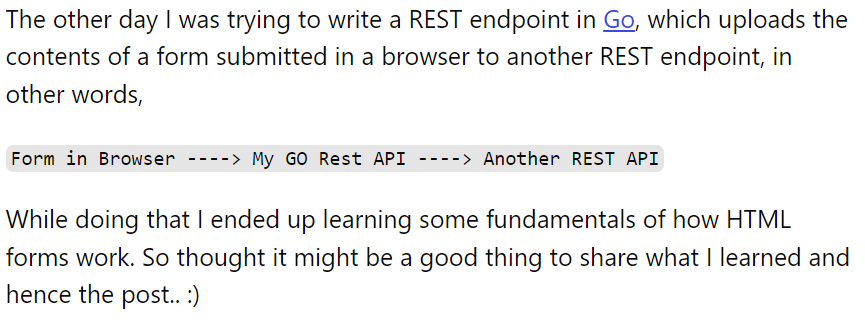
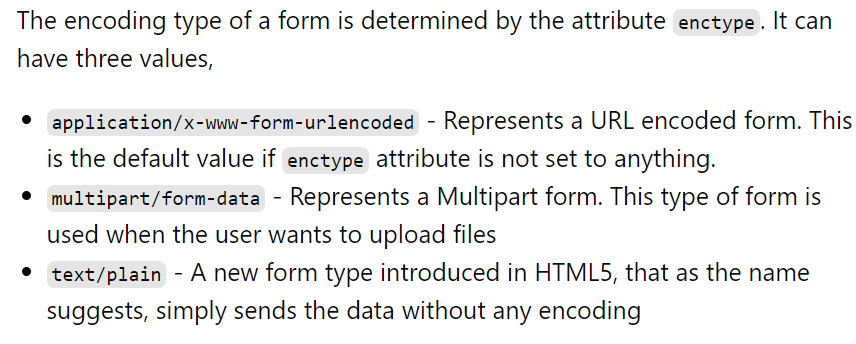
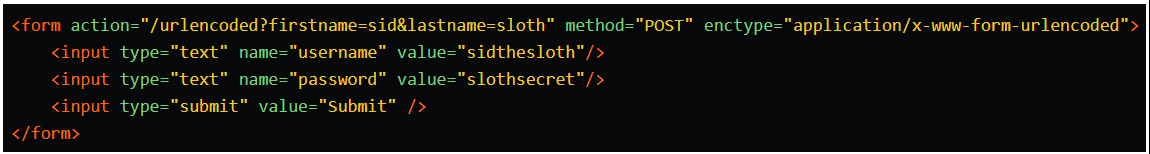
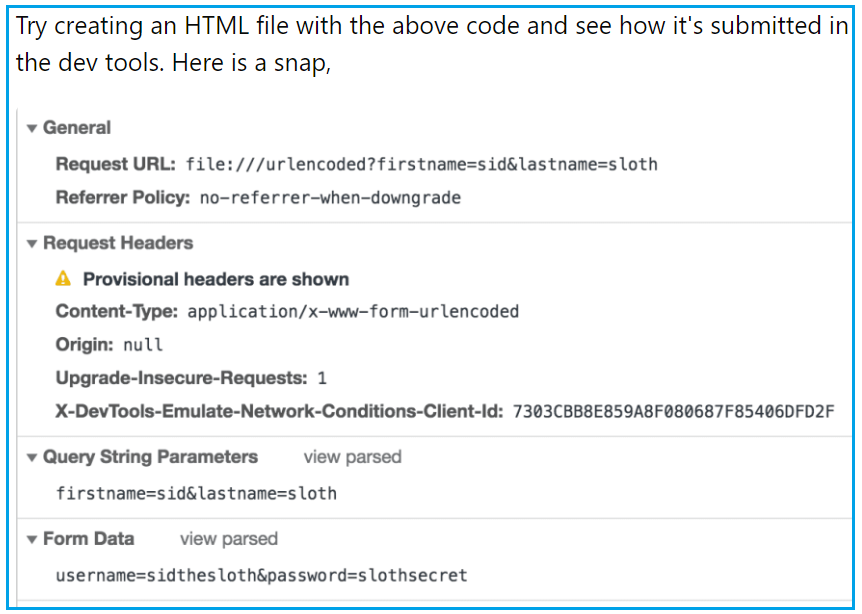
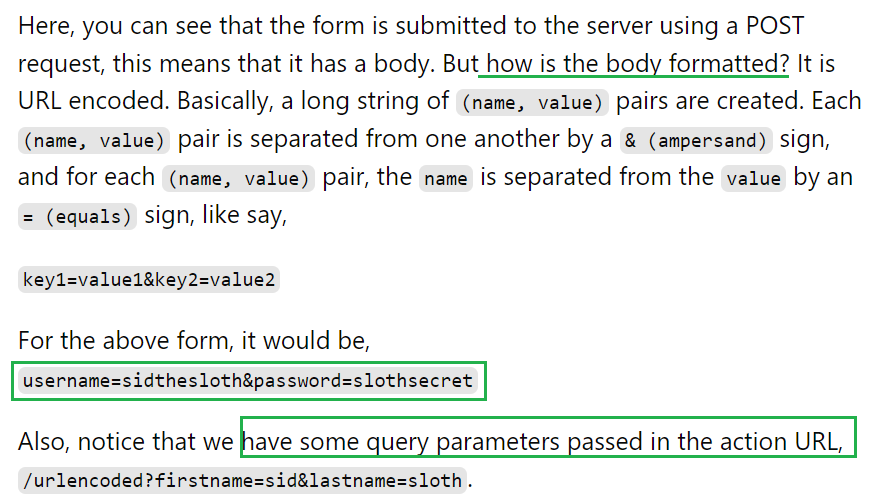
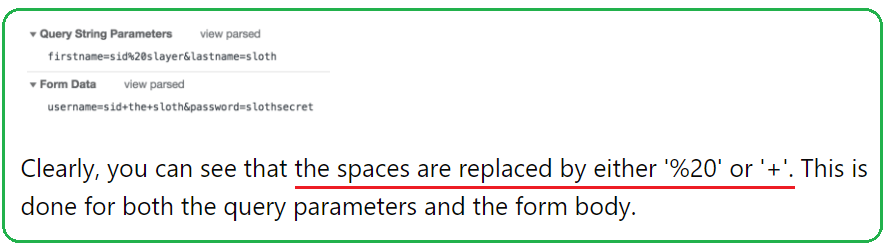
