BPRI2 – HANDS-FREE WEB INTERFACE

Project Report

Group Members:	Rareș Dan Pologea	student number: 208253
	Pavel Kočarian	student number: 208237
	Mihai Armand Enea	student number: 142529

Supervisors: VIA: Stephan Erbs Korsholm, Asbjørn Thalund Binderup

STIBO: Kim Svendsen

Table of Contents

Table of figures

Table of tables

List of keywords used throughout the project

Here you will be able to find a complete list with all the key words used in this report and what they mean, ordered alphabetically. The key words will be found within the content of the report by being italic.

Alexa – An intelligent personal assistant developed by Amazon. The Amazon Echo ships with Alexa preinstalled.

Alexa Skills – see Skills.

Amazon Echo – Smart speaker developed by Amazon. It is further discussed below.

Amazon Echo API – A set of tools and protocols used for building *Skills* for the Amazon Echo.

Amazon.com – An American electronic commerce and cloud computing company, the manufacturer of the Amazon Echo. From now on, it will be referred to as "Amazon".

API – A set of protocols definitions and tools meant for building software applications. It can be described as a way of communication between various software components.

CSS – Cascading Style Sheets, is a style sheet language used for describing the presentation of a document written in a markup language, in this case HTML.

Git - a version control system for tracking changes in files and coordinating work among multiple people.

GitHub - GitHub is web-based version control repository based heavily on Git.

HTML – HyperText Markup Language, is the standard markup language for creating web pages and web applications. Along with CSS and JavaScript, HTML is one of the cornerstone technologies for the World Wide Web (WWW).

HTTP – A communication protocol for information systems. It is the foundation of data communication for the World Wide Web.

HTTPS – A protocol for secure communication over a computer network which is widely used on the Internet.

JavaScript – A high-level, dynamic, untyped and interpreted programming language.

JQuery – A cross-platform JavaScript library designed to simplify the client-side scripting of HTML.

JSON – A format which uses human-readable text to transmit data objects consisting of attribute-value pairs.

Localhost – A hostname which means *this computer*. On most computers, *localhost* resolves to the IP address 127.0.0.1 for IPv4, or ::1 for IPv6.

Node.js – An open-source, cross-platform JavaScript runtime environment.

RaspberryPI – A single-board computer used for the server in our project. Further discussed below.

Skill – Term used by Amazon to describe any of the Amazon Echo's functions. It is further discussed below.

Speech API – An API that allows developers to provide a web browser with speech recognition input and text to speech display output; also called *WebSpeech API*.

SSH – Secure File Transfer Protocol, is a network protocol that provides file access, file transfer and file management over any reliable data stream.

SSL Certificate – A type of public key certificate, an SSL Certificate is an electronic document used to prove the ownership of a public key.

STIBO – Stibo Systems, the company with whom the team has collaborated on the project.

STIBO Supervisor – Kim Svendsen. He is our supervisor from the STIBO Accelerator and one of the persons who guided us throughout the project. More details about STIBO, the STIBO Accelerator and the STIBO Supervisor can be found in the Process Report Chapter 2.

URL – Uniform Resource Locator, is a reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it. It is informally known as a web address.

VIA Supervisor – Asbjørn Thalund Binderup. He is our supervisor from VIA University and one of the persons who guided and helped us out throughout this project.

YouTube API – An API intended for developers who write applications that interact with YouTube, such as this project.

1. Introduction

1.1. Background Description

Oftentimes in their daily lives people find themselves wanting to have a device that can be controlled without a physical interface. One such situation would be when people are in their cars, driving, and they do not want to divert attention from the road. Another such situation is at home, when people often have their hands full when dealing with certain situations and do not want to interrupt what they are doing, go and search something on the internet. In such situations, a computational device with which one can interact hands-free is needed. Thus, the project aims to create a system which takes audio input from users and interprets it into commands that can be executed and whose outcome can be observed on a screen.

Here is a rich picture that describes the idea of the project and the direction of its workflow.

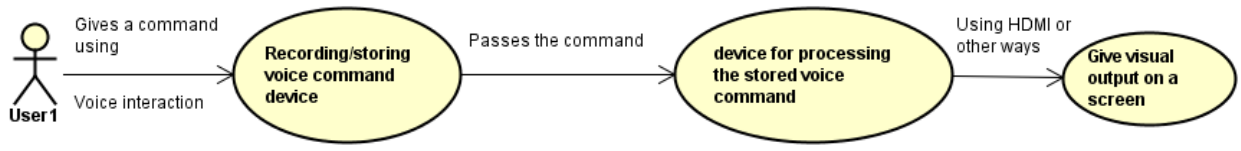


Figure 1. Project rich picture

1.2. Scenarios

House scenario – news

Imagine yourself at home, cooking in the kitchen. Everything is going along nicely when you glance at the clock on the wall and you realize it is 4 o'clock and the news are on. The first thing you think of is where is the remote, but then you realize that your hands are dirty and that the food is not done yet. If you go to the bathroom, wash your hands, then turn on the TV, you'll have to get your hands dirty again. Not to mention that food does not wait for the reporter to finish the story, which means that when the chicken comes out from the over, or when the soup needs to be stirred, or when the table need to be set you are not going to hear the story.

House scenario – music

Imagine yourself again at home. You are cooking in the kitchen, but this time it not 4 o'clock yet, so the meal preparations are in full swing. Vegetables are flying left and right, a pot of water is happily sitting on the stove, both the table and the counter are covered in plates, cutlery, and kitchen towels, but everything is in its place. Suddenly you remember that earlier, when you were driving home, they put your favorite song on the radio but you didn't get to listen to it to the end, so you want to listen now. The laptop is in the living room, surely you could just search

for the song on YouTube, but now your hands are full, literally, and there's no sign you would have enough time anytime soon.

Car scenario – music

This time imagine yourself driving. You're coming home from work. It's a 30 minutes' commute, and while the road conditions are not particularly challenging, a certain amount of attention is needed to drive safely. Right about when you encounter a longer stretch of road you find yourself wanting to listen to some relaxing music. You even have a song in mind. The problem is that while the car has a touchscreen and a state-of-the-art entertainment system, you can't afford to take your eyes off the road long enough to use any of them. You don't want to stop just for that either, as you're not far from home and you'd rather keep going than go through the process of finding a place to stop, finding the song, and then reentering traffic.

2. Analysis

Based on the Scenarios earlier described the team made an idea about how the project should be structured. The project is broadly described with the help of an Abstract Class Diagram pictured below:

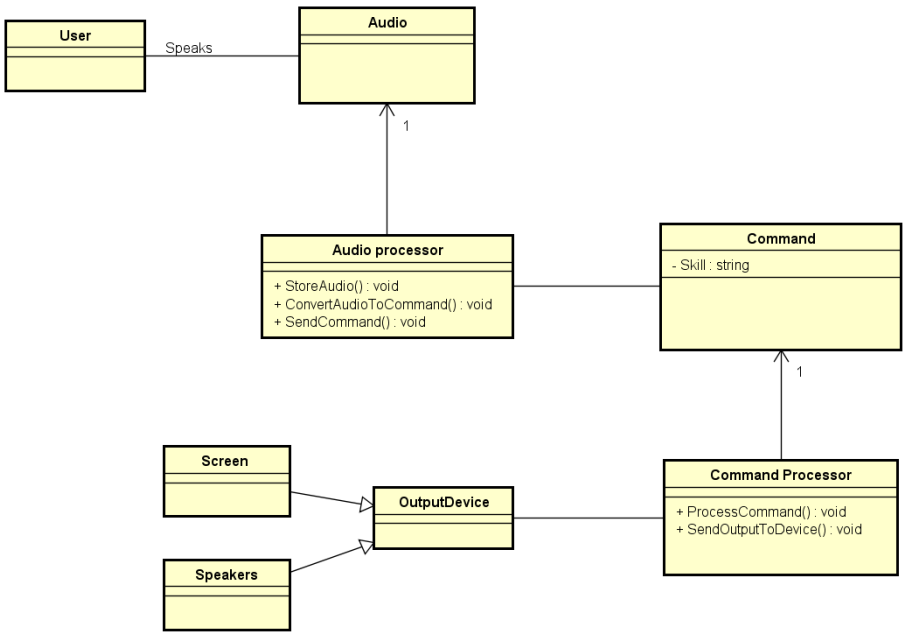


Figure 2. Project abstract class diagram

From this Abstract Class diagram the team could figure out what technical details, data manipulation, processing and calculations are needed and then how to structure the core system. More of this can be found in the next subchapter which will cover both functional and non-functional requirements as system delimitations.

2.1. Requirements

These are the requirements which have been chosen after deliberating with the team, *STIBO Supervisor* and *VIA Supervisor*. The functional requirements are meant to specify one or more particular results of the system. They are then supported by the non-functional requirements which impose constrains on the implementation and design of the system such as performance, security, or reliability. Both functional and non-functional requirements are explained in detail in their particular sections.

2.1.1. Functional requirements

A requirement that is crucial to the success of the project is the capability of the system to translate voice commands into text and then text into actual commands which need to be run. The finished system must be able to accept voice commands, process them and turn them into text, process the resulting text and turn it into commands, run the resulting commands and find the appropriate content, then displaying said content on a screen. Should the voice commands be interpreted into commands pertaining to a video output that is already being displayed (such as playback commands: “Pause”, “Play”, “Replay”, etc.), the respective commands’ results must be visible on the video content on the screen (the displayed video must be paused, playback must be resumed, the video must be replayed, etc.).

1. The system must convert human voice input into text.
2. The system must convert the text into a command.
3. The system must execute the command and display output.
4. The system must only interact with the user when the user specifically interacts with the system.
5. The system must have a Graphical User Interface.
6. The system must allow the users to search YouTube videos using voice commands.
7. The system must allow the users to have voice over media control over videos.
8. The system must allow the users to have voice over Web Page control.
9. The system must allow users to have multiple profiles.
10. The system must allow users to switch between profiles using voice commands.
11. The system must allow the users to change the Graphical User Interface.

2.1.2. Non-functional requirements

1. The system shall ensure an output regardless of the environment in which it is used.
2. The system shall correctly convert voice input into text in at most 0.5s.
3. The system shall correctly convert text into a command in at most 0.5s.
4. The system shall correctly convert voice input into text in at least 90% of the cases in a quiet environment.
5. The system shall correctly convert voice input into text in at least 40% of the cases in a loud environment.

6. The system shall correctly convert text into a command in at least 95% of the cases.
7. The system shall filter video output in relation to which user is using the system.
8. The system shall not check whether a user logs in their account and not someone else's.

2.1.3. Delimitations

1. The system will accept and interpret commands only in English.
2. The system's output will only be visible from a single web-page application.

2.2. Use Cases

To clarify and organize the system requirements the team made up a set of possible sequences of interactions between the system and users in an environment and related to a particular goal. This was done using Use Case Diagrams and Descriptions.

The following image represents the full functionality of the system.

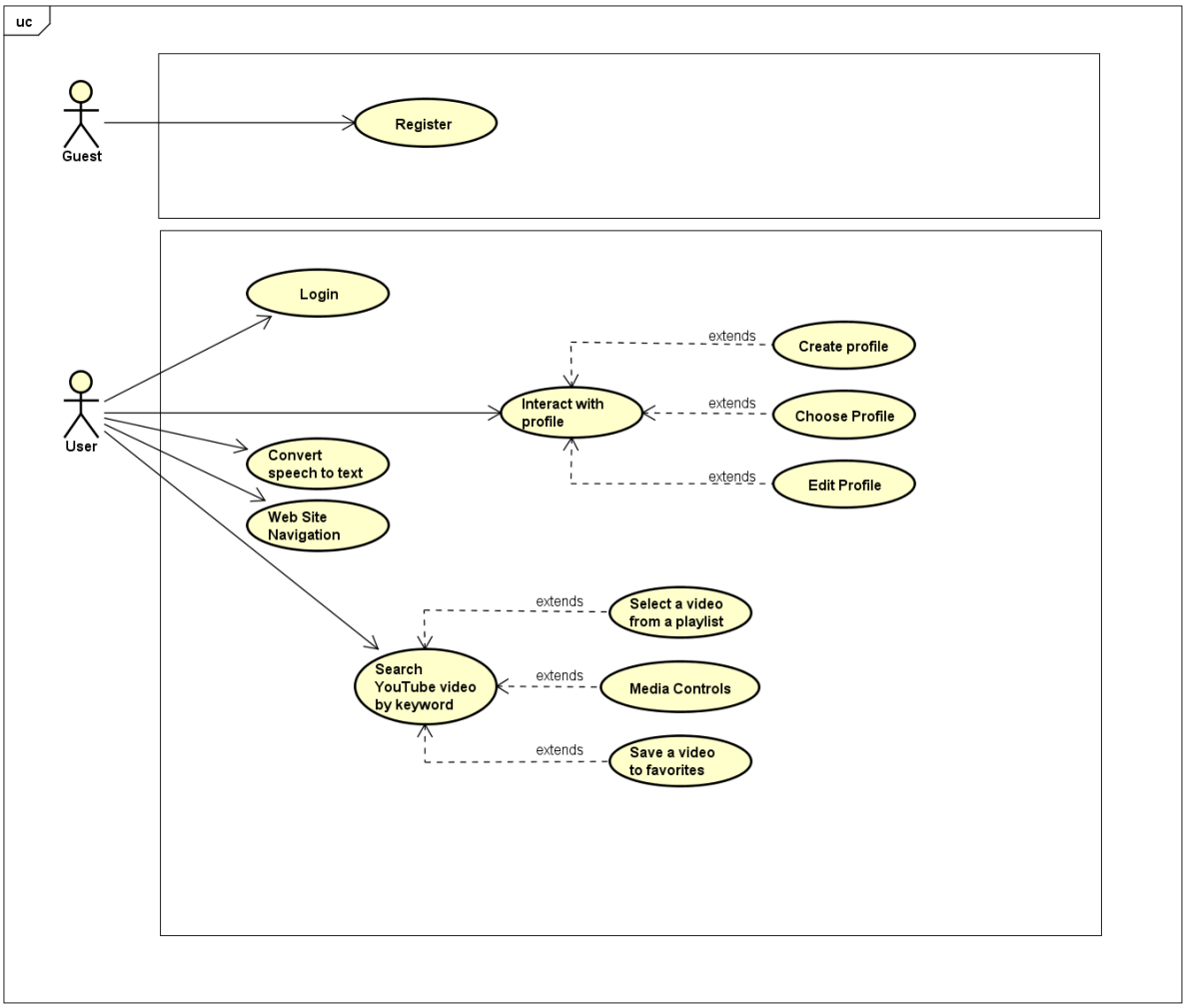


Figure 3. Full functionality project use case diagram

The Use Case Descriptions are presented in the following content, but only three of the most important ones will be shown. The first Use Case will concern the speech conversion to text and

then to a command that will display it on the webpage, and the second Use Case focuses on the usage of the speech conversion to give the system a command to see a YouTube Video. The final one is a Sub Use Case related with the second Use Case that will save a watched video to favorites. The other Use Cases as well as their descriptions can be found in Appendix D – Diagrams and Descriptions for those who would like to see more about them.

2.2.1. Convert speech to text

This Use case emerged from grouping the four of the functional requirements which are:

- The system must convert human voice input into text.
- The system must convert the text into a command.
- The system must execute the command and display output.
- The system must have a Graphical User Interface.

The only purpose of the GUI mentioned here is to give the user an interface where the user can see the spoken words. The GUI allows the user to see exactly what the system has understood from the vocal command, which is useful because the user can quickly figure out whether he has made himself well understood, or, in case he hasn't, the user can see that the system did not understand the command either because the words were mispronounced, the microphone is of low quality, or the surrounding environment is too loud, drowning out the commands.

The image below represents one part of the full system Use Case diagram.

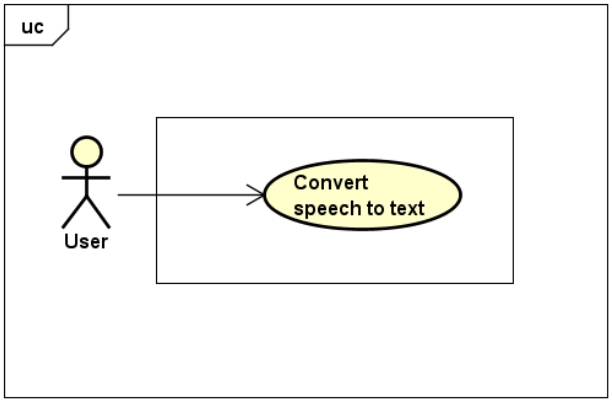


Figure 4. Convert speech to text use case diagram

Use case description

Use Case:	**Convert speech to text**
Summary:	This use-case describes a user speaking a phrase to the system, which will then be displayed as a text version of that phrase on Web Page which can be seen on a screen.
Actor:	**User**
Precondition(s):	The user must have a stable internet connection to the server. The user must have the system started.

	At the very first try of using the website, the user must allow the web site to use the microphone.
Post condition(s):	The system successfully converts the users spoken phrase into text. The system successfully converts the text into the Display Command. The system successfully displays the user's phrases on the Home Page.
Base sequence:	1. User opens web site on the home page. 2. User starts speaking. 3. The result will be displayed on the screen that displays the web page.
Branch Sequence:	
Exception Sequence:	**The command can fail to be processed by the server due to internet connection errors.** 1. The Web Page browser will display the appropriate error message.
Sub Use Case:	**None**
User Description:	A person who is using the system.

2.2.2. Search YouTube Videos by keywords

This Use Case emerged from the same four requirements used for the previous "Display Words on Web Page" but the result will be different from the previous one in which a user would only see the words displayed. The GUI has been improved to give the user a good feeling when browsing videos on YouTube and the new command phrases which will provide the necessary functionality have been made.

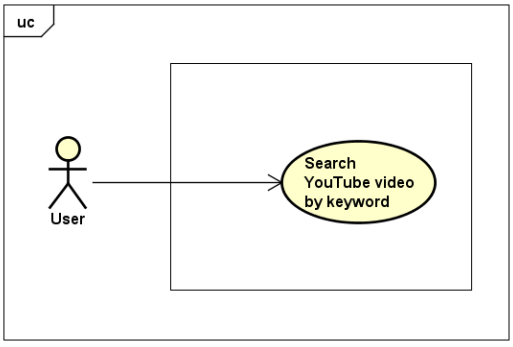


Figure 5. Search video by keyword use case diagram

Use case description

Use Case:	**Search YouTube Videos by keywords**
Summary:	This use-case describes a user asking the system to search for a video on YouTube by specific keywords.
Actor:	**User**
Precondition(s):	The user must have a stable internet connection to the server The user must have the system started. At the very first try of using the website, the user must allow the web site to use the microphone.
Post condition(s):	The system successfully converts the users spoken phrase into text. The system successfully converts the text into the YouTube Videos Command.

	The system successfully displays the user's YouTube Video and the keywords used for searching this Video.
Base sequence:	1. User pronounces Search on YouTube function activation command followed by the keywords meant for searching on YouTube. 2. The resulting video will be displayed on the screen with the help of the GUI. 3. The resulting video will play automatically. 4. The search bar will display the keywords used for this search.
Branch Sequence:	• Video does not exist: The video frame will display an appropriate error message. • Video cannot be reached: The video frame will display and appropriate error message.
Exception Sequence:	**The command can fail to be processed by the server due to internet connection errors.** 1. The Web Page browser will display the appropriate error
Sub Use Case:	**1. Select a video from playlist.** **2. Media controls.** **3. Save a video**
Actor Description:	A person who is using the system

2.2.3. Save a video to favorites

This next Sub Use Case emerged after completing the "**Search YouTube Videos by keywords"** Use Case and it extends its functionality. This sub use case will allow a user that is watching a video in The Video tab, to save that video by invoking a set of keywords.

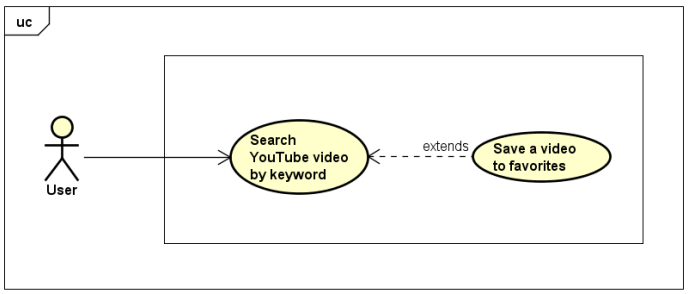


Figure 6. Save video to favorites use case diagram

Sub Use case description

Sub Use Case:	**Save a video to favorites**
Summary:	This use-case describes a user using voice commands to save a video to favorites tab.
Actor:	**User**
Precondition(s):	The user must have a stable internet connection to the server The user must have the system started. At the very first try of using the website, the user must allow the web site to use the microphone.
Post condition(s):	The system successfully converts the users spoken phrase into text.

