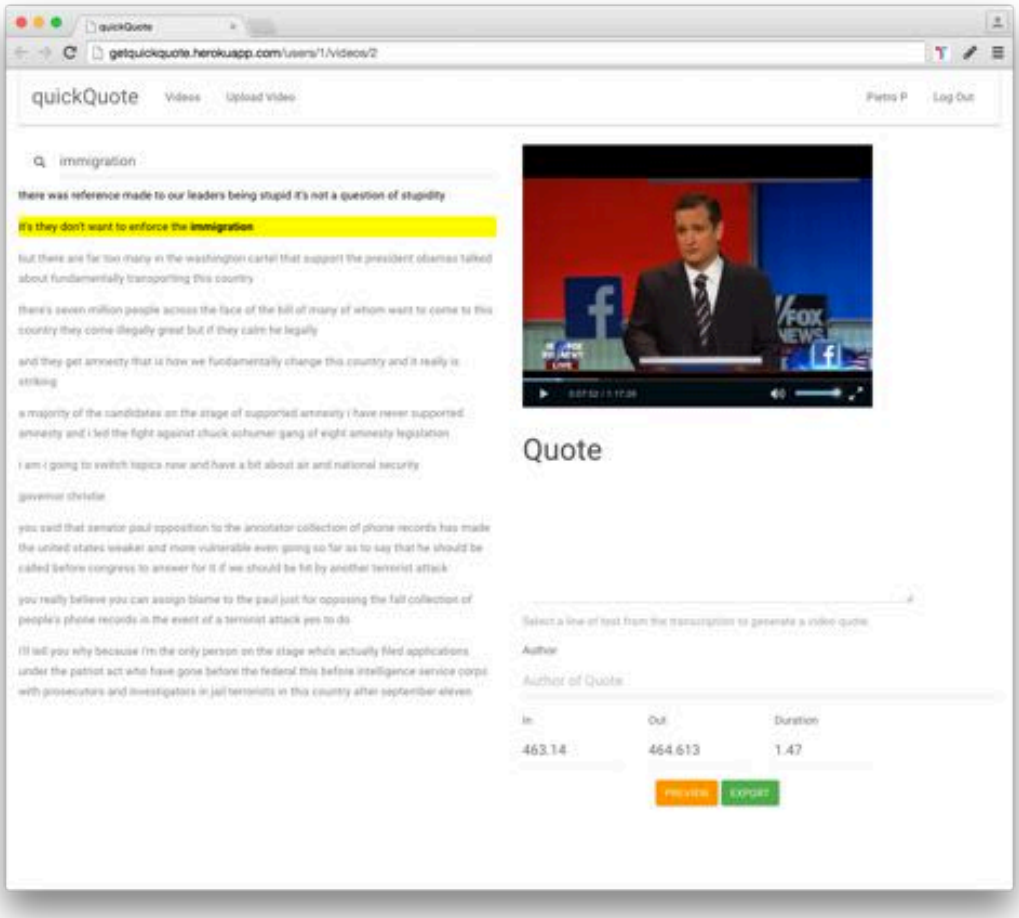


Videos

9.4. Hypertranscript

9.4.1. Search Video

You can search for specific words in the transcription. The line containing a match is highlighted and the word becomes bold.

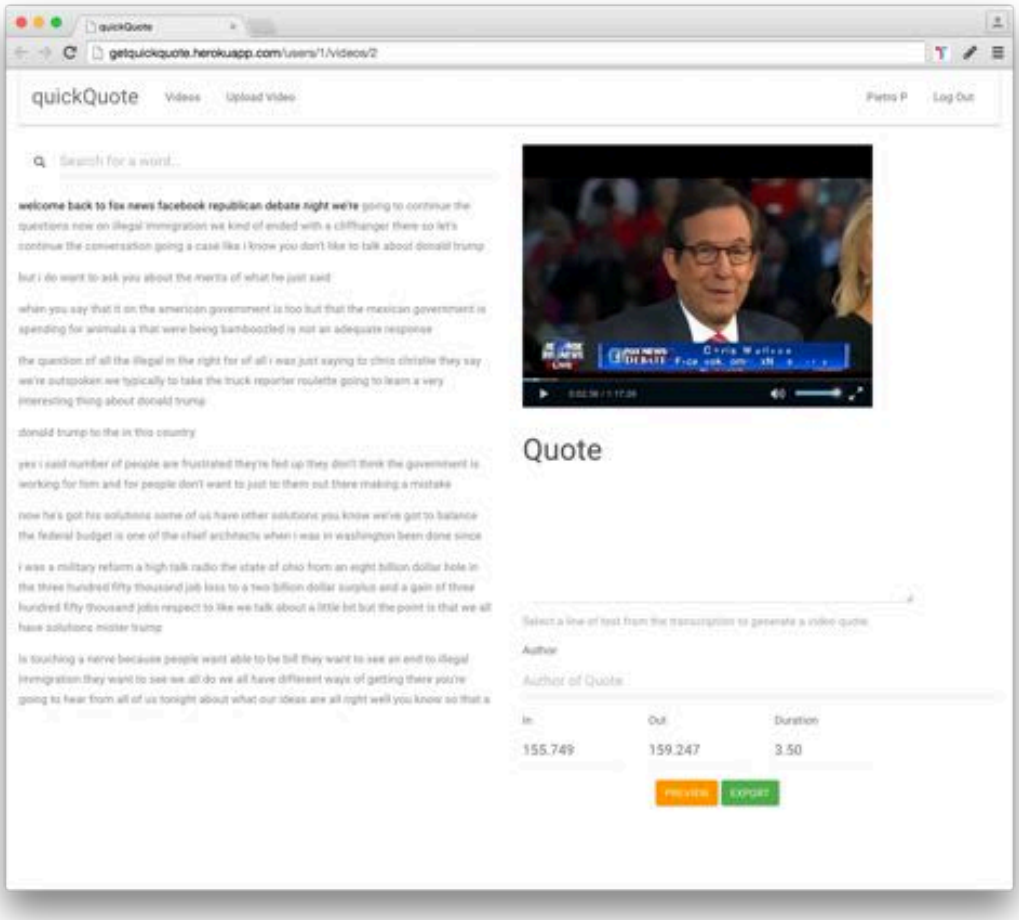


Search

9.4.2. Clickable transcription

The text in the hypertranscript is sync with the video playhead. Clicking on the transcription takes you to the corresponding point in the video.

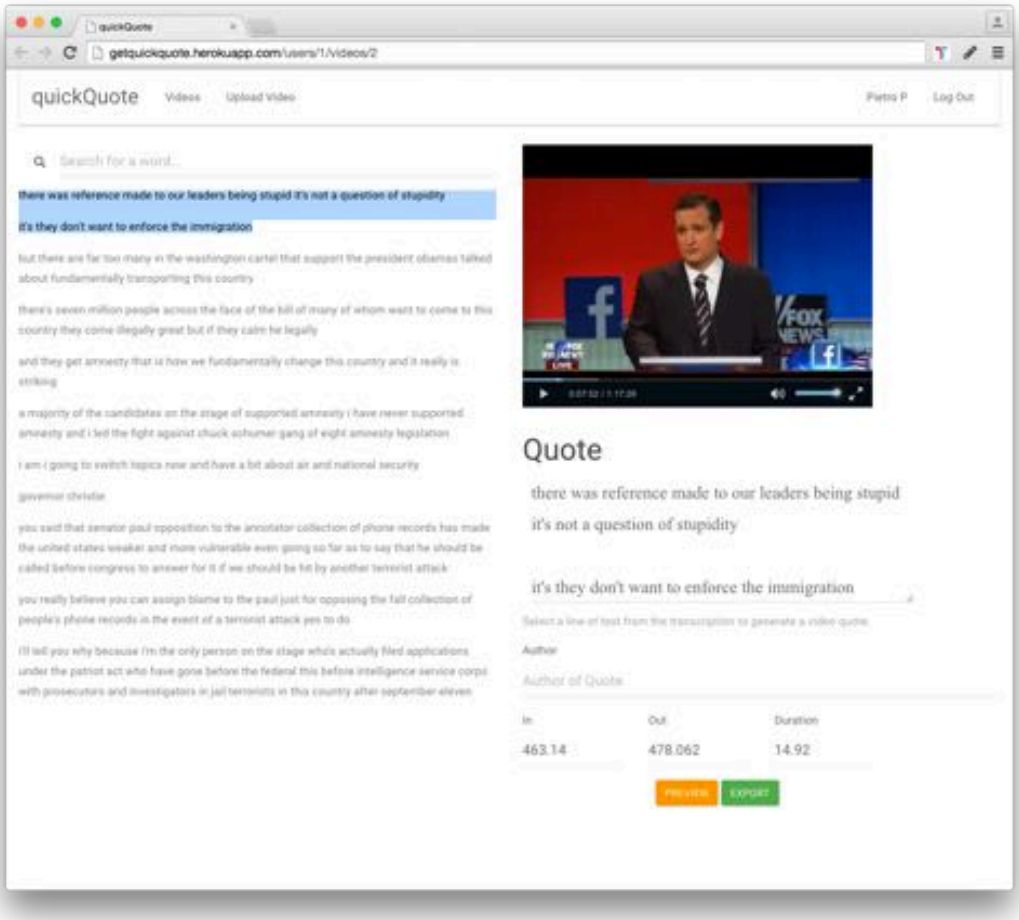
As the video playes the words change color to show the corresponding text.



