

Toggle navigation

miguelgrinberg.com

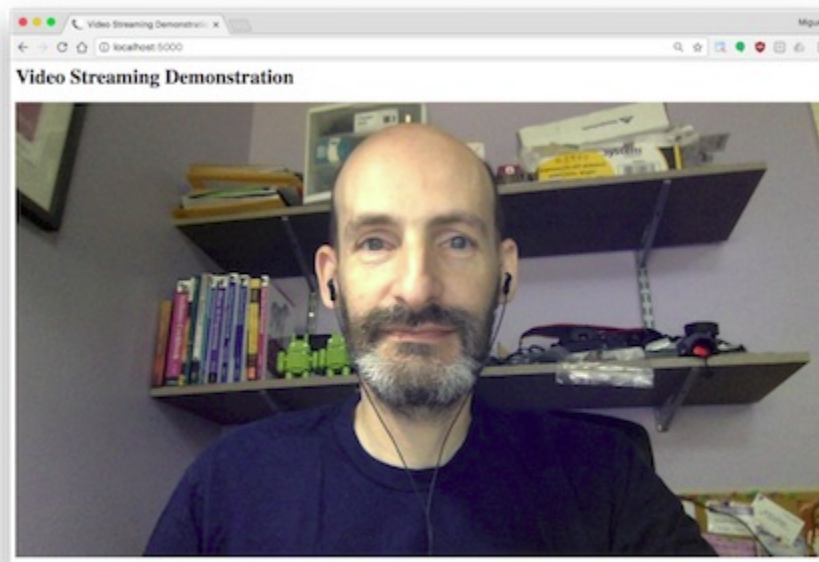
- [Home](#)
- [My Courses](#)
- [About Me](#)
- [Hire me!](#)



Flask Video Streaming Revisited

August 6 2017

Posted by [Miguel Grinberg](#) under [Flask](#), [Python](#), [Programming](#), [Video](#).

[Tweet](#)

Almost three years ago I wrote an article on this blog titled [Video Streaming with Flask](#), in which I presented a very modest

streaming server that used a Flask generator view function to stream a [Motion-JPEG](#) stream to web browsers. My intention with that article was to show a simple, yet practical use of [streaming responses](#), a not very well known feature in Flask.

That article is extremely popular, but not because it teaches how to implement streaming responses, but because a lot of people want to implement streaming video servers. Unfortunately, my focus when I wrote the article was not on creating a robust video server, so I frequently get questions and requests for advice from those who want to use the video server for a real application and quickly find its limitations. So today I'm going to revisit my streaming video server and describe a few improvements I've made to it.

Recap: Using Flask's Streaming for Video

I recommend you read the [original article](#) to familiarize yourself with my project. In short, this is a Flask server that uses a streaming response to provide a stream of video frames captured from a camera in Motion JPEG format. This format is very simple and not the most efficient, but has the advantage that all browsers support it natively and without any client-side scripting required. It is a fairly common format used by security cameras for that reason. To demonstrate the server, I implemented a camera driver for a Raspberry Pi with its camera module. For those that didn't have a Pi with a camera at hand, I also wrote an emulated camera driver that streams a sequence of jpeg images stored on disk.

Running the Camera Only When There Are Viewers

One aspect of the original streaming server that people did not like is that the background thread that captures video frames from the Raspberry Pi camera starts when the first client connects to the stream, but then it never stops. A more efficient way to handle this background thread is to only have it running while there are viewers, so that the camera can be turned off when nobody is connected.

I implemented this improvement a while ago. The idea is that every time a frame is accessed by a client the current time of that access is recorded. The camera thread checks this timestamp and if it finds it is more than ten seconds old it exits. With this change, when the server runs for ten seconds without any clients it will shut its camera off and stop all background activity. As soon as a client connects again the thread is restarted.

Here is a brief description of the changes:

```
class Camera(object):
    # ...
    last_access = 0 # time of last client access to the camera

    # ...

    def get_frame(self):
        Camera.last_access = time.time()
        # ...

    @classmethod
    def _thread(cls):
        with picamera.PiCamera() as camera:
            # ...
            for foo in camera.capture_continuous(stream, 'jpeg', use_video_port=True):
```

```
# ...  
# if there hasn't been any clients asking for frames in  
# the last 10 seconds stop the thread  
if time.time() - cls.last_access > 10:  
    break  
cls.thread = None
```

Simplifying the Camera Class

A common problem that a lot of people mentioned to me is that it is hard to add support for other cameras. The `Camera` class that I implemented for the Raspberry Pi is fairly complex because it uses a background capture thread to talk to the camera hardware.

