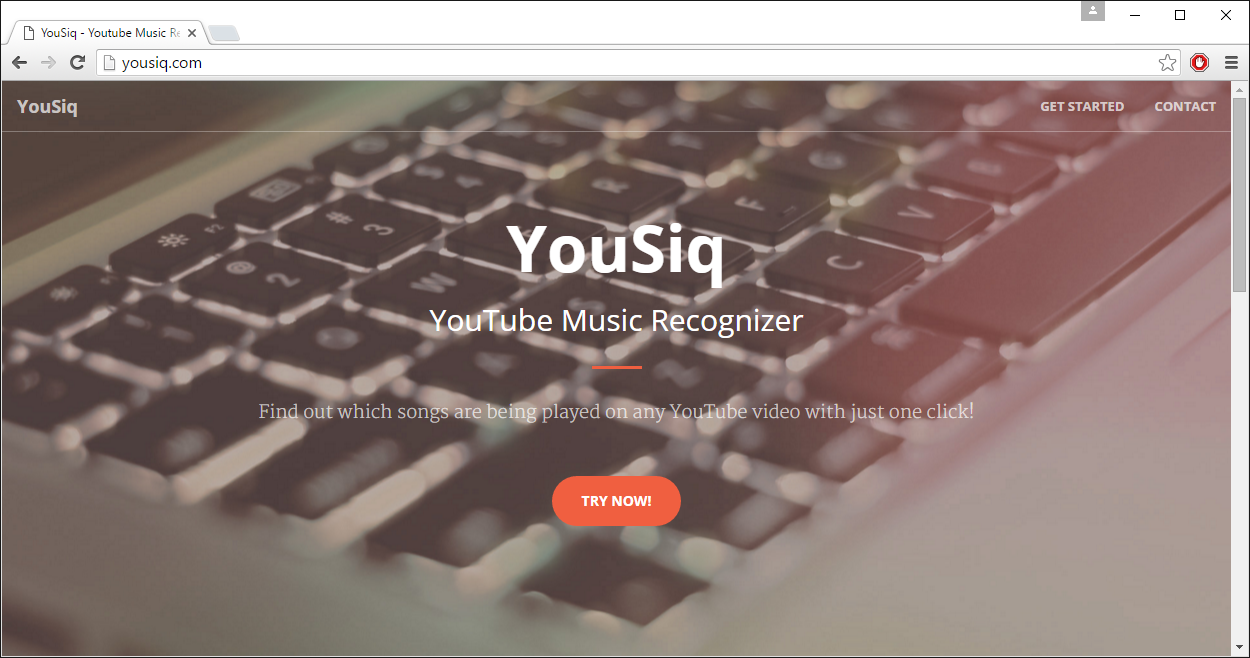
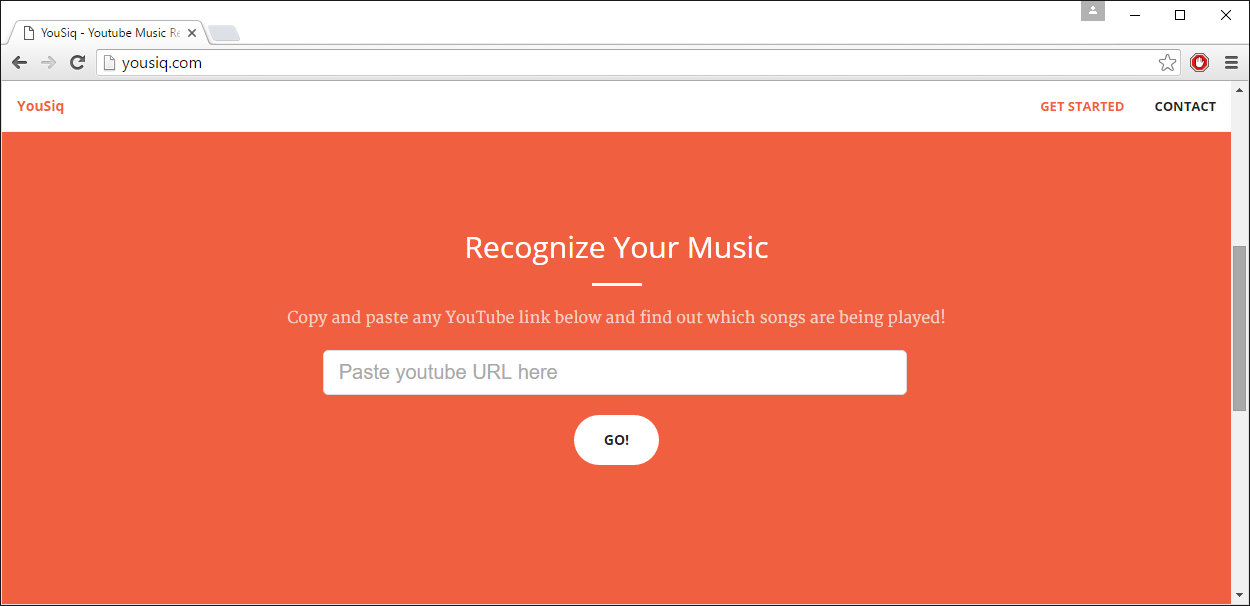
**YouSiq**