Hypertranscript

9.5. Select a quote

To select a quote simply select it with your mouse. The Quote text area auto populates with the text you've selected.

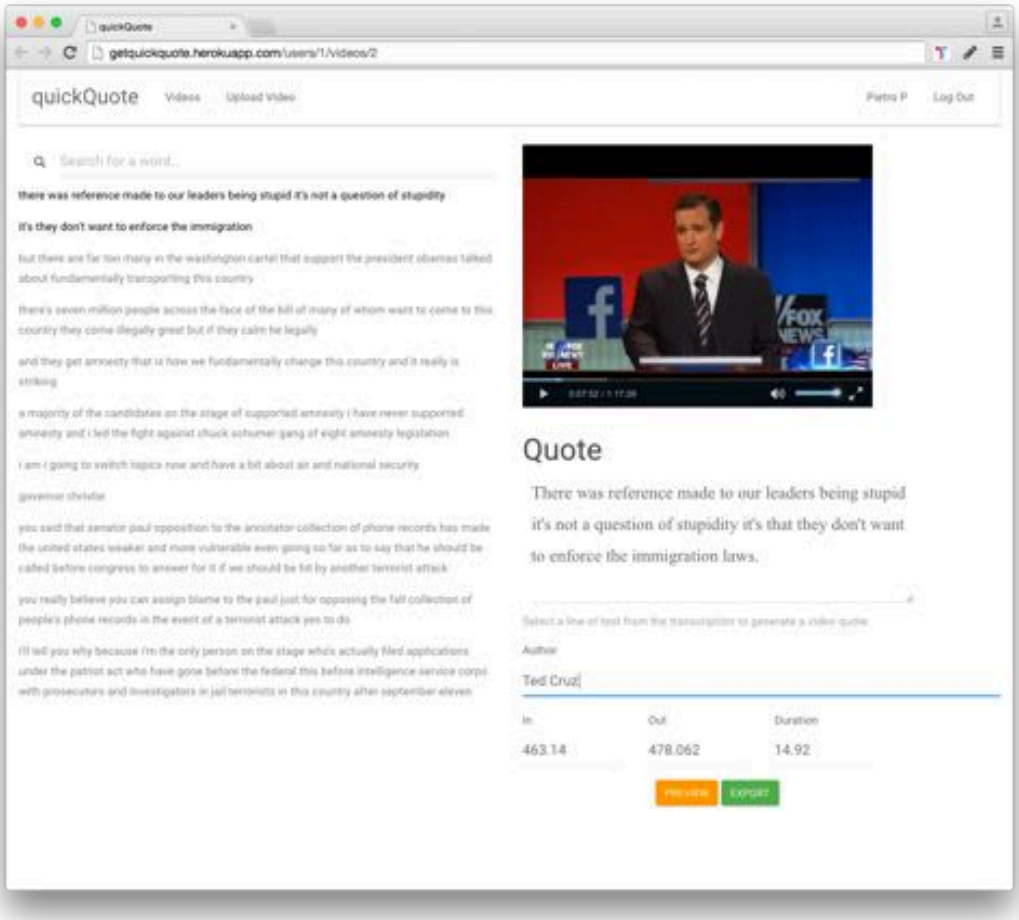


Selection

9.5.1. Edit a quote

In this example, the text to speech API was very accurate, but we notice that the sentence doesn't add up, and if we play the clip we can hear it missed the word "law".

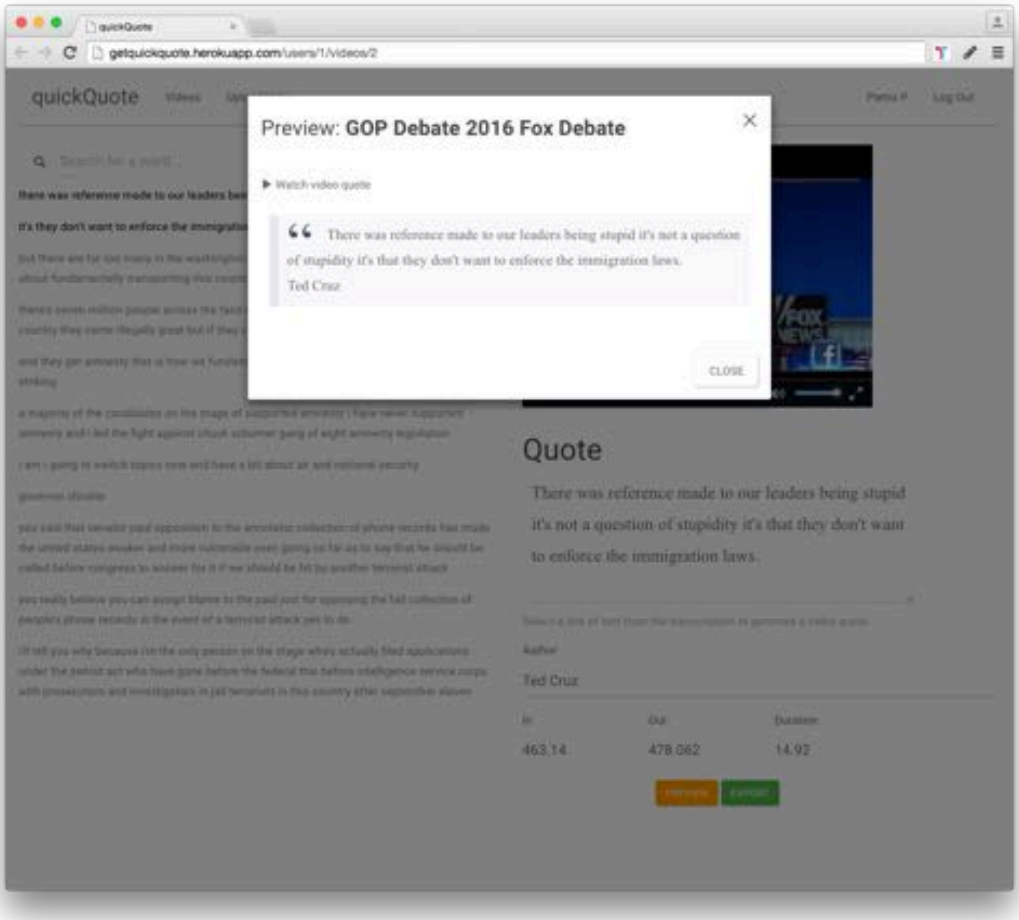
To fix this we can edit the selection of the quote for accuracy by editing the text in the Quote text area.



Edit the text of a quote selection

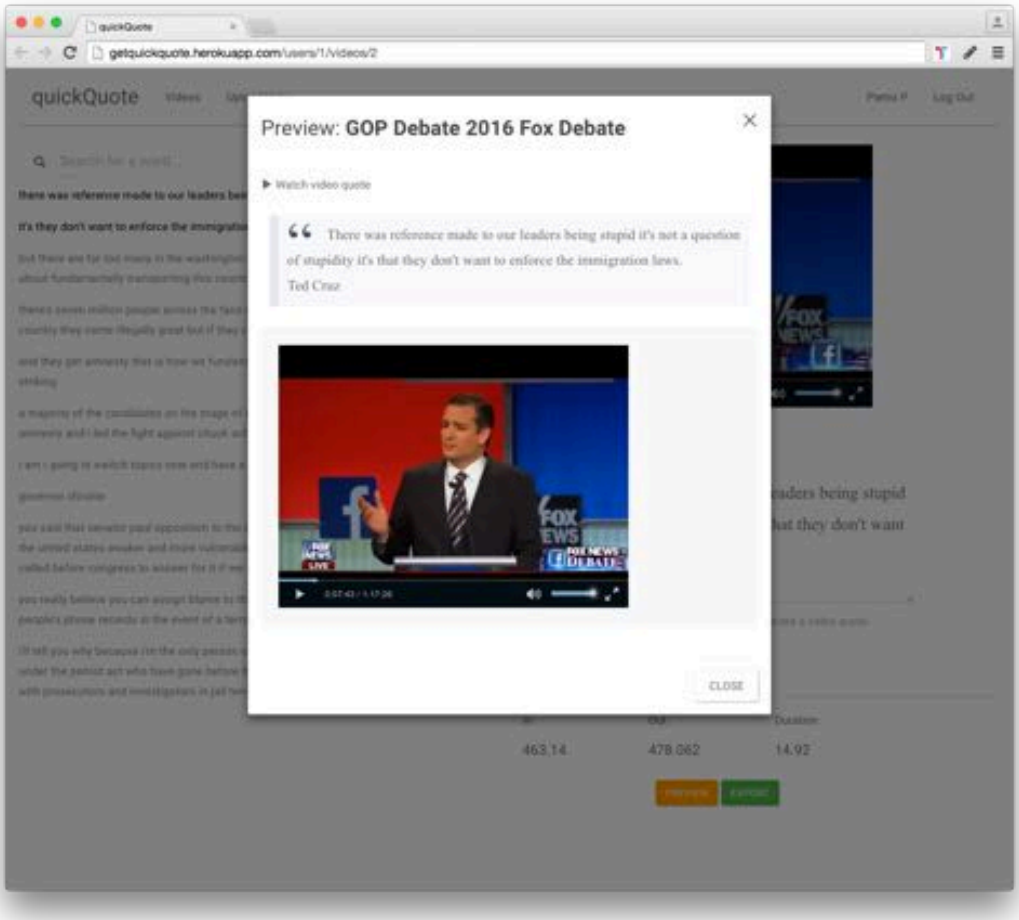
9.6. Preview

To quickly preview what our quote will look like once published, we can click on preview, and a pop up window with a preview of the quote will come up.