To make this easier, I decided to move the generic functionality that does all the background processing of frames to a base class, leaving only the task of getting the frames from the camera to implement in subclasses. The new `BaseCamera` class in module `base_camera.py` implements this base class. Here is what this generic thread looks like:

```
class BaseCamera(object):  
    thread = None # background thread that reads frames from camera  
    frame = None # current frame is stored here by background thread  
    last_access = 0 # time of last client access to the camera  
    # ...  
  
    @staticmethod  
    def frames():  
        """Generator that returns frames from the camera."""  
        raise RuntimeError('Must be implemented by subclasses.')  
  
    @classmethod  
    def _thread(cls):  
        """Camera background thread."""  
        print('Starting camera thread.')
```

```

frames_iterator = cls.frames()
for frame in frames_iterator:
    BaseCamera.frame = frame

    # if there hasn't been any clients asking for frames in
    # the last 10 seconds then stop the thread
    if time.time() - BaseCamera.last_access > 10:
        frames_iterator.close()
        print('Stopping camera thread due to inactivity.')
        break
BaseCamera.thread = None

```

This new version of the Raspberry Pi's camera thread has been made generic with the use of yet another generator. The thread expects the `frames()` method (which is a static method) to be a generator implemented in subclasses that are specific to different cameras. Each item returned by the iterator must be a video frame, in jpeg format.

Here is how the emulated camera that returns static images can be adapted to work with this base class:

```

class Camera(BaseCamera):
    """An emulated camera implementation that streams a repeated sequence of
    files 1.jpg, 2.jpg and 3.jpg at a rate of one frame per second."""
    imgs = [open(f + '.jpg', 'rb').read() for f in ['1', '2', '3']]

    @staticmethod
    def frames():
        while True:
            time.sleep(1)
            yield Camera.imgs[int(time.time()) % 3]

```

Note how in this version the `frames()` generator forces a frame rate of one frame per second by simply sleeping that amount between frames.

The camera subclass for the Raspberry Pi camera also becomes

much simpler with this redesign:

```
import io
import picamera
from base_camera import BaseCamera

class Camera(BaseCamera):
    @staticmethod
    def frames():
        with picamera.PiCamera() as camera:
            # let camera warm up
            time.sleep(2)

            stream = io.BytesIO()
            for foo in camera.capture_continuous(stream, 'jpeg', use_video_port=True):
                # return current frame
                stream.seek(0)
                yield stream.read()

                # reset stream for next frame
                stream.seek(0)
                stream.truncate()
```

OpenCV Camera Driver

A fair number of users complained that they did not have access to a Raspberry Pi equipped with a camera module, so they could not try this server with anything other than the emulated camera. Now that adding camera drivers is much easier, I wanted to also have a camera based on [OpenCV](#), which supports most USB webcams and laptop cameras. Here is a simple camera driver for it:

```
import cv2
from base_camera import BaseCamera

class Camera(BaseCamera):
    @staticmethod
```

```
def frames():
    camera = cv2.VideoCapture(0)
    if not camera.isOpened():
        raise RuntimeError('Could not start camera.')

    while True:
        # read current frame
        _, img = camera.read()

        # encode as a jpeg image and return it
        yield cv2.imencode('.jpg', img)[1].tobytes()
```

With this class, the first video camera reported by your system will be used. If you are using a laptop, this is likely your internal camera. If you are going to use this driver, you need to install the OpenCV bindings for Python:

```
$ pip install opencv-python
```

Camera Selection

The project now supports three different camera drivers: emulated, Raspberry Pi and OpenCV. To make it easier to select which driver to use without having to edit the code, the Flask server looks for a `CAMERA` environment variable to know which class to import. This variable can be set to `pi` or `opencv`, and if it isn't set, then the emulated camera is used by default.

