ytdl

Asynchronous YouTube downloader

Sam Machiels

26 January 2016

Table of Contents

Preface

This document aims to describe the *ytdl (*YouTube downloader) application, in a way people with little programming experience can follow along, but it also covers technical aspects which are better understandable for people with a programming background. To be more specific, this document covers the design stage of *ytdl,* development and maintenance of *ytdl, ytdl* its releases and the future of *ytdl*. During the development of *ytdl,* a lot of bumps were hit whilst on the road, most of these bumps will be discussed in this document. These bumps often lead to discoveries which are beneficial to me as a programmer. At first I thought it'd be rather easy to develop a YouTube downloader, like *ytdl* . But it turned out it'd be harder than what I expected. What I was expecting was that it'd be as easy as scraping out a URL from the YouTube source code and to download the data served on that URL. Instead, I found out that Google is trying to make it hard to automate the task of downloading media data by obfuscating their source code and by suffixing media URLs with signatures that need to be deciphered in order not to get a 403 (forbidden) HTTP response when requesting a download.

Introduction

Ytdl is an application with which it's easy to download binary media data from Google it's media servers. *Ytdl* has been set up in a way that makes it easy to build upon. Take for example a browser extension which inserts a download button in the YouTube it's document object model (DOM) so it's easy to initiate a download from the YouTube website itself, or a web server that initiates and handles downloads across different machines within a (local) distributed network.

For the development of *ytdl* cutting edge technologies are used, such as the latests features in development for the JavaScript programming language, these features are planned to be implemented according to the ECMAScript 2016/2017 specifications. The cutting edge JavaScript gets compiled to the ECMAScript 5 version of JavaScript, which can run on the simple and flexible Node.js platform. Also, since the entire application is written in JavaScript, it's easy for developers to contribute to the development of *ytdl*.

Ytdl runs most tasks it performs asynchronous, so that for example a web server that handles download requests can be scaled up in performance because the load can easily be balanced. Because of the way *ytdl* its modules are set up it's even possible to handle a single download across a distributed network. Think for example about a service which allows customers to run heavy computations remotely. Scalable and distributable software is the standard nowadays, *ytdl* was developed with that in mind.

It was possible to download media data from YouTube in a way which showed that *ytdl* its footprint on download times was small, in comparison to the time it takes to download the media data (network dependent) and convert the media data (CPU and 3rd party software dependent). This shows that *ytdl* itself can be deployed in an environment in which it has to handle a lot of downloads simultaneously.

Application development

In the following chapters the development stage of *ytdl* is discussed, which technologies are used to develop *ytdl* and why certain tools are chosen. A list of interesting moments during development is shown and discussed.

Time table

- 25/12/16 Application structure and API
- 28/12/16 - Decipher prototype
- 3/1/16 - Beta release
- 12/1/16 - Implemented beta release feedback
- 30/1/16 - Stable release, APIs frozen, feedback handled
- 15/1/16 - Documentation

Development stage and feedback

After the application development environment was set up, the first part of *ytdl* that was developed was its API. The API was developed first because it had previously been written out on paper so that the development of specific modules would not result in a change of the interface with which one can interact with the *ytdl* application its modules. Once the API was set up, the first modules were implemented according to the *ytdl* design, which were simple ones. The interesting stuff started when *ytdl* was developed to a point that media data without an enciphered signature could be downloaded and media data with an enciphered signature could not be. Some research was being done about the location of the deciphering logic its location, it was located in the JavaScript file responsible for YouTube its video player, {*some_sort_of_id}player.js*. After some further investigation it became clear there is a pattern in the *player.js* code, some object was being checked for the presence of two properties, *sig* and *s*: a.*sig*||*a.s.* After the discovery of the pattern, it was just a matter of tracking what happened if the if statement would be evaluated to true. Once the media with an enciphered signature could be downloaded some final patches were made and a beta version of *ytdl* was released.