	The system successfully converts the text into the commands used for saving a video to favorites. The system successfully saves the video into the Favorites tab.
Base sequence:	1. User navigates from the Home Page to the Videos Page. 2. User searches for a video on YouTube. 3. After the video is displayed successfully, the user invokes the keywords for saving the video to favorites. 4. The user can navigate the Favorites tab to see the video.
Branch Sequence:	
Exception Sequence:	**The command can fail to be processed by the server due to internet connection errors.** 1. The Web Page browser will display the appropriate error
Sub Use Case:	**None**
Actor Description:	A person who is using the system.

2.3. Activity Diagram

With the purpose of giving a more detailed diagram about how a specific Use Case acts from the starting point until it finished the process the team decided to show an Activity Diagram which will reflect the workflow of the second Use Case described in this report which is "**Search YouTube Videos by keywords"**.

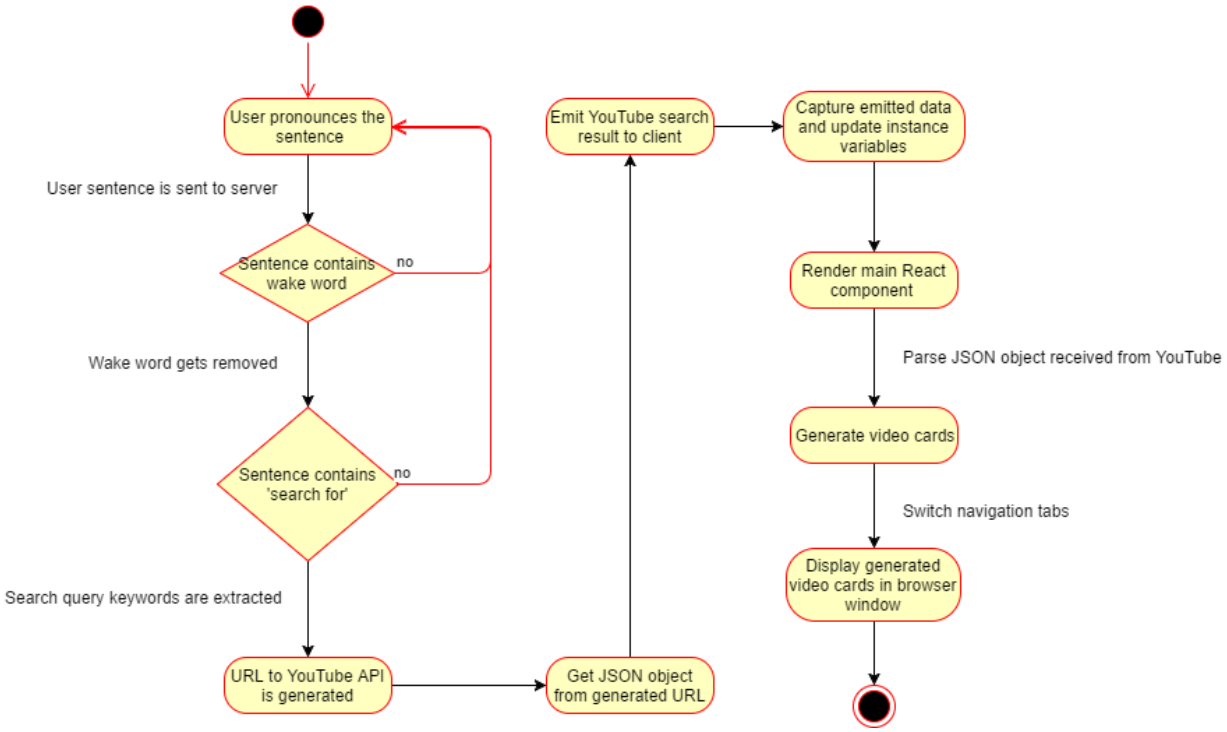


Figure 7. System activity diagram

3. Design

As one of the most important tasks of this project was to use a speech recognition technology, the team had to make choice considering all the possible choices which are out there today. Thus, the first part of the design will cover this particular subject.

3.1. Speech recognition

Speech recognition is the branch of computational linguistics which enables the recognition and translation of spoken language into text by computers using methodologies and technologies. Also, known as “automatic speech recognition”, “computer speech recognition” or “speech to text”, speech recognition is a vast field of study which incorporates knowledge and research in linguistics, computer science and electrical engineering.

Research in speech recognition has a history which spans decades and has had several waves of major innovations. The most recent wave of such type has been caused by advances in deep learning and big data. This wave of innovations based on deep learning and big data is evident not only because of the large number of academic papers written on the topic, but also by the wide industry adoption and deployment of speech recognition systems based on deep learning and big data. Companies like Google, Microsoft, IBM, Baidu, Apple, Amazon, Nuance, Sound Hound, and many others have made it public that the core technology in their speech recognition systems is based on deep learning.

However, speech recognition has not always been the way it is today. Much of the progress in the field is owed to the rapidly increasing capabilities of computers. In 1971, DARPA (the Defense Advanced Research Projects Agency) funded five years of speech recognition research through its Speech Understanding Program. At the end of the DARPA program in 1976, the best computer available to the researchers was the PDP-10, which featured, among others, 4 MB of RAM memory. Using such computers meant that it would take up to 100 minutes to decode as little as 30 seconds of speech. As years went by and computers became more capable, speech recognition researches began tackling harder problems such as larger vocabularies (one of the end goals of DARPA’s program was a minimum vocabulary size of 1.000 words; a native speaker adult’s vocabulary is 20.000-35.000 words), speaker independence, noisy environments, and conversational speech. Speaker independence, in particular, was a difficult obstacle to overcome. Early speech recognition programs were speaker dependent, which meant that the program first had to be “trained” on a certain speaker so that it would become accustomed to the speaker’s characteristics, such as accent, pronunciation, articulation, roughness, nasality, pitch, volume, and speed. Progress was made on speaker independence first by training on a larger variety of speakers and then by doing explicit speaker adaptation during decoding.

In 1992 DARPA held an evaluation of speech recognition systems. The system which had the best performance was Xuedong Huang’s Sphinx-II. The Sphinx-II system was the first to do speaker-independent, large vocabulary, continuous speech recognition. Huang went on to found the speech recognition group at Microsoft in 1993.

“The 1990s saw the first introduction of commercially successful speech recognition technologies. By this point, the vocabulary of the typical commercial speech recognition system

was larger than the average human vocabulary. In 2000, Lernout & Hauspie acquired Dragon Systems and was an industry leader until an accounting scandal brought an end to the company in 2001. The L&H speech technology was bought by Scan Soft which became Nuance in 2005. Apple originally licensed software from Nuance to provide speech recognition capability to its digital assistant Siri.

In the 2000s DARPA sponsored two speech recognition programs: Effective Affordable Reusable Speech-to-Text (EARS) in 2002 and Global Autonomous Language Exploitation (GALE). Four teams participated in the EARS program: IBM, a team led by BBN with LIMSI and Univ. of Pittsburgh, Cambridge University, and a team composed of ISCI, SRI and University of Washington. The GALE program focused on Arabic and Mandarin broadcast news speech. Google's first effort at speech recognition came in 2007 after hiring some researchers from Nuance. The first product was GOOG-411, a telephone based directory service. The recordings from GOOG-411 produced valuable data that helped Google improve their recognition systems. Google voice search is now supported in over 30 languages." [1]

3.1.1. Models, methods, and algorithms

Both acoustic modeling and language modeling are important parts of modern statistically-based speech recognition algorithms. Hidden Markov models (HMMs) are widely used in many systems.

Acoustic modelling

Acoustic Modeling is the process of taking a waveform of speech and analyzing it using statistical models. The most common method for this is Hidden Markov Modeling, which is used in pronunciation modeling to break speech down into component parts called *phones*. Microsoft has been a leading researcher in this field for many years.

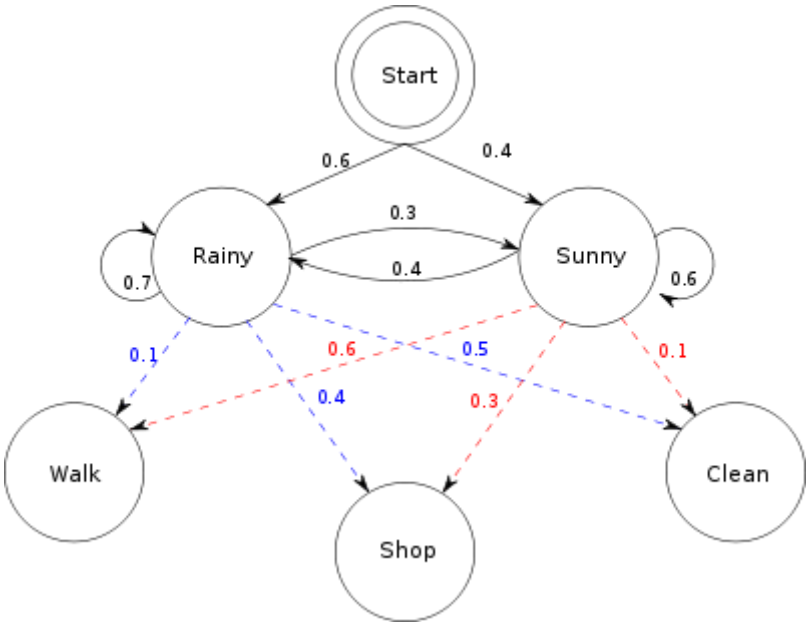
Hidden Markov models

Hidden Markov Modeling is a predictive mathematical model where the current state is determined by analyzing the output. The following example aims to clarify how Hidden Markov models work:

"Consider two friends, Alice and Bob, who live far apart from each other and who talk together daily over the telephone about what they did that day. Bob is only interested in three activities: walking in the park, shopping, and cleaning his apartment. The choice of what to do is determined exclusively by the weather on a given day. Alice has no definite information about the weather where Bob lives, but she knows general trends. Based on what Bob tells her he did each day, Alice tries to guess what the weather must have been like.

Alice believes that the weather operates as a discrete Markov chain. There are two states, "Rainy" and "Sunny", but she cannot observe them directly, that is, they are hidden from her. On each day, there is a certain chance that Bob will perform one of the following activities, depending on the weather: "walk", "shop", or "clean". Since Bob tells Alice about his activities, those are the observations. The entire system is that of a hidden Markov model (HMM).

Alice knows the general weather trends in the area, and what Bob likes to do on average. In other words, the parameters of the HMM are known." [2]



HMMs work by checking sounds against most probable sounds that come after them. When the pattern matches up correctly, the whole word is detected. For instance, after the sound "th", the system will check against sounds "e", "at", and so forth.

One of the main reasons why HMMs are popular is because they can be trained automatically and are simple and computationally feasible to use.

Modern speech recognition systems use various combinations of several standard techniques to improve results over the basic approach described above. A typical large-vocabulary system would need context dependency for the phonemes (so phonemes with different left and right context have different realizations as HMM states); it would use cepstral normalization to normalize for different speaker and recording conditions; for further speaker normalization, it might use vocal tract length normalization (VTLN) for male-female normalization and maximum likelihood linear regression (MLLR) for more general speaker adaptation.

Language modeling

While Acoustic Modelling is the starting point for speech recognition, it still does not account for several things, such as homonyms and regional variations in pronunciation. That is why Language Modeling is also used. Google has done a lot of research in this area, mainly through the use of N-gram modelling.

When a user interacts with a Google service which makes use of speech recognition, Google makes use of its massive bank of Voice Search and YouTube transcriptions. Google also uses information gathered through their GOOG-411 initiative in order to further improve their accuracy.

This language collection effort resulted in a vast array of pronunciations and dialects, which allowed Google to create a robust dictionary of words and how they sound. This allows for matches that have a greatly reduced error rate than brute force matching based on raw probabilities.

"While Google is a leader in this field, there are other mathematical models being developed, including continuous space models and positional language models, which are more advanced techniques born from research in artificial intelligence. These methods are based on replicating the sort of reasoning humans do when listening to each other. These are much more advanced both in terms of the tech behind them, but difficulty in mapping out these models is also far higher." [3]

N-gram modelling

In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sequence of text or speech. The items can be phonemes, syllables, letters, words, or base pairs per the application. The n-grams typically are collected from a text or speech corpus. When the items are words, n-grams may also be called shingles. N-gram Modeling works based on probabilities, but it creates a branching tree of possibilities using an existing dictionary of words. Thus, N-gram Modeling eliminates much of the uncertainty in the Hidden Markov Modeling.

As noted above, this method's strength comes from having a large dictionary of words and usage, not just primitive sounds. This gives the program the ability to tell the difference between homophones, like "beat" and "beet". N-gram modelling is also contextual, which means that it will know the difference between "beat" and "beet" from the sentence, being able to tell apart a conversation about sports from a conversation about vegetables, for instance.

Example:

Field	**Unit**	**Sample sequence**	**1-gram sequence**	**2-gram sequence**	**3-gram sequence**
Vernacular name			unigram	bigram	trigram
Computational linguistics	character	…to_be_or_not_to_be…	…, t, o, _, b, e, _, o, r, _, n, o, t, _, t, o, _, b, e, …	…, to, o_, _b, be, e_, _o, or, r_, _n, no, ot, t_, _t, to, o_, _b, be, …	…, to_, o_b, _be, be_, e_o, _or, or_, r_n, _no, not, ot_, t_t, _to, to_, o_b, _be, …
Computational linguistics	Word	… to be or not to be …	…, to, be, or, not, to, be, …	…, to be, be or, or not, not to, to be, …	…, to be or, be or not, or not to, not to be, …

3.1.2. Other methods and algorithms used in speech recognition

Dynamic time warping

“Dynamic time warping is an approach that was historically used for speech recognition but has now largely been displaced by the more successful HMM-based approach.

Dynamic time warping is an algorithm for measuring similarity between two sequences that may vary in time or speed. For instance, similarities in walking patterns would be detected, even if in one video the person was walking slowly and if in another he or she were walking more quickly, or even if there were accelerations and deceleration during the course of one observation. DTW has been applied to video, audio, and graphics – indeed, any data that can be turned into a linear representation can be analyzed with DTW.

A well-known application of DTW has been automatic speech recognition, where it is particularly useful when comparing speech recording with different speaking speeds. In general, it is a method that allows a computer to find an optimal match between two given sequences with certain restrictions. That is, the sequences are "warped" non-linearly to match each other. This sequence alignment method is often used in the context of hidden Markov models.” [1]

Neural networks

“Neural networks were devised as an acoustic modeling approach in ASR in the late 1980s. Since then, neural networks have found uses in several problems of speech recognition such as phoneme classification, isolated word recognition, and speaker adaptation.

Compared to HMMs, neural networks make no assumptions about feature statistical properties and have several qualities making them attractive recognition models for speech recognition. However, despite their effectiveness in classifying short-time units such as individual phones and isolated words, neural networks are rarely successful for continuous recognition tasks, largely because of their lack of ability to model temporal dependencies.

However, recently LSTM Recurrent Neural Networks (RNNs)and Time Delay Neural Networks(TDNN's) have been used which have been shown to be able to identify latent temporal dependencies and use this information to perform the task of speech recognition.” [1]

Deep Feedforward and Recurrent Neural Networks

A deep feedforward neural network (DNN) is an artificial neural network with numerous layers of units hidden between the input and output layers. Not unlike shallow neural networks, DNNs can model complex non-linear relationships. DNN architectures generate compositional models, where supplementary layers enable composition of features from lower layers, giving a huge learning capacity and thus the potential of modeling complex patterns of speech data.

It is in 2010 that DNNs have been proven successful in large vocabulary speech recognition by industrial researchers, in collaboration with academic researchers. In this case, large output layers of the DNN based on context dependent HMM states constructed by decision trees were adopted.

3.1.3. The necessity of a network connection

Probably the most notable examples today of speech recognition are the intelligent personal assistants offered by Microsoft(Cortana), Apple(Siri) and Amazon(Alexa). And anyone who has used Cortana, Siri or Alexa with a slow internet connection knows that it suddenly becomes a very frustrating experience. That is because the commands sent to Siri are sent over the network to be decoded by Apple, the commands sent to Cortana are sent to Microsoft to be decoded, and commands sent to Alexa are sent to Amazon to be decoded.

The reason these companies have decided to use off-site servers for decoding is that while the decoding could be done on mobile phones or tablets, the sheer amount of processing to be done would mean that delays might be introduced and the battery life of the respective mobile phones or tablets would plummet.

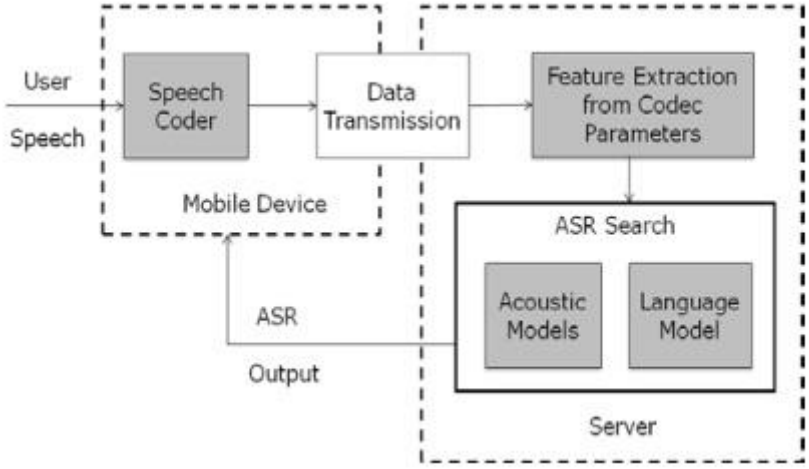


Figure 8. Speech recognition diagram

There are, however, desktop models, like Nuance, which use local resources due to the much more powerful hardware and reduced, if not nonexistent, power consumption limitations.

Despite this, offline speech recognition on mobile device is starting to become more widely spread, as Android now allows developers to include offline speech recognition in their apps. Other platforms are also expected to allow developers to make use of offline speech recognition as hardware becomes more powerful.

Being one the most important parts in the project, the team did research on the topic of speech recognition and concluded that implementing from scratch a solution that converts audio input into text is beyond the scope of the project considering the size of the team and the time available. Thus, the team decided to hardware or software components for the speech recognition. The hardware and software choices are explained in detail in Chapter 3.2.

3.2. Hardware and software choices

The idea behind all the technology which had to be used has been well defined from the beginning of the project at *STIBO* together with the *STIBO Supervisor*, but the market offered numerous hardware components and technologies which the team had to do research to be able to make the best choices. For keeping the research to a minimum a small list of what features should the hardware be able to support, the team made a small list of hardware requirements:

1. The hardware/software must be capable of voice service interaction.
2. The hardware must be capable of hosting a server.
3. The hardware must be capable of displaying video output.

3.2.1. Amazon Echo and Google Home

As devices for voice service interaction-capable hardware we had to choose between the *Amazon Echo* and Google Home.

Figure 9. Voice service interaction-capable hardware

The *Amazon Echo* is a smart speaker developed by *Amazon.com*. The device consists of a 9.25-inch (23.5 cm) tall cylinder speaker with a seven-piece microphone array. The device responds to the name "Alexa". This "wake word" can be changed by the user to either "Amazon" or "Echo".

The device is capable of voice interaction, music playback, making to-do lists, setting alarms, streaming podcasts, playing audiobooks, and providing weather, traffic, and other real time information. It can also control several smart devices using itself as a home automation hub [4].

There is currently only one more device on the market which is targeted to the same group of customers the *Echo* is, and performs the same tasks: Google Home. Google Home is a smart speaker developed by Google and supports commands and features similar to the *Amazon Echo*, like streaming through Google Play Music, YouTube Music, Spotify, TuneIn, Pandora, and

iHeartRadio. Google Assistant, Google's intelligent personal assistant, will be included as the main assistant in the operating system of Google Home.