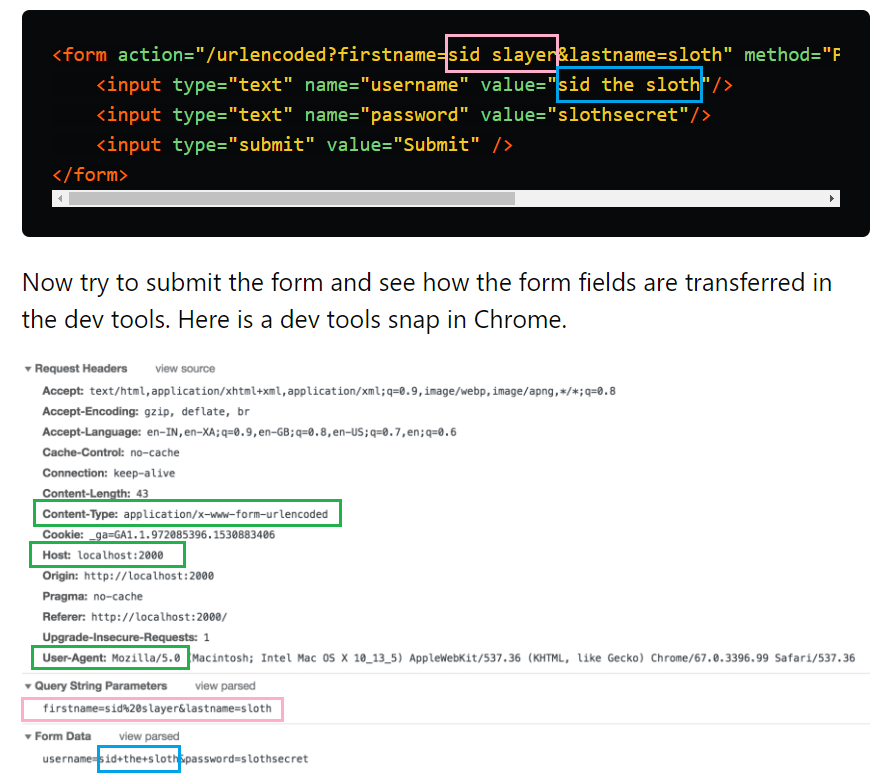
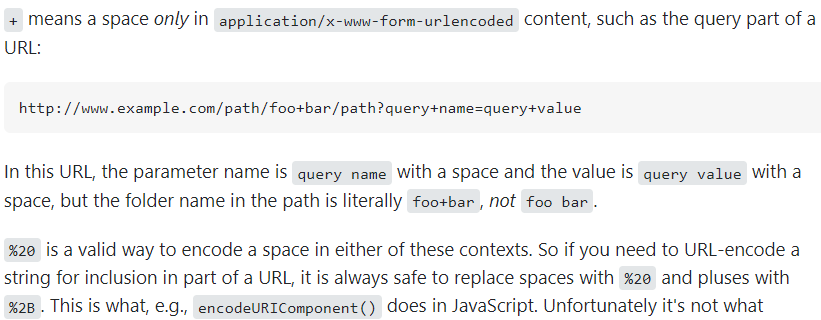
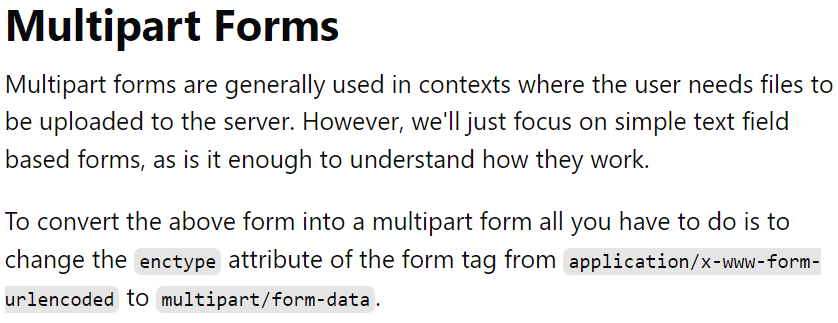
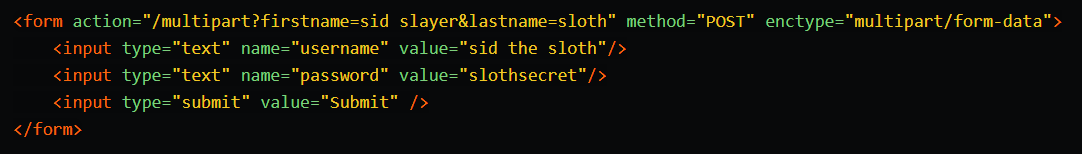
# Understanding HTML Form Encoding: URL Encoded and Multipart Forms