The beta release of *ytdl* resulted in usable feedback. It was clear that most users agreed with the fact that *ytdl* is set up in a way that it is an audio (mp3) downloader, rather than an application with which videos could be downloaded as well. Also *ytdl* its API received positive feedback. Some code fixes and upgrades were proposed and implemented. After some final feedback implementations and fixes, *ytdl* its API was frozen and a stable release was released.

Technology

For the development of *ytdl,* cutting edge technologies are used, of which the most notable are discussed below along with motivations for using the tool.

ffmpeg

In order to handle media data conversions, ffmpeg is used, it's fast an easy to use. Child processes are spawned when downloaded media data has to be converted with ffmpeg.

Node.js

The *ytdl* application runs on the Node.js platform, version 5.6.0 and up. Node.js was chosen as a target platform because it runs JavaScript blazingly fast with the V8 JavaScript engine. Because Node.js (JavaScript) is really flexible, *ytdl* could be written with abstract code. The abstract code is easy to reason about, so it's easy to keep track of what is going on whilst developing and maintaining the *ytdl* application.

ECMAScript 2016/2017

Node.js was also chosen as a target platform for *ytdl* because it runs the JavaScript that is generated by the *Babel.js* compiler, JavaScript to JavaScript compiler discussed below. Some features which are used extensively are discussed below.

Babel.js

Babel.js is a tool which compiles next generation (2016/2017) JavaScript to JavaScript (2015). This makes sure developers can test upcoming JavaScript features so they can pass feedback to the technical comities who standardize and propose new features. Of course it can also be beneficial to programmers to use features that are not yet (fully) standardized/implemented because they may allow programmers to write more abstract code. *Babel.js* is fully configurable, developers get to choose which specific features of next generation JavaScript they want to be compiled. *Babel.js* also has a *gulp* plug-in, which makes the usage of *Babel.js* even more pleasant.

Promises

A promise in programming terms is a simple concept based on the principle of making a promise that something is going to happen in the future, it's great when working on asynchronous code. For example: a function called *httpGet* which promises that it will send an HTTP get request and then either, if everything goes well, it resolves the promise with the received response or it will reject the promise with an error. When we call *httpGet* the immediate return value won't be the received response or an error, instead its immediate return value will be a newly initialized promise, a promise that the *httpGet* function body will execute and whilst executing it will either reject or resolve the promise at some moment in the future. In combination with asynchronous functions, promises are even more useful, as described below.

Asynchronous functions

Promises are by them selves useful, that's why they are implemented in the 2016 version of ECMAScript. But in order to write truly beautiful asynchronous code, another feature is going to be implemented in the 2017 version of ECMAScript, asynchronous functions. In JavaScript, a function body gets executed sequentially, immediately upon calling the function. There is no way to make a function body pause execution until some other asynchronous operation is completed. A callback function can be supplied to be called upon completion of the asynchronous operation, but using just callbacks for asynchronous programming can result in a hard to maintain code base, unless some sort of library/framework is used to provide structure, this is demonstrated in the following example.

```
function getAndWriteFile(domain, cb) {
  https.get(`https://${domain}`, res => {
    let body = '';
    res.on('data', chunk => body += chunk);
    res.on('end', () => {
      console.log('get success');
      fs.writeFile(domain, body, err => cb(err, {
        file_name: domain,
        content: body,
        length: body.length
      }));
    });
  }).on('error', err => cb(err));
}

getAndWriteFile('google.com', (err, file) => {
  if (err
    console.log(err);
  else
    console.log(`getAndWriteFile success, file.length: ${file.length}`);
});
```

