
# Special Problem in IT

Information Technology and Management

# FACTOR

Speech-to-Text API implementation for LMS

*Mikel Martin*

Chicago, August 22, 2021

# 1 ABSTRACT

The proliferation of the web and the new online teaching methodologies based on videoconferences has provoked a deficit in the amount of written learning material available. Even though this method is an obvious step forward from the classic teaching methods, being able to read the materials offers the students a more personalized experience in which they can benefit from learning the concepts at their own pace. Due to this fact, this project aims to support video lectures with a technology that has been hugely developed the last decade, speech recognition. Factor is able to offer text transcriptions of the videos thanks to this technology. Factor offers students the possibility of reading what professor's say in their lectures and even look for key concepts. It also allows professors to create materials out of their own lectures. This project is focused on allowing the schools to create multimedia libraries that complement traditional video learning platforms.

# 2 RESUMEN

La proliferación de la web y las nuevas metodologías de educación online basadas en sesiones de videoconferencia ha provocado un déficit en el material educativo que se encuentra sobre papel. A pesar de que es una clara mejora sobre el método tradicional de impartir clases de forma oral, leer la información permite realizar un estudio de los conceptos de forma más detallada y personalizada al ritmo y las necesidades del estudiante. Es por eso que este proyecto pretende complementar las sesiones en video con una tecnología que ha sido perfeccionada durante la última década, el reconocimiento de voz. Mediante esta tecnología, Factor, es capaz de ofrecer texto junto a los videos de su plataforma. Factor, permite a los alumnos poder leer las clases impartidas y realizar búsquedas de texto. Así mismo, permite a los profesores generar documentos a partir de sus clases. Este proyecto se centra en permitir al sector educativo crear una biblioteca de contenido multimedia que complemente a los sistemas de enseñanza online tradicionales.

# 3 KEYWORDS

Speech to Text, Learning Management System, Cloud APIs, JavaScript, Docker

# CONTENT

# 4 INTRODUCTION

In 1965, Gordon Moore, before he even co-founded Intel, made an observation of high relevance to the tech industry. Moore's Law [1] refers to an historically proven observation that the number of transistors on a microchip doubles every two years. What this means is that computers, and basically anything with a microchip, can reduce its price by half every two years. This offers the possibility for technology to improve almost every year, making the IT industry incredibly fast-evolving.

Universities, take between 2 to 6 years to educate student in a certain field. This means that the study plan they set has to still be useful all those years later. These study plans are mostly based on the materials and textbooks that will be studied. Although the history of textbooks dates back to ancient civilizations, they have been the primary teaching instrument for schools for the last two centuries. Since technology changes so rapidly, imagine having to change these textbooks every two years, books that take between 3 and 4 years to be written, published, sold and distributed across libraries. Universities can't afford such investments and publishers can't afford to make books with such short lifespan.

Due to these reasons, the schooling system, the one related to the tech industry specially, has seen a shift in its knowledge transfer method. Lectures, videos and hand outs are the preferred method now. But there's a catch, not many of those lectures and videos are published in text format, being voice the predominant format. This means that there are no written records, the knowledge is not being properly stored and everything is in digital format.

This is where Factor comes in handy, the project, originally brought to life by IIT's Industry Associate Professor Jeremy Hajek, was an AWS based Learning Management System for speech-to-text transcription of online lectures. The tool can help create human readable text based on the concepts explained by professors in their classes, providing all the benefits that text provides over video or voice.

# 5 OBJECTIVES

Based on the problem that wants to be solved with Factor, there are a couple of objectives that have to be achieved in this particular project.

First of all, we want to create a fully working instance of an application that presents a user-friendly front-end. This frontend should allow end users to transcribe videos and watch them with their appropriate transcription.

For the transcription options, we want to offer the widest range of speech-to-text APIs possible. The objective is to allow the user to choose the most convenient recognition engine when using the application.

We also want the application to be fully accessible. For that, it should follow a production rate deployment that meets industry standards for a school scale client-server application. The objective is to offer the tool to anyone that needs it.

Another side objective is the possibility of restricting access to the application to students or professors only. This means a school-based login, most likely with google sign in restricted to the members of the organization that owns the deployment.

Ultimately, we are developing an application for transferring knowledge so one of our objectives should be creating a persistent video library with all the content ever uploaded and all the transcripts ever created. We want to offer the most amount of information to the biggest crowd possible.

# 6 REQUIREMENTS

Based on the problem we want to solve and the objectives explained in the previous chapter, we can specify the following requirements for the project. Through these, we will set a base for the specifications of the application we are trying to build.

## 6.1 ENVIRONMENT

The application needs to be able to run in Docker. This means it needs to be developed for any of the available Linux distributions. Development can be done in any environment as long as the result is based on a Linux distribution.

## 6.2 MODULARITY

The application must follow a loosely coupled structure, implementing a cloud-based architecture of microservices that expose Application Programming Interfaces (APIs). It must consist of at least the following components: a user interface, cloud services and a database.

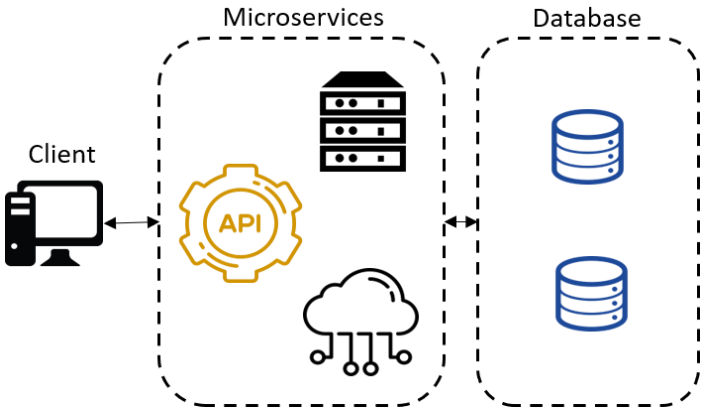


*Figure 1: Cloud based architecture*

This type of design offers the following benefits:

- Single responsibility modules for easier development and troubleshooting.
- No dependencies between services, providing the ability to develop and update independently.
- Agnosticism of another module's implementation for interface standardization.
- Independence in scalability and deployment for efficiency in resource distribution.
- Increased security by reducing exposed areas and providing deeper granularity in access.

