Project report

Implementation of Google Drive API through JavaScript

**Abhishek Acharya**

**KIIT University**

**1405272**

**Mentor : Mr. Nilesh Shah**

****

**Contents**

[1. Introduction to Google Drive 3](#_Toc500336665)

[2. Advantages and Disadvantages of Google Drive 4](#_Toc500336666)

[3. Project Aim 4](#_Toc500336667)

[4. Implementation of Google Drive API and Google Picker API 4](#_Toc500336668)

[5. Future Scopes 16](#_Toc500336669)

**Acknowledgements**

I am profoundly grateful to **Mr. Nilesh Shah** for his expert guidance and continuous encouragement throughout to see that this project achieves its target since its commencement to its completion. He has helped me to understand the recent industry trends in web development and the continuous evaluation of the progress helped in bringing quality in this project. This had made me exposed to the recent industrial needs in IT Industry.

**Abhishek Acharya**

# Introduction to Google Drive

Google Drive is a cloud storage service created by Google for millions of users to store and share their files.

Google Drive can reach millions of users due to extensive integration of Google and its services in today’s user life. Google Drive currently offers users 15GB free space on the cloud and has premium plans for expanding storage on a time basis.

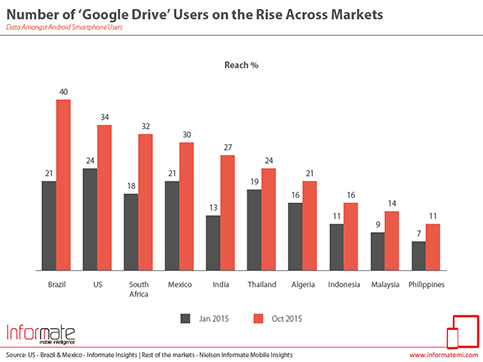
Currently it has the maximum upload size for following:

**Documents:**Up to 1.02 million characters. If you convert a text document to Google Docs format, it can be up to 50 MB.

**Spreadsheets:**Up to 2 million cells for spreadsheets that are created in or converted to Google Sheets.

**Presentations:**Up to 100 MB for presentations converted to Google Slides.

**All other files:**Up to 5 GB.



Reach of Google Drive across countries

The core functionality of Drive apps is to download and upload files in Google Drive. However, the Drive platform provides a lot more than just storage. In this project I have focused on download and upload

The drive platform uses a model based on file IDs — rather than a traditional folder hierarchy — when working with Google Drive files and folders.

# Advantages and Disadvantages of Google Drive

Advantages of Google Drive are:

* Google Drive comes with an initial 15 gigabytes of storage, which is more than enough for the average user. Others such as Onedrive offer only 5GB of free storage.
* Google Drive also allows you to control the level of access that you offer to shared documents. You can choose between 'can edit', 'can comment' and 'can view' settings.

Disadvantage of Google Drive:

* While most users will feel comfortable relying on Google to safely store their documents, not everyone feels at ease knowing that the company has access to their data. Google's business model is predicated on analyzing vast amounts of personal information and then selling it to advertisers.

# Project Aim

This project aims to make one understand how an API of a service such as Google Drive is implemented in one’s own service offering such as an app or a webpage. This project can be integrated into future projects with ease to help users upload or download from their personal Google Drive.

# Implementation of Google Drive API and Google Picker API

This is a documented guide to the files written to download and upload files through the implementation of Google Drive API and Picker API. The language preferred here is JavaScript.

**index.html**

This is the main page where we have two hyperlinks one for download and one for upload. The icons used for the page are loaded from CSS folder present

The scripts involved with this page are:

* filepicker.js
* [https://www.google.com/jsapi?key=[API\_Key]/](https://www.google.com/jsapi?key=%5bAPI_Key%5d/)
* <https://apis.google.com/js/client.js?onload=initPicker>

The API Key can be obtained from the Google API Console (shown in presentation).

The last script mentioned is the script loaded to load the initpicker function.

**The initpicker function.**

function initPicker() {

var picker = new FilePicker({

apiKey: 'AIzaSyDE3cQVyu3ahiTdpWbuMstrrcK07ImhCDw',

clientId: '104151419196-ubiqtgu754n0nj6jddklqkp02btq9j5j.apps.googleusercontent.com',

buttonEl: document.getElementById('pick'),

onSelect: function(file) {

console.log(file);

alert('Selected ' + file.alternateLink);

if(file.downloadUrl)

{ window.open(file.downloadUrl, '', 'width=1000');}

else

{window.open(file.alternateLink, '', 'width=1000');}

}

});

}

The initpicker function is meant for Google Picker API. As soon as the Picker API is loaded by initpicker function a new FilePicker variable is loaded which sends apiKey and clientId. Moreover the buttonE1 is created which on selected loads the file credentials.

The statements console.log and alert are for testing.

We have used two downloadUrl and alternateLink to avoid failures. window.open helps to open the link loaded in new tab.

This all helps in downloading files through Google Picker API.

For uploading files…

This is done by loading the separate html file present in the gdu folder which goes by the name a.html .

**a.html**:

This html page is designed simply only to have a drop zone. And the style credentials written are for designing signin logo placement and drop zone.