It should also be noted that it's hard to handle errors. Since the code runs asynchronous, it cannot be wrapped in a *try/catch* statement, which “executes” it's body the same way a function does. Asynchronous functions make this possible, they allow programmers to write asynchronous code just like they would write any other (sequential) code. Asynchronous function bodies can pause (*await)* their execution until another (asynchronous) action is completed. This is done by using the *await* operator. Take a look at the code below and on the next page, which does pretty much the same thing as the example above, but in a clean and maintainable way.

```
function get (url) {
  return new Promise((resolve, reject) => {
    https.get(url, (res) => {
      let body = '';
      res.on('data', chunk => body += chunk);
      res.on('end', () => resolve(body));
    }).on('error', err => reject(err));
  });
}
```

```
function writeFile(file_name, content) {
  return new Promise((resolve, reject) => {
    fs.writeFile(file_name, content, err => {
      err ? reject(err) : resolve({
        file_name,
        content,
        length: content.length
      });
    });
  });
}

(async function getAndWriteFile(domain) {
  try {
    let body = await get(`https://${domain}`);
    console.log('get success');
    let file = await writeFile(domain, body);
    console.log(`writeFile success, file.length: ${file.length}`);
  }
  catch (err) {
    console.log('getAndWriteFile error', err);
  }
}('google.com'));
```

Tests

In order to make sure every component of *ytdl* works (together), unit and integration tests are adopted in *ytdl* its development work flow. All the abstract methods of *ytdl* its modules are tested separately to make sure at least all components work as expected. If every method works, tests are run to check if all the tested methods work together. Using this test setup it's easy to ensures bugs and possible side effects of maintenance are easy to trace.

Repository management and version control

During the development of *ytdl* it is important to keep track of what happens when and why. Git is used to manage the *ytdl* application its (remotely hosted) repository. As the remote host of the *ytdl* repository GitHub was chosen. With GitHub it's easy to set up a continuous integration environment, because whenever an update is pushed the the remotely hosted repository, 3rd party applications hooks are fired. The firing of those hooks could result in something like a rebuild of the *ytdl* application on the Travis-CI application servers.

Application integration

Ytdl gets re-build on different operating systems, running different versions of Node.js whenever the remotely hosted project repository receives an update. This makes sure *ytdl* is running on the supported platforms. The builds and tests are handled by Travis-CI its servers. Every time the remotely hosted *ytdl* repository gets updated, *ytdl* gets re-build and tested.

Application design

Ytdl has been designed to be set up in a modular fashion, therefor it's easy to build upon *ytdl*. Also because all of *ytdl* its logic is loosely coupled, every piece of application logic is testable without it knowing about other pieces of application logic. I've chosen to not expose underlying logic to programmers via a interface, because of the fact that it may complicate usage without offering any benefit over a simple interface. If one really wants to require for example the *WorkingUrlFinder* class, it is possible to do so with Node.js its module requiring system, since the module isn't hidden, but openly exposed.

Download class

This module gets exported when the *ytdl* module is “required”. From the *Download* class, instances can be drawn. When passed the correct data upon initialization, a download can be started by calling the instance its *start* method. The *start* method starts the chain of events required to download from YouTube. The *Download* class has been set up in a way that allows for a hardware dependent finite amount of *Download* instances to run simultaneously.

Events

The *Download* class inherits the event system of Node.js. Before the *start* method is called, event handlers should be defined. The *success* and *error* events should always be handled, otherwise it can become hard to track which *Download* instance is doing what, when and why.

success

When a *Download* instance is completed, the *success* event gets emitted along with an object containing information about the completion of the *Download* instance, in the case of the *Download* class it's the location of the file that has been downloaded and converted.

error

The *error* event gets emitted when the *Download* instance runs into an error it cannot recover from. Some examples are: unable to HTTPS GET, invalid data passed upon initialization and dependency corruption. When an *error* event is emitted, an error object is passed, containing information about why the error occurred and if possible, solutions to the error.

Flowchart

In the flowchart at the end of the document, the flow of events are described that occur after a *Download* instance its *start* method is called.

WorkingUrlFinder class

The *WorkingUrlFinder* class module gets initialized when a *Download* instance its *getWorkingUrl* method is called. The *WorkingUrlFinder* class attempts attempts to find a URL on which media data is served. If the *WorkingUrlFinder* instance finds out that a signature needs to be deciphered in order to make the URL usable, the *SignatureDecipherer* class module gets initialized. The results of the *SignatureDecipherer* are captured in the *WorkingUrlFinder* instance, so that the *WorkingUrlFinder* instance can pass back a working URL (with a deciphered signature). If an instance of the *WorkingUrlFinder* class fails to find a working URL, the next URL which was supplied to the *WorkingUrlFinder* its constructor is tested, and so forth till all supplied URLs are tested.