Some of the advantages Google Home has on its side are that it can pull information from various Google accounts and apps, will answer follow-up questions, can communicate with the Google Chromecast, Google-powered speakers, and TVs out of the box, and it is better suited for use in multi-room environments. [5]

However, *Amazon Echo* has far wider product integration and features more than 3000 Skills (which are third-party services such as Philips Hue or Uber). The most important advantage *Amazon Echo* has over Google Home is that anyone who owns an Amazon Developer Account (free to create) can create and publish a Skill. This is crucial for developers as it allows anyone to build functionality on top of *Amazon's Alexa*. Another very important advantage *Amazon Echo* has over Google Home is that *Amazon* has made *Alexa* available to developers to use in their own creations. As such, the *Amazon* ecosystem is an obvious choice for companies looking to design their own hardware or for DIY-ers wanting to experiment.

Besides these two hardware components, there are also several software components made for human to machine voice interaction that the team researched. The ones which they took into consideration will be presented below, as well as the motivation and reasoning behind the team's choices.

3.2.2. The Cloud Speech API

This API is a BETA project created by Google which allows developers to do speech to text conversion by using powerful neural network models in an API capable of recognizing over 80 languages and variants.

It has various use cases from which some that could be of great use to this project such as:

a) Automatic Speech Recognition or ASR which is powered by deep learning neural networking made to support applications like voice search or speech transcription.

b) Word Hints allows speech recognition to be customized. This is done by providing keywords or phrases that are likely to be spoken in a specific context.

c) Streaming Recognition provides the user with the recognition results while the user is still speaking.

d) Global Vocabulary which has a great vocabulary of over 80 languages from which to recognize.

e) Inappropriate Content Filtering which comes in very useful in case of parental control or to avoid inappropriate language harassment. As this API is not yet complete this feature is only available in some languages.

f) Real-time or Pre-recorded Audio Support allows audio to be captured by an application's microphone or send from a pre-recorded audio file. It supports multiple audio encodings including FLAC, AMR, PCMU and Linear-16.

g) Noise Robustness takes care of extra noisy audio from different environments without the addition of extra noise cancellation.

h) Integrated API allows all audio files to be uploaded in a request or integrated with Google Cloud Storage.

Unfortunately, starting with August 2016, *Cloud Speech API* has switched to a paid system, the price being $0.006 per 15 seconds. There is, however, a 60-minute free trial. The monthly usage is capped at 1 million minutes per month. Also, an approval is needed to use Speech API on embedded devices such as cars, TV's appliances, or speakers. [6]

3.2.3. Web Speech API

This API is at its core a JavaScript API which allows developer to incorporate speech recognition and synthesis into web pages. With this, developers can script their own text to speech output and use speech recognition as input for forms, continuous dictation, and control. It also allows web pages to control activation, timing and to handle results.

This concludes the small presentation about this API as the team considered this to be the perfect replacement for the *Amazon Echo* approach, and further details will be presented later in Chapter 3.3.

3.2.4. Wit.ai

Wit.ai is a service that allows developers to implement natural language recognition into their apps. Built by Alexandre Lebrun, *wit.ai* pools together voice samples taken from its users (with their accord) with the hope that it will soon enough rival the depth and breadth of tools available to companies with much more resources, such as Google or Apple. Alexandre built *wit.ai* after his experience with his last company, VirtuOz, which developed speech recognition software for companies like AT&T.

The problem that Alexandre wants to address with wit.ai is that whenever a speech recognition software needs to be built, the developers usually need to start from scratch. Despite customers usually wanting the system to perform the same commands, the most important body of information for the system, the set of voice samples needed to be re-recorded. Thus, through *wit.ai* Alexandre hopes that companies will be able to share voice samples much in the way that developers usually share code through platforms like *GitHub*. And just like *GitHub*, *wit.ai* is also free to anyone willing to share their data. The actual voice recordings will not be shared, for privacy reasons and practicality. Companies not willing to share their voice samples, however, can pay a fee to use *wit.ai*. [7]

Figure 10. Wit.ai screenshot from demo video available of the website

Per the official website, the applications *wit.ai* is best suited for are Bots, Mobile apps, Home automation, Wearable devices and Robots. A peek inside the "Community" tab confirms that, and shows that Home automation and Bots are very popular among developers choosing to use wit.ai for their creations. [8]

Starting out with wit.ai revolves around three main parts: 1. Get or create a command, 2. Make a request, and 3. Create the app itself. Under the "Quick start" tab the process of creating an app based on wit.ai is further explained in a user-friendly manner so that previous experience with wit.ai is not required:

1. "Sign up with GitHub or Facebook."

In this step the developer must login through one of the two accounts supported, GitHub or Facebook. This is the only way the developer can access the Wit console, which is where the wit.ai-enabled app is configured and trained.

2. "Create an app"

This step is where the creation of the app start. For now, only some details needs to be specified, such as the name of the application, the language to which it will respond (defaults to English), and whether the app data will be open to the community or private. The default choice is "Open", but the developer can also choose "Private", in which case the data will only be accessible by the app creator and the developers which the app creator gives permission to. Here lies the main strength of *wit.ai*, as anyone who opts for the "Open" choice has access to the ever-growing bank of voice recordings without needing to invest the money of man-hours industry

leaders have done before them. And with wit.ai's increasing popularity, said voice recording bank will only become larger.

3. "Your first story"

This is where the developer teaches *wit.ai* by example, and each example conversation is called a Story. The first Story would be the most important use case that needs to be implemented. The example on the website helps with building a simple bot that gives weather forecasts.

4. "Testing the story in the Chat window"

The Story defined in the previous step is tested to check if it behaves correctly. If that is the case, the developer can go to the next step.

5. "Adding a branch to the story"

In this step, branches are added to the story so that the bot can respond to the same Story in different circumstances, such as asking for a weather forecast and forgetting to provide the location. In this case, the bot should simply ask the user for a location, then continue the Story. Adding branches to the story greatly increases the flexibility of the bot, as it gives it the possibility to answer a much wider variety of questions, and allows users to communicate with the bot in a more natural, less strict way.

6. "Implementing the business logic of the bot"

Actions executed by *wit.ai* are implemented and executed on the client side, not on *wit.ai*'s side, which means that the developers are not bound by any programming language or framework and can use any programming platform and execute code of their choice, call APIs, etc.

The example on the website shows how a Node.js client is initialized a function called getForecast is implemented, together with its functionality.

7. "Strengthening the Natural Language Understanding"

This step is necessary in teaching *wit.ai* that not everything that the user says is actual input or a command that needs to be executed. Since Stories rely on entities and context to predict the next action, this step is very useful because it teaches wit.ai raw natural language sentences and their meaning.

8. "Next steps"

The eighth and final step is integrating *wit.ai* to the application. This can be done using one of the *wit.ai* clients provided for Node.js, Python or Ruby, or by using the HTTP API.

Wit.ai recipes

Wit.ai can help parse a message into structured data (Understand) or predict the next action the bot should perform (Converse) using Recipes. *Wit.ai* comes preloaded with several recipes made to address common problems found in both Understand and Converse scenarios.

Under the Understand category some of the Recipes are extracting date, time and location or extracting a keyword entity. Under the Converse category we have extracting information from user messages, handling Yes/No answers and building a flow-based bot.

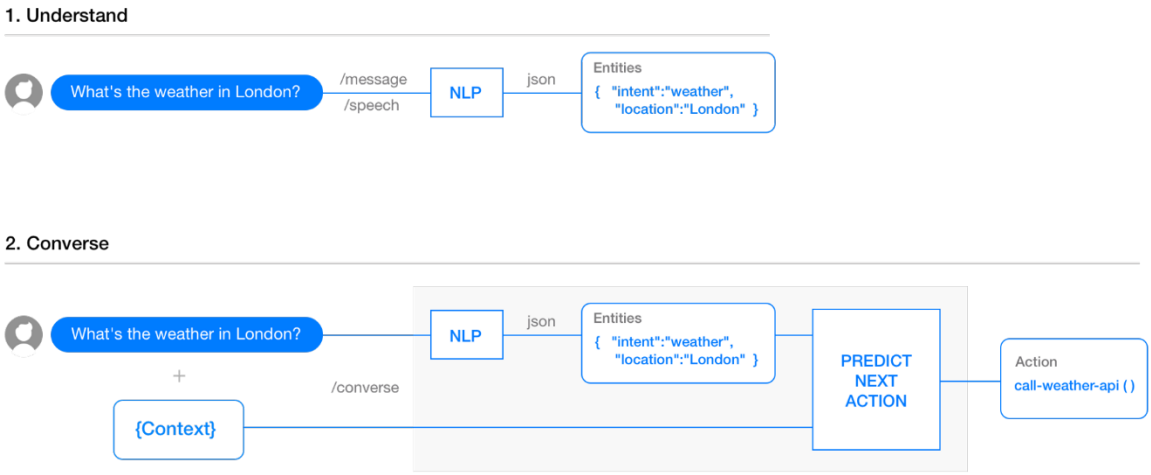


Figure 11. Overview of Understand and Converse

Understand

The meaning of user input is analyzed and understood using Natural Language Processing(NLP). NLP may be needed as a stand-alone layer to parse text or speech into structure data. NLP is also used in building a conversational app to understand a user query and extract useful information from it.

The most important problem to be solved when creating a conversational app is categorizing user input. The app must be capable of understand what the user means through saying “Ask about the weather”, “Book a restaurant” or “Open the garage door”. *Wit.ai* solves this problem by allowing the developer to enter a default expression for a certain command, such as “What is the weather?”, teaching *wit.ai* what needs to be extracted from the expression, and then adding additional expressions for the same intent (“How is the weather?”).

Converse

One of the most important problems that need to be solved when building conversational apps is building a flow-based bot. A flow-based bot’s questions need to consider previous answers and potential pathways so that users are not asked irrelevant questions and thus worsen the user experience.

Wit.ai solves this problem with the use of branches, but also jump and bookmark. These functions, paired with the ability to extract and store information from user messages, allows the bot to ask the user questions in a manner not unlike a conversation with a human. [9]

Conclusion

Overall, *wit.ai* is a very powerful tool when it comes to speech recognition, however the team has chosen not to use it since wit.ai is more suited towards Bots and Home automation, topics which differ from the theme of the project. Should a future version of the project be planned and further functions plan to be added, wit.ai may be once again considered.

3.2.5. Intel Compute Stick/RaspberryPI/BeagleBone Black

For hardware, capable of hosting a server we had to choose between the Intel Compute Stick, the *RaspberryPI* and the Beagle Bone Black.

Figure 12. Server hosting-capable hardware

The *RaspberryPI* is a series of credit card-sized single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and developing countries. The project makes use of the *Raspberry's* latest iteration, the Raspberry Pi 3 [10].

The *RaspberryPI* is by no means the only single-board computer on the market. One of the most well-known alternatives is the Beagle Bone Black, with which the team has previous experience. However, The Beagle Bone Black is unsuited for the project due to its lack of a standardized video output, such as the Raspberry's integrated HDMI-out, and its lack of a GPU [11]. Both the GPU and the video output are crucial features for this project. The *RaspberryPI* also has the advantage of being one of the most popular single-board computers in the world, and has a very large community. Thus, using the *RaspberryPI* as a coding base is a more logical choice since help can more easily be found and many more options are available when implementing functions since many more toolchains have been previously compiled for the *RaspberryPI* by the community.

Another option considered by the team is the Intel Compute Stick. The Intel Compute Stick is a single-board computer developed by Intel, and is designed to be smaller than other small-form-factor PCs, while keeping comparable performance. The Intel Compute Stick has been considered for the project due to its small physical footprint, out-of-the-box HDMI output, and the capability of running either Windows-based or Linux-based operating systems [12].

However, the Intel Compute Stick is both more expensive than the *RaspberryPI* and significantly more difficult to modify due to the different design (the RaspberryPI is expressly designed to be built on, while the Intel Compute Stick is a closed system from a hardware point of view).

Despite this, the Intel Compute Stick can still be used for the project and would probably perform better than the RaspberryPI due to its more capable hardware, but the team has chosen not to use it because of price limitations. The situation might be different in the future, should a second iteration of the project be considered.

The hardware necessary to display video output was chosen to be a generic HDMI screen. Thus, any screen with an HDMI connection is sufficient.

The items chosen for the system are illustrated below. More technical and detailed information about each hardware component can be found in Appendix B.

Figure 13. Final system choice

3.3. Web Speech API Approach

This design sub-chapter presents the project build using the Web Speech API from Google approach. This is the approach that provided the team with the perfect solution for their needs, but of course as it could be seen in the previous subchapter, there are a lot of technologies from which to choose from, all of them having their own advantages/disadvantages. This approach was found after the team reach several dead ends with some of these approaches. For those interested in more technical details, a full description is found in Appendix C – "Amazon Echo approach". For those interested in the process that took place during this time please check the Process Report Chapters 5.9, 6.11 and 6.12.

The image below represents the core system and shows how the components are connected to one another and the arrows dictate the workflow.

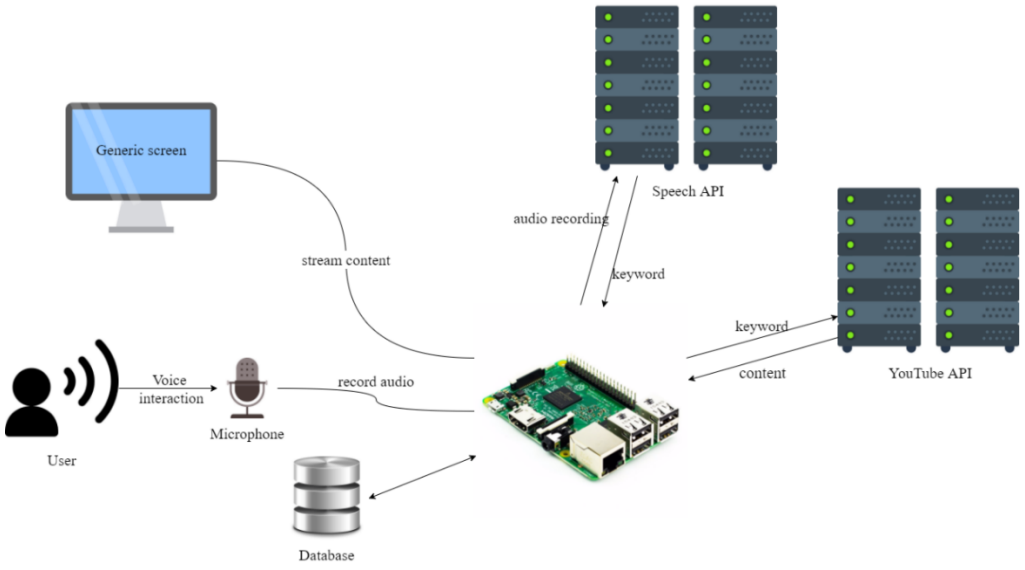


Figure 14. System diagram

By relating with the Abstract Class Diagram presented in the Analysis Chapter a decision for each part of the system has been made in terms of technologies, hardware, and the communication between these, to reach the goal of the project.

As it can be seen in Figure 13 (above), the candidate for handling Voice Interaction is Web Speech API, which converts the user's speech to text. The command will then be processed on the server which will then send execute the commands and post them on a Single Web Page Application. At that point, the *Raspberry PI* which has a connection to a screen trough HDMI, is responsible for streaming the output to the user.

3.3.1. System architecture

The system will run on a client/server architecture. To do this the team made use of multiple platforms of computing equipment by linking them together, thus allowing them to make use of each piece of hardware efficiently. Application logic and databases are developed on client workstations, as well as the servers.

There are several kinds of client/server models. The most popular out there are the 2-tier Client/server and the 3-tier Client/server models. Even though a n-tier Client/server model is possible, the complexity in architecture and the access time between top and low tiers discouraged such an approach, so the team decided to stick to the 3-tier model.

The 3-tier architecture is a second generation of client server architecture because it is an extended version of the traditional 2-tier architecture. The 3-tier architecture adds an application server as a middleware tier between the client and the database servers.

This middleware is a separate piece of software, which is typically running on a separate piece of hardware which can output high performance. It performs most of the application logic such as performing complex processing and accomplishes business logic.

Here is a comparison made between the 2-tier and 3-tier Client/server database systems which will further strengthen the team's choices.

In a 2 Tier Client/server model all clients get server access directly. The advantages of this model are that it is simple and easy to maintain.

On the other hand, in a 3-tier Client/server model the computing resources are distributed vertically. By doing this, it extends computing capability while still maintaining database integrity, traceability, consistency, and efficiency.

The evolution of application architecture is follows:

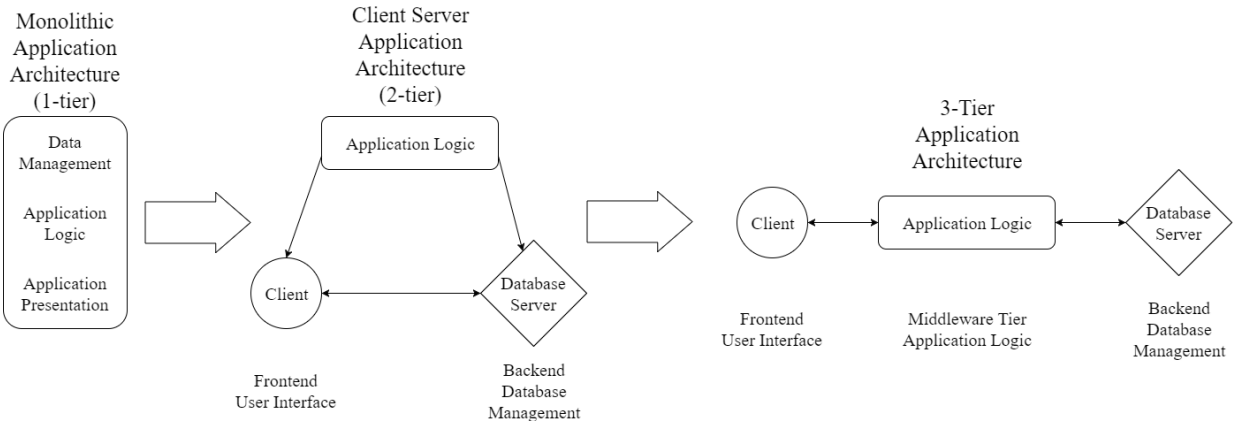


Figure 15. Application architecture evolution

When thinking of the ideal 3-tier model, the first thing in mind must be to partition the application logic completely into a middle machine. To be able to maintain and change each of these layers more easily the application design must separate the interface, business logic and data service.

The picture below shows how the 3-tier architecture separates its tiers:

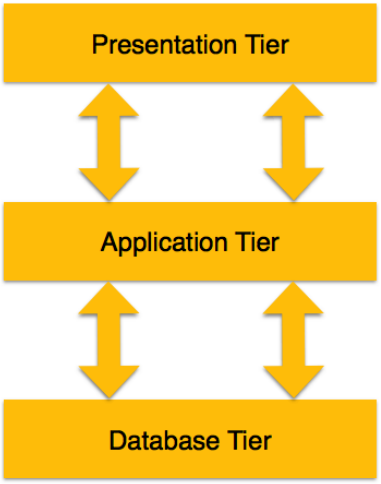


Figure 16. 3-tier architecture

Presentation Tier – In this tier end-users operate. Also, multiple views of the data can be provided by the application. All views are built by software that resides in the application tier.

Application Tier – In this tier lies the application server and database accessing programs. End-users are not aware of the existence of a database beyond the application. The database tier is also not aware of the other end; thus, the application layer stays in the middle and acts as a mediator between the other two.

Database Tier – In this tier the database is located along with the query language processing. In case of a relational database, the relations that define the data and constraints can also be found here.

Client/server processing techniques:

Data passing – Each tier contains a process that monitors incoming data continuously and triggers database updated if needed.

Remote execution – Invokes a program to update remote database

Interactive execution – the program that will update the database in a tier, will update those changes in other tiers simultaneously.

The system discussed in this document has the following technologies standing behind each tier:

1. ReactJS for the Presentation tier
2. ExpressJS for the Application tier
3. MongoDB for the Database tier

Class diagrams for both the server and the client side can be found on the next page.