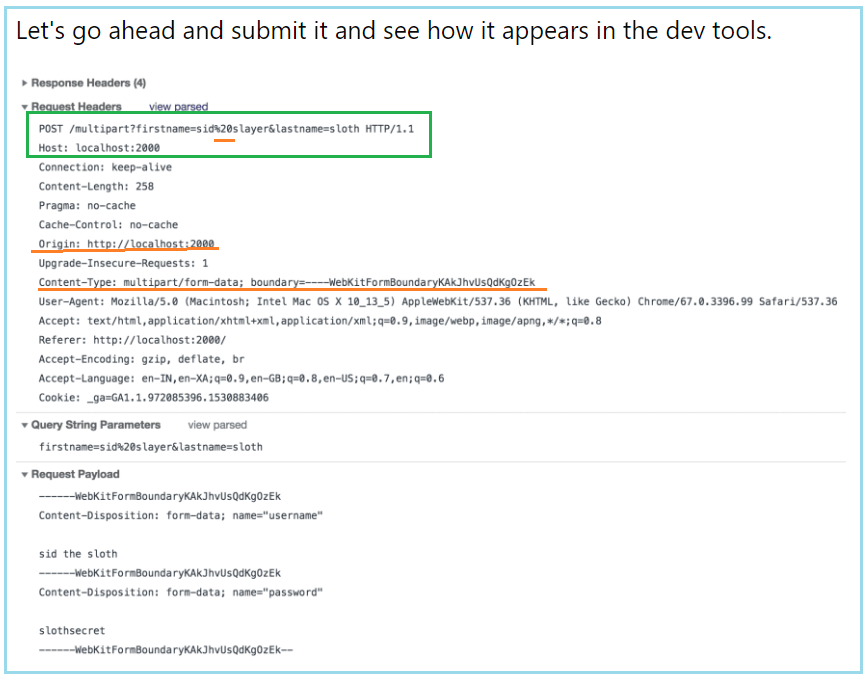
 

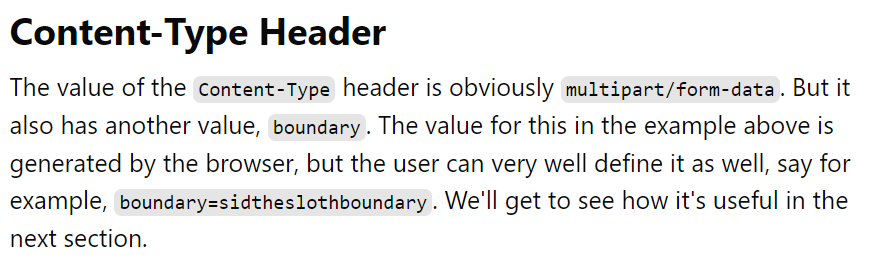
**Send a form through form-urlencoded**

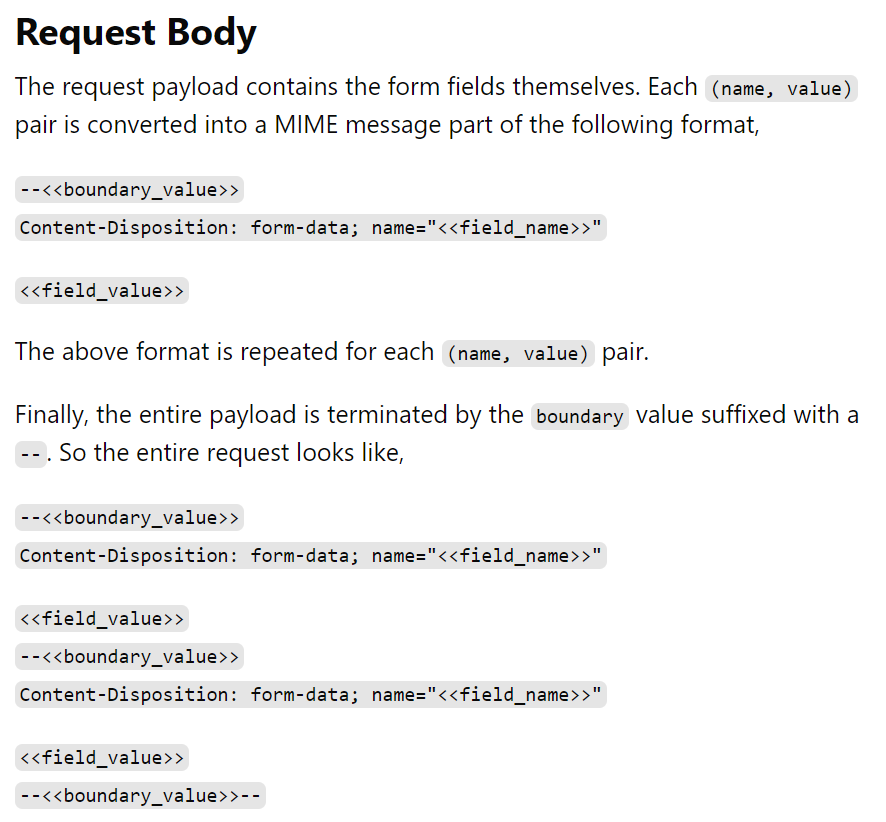
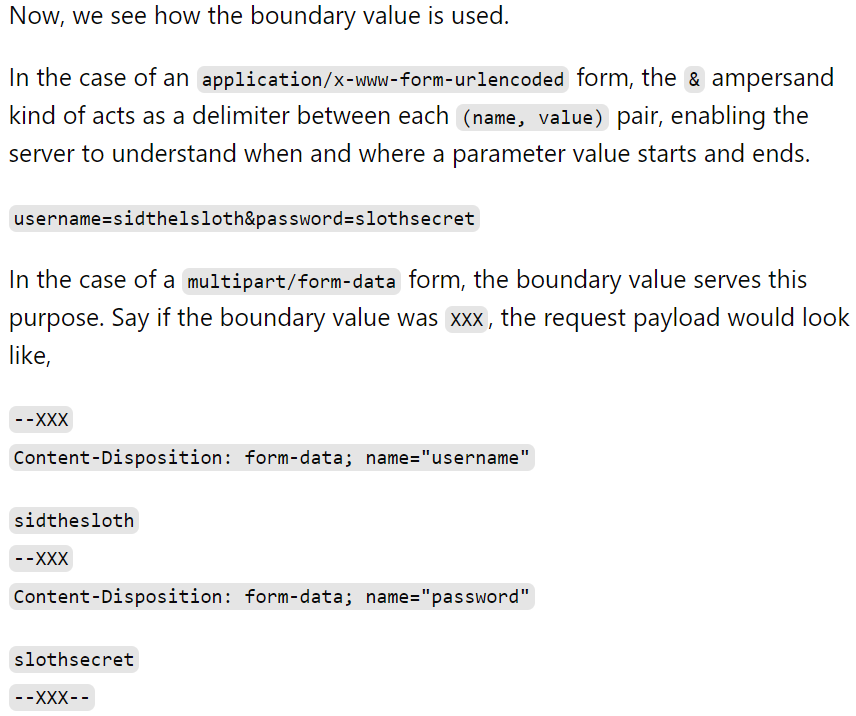