## 6.3 AUTHENTICATION

The application must not maintain any sensitive user information. It must implement a web authentication API provided by a trusted cloud provider. This way, the application is not susceptible to any form of private data leaks and can simplify its security requirements. It can use any provider but Google is recommended to facilitate the access to IIT's student/professor

body. The project will be open source but the cloud providers charge for transcribing so access will be limited to avoid unaccountable costs.

## 6.4 User interface

The application must be web based. The HyperText Markup Language (HTML) is the industry standard for documents designed to be rendered by web browsers. This technology will be assisted by Cascading Style Sheets (CSS) to offer a user-friendly interface and by JavaScript to control the behavior of the elements showed. These technologies can be implemented with any desired pattern.

## 6.5 Application Logic

The application must have the following services available:

- Save videos: be able to receive videos and store them in an efficient way.
- Delete videos: be able to delete videos and related information.
- Serve videos: send any video ever stored in the application.
- Create transcriptions: implement either a cloud speech to text or a local API for text generation based on school classes.
- Serve transcripts: send any transcript ever stored in the application.

## 6.6 Data Tier

The application must persist all the videos and their respective transcripts as well as offer them on demand to any user of the platform. The application is meant for educational purposes thus it will be developed with the principle of education as a universal right.

# 7 SPECIFICATIONS

## 7.1 ENVIRONMENT

The application will be developed for Debian as it is the main distribution used by Docker Hub's Node official image. Node is a backend JavaScript runtime environment that will allow the application to run JavaScript for the backend. This means the application will fully be developed in JavaScript.

## 7.2 MODULARITY

The application will follow a loosely coupled architecture, implementing three different components.

- A static file sharing web application for the frontend.
- Two NodeJS backends, one for video management and another for transcript management.
- Two different databases.

## 7.3 AUTHENTICATION

The application will not maintain any sensitive user information. It will implement Google's OAuth authentication to allow IIT's members to login with the organization account.

## 7.4 WEB DESIGN (FRONTEND)

The application will be developed as a Single Page Application. Using a combination of HTML/CSS and a JavaScript framework.

## 7.5 BACKEND

The application will have two different services available.

### 7.5.1 Video API

This part of the service will be in charge of three functions:

- Sending a video list.
- Serving a video.
- Saving a video.

### 7.5.2 Transcript API

This part of the service will be in charge of two functions:

- Creating a transcript.
- Serving a transcript.

## 7.6 DATA

The application will save the data in two different resources. One for object storage and another for object reference and related information/metadata.

The reference database will keep the following information:

- Owner: the name of the uploader
- Email: the email of the uploader
- Title: the title of the video
- File Name: the file name (id)
- File Path: the URI to the video
- Transcript Path: the name and URI to each transcript.
- Date: Date stamp

The object database will persist the following files:

- Videos
- Audios
- Transcripts

# 8 CLOUD PROVIDERS COST ANALYSIS

In order to continue the development, we must analyze the API alternatives for the speech to text recognition.

## 8.1 AWS TRANSCRIBE [2]

Amazon Transcribe API for batch transcriptions is billed monthly and varies by region, although the price is the same for loads under 5 million minutes.

*Table 1: AWS transcribe pricing*

| TIER | MINUTES/MONTH | $/SECOND | $/MINUTE |
|---|---|---|---|
| **T1** | First 250,000 minutes | $0.00040 | $0.0240 |
| **T2** | Next 750,000 minutes | $0.00025 | $0.0150 |
| **T3** | Next 4,000,000 minutes | $0.00017 | $0.0102 |

## 8.2 AZURE COGNITIVE [3]

Azure follows a slightly different pricing model and varies based on the number of channels and model customization.

*Table 2: Azure pricing*

| TYPE | PRICE |
|---|---|
| **STANDARD** | $1 per audio hour |
| **CUSTOM** | $1.40 per audio hour |
| **MULTICHANNEL AUDIO** | $2.10 per audio hour |

## 8.3 GOOGLE HELLO [4]

Google follows Amazon on the pricing with an almost identical table that also applies to minutes of audio processed per month. The difference is that Google varies pricing based on three factors: Standard/Enhanced Model, data logging and the number of channels in the audio.

*Table 3: Google Hello pricing*

| FEATURE PER CHANNEL | STANDARD | | ENHANCED (VIDEO, PHONE CALL) | |
|---|---|---|---|---|
| **MINUTES** | **0-60** | **60 - 1M** | **0-60** | **60 - 1 M** |
| **NO DATA LOGGING** | Free | $0.006/15s | Free | $0.009/15s |
| **DATA LOGGING** | Free | $0.004/15s | Free | $0.006/15s |

## 8.4 IBM WATSON [5]

IBM Watson Speech to Text offers 4 different plans but we will focus on the Plus plan, the most basic one after the Lite (Free) plan.

*Table 4: IBM Watson pricing*

| TIER | MINUTES/MONTH | $/MINUTE |
|---|---|---|
| **T1** | First 999,999 minutes | $0.02 |
| **T2** | 1,000,000+ minutes | $0.01 |

## 8.5 SUMMARY

After a careful analysis of the prices for each cloud speech recognition API we can determine that all four providers are completely affordable and will be implemented in the final design. Their average price is summarized in the following table.

*Table 5: Average pricing per cloud*

| AWS | Azure | Google | IBM |
|---|---|---|---|
| **$1.45/h** | **$1/h** | **$1.45/h** | **$1.2/h** |

This means that a 3-credit course (45h) could have transcriptions for all of its classes at a price around $45 -$65**.**

# 9 DESIGN

The design is based on IBM's definition of a cloud-based architecture [6]. A traditional cloud-based application involves many components such as load balancers, web servers, application servers and databases. Our design is divided into 5 main components, each of them with a very unique and necessary role.