The way this is implemented is fairly generic. Whatever the value of the `CAMERA` environment variable is, the server will expect the driver to be in a module named `camera_${CAMERA}.py`. The server will import this module and then look for a `Camera` class in it. The logic is actually quite simple:

```
from importlib import import_module
import os

# import camera driver
if os.environ.get('CAMERA'):
    Camera = import_module('camera_' + os.environ['CAMERA']).Camera
else:
    from camera import Camera
```

For example, to start an OpenCV session from bash, you can do this:

```
$ CAMERA=opencv python app.py
```

From a Windows command prompt you can do the same as follows:

```
$ set CAMERA=opencv
$ python app.py
```

Performance Improvements

Another observation that was made a few times is that the server consumes a lot of CPU. The reason for this is that there is no synchronization between the background thread capturing frames and the generator feeding those frames to the client. Both run as fast as they can, without regards for the speed of the other.

In general it makes sense for the background thread to run as fast as possible, because you want the frame rate to be as high as possible for each client. But you definitely do not want the generator that delivers frames to a client to ever run at a faster

rate than the camera is producing frames, because that would mean duplicate frames will be sent to the client. While these duplicates do not cause any problems, they increase CPU and network usage without any benefit.

So there needs to be a mechanism by which the generator only delivers original frames to the client, and if the delivery loop inside the generator is faster than the frame rate of the camera thread, then the generator should wait until a new frame is available, so that it paces itself to match the camera rate. On the other side, if the delivery loop runs at a slower rate than the camera thread, then it should never get behind when processing frames, and instead it should skip frames to always deliver the most current frame. Sounds complicated, right?

What I wanted as a solution here is to have the camera thread signal the generators that are running when a new frame is available. The generators can then block while they wait for the signal before they deliver the next frame. In looking through synchronization primitives, I've found that [`threading.Event`](#) is the one that matches this behavior. So basically, each generator should have an event object, and then the camera thread should signal all the active event objects to inform all the running generators when a new frame is available. The generators deliver the frame and reset their event objects, and then go back to wait on them again for the next frame.

To avoid having to add event handling logic in the generator, I decided to implement a customized event class that uses the thread id of the caller to automatically create and manage a separate event for each client thread. This is somewhat complex,

to be honest, but the idea came from how Flask's context local variables are implemented. The new event class is called `CameraEvent`, and has `wait()`, `set()`, and `clear()` methods. With the support of this class, the rate control mechanism can be added to the `BaseCamera` class:

```
class CameraEvent(object):
    # ...

class BaseCamera(object):
    # ...
    event = CameraEvent()

    # ...

    def get_frame(self):
        """Return the current camera frame."""
        BaseCamera.last_access = time.time()

        # wait for a signal from the camera thread
        BaseCamera.event.wait()
        BaseCamera.event.clear()

        return BaseCamera.frame

    @classmethod
    def _thread(cls):
        # ...
        for frame in frames_iterator:
            BaseCamera.frame = frame
            BaseCamera.event.set() # send signal to clients

        # ...
```

The magic that is done in the `CameraEvent` class enables multiple clients to be able to wait individually for a new frame. The `wait()` method uses the current thread id to allocate an individual event object for each client and wait on it. The `clear()` method will reset the event associated with the caller's thread id, so that each generator thread can run at its own speed. The `set()`

method called by the camera thread sends a signal to the event objects allocated for all clients, and will also remove any events that aren't being serviced by their owners, because that means that the clients associated with those events have closed the connection and are gone. You can see the implementation of the `CameraEvent` class in the [GitHub repository](#).

To give you an idea of the magnitude of the performance improvement, consider that the emulated camera driver consumed about 96% CPU before this change because it was constantly sending duplicate frames at a rate much higher than the one frame per second being produced. After these changes, the same stream consumes about 3% CPU. In both cases there was a single client viewing the stream. The OpenCV driver went from about 45% CPU down to 12% for a single client, with each new client adding about 3%.

Production Web Server

Lastly, I think if you plan to use this server for real, you should use a more robust web server than the one that comes with Flask. A very good choice is to use Gunicorn:

```
$ pip install gunicorn
```

With Gunicorn, you can run the server as follows (remember to set the `CAMERA` environment variable to the selected camera driver first):

```
$ gunicorn --threads 5 --workers 1 --bind 0.0.0.0:5000 app:app
```

The `--threads 5` option tells Gunicorn to handle at most five concurrent requests. That means that with this number you can get up to five clients to watch the stream simultaneously. The `--workers 1` options limits the server to a single process. This is required because only one process can connect to a camera to capture frames.