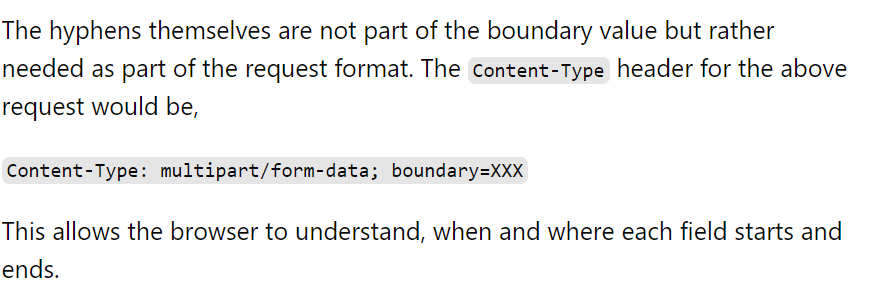
  

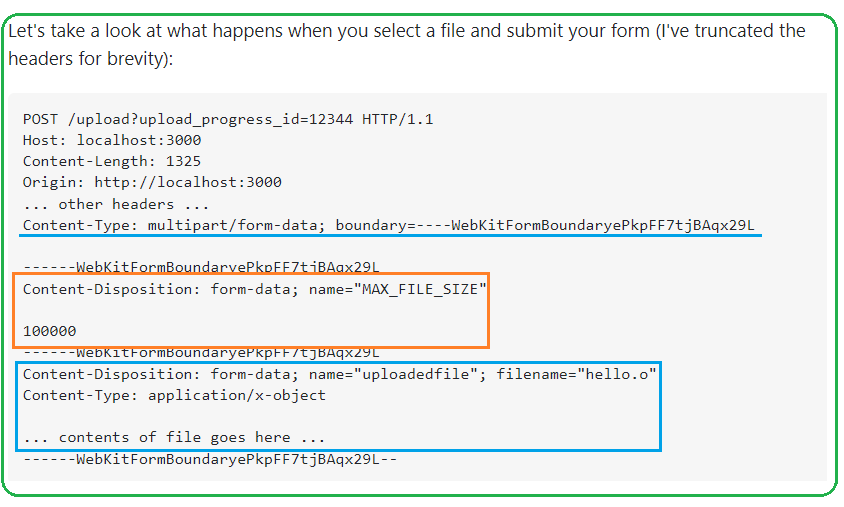
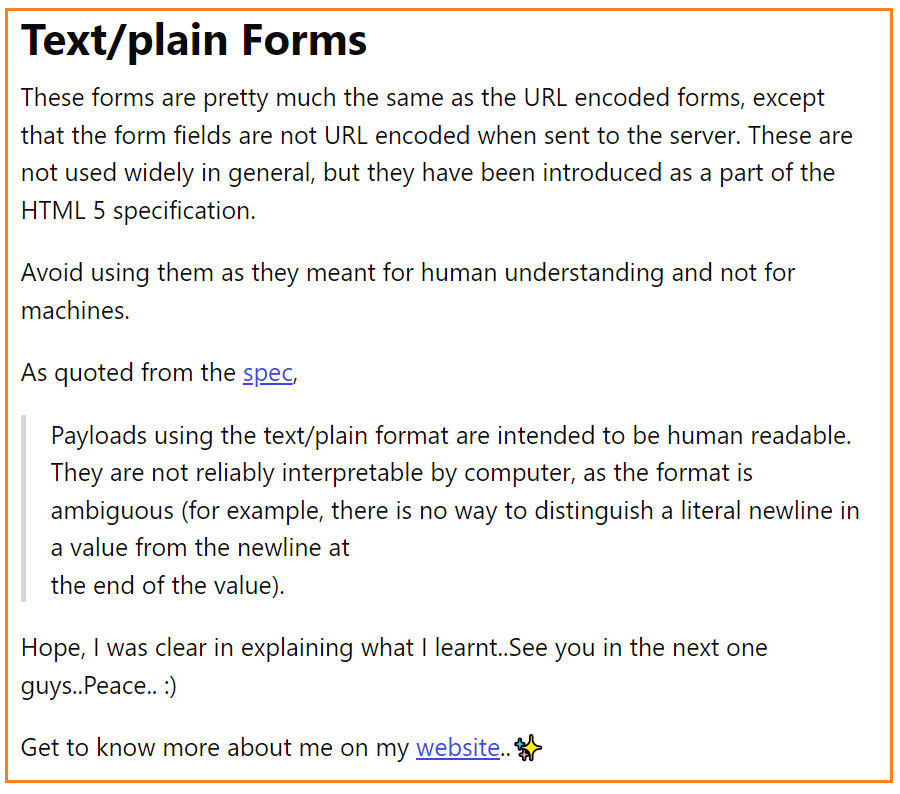
 











## **How to generate the examples**

**https://stackoverflow.com/questions/8659808/how-does-http-file-upload-work**

Once you see an example of each method, it becomes obvious how they work, and when you should use each one.

You can produce examples using:

* nc -l or an ECHO server: [HTTP test server accepting GET/POST requests](https://stackoverflow.com/questions/5725430/http-test-server-accepting-get-post-requests/52351480#52351480)
* an user agent like a browser or cURL

Save the form to a minimal .html file:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8"/>

<title>upload</title>

</head>

<body>

<form action="http://localhost:8000" method="post" enctype="multipart/form-data">

<p><input type="text" name="text1" value="text default">

<p><input type="text" name="text2" value="a&#x03C9;b">

<p><input type="file" name="file1">

<p><input type="file" name="file2">

<p><input type="file" name="file3">

<p><button type="submit">Submit</button>

</form>

</body>

</html>

We set the default text value to a&#x03C9;b, which means aωb because ω is U+03C9, which are the bytes 61 CF 89 62 in UTF-8.

Create files to upload:

echo 'Content of a.txt.' > a.txt

echo '<!DOCTYPE html><title>Content of a.html.</title>' > a.html

# Binary file containing 4 bytes: 'a', 1, 2 and 'b'.

printf 'a\xCF\x89b' > binary

Run our little echo server:

while true; do printf '' | nc -l 8000 localhost; done

Open the HTML on your browser, select the files and click on submit and check the terminal.

nc prints the request received.

Tested on: Ubuntu 14.04.3, nc BSD 1.105, Firefox 40.

## **multipart/form-data**

Firefox sent:

POST / HTTP/1.1

[[ Less interesting headers ... ]]

Content-Type: multipart/form-data; boundary=---------------------------735323031399963166993862150

Content-Length: 834

-----------------------------735323031399963166993862150

Content-Disposition: form-data; name="text1"

text default

-----------------------------735323031399963166993862150

Content-Disposition: form-data; name="text2"

aωb

-----------------------------735323031399963166993862150

Content-Disposition: form-data; name="file1"; filename="a.txt"

Content-Type: text/plain

Content of a.txt.

-----------------------------735323031399963166993862150

Content-Disposition: form-data; name="file2"; filename="a.html"

Content-Type: text/html

<!DOCTYPE html><title>Content of a.html.</title>

-----------------------------735323031399963166993862150

Content-Disposition: form-data; name="file3"; filename="binary"

Content-Type: application/octet-stream

aωb

-----------------------------735323031399963166993862150--

For the binary file and text field, the bytes 61 CF 89 62 (aωb in UTF-8) are sent literally. You could verify that with nc -l localhost 8000 | hd, which says that the bytes:

61 CF 89 62

were sent (61 == 'a' and 62 == 'b').

Therefore it is clear that:

* Content-Type: multipart/form-data; boundary=---------------------------735323031399963166993862150 sets the content type to multipart/form-data and says that the fields are separated by the given boundary string.

But note that the:

boundary=---------------------------735323031399963166993862150

has two less dadhes -- than the actual barrier

-----------------------------735323031399963166993862150

This is because the standard requires the boundary to start with two dashes --. The other dashes appear to be just how Firefox chose to implement the arbitrary boundary. RFC 7578 clearly mentions that those two leading dashes -- are required:

4.1. "Boundary" Parameter of multipart/form-data

As with other multipart types, the parts are delimited with a boundary delimiter, constructed using CRLF, "--", and the value of the "boundary" parameter.

* every field gets some sub headers before its data: Content-Disposition: form-data;, the field name, the filename, followed by the data.

The server reads the data until the next boundary string. The browser must choose a boundary that will not appear in any of the fields, so this is why the boundary may vary between requests.

Because we have the unique boundary, no encoding of the data is necessary: binary data is sent as is.

TODO: what is the optimal boundary size (log(N) I bet), and name / running time of the algorithm that finds it? Asked at: <https://cs.stackexchange.com/questions/39687/find-the-shortest-sequence-that-is-not-a-sub-sequence-of-a-set-of-sequences>

* Content-Type is automatically determined by the browser.