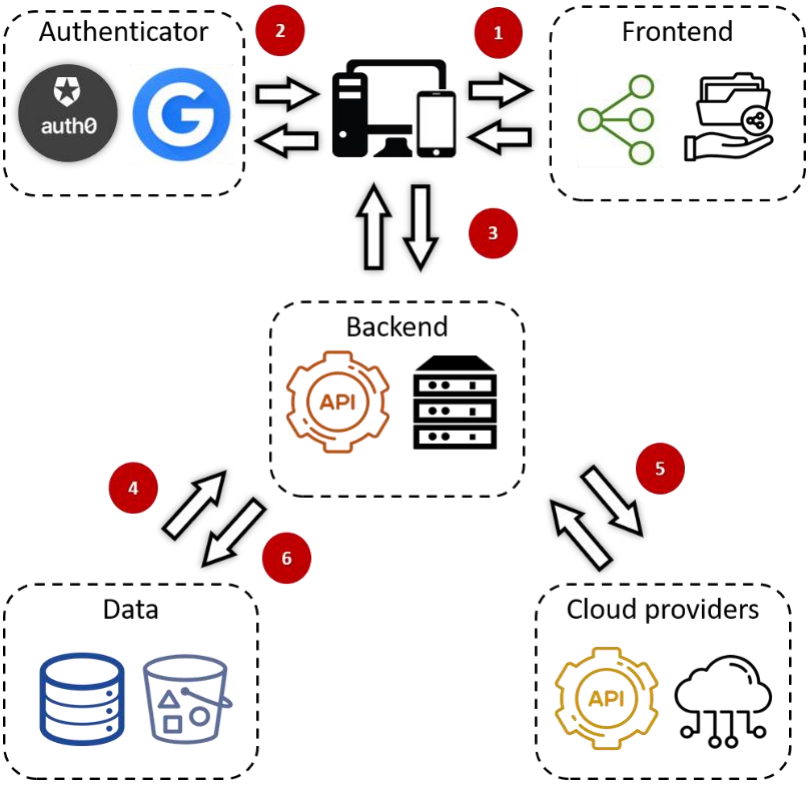


*Figure 2: Application architecture design*

## 9.1 FRONTEND

The first step to use the application is to access the URL through a browser with a JavaScript runtime environment (Firefox, Edge, Chrome, etc.). The web browser will download the index.html of the application and from there it will start fetching the resources that it needs to load all of the functionalities. The design follows a single page application (SPA) structure. The user interface will be interactive and the web page will dynamically rewrite to the screen the data needed for each interaction. This differs from a traditional web page that would load a new entire page for each interaction.

The files are served statically via a load balancer. This means that the only running process is that of the server/load balancer since the actual application runs in the client's machine.

## 9.2 AUTHENTICATOR

After the application has been loaded in the client's browser, there will be an authentication screen. This is another of the main components of the application. If the user is not properly authenticated, the user will not be able to use any of the applications functionalities. For logging in, the application will redirect the user to the authenticator's domain, Google in this case, and

the user will be able to sign in to its Google account like it normally does with Gmail or YouTube. After successful login, google will automatically return the user to the app's main page.

This component consists of two different parts:

- The SDK that allows to show the login screen, redirect to Google and get back with the user's information for authentication.
- Google's authentication API, which is totally external to the application and is provided for free by Google.

## 9.3 BACKEND

Once the user is authenticated, the user interface will offer actions that involve the application's backend. The backend is the component in charge of all the business logic. It performs 3 essential functions.

- Store and retrieve data from the database.
- Make use of cloud APIs.
- Exchange data with the frontend.

From the initial diagram we can clearly see that everything is interconnected through the backend.

## 9.4 DATA

All de data is persisted across the two different databases. It's the most valuable component of the design. It holds all the videos and transcriptions as well as all the information that relates users, videos and transcripts. The backend is the only component allowed to access and modify it. It consists of two different parts:

- Object storage.
- Information document storage.

## 9.5 CLOUD PROVIDERS

This are cloud APIs that offer their software as a service. They allow to move the application's computing needs to the cloud. This is what's known as cloud computing. The application sends the videos/audios to the cloud to be processed in a remote datacenter controlled by a third party. The cloud then returns the transcriptions.

# 10 IMPLEMENTATION

## 10.1 CLOUD PROVIDERS

### 10.1.1 Azure Speech to Text, Google Hello and IBM Watson

The backend treats these cloud providers almost identically by following a 4-step process:

1. Using the "FFmpeg" module for Node, the audio is extracted from the video.
2. Using the "fs" module from Node, the audio is loaded into a stream.
3. The stream is sent to the cloud for recognition via an asynchronous job that waits for the result to come back.
4. The result is parsed and saved as text.

### 10.1.2 Aws Transcribe

The backend has 3 different methods that interact with AWS:

- Upload a video to an S3 bucket.
- Request for an S3 video to be transcribed.
- Download and store in local library the transcript from the S3 bucket.

These three methods are used independently by the backed to perform the different actions needed to successfully transcribe a video from our library using AWS.

### 10.1.3 Factor (Mozilla DeepSpeech)

This is our own "cloud" infrastructure. It's based on Mozilla's open-source project for speech to text recognition called DeepSpeech. The backend invokes a child process that is in charge of calling the DeepSpeech process with the appropriate parameters.

- The path to the language model.
- The path to the scorer.
- The path to the audio to transcribe.
- The name of the output file to store the text.