Events

The *WorkingUrlFinder* class inherits the event system of Node.js. Before the *start* method is called, event handlers should be defined. The *success* and *error* events should always be handled, otherwise it can become hard to track which *WorkingUrlFinder* instance is doing what, when and why.

success

When a *WorkingUrlFinder* instance is completed, the *success* event gets emitted along with an object containing information about the completion of the *WorkingUrlFinder* instance, in the case of the *WorkingUrlFinder* class it's a working URL.

error

The *error* event gets emitted when the *WorkingUrlFinder* instance runs into an error it cannot recover from. Some examples are: invalid data passed upon initialization, dependency corruption and an error in a *SignatureDecipherer* instance. When an *error* event is emitted, an error object is passed, containing information about why the error occurred and if possible, a solution to the error.

Flowchart

In the flowchart at the end of the document, the flow of events are described that occur after a *WorkingUrlFinder* instance its *start* method is called.

SignatureDecipherer class

Some of the real interesting stuff *ytdl* does, is done in the *SignatureDecipherer* class module its methods. In order to download media data from some URLs, it's required that the URL is suffixed with the deciphered signature. If a URL requires a deciphered signature, and it's not present, no media data can be downloaded, the server responds with a 403 (forbidden) HTTP header. In order to decipher a signature, *ytdl* needs to know where the signature decipher functionality is located at. The signature decipher functionality is located in the JavaScript file which is responsible for the YouTube web player. The player source code gets downloaded, and using some re-occurring patterns it's possible to extract the decipher functionality, even though the source code gets obfuscated in a way that a function will likely not have the same name, argument names and body two times in a row. It wasn't as easy as to find something like a *decipherSignature* function. But there had to be at least some consistent patterns in the obfuscated web player source code, somewhere it had to be defined that if a piece of video format data had a signature in it, it needs to be deciphered. After some thinking and experimenting, I found the following if statement and it's body.

```
if(e.sig||e.s){var h=e.sig||zr(e.s);e.url=kj(e.url,{signature:h})}
```

It is quite clear a check is being done about whether a *sig* or *s* property is present in the *e* object. If so, the if statement it's body gets executed. After which the *h* variable gets assigned a value, either the *e* object it's *sig* property, or the result of calling a function called *zr* with as argument the *e* object its *s* property. So, what is the *zr* function doing? Here is the obfuscated code, spread across multiple lines so its a bit more readable.

```
zr=function(a){
  a=a.split("");yr.Ps(a,2);yr.yl(a,61);yr.Ps(a,2);return a.join("")
}
```

It's pretty obvious that some operations happen on the *a* argument, which is a string, and after those actions, the *a* argument gets returned. It's also clear that some helper functions are used, stored in the *yr* object. So the next part would be to find the *yr* object, which is shown below, beatified a bit for readability.

```
var yr={
  yl:function(a,b){var c=a[0];a[0]=a[b%a.length];a[b]=c},
  JZ:function(a){a.reverse()},
  Ps:function(a,b){a.splice(0,b)}
};
```

All of the key value pairs contain functions that perform operations on the arguments they are passed upon calling them. Putting it all together as shown on the next page certainly looks like some deciphering functionality.

```
var yr = {
  yl: function(a,b) {
    var c=a[0];a[0]=a[b%a.length];a[b]=c
  },
  JZ: function(a){ a.reverse() },
  Ps: function(a,b) { a.splice(0,b) }
};

var zr = function(a) {
  a=a.split("");
  yr.Ps(a,2);
  yr.yl(a,61);
  yr.Ps(a,2);
  return a.join("")
};

if(e.sig||e.s){
  var h=e.sig || zr(e.s);
  e.url = kj(e.url, {signature: h})
}
```

After all of the decipher logic has been found and extracted, it's put together by initializing a new function from the JavaScript *Function* class. The newly initialized function can then be called with an enciphered signature as an argument, in order to decipher the signature. The advantage of using YouTube its own decipher logic, is that when the source code gets obfuscated again, the extraction logic stays the same, because logic is extracted using patterns rather than variable and function names.