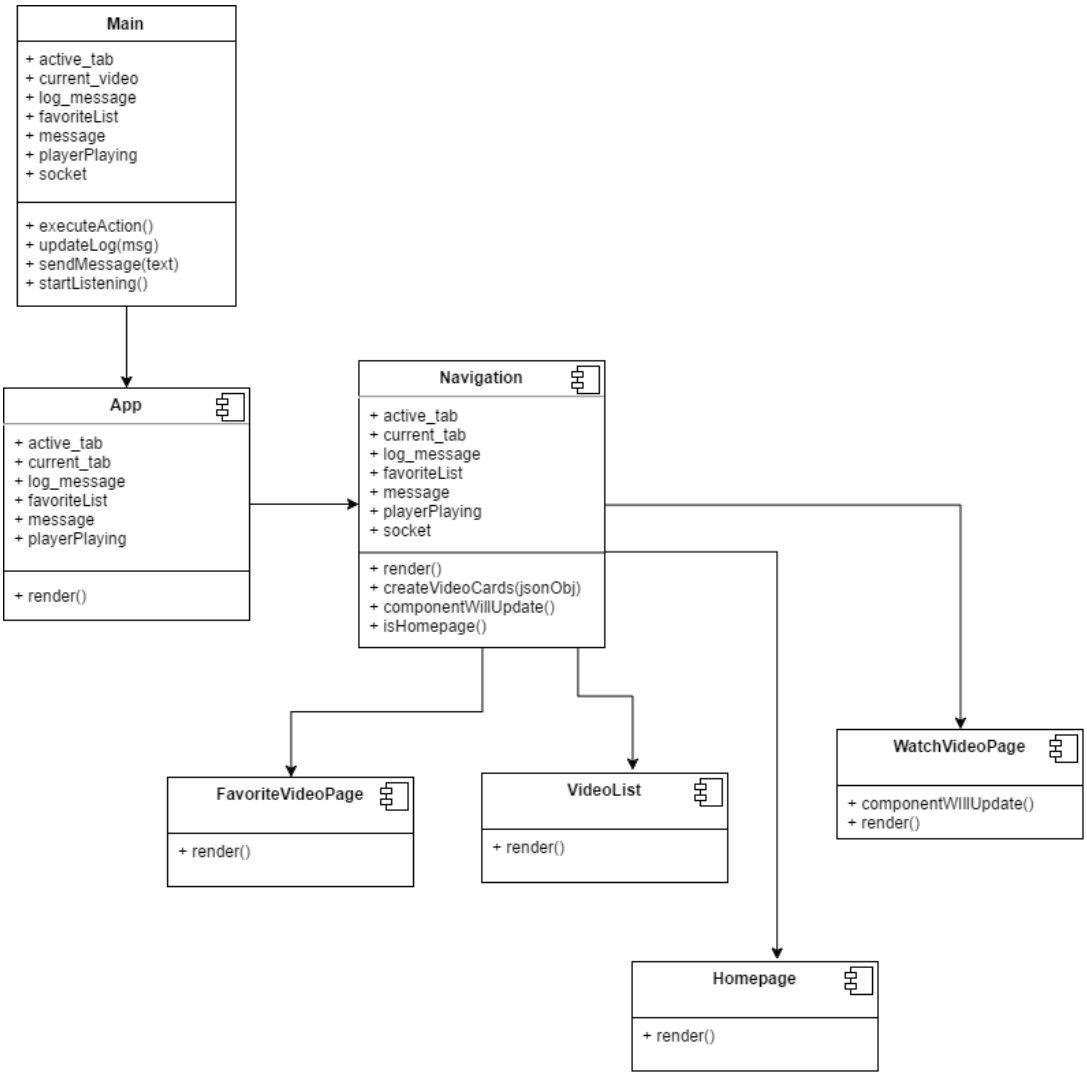


Figure 17. Client class diagram

Figure 18. Server class diagram

3.3.2. API's

An Application Programming Interface or API is defined in programming as a set of protocols definitions and tools meant for building software applications. It can be described as a way of communication between various software components.

In general, by providing the developer with all the building blocks which can be put together to create a program, the API should provide an easier way across.

There a few types of API's each build for their own purpose. These API's are:

- Web-based systems
- Operating systems
- Database systems
- Software library
- Computer hardware

Most often the API specifications are made of variables, object classes, data structures and routines. Normally the documentation for an API is given to facilitate usage.

The purpose of these application programming interfaces is to make it easier for developers to use certain technologies when creating applications. The API does this by covering the implementation and giving the developer only objects or actions which are needed, thus reducing the work load on the developer.

This project makes use of such API's and these two are explained below.

3.3.2.1. Web Speech API

After seeing in Chapter 3.2. why the Web Speech API was a good candidate for this project, more in depth knowledge that stands behind this API can be seen here, as well as some examples.

The aim of the Web Speech API is to allow developers to provide a web browser with speech recognition input and text-to-speech display output. These are features which are not typically available when using screen-reader software. It can support both server-based and client-based embedded recognition and synthesis.

The API is also designed to support both brief and continuous speech inputs.

When considering security and privacy there are a few rules which must be met:

- The browser must only start speech input session with explicit, informed user consent. The user consent can be for example:
  - The user clicking an element present on the webpage which has an icon that clearly signifies that it will start speech input.
  - The user granting consent that speech input will always be allowed for the respective webpage.

- The browser must give the user an obvious indication when audio is being recorded.
- The browser may also give the user an explanation when interacting with it for the very first time so that the user is aware of what the webpage is doing, what are its permissions and give the user the possibility to change the respective settings.
- To minimize the chances of users accidentally forgetting that they allowed web pages to record speech without their knowledge, implementations must abort an active speech input session in case the user lost focus of that web page.

There are two things which must be taken into consideration before implementing this API:

- The choice of a spoken password input can become problematic from some security perspectives, but it is up to the user to decide if they wish to speak their password.
- Speech input can be potentially used to eavesdrop on users. There are malicious webpages that could trick the user to believe that the speech recording has stopped, when it is still recording.

“Speech Recognition Attributes:

The Web Speech API comes with a set of attributes that allow the developer access to various options trough them such as:

- **lang** attribute will set the language of the recognition for the request, using a BCP 47 language tag.
- **continuous** attribute which can be set to false in which case the user agent must return no more than one final result in response to starting recognition. It can also be set to true, in which case the user agent must return zero or more final results representing multiple consecutive recognitions in response to starting recognition, like a dictation for example.
- **interimResults** attribute controls if interim results are returned. If set to true interim results should be returned, otherwise interim results must not be returned.” [13]

“Speech Recognition Methods:

Besides the attributes shown earlier the API also has a set of methods meant to help the developers use the Speech Recognition. These can be seen below:

- **start** method – when calling this method, the web application can begin recognition.
- **stop** method – this represent an instruction to the recognition to the recognition service to stop listening to more audio, and try to come up with a result using the audio received up until this point.
- **abort** method – this represents a request that will immediately stop listening and stop recognizing and do not return any other information except that the system is done recognizing.“ [14]

“Speech Recognition Events:

This API uses DOM Level 2 Event Model for speech recognition events. Here are some examples of these events:

- **audiostart** event- is fired when the user agent started the audio capture.
- **audioend** event- is fired when the user agent has finished capturing audio. The **audiostart** event must always be fired before **audioend**.
- **soundstart** event – is fired when sound, possibly speech is detected.
- **soundend** event – is fired when sound is no longer detected. The **soundstart** event must always be fired before **soundend**.
- **speechstart** event – is fired when the speech used for speech recognition has started.
- **speechend** event – is fired when the speech used for speech recognition has ended. The **speechstart** event must always be fired before **speechend**.
- **result** event- is fired when a result is return from the speech recognizer. The **audiostart** event must be fired before
- **start** event – is fired when audio is beginning to be recognized by the system.
- **end** event- is fired when the service is disconnected.
- **error** event – is fired when an error occurs during speech recognition.” [15]

“Speech Recognition Error:

The error event has an interface called the Speech Recognition Error event. It comes with numerous attributes which indicate what went wrong. It is especially useful when debugging and here is a list which will explain what each of them mean:

- **no-speech** – this occurs when no speech was detected.
- **aborted** – this occurs when the speech input has been aborted such as canceling the user speech input from the website.
- **audio-capture** – this occurs when the audio failed to be captured.
- **network** – this occurs in the case of a network communication failure.
- **not-allowed** – this occurs when the user agent is not allowing the speech input to initialize for security or privacy reasons.
- **service-not-allowed** – this occurs when the user agent does not allow the web application to request this speech service, but it could allow other services.
- **bad-grammar** – this occurs when the either there was a grammar or semantic tag error, or when the tags are unsupported.
- **language-not-supported** – this occurs when the choose language is not supported.” [16]

“Speech Recognition Result:

This represents an object of a single one-shot recognition match, either one small part of a continuous recognition or as the complete return result for non-continuous recognition.

It has the following two attributes:

- **length** attribute – represents how many best alternatives are in the item array
- **isFinal** attribute – if the boolean is set to true, then this will represent the final form that the speech recognition will return. If the values are false on the other hand, then this will represent an interim result that could still suffer further changes." [17]

"Speech Recognition List:

This object contains a sequence of results given by the speech recognition which represent the complete return set of a continuous recognition. In the case of a non-continuous recognition model, it will only hold a single value.

- **length** attribute – represents the number of results present in the item array.
- **Item getter** – this will return a result from the index into an array of result values. In case of the index being greater or equal to the length, then this will return a null value. It is the user agent's job to ensure that the length attribute is set to the number of elements in the array." [18]

This concludes the Web Speech API introduction. As an alternative for Speech API the team has considered ***wit.ai***, which is explained more in detail in Chapter 3.2.4:

3.3.2.2. YouTube API

This API is needed, regardless of the design approach and hardware/software choices, as using YouTube for the project's application requires developers to make use the it and as this project was intended for video or music streaming the team considered YouTube a very practical option, as it is very popular and almost any person with internet has used it at least once in their lives.

It comes with a very nice set of documentations, intended for developers who write applications that interact with YouTube such as this project.

It explains the basic concepts of YouTube, the API itself and an overview of the different functions that the API supports. Some of these will be presented here to give a better understanding of how the team made use this API.

Now, before anyone can start using the YouTube API there a few steps which need to be followed:

a) A developer needs to have or make a Google Account to access the Google Developers Console. Only then can he request an API key which will be used to register the application.

b) To submit API requests a project must be created in the Google Developers Console and obtain authorization credentials.

c) After the project has been created, a developer needs to specify in the application that it is registered to use YouTube Data API v3.

d) In case the application uses API methods that require user authentication, OAuth 2.0 authorization is the choice to go for.

Now that everything is set to use the API, it is time to look at the resources and resource types.

A resource can be described as an individual data entity with a unique identifier. In the table below there is a description for each different data type of resource that a developer can interact with using this API.

Resources

activity	Contains information about an action that a particular user has taken on the YouTube site. User actions that are reported in activity feed include rating a video, sharing a video, marking a video as a favorite.
channel	Contains information about a single YouTube channel.
channelBanner	Identifies the URL to use in order to set a newly uploaded image as the banner image for a channel.
channelSelection	Contains information about a set of videos that a channel has chosen to feature. For example, a section could feature a channel's latest uploads, most popular uploads, or videos from one or more playlists.
guideCategory	Identifies a category that YouTube associates with channels based on their content or other indicators, such as popularity. Guide categories seek to organize channels in a way that makes it easier for YouTube users to find the content which they are looking for.
i18nLanguage	Identifies an application language that the YouTube website supports. The application language can also be referred to as a UI language.
i18nRegion	Identifies a geographic area that a YouTube user can select as the preferred content region. The content region can also be referred to as a content locale.
playlist	Represents a single YouTube playlist. A playlist is a collection of videos that can be viewed sequentially and shared with other users.
playlistItem	Identifies a resource, such as a video, that is part of a playlist. The playlistItem resource also contains details that explain how the included resource is used in the playlist.
search result	Contains information about a YouTube video, channel, or playlist that matches the search parameters specified in an API request. Although a search result points to a uniquely identifiable resource, like a video, it does not have its own persistent data.

subscription	Contains information about a YouTube user subscription. A subscription notifies a user when new videos are added to a channel or when another user takes one of several actions on YouTube, such as uploading, rating or commenting a video.
thumbnail	Identifies thumbnail images associated with a resource.
video	Represents a single YouTube video
videoCategory	Identifies a category that has been or could be associated with uploaded videos.
watermark	Identifies an image that displays during playbacks of a specified channel's video.

The next table below represent a list of the most common methods that the API supports.

Methods

list	Retrieves (GET) a list of zero or more resources.
insert	Create (POST) a new resource.
update	Modifies (PUT) an existing resource to reflect data in the request.
delete	Removes (DELETE) a specific resource.

After seeing what resources can be used the next step is to have a look at the supported operations which developers can use.

The API currently supports methods to list each of the supported resource types, and the table below identifies the operations that are supported for different types of resources. The cells filled with a green OK represent the operations which can be used, and the ones marked with red X represent the operations which are denied.

Supported operations

	list	insert	update	delete
activity	OK	OK	X	X
channel	OK	X	X	X
channelBanner	X	OK	X	X
channelSelection	OK	OK	OK	OK
guideCategory	OK	X	X	X
i18nLanguage	OK	X	X	X
i18nRegion	OK	X	X	X
playlist	OK	OK	OK	OK
playlistItem	OK	OK	OK	OK
search result	OK	X	X	X
subscription	OK	X	X	X
thumbnail	X	X	X	X
video	OK	OK	OK	OK
videoCategory	OK	X	X	X

watermark	X	X	X	X

To make sure that all developers who use the YouTube Data API services do not create applications that unfairly decrees service quality or limit access for other users, Google made a quota counter for each operation. All the requests made with this API including the invalid ones, incur at least one-point quota cost. The quota available to each developer's application can be found in the Developers Console.

Projects such as this that enable the YouTube Data API will have a default of 1 million units per day, which Google considers sufficient for most API users.

To calculate quota usage Google attributes a cost to each request, which will not be the same for all requests as some can have a bigger data pull then others, or none such as an empty request.

There are two major factors that influence the quota cost for a request:

1. Different types of operations have different quota costs. Some examples can be seen below:

- A simple read operation that just retrieves the ID of the returned resource has a cost of approximately 1 unit.
- A write operation has a cost of approximately 50 units.
- A video upload has a cost of approximately 1600 units.

2. Read and write operations use different amounts of quotas depending on the number of resource parts that each request retrieves. An insert or update operations that write data and return a resource.

As an example, inserting a playlist has a quota cost of 50 units for the write operation plus the cost of the returned playlist resource.

Having these rules in mind a developer can try and estimate the number of read, write, or upload requests that the application will send per day, thus ensuring that it will not exceed the limitation of one million units per day.

As it can be seen, a total of one million units per day is actually sufficient for developing a simple application that uses the YouTube Data API. Also in case the quota limit is reached any developer can request additional quota in the Developers Console in the Quotas tab.

To ensure that the API uses network, CPU, and memory resources as efficiently as possible, it requires the retrieval of partial resources so that applications avoid transferring, parsing, and storing unnecessary data. Two request parameters allow a developer to identify the resource properties that the API response should include.

1. The "part" parameter which identifies groups of properties that should be returned for a resource.

The API requires the "part" parameter for any requests that retrieve or return resources. The parameter can identify one or more resource properties that should be included in the response.

Thus, using this parameter requires a developer to choose the resource components which the application will use.

The “video” resource for example has the following parts:

Snippet	contentDetails
fileDetails	player
processingDetails	recordingDetails
statistics	status
suggestions	topicDetails

These parts seen in the table above are objects that contains groups of metadata that the API server will or will not retrieve depending on the request or return which will be made.

These servers for multiple purposes such as:

- Allows API quota management. By increasing the number of parts retrieved by the API response, the API usage will increase accordingly thus decreasing the daily allocated quota amount.
- By not allowing the server to retrieve metadata fields which an application does not use will resolve in reduced latency and less stress on retrieves.
- By reducing the amount of data which a developer does not use in his application will result in reduced bandwidth usage. This could also work the other way around by retrieving all parts of data.

2. The “fields” parameter which filters the API response to only return specific properties within the requested resource parts.

By using the “fields” parameter a developer can filter the API response in a way that it will only include a specific set of fields. By doing this, nested properties from the API response can be removed thus reducing the bandwidth even more.

The rules for the syntax which are supported by the “fields” parameter can be seen below:

- **fields=a, b** – used to select multiple fields.
- **fields=*** - used as a wildcard to identify all fields.
- **fields=a(b, c)** – used to specify a group of nested properties that will be included in the API response.
- **fields=a/b** – used to identify a nested property.

Here is an example which shows three possible ways to retrieve a playlist’s item ID, title, and position for every item in the playlist:

- **fields=items/id, playlistItems/snippet/title, playlistItems/snippet/position**
- **fields=items(id, snippet/title, snippet/position)**

- **fields=items(id, snippet(title, position)**

For more information about how this project uses the YouTube API please check the Implementation chapter where the Use Case "*Search YouTube video by keyword*" will include this API.

3.1.3. MEAN Stack

The name MEAN refers to a collection of JavaScript based technologies used to develop web applications. These can be seen in the list below.

- Mongo DB: a powerful, agile, and scalable NoSQL database
- Express: a node.js web application framework which is flexible and provides a set of features to build single and multi-page web applications.
- AngularJS: an open-source front-end web application framework
- Node.js: an open-source JavaScript runtime environment made to develop applications and server tools.

By having all components of the MEAN stack support programs written in JavaScript, applications can have both the server-side and the client-side written in one language.

There are several variations which can alter the original MEAN stack by changing one or more of its components with other, mostly JavaScript frameworks.
One example is the MEEN stack which replace the front-end of Angular with Ember.js.
Even this project will not make use of the full MEAN stack as the front-end side given by Angular.JS will be replaced by React.JS.

1. React.JS Library

React is a library developed by Facebook which is used for front end. It handles the view layer for a web application and can also work with mobile apps.
React allows developers to create user interfaces in a declarative, efficient, and flexible way.

2. Node.js

Node.js is an open source, cross-platform runtime environment build on Google Chrome's JavaScript V8 Engine. Its purpose is to help develop server-side and network applications. These applications are written in JavaScript and can run on Microsoft Windows, Linux, and OS X. Node.js also comes with a rich library of various JavaScript modules which makes web application development easier using Noje.js to a bigger extent.

The reasons behind the team's choice into working with Node.js lie in its features:

- As mentioned earlier Node.js is built on Google Chrome's V8 JavaScript Engine which give the libraries the possibility to execute code faster.
- By simply outputting the data in chunks the Node.js applications never need to buffer data.

- Traditional servers create a limited number of threads to handle requests. Unlike them, Node.js uses a single threaded model and an event looping mechanism which helps the server respond in a non-blocking way. This gives the server high scalability.
- All API's which are part of the library are asynchronous.

To further back up this list of features showed above, comes a list of great known companies which developed projects and applications using Node.js. Here are only a few of them: Microsoft, eBay, Yahoo!, and PayPal.

What is V8?

It is Google's open source JavaScript engine. It has high performance and it is written in C++. It is used in Chrome, Node.js and other embedding applications. The V8 implements ECMAScripts and runs on Windows XP or higher versions, Linux systems that use IA-32, Mac OS X 10.5+. It can run as a standalone or can be embedded into C++ applications. It also compiles and executes JavaScript source code, manages object memory allocation and can garbage collect unnecessary objects which it no longer uses or needs.

Also Node.js has proven that to be best suited for applications such as:

- Single Web Page Applications
- JSON API Applications
- Data Streaming Applications
- Data Intensive Real-time Applications
- I/O bound Applications

As this project's interface is based on a Single Web Page Application, and using JavaScript the team found it by far the best choice.

The following diagram shows some important parts of Node.js which were of great use to this project.

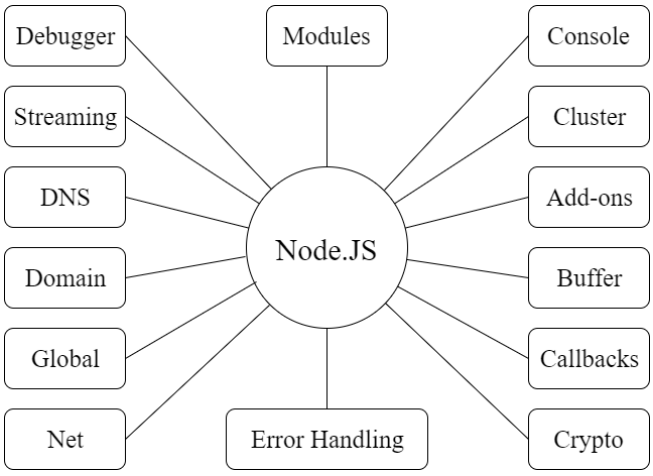


Figure 19. Node.JS features

3.Express.JS

Express is a web application framework that is both flexible and minimal, and provides a decent set of features to develop both web and mobile applications.