Preview Quote

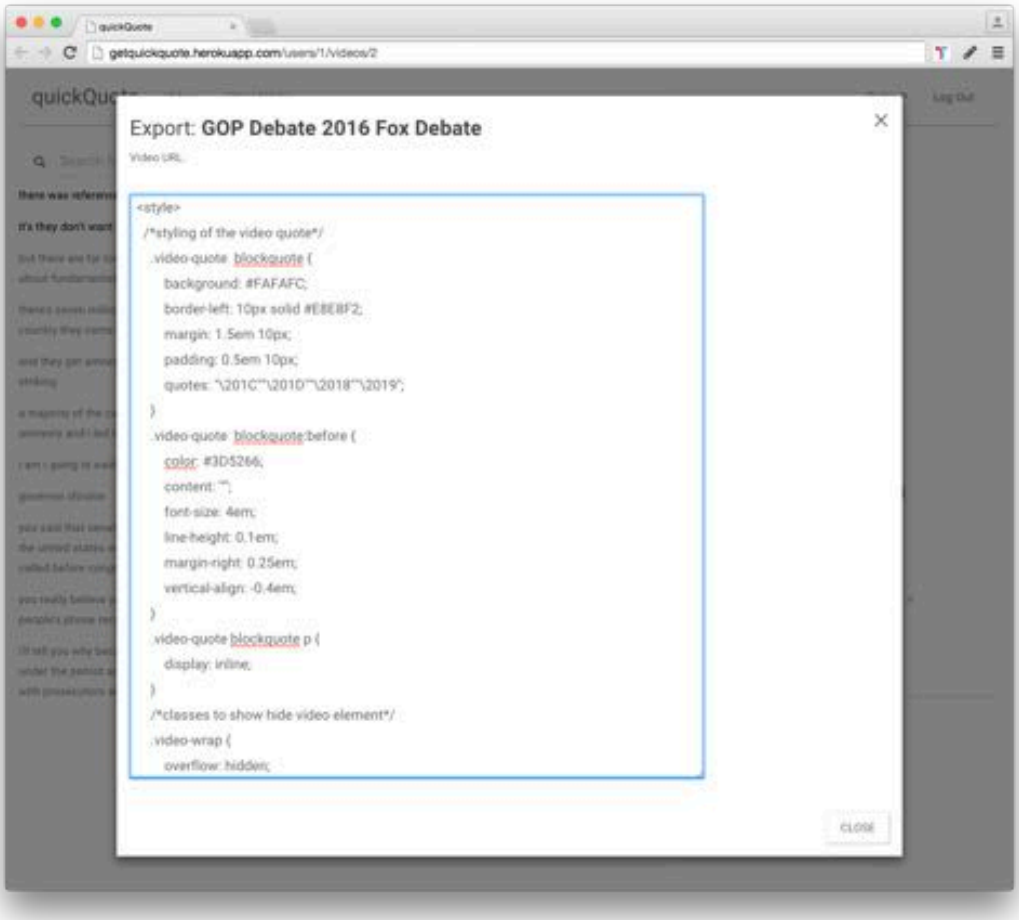
Clicking on watch video triggers the dropdown, and we see the video corresponding to the text we've selected.



Preview Video

9.7. Export

Once pleased with our selection and ready to publish, click on the export button. Another pop up window comes up, this time containing the HTML embed code, to insert in your article.

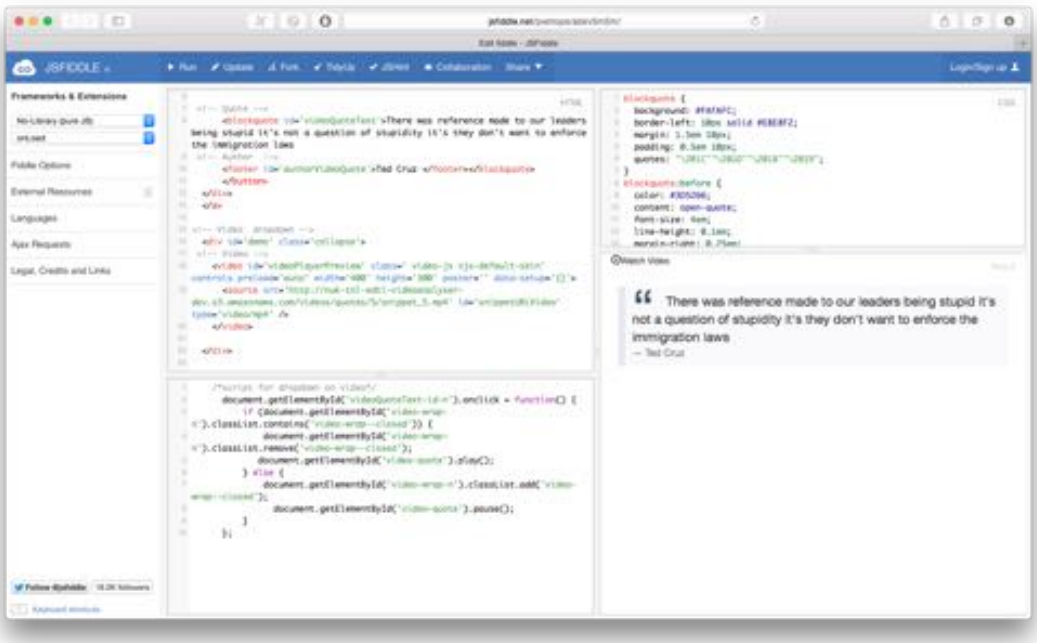


Export

9.7.1. Embedding the code

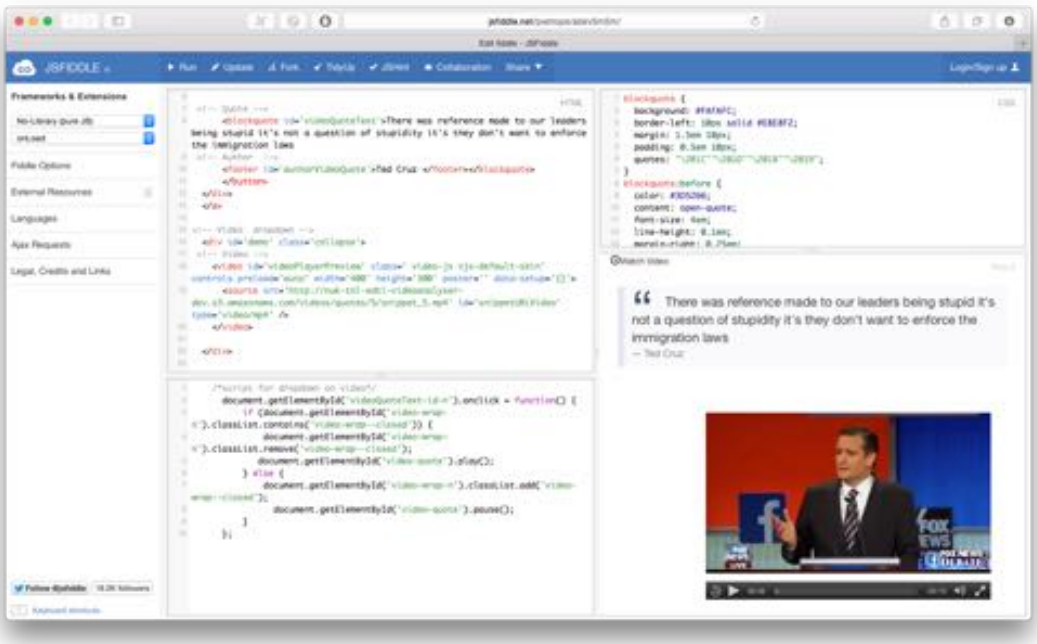
To demonstrate embedding the code, with use the JSFiddle HTML browser based preview editor.

As you can see the embed code renders the quote with a link to the dropdown of the video.



Export Quote

If we click on watch video the video segment comes down.



Export Dropdown

88

As you can see from the length of the clip in this example (7 seconds) the clip has been trimmed to the corresponding text of the quote to make for a faster loading of the video file.

10. OUTLINE OF PHASES

An outline of the phases of the project.