<span id="signin">

<span

class="g-signin"

data-callback="signinCallback"

data-clientid="176222212681-oms8rc5ug3fvkm7oqvhgirako23so7ju.apps.googleusercontent.com"//API Key

data-cookiepolicy="single\_host\_origin"

data-scope="https://www.googleapis.com/auth/drive.file">

</span>

</span>

**Signin Callback part**

This signIn callback part loads the scope <https://www.googleapis.com/auth/drive.file> for the necessary permissions. As soon as the user signs-in and gives permission to the application the function mentioned below is loaded.

function signinCallback(result) {

if(result.access\_token) {

accessToken = result.access\_token;

document.getElementById('signin').style.display = 'none';

document.getElementById('drop\_zone').style.display = null;

}

}

**signIn Callback function**

As soon as the signIn Callback function is loaded if the signIn is a success then the span defining signIn disappears and the drop zone appears.

function handleFileSelect(evt) {

evt.stopPropagation();

evt.preventDefault();

var files = evt.dataTransfer.files; // FileList object.

var output = [];

for (var i = 0, f; f = files[i]; i++) {

var uploader = new MediaUploader({

file: f,

token: accessToken,

onComplete: function(data) {

document.getElementById('container').style.display = 'none';

/\*

var element = document.createElement("pre");

element.appendChild(document.createTextNode(data.originalFilename));

document.getElementById('results').appendChild(element);//Shows results

}});

document.getElementById('container').style.display = null;

var bar = new ProgressBar.Line(container, {

strokeWidth: 40,

easing: 'easeInOut',

duration: 2000,

color: '#FFEA82',

trailColor: '#eee',

trailWidth: 1,

svgStyle: {width: '325%', height: '80%'}

});

uploader.upload(); bar.animate(1.0);}

}

The previous script has a function which is called when files are dropped on to the drop target. For each file, it uploads the content to Drive & displays the progressbar during the file is being uploaded.

For progressbar I have imported progressbar.js and progressbar.min.js for implementing progressbar to show the status of being uploaded. I have implemented this to avoid wasting time in designing progressbar in javascript or css, hence implementing the concept of reuse of what’s available in open.

**progressbar.js** is sourced from <https://kimmobrunfeldt.github.io/progressbar.js/>

The drop zone designed after referring to:

<https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API>

The loop is created to continue till the selected files are not uploaded.

The uploader variable creates a new MediaUploader object which gets the credentials of the file dragged and dropped to the dropzone. And as soon as a new bar object of Progressbar is loaded then uploader.upload() starts the uploads and simultaneously bar(progress bar) starts to animate (with 1px width).

**Dragover handler to set the drop effect**

function handleDragOver(evt) {

evt.stopPropagation();

evt.preventDefault();

evt.dataTransfer.dropEffect = 'copy';

}

The a.html of gdu folder is powered by scripts:

* <https://apis.google.com/js/client:plusone.js>
* upload.js
* progreesbar.js
* progressbar.min.js

**In upload.js**

var RetryHandler = function() {

this.interval = 1000;

this.maxInterval = 60 \* 1000;

};

This function defined under RetryHandler defines the parameters of interval and retrial interval.

RetryHandler.prototype.retry = function(fn) {

setTimeout(fn, this.interval);

this.interval = this.nextInterval\_();

};

RetryHandler.prototype.reset = function() {

this.interval = 1000;

};

RetryHandler.prototype.nextInterval\_ = function() {

var interval = this.interval \* 2 + this.getRandomInt\_(0, 1000);

return Math.min(interval, this.maxInterval);

};

RetryHandler.prototype.getRandomInt\_ = function(min, max) {

return Math.floor(Math.random() \* (max - min + 1) + min);

};

The RetryHandler defined in the first part used to create objects for defining timeout, reset interval and interval between two uploads.

var MediaUploader = function(options) {

var noop = function() {};

this.file = options.file;

this.contentType = options.contentType || this.file.type || 'application/octet-stream';

this.metadata = options.metadata || {

'title': this.file.name,

'mimeType': this.contentType};

this.token = options.token;

this.onComplete = options.onComplete || noop;

this.onProgress = options.onProgress || noop;

this.onError = options.onError || noop;

this.offset = options.offset || 0;

this.chunkSize = options.chunkSize || 0;

this.retryHandler = new RetryHandler();

this.url = options.url;

if (!this.url) {

var params = options.params || {};

params.uploadType = 'resumable';

this.url = this.buildUrl\_(options.fileId, params, options.baseUrl); }

this.httpMethod = options.fileId ? 'PUT' : 'POST';};

The MediaUploader function mentioned in a.html(of gdu folder) is defined here.

**MediaUploader.prototype.sendFile\_** defines the function which defines the file credential and size and hence chunk size and offset size.

On upload success or failure this gets executed:

xhr.onload = this.onContentUploadSuccess\_.bind(this);

xhr.onerror = this.onContentUploadError\_.bind(this);

xhr.send(content);

**MediaUploader.prototype.resume\_** is defined similarly to **MediaUploader.prototype.sendFile\_** This object gets executed when the upload resumes automatically.

MediaUploader.prototype.onContentUploadSuccess\_ = function(e) {

if (e.target.status == 200 || e.target.status == 201) {

this.onComplete(e.target.response);

} else if (e.target.status == 308) {

this.extractRange\_(e.target);

this.retryHandler.reset();

this.sendFile\_();

} else {

this.onContentUploadError\_(e);}};

MediaUploader.prototype.onContentUploadError\_ = function(e) {

if (e.target.status && e.target.status < 500) {

this.onError(e.target.response);

} else {

this.retryHandler.retry(this.resume\_.bind(this));}};

MediaUploader.prototype.onUploadError\_ = function(e) {

this.onError(e.target.response);

};

MediaUploader.prototype.buildQuery\_ = function(params) {

params = params || {};

return Object.keys(params).map(function(key) {

return encodeURIComponent(key) + '=' + encodeURIComponent(params[key]);

}).join('&');

};

These objects of MediaUploader helps to return the error status if the files fail to get uploaded.

The object MediaUploader.prototype.onContentUploadSuccess\_

Helps to send the success code on successful upload. These codes are helpful in knowing the appropriate errors.

# Future Scopes

Since, the project is based on open source code this project can be implemented in any future project and it will be lightweight since it is in JavaScript.