# 1. Introduction

YouSiq is a lightweight, simple, and user-friendly web application which offers YouTube music recognition service. Copy and paste any YouTube URL, and watch the video while the music in the video is being analyzed.

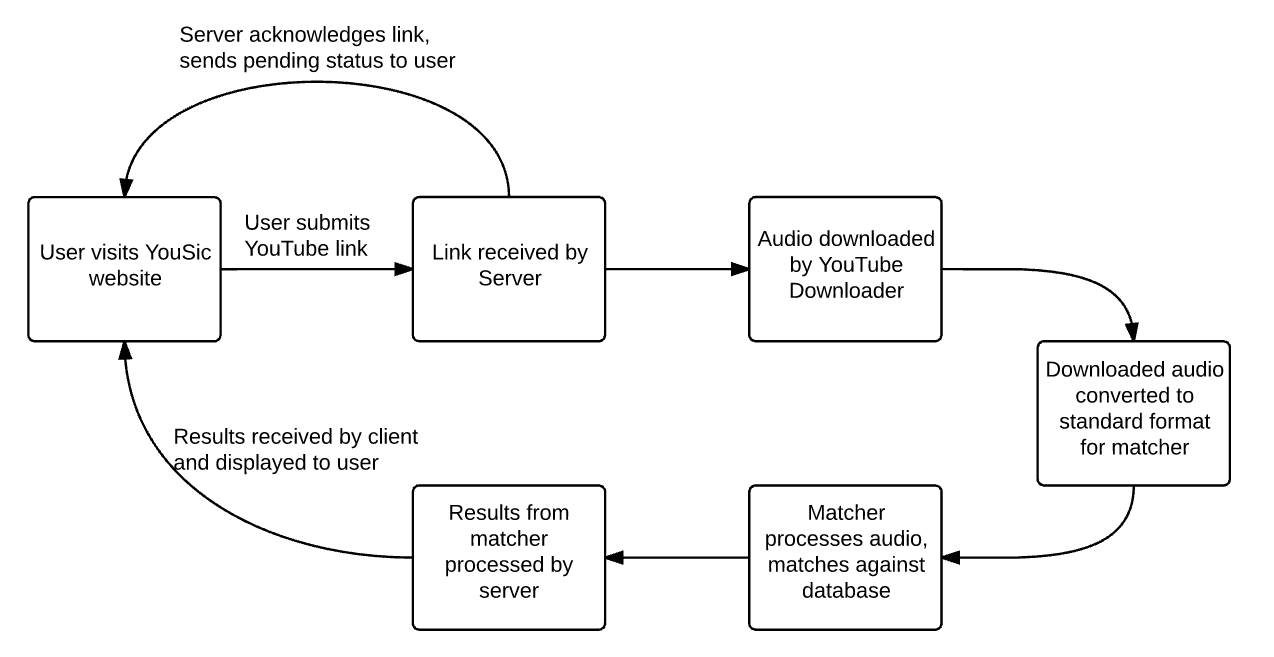
A working demonstration is available at <http://yousiq.com/>





# 2. Implementation

The figure below show the high-level flowchart as to how YouSiq works.