1. Preliminary research
    1. Initial stack research - *June*
2. Investigation into problem domain *Hypothesis and prototype*
    2. hyper transcript prototype - *end June begin July*
    3. NLP Prototype (Budget speech) - *beginning July*
        1. NLP summarisation
        2. NLP Keywords
        3. NLP Concepts
        4. NLP Entities
    4. Define requirements.- *beginning July*
        1. draft proposal
        2. draft functional requirements - *beginning/mid July*
3. Implementation
    5. Set up continuous deployment system- *mid July*
    6. Decide testing strategy - *mid July*
    7. Baseline project - mid/end July /mid August
        1. login
        2. Video upload
        3. Retrieve captions
        4. Dashboard
        5. Select a quote
        6. Preview
        7. Export video and text file
        8. Export to HTML embed code
    8. Documentation and refactoring - *beginning/mid August*
        1. Refactoring and tidying up
        2. code documentation
        3. system manual
        4. user manual
    9. use case
        10. Top 5 best quotes from “Donald Trump”
4. Writing up Report - *end August*

11. INTERNSHIP TIMELINE

Date	Week Number	Tasks / Phase
June	0	Preliminary research + BBC Hackathon
29th June	1	Hypertranscript + NLP prototype
6th July	2	Budget prototype
13th July	3	Define scope and requirements
20th July	4	Baseline project
27th July	5	Baseline project
3rd August	6	Baseline project + documentation
10th August	7	Documentation (user and system manual, code documentation)
17th August	8	Write up and refactoring

12. ABOUT RUBY ON RAILS MVC

Ruby on Rails is an MVC framework and it was chosen because of the speed it gives when prototyping.

MVC stands for Model View Controller, and is a way of separating the concerns of various parts of a web application.

The model contains abstract classes, with the business logic. The controller initialises the classes as needed and the views are used by the controller to populate templates as they contain any “display logic”.

Rather than giving an explanation of the framework as traditionally proposed by conventional books on the subject (Bigg et al. 2015), this paragraph will instead give an overview of the flow of data in the Rails framework from a user request from the browser through the Rails MVC framework (Passarelli Pietro 2013).

This will be useful to better understand the design and implementation chapter.

The user requests a page in the browser through a URL. Rails web server - Rack - handles the request. Firstly, it looks in the public folder, which is at root level of the application, then if it does not find it, it passes on the request to the routes.

The public folder contains static HTML and error pages.

The routes matches the URL to the action controller that handles CRUD, creates read update and delete methods.

The controller, initialised any object from the model classes, it also contains any conditional logic, such as ‘display this’ if the user is logged in, otherwise it will display a login page.

The controller uses instance variables to communicate with the views.

The model in Rails is implemented as Active Record Object Relational Manager (ORM).

The ORM handles the connection with the database and through migrations, Rails writes model classes as tables into the database, consistent with Object oriented principles where data modelling is class modelling.

The controller then uses the default Ruby erb templating to populate the views with instance variables. The views contain any display logic, and are rendered as HTML, which is then returned to the browser.

Rails was originally designed to work following RESTful principles, to provide navigation between pages. This is different from modern one page frameworks such

as Angular or React where parts of the page are updated without reloading the whole of the page.

It will become clear in the implementation chapter how AJAX, JQuery and Rails routes were used to achieve this.

13. RAILS AND CLIENT SIDE INTERACTIVITY WITH AJAX

In recent years there has been an increasing need for client side interactivity with client side Javascript and JQuery as well as the so called “single page applications” such as Angular and React.

There was a need for this project to work this kind of client side interactivity into the Rails framework. Rather then add one of the frameworks mentioned above, that although very powerful have quite a steep learning curve, a combination of JS, JQuery and AJAX were used instead.

AJAX stands for Asynchronous JavaScript and XML and is a way to send a request to the server of the application and update only part of the application without having to reload the page.

In the AJAX request we specify a URL and HTTP method to send the request to.

Here is an overview of an AJAX request with some pseudo-code in the inline comments.

```
//prep some variables to send in the ajax request
$.ajax({
   //setup and send the request
        type: 'POST',
        url: // the RESTfull url we are sending the request to,
        data: // a json object,
        dataType: 'json',
        async: true
    }).done(function(data){
      // specifies wthat to do with the call back function
          // for instance updating an html element on the page
    });
```

Then in the routes in Rails, we specify how to handle such requests when it comes in.

The routes specify the URL from which we receive the request and which controller, and which controller method should then handle it.

In the controller, the specific method handles the request, and returns data back to the request.

This could be as simple as a notice saying the request has been successful, or could be more complex by sending back a JSON with some processed data.

On the client side, when AJAX receives the response, the second part of AJAX callback can take place. And through JS or JQuery this could do something like updating or creating some HTML element on the page.

This chapter will now discuss this implementation in greater detail.

14. TEST RESULTS AND TEST REPORTS

14.1. Video Mode Unit Tests

```
---------------------------------------------------------------------
VideoTest: test_Should_save_a_video,_with_a_valid_video_file_attachment
---------------------------------------------------------------------
-------------------------------------------------------------
VideoTest: test_should_not_save_video_without_video_file_media
-------------------------------------------------------------
-----------------------------------------------------
VideoTest: test_Should_save_HyperTranscript_to_database
-----------------------------------------------------
-----------------------------------------------
VideoTest: test_should_not_save_video_without_title
-----------------------------------------------
-------------------------------------------------------------------------
VideoTest: test_Should_convert_srt_string_into_hyperTranscript_data_structure
-------------------------------------------------------------------------

Finished in 8.169125s, 0.6121 runs, 0.7345 assertions

5 runs, 6 assertions, 0 failures, 0 errors, 0 skips
```

15. CODE LISTING

15.1. parsing srt file into hypertranscript data structure

a method that takes the content of srt file as a string and outputs a ruby hash/json data hypertranscript structure.

```
require 'srt'
 # requires srt gem
 # parses srt string into hypertranscript ruby hash/json

  def convert_srt_to_word_accurate_hypertranscript(srt_string)
    file = SRT::File.parse(srt_string)
    srt_hash={}
    srt_hash['lines']={}
    line_number = 1
    file.lines.each do |line|
      srt_hash['lines']["#{line_number}"] ={}

      # array of words in the line
      words_in_a_line = line.text.join(" ").split(" ")

      puts "words_array_size: "+words_in_a_line.size.to_s

      number_of_words_in_line = words_in_a_line.size
      #duration of line time start - time end
      line_duration = line.end_time - line.start_time

      #the duration of the line divided by the number of words in the line
      # time increment
      average_word_duration  = line_duration / number_of_words_in_line

      # to calculate the number of letters in a sentence
      number_of_letters_in_a_sentence =0
      # we loop through the array of words, and add up the size of each word
      words_in_a_line.each do |word|
        number_of_letters_in_a_sentence += word.size
      end

      duration_for_each_letter = line_duration / number_of_letters_in_a_sentence

      word_start_time = 0
      word_counter =0
      srt_hash['lines']["#{line_number}"]=[]
      words_in_a_line.each do |word|
        # word duration is equal to number of letters
        word_duration = word.size * average_word_duration
        word_start_time = line.start_time + word_counter * average_word_duration # 1000 if you want it in
milliseconds?
        # or
        # word_time = average_word_duration line.start_time

        # word_start_time += word_duration + word_time
        word_end_time = word_start_time+word_duration
```

```
      corresponding_word = line.text.join(" ").split(" ")[word_counter]

      word_hash ={}
      word_hash['tc_in'] = word_start_time
      word_hash['tc_out'] = word_end_time
      word_hash['word']= corresponding_word
      srt_hash['lines']["#{line_number}"] << word_hash

      word_counter +=1
    end

    line_number +=1
  end
  return srt_hash

end
```

15.2. save_quote_with_video_snippet

Method used in video Quote controller for saving quote from ajax request.

```
def save_quote_with_video_snippet
  # params hash quoteDetails field parsed as json / ruby hash
  data =  (JSON.parse(params["quoteDetails"]))
  # retrieve relevant video from video id in params hash
  @video = Video.find(data["video_id"])
  # create quote associated with that video
  @quote = @video.quotes.new
  # give time code in, timecode out, duration attribute to quote, using value in params hash and converti
ng it to float
  @quote.tc_in = data["tc_in"].to_f
  @quote.tc_out = data["tc_out"].to_f
  @quote.duration = data["duration"].to_f
  # give time text and author attribute to quote, using value in params hash and converting it to string
  @quote.text = data["text"].to_s
  @quote.author = data["author"].to_s
  # saves the quote
  @quote.save

  # Exporting Video
  # naming video segment associated with quote, using id of quote
  @snippet_file_name = "snippet_#{@quote.id}.mp4"
  # defining temporary location to store cut video
  @path_to_file= Rails.root.join('public',"#{@snippet_file_name}").to_s
  # system call to cut video with ffmpeg. the video is saved in the public folder of the applciaiton
  `ffmpeg -i 'http://#{ENV['AMAZON_S3_BUCKET']}.s3.amazonaws.com/videos/media/#{@video.id}/#{
@video.media_file_name}' -ss #{@quote.tc_in} -t #{@quote.duration} -vcodec libx264 -acodec aac -asy
nc 1 -strict -2 #{@path_to_file}`
  # to use paperclip gem to save the video segment, snippet onto amazon s3 we have to open the vide
o segment we just cut into the public folder.
  # iterate through the file, and associate that with @quote.snippet.
  File.open(@path_to_file, "r") do |f|
  # @quote.snippet is a paperclip method to define the file in attachment.
      @quote.snippet = f
  end
```

```
    #save the quote
    @quote.save
    # associate the link of the quote video segment as uploaded onto amazon s3 through the paperclip g
em to the quote
    @quote.link = "http://#{ENV['AMAZON_S3_BUCKET']}.s3.amazonaws.com/videos/quotes/#{@quote.i
d}/#{@snippet_file_name}"
    #save the quote
    @quote.save
    #   delete file temporary_snippet.mp4
    `rm -rf #{@path_to_file}`
    # return the link of the video segment on s3 to the view through the ajax call back
    @temp1 =  @quote.link
    temp = {response: @temp1}
# format support.
    respond_to do |format|
      format.json { render json: temp, status: :ok} #
      format.js { render :nothing => true }
    end # end of respond to format
  end
```