Events

The *SignatureDecipherer* class inherits the event system of Node.js. Before the *start* method is called, event handlers should be defined. The *success* and *error* events should always be handled, otherwise it can become hard to track which *SignatureDecipherer* instance is doing what, when and why.

success

If a signature has been deciphered successfully, the success event is emitted along with the deciphered signature.

error

When the *SignatureDecipherer* module fails to find the logic required to decipher a signature, an error event is emitted along with an error object describing the occurred error.

Flowchart

In the flowchart at the end of the document, the flow of events are described that occur after a *SignatureDecipherer* instance its *start* method is called.

Usage as NPM (Node.js) module

Since *ytdl* is build targeting the Node.js platform, it is packaged as an *NPM* module. When installed it can simply be *required* from a Node.js script.

Example

```
const Download = require('ytdl').Download;
const download = new Download(options);
```

Installation

Ytdl can be installed manually and as an *NPM* module. The *NPM* method of installing *ytdl* is recommended.

NPM

In order to use *ytdl* as an *NPM* module, install it using *NPM,* by running *npm install ytdl*. Or add the *ytdl* dependency to your project its *package.json*. Both ways ensure you can *require* the package from the directory *ytdl* is installed in.

Manual installation

If you want to build *ytdl* yourself, you can do so by using *git* to *clone* the repository hosted at *https://github.com/opensoars/ytdl*. Then run *npm install* to install all *ytdl its* dependencies. After that, the application can be build by running *npm run gulp-babel*.

Extensions

Since *ytdl* is set up in a way that makes it easy to build upon (from the inside and outside), a lot of functionality extensions can be developed. A demonstration extension is shown below.

Web server

The example below demonstrates a simple route within an existing Node.js server environment. Whenever an *HTTP POST* request is made to the */download* path and a parameter called *video_id* holding the YouTube video id is present, a new download is started. When a download either fails or succeeds, the HTTP response gets handled accordingly.

```
server.routes.post.add('/download:video_id', (req, res) => {
  new ytdl.Download({
    v: req.params.video_id,
    out: __dirname + '/done'
  }).on('success', result => {
    res.endWithMedia(result.file_location);
  }).on('error', error => {
    res.end(500, {error});
  });
});
```

Future of ytdl

There are some features which are proposed to be implemented into *ytdl*. None of these features will result in changes to the current API, additions will only be made. Some of the proposed features are discussed below.

More detailed events

In order to provide more feedback whilst downloading, more events need to be emitted. Whenever a piece of functionality starts executing an event is emitted. Also when a piece of functionality is completed, an event will be emitted. Using such a detailed event system results in a lot of clarity for both developers and users of *ytdl* about what is going on behind the scenes.

Browser version

In theory it's possible to run *ytdl* completely in the browser, since the whole application in written in JavaScript it'll barely take any time to deploy a browser version of *ytdl*. But since *ytdl* uses a third party tool to handle media file conversions, ffmpeg, another solution has to be found to the file conversion problem. As an experiment, I'll be using a C++ to JavaScript compiler which will compile the entire ffmpeg C++ code base to JavaScript, so ffmpeg can be bundled with the *ytdl* application. This method of course has some downsides, since JavaScript runs a bit slower than the byte code generated by the C++ compiler. But the C++ to JavaScript compilers are getting more advanced every day, and so are JavaScript engines, which could mean it's very well possible that the browser version of *ytdl* will work nicely. As a side project I am also working on some media conversion tools which each do just one thing, for example a flac to mp3 converter. These converters could then also be bundled with *ytdl*, which in turn would run much more lightweight than a whole desktop application written in C++ converted to JavaScript.