How it is determined exactly was asked at: [How is mime type of an uploaded file determined by browser?](https://stackoverflow.com/questions/1201945/how-is-mime-type-of-an-uploaded-file-determined-by-browser)

## **application/x-www-form-urlencoded**

Now change the enctype to application/x-www-form-urlencoded, reload the browser, and resubmit.

Firefox sent:

POST / HTTP/1.1

[[ Less interesting headers ... ]]

Content-Type: application/x-www-form-urlencoded

Content-Length: 51

text1=text+default&text2=a%CF%89b&file1=a.txt&file2=a.html&file3=binary

Clearly the file data was not sent, only the basenames. So this cannot be used for files.

As for the text field, we see that usual printable characters like a and b were sent in one byte, while non-printable ones like 0xCF and 0x89 took up **3 bytes** each: %CF%89!

## **Comparison**

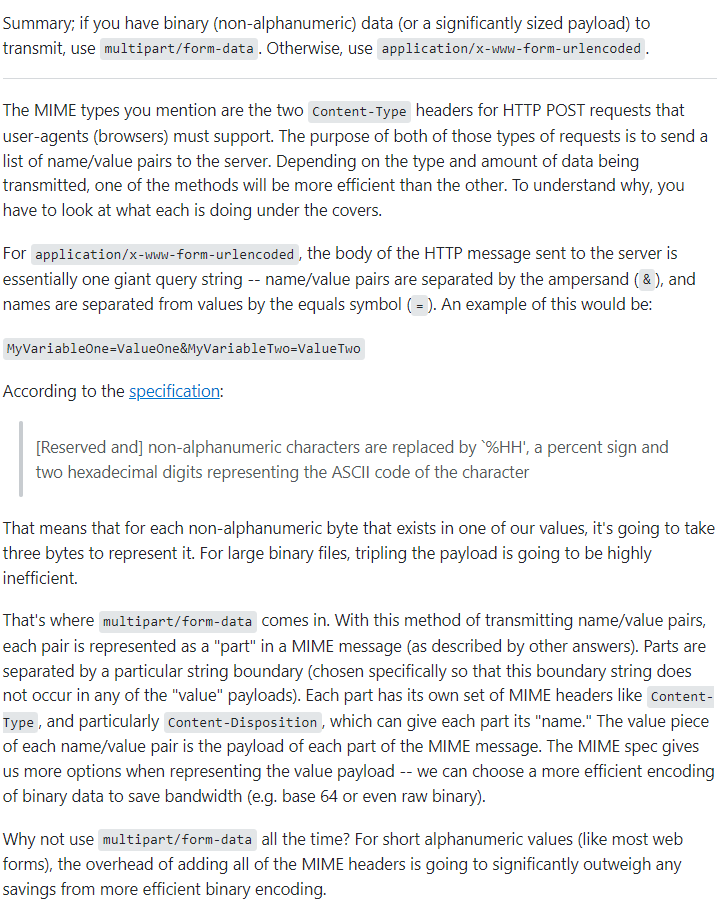
File uploads often contain lots of non-printable characters (e.g. images), while text forms almost never do.

From the examples we have seen that:

* multipart/form-data: adds a few bytes of boundary overhead to the message, and must spend some time calculating it, but sends each byte in one byte.
* **application/x-www-form-urlencoded**: has a single byte boundary per field (&), but adds a linear overhead factor of **3x** for every non-printable character.

Therefore, even if we could send files with application/x-www-form-urlencoded, we wouldn't want to, because it is so inefficient.

**But for printable characters found in text fields, it does not matter and generates less overhead, so we just use it**.



In **HTTP** there are **two ways to POST data**:

**application/x-www-form-urlencoded** and

**multipart/form-data**.

**Regarding Restful API over HTTP** the most popular content-types I came in touch with are **application/xml** and **application/json**.

Summary; if you have binary (non-alphanumeric) data (or a significantly sized payload) to transmit, use multipart/form-data. Otherwise, use application/x-www-form-urlencoded.

For **application/x-www-form-urlencoded**, the body of the HTTP message sent to the server is essentially one giant query string -- name/value pairs are separated by the ampersand (&), and names are separated from values by the equals symbol (=). An example of this would be:

MyVariableOne=ValueOne&MyVariableTwo=ValueTwo

The content type "application/x-www-form-urlencoded" is inefficient for sending large quantities of binary data or text containing non-ASCII characters.

The content type "multipart/form-data" should be used for submitting forms that contain files, non-ASCII data, and binary data.

**application/xml:**

* data-size: XML very verbose, but usually not an issue when using compression and thinking that the write access case (e.g. through POST or PUT) is much more rare as read-access (in many cases it is <3% of all traffic). Rarely there where cases where I had to optimize the write performance
* existence of **non-ascii chars**: you can use utf-8 as encoding in XML
* **binary data**: would need to use base64 encoding
* filename data: you can encapsulate this inside field in XML

**application/json**

* data-size: more compact less that XML, still text, but you can compress
* **non-ascii chars**: json is utf-8
* **binary data**: base64 (also see [json-binary-question](https://stackoverflow.com/questions/1443158/binary-data-in-json-string-something-better-than-base64))
* filename data: encapsulate as own field-section inside json

What is the difference between

request.ContentType = "application/json; charset=utf-8";

and

webRequest.ContentType = "application/x-www-form-urlencoded";

The first case is telling the web server that you are posting JSON data as in:

{ Name : 'John Smith', Age: 23}

The second option is telling the web server that you will be **encoding** the parameters like URL-encoding as in:

Name=John+Smith&Age=23, and send it through payload not through URL.

webRequest.ContentType = "application/x-www-**form-urlencoded**";

1. Where does **application/x-www-form-urlencoded**'s name come from?

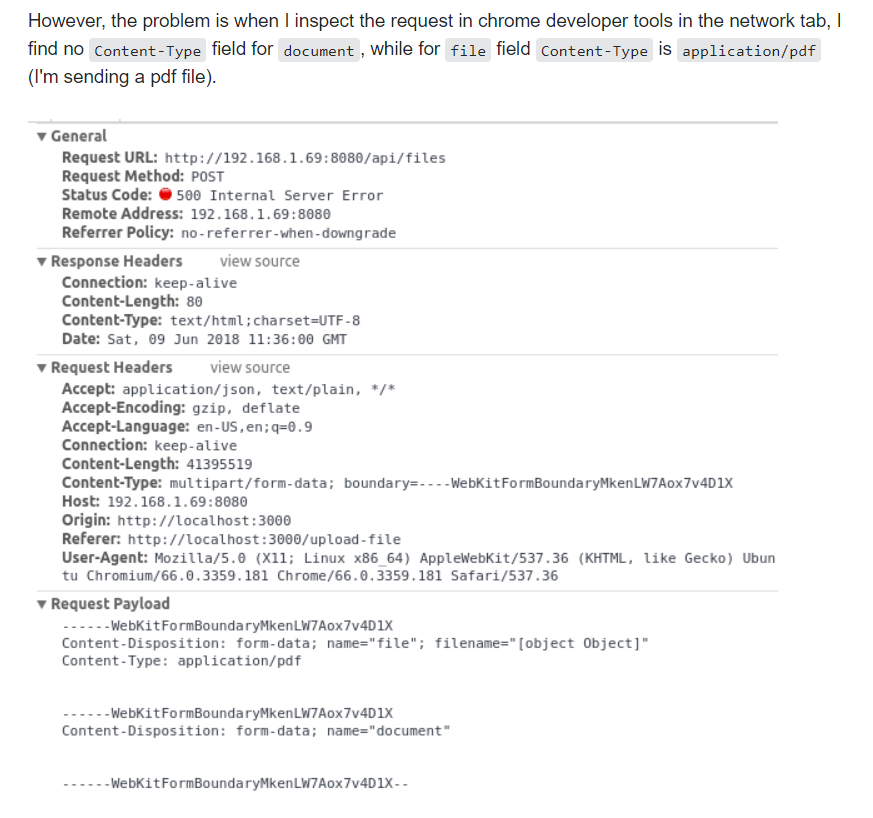
If you send HTTP **GET** request, you can use **query parameters** as follows:

http://example.com/path/to/page**?name=ferret&color=purple**

The content of the fields is encoded as a query string. The application/x-www-form- urlencoded's name come from the previous url query parameter **but the query parameters is in where the body of request instead of url**.

The whole form data is sent as a long query string. The query string contains **name- value** pairs separated by **&** character

e.g. field1=value1&field2=value2



**This is an example full HTTP request:**

Again, **the overall HTTP content type** (**multipart/form-data; boundary="----=\_Part\_4\_35071402.1568998611352"**) is automatically generated. The boundary value is random and you cannot control that (you do not need to). Message ID and MIME version are automatically added as part of the HTTP request

**POST** /api/files.upload?channels=THNMAZN8G HTTP/1.1

**Content-Type**: **multipart/form-data**; boundary="----=**\_Part\_4\_35071402.1568998611352**"

User-Agent: Boomi Http Transport

Message-ID: <519117923.5.1568998611383.JavaMail.W10CV0B0X2$@W10CV0B0X2>

MIME-Version: 1.0

Connection: close

**Authorization**: Bearer xoxp-75624234-834346893544234-1256634555-876ff8fc520e18c5fdd7b30ac5f8988e

Cache-Control: no-cache

Pragma: no-cache

**Host:** [**slack.com**](http://slack.com/)

Accept: text/html, image/gif, image/jpeg, \*; q=.2, \*/\*; q=.2

Content-Length: 214529

------=\_Part\_4\_35071402.1568998611352

Content-Type: application/octet-stream

Content-Transfer-Encoding: binary

Content-Disposition**: form-data; name="file"; filename=test.pdf**

**<<PDF\_Data>>**

------=\_Part\_4\_35071402.1568998611352—

# Sending a file and other form data together

If you need to send a file and other form data, you need to add multiple JSON objects. Combining the examples above,

'{

      "ID" : {

         "value" : "XYZ123ABC",

         "Content-Type" : "application/text-plain",

         "Content-Transfer-Encoding":"8bit",

         "type" : "data"

      },

"service-name" : {

         "value" : "Bob",

         "Content-Type" : "application/text-plain",

         "Content-Transfer-Encoding":"8bit",

         "type" : "data"

      },

  "file" : {

         "value" : "test.pdf",

         "Content-Type" : "application/octet-stream",

"Content-Transfer-Encoding":"binary",

         "type" : "key"

      }

}'

**post the contents of a file that contains JSON. The contents of the file look like this**:

{

"id”:99999999,

"orders":[

{

"ID”:8383838383,

"amount":0,

"slotID":36972026

},

{

"ID”:2929292929,

"amount":0,

"slotID":36972026

},

{

"ID”:4747474747,

"amount":0,

"slotID":36972026

}]

}

import requests

**url** = 'https://api.example.com/api/dir/v1/accounts/9999999/orders'

**headers** = {'Authorization' : ‘(some auth code)’, 'Accept' : 'application/json', 'Content-Type' : 'application/json'}

r = **requests.post**(url, data=open('example.json', 'rb'), **headers**=headers)

#!/usr/bin/env python3

import requests

import json

files = {'file': open(‘example.json’, 'rb')}

headers = {'Authorization' : ‘(some auth code)’, 'Accept' : 'application/json', 'Content-Type' : 'application/json'}

r = **requests.post**('https://api.example.com/api/dir/v1/accounts/9999999/orders', files=files, headers=headers)

import requests

import json

**with open('example.json') as json\_file:**

**json\_data = json.load(json\_file)**

**headers = {'Authorization' : ‘(some auth code)’, 'Accept' : 'application/json', 'Content-Type' : 'application/json'}**

**r = requests.post('https://api.example.com/api/dir/v1/accounts/9999999/orders', data=json.dumps(json\_data), headers=headers)**

with open('example.json') as json\_file:

json\_data = json.load(json\_file)

auth=('token', 'example')

r = **requests.post**('https://api.example.com/api/dir/v1/accounts/9999999/orders', **json=json\_data**, auth=auth)

**If you want to send a smaller file, send it as a string.**

contents = open('example.json', 'rb').read()

r = requests.post(url, data=contents, headers=headers)

We are using API Gateway in our project and are running into a weird issue. We have a virtualized REST API allowing us to send images to a server.

If I send my request directly to the server (through the backend URL), it handles it with success and saves the image as expected. But when I send the request to the virtualised API (through the frontend API), the gateway seems unable to understand the request. The requests are exactly the same, only the urls I use diverge.

The request is as follows:

**POST** /v1/img HTTP/1.1

**Host**: xx.xx.xx.xx:8065

**User-Agent**: curl/7.47.0

Accept: application/json

Cache-Control: no-cache

Content-Length: 15469

Expect: 100-continue

**content-type**: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW; boundary=------------------------5180b2a73f429dfd

--------------------------5180b2a73f429dfd

Content-Disposition: form-data; name="image"; filename="testimage.jpg"

Content-Type: image/jpeg

(((((((((((((((((((((((((((((((((((((((((((((((((((ï¿½ï¿½ï¿½ï¿½"ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½ï¿½@AJ(ï¿½ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½Ë†ï¿½3

iÊ»@(ï¿½rï¿½QQlU8bï¿½ï¿½5ï¿½ï¿½ï¿½qÄ�ï¿½uï¿½ï¿½2ï¿½R\*ï¿½Rï¿½q g,AYï¿½Wï¿½ï¿½Jï¿½Zï¿½[5+ï¿½ï¿½Qlï¿½JÛ¸ï¿½[ï¿½ï¿½Ò”ï¿½ï¿½ï¿½?ï¿½|&È³Lï¿½ï¿½.eï¿½\*8W5Â¹ï¿½ï¿½pï¿½kï¿½s\+ï¿½t>eï¿½PÔ›Ð®|

.......