15.3. spoken data ruby SDK

```
=begin
an attempt to make a SDK for the spoken data API.
spokenData API ruby SDK v2
@author : Pietro Passarelli
@date: 22/07/2015
@url:  http://spokendata.com/api

It provides an SDK for the spokendata API.
It does not implement all of the RESTful methods,
it implements
  - retrieving srt file of recording if ready **`retrieve_subtitles_srt(recording_id)`** if ready returns srt file s
tring, if not returns false boolean.
  -  send video by URL **`send_by_video_url(video_url)`** which returns the video uid,that you should sa
ve for later retrieval.
  - also implemented getting **list of recordings** (not in use in example)
  - **`get_recording_by_recording_id`**, returns a recording object  (also used as helper method)

you can test this as rails runner, save this file in the `lib` folder
and run from terminal, fom root of the project
    rails r lib/spokenDataAPI.rb

to use it in rails add it as a model `spokenDataAPI.rb`
=end

  require 'net/http'
  require 'open-uri'
  require 'json'
  require 'nokogiri'

  ######################
  class SpokenDataAPI  < ActiveRecord::Base
    belongs_to :video
```

```
  @@BASE_URL ="http://spokendata.com/api/"
  #Setup the endpoints
  @@ENDPOINTS = {}

  #recordingList  returns all user recordings --> this contains `status` , either 'processing' or 'done'
  @@ENDPOINTS['recordingList']="/recordingList"


  ######################## initializer
  def initialize(user_id, api_token)
    @@USER_ID = user_id.to_s
    @@API_TOKEN = api_token.to_s
  end

  ######################## helper methods
  # external libraries helper methods
  # open URLs
  def open(url)
    Net::HTTP.get(URI.parse(url))
  end

  # parses url / XML (as received by the API)
  # returns a ruby hash
  def parse_xml(url)
    Rails.logger.info "url in parse_xl #{url}"
    # IMPORTANT: there is an issue with the xml, the encoding returned by the API is written `utf8` inst
ead of `utf-8`. and that trips up the parser. enche the substitution
    result_string = open(url).gsub("utf8","utf-8")
    # Parse the xml with nokogiri
    nokogiri_xml_document = Nokogiri::XML(result_string)
    # transform the xml into a ruby hash using built in active support methods.
    result = Hash.from_xml(nokogiri_xml_document.to_s)
    # `data` tag encapsulate the rest of the keys /tags/
    return result['data']
  end

  ######################## API Query builder helper methods
  def get_base_url
    return @@BASE_URL
  end

  def get_api_key
    return @@API_TOKEN
  end

  def get_user_id
    return @@USER_ID
  end

  def get_base_api_request
    url = get_base_url + get_user_id+"/" + get_api_key
    return url
  end

  ######################## getter methods recordings list
  # def get_user
  #   url = get_base_api_request + @@ENDPOINTS['user']
```

```
  #   xml_Hash = parse_xml(open(url))
  #   return xml_Hash #['user'] #TODO to check if it's actual combination or not some nested ['var']['user
']
  # end

  def get_recordings_list
    url = get_base_api_request + @@ENDPOINTS['recordingList']
    result = parse_xml(url)
    return  result['recordings']['recording'] # TODO: fix this
  end


  # gets the recording based on id
  # the id is the one defined by spokendata
  # returns the whole recording object.
  def get_recording_by_recording_id(recording_id)
    url = get_base_api_request + @@ENDPOINTS['recordingList']
    Rails.logger.info  "url: #{url}"
    result = parse_xml(url)
    Rails.logger.info "result: #{result}"
    recordings =  result['recordings']['recording'] # TODO: fix this
    # compares `id`
    recordings.select do |k,v|
      if k['id'].to_i == recording_id.to_i
        result = k
      end
    end
    #retursn the recording
    return result
  end

  ######################## getter method recording status
  # helper method for get_recordings_status
  # takes in a recording object
  # returns true if "done"
  # returns false if "processing"
  def get_recording_status(recording)
    recording_status = recording['status']
    if recording_status =="done"
      return true
    elsif recording_status =="processing"
      return false
    else
      "There was an error assessing the status of the recording"
    end
  end

  # from recording `id` return boolean for status of the recording.
  # true if "done" false if "processing"
  def recording_processed?(recording_id)
    Rails.logger.info "recording_id: #{recording_id}"
    recording = get_recording_by_recording_id(recording_id)
    return get_recording_status(recording)
  end

  ######################## getter methods srt
  # helper method for retrieve_subtitles_srt
  def get_srt(recording_id)
```

```
    url = get_base_api_request.to_s + "/recording/#{recording_id.to_s}/subtitles.srt"
    return open(url)
  end
  # retrieves subitles srt
  # takes in recording_id
  # returns false if recording status is "processing".
  # returns string containing srt file if status is "done".
  def retrieve_subtitles_srt(recording_id)
    if recording_processed?(recording_id)
      return get_srt(recording_id)
    else
      false
    end

  end

  ######################## send video for captioning
  # takes in the location url of the video, for instance if you are using amazon S3 this is the full path, if u
sing youtube is just normal URL, also works with Vimeo.
  # returns the recording id, to be able to check status and retrieve captions subsequently, best to save
this in the db
  # language options are
  # from API documentation:
  #   RECORDING-URL - YouTube or any direct URL of a media file
  #   LANGUAGE - english | english-broadcastnews | english-test | russian | chinese-ma | spanish-us | c
zech | czech-medicine | czech-broadcastnews | slovak
  #   ANNOTATOR-ID = id of assigned annotator (leave empty if no annotator)
  # if you are working with languages other then english you could modify params of this url to change l
anguage option.
  def send_by_video_url(url)
    request_url = get_base_api_request.to_s + "/recording/add?url=#{url}&language=english"
    response = parse_xml(request_url)
    # example response from api #{"message"=>"This media URL and language have already been ent
ered.", "recording"=>{"id"=>"5747"}}
    # return response['recording']['id']
    Rails.logger.info "response: #{response.inspect}"
    return response.inspect
  end

end # end of class SpokenDataAPI
######################

# ######################## How to use, example in 3 steps
# # 1. Create a spoken data api instance initialising it with user_id and API key.
# # you'll find these in http://spokendata.com/api once you have logged in.
# # use enviroment variables to store user id and api key, and remember to put the file in `.gitignore`
#
# spokenDataAPIObject = SpokenDataAPI.new(ENV['SPOKEN_DATA_USER_ID'],ENV['SPOKEN_DA
TA_API_TOKEN'])
#
#
# # 2. send video by URL
# # sending a video by url will return you the spoken data api assigned video ID
# uid = spokenDataAPIObject.send_by_video_url("https://www.youtube.com/watch?v=u5CVsCnxyXg")
# puts "UID: "+uid
#
# # 3. retrieve captions
```

```
# # use recording id / uid to retrieve subtitles, when ready. (will return false if they are not ready)
# # (good idea to save uid in db for checking on processing of video captions)
# # returns srt file if ready, boolean false if not => you can `if response != false` etc..
# response= spokenDataAPIObject.retrieve_subtitles_srt(uid)
# puts "Response: "+response.to_s
# ######################
```

15.4. Deployment script

first_time_deploy.sh, deploy.sh, new_deploy.sh

15.4.1. deploy.sh

```
#!/bin/bash
#chmod +x deploy.sh to make script executable
echo "Launching Heroku deployment script"

# if app already been created using this to rename
#heroku apps:rename $appName

echo "set enviroment variables from application.yml"
figaro heroku:set -e production

echo "bundle install"

#bundle exec figaro install

bundle install

#echo "precompiling rails assets"
#rake assets:precompile

# git add and commit, optional
echo "git add -A"
git add -A

echo "git commit"
git commit -m"commit before pushing to heroku"

echo "pushing onto github"
git push

echo "git push heroku master"
git push heroku master

echo "migrating PG db on heroku"
```

```
heroku run rake db:migrate

echo "heroku ps:scale web=1 worker=1"
# The worker needs to be on 1 for ffmpeg to work.
heroku ps:scale web=1 worker=1


echo "opening deployed website"
heroku open

echo "heroku status to see if there's any issue with the system "
heroku status

herokuAppNameURL=`heroku info -s | grep web_url | cut -d= -f2`

echo "_____IMPORTANT_____
_____"
echo "add this URL to Google Console Redirect URIs: ${herokuAppNameURL}auth/
google_oauth2/callback"

echo "press any key to run heroku logs stream"

read anykey

echo "running live stream of heroku logs"
heroku logs -t
```