It has the following features:

- Allows to dynamically render HTML Pages
- Allows to middleware setup which respond to HTTP Requests
- Defines a routing table used to perform different actions based on HTTP Method and URL.

When using Express it should be installed along with the following modules:

- body-parser – a Node.js middleware which handles JSON, Raw, Text and URL encoded form data.
- cookie-parser – used to parse Cookie header and populate cookie requests with an object keyed by the cookie name.
- multer – a Node.js middleware for handling multipart data.

The Express.js application uses a callback function which has two parameters. These are:

- The request object which is the HTTP request. It has properties for the request query string, parameters, body, HTTP headers.
- The response object which represent the HTTP response that is send when an HTTP request is received.

By printing a request or response object a developer can find information such as cookie, session, URL.

4. MongoDB

When choosing a database, the team had to take into consideration a few aspects. They had to choose between a SQL database which deals with complex queries and transitions and builds relations between entities to ensure integrity of the data, and a NoSQL database which designed to excel in speed and volume but gains these advantages at the cost of losing relations.

By using Node.js, which can talk to and query almost any database, and after some research it was discovered that the NoSQL database, MongoDB, is best suited for a low-powered and low-resource device like the Raspberry PI which is used in this project.

MongoDB is an agile and scalable NoSQL database. Compared to the traditional columns and rows of relational database, MongoDB has document store model, in which data objects are stored as separate documents inside a collection. These documents are stored as binary JSON or BSON objects.

The motivation of using the MongoDB language stands behind the possibility to implement a data store that provides high performance, high availability, and automatic scaling.

MongoDB is simple to install and offers great website back-end storage for high-traffic websites that need to store data.

Here are some reasons why MongoDB was chosen for this project and why it is such a popular NoSQL database:

- MongoDB is one of the highest-performing database available.
- The MongoDB replication model give the possibility to maintain scalability while still being able to output high performance.
- MongoDB scales horizontally using sharding. The developer chooses a shard key, which determines how the data in a collection will be distributed. Afterwards, the data is split into ranges and distributed across multiple shards.
- MongoDB can run over multiple servers, balancing the load or duplicating data to keep the system up and running in case of hardware failure.
- MongoDB uses collections also known as capped collections. Using this type of collection, MongoDB maintains insertion order and, once the specified size has been reached, it will start acting like a circular queue.
- MongoDB is document oriented, thus eliminating the need for data to be transferred from rows to objects and back.
- JavaScript can and is used in queries, aggregation functions, and are sent directly to the database to be executed.
- Because objects are stored as objects and not by using SQL strings, MongoDB is made to be not susceptible to SQL injection.

As mentioned in the list above MongoDB groups all its data with the use of collections. This collection is a simple grouping of documents that have the same or a close similar purpose. A collection can be considered as a table in a traditional SQL database but with one major difference which is, that a collection is not forced to follow a strict schema. This reduces the need to break a document's items apart into more different tables, as the practice goes in SQL implementations.

The document represents a single entity of data in MongoDB. They can contain embedded subdocuments that provide the application with an inherent data model. These documents are stored as BSON, which is a lightweight binary form of a JSON. Here is a sample of a BSON document:

```
video document
{
  'fav_vid_id': 10,
  'videoTitle': 'Relaxing music',
  'url': 'https://www.youtube.com/watch?v=VnQ8AhkX7bQ',
  'videoDescription': 'Enjoy this Relaxing Music with Piano, Flute.',
  'image': [
            {'id': '30', thumbs: {'small': '15', 'medium': '35'},
            {'id': '37', thumbs: {'small': '45', 'medium': '55'}}

           ]
}
```

Figure 20. BSON object example

3.3.3. Socket.IO

Socket.IO is a JavaScript library which enables realtime, bi-directional communication between clients and servers. Socket.io consists of two parts: a client-side library that runs in the browser, and a server-side library for Node.js.

The “Docs” tab on the official website lists a chat application tutorial which is highly recommended for people who want to become familiar with Socket.IO. It also requires next to no prior knowledge of Node.js or Socket.IO, therefore making it highly accessible.

Despite the project not featuring a chat system, the team decided to use Socket.IO due to its features such as being event-driven, having support for auto-reconnect, having a simple and convenient API, it is fully cross-browser and it has a large community behind it.

3.3.4. Git

Git is a version control system for tracking changes in files and coordinating work among multiple people. The team has decided to use it since working on the project meant that the three team members would work on numerous files and had to make sure that everyone knows what everyone else does and everyone has the most recent version of everything.

Created by Linus Torvalds in 2005, Git replaced another source control management (SCM) system called BitKeeper, whose creator had withdrawn free use of the product after claiming that a developer had reverse-engineered the BitKeeper protocols. After launching the Linux kernel development release Linus was working on (2.6.12-rc2), Linus started working on his own system, since the other SCM systems available did not fulfill his requirements.

Features:

Branching and Merging

The most important feature that Git offers is its branching model. Git allows and encourages developers to have local branches that are completely independent of one another. This feature allows developers to:

1. Create a new branch just to try out a new idea, then switch back to the main branch, then either continue experimenting on the new branch or merge the two branches together.

2. Split the project into branches with well-defined purposes: one branch can have only code that goes into production, another branch can have only code that needs testing and several other branches can have code that the developers are actively working on, and that will be merged with one of the other branches at some point.

3. Create different branches for different features, which allows developers to work on features independently from other features or the main branch, and then merge the branch with the main product when it is ready to be shipped.

4. Create a branch solely for trying out an idea that might not make it into the final product, experimenting with the idea, and then abandoning the idea by deleting the branch entirely. This way, any idea can be tried out without modifying the product.

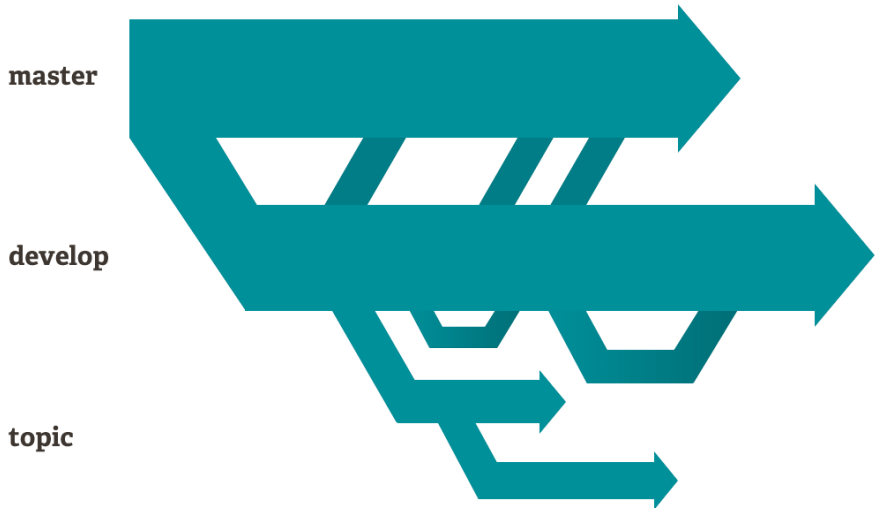


Figure 21. Git branching diagram example

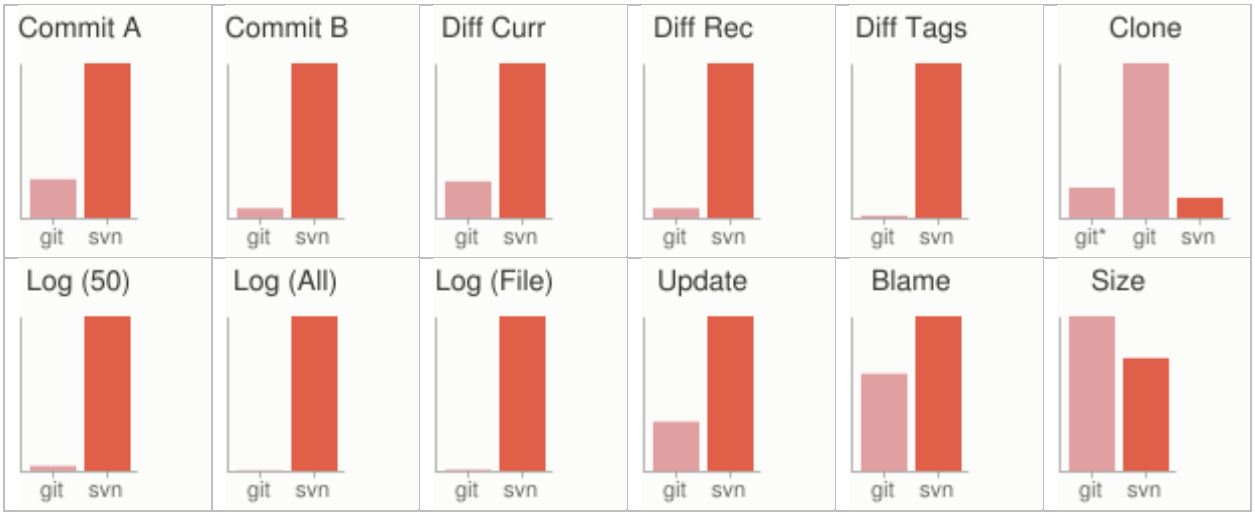
One thing to remember when using Git is that when the user pushes to a remote repository, it is not compulsory to push all the branches. The user always can choose whether to push only one of the branches, several of the branches or all the branches. Thus, the user can choose what he/she shares with whom and when.

Another advantage of Git is speed. Because nearly all the operations are performed locally, Git has an advantage over systems that need to communicate with a server before performing

changes. Furthermore, since Git was created to handle the large repositories involved in developing the Linux kernel, Git was designed to be fast and efficient from the very beginning.

The graphs below show the difference in speed between Git and another common SCM system called Subversion (smaller is faster):

Table 1. Speed differences between Git and Subversion



Note: The tests were done on two identical machines with a repository copied to both Git and Subversion servers. The same commands were executed on both machines, and when the commands didn't match up perfectly, the best-case scenario for Subversion was considered.

One of the advantages of Git that the team made use of in particular is Git's distributed nature. This means that at any given moment, every user whose local repository is up-to-date has a full-fledged copy of the whole system. Should the main server crash or be unavailable for some reason, any user can distribute a copy to another server or another user.

Git also offers a comprehensive commit timeline, making it easy to understand who committed what files and when, and on top of that, Git automatically calculates the SHA-1 checksum for every commit and displays it at the user's desire. Thus, it is very easy to check the integrity of the files and impossible to alter files without making it obvious.

One more feature that Git offers is the "staging area". Traditionally, in SCM, before pushing the changes to the server, the user must commit the changes. Git also offers the option to first add some changes or all the changes to the "staging area" and then commit only them. Thus, the user can choose to stage changes for this commit or for the next commit, which makes the whole process more flexible.

Lastly, another reason why Git is so popular among developers is that it's free. Git being released under the GNU General Public License version 2.0 makes it open source, and thus free for all its users.

3.3.5. GitHub

GitHub is a web-based version control repository based heavily on Git. GitHub offers all the distributed version control and source control management functionality of Git and further add on top of it access control and several other features such as bug tracking, feature requests and task management for every project.

The reason the team has decided to use GitHub alongside Git is because it offers free repository hosting, thus allowing the team members to always have a copy of the project on the internet. This allowed to team to continue working despite potential hardware problems, which proved especially useful, as one team member had to switch temporarily to a different laptop because of a charger failure. The development process was unaffected.

Another reason why the team decided to use GitHub is that GitHub offers a graphical user interface to the repository, making it very easy for team members to interact with the repository.

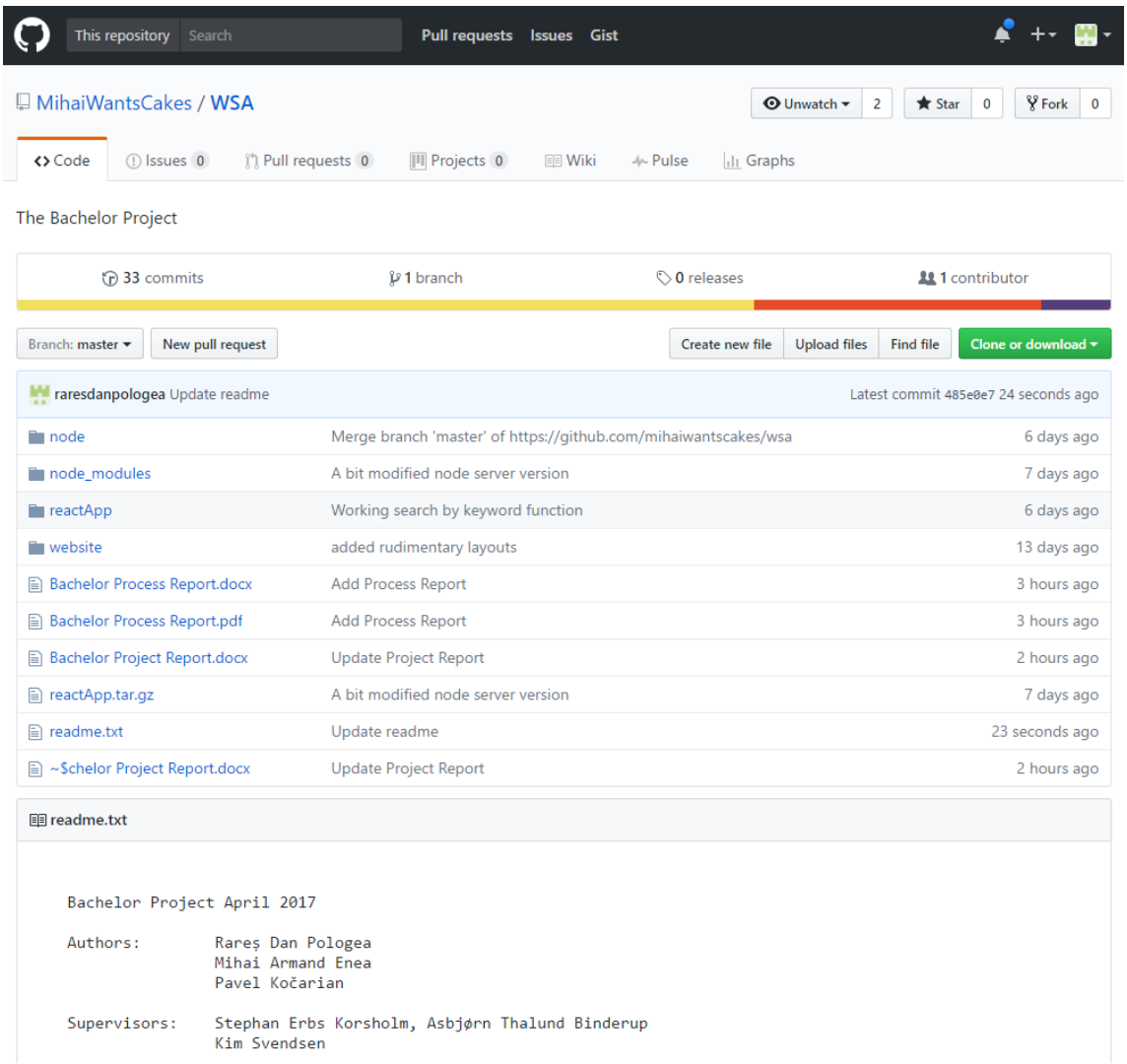


Figure 22. GitHub interface for the current project

3.3.6. Application structure

Client side file structure

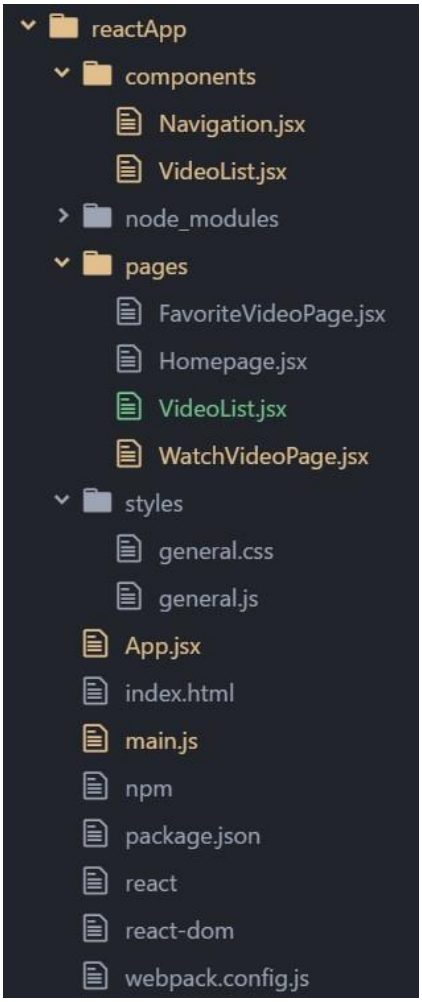


Figure 23. Client side file structure

- **reactApp/components/** – this directory contains custom React components. In this project the only custom component is Navigation.jsx.
- **reactApp/node_modules/** – folder contains all the necessary packages/libraries used by React application.
- **reactApp/pages/ -** this folder contains custom page modules used in Navigation component.
- **reactApp/styles/** – application CSS styles.
- **reactApp/App.jsx** – primary React parent component.
- **reactApp/index.html** – a main HTML where all React components and pages are rendered.

• **reactApp/main.js** – a file which contains implementation of socket data connectors, emitters and React component renderer.

Server side file structure

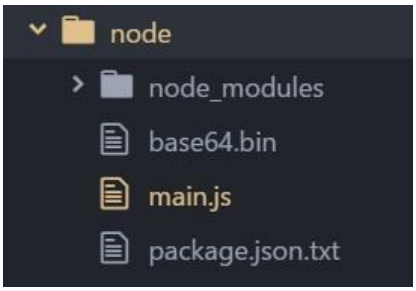


Figure 24. Server side file structure

• **node/node_modules -** folder contains all the necessary packages/libraries used by Node server.

• **node/main.js** – a main file containing all server side functionality.

4. Graphical User Interface

This chapter will cover the details behind the implementation of the Graphical User Interface which can be seen on the client side.

For this project, the team used Bootstrap v3.3.7 to create the design of the web pages.

Bootstrap is a front-end web framework for designing web applications and websites. It is open-source software and it is free to use.

It comes with templates for buttons, lists, tabs, navigation and other interface components which are HTML and CSS based. So, by using it, Bootstrap enables the chance to use already made blocks that help developers get started.
In addition to this, all Bootstrap 3 or above versions support the latest browser models of Google Chrome, Firefox, Opera, Safari, and Internet Explorer. It also supports responsive web design, meaning that the layout of any web page will adjust accordingly depending on the type of device being used such as mobile phone, desktop, or tablet.

Bootstrap comes equipped with stylesheets that give basic style definitions for key HTML components. They provide uniform and modern appearances for text formatting. Bootstrap also contains other elements which are implemented as CSS classes and must be applied accordingly to certain HTML elements. Besides this, Bootstrap comes with additional user interface elements such as tooltips and dialogue boxes. They are JavaScript components in the form of JQuery plugins.

As a final saying, a large community supports Bootstraps so even if problems where to occur, it would prove rather easy to look for help in the right place.

Besides Bootstrap, React.JS also plays a part in the Graphical User Interface. React is a NodeJS environment module created by Facebook developers for front-end development. This technology allows programmers to easily add/remove/modify front-end components in real time without refreshing the whole HTML page. React components have lifecycle functions, which are very useful when outputting non-static data on a page. A huge advantage of using React over, for example JQuery, is the environment where it runs –NodeJS, allowing developers to use the Node Packet Manager (npm) for installing additional custom components.