We will describe our implementation in greater detail in 2 parts. The first part consists of the client/server, where we will show how we deliver the music detection service to the user. The second part consists of the music detection itself; here we will have an in depth discussion of the of the algorithm used.

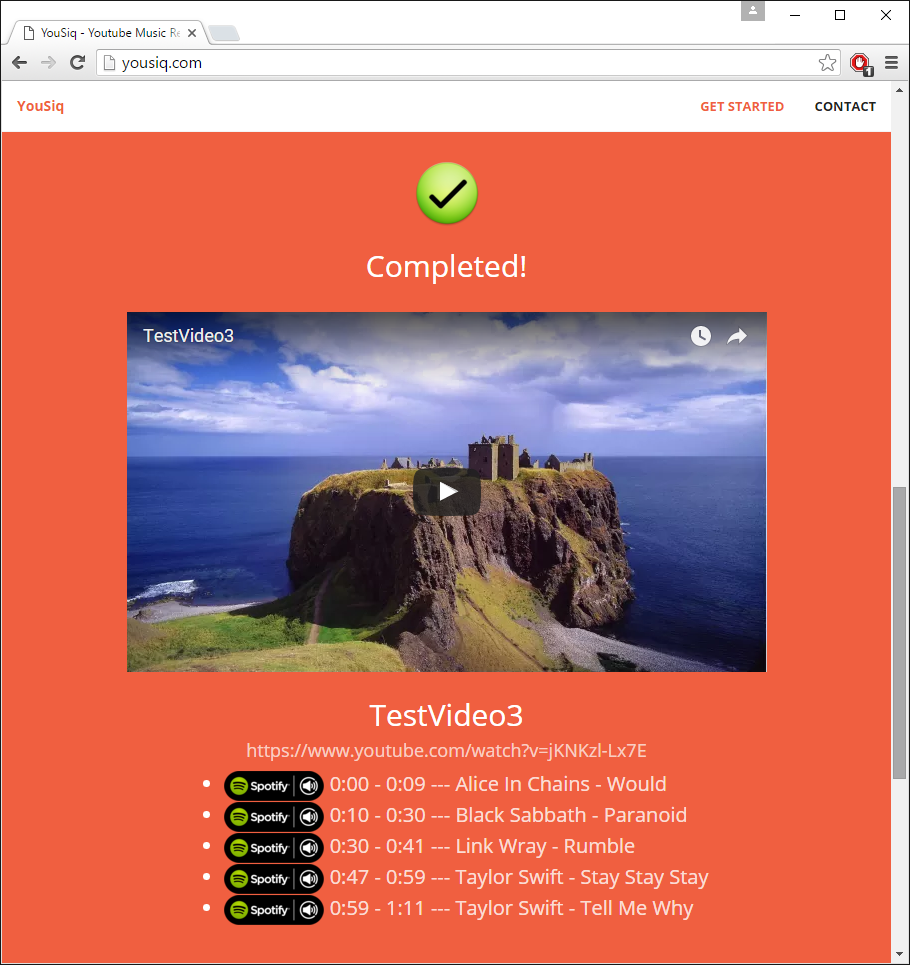
## 2.1. Client & Server

YouSiq uses Node.js as its web application framework. On top of that, it also uses 3 other JavaScript frameworks namely express, socket.io, and ytdl-core. They are available through the NPM (Node Package Manager) module. Express is used to handle common HTTP requests and responses with the respective webpages to the user. Socket.io is used for client-server two-way communication.