Application maintenance

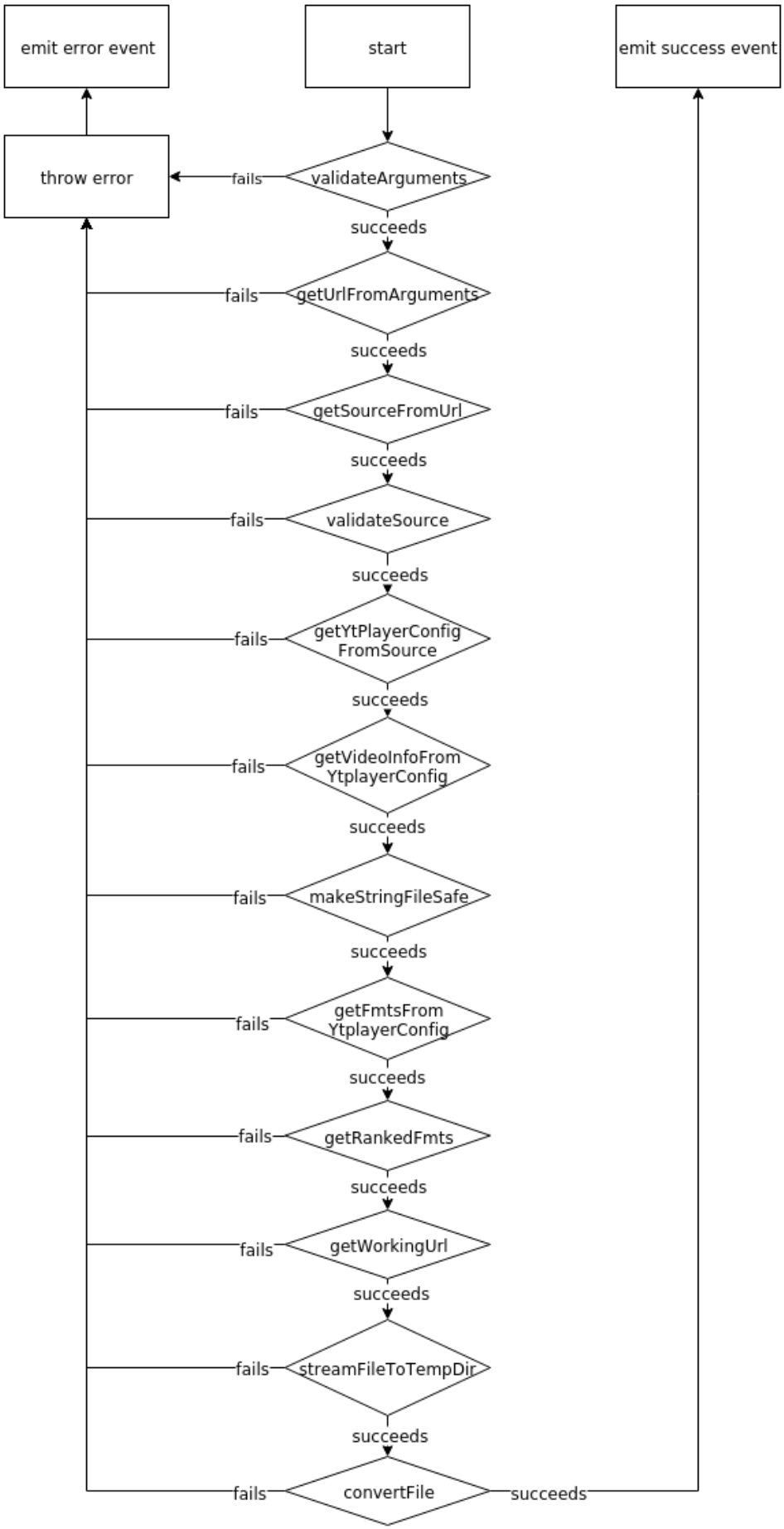
The *ytdl* project its API won't change. It could be possible that new features are added, but 100% backward compatibility will always be maintained.