--------------------------5180b2a73f429dfd

Content-Disposition: form-data; name="id"

testFrontend

--------------------------5180b2a73f429dfd--

**The Accept header should be changed to application/json in order to get a JSON response**

#

# [upload a file and post JSON data in the same request](https://forums.servicestack.net/t/how-to-upload-a-file-and-post-json-data-in-the-same-request/3318)

**POST** [http://ws-local.myhost.com/UploadFile?format=json 75](http://ws-local.myhost.com/UploadFile?format=json) HTTP/1.1  
**Accept**: multipart/form-data  
Authorization: IRA-HMAC

**Content-Type**: multipart/form-data; boundary=---------------------------**–b4537661-9989-40c1-977b-41dd400fe6c9**  
Content-Type: multipart/form-data  
**Host**: [ws-local.myhost.com 1](http://ws-local.myhost.com/)  
Content-Length: 491789  
Expect: 100-continue

–b4537661-9989-40c1-977b-41dd400fe6c9  
Content-Type: application/json; charset=utf-8  
Content-Disposition: form-data

{“File”:{“AccountId”:“6d1c818a-7b3a-40e4-ab72-cadf4eae7c85”}}  
–b4537661-9989-40c1-977b-41dd400fe6c9  
Content-Disposition: form-data; name=file; filename=MyFile.pdf; filename\*=utf-8’'MyFile.pdf

%PDF-1.7  
%  
455 0 obj  
…

Which parameter between (data / json / files) should be used,it's actually depends on a request header named ContentType(usually check this through developer tools of your browser),

when the Content-Type is application/x-www-form-urlencoded, code should be:

**requests.post(url, data=jsonObj)**

when the Content-Type is application/json, your code is supposed to be one of below:

**requests.post(url, json=jsonObj)**

**requests.post(url, data=jsonstr, headers={"Content-Type":"application/json"})**

when the Content-Type is multipart/form-data, it's used to upload files, so your code should be:

**requests.post(url, files=xxxx)**

**In an application I am developing RESTful API and we want the client to send data as JSON. Part of this application requires the client to upload a file (usually an image) as well as information about the image.**

**I'm having a hard time tracking down how this happens in a single request. Is it possible to Base64 the file data into a JSON string? Am I going to need to perform 2 posts to the server? Should I not be using JSON for this**?

"multipart/form-data" contains a series of parts. Each part is expected to contain a content-disposition header [RFC 2183] where the disposition type is "form-data", and where the disposition contains an (additional) parameter of "name", where the value of that parameter is the original field name in the form. For example, a part might contain a header:

**Content-Disposition**: form-data; name="user"

with the value corresponding to the entry of the "user" field.

You basically have 45 choices:

1. **Base64** encode the file, at the expense of increasing the data size by around 33%, and add processing overhead in both the server and the client for encoding/decoding.
2. Send the file first in a multipart/form-data POST, and return an ID to the client. The client then sends the metadata with the ID, and the server re-associates the file and the metadata.
3. Send the metadata first, and return an ID to the client. The client then sends the file with the ID, and the server re-associates the file and the metadata.
4. **You can send the file and data over in one request using the**[**multipart/form-data**](http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.2)**content type:**

**We are developing server with REST API, which accepts and responses with JSON.** The problem is, if you need to upload images from client to server.

Note: and also I am talking about a use-case where the entity (user) can have multiple files (carPhoto, licensePhoto) and also have other properties (name, email...), but when you create new user, you don't send these images, they are added after the registration process.

**1. Use multipart/form-data instead of JSON**

*good* : POST and PUT requests are as RESTful as possible, they can contain text inputs together with file.

*cons* : It is not JSON anymore, which is much easier to test, debug etc. compare to multipart/form-data

**2. Allow to update separate files**

**POST** request for creating new user does not allow to add images (which is ok in our use-case how I said at beginning), uploading pictures is done by **PUT** request as multipart/form-data to for example /users/4/carPhoto

*good* **: Everything (except the file uploading itself) remains in JSON**, it is easy to test and debug (you can log complete JSON requests without being afraid of their length)

*cons* : It is not intuitive, you cant POST or PUT all variables of entity **at once** and also this address /users/4/carPhoto can be considered more as a collection (standard use-case for REST API looks like this /users/4/shipments). Usually you cant (and dont want to) GET/PUT each variable of entity, for example users/4/name . You can get name with GET and change it with PUT at users/4. If there is something after the id, it is usually another collection, like users/4/reviews

**3. Use Base64**

**Send it as JSON but encode files with Base64**.

*good* : Same as first solution, it is as RESTful service as possible.

*cons* : Once again, testing and debugging is a lot worse (the body can have megabytes of data), there is increase in size and also in processing time in both - client and server

1. POST to /users to create the user entity.
2. POST the image to /images, making sure to return a Location header to where the image can be retrieved per the HTTP spec.
3. PATCH to /users/carPhoto and assign it the ID of the photo given in the Location header of step 2.

There's no easy solution. Each way has their pros and cons . But the canonical way is using the first option: multipart/form-data. As [W3 recommendation guide](http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.2) says

The content type "multipart/form-data" should be used for submitting forms that contain files, non-ASCII data, and binary data.

We aren't sending forms,really, but the implicit principle still applies. Using base64 as a binary representation, is incorrect because you're using the incorrect tool for accomplish your goal, in other hand, the second option forces your API clients to do more job in order to consume your API service. You should do the hard work in the server side in order to supply an easy-to-consume API. The first option is not easy to debug, but when you do it, it probably never changes.

Using multipart/form-data you're sticked with the REST/http philosophy. You can view an answer to similar question [here](https://stackoverflow.com/questions/4083702/posting-a-file-and-data-to-restful-webservice-as-json).

Another option if mixing the alternatives, you can use multipart/form-data but instead of send every value separate, you can send a value named payload with the json payload inside it. (I tried this approach using ASP.NET WebAPI 2 and works fine).

The content type "multipart/form-data" should be used for submitting forms that contain files, non-ASCII data, and binary data

Is it possible to send binary data in JSON ?

If you need a JSON or XML wrapper to your file you'll have to text-encode it, Base 64 for example

The easiest way is to **convert the binary data into string by using Base64 encoding**. However this has space overhead as Base64 encoding means around +33% more memory.

convert the pdf document to base64, set that in a property of your json and on the server side take it out from base64 again to the binary and process further.

Today's web applications heavily rely on json for client server communication. Because json is a totally text based standard it goes very well until we need to send anything which is not text, like pdf files and images. And it is not a good idea that you implement a separate module to transfer binary files, it will defeat the whole purpose of using json.  
I didn't found a straightforward and simple tutorial to handle binary files with json.