A paper sketch made by the team in which the design of the "Home" Web Page can be seen in the following pictures (the pictures are intentionally large to be easily readable):

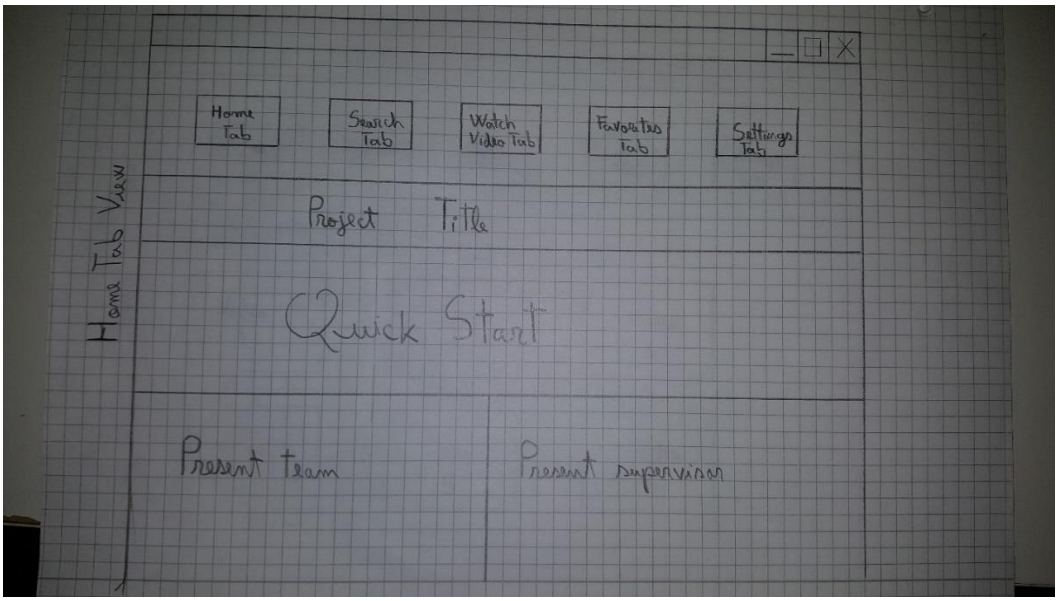


Figure 25. Home page sketch

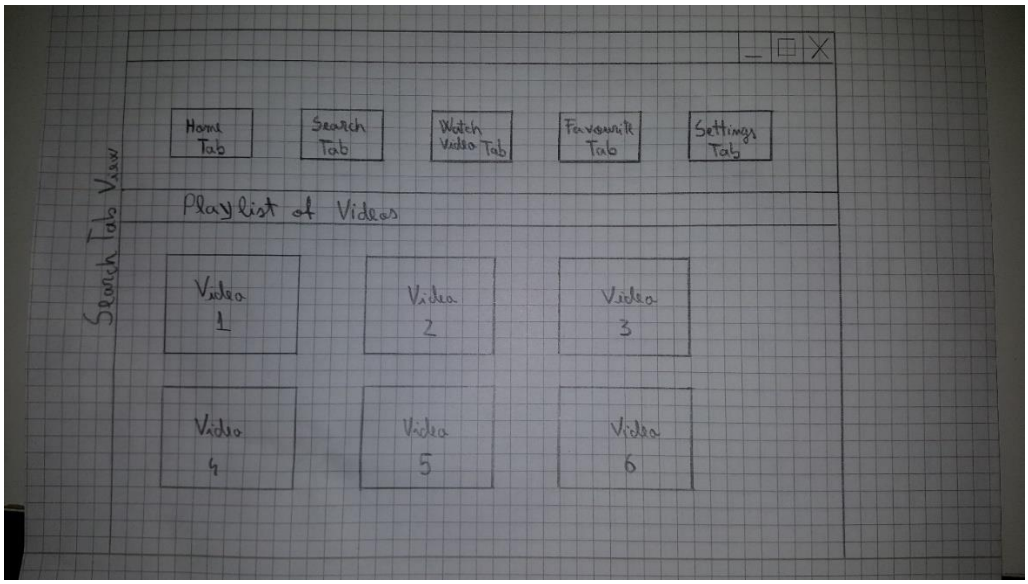


Figure 26. Select result from list sketch

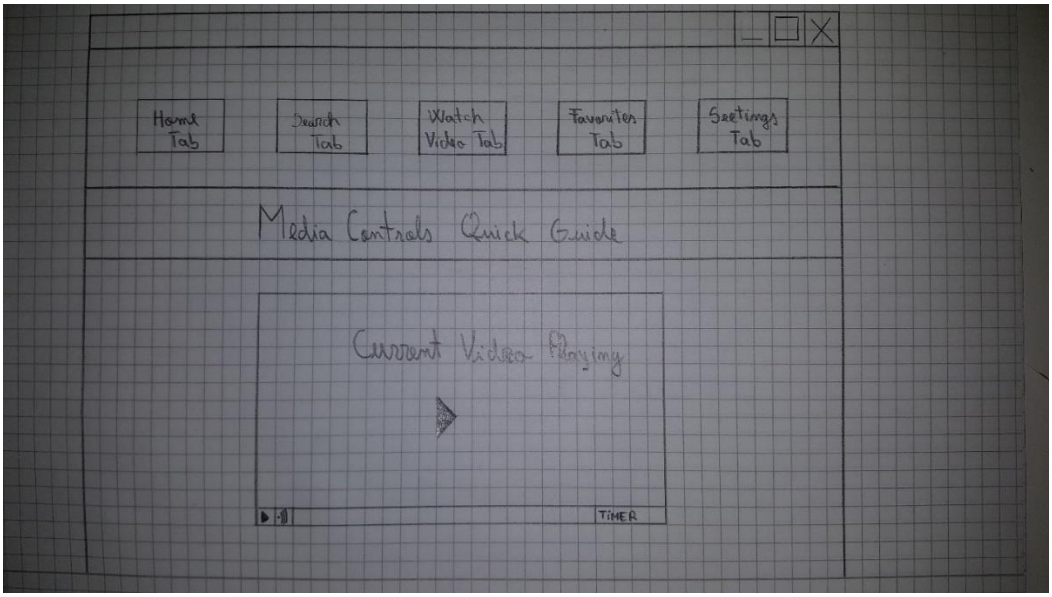


Figure 27. Watch video sketch

As a comparison, the next three pictures show what the actual website looks like:

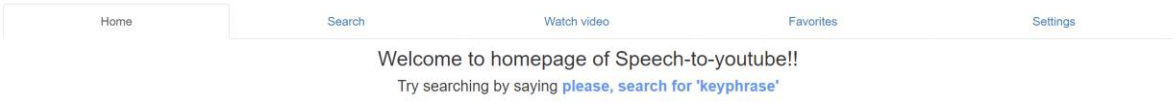


Figure 28. Homepage screenshot

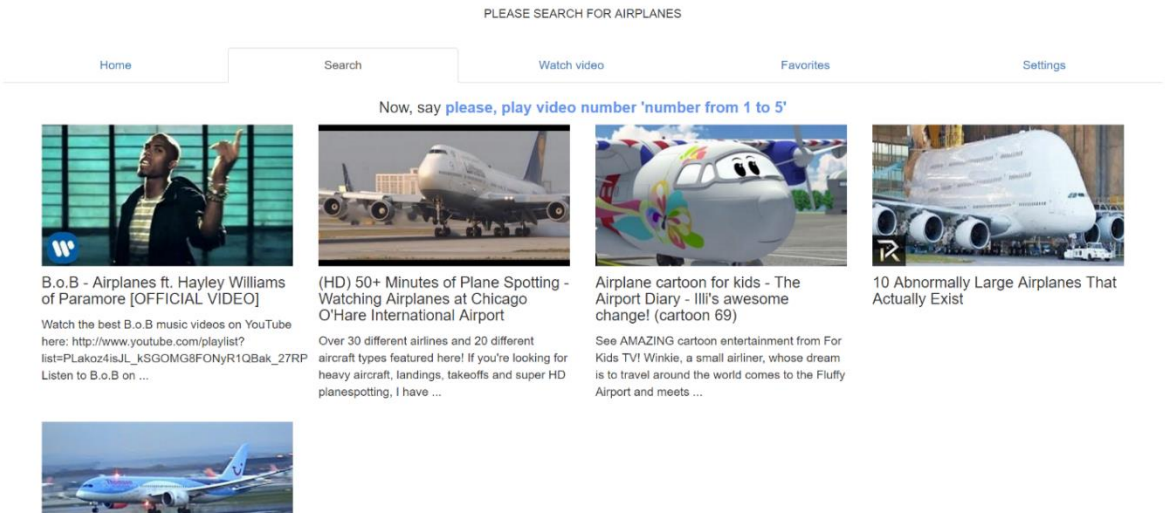


Figure 29. Select result from list screenshot

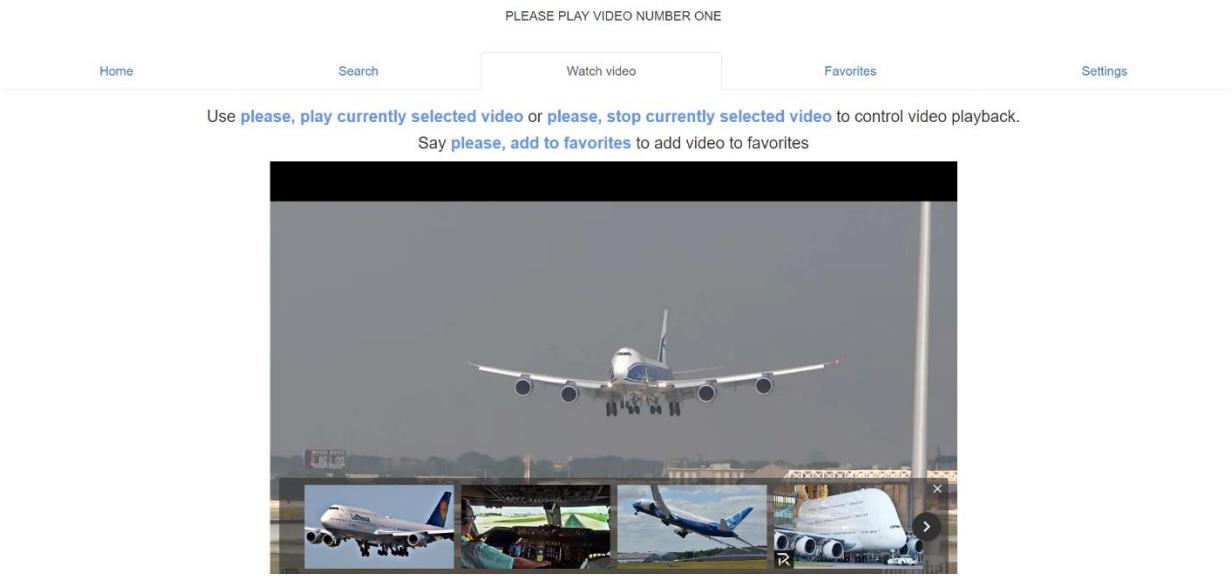


Figure 30. Watch video screenshot

As it can be seen, the actual result reflects the original design, except for several modifications so that the webpage can fit the requirements.

5. Implementation

Speech to text

Probably the most important function in this project is the speech recognition part, which was implemented with the help of the *WebkitSpeechRecognition API* and the *speech-to-text* module on the client side. This API is recording human voice from the browser window, transforming the sound into text in real time.

Speech-to-text is a Node module used for utilization of *WebkitSpeechRecognition*. As it is presented in an image below, the constructor of the **SpeechToText** function takes three objects as arguments:

```
function SpeechToText(onAnythingSaid, onFinalised, onFinishedListening)
```

Figure 31. Speech-To-Text node module on client side function declaration

- *onAnythingSaid* is a callback function that adds words to the unfinished sentence. The result from this callback is not very precise, but gets sent immediately.
- *onFinalised* - returns a whole user sentence as a text after a pause (or the end of a sentence) was detected. It is much more precise than *onAnythingSaid*, but slower in returning a result.
- *onFinishedListening* - a callback function called when WebkitSpeechRecognition stops listening.

More about speech-to-text can be found here: https://www.npmjs.com/package/speech-to-text

After the *Speech-To-Text* module was installed and imported to the app, the following code was written to send every finalized sentence to the Node server for further processing:

```
var startListening = function() {
  try {
    const onAnythingSaid = text => console.log(`Interim text: ${text}`);
    const onFinalised = text => sendMessage(text) & updateLog(text) ;
    const onFinishedListening = text => startListening();
    const listener = new SpeechToText(onAnythingSaid, onFinalised, onFinishedListening);
    listener.startListening();
  } catch (error) {
    console.log(error);
  }
}
```

Figure 32. Speech recording function implementation on client side

As *Speech-To-Text* is started, the callback function *onFinalised* is used to pass finalized sentences in the form of strings to the *sendMessage()* function, where data gets sent to the server for further processing. Additionally, the function *onFinalised* will update the user sentence log on the front end by invoking *updateLog().*The unprecise *onAnythingSaid* value can be seen being updated in a browser console, which looks like in the following picture:

```
Interim text: police
Interim text: Lisa
Interim text: research
Interim text: please search for
Interim text: please search for car
Interim text: research for gas
Interim text: please search for cats
```

Figure 33. Output from the browser console after the user said "please search for cats"

To make the system more scalable, it was decided to only have the client transform speech to text and send it to the server, whose response will influence an output on the front end.

The data exchange channel between the client and the server was established using the *Socket.io* library, which requires the server (the listening component) and the client (the connecting component) to be implemented. The working principle of *Socket.io* reminds of *Remote Method Invocation (RMI)* used in the Java programming language.

It is visible in a *Socket.io* connection function on the client side presented below that the socket is pointed to emit data to *localhost* on port 8000. Further on in the program flow this object will be used as an adapter to reach the server side.

```
var socket = io('http://localhost:8000');
socket.on('connection', function(){
  this.socket.emit('message');
  console.log("connection established!");
});
```

Figure 34. Socket connection function on the client side - Main.js

A successful socket connection grants the possibility to use the *emit* function on a socket object to transfer data to server, and the *sendMessage()* function presented below was implemented for this particular purpose taking as an argument the sentence generated by the speech to text object:

```
var sendMessage = function(text){
  socket.emit('message', text);
  console.log("sentence sent: " + text );
};
```

Figure 35. Send message to server function on client side - Main.js

The server side socket implementation is more demanding in terms of requirements than the client, in particular – the socket object needs to use an HTTP server. This was solved with *ExpressJS*, a *Node.js* package for running HTTP servers. The code snippet below shows the relation between *ExpressJS* and *Socket.io:*

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);
```

Figure 36. Socket object creation on server side – Main.js

On a successful connection, the socket callback function initializes the listener responsible for catching incoming data from client(s). The socket listener will be triggered every time it detects the header 'message' in incoming data, which is the cause of the function *sendMessage()* mentioned earlier being invoked (see Figure 27).

```
io.on('connection', function(socket){ //////on connection succes
  console.log('web interface connected!');

  socket.on('message', function(mySentence){
    console.log("Sentence received:" + mySentence);
```

Figure 37. Socket server implementation of listener – Main.js

To avoid unintended function invocations in the system it was decided to add a **wake** word, meaning that the user sentence will be treated as a command only when the wake word is present in the beginning of the sentence, like in the example below:

- *Please search for kittens* – the program will search for kittens on YouTube
- *Search for kittens* – the program will take no action

The code snippet below shows how the server detects the wake word in a sentence:

```
//function split will split a string into words and insert them into an array
var transformedSentence = mySentence.split(" ");
console.log("transformed sentence:" + transformedSentence);
//catch wake word "please"
//an if statement below will deal with a white space in a beginning of some sentences
if(transformedSentence[0] != ''){
  if( transformedSentence[0] == 'please'){
    var data = transformedSentence.slice(1, transformedSentence.length);
    executeLogic(data);

  }
}
else{
  if( transformedSentence[1] == 'please'){
    var data = transformedSentence.slice(2, transformedSentence.length);
    executeLogic(data);

  }
}
```

Figure 38. White space remover / wake word detector. Server side - Main.js

The sentence, which is a string, is converted into an array for easier data manipulation. Since some sentences received from the WebSpeech API contain a whitespace in the beginning, it was necessary to implement a whitespace checker. If position 0 of an array is a whitespace – copy data to a new array from position 1 and pass it to the function *executeLogic()*, else if whitespace was not detected on position 0, just pass the initial array to *executeLogic()*.

The function *executeLogic()* implemented inside of a previously mentioned socket object listener is meant for analyzing received user sentences, finding keywords and executing a particular action. An implementation of a search by key-phrase function can be found in the code below.

```
var executeLogic = function(onWhat){
     if(onWhat.indexOf('search') > -1 && onWhat.indexOf('for') > -1){
      //take all words besides "for" and "search"
      var plainkeywords = onWhat;
      plainkeywords.splice(plainkeywords.indexOf('search'), 1);
      plainkeywords.splice(plainkeywords.indexOf('for'), 1);

      console.log("plain keywords: " + plainkeywords);
      //Use plainkeywords to make a request to youtube
      searchOnYoutube(plainkeywords);
        console.log("Search was invoked! ");
    }
```

Figure 39. Function executeLogic(), part responsible for extracting the key-phrase from the user sentence and posting it to YouTube.

By using *indexOf()* on an array of words said by user, the program locates words ‘search’ and ‘for’ which should not be a part of the search query, and removes them by applying *splice()*. The resulting array of plain keywords gets passed to the *searchOnYoutube()* function shown in the picture below:

```
//=============================================search by keyphrase or keyword
function searchOnYoutube(keyphrase){
  var joined = keyphrase.join('&');
  var apikey = "AIzaSyDN3GpOWwYKIbldoDC-h_nkcTzz8aA3tc";
  var baseURL = "https://www.googleapis.com/youtube/v3/search?part=snippet&q=";
  var searchquery = joined + "&key=";
  var preparedrequest = baseURL + searchquery + apikey;
  console.log("YOUTUBE URL: " + preparedrequest);
  return postToYoutube(preparedrequest);
}
```

Figure 40. Function for building a URL to YouTube API.

Inside the *searchOnYoutube()* function plain keywords are used to generate a URL, along with a base URL and API key. Finally, the *postToYoutube()* function with previously generated URL is called, returning a JSON object back from YouTube API.

In the picture below an implementation of *postToYoutube()* is presented. The function simply adds request headers and options, making a GET request to the previously generated URL and getting back the response as a list of videos in form of a JSON string. Data is emitted back to the client with the message title *"search for"*.

```
function postToYoutube(url){
  var headers = {
      'User-Agent':       'Super Agent/0.0.1',
      'Content-Type':     'application/x-www-form-urlencoded'
  }
  var options = {
    url: url,
    method: 'GET',
    headers: headers,
    qs: {'key1': 'xxx', 'key2': 'yyy'}
  }
  request( options, function (error, response, body) {
          if (!error && response.statusCode == 200) {
            console.log("2. parsing response body...");
                  socket.emit("search for", body);
          }
          else if(error){
            console.log(error);
            console.log("something went wrong :(");
            return "";
          }
      });
}
```

Figure 41. Server side function which posts a request to YouTube and forwards received data to client. Server side - Main.js

Data processing on the client side

Just like on the server, the client side also has handlers for incoming socket data. One of them is presented in the picture below, and it is triggered every time the server emits data with the title "*search for*".

```
socket.on('search for', function(msg){
  active_tab = 'Search for';
  message = msg;
  executeAction();
});
```

Figure 42. Socket data handler triggered upon detection of 'Search for' in a received message. Client side -Main.js

The callback function inside the socket data handler sets an active tab variable to "search for" and assigns received data to the variable "message", afterwards calling the function *executeAction()*. The implementation of this function is presented below:

```
var executeAction = function(){
  ReactDOM.render(<App data={message}
                       active_tab={active_tab}
                       current_video={current_video}
                       playerPlaying={playerPlaying}
                       log_message={log_message}
                       favoriteList={favoriteList}
                       />, document.getElementById('app'));
};
```

Figure 43. The function responsible for rendering App component. Client side – Main.js

Every time the *executeAction()* function is called, the main front-end component *App* gets re-initialized with the data previously changed by socket data handlers.

In this project, the webpage front-end was built using **ReactJS**, a modern JavaScript framework for web-app development. One of the main advantages of React is that it allows component rendering without refreshing the whole page through refreshing separate page elements.

Inside the component *App*, the user message log and navigation component are located. The message log simply reads a log message property, which gets updated every time the socket data handler changes the contents of "log_message" and uses *executeAction()* to render front-end elements.