15.4.2. first_time_deploy.sh

```
#!/bin/bash
echo "Launching Heroku first time deployment script"


#echo "give a name to your app(no spaces):"

#read appName
#heroku login

#comment out after the first time of deployment, unless you want to create a new inst
ance
#echo "add this URL to Google Console Redirect URIs: http://${appName}.herokuap
p.com/auth/google_oauth2/callback"


#echo "running heroku create ${appName}"
```

```
#heroku create $appName

echo "running heroku create getquickquote"
heroku create  --buildpack https://github.com/ddollar/heroku-buildpack-multi
# or use line below with a name you'd like to give to your application, fyi getquickquot
e is already taken
#heroku create getquickquote --buildpack https://github.com/pietrop/heroku-buildpack
-multi

echo "running deployment script"
sh ./deploy.sh
```

15.4.3. new_deploy.sh

```
#!/bin/bash
echo "removing existen heroku remote"
git remote rm heroku

#heroku apps:destroy --app $app --confirm $app

sh ./first_time_deploy.sh
```

16. HYPERTRANSCRIPT

16.1.1. Hyper Audio Converter Algorith analyses and refactoring.

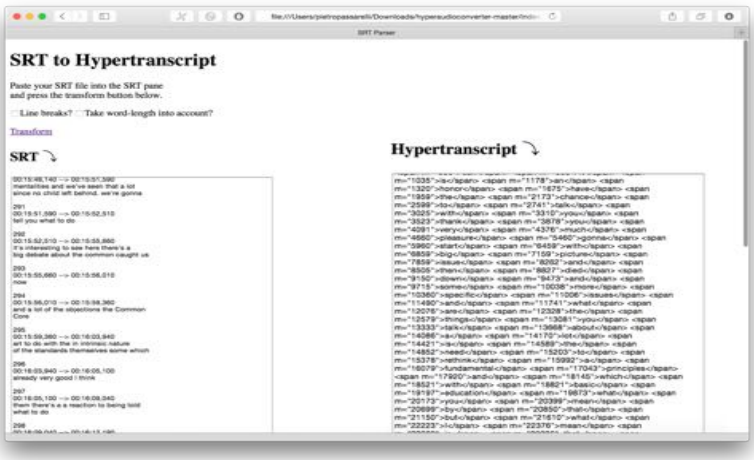
Analyses of Hyper Audio Converter javascript Algorithm (Mark Boas 2013b), and refactoring into ruby code.

Rationale: as JS is client facing code, re-factoring into ruby code, allows to package it as a ruby gem component, and to do the parsing in the model of rails MVC.

16.1.2. Hyper Audio Converter JS

Hyperaudio converter It is a one page application stored in index.html in JS using JQuery and PopcornJS libraries in js project folder.

The original code can be found at
https://github.com/maboa/hyperaudioconverter/blob/master/index.html



hyperaudio converter

Parser is a modified version of popcorn.parserSRT.js

16.1.3. Code Analyses

Let's look at this with a top down approach,

Button clicking

If we consider the last part of the code, it is adding the click funcioinality to the button element with the transform ID. Seen in the body of the index.html as Transform.

```
  $('#transform').click(function() {
    var srt = $('#subtitles').val();
    var ht = parseSRT(srt);
      $('#htranscript').val(ht);
    });
});
</script>
```

Then it grabs the value of the text are with id subtitles and stores it into a variable srt. In the GUI this is where you paste the content of your srt file. The same thing could have been done with a file upload.

```
var srt = $('#subtitles').val();
```

This can be found in index.html as <textarea id="subtitles" class="entry-content" rows="40" cols="60"></textarea>

The next 3 lines setup a regex regular expression to remove any new line carachters \n.

Now, this below is the line we are interested in as it is where the method parseSRT(srt); is called, which is ultimately what does the parsing of the subtitle file, and what we are interested in refactoring into Ruby.

```
var ht = parseSRT(srt);
```

This is where the result of the parsing, parseSRT(srt); method previously stored in the varial ht is added to the element with htranscript ID. using the .val(ht); method.

```
$('#htranscript').val(ht);
```

The .val(ht); method changes the value of a text field. In this case as mentioned above the element with ID htranscript, which corresponds to the HTML elment in index.html <textarea id="htranscript" class="entry-content" rows="40" cols="60"></textarea>.

The function parseSRT(srt);

Function Argument

First thing first this method takes in a data variable, which is a string, the content of an srt file.

Here is an example of what an srt file looks like:

```
1
00:00:00,680 --> 00:00:04,660
can it is an honor have the chance to
talk with you thank you very much

2
00:00:04,660 --> 00:00:07,859
pleasure gonna start with big picture
```

Return Statment

Then let's look at what it returns, it returns a outputString variable, which is defined as

```
outputString += '<span m="'+stime+'">'+stext+'</span> ';
```

We will look at how we got there in a second, but for now we can just consider that outputString contains the text of the srt file transformed into milliseconds word accurate span tags. Every word has the corresponding time starting point in the m attribute of the span tag.

function Variables

```
var i = 0;  //int
var len = 0;  //  int
var idx = 0;  //int
var line;   // array
var time;  //floats
var text;    // array
var sub;   //JS Object
```

function toSeconds

Now that we have a sense of what this function takes in as a argument parameter and returns, let's go to the rest of the code from the top.

The toSeconds funciton, does what it says on the tin. It takes in a timecode from .srt, something ijn the form of 00:00:04,660 and converts it into seconds with fractional milliseconds

As explained in the comments of the code:

```
//Simple function to convert HH:MM:SS,MMM or HH:MM:SS.MMM to SS.MMM
```

Split on line breaks

```
// Here is where the magic happens
// Split on line breaks
```

```
lines = data.split( /(?:\r\n|\r|\n)/gm );
len = lines.length;
```

The lines variable splits the string containing the srt file on new line, pattern matching with a regex, taking into a account compatibility with windows carriage return \n\r

This returns an array of lines. The variable len tells you how many lines there are in the subtitle file, by using the .length method on the lines array.

loop

javascript for(i=0; i < len; i++) {

Loops over the srt lines array using len, the number of lines to define the upper limit of the loop.

The two previously declared variables sub and text are then initialised as follows.

```
sub = {};
text = [];
```

sub is a javascript object and text as an array.

The object sub is then given an attribute of .id

```
sub.id = parseInt( lines[i++], 10 );
```

the method `parseInt() - http://www.w3schools.com/jsref/jsref_parseint.asp

```
// Split on '-->' delimiter, trimming spaces as well
      time = lines[i++].split( /[\t ]*-->[\t ]*/ );

      sub.start = toSeconds( time[0] );
```

The variable time splits the srt timecode start and finish uses a regular expression to HH:MM:SS,MMM --> HH:MM:SS.MMM, isolating start time code and end timecode

sub.start is then associated with the first one.

however to calculate sub.end

```
// So as to trim positioning information from end
      idx = time[1].indexOf( " " );
        if ( idx !== -1) {
          time[1] = time[1].substr( 0, idx );
        }
sub.end = toSeconds( time[1] );
```

Marking it word accurate

1.get all words in array

```
var swords = sub.text.split(' ');
```

2.get duration of line

```
var sduration = sub.end - sub.start;
```

3. word time estimation

divided duration of line, for number of words in it.

```
var stimeStep = sduration/swords.length;
```

4. determine lenght of word

```
var swordLengths = [];
var swordTimes = [];
var totalLetters = 0;
```

While is less then the number of words. loop iterating through the words in the line counting the letters in the whole of the sentence, by counting length of each word, and adding it up

swordLengths is an array of the length of the words, guess would it would look like [6, 5, 3] where 6 would be the length of the first word, 5 of the second word in the line etc..

```
for (var si=0, sl=swords.length; si<sl; ++si) {
  totalLetters = totalLetters + swords[si].length;
  swordLengths[si] = swords[si].length;
}
```

letterTime is calculated by dividing the total number of letters in a line, by the duration of a line.

```
var letterTime = sduration / totalLetters;
var wordStart = 0;
```

swordLengths is an array of the length of the words, while less than the number of words (or for each word in the line)

The wordTime is the length of the word [swordLengths (corresponding array value)] * times the letterTime, which we saw before was duration of line(time start - time end of line) / total letters in the line.