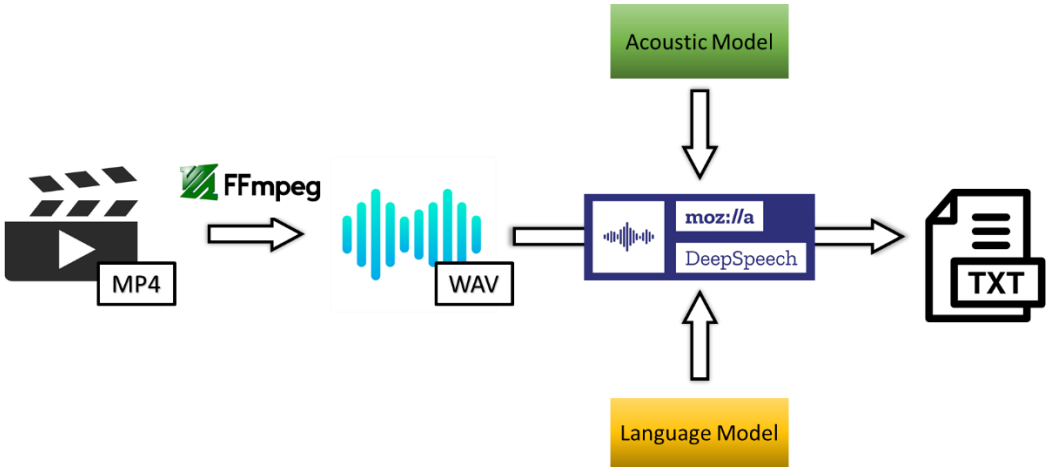


*Figure 3: Backend DeepSpeech process*

## 10.2 FRONTEND

The frontend has been developed with React. Originally developed by Facebook, it's now a really popular open-source front-end JavaScript library for building simple user interfaces. The implementation follows a hierarchical tree of reusable components that creates a really simple single page application.

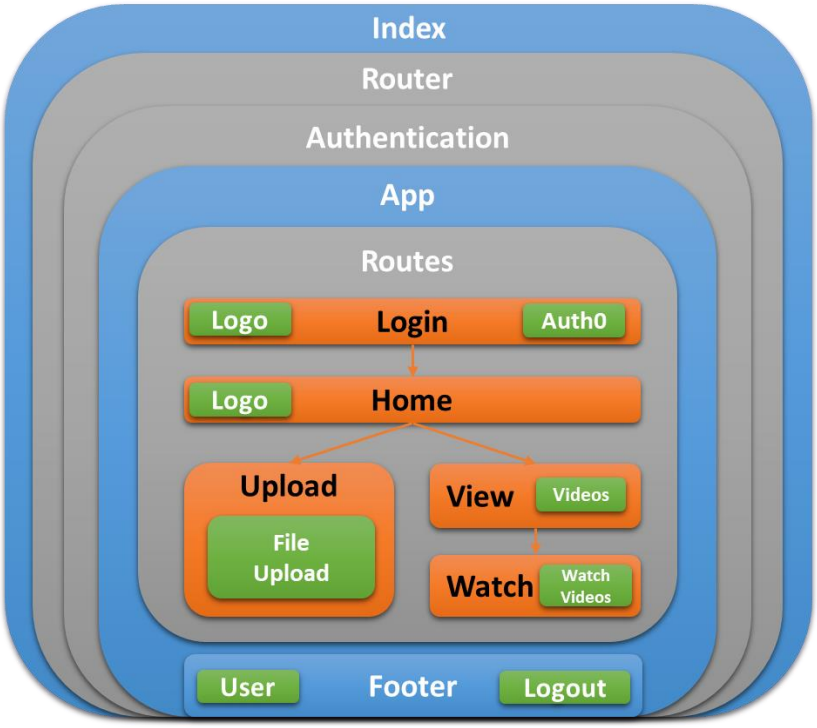


*Figure 4: Frontend hierarchy design*

There are 5 vertical levels before we reach the components that will show the user interface. The Index is the root of the application, it's the component in charge of rendering the root of the html document. It also wraps the App in the Authentication provider and a Router that provides URI management for dynamic routing inside the single page application.

Inside the Routes component, the application is distributed into 5 different components to navigate to.

- Login
- Home
- Upload
- View
- Watch

There is also a footer that will always be available at the bottom of the application. This footer displays either the user and a logout button or a message letting the user know that it is not authenticated. The footer also shows a home button to navigate to the main page.

### 10.2.1 Login

The login component is composed of the application logo and the authentication box, it's the first thing anyone using the application for the first time will see. It's also the default component.

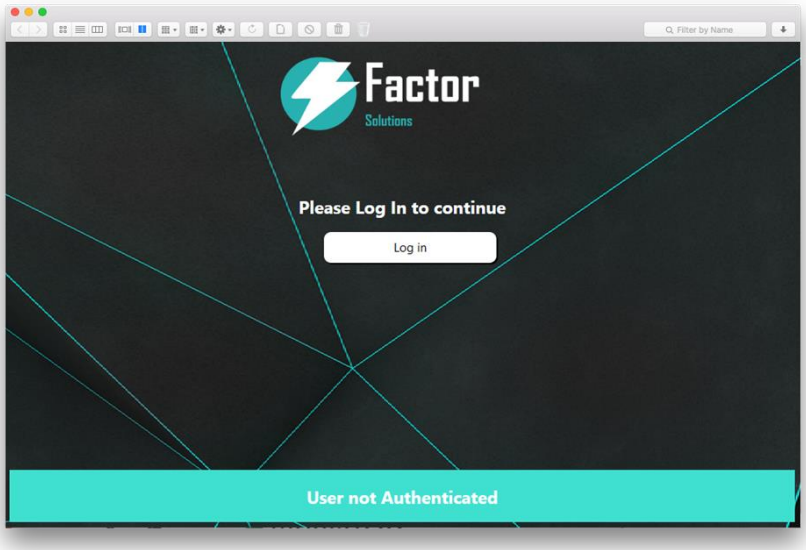


*Figure 5: Frontend login page*

Once the Log In button is clicked, the user is offered to choose from a variety of options. If the user selects Google, it will be redirected to Google's authentication screen and only returned back to the app upon successful authentication.