```
render() {
    return (
        <div id="render_zone" className="container-fluid">
          <div id="user_sentence_log_wrapper" className="col-md-12 med-top-bot-margin">
            <p id="user_sentence_log">{this.props.log_message}</p>
          </div>
          <div className="col-md-12">
            <Navigation data={this.props.data}
                        active_tab={this.props.active_tab}
                        current_video={this.props.current_video}
                        playerPlaying={this.props.playerPlaying}
                        />
          </div>
        </div>
    );
  }
```

Figure 44. First rendered front-end component. Client side - App.jsx

The rest of the data is passed to *Navigation*, where after the processing of active tab variable a tab switch occurs. A fragment of the function *render()* inside the *Navigation* component is shown below:

```
render(){
  var content_class = 'tab-pane fade in ';
  return(
    <div className="">
        <ul className="nav nav-tabs nav-justified">
          <li className={this.isHomepage()}><a data-toggle="tab" href="#home">Home</a></li>
          <li className={this.isSearch()}><a data-toggle="tab" href="#settings">Search</a></li>
          <li className={this.isWatch()}><a data-toggle="tab" href="#watch">Watch video</a></li>
          <li className={this.isFavorites()}><a data-toggle="tab" href="#favorites">Favorites</a></li
          <li className={this.isSettings()}><a data-toggle="tab" href="#settings">Settings</a></li>
        </ul>
```

Figure 45. Website navigation implementation. Client side - Navigation.jsx part 1

Each time the component rendering occurs, spanning HTML elements invoke functions which will check for the current active tab property value, and return the appropriate class name as a result. The example below is the function *isSearch()*, simply returns either the string *"active"* or an empty string, which will be used as the HTML class name. The presence of this class name will make the navigation tab selected.

```
isSearch(){
  if(this.props.active_tab == 'Search for'){
    return 'active';
  }
  return '';
}
```

Figure 46. Function checking property active_tab. Client side - Navigation.jsx

The same approach was used for displaying the page body, as it can be seen in the code snippet below. The selection of the tab will create the appropriate component enclosed inside *div* elements:

```
<div className="tab-content">
  <div id="home" className={content_class + this.isHomepage()}>
    <Homepage/>
  </div>
  <div id="search" className={content_class + this.isSearch()}>
    <div className="col-md-12 med-top-bot-margin">
      <VideoList cards={this.state.videoCards}/>
    </div>
  </div>
  <div id="watch" className={content_class + this.isWatch()}>
    <div className="col-md-12 med-top-bot-margin">
     <WatchVideoPage videoList={this.state.formattedResult} current_video={this.state.current_video_id} playerPlaying={this.props.playerPlaying}/>
    </div>
  </div>
  <div id="favorites" className={content_class + this.isFavorites()}>
    <FavoriteVideoPage/>
  </div>
  <div id="settings" className="tab-pane ">
    <h3>Settings</h3>

  </div>
</div>
```

Figure 47. Website navigation implementation. Client side - Navigation.jsx part 2

As it was mentioned before, received data from server is in form of a JSON string, so it needs to be formatted before it gets displayed on the page. To solve this problem, the function *createVideoCards()* was implemented. Inside, two arrays are created. The array named *formattedResult* will hold relevant formatted video data extracted from the parsed JSON string received in the parameter: YouTube video ID, video title, video description and a URL to the video thumbnail.

```
createVideoCards(jsonObj){
  var cardList = [];
  var formattedResult = [];
  if(jsonObj != undefined){
    try{
      this.parsedResponseBody = JSON.parse(jsonObj);
      console.log("number of items: " + this.parsedResponseBody.items.length);
      for(var i = 0; i < this.parsedResponseBody.items.length; i++){
        console.log("one video ID :"+ this.parsedResponseBody.items[i].id.videoId);
        //generate web page video card elements
        formattedResult.push({
          videoId: this.parsedResponseBody.items[i].id.videoId,
          videoTitle: this.parsedResponseBody.items[i].snippet.title,
          videoDescription: this.parsedResponseBody.items[i].snippet.description,
          videoThumbnailUrl: this.parsedResponseBody.items[i].snippet.thumbnails.medium.url
        });
```

Figure 48. Fragment of a function createVideoCards() which generates a simplified version of received video data. Client side - Navigation.jsx

The other array named *cardList* will be holding the front-end component version of output with all the data put in HTML elements, which will be outputted in the *VideoList* component. The screenshot below shows how a single video card is created:

```
cardList.push(
  <div key={'single_video_card_'+i} className="col-md-3 single-video-card">
    <img className="video_card_image" key={'video_card_image_'+i } src={this.parsedResponseBody.items[i].snippet.thumbnails.medium.url} />
    <h4 key={'video_card_title_'+i }>{this.parsedResponseBody.items[i].snippet.title} </h4>
    <p key={'video_card_description_'+i }>{ this.parsedResponseBody.items[i].snippet.description}</p>
  </div>
);
```

Figure 49. Fragment of a function createVideoCards() which puts video data into HTML elements. Client side - Navigation.jsx

After all the necessary data was pushed to arrays, the program changes the state of the class, making the function results visible to the rest of the functions in the class:

```
    this.setState({
      videoCards: cardList,
      formattedResult: formattedResult
    });
    console.log("Video cards created and list changed!");
  }
```

Figure 50. Fragment of a function createVideoCards() which updates cardList and formattedResult state values. Client side - Navigation.jsx

The **ReactJS** library supports lifecycle functions, which give a lot more control over the program flow. Below is a fragment of the function *componentWillUpdate()* in the *Navigation* component, which is triggered every time received incoming data is updated:

```
componentWillUpdate(nextProps, nextState){
  ////////////////////////////////////on data change
  if (this.props.data != nextProps.data) {
    this.createVideoCards(nextProps.data);
    this.setState({
      data: nextProps.data
    });
    console.log("navigation state has changed. data ");
  }
```

Figure 51. Fragment of lifecycle function componentWillUpdate(). Navigation.jsx

By comparing current class properties and newly received class properties it is possible to distinguish what needs to be updated. In the above example - the class state *data* is being updated with the newly received property if the difference was noticed. Additionally, the function *createVideoCards()* is invoked again to refresh the video card output arrays.

When the active tab gets set to ‘search for’, the *VideoList* component gets the previously generated video card list as a parameter, where it gets outputted on a page. This is visible in the *render()* function of the *VideoList* component shown in the code snippet below:

```
render(){
    return(
      <div className="col-md-12">
        <h4 className="text-center med-top-bot-margin">Now, say <strong>please, play video number 'number from 1 to 5'</strong></h4>
        <div>
          {this.props.cards}
        </div>
      </div>
    );
}
```

Figure 52. Function render() where cardList is displayed. VideoList.jsx

The output produced on the video list page will look like in the picture below:

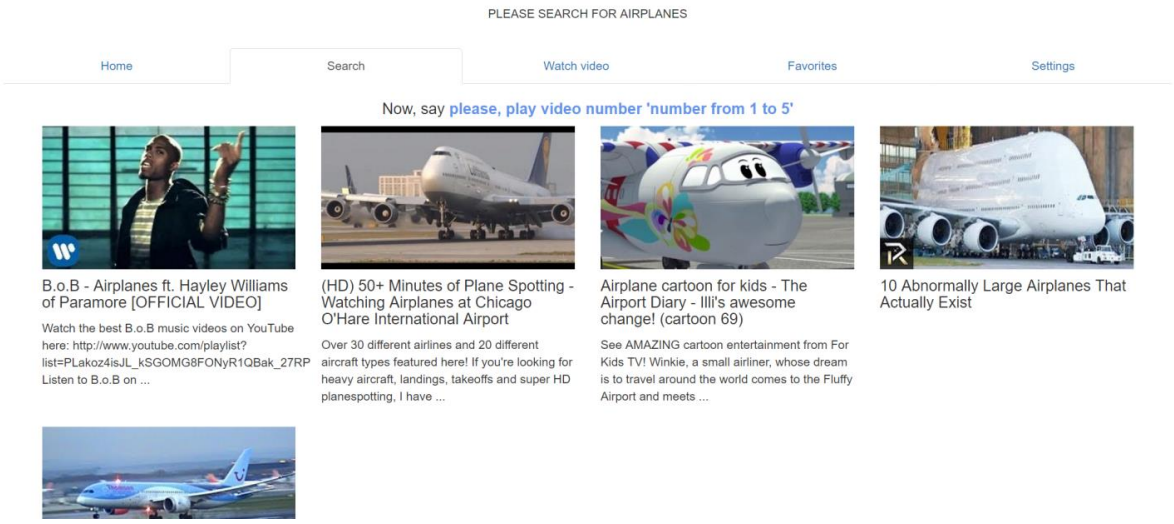


Figure 53. Search tab in a browser after the 'search for' command is triggered

After the search result is displayed, it is now possible to select a single video from the list for further playback. This was achieved by adding one more keyword combination case check to *executeLogic()* function on the server side.

```
else if(onWhat.indexOf('play') > -1 && onWhat.indexOf('video') > -1 && onWhat.indexOf('number') > -1){
    var msg = onWhat[onWhat.indexOf('number') + 1];
    //emmit command and video number to client
    switch( onWhat[onWhat.indexOf('number') + 1] ){
      case "0":
      console.log("Selecting video number 0 from the list");
      socket.emit("Select video", 0);
      break;
      case "zero":
      console.log("Selecting video number 0 from the list");
      socket.emit("Select video", 0);
      break;

      case "1":
      console.log("Selecting video number 1 from the list");
      socket.emit("Select video", 1);
      break;
      case "one":
      console.log("Selecting video number 1 from the list");
```

Figure 54. Part of the function executeLogic() on a server side executed if the user sentence contains 'play', 'video', and 'number'. Main.js

As it is shown in the screenshot above, this part of the function gets the index of the word "number" and checks for the value of the following index which is expected to be a digit or a number written in letters, returning a selected video number as an integer to the client with a message title "Select video". This output unifier was necessary to implement, since the numbers in the result coming from *WebSpeech* API can be represented in both numeric and literal form.

From the server, the selected video number travels back to the client side, where it gets captured by the socket listener. In the callback function of the listener the value of the current video index is updated and the active tab value gets changed to "Watch", making the *Navigation* component set the active tab to *WatchVideoPage* by executing the *executeAction()* function and applying the changes to the page.

```
socket.on('Select video', function(msg){
  console.log("Selecting video number " + msg);
  active_tab = 'Watch';
  current_video = msg;
  executeAction();
});
```

Figure 55. Socket data handler triggered upon detection of 'Select video' message header. Client side - Main.js

The following code fragment was taken from the function *componentWillUpdate()* of the *Navigation* component. It will update the value of the current video ID located in the component state every time the change in a component property is detected. Additionally, at this point the active tab value is also updated, switching it to video player page.

```
if (this.props.current_video != nextProps.current_video) {
  this.setState({
    current_video_id: this.parsedResponseBody.items[nextProps.current_video].id.videoId
  });
  console.log("navigation state has changed. current_video:  " + this.parsedResponseBody.items[nextProps.current_video].id.videoId);
}
```

Figure 56. State update on property change. Client side – Navigation.jsx

The current video ID is extracted from the parsed response body generated by the function *createVideoCards()* mentioned previously. The user chosen video number is used to indicate the index in the array of video data.

Part of the *render()* function in Figure 40 shows how the *WatchVideoPage* component gets initialized with the current video ID passed in the parameters. Inside *WatchVideoPage* on rendering, the *YouTubeVideo* component gets initialized with the video ID, preparing the video for watching. The picture below shows the *render()* function of *WatchVideoPage* where the YouTube player component gets created:

```
<div>
  <YouTubeVideo
    position={this.state.position}
    videoId={this.props.current_video_id}
    playing={this.state.playing}
    volume={this.state.volume}
    width="760"
    height="515"
    playerVars={{
```

Figure 57. YouTube component creation. Client side – WatchVideoPage.jsx

The code above shows that the control over the player is invoked using the writable state variables. The state editing happens in the *componentWillUpdate()* function of *WatchVideoPage*. Output of the selected video in the browser should look like in the picture below:

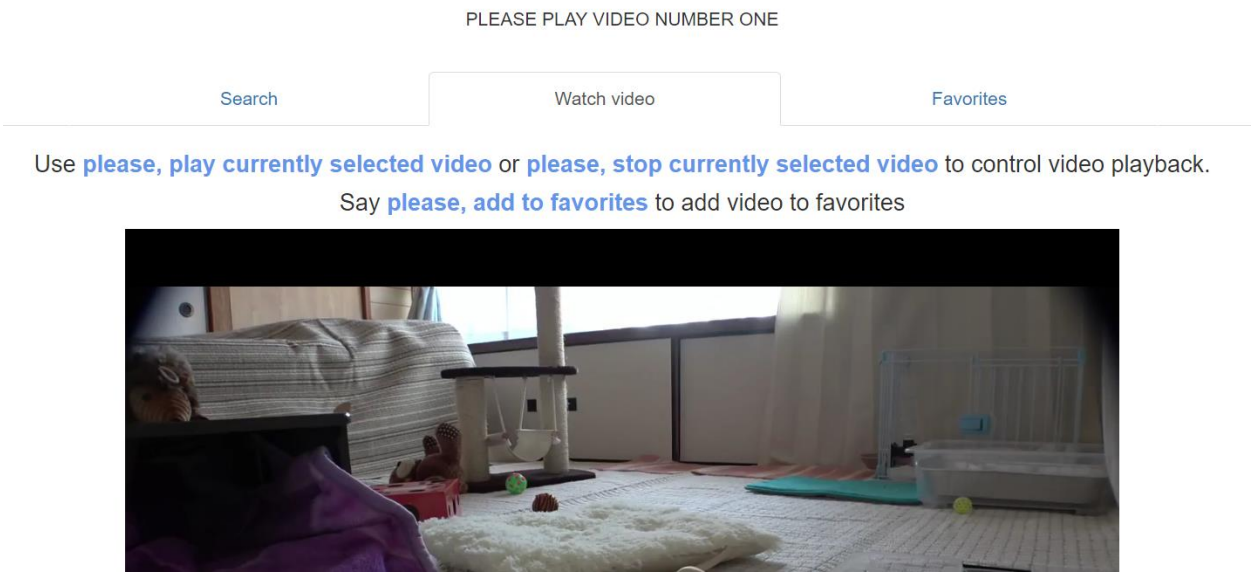


Figure 58. Viewing the selected video in a browser.

Implementation of the video playback control starts on the server. By adding another code block for detection of the keywords ‘play’ or ’stop’, ‘video’ into *executeLogic()* located inside the socket listener on the server can now change the player state on user command.

```
else if((onWhat.indexOf('play') > -1 && onWhat.indexOf('video') > -1)){
  //start playing video currently set in a player window
    var msg = "Playing currently selected video";
    socket.emit("play current", msg);
    console.log("emitting data: " + msg);
}
```

Figure 59. The implementation of the player playback status changer inside the executeLogic() function. Server side – Main.js

The socket listener created on the client side will change the value of the *playerPlaying* variable and call the *executeAction()* function to push this variable to the player window. When the player status value reaches *WatchVideoPage* the selected video starts playing.

```
socket.on('play current', function(msg){
  playerPlaying = true;
  executeAction();
});
```

Figure 60. The socket listener responsible for changing the YouTube player playback status. Client side – Main.js

Interaction with list of favorites

The list of favorite videos is stored on a MongoDB database deployed on the server side. For the connection to the database a package called Mongoose needs to be imported into the *Node.js* server. After defining a table model, the Mongoose object connects to the database on a given address, and allows direct access to the favoriteVideo table. The code presented below contains the database object definition followed by the connection with the database initialization:

```
var favoriteVideoModel = mongoose.model('favoriteVideo',
{
  videoId: String,
  videoTitle: String,
  videoDescription: String,
  videoThumbnailUrl: String
});
mongoose.connect('mongodb://localhost/favoriteVideos');
```

Figure 61. Connection to MongoDB. Server side – Main.js

Next, the function for adding an object to the database was created.

```
var addToFavorites = function(video) {
  try{
    mongoose.favoriteVideos.save(video);
    console.log("Video has been successfully saved!");
  }catch(ex){
    console.log("ERRRRRRR: "+ex);
  }
}
```

Figure 62. Function which adds object to database. Server side – Main.js

After preparing the database, some additional functionality had to be added into the function *executeLogic()* to capture keywords which are needed to trigger the next stages of the program. An additional keyword extractor is shown in the snippet below. The code will emit messages to the client, requesting current selected video data.

```
else if(onWhat.indexOf('add') > -1 && onWhat.indexOf('to') > -1 && onWhat.indexOf('favorites') > -1){
    //Add video to favorites
    var msg = "adding video to favorites";
    console.log("received favorite video from client! adding to db... " + msg );
    socket.emit('get current video');
}
```

Figure 63. Fragment of function executeLogic() responsible for detecting keywords "add" "to" "favorites".

By adding one more socket listener on the client side and making it listen for “get current video” requests, the client program can now respond to the server with the currently selected video object. The current video data gets extracted from the object previously received from YouTube JSON.

```
socket.on('get current video', function(){
    var parsedResponseBody = JSON.parse(message);
    var favoriteVideo = {
      videoId: parsedResponseBody.items[current_video].id.videoId,
      videoTitle: parsedResponseBody.items[current_video].snippet.title,
      videoDescription: parsedResponseBody.items[current_video].snippet.description,
      videoThumbnailUrl: parsedResponseBody.items[current_video].snippet.thumbnails.medium.url
    };
    socket.emit('sending current video', favoriteVideo);
});
```

Figure 64. Socket listener activated upon detection of “get current video” in message header. Client side – Main.js

Finally, when single video data gets back to the server where it was requested from, a socket listener will be called, invoking the *addToFavorites()* function, saving the received video object to the database.