Node.js, the runtime JavaScript engine, runs on the server persistently. When a user visits our website, the client (browser) will send a request to the server and it will obtain the website layout via the express framework. User is then able to copy and paste the desired YouTube URL into the box provided and submit. The client will send the YouTube URL to the server via the Socket.io framework, and the server will process the request.

The server will use the ytdl-core (YouTube downloader) library, specifying the format and the quality of the youtube video (AAC 128-bit audio only). YouTube video information such as the video ID and title are extracted from the URL, sent to the client, and shown to the user on the YouTube video player embedded in the web application. The user can watch the video while waiting for the video to be processed and analyzed by YouSiq. Once the YouTube video (just the audio) is successfully downloaded, ffmpeg (an audio/video encoder/decoder application) is used to convert the downloaded audio into a Wave (PCM) file, sample rate of 44100 Hz, with a single mono channel. The audio is cached in local directory, such that downloading from YouTube is not required if any other users submit the same YouTube URL.

The converted wave file is then passed to the Music Detector (Matcher), and the server will wait for the Matcher to return relevant songs results with their respective timings. Once the Matcher has finished running, the server will send the results to the client, showing the music titles present in the specified youtube video. The Matcher will be discussed in detail in the next section.



## 2.2. Music Detection (Matcher)

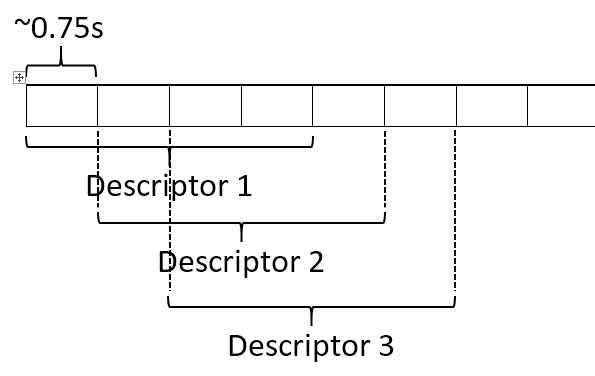
The core of YouSiq is simply referred as the matcher. It is written in Java, and is invoked and managed by the node.js server. It takes 2 file paths as its runtime arguments - a path to the database file containing the descriptors of all files in the song database, and the path to the query song (called the sample).

The input song has been (and has to be) standardized to a Wave (PCM) file, sample rate of 44100 Hz, with a single mono channel. Similarly all the songs in the database have been converted to the said format before being indexed.

### 2.2.1. Descriptors

The songs are represented a list of MFCC descriptors. Each MFCC descriptor is a list of 12 double values, and each descriptor represents approximately 2.972 = ~ 3 secs worth of audio samples (this value is explained later). There is a descriptor for every 0.743 = ~0.75 secs of audio, thus forming a “rolling window” of descriptors.

A visual illustration is given below. The chain of boxes represent the song, while each individual box represents ~0.75 secs of samples. The first descriptor represents the first ~3 secs of samples, while the second descriptor starts ~0.75 secs after the first (up to ~3.75 secs mark). The third starts ~0.75 secs after the second, and so on.



The size/length of the descriptors are of the given values due to the MFCC algorithm’s nature. For each window (which produces one descriptor), the algorithm accepts number of samples in the power of two, hence 217 = 131072 samples was chosen, with a rolling offset of 215 = 32768 samples. As all audio input has been standardized to a sampling rate of 44100 Hz, the result is a window size of 131072/44100 ~= 2.972 and an offset of 32768/44100 ~= 0.743 samples. A song will have *[Total Number of Samples]*/32768 descriptors.

For clarity sake, we will refer to each window as having size 3 seconds, and each offset at 0.75 secs between each window; however the actual implementation works with the samples directly, so no lost of precision is obtained.

### 2.2.2. Song Database

All the songs in the database would have their MFCC descriptors precomputed and stored inside a database. Each song is represented by a 2-dimensional array of double values, the first dimension is of size *[Total Number of Samples]*/32768 - the number of descriptors in the song, and the second dimension always of size 12 (number of double values a descriptor has).