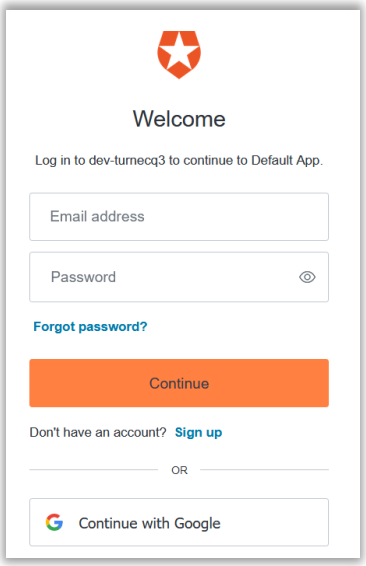


*Figure 6: Frontend authentication page*

This is also where the user's username and email are retrieved for later purposes, using Auth0's react hook.

### 10.2.2 Home

Once the user is successfully logged in with google, the application will show the home screen with two options.

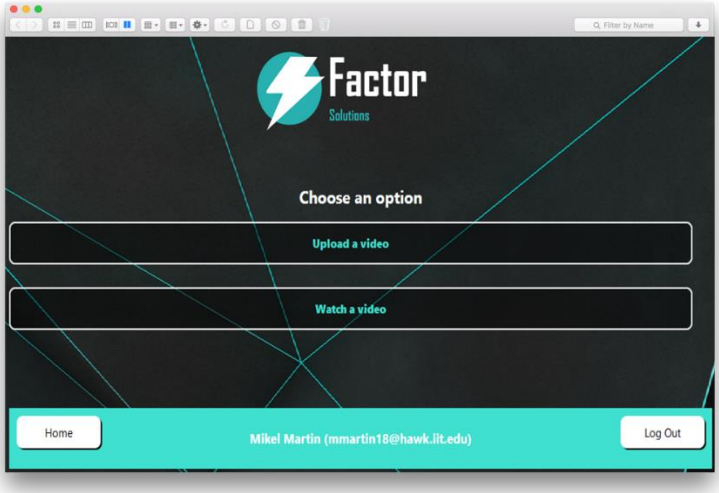


*Figure 7: Frontend home page*

The “Upload a video” option will show the Upload component and the “Watch a video” option will show a list of all the videos available.

### 10.2.3 Upload

The Upload component shows two buttons and a text input. The text input gives the video a title to be saved in the database. The brose button allows the user to select a video from his file system. The upload button sends the video to the backend’s API through a formData.

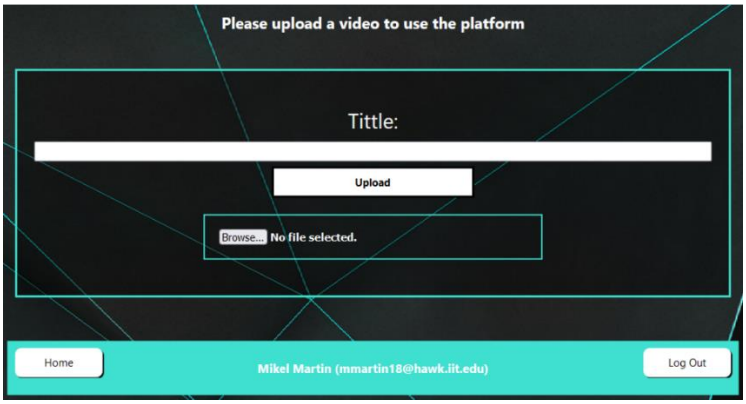


*Figure 8: Frontend upload page*

### 10.2.4 View

The View component shows the list of all available videos in the database library that the user is allowed to watch. This list is retrieved from the backend's API. The blue link can be clicked to access the corresponding video.

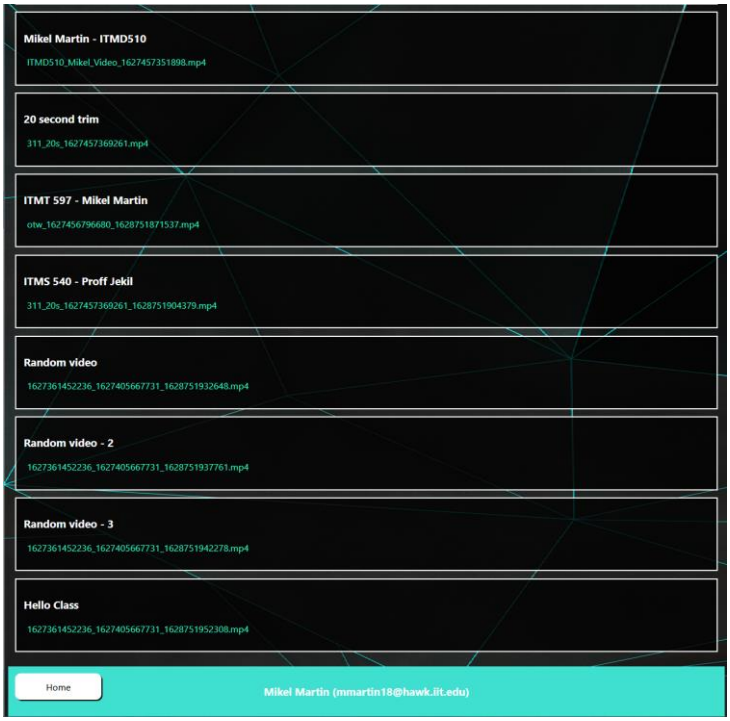


*Figure 9: Frontend video list page*

### 10.2.5 Watch

This is the most important component of the application. It is the page where the user will spend most of the time.