**As json only supports text so we have to convert binary file (pdf, image etc) in to a string**. And then we can easily add it to a json field.

Hi Ivan, Definitely you can use it in your application. For example if you want to show a list of items in the browser, you can get a list of json objects from backend containing the information about the individual item of the list. Now if you want to show thumbnails along with their information, you can simply include the thumbnail image in the json object as a field. It will make it very easy to handle the code. And also you are not dependent on the image url to show images on the browser because you have included the image in the json itself, so you are free from creating and maintain images on the disk(or any web service) to create urls, you can directly get images from database (or any other storage) and send them directly to the browser. There are as many benefits as you can think of. But having said that, you must take in to account, the size of payload. **You can't just include any size of binary content in json**, **because it will increase the size of json response, so you must consider the memory requirement of your project**. But in general purpose scenarios it is good to use. Even I have created a framework to handle all this work in one of my projects, and out of there I have created this tutorial. I hope it answers your question.

**It looks like you’re posting an xlsx file. So in your case you can either use the specific mime type for the file “application/vnd.openxmlformats-officedocument.spreadsheetml.sheet” or preferably the generic binary stream content type “application/octet-stream”. By doing that you let the target server know what kind of file you’re posting.**

# [How to send a PDF file via JSON from my REST Service to my clients](https://stackoverflow.com/questions/10759970/how-to-send-a-pdf-file-via-json-from-my-rest-service-to-my-clients)

# 

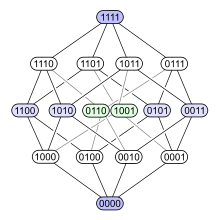
# 

# What Is Base64 Encoding

<https://levelup.gitconnected.com/what-is-base64-encoding-4b5ed1eb58a4>

Nowadays we can transfer any format of data across the globe and that data can be kind of **text**, **Blob**, and **CLOB** data, and when the data transfers through some communication medium the data is **chopped** into **chunks** called **packets** and each packet contains data in **binary format(0101000101001)**. Each packet then moves through the network in a **series** of **hops**.





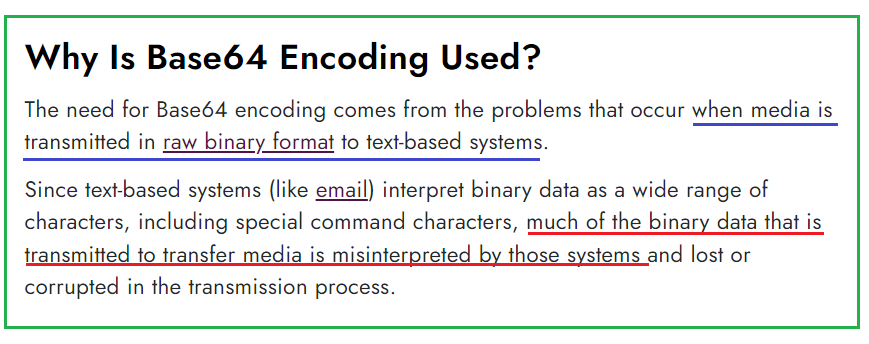
Before diving into the **base64** Algorithm. Let’s talk about **Blob** and **CLOB**first**.**

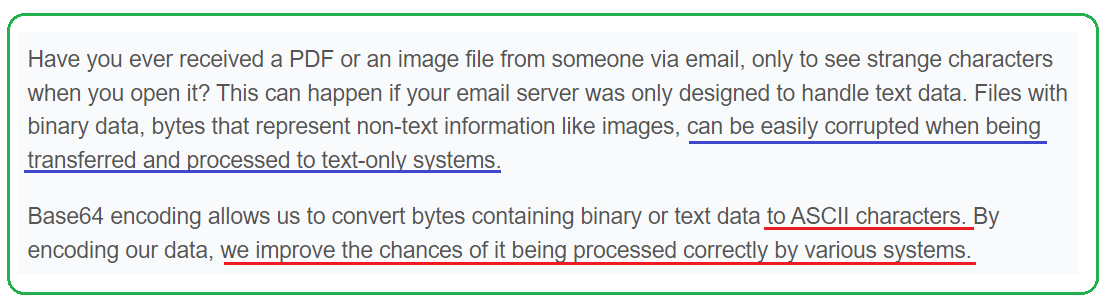
**What BLOB and CLOB mean?**

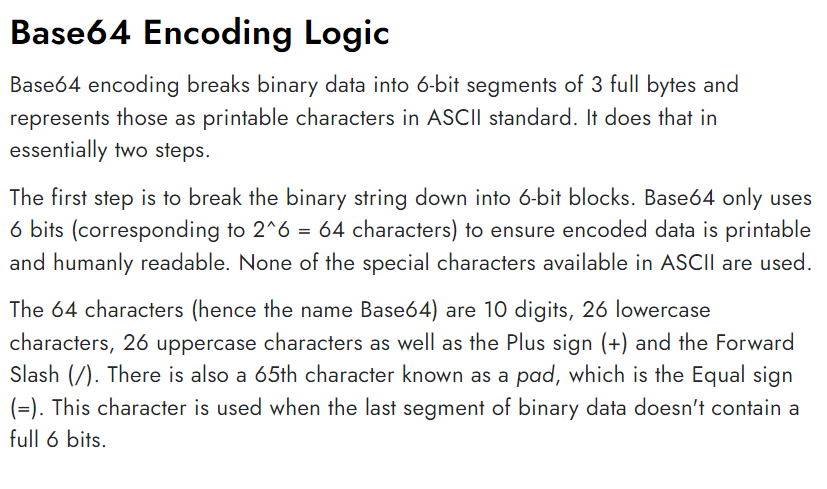
**Blob** means*Binary Large Object File*. When we transfer an Image, Audio, and Video kind of data format over the network, then that data is a kind of **BLOB** data. **CLOB** means *Character Large Object File.* When we transfer Text, XML, or any kind of **character** data format over the network then that data is a kind of **CLOB** data.

Now, let's dive into **base64 encoding**.

The **base64** is a ***binary to a text encoding scheme*** that represents binary data in an **ASCII** string format. **base64** is designed to carry data stored in binary format across the channels. It takes any form of data and **transforms it into a long string of plain text**. Earlier we can not transfer a large amount of data like files because it is made up of 2⁸ bit bytes but our actual network uses 2⁷ bit bytes. This is where **base64 encoding**came into the picture. But, what actually **base64** means?







let’s understand the meaning of **base64**.

**base64** = **base**+**64**

base64 encoding contains 64 characters to encode any string.

base64 contains:

* 10 numeric value i.e., 0,1,2,3,…..9.
* 26 Uppercase alphabets i.e., A,B,C,D,…….Z.
* 26 Lowercase alphabets i.e., a,b,c,d,……..z.
* two special characters i.e., +,/. Depends upon your OS