Bug fixes will of course happen, though they won't change the functionality nor will they change *ytdl* its API, they can be submitted using the GitHub issue tracker.

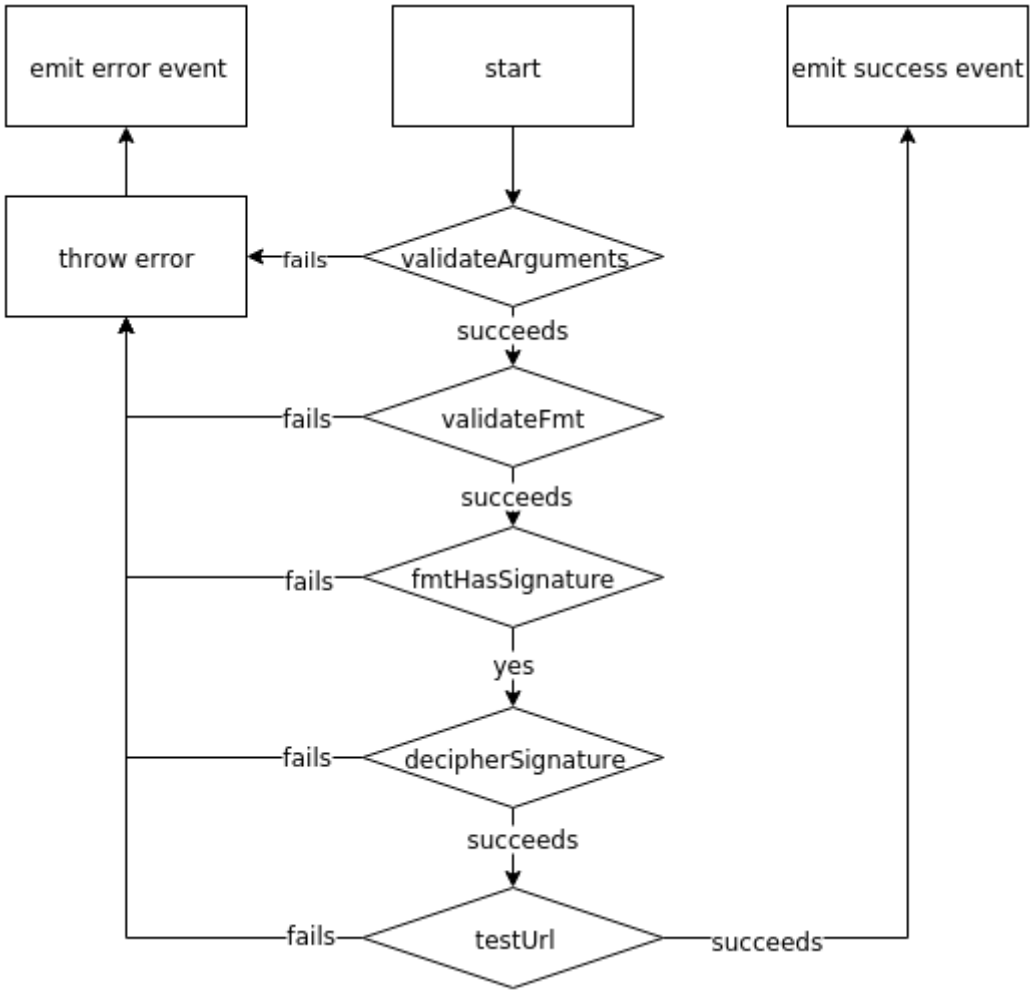
It is expected the *ytdl* project won't require much changes in order to continue working over the years since the code base has been reduced to a minimum in order to run lightweight, whilst maintaining loosely coupled application logic and modularity.

Any maintenance to the *ytdl* project can be done easily, since all application logic is abstracted away behind APIs and all application logic is loosely coupled. Therefor maintenance can often be done to a single module without it affecting other modules.

Download flowchart



WorkingUrlFinder flowchart



SignatureDecipherer flowchart