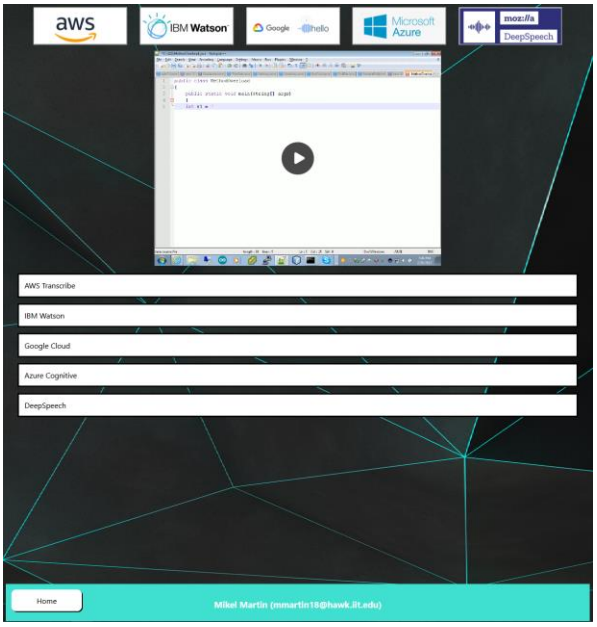


*Figure 10: Frontend video page*

This component is in charge of three critical functions:

1. Displaying the video that wants to be watched. The frontend calls the backend API to retrieve the video that wants to be watched using html5's native functions.

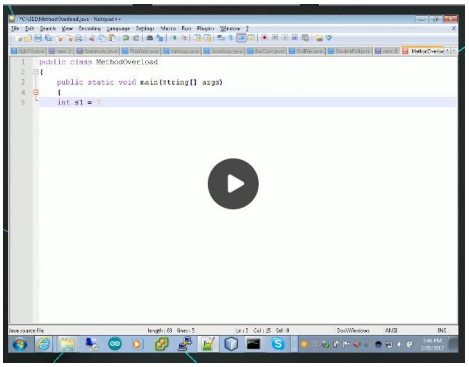

*Figure 11: Frontend video player*

2. Offering the cloud providers that the backend uses to create transcriptions for the video on display. The frontend calls the backend's API indexing the cloud provider and the name of the video.


*Figure 12: Frontend transcriber options*

3. Showing the transcriptions from the library. The frontend asks the backend's API if the video has any transcripts and shows the text for each provider independently.

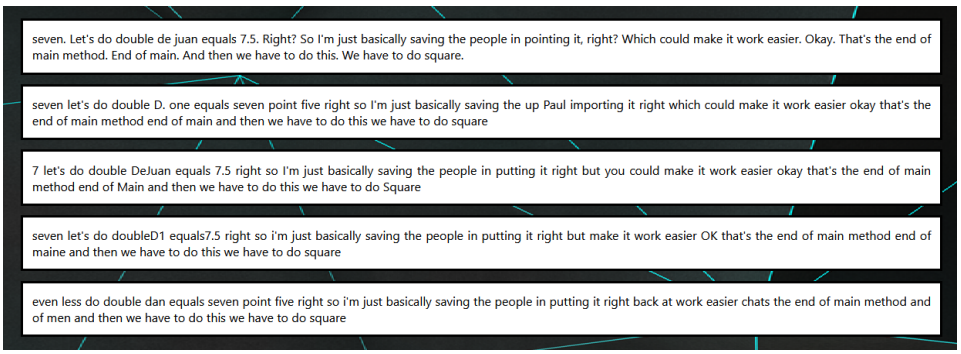


*Figure 13: Frontend transcripts*

## 10.3 AUTHENTICATOR

This is the cloud provider that allows for external login to avoid management of user private information.

### 10.3.1 Auth0

Auth0 is a Seattle based startup, leader in identity authentication software. It's easy to implement and allows to connect almost any modern application to a login cloud provider like Google. It can be setup in their web page and allows for simple user management. It is used to authenticate users with Google via their IIT email. It offers a simple React SDK that allows us to implement user authentication to the project without any necessity to manage users. It offers a

free plan with up to 7,000 active users and unlimited logins, more than enough for the size of the application we are building

### 10.3.2 Google

Google Cloud has an API to authenticate users, accessing their profile information via their Gmail account. It also allows for organizational accounts, like the iit.edu and hawk.iit.edu domains. This is were the application takes the user and email information to show/upload videos.

## 10.4 BACKEND

The backend was built using Node.js, a really popular open-source JavaScript runtime environment that can run outside a web browser.

### 10.4.1 Express

Express.js is a backend web application framework for Node.js. It is a free and open-source software under the MIT License. Originally designed for building web applications and APIs. It has been called the de facto standard server framework for Node.js. We used this technology to create an express server with all the APIs for our backend.

### 10.4.2 Videos APIs

*Table 6: Backend video APIs*

| NAME | METHOD | URI |
|---|---|---|
| **GET A LIST OF THE VIDEOS** | GET | /videos/ |
| **GET A VIDEO** | GET | /videos/video/ ***{id}*** |
| **UPLOAD A VIDEO** | POST | /videos/upload |

### 10.4.3 Transcripts APIs

*Table 7: Backend transcript APIs*

| NAME | METHOD | URI |
|---|---|---|
| **GET A TRANSCRIPT** | GET | /transcript/ ***{api}*** / ***{id}*** |
| **TRANSCRIBE VIA AWS** | GET | /transcribe/aws/ ***{id}*** |
| **TRANSCRIBE VIA DEEPSPEECH** | GET | /transcribe/ds/ ***{id}*** |
| **TRANSCRIBE VIA IBM** | GET | /transcribe/ibm/ ***{id}*** |
| **TRANSCRIBE VIA GOOGLE** | GET | /transcribe/gc/ ***{id}*** |
| **TRANSCRIBE VIA AZURE** | GET | /transcribe/az/ ***{id}*** |

## 10.5 DATA

### 10.5.1 Database

The application uses MongoDB for storing all the video related information. MongoDB is a document-oriented database. Classified as NoSQL, it uses JSON-like documents to write variable schemas. Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment. The backend, using the “mongoose” module for Node, sends documents to this database with the following schema:

- Owner: the name of the uploader
- Email: the email of the uploader
- Title: the title of the video
- File Name: the file name (id)
- File Path: the URI to the video
- Transcript Path: the name and URI to each transcript.
- Date: Date stamp

### 10.5.2 File storage

There are three types of files to be stored: videos, audios and transcripts. This will be stored in the following hierarchy:

- Assets:
    - Videos
    - Audios
    - Transcripts
        - AWS
        - DS
        - IBM
        - AZ
        - GC

# 11 DEPLOYMENT

The deployment of the application was, as initially stated in the requirements, deployed using Docker. Docker is an open platform for running applications in loosely isolated environments called containers. It allows applications to be reduced to small sized packages that run seamlessly in any docker engine deployed. This makes the application portable and compatible with any system. In order to deploy the application correctly there are a couple of steps that need to be performed. First, the code needs to be compiled. Second, the compiled files need to be dockerized into Docker images and finally, the docker images need to be run in containers.