You can increase the number of threads some, but if you find that you need a large number, it will probably be more efficient to use an asynchronous framework instead of threads. Gunicorn can be configured to work with the two frameworks that are compatible with Flask: `gevent` and `eventlet`. To make the video streaming server work with these frameworks, there is one small addition to the camera background thread:

```
class BaseCamera(object):
    # ...
    @classmethod
    def _thread(cls):
        # ...
        for frame in frames_iterator:
            BaseCamera.frame = frame
            BaseCamera.event.set() # send signal to clients
            time.sleep(0)
        # ...
```

The only change here is the addition of a `sleep(0)` in the camera capture loop. This is required for both `eventlet` and `gevent`, because they use cooperative multitasking. The way these frameworks achieve concurrency is by having each task release the CPU either by calling a function that does network I/O or explicitly. Since there is no I/O here, the `sleep` call is what achieves the CPU release.

Now you can run Gunicorn with the `gevent` or `eventlet` workers as follows:

```
$ CAMERA=opencv gunicorn --worker-class gevent --workers 1 --bind 0.0.0.0:5000 app:app
```

Here the `--worker-class gevent` option configures Gunicorn to use the `gevent` framework (you must install it with `pip install gevent`). If you prefer, `--worker-class eventlet` is also available. The `--workers 1` limits to a single process as above. The `eventlet` and `gevent` workers in Gunicorn allocate a thousand concurrent clients by default, so that should be much more than what a server of this kind is able to support anyway.

Conclusion

All the changes described above are incorporated in the [GitHub repository](#). I hope you get a better experience with these improvements.

Before concluding, I want to provide quick answers to other questions I have received about this server:

- How to force the server to run at a fixed frame rate?
Configure your camera to deliver frames at that rate, then sleep enough time during each iteration of the camera capture loop to also run at that rate.
- How to increase the frame rate? The server as described here delivers frames as fast as possible. If you need better frame rates, you can try configuring your camera for a smaller frame size.

- How to add sound? That's really difficult. The Motion JPEG format does not support audio. You are going to need to stream the audio separately, and then add an audio player to the HTML page. Even if you manage to do all this, synchronization between audio and video is not going to be very accurate.
- How to save the stream to disk on the server? Just save the sequence of JPEG files in the camera thread. For this you may want to remove the automatic mechanism that ends the background thread when there are no viewers.
- How to add playback controls to the video player? Motion JPEG was not made for interactive operation by the user, but if you are set on doing this, with a little bit of trickery it may be possible to implement playback controls. If the server saves all jpeg images, then a pause can be implemented by having the server deliver the same frame over and over. When the user resumes playback, the server will have to deliver "old" images that are loaded from disk, since now the user would be in DVR mode instead of watching the stream live. This could be a very interesting project!

That is all for now. If you have other questions please let me know!

*Hello, and thank you for visiting my blog!
If you enjoyed this article, please consider
supporting my work on this blog on
[Patreon!](#)*

A red rectangular button with a white Patreon logo icon on the left and the text "BECOME A PATRON" in white capital letters on the right.

Tweet

143 comments

#1 JohnDen said 2 years ago

Thx!

#2 Tony said 2 years ago

Thanks for the great write-up, is it possible to have simultaneous multiple cameras?

#3 Jose Luis said 2 years ago

Another way to store the video since you are using opencv is to use VideoWriter functions. Something like:

```
class VideoCamera(object):
    def __init__(self):
        self.video = cv2.VideoCapture(0)
        fourcc = cv2.VideoWriter_fourcc('M','J','P','G')
        self.out = cv2.VideoWriter('output.avi',fourcc, 20.0, (1280,720))

    def __del__(self):
        self.video.release()
        self.out.release()
```

```
def get_frame(self):
    success, image = self.video.read()
    self.out.write(image)
    ret, jpeg = cv2.imencode('.jpg', image)
    return jpeg.tobytes()
```

Regarding the possibility to control the process, I have a question: would it be possible to add some kind of checking after the yield statement? Something like

```
@app.route('/stop_recording')
def stop_recording():
    control_var = "STOP"
    ....
    while True:
        frame = camera.get_frame()
        yield (b'--frame\r\n'
              b'Content-Type: image/jpeg\r\n\r\n' + frame + b'\r\n\r\n')
        if (control_var == "STOP"):
            ... do something
```

I am trying something similar with a javascript