Then this is put relative to the starting timecode of the time start of the line. I think he multiplied by 1000 because his hypertranscript takes milliseconds, becauses it uses jplayer. While if you use videojs then it would be in seconds.

```
for (var si=0, sl=swords.length; si<sl; ++si) {
        var wordTime = swordLengths[si]*letterTime;
        var stime;
        if (wordLengthSplit) {
          stime = Math.round((sub.start + si*stimeStep) * 1000);
        } else {
          stime = Math.round((wordStart + sub.start) * 1000);
        }
```

String interpolation to generate HTML

```
  wordStart = wordStart + wordTime;
  var stext = swords[si];
  outputString += '<span m="'+stime+'">'+stext+'</span>';
```

16.1.4. Refactoring in ruby

```
 def convert_srt_to_word_accurate_hypertranscript(srt_string)
  file = SRT::File.parse(srt_string)
  srt_hash=[]
  line_number = 1
  file.lines.each do |line|
    # array of words in the line
    words_in_a_line = line.text.join(" ").split(" ")

    puts "words_array_size: "+words_in_a_line.size.to_s

    number_of_words_in_line = words_in_a_line.size
    #duration of line time start - time end
    line_duration = line.end_time - line.start_time

    #the duration of the line divided by the number of words in the line
    # time increment
    average_word_duration  = line_duration / number_of_words_in_line

    # to calculate the number of letters in a sentence
    number_of_letters_in_a_sentence =0
    # we loop through the array of words, and add up the size of each word
    words_in_a_line.each do |word|
      number_of_letters_in_a_sentence += word.size
    end


    duration_for_each_letter = line_duration / number_of_letters_in_a_sentence
    word_start_time = 0
    word_counter =0
```

```
    one_line_array_or_words =[]
    words_in_a_line.each do |word|

      # word duration is equal to number of letters
      word_duration = word.size * average_word_duration
      word_start_time = line.start_time + word_counter * average_word_duration
      # word_time = average_word_duration line.start_time

      # word_start_time += word_duration + word_time
      word_end_time = word_start_time+word_duration
      corresponding_word = line.text.join(" ").split(" ")[word_counter]

      word_hash ={}
      word_hash['tc_in'] = word_start_time.to_f
      word_hash['tc_out'] = word_end_time.to_f
      word_hash['word']= corresponding_word.to_s

      one_line_array_or_words << word_hash


      word_counter +=1
    end
    srt_hash << one_line_array_or_words
    one_line_array_or_words =[]
    line_number +=1
  end
  return srt_hash

end
```

17. SELENIUM AUTOMATED TEST OF VIDEO UPLOAD.

17.1.1. Overview of how selenium works

firs you need to install running this command from terminal

$ gem install selenium-webdriver

if running within rails you can then add the gem to the gem file of the application.

gem 'selenium-webdriver', '~> 2.47.1'

basics

The following code shows how selenium was initialized, and basics of it's working.

```
require 'selenium-webdriver'
# a firefox driver is initialized
driver = Selenium::WebDriver.for :firefox
# it is set to navigate to the given url
driver.navigate.to "http://pietrosmagicdomain.com:3000"

...

# select an element by id to then click it, the video upload element in the nav bar in th
is case
new_video_uplaod = driver.find_element(:id => "newVideoUpload")
new_video_uplaod.click()
#some more code testing
...
# closing the Firefox instance
driver.quit
```

In other words, first the Selenium web-driver is required, then the web driver initializes a driver, which is an object that simulates a Firefox instantiate, as specified on that line

navigate.to is then used to open a URL.

HTML elements can be selected using the find_element method. Then when the element you select is a form field the method send_keys takes a string as an argument and is used to fill in the field.

.submit or .click() can then be used to submit the form.

driver.quit closes the Firefox instance when the script has terminated.

A Ruby Selenium script can be saved as a ruby .rb file such as selenium.rb. This can then run from terminal, by navigating to the folder that contains it and run it using following command.

$ **ruby** selenium.rb

17.1.2. Selenium Test

```
require 'test_helper'
# to run the test, from terminal in root of application:
# and make sure local MySql db is turned on for testing (from os x system preference
s)
# rake test test/models/video_test.rb

class VideoTest <  ActiveSupport::TestCase

  test "should not save video without title" do
    video = Video.new
    assert_not video.save, "Saved the video without a title"
  end

  test "should not save video without video file, media" do
    video = Video.new(title: "Test Video")
    assert_not video.save, "Saved the video without a video file"
  end

  test "Should save a video, with a valid video file attachment" do
    # Sample video attachment
    def sample_video_file(filename = "test_video.mp4")
      File.new("test/selenium/#{filename}")
    end
    @video = Video.new(title: "Test Video")
    @video.media = sample_video_file("test_video.mp4")
    assert @video.save, "Saved the video with a video file"
  end

  test "Should convert srt string into hyperTranscript data structure" do
    # Sample video attachment
    def sample_video_file(filename = "test_video.mp4")
      File.new("test/selenium/#{filename}")
    end
    # open sample srt file, and returns it's content as a String
    def sample_srt_file(filename = "Kyle_captions.srt")
      data = ""
      file = File.open("test/srt_test_file/#{filename}", "r")
      file.each_line do |line|
        data += line
      end
      return data
    end
    # regex made using http://www.regexr.com/
    # HyperTranscript data structure example
    # [[{"tc_in"=>0.049, "tc_out"=>1.1656666666666666, "word"=>"God"}, {"tc_in"=>0.421222222222222
2, "tc_out"=>3.399, "word"=>"maintain"}, {"tc_in"=>0.7934444444444445, "tc_out"=>1.16566666666666
66, "word"=>"I"}, {"tc_in"=>1.1656666666666666, "tc_out"=>2.2823333333333333, "word"=>"and"}, {"tc
```

```
_in"=>1.5378888888888889, "tc_out"=>3.026777777777778, "word"=>"Kyle"}, {"tc_in"=>1.9101111111
11111, "tc_out"=>3.399, "word"=>"show"}, {"tc_in"=>2.2823333333333333, "tc_out"=>3.771222222222
2223, "word"=>"from"}, {"tc_in"=>2.6545555555555556, "tc_out"=>6.004555555555555, "word"=>"Lewi
stown"}, {"tc_in"=>3.026777777777778, "tc_out"=>5.632333333333333, "word"=>"Montana"}], [{"tc_in"=
>3.399, "tc_out"=>6.5, "word"=>"atomic"}, {"tc_in"=>3.9158333333333335, "tc_out"=>7.5336666666666
67, "word"=>"Western"}, {"tc_in"=>4.432666666666667, "tc_out"=>7.533666666666667, "word"=>"calle
d"}, {"tc_in"=>4.9495000000000005, "tc_out"=>8.567333333333334, "word"=>"Bochner"}, {"tc_in"=>5.4
66333333333333, "tc_out"=>8.0505, "word"=>"since"}, {"tc_in"=>5.983166666666667, "tc_out"=>8.050
500000000001, "word"=>"2008"}],...]
    hyperTranscript_regex = /(\[?(\[(\{(("\w+'?\/?-?\w+"=>(\d+\.\d+|"\w+'?\/?-?\w*"))[,\ ]*)+\}(,? ?))+\]?,?\ ?)\]
?)*/
    video = Video.new(title: "Test Video")
    video.media = sample_video_file("test_video.mp4")
    video.save
    assert_match hyperTranscript_regex, video.convert_srt_to_word_accurate_hypertranscript(sample_s
rt_file("Kyle_captions.srt")).to_s, "Matches HyperTranscript Regex"
    # assert_match /\[#{word}/ , video.convert_srt_to_word_accurate_hypertranscript(sample_srt_file("Kyl
e_captions.srt"))
  end

  test "Should save HyperTranscript to database" do
    # Sample video attachment
    def sample_video_file(filename = "test_video.mp4")
      File.new("test/selenium/#{filename}")
    end
    #sample srt file content String
    def sample_srt_file(filename = "Kyle_captions.srt")
      File.new("test/srt_test_file/#{filename}")
    end
    video = Video.new(title: "Test Video")
    video.media = sample_video_file("test_video.mp4")
    video.save
    hyperTranscript = video.convert_srt_to_word_accurate_hypertranscript(sample_srt_file("Kyle_caption
s.srt"))
    video.save_to_db(hyperTranscript)
    assert video.transcriptions.exists?
  end

end
```

18. USING MULTIPACK IN HEROKU DEPLOYMENT TO INSTALL FFMPEG

To deploy onto Heroku we need to make sure it has Ruby and FFMPEG installed on it, this can be done using heroku buildpacks. Normally you can only install one buildpack at a time, so to install more then one you need to use multibuildpack.

The file .buildpacks contains the buildpacks you want to install.

these buildpacks install Ruby and ffmpeg on the Heroku server
visit this url for more info https://github.com/shunjikonishi/heroku-buildpack-ffmpeg
https://github.com/brooks/heroku-buildpack-ffmpeg-x264
https://github.com/heroku/heroku-buildpack-ruby.git

Then running

$ **heroku** buildpacks:set https://github.com/ddollar/heroku-buildpack-multi.git

sets the buildpack from the .buildpacks file to your Heroku application.

this can also be done when creating the application using

$ **heroku** create getquickquote --buildpack https://github.com/pietrop/heroku-buildpack-multi

For more information on heroku buildpacks see
https://devcenter.heroku.com/articles/buildpacks

19. BIBLIOGRAPHY

Al Jazeera staff, 2013. Deconstructing Obama's States of the Union - Al Jazeera English. Available at: http://www.aljazeera.com/indepth/interactive/2013/02/201321213243145814.html [Accessed August 17, 2015].