## 11.1 PRODUCTION BUILD

As mentioned, the first step is the building process. This is independent from Docker, and it's about compiling the code that will afterwards be run in the appropriate environment.

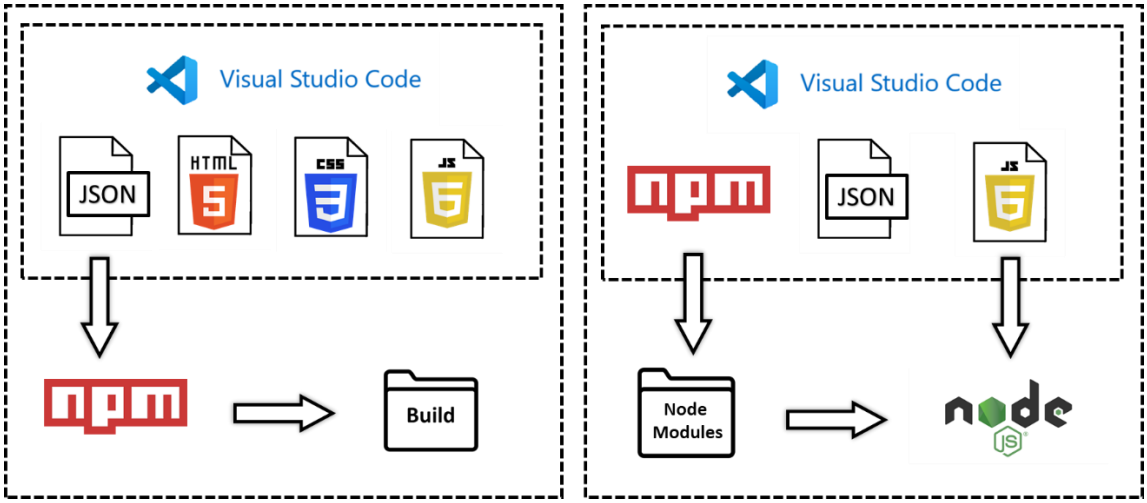


*Figure 14: Deployment building phase*

As we can see, the code was developed using Visual Studio Code. The backend and the frontend were both built using Node's Package Manager but their process was different so they will be explained separately.

### 11.1.1 Frontend

The frontend is composed of 4 different types of files.

- JSON files contains instructions for NPM about building the application and installing dependency packages.
- An HTML file that contains the structure of the React page.
- Two CSS files that describe the aspect of the components of the page.
- All the JavaScript files that describe React components' behavior.

NPM, using all of these files, can create a folder called build with all the compiled files that can then be served statically by a load balancer as the frontend.

### 11.1.2 Backend

The backend is only composed of JSON and JavaScript files. NPM in this case is only used to create the dependency folder called Node Modules that will then be available, along with the source code, for NodeJS to run the backend.

## 11.2 Docker Build

In this stage, we use "Dockerfiles" to create Docker images. A "Dockerfile" is a text file that Docker reads from top to bottom. It contains a recipe for building images specifying Linux commands and a few other docker specific directives.

### 11.2.1 Frontend

The recipe for the frontend is simple. Taking a NGINX base image, we copy the build folder into the container in the right directory and we add a configuration file for NGINX to serve those files. Finally, we expose port 80 for http to be accessible.

### 11.2.2 Backend

The recipe for the backend has some more complexity. First, we need to install some dependencies used by child process. This includes:

- Python for running DeepSpeech.
- FFMPEG for audio extraction.
- Sox for DeepSpeech.
- DeepSpeech itself.

We also need to copy the source code into the container, along with the node modules. After creating a volume for the asset storage, we can expose the API port.

## 11.3 Docker Compose

The last step of the deployment is to define a docker compose to run the images in containers. Compose is a tool for defining and running multi-container Docker applications. With Compose, we can use a YAML file to configure all the application's services. For this deployment we need to define the services and the networking between them.

There are 4 different services:

- Frontend – NGINX instance.
- Backend – NodeJS instance.
- Database – MongoDB instance.
- Database user interface – ExpressJS instance.

As we can see, there are three different networks.

- Frontend: connects the frontend and the backend.
- Backend: connects the backend and the database.
- DB: connects the database and the database's management interface.

We can also appreciate 3 environment files, 2 volumes and 2 ports exposed.

## 11.4 DOCKER RUN

The final step is to run the docker compose in a Docker engine, achieving the final step of our application deployment. Our application is now running and accessible on port 80. In the following image we can see a final overview of the whole process.

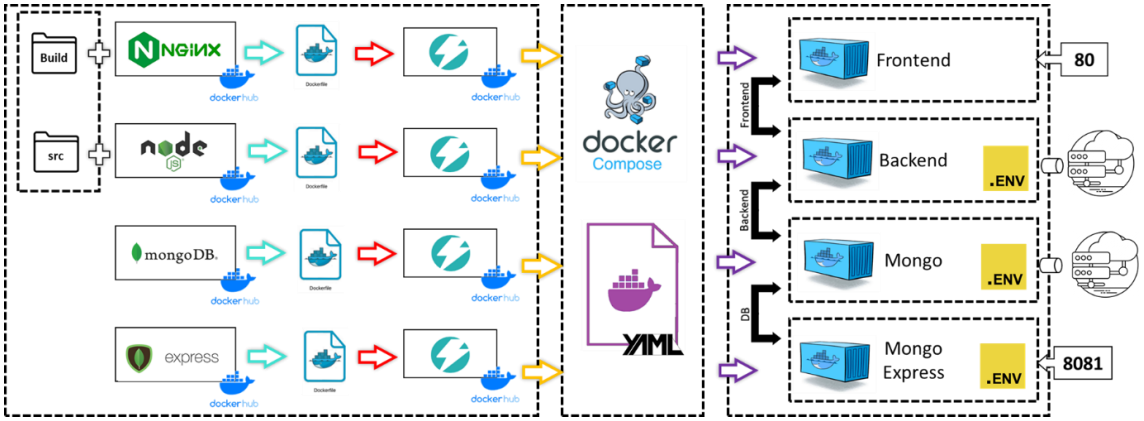


*Figure 15: Deployment overview*

# 12 WORK PLAN

The project was divided into 6 unique workloads and distributed along 5 months.