```
socket.on('sending current video', function(msg){
  addToFavorites(msg);
  console.log("video added to favorite list!");
});
```

Figure 65. Saving video to database.

6. Testing

In this chapter, the team will present the process of evaluating this project and its components with the intent to find out if it satisfies the project’s requirements or not.

The evaluation process is conducted with the help of integration testing. The method used is Bottom-up integration. This method begins with unit testing, followed by tests of combinations of units called modules or cases. In this document, they will be referred to as cases.

The unit tests are performed on individual units of source code assigned with specific goals. This is done to prove that individual parts of the code are correct in terms of requirements and functionality.

In this document, the requirements from the Analysis chapter of the project are analyzed and tests are conducted to ensure whether the software passes or fails in terms of its functionality. To ensure that there are no misunderstandings regarding the scope of this “Testing” chapter the following sentences are presented below:

- This document does not address the security and evaluation testing for the project.
- This document does not address the system compatibility testing for the project.

6.1. Integration testing specifications and procedures

1. Features to be tested

Specific functionality and features of the project to be tested can be seen in the Requirement Traceability Matrix which is placed at the end of the testing chapter to better reflect the results. Each requirement in the RTM is linked to a test case so that testing can be done for each requirement.
Furthermore this RTM will have a Bug/Error ID associated to these test cases. The Bug/Error ID is classified in the table below.

Bug/Error ID Table:

Table 2. Bug/Error ID Table

Bug/Error ID	Bug/Error Priority Level	Priority Description
5	Must Fix	This bug must be fixed immediately; the product cannot be delivered with this bug.
4	Should Fix	These are important problems that should be fixed as soon as possible. It would be an embarrassment to the team if this bug should be delivered.
3	Fix when have time	The problem should be fixed within the time available. If the bug does not delay the delivery date, then fix it.
2	Low Priority	It is not important that these bugs be addressed. Fix these bugs after all other bugs have been fixed.
1	Trivial	Enhancements/Good to have features incorporated- just are out of the current scope.

2. Test Specifications and Procedures

The input specifications for these test cases can be seen in the “Input Action” column. These can also be referred to as Unit Tests. In the “Expect Results” you will find the expected outputs given each input. In the “Observed results” column, the outcome is dictated by either a success or a failure, given if the input succeeds or not. Last is the “Pass/Fail” column which will be completely influenced by the outcome of its previous column.

The test cases shown here will be the Use Case presented in the analysis and described in the Implementation chapter.

Test Name: Test Case 1: Convert Speech to Text
Description: The user can convert speech to text and see the results
Requirement ID: 01
Prerequisites: Active internet connection, user is on the home tab page
Setup: The user opens the web page which will open on the home tab page by default.

Steps Table for Test Case 1:

Table 3. Test Case 1 steps table

Step	Operator Action (Unit Test)	Expected Result	Observed Result	Pass/Fail
1.	User says any sentence in English language to the microphone	User sentence has to appear in a browser console	User sentence appears in a browser console, but in some cases some words were not recognized	Pass

Test Name: Test Case 2: Web Site Navigation
Description: The user can navigate the website using voice commands
Requirement ID: 02
Prerequisites: Active internet connection, user open's web site
Setup: The user opens the web page, and navigates to either tab presented on the Home Page.

Steps Table for Test Case 2:

Table 4. Test Case 2 steps table

Step	Operator Action (Unit Test)	Expected Result	Observed Result	Pass/Fail
1.	User says 'go to "tab name"'	User sentence in form of a string has to be emitted to server via socket	User sentence in form of a string is emitted to server via socket	Pass
2.	Server captures the user sentence emitted by client	User sentence should appear in a server console window	User sentence appears in a server console	Pass
3.	Server emits the name of a selected tab back to client	Active tab in a browser window should switch	Active tab changes	Pass

Test Name: Test Case 3: Search YouTube video by keyword
Description: The user can search a video on YouTube using voice commands
Requirement ID: 03

Prerequisites: Active internet connection, user is on the video tab page.
Setup: The user opens the web page and navigates to the Video tab page.

Steps Table for Test Case 3:

Table 5. Test Case 3 steps table

Step	Operator Action (Unit Test)	Expected Result	Observed Result	Pass/Fail
1.	User says 'search for "keyphrase"'	User sentence in form of a string has to be emitted to server via socket	User sentence in form of a string is emitted to server via socket	Pass
2.	Server captures the user sentence emitted by client	User sentence should appear in a server console window	User sentence appears in a server console	Pass
3.	Server generates YouTube URL with search keyphrase	URL to YouTube API should be displayed in a server console window	URL gets displayed in a server console window	Pass
4.	Server extracts search result data from a generated URL and emits it to client along with a new active tab value	Number of received objects should appear in a browser console	Number of received objects will be shown in a browser console	Pass
5.	Client generates video cards on front-end	Web page should display a formatted video list	Formatted video result list is being generated on a page	Pass

Test Name: Test Case 4: Select Video from a playlist
Description: The user can select a video of his choice from a playlist of videos
Requirement ID: 04
Prerequisites: Active internet connection, user gives voice commands to search for a video.
Setup: The user opens the web page, navigates to the Video Tab page and searches for a video.

Steps Table for Test Case 4:

Table 6. Test Case 4 steps table

Step	Operator Action (Unit Test)	Expected Result	Observed Result	Pass/Fail
1.	User says ‘play video number “ 4 > n > 0 ”’	Server should print a chosen video number in a server console	Selected video number is printed out in a server console	Pass
2.	Server emits an index of a chosen video back to client along with a new active tab value	Navigation tab in a browser window should switch to YouTube player window with a video selected by user	Active tab changes to single video page and displays user selected video on pause	Pass

Test Name: Test Case 5: Media Controls
Description: The user can control the video such as Play/Pause/Stop using voice commands.
Requirement ID: 05
Prerequisites: Active internet connection, user is on the Video Tab page and has already searched for a video.
Setup: The user opens the web page, navigates to the Video Tab page and searches for a video.

Steps Table for Test Case 5:

Table 7. Test Case 5 steps table

Step	Operator Action (Unit Test)	Expected Result	Observed Result	Pass/Fail
1.	User says ‘play currently selected video’	Server should print a success message in a console and emit boolean value back to client	New player playback value gets emitted to client successfully	Pass
2.	Player playback value is received	A video previously selected by user should start playing	User selected video starts playing	Pass

Test Name: Test Case 6: Save to Favorites
Description: The user can choose to save a video using voice commands
Requirement ID: 06
Prerequisites: Active internet connection, user is on the Video Tab page and has already searched for a video.
Setup: The user opens the web page, navigates to the Video Tab page and searches for a video.

Steps Table for Test Case 6:

Table 8. Test Case 6 steps table

Step	Operator Action (Unit Test)	Expected Result	Observed Result	Pass/Fail
1.	User says ‘add to favorites’	Server should capture the request and print an approval message in server console	Approval message appears in the console, signalizing that client request was received	Pass
2.	Server requests currently selected video data from a client	Received object data should be printed in a console	Received object gets printed out in a server console	Pass
3.	Server adds previously received favorite video object to database	Video object should be added to database followed by a success message visible in a server window	Success message is logged in a server console, indicating operation success	Pass

RTM matrix:

Table 9. RTM Matrix

Requirement ID	Requirement description	Comments	Bug/Error ID	Approved by
01	The user can convert speech to text and see the results	All team members agreed	5	Dev team
02	The user can navigate the website using	All team members agreed	5	Dev team

	voice commands			
03	The user can search a video on YouTube using voice commands	All team members agreed	4	Dev team
04	The user can select one video to watch from a playlist of videos.	All team members agreed	4	Dev team
05	The user can control the video such as Play/Pause/Stop using voice commands.	All team members agreed	3	Dev team
06	The user can choose to save a video to Favorites.	All team members agreed	3	Dev team

7. Results

When the implementation of the system reached, the expectations imposed in the requirements of this project as well as within the use cases, the team observed the following results.

- Speech to text

A user can open the application and when reaching the home page, can start speaking into the default microphone or a personal added microphone. The system will successfully capture the spoken input and transform it into text. The transformation is successfully visible in the website's console.

- Website Navigation

A user can successfully navigate throughout the website's content which is stored in tabs. Some of the tabs are successfully accessed with commands that are made to give the user a shortcut directly to those tabs. The rest can be navigated successfully using the same command and the keyword associated with each tab.

- Search video on YouTube

A user can successfully make use of one of the voice commands made for this particular case, followed by his/hers desired request. After the command is processed, the user is successfully rewarded with a list of videos received from YouTube.

- Select video from a playlist

A user can successfully select a video from the playlist using another prepared voice command and get automatically moved to the Watch Video tab where a new series of commands can be used.

- Media controls

A user can successfully use the media controls implemented, such as play, pause and stop.

- Save to favorites

A user can successfully ask throughout yet another voice command to have his/her videos saved to Favorites. By doing this, the user can then navigate to the Favorites tab to see the desired video saved in a playlist successfully.

6.2. Performance testing specifications and procedures

The system performance was split into several separate requirements as such:

1. The system shall correctly convert voice input into text in at most 1s.
2. The system shall correctly convert text into a command in at most 1s.
3. The system shall correctly convert voice input into text in at least 90% of the cases in a quiet environment.
4. The system shall correctly convert voice input into text in at least 40% of the cases.
5. The system shall correctly convert text into a command in at least 95% of the cases.

1. The time it takes the system to convert voice input into text was calculated by running 25 different tests with 25 different phrases in a quiet environment. For the testing environment, the team members used an empty classroom. During the quiet environment test, the only sound source in the room was the system user.
   The average time it took the system to convert a phrase into text was 3 milliseconds. Since 3 milliseconds is much lower than 1 second, the test was a success.

```
Interim text:  please have some words to the sentence at 1492980009309
word added in 2 milliseconds
```

Figure 66. Voice input into conversion text speed test

   Note: The number at the end of the first row is the local time in milliseconds.

2. The time it takes the system to convert voice input into text was calculated by running 25 different tests with 25 different phrases in a quiet environment. For the testing environment, the team members used an empty classroom. During the quiet environment test, the only sound source in the room was the system user.

The average time it took the system to convert the text into a command and return a result was around 700 milliseconds. Since 700 milliseconds is less than 1 second, the test was a success.

```
Interim text: please search for dark matter at 1492979458944
sentence sent: please search for dark matter
log bar updated
response received in 654 milliseconds
number of items: 5
```

Figure 67. Text to command conversion speed test

Note: The number at the end of the first row is the local time in milliseconds.

3. The percentage in which the system converts voice input into text was calculated by running 25 tests with 25 different phrases in a quiet environment. For the testing environment, the team members used an empty classroom. During the quiet environment test, the only sound source in the room was the system user.

   The results observed are as follows:

   In the quiet environment, the system correctly converted voice input into text in 23 out of 25 cases, meaning it had a success rate of 92%.

   The reason behind the system's failure in noisy environments is the low-quality microphone, which makes the recording of voice input more difficult. A higher-quality microphone would further increase the success rate.

   Note: the success rate depends on the internet connection of the system. If the system is unable to connect to the internet, the system will not be able to do the conversion.

4. The percentage in which the system converts voice input into text was calculated by running 50 tests: 25 tests with 25 different phrases in a quiet environment and 10 tests with 10 different phrases in a noisy environment. For the testing environment, the team members used an empty classroom. During the quiet environment test, the only sound source in the room was the system user. During the noisy environment test, the other two team members had a normal conversation around 2 meters away.

   The results observed are as follows:

   In the quiet environment, the system correctly converted voice input into text in 23 out of 25 cases, meaning it had a success rate of 92%.

   In the noisy environment, the system correctly converted voice input into text in 10 out of 25 cases, meaning it had a success rate of 40%.

   The combined success rate is 66.67%.

The reason behind the system's failure in noisy environments is the *WebSpeech API*'s inability to discern between the user's commands and ambient sound. Another source of error is a low-quality microphone, which makes the recording of voice input more difficult. A higher-quality microphone would further increase the success rate.

Note: the success rate depends on the internet connection of the system. If the system is unable to connect to the internet, the system will not be able to do the conversion.

5. The percentage in which the system converts text into commands was calculated by running 25 tests: the system was checked whether it had converted in a command text received from 25 successful voice-to-text conversions. This way, the voice-to-text operation did not affect the results.

   The results are that the system correctly converted text into commands in 25 out of 25 cases, meaning it had a success rate of 100%.

   Note: the success rate does not depend on the internet connection of the system. If the system is unable to connect to the internet, the system will still be able to do the conversion.

8. Discussion

In this chapter are presented ideas that the team had about features which would enrich the experience of using the system.

1. User control

As it may have been noticed up to this point, the user control use cases have never been approached in this project. Although the implementation of user control in such a system is ideal and gives a better sense of control to everything, and of course makes the users feel more secure and private, the team has decided against it in this project, as it has proven to be of the scale of a standalone project, so they left it for future improvements as it would be great to have.

A simple method like just adding a keyboard and typing the password it was approached, but the team considered that by adding a keyboard of for this purpose would beat the purpose of a hands-free system.

Another problem is that the *Web Speech API* used in this project is not capable of making the difference between different speech patterns used by different users, and neither are the Amazon Echo, Google Home or other hardware or software choices that have been considered for this project.

Of course, such technology exists but it is either very expensive or not working properly yet, as no perfect speech analyzer has been found the team in their research.

By not having the possibility of recognizing different speech pattern the system cannot make the difference between a 10-year-old user that might accidentally stumble upon some restricted content that only a user over the age of 18 would have access to.

Any user that would eventually have to log in by voice commands will be heard by either a family member or some guests and such the user privacy loses its consistency.

With a system that could make a user record several sentences in which he pronounces series of strings that will contain both letters and numbers which are commands that the system understands, then only that user will be able to use the system as his commands will be forever compared to the ones recorded.

Even this method has possible flaws in which the user's voice may change in time due to a common cold or even aging, thus having to go through the recording stage again, or else the system will not be able to understand the commands.

So even though it's not a perfect speech pattern recognizer it would a solution to the hands-free user control use case.

2. User profile and settings

This is yet another use case presented which ended up in the improvements chapter. It is directly related with the previous improvement mentioned in this chapter.

If the system had been improved with the possibility to have users log-in by voice commands then, each user would have the possibility of making a profile.

By making a profile a user would get access to a new tab which handles Profile settings.

These settings would involve changing the visual aspect of the website such as tweaking the layouts and changing colors among others.

By adding profiles, users will also have a tab that keeps count of the most watched videos, the most visited web sites, or the most listed music. This will be the History tab.

3. Using better equipment.

At this moment, after doing the research upon which to reflect along with the Amazon Echo approach, the team has concluded that a lot of improvements can be done, mostly by using better equipment. In time, if Amazon would approve this application, the system could be modified to use the *Alexa* Voice Service, or the new *Google Cloud API*, in which case both have a more evolved technology in speech recognition and noise cancellation.

Alongside these chances, a top-quality microphone would improve the system's understanding of voice input.

4. Advanced media controls

The system already gives the user a small control over the video that he receives, such as play, stop, pause. However, nowadays players come equipped with even more commands, and as the system is made to give the user a good experience when using the hands-free video player, the team considers that adding even more commands would improve the user experience.

Some examples of commands are:

- fast forward
- volume control
- skip to next video, skip to previous video
- change between small and full screen
- like, dislike videos or subscribe to channels
- have a list with subscribed channels and a possibility to navigate it

In a few words this improvement can be described as making the whole YouTube interface hands-free so that the user can experience it to the fullest.

5. Expand the system to use more popular API's

This improvement evolved around the idea that if the team succeeded in making use of the *YouTube API* to do various operations, then it should be possible to adapt the system in future development to use more API's.

Here are some examples:

- Adapt the system to use the Google Search engine and receive responses in an ordered list.
- Adapt the system to use Facebook and perform various tasks such as checking in, browsing through the time line, or checking out other friends.
- Adapt the system to use Spotify to browse for music, select music from various playlists, save them to favorites and so on.

6. Speech synthesizer

This improvement involves the user of the extra functionality which the *webkitSpeechAPI* can offer.

As mentioned in the Design chapter, this API has functions and methods for adding an A.I. voice meant for responding to certain commands, or simply chatting with the user.

Since the team now obtained some experience in using the *webkitSpeechAPI*, this idea raised new possibilities.

It will make the system even more user friendly, as any user will now have a personal assistant that will interact either as a helper or a guide, much like the well-known Siri from Apple or Cortana from Microsoft.

7. Website Pop-ups

This idea involves around one of the previous improvements mentioned in this chapter.

The website can have pop-us which will guide the user in some cases, such as:

- when a user reaches a new page, the pop-ups appear giving him guidance on what can he do on that page.

- after a user fails to give a proper command and continues for an amount of time, the webpage will give a pop-up with suggestions on what can be done to get passed that situation.

- a tutorial that is given to users on their first try of the system. The tutorial can, of course, be de-selected so the user will not be stuck with pop-ups for the entire experience.

8. Remote control with microphone

As people may or may not have heard, there are remote control hardware devices that can be equipped with a microphone and bound to either smart TV's, laptops or others, including the system presented in this document.

By adding such a remote control to the system, the user will gain more control over the microphone and make the system more precise and reliable. By having the user speak directly to the remote it will make it easier for the system to understand the spoken words even in more noisy moments.

Also, as an extra feature the remote can also come equipped with a mouse-like pointer and act like a mouse on the screen. This comes in handy in case of a microphone drop or anything that would lead to the voice recognition not behaving as expected, as the system could still be used until the problem is solved.

Such a remote is already present on the market for the latest TV models and other devices, but it cannot be so easily added to the system, by just buy purchasing it. The remote will first have to be configured and connected to work for this project. This is where the actual improving will happen.

9. Conclusion

The aim of this project was to create a system which takes input from users in the form of voice commands, converts said commands into text, interprets the text into commands that can be executed, execute them and display the output on a screen.

The current version of the system offers a system which allows the user to search YouTube videos by keyword, play a certain YouTube video from a playlist, play/pause a YouTube that is selected, offers a GUI and allows the user to switch tabs in the respective GUI, all through voice commands.

The most important feature of the system is the fact that the system can convert voice commands into text and, ultimately, into actual commands that can be executed. This feature lies beneath every other feature, as the voice-to-text and text-to-command conversions are necessary for all the other features.

Another feature that the user can use is searching a video on YouTube. To do that, the user must say a certain phrase: "please search for <keyword>". When the user does that, the system will display several results. After this, the user must select one of the results using a key phrase which

specifies which result to open. Finally, the user is presented with the choice of playing or pausing the respective video.

Throughout the development period there were some features that the team thought of but were not able to implement. The team has also thought of features that could be implemented in a future iteration to further add to the system and improve it overall.

All the problems encountered during the development period as well as future improvements were taken into consideration, however, the system fulfills the requirements set out earlier and reaches the aim of the project. The system is considered a success.

10. References

1 – Wikipedia.org. 2017. Speech recognition - Wikipedia. [ONLINE] Available at: https://en.wikipedia.org/wiki/Speech_recognition. [Accessed 24 August 2016].

2 – Wikipedia.org. 2015. Template:HMM example. [ONLINE] Available at: https://en.wikipedia.org/wiki/Template:HMM_example. [Accessed 26 August 2016].

3 - Michael McConnell. 2016. Alexa, How Does Siri Work? Voice Control Explained. [ONLINE] Available at: http://www.makeuseof.com/tag/alexa-siri-work-voice-control-explained/. [Accessed 29 August 2016].

4 - Amazon.com. 2017. Amazon Echo - Amazon Official Site - Alexa-Enabled. [ONLINE] Available at: https://www.amazon.com/Amazon-Echo-Bluetooth-Speaker-with-WiFi-Alexa/dp/B00X4WHP5E. [Accessed 1 September 2016].

5 - Google.com. 2017. Google Home - Made by Google. [ONLINE] Available at: https://madeby.google.com/home/. [Accessed 2 September 2016].

6 - Google.com. 2017. Speech API - Speech Recognition | Google Cloud Platform. [ONLINE] Available at: https://cloud.google.com/speech/. [Accessed 16 March 2017].

7 - Frederic Lardinois. 2014. Natural Language Platform Wit.ai Raises $3M Seed Round Led By Andreessen Horowitz. [ONLINE] Available at: https://techcrunch.com/2014/10/15/natural-language-platform-wit-ai-raises-3m-seed-round-led-by-andreessen-horowitz/. [Accessed 18 March 2017].

8 - Wit.ai. 2017. Wit.ai. [ONLINE] Available at: https://wit.ai/. [Accessed 19 March 2017].

9 - Wit.ai. 2017. Wit --- Recipes. [ONLINE] Available at: https://wit.ai/docs/recipes. [Accessed 19 March 2017].

10 - raspberrypi.org. 2017. Raspberry Pi - Teach, Learn, and Make with Raspberry Pi. [ONLINE] Available at: https://www.raspberrypi.org/. [Accessed 5 September 2016].

11 - beagleboard.org. 2017. BeagleBoard.org - black. [ONLINE] Available at: https://beagleboard.org/black. [Accessed 8 September 2016].

12 - intel.com. 2017. Introducing the Intel® Compute Stick. [ONLINE] Available at: http://www.intel.com/content/www/us/en/compute-stick/intel-compute-stick.html. [Accessed 9 September 2016].

13 - Glen Shires, Hans Wennborg. 2014. Web Speech API Specifications. [ONLINE] Available at: https://dvcs.w3.org/hg/speech-api/raw-file/tip/webspeechapi.html#speechreco-attributes. [Accessed 3 April 2017].

14 - Glen Shires, Hans Wennborg. 2014. Web Speech API Specifications. [ONLINE] Available at: https://dvcs.w3.org/hg/speech-api/raw-file/tip/webspeechapi.html#speechreco-methods. [Accessed 3 April 2017].

15 - Glen Shires, Hans Wennborg. 2014. Web Speech API Specifications. [ONLINE] Available at: https://dvcs.w3.org/hg/speech-api/raw-file/tip/webspeechapi.html#speechreco-events. [Accessed 3 April 2017].

16 - Glen Shires, Hans Wennborg. 2014. Web Speech API Specifications. [ONLINE] Available at: https://dvcs.w3.org/hg/speech-api/raw-file/tip/webspeechapi.html#speechreco-error. [Accessed 3 April 2017].

17 - Glen Shires, Hans Wennborg. 2014. Web Speech API Specifications. [ONLINE] Available at: https://dvcs.w3.org/hg/speech-api/raw-file/tip/webspeechapi.html#speechreco-result. [Accessed 3 April 2017].

18 - Glen Shires, Hans Wennborg. 2014. Web Speech API Specifications. [ONLINE] Available at: https://dvcs.w3.org/hg/speech-api/raw-file/tip/webspeechapi.html#speechreco-resultlist. [Accessed 3 April 2017].

11. Appendix

Appendix A – Alexa-enabled RaspberryPI

Appendix B – Description for all hardware components considered in this project

Appendix C – Amazon Echo approach

Appendix D – Diagrams and Descriptions