AlchemyAPI Inc, 2015a. A sdk for AlchemyAPI using Ruby. Available at: https://github.com/AlchemyAPI/alchemyapi_ruby [Accessed August 25, 2015].

AlchemyAPI Inc, 2015b. AlchemyAPI Powering the New AI Economy. Available at: http://www.alchemyapi.com/ [Accessed August 24, 2015].

Alex Fletcher, 2014. The story of Cassetteboy: The kings of the YouTube funny cut-up video. *Digital Spy*. Available at: http://www.digitalspy.co.uk/displayarticle.php?id=615489 [Accessed August 17, 2015].

BBC R&D, 2011a. BBC Snippets. Available at: https://snippets.bbcredux.com/welcome/ [Accessed August 25, 2015].

BBC R&D, 2011b. Snippets - Projects - BBC R&D. Available at: http://www.bbc.co.uk/rd/projects/snippets [Accessed August 25, 2015].

Bigg, R. et al., 2015. *Rails 4 in Action* 2 edition., Manning Publications.

Brightcove Inc, 2010. Videojs. Available at: https://github.com/videojs/video.js [Accessed August 25, 2015].

cassetteboy, 2006. Cassetteboy YouTube Channel. Available at: https://www.youtube.com/user/cassetteboy [Accessed August 25, 2015].

CNN, 2015. Donald Trump's best lines during his 2016 speech - CNN Video. Available at: http://www.cnn.com/videos/tv/2015/06/16/donald-trump-presidential-announcement-supercut-tsr-vo.cnn [Accessed August 24, 2015].

Elliot Bentley, 2013. oTranscribe. Available at: http://otranscribe.com/ [Accessed August 25, 2015].

Elmaani LLC, 2009. Summarize Articles, Editorials and Essays Automatically. Available at: http://smmry.com/ [Accessed August 24, 2015].

Eric Ries, 2011. The Lean Startup: How Constant Innovation Creates Radically Successful Businesses: Amazon.co.uk: Eric Ries: 9780670921607: Books. Available at: http://www.amazon.co.uk/Lean-Startup-Innovation-Successful-Businesses/dp/0670921602/ref=sr_1_1?ie=UTF8&qid=1440440442&sr=8-1&keywords=lean+startup [Accessed August 24, 2015].

FFmpeg team, 2000. FFmpeg. Available at: https://www.ffmpeg.org/ [Accessed August 25, 2015].

Google YouTube, 2015a. YouTube API v2.0 – Captions. Available at: https://developers.google.com/youtube/2.0/developers_guide_protocol_captions [Accessed August 25, 2015].

Google YouTube, 2015b. YouTube data API Captions. Available at: https://developers.google.com/youtube/v3/docs/captions [Accessed August 25, 2015].

Happyworm LTD, 2009. jPlayer HTML5 Audio and Video for jQuery. Available at: http://jplayer.org/ [Accessed August 25, 2015].

HM Treasury and The Rt Hon George Osborne MP, 2015. Chancellor George Osborne's Summer Budget 2015 speech - Speeches - GOV.UK. Available at: https://www.gov.uk/government/speeches/chancellor-george-osbornes-summer-budget-2015-speech [Accessed August 25, 2015].

Igor Szoke, 2015. Your Speech-to-Text all in Cloud SpokenData. Available at: http://spokendata.com/ [Accessed August 25, 2015].

jQuery Team, 2006. jQuery. Available at: https://jquery.com/ [Accessed August 25, 2015].

Julik Tarkhanov, 2009. Timecode module. Available at: http://guerilla-di.org/timecode/ [Accessed August 25, 2015].

Kevin Sawicki, 2015. Atom Shell is now Electron. *Atom*. Available at: http://blog.atom.io/2015/04/23/electron.html [Accessed August 25, 2015].

Madalina Ciobanu, 2015. 5 innovative ideas for digital journalism from Build The News. Available at: https://www.journalism.co.uk/news/5-digital-storytelling-ideas-from-build-the-news/s2/a564659/ [Accessed August 25, 2015].

Marb Boas, 2013. US Election Debate Hyperaudio. Available at: https://github.com/maboa/uselect [Accessed August 24, 2015].

Mark Boas, 2014. Hyperaudio Hypertranscripts. Available at: http://hyperaud.io/blog/hypertranscripts [Accessed August 25, 2015].

Mark Boas, 2013a. Hyperaudio Pad. Available at: http://hyperaud.io/pad/ [Accessed August 17, 2015].

Mark Boas, 2013b. Hyperaudioconverter. Available at: https://github.com/maboa/hyperaudioconverter [Accessed August 17, 2015].

Mark Boas, 2011. Hyperaudiopad. Available at: https://github.com/maboa/hyperaudiopad [Accessed August 17, 2015].

Mark Boas, 2012. Interactive video of the Obama-Romney rematch - Al Jazeera English. Available at: http://www.aljazeera.com/indepth/interactive/2012/10/201210179222591398O.html [Accessed August 17, 2015].

Maurya, A., 2012. *Running Lean* 2 edition., Sebastopol, CA: O'Reilly Media.

Mozilla, 2010. Popcornjs. Available at: http://popcornjs.org/ [Accessed August 25, 2015].

Neil Flemming, 2015. Introduction to VARK VARK. Available at: http://vark-learn.com/introduction-to-vark/ [Accessed August 18, 2015].

NY Times, 2015. New York Times innovation report,

Passarelli, P., 2015. It's time to rethink how we do "x in quotes" pieces on the web: NY Times, BuzzFeed, The Guardian, BBC. *Medium*. Available at: https://medium.com/digital-times/it-s-time-to-rethink-how-we-do-x-in-quotes-pieces-on-the-web-1328f1ccf039 [Accessed August 24, 2015].

Passarelli Pietro, 2015a. autoEdit. Available at: http://www.autoedit.io/ [Accessed August 17, 2015].

Passarelli Pietro, 2015b. autoEdit COMPGC18 Entrepreneurship Theory and Practice,

Passarelli Pietro, 2015c. autoEdit, digital paper-editing - Tips Tricks & Quick Fix. Available at: http://pietropassarelli.com/autoEdit.html [Accessed August 25, 2015].

Passarelli Pietro, 2015d. Interactive databate #buildTheNews · Tips Tricks & Quick Fix. Available at: http://pietropassarelli.com/buildTheNews.html [Accessed August 25, 2015].

Passarelli Pietro, 2013. Rails Overview. *prezi.com*. Available at: https://prezi.com/bzrh_54z119l/rails-overview/ [Accessed August 25, 2015].

Passarelli Pietro, Striesow Axel & Start Sami, 2015. *GC02 - App Design Software Engineering Report*,

saaaam, 2014. What I can tell you. Available at: https://www.youtube.com/watch?v=D7pymdCU5NQ [Accessed August 25, 2015].

Sam Lavigne, 2014a. Audiogrep. Available at: https://github.com/antiboredom/audiogrep [Accessed August 25, 2015].

Sam Lavigne, 2014b. Videogrep. Available at: https://github.com/antiboredom/videogrep [Accessed August 25, 2015].

Sam Lavigne, 2014c. Videogrep Automatic Supercuts with Python. *Sam Lavigne Blog*. Available at: http://lav.io/2014/06/videogrep-automatic-supercuts-with-python/ [Accessed August 25, 2015].

The Sphinx group Carnegie Mellon University, 2015a. CMU Sphinx. Available at: http://cmusphinx.sourceforge.net/ [Accessed August 25, 2015].

The Sphinx group Carnegie Mellon University, 2015b. Pocketsphinx. Available at: https://github.com/cmusphinx/pocketsphinx [Accessed August 25, 2015].

Thomas Park, 2014. Bootswatch. Available at: https://github.com/thomaspark/bootswatch [Accessed August 25, 2015].

Thorsten Pehl, Thorsten Dresing, 2013. F4analyse - type, code, comment and analyze your interviews. Available at: https://www.audiotranskription.de/english/f4-analyse [Accessed August 25, 2015].

Thorsten Pehl, Thorsten Dresing, 2010. F4transkript - speeds up your transcription audiotranscription.de. Available at: https://www.audiotranskription.de/english/f4.htm [Accessed August 25, 2015].

thoughtbot, 2008. Paperclip. Available at: https://github.com/thoughtbot/paperclip [Accessed August 25, 2015].

Times Digital, 2013. Doctop. Available at: https://github.com/times/doctop [Accessed August 25, 2015].

Twitter, 2012. Bootstrap · The world's most popular mobile-first and responsive front-end framework. Available at: http://getbootstrap.com/ [Accessed August 25, 2015].

Winder, R., Roberts, G. & Winder, R., 2006. *Developing Java Software* 3rd Edition edition., Chichester, UK ; Hoboken, NJ: John Wiley & Sons.

Zulko, 2014. MoviePy. Available at: https://github.com/Zulko/moviepy [Accessed August 25, 2015].

Ændrew Rininsland, 2013. Obituary Nelson Mandela 1918-2013 The Times. Available at: http://thetim.es/lifeofmandela [Accessed Augusst 25, 2015].