| TimeLine | March | | | | April | | | | May | | | | June | | | | July | | | | T (h) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | W1 | W2 | W3 | W4 | W1 | W2 | W3 | W4 | W1 | W2 | W3 | W4 | W1 | W2 | W3 | W4 | W1 | W2 | W3 | W4 | |
| **PT0 Start and familiarizatior** | **4 weeks** | | | | | | | | | | | | | | | | | | | | **24** |
| J001 Project Definition | 4 | 4 | 4 | | | | | | | | | | | | | | | | | | 12 |
| J002 Stablish Objectives | | | 4 | 4 | | | | | | | | | | | | | | | | | 8 |
| J002 Overview | | | | 4 | | | | | | | | | | | | | | | | | 4 |
| **PT1 Analysis** | **4 weeks** | | | | | | | | | | | | | | | | | | | | **55** |
| J101 Check requirements | | | | | 5 | | | | | | | | | | | | | | | | 5 |
| J102 Set up enviroment | | | | | 10 | 10 | | | | | | | | | | | | | | | 20 |
| J103 Up and running | | | | | | 10 | | | | | | | | | | | | | | | 10 |
| J104 Testing | | | | | | | 10 | 10 | | | | | | | | | | | | | 20 |
| **PT2 Specifications** | **3 weeks** | | | | | | | | | | | | | | | | | | | | **20** |
| J201 Set improvements | | | | | | | 5 | 5 | | | | | | | | | | | | | 10 |
| J202 Set priorities | | | | | | | | 5 | | | | | | | | | | | | | 5 |
| J303 Set requisites | | | | | | | | | 5 | | | | | | | | | | | | 5 |
| **PT3 Study Alternatives** | **2 weeks** | | | | | | | | | | | | | | | | | | | | **40** |
| J301 Check Alternatives | | | | | | | | | 10 | | | | | | | | | | | | 10 |
| J302 Evaluate Alternatives | | | | | | | | | | 10 | | | | | | | | | | | 10 |
| J303 Stablish a design | | | | | | | | | | 20 | | | | | | | | | | | 20 |
| **PT4 Build** | **9 weeks** | | | | | | | | | | | | | | | | | | | | **305** |
| J401 Implement design | | | | | | | | | | | 25 | 25 | 25 | 20 | 20 | 10 | 10 | 5 | 5 | | 145 |
| J402 Testing | | | | | | | | | | | | | | 20 | 20 | 10 | 10 | 5 | 5 | | 70 |
| J403 Fixing | | | | | | | | | | | | | | | | 20 | 10 | 5 | 5 | | 40 |
| J404 Deploy to enviroment | | | | | | | | | | | | | | | | | 10 | 20 | 20 | | 50 |
| **PT7 Management** | **20 weeks** | | | | | | | | | | | | | | | | | | | | **125** |
| J701 FollowUps | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 20 |
| J702 Documentation | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 10 | 10 | 10 | 20 | 40 | 105 |
| **Weekly hours:** | 6 | 6 | 10 | 10 | 17 | 22 | 17 | 22 | 17 | 32 | 27 | 27 | 27 | 42 | 42 | 51 | 51 | 46 | 56 | 41 | **569** |

The first month was used for meetings and project definition, familiarizing with the original project and defining the objectives for this project.

The second month was mainly used for launching the original project, analyzing it, defining improvements and setting the requirements for this project.

The third month was used for checking alternatives, setting specifications and defining a base design for the project.

Most of last two months were focused on implementation, deployment, testing and troubleshooting. The last couple of weeks were more related to documentation.

# 13 CONCLUSION

This project ended up achieving all the objectives that were set. The application fully runs in Docker and offers professors the option to upload videos and transcribe them with 5 different recognition technologies. It also offers any student access to the data and the ability of watching any video with its corresponding transcripts. The problem of not being able to read at your own pace or take your time to understand the concepts is now solved. Anybody can access the text at any time for their own study purposes, without the need of going through the whole video to try and find the information they are looking for. Moreover, this solution not only offers benefits for students, it also opens the possibility for hundreds of other applications. From helping professors create books, to helping students with hearing problems access the traditional school system.

# 14 REFERENCES

[1] G. Moore, "Cramming More Components onto Integrated Circuits," *Electronics,* vol. 38, no. 8, p. 114, 1965.

[2] AWS, "Amazon Transcribe Pricing," [Online]. Available: https://aws.amazon.com/transcribe/pricing. [Accessed 15 July 2021].

[3] Microsoft, "Cognitive Speech Services Pricing | Microsoft Azure," [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/cognitive-services/speech-services/. [Accessed 15 July 2021].

[4] Google, "Pricing | Cloud Speech-to-Text | Google Cloud," [Online]. Available: https://cloud.google.com/speech-to-text/pricing. [Accessed 15 July 2021].

[5] IBM, "Watson Speech to Text - Pricing," [Online]. Available: https://www.ibm.com/cloud/watson-speech-to-text/pricing. [Accessed 15 July 2021].

[6] Ravi Saraswathi, IBM Chief Architect, "IBM Blog," 6 Janurary 2020. [Online]. Available: https://www.ibm.com/cloud/blog/four-architecture-choices-for-application-development.