The total database is stored in a single file (approximately 30 MB for our database of 900 songs), while the original audio files are discarded.

### 2.2.3. Matching, Flattening, Pruning

Deciding what song appears in the given sample is a 3 step process. The first two process is for each song, while the third sample works on all songs that has been detected to be in the sample.

The first is the matching processes, where we determine if a part of song appears within a part of a sample. This will produce a list of results, which then goes through the flattening process, where individual results are combined into a single result as much as possible.

Thereafter, after each song has gone through the first two processes, the entire list of results (results with all songs) goes through pruning. Here the song with the highest similarity score are chosen, and songs with poorer scores discarded. We will describe each process in detail below:

### 2.2.4. Matching Process

Before we begin, we will define the following:

The list of descriptors of a song is denoted by n, composed of n1, n2, n3…

The list of descriptors of a sample is denoted by m, composed of m1, m2, m3…

The size of the list of descriptors of a song is denoted as j.

The size of the list of descriptors of a sample is denoted as k.

The matching process works as such: we are given a list of descriptors of a song and that of a sample. We first calculate the cosine distance between n1 and m1. If the cosine distance between the two is within a threshold (defined as a similarity value of 98.5%), we accept this two descriptors as matched. We then continue the attempt to match linearly down the two lists (i.e. n2 with m2, followed by n3 with m3), accepting as a match if the cosine distance is above the said threshold (98.5%). If we find a mismatch (i.e. similarity < 98.5%), we stop.

At this point, we count the number of matches, and the average similarity score of all the matches. If the number of matches is larger than a second threshold (12 matches), and the average similarity score above a third threshold (99.3%), we accept this set of matches as a *result*. Thus a result is simply a reference to the location of a sub-list descriptors on both the song and the sample where all 3 thresholds are met.

We then repeat this entire process using each pair of song-descriptor and sample-descriptor as the starting point. E.g. We use n1, m2 as a starting point; then n1, m3; then n1, m4; later n2, m1; then n2, m2; then n2, m3; *etc.* Thus overall we will have j \* k starting points.

### 2.2.5. Flattening Process

The number of results produced by the matching process is huge, often with overlapping results. Hence the aim of this process is to combine the overlapping (and consecutive) results into one single result. Often a list of 1000+ results can be reduced just a few (even singular) results.

The workings are as such: first, the (average) similarity score of the highest result is chosen to set a fourth threshold - any result with a similarity score 0.004% smaller than the highest similarity score is discarded. For example, if the highest similarity score is 100%, any result with a similarity score below 99.6% will be discarded.

Subsequently, for each descriptor location of the sample, the highest obtained similarity score will be assigned. Locations where no results are obtained are simply given a score of 0.

Following, a scan along the descriptors is done, where consecutive descriptors that are matched (i.e. non-zero) will collectively form a single result.

Finally, if two results are separated by 3 descriptor locations or less (i.e. are only at most 0.75\*3 = 2.25 secs apart), we combine the two results into one.

### 2.2.6. Pruning Process

The pruning process considers all results from all songs. The principle is simple: the results with the highest similarity scores are selected first. If there are overlaps between two results, the result with the lower score is simply truncated.

If a result is truncated to a length less than the second threshold mentioned in the matching process (i.e. the number of match is less than 12), it is discarded. This translates to if the result is less than 12\*0.75 = 9 secs, it is discarded.

Another produce of this behaviour worth mentioning is also if a result with a higher score encompasses the entire range of a result with a lower score, the lower score result will be discarded completely.

### 2.2.7. Output of Results

Finally, after all 3 processes, the results are printed to standard output in a pre-defined format. The output are in the form of minutes and seconds, thus the results will be rounded to the nearest second. (We note that it is only at this point does any loss of precision with regards to timing occurs.) The output is read by the Node.js server as mentioned in the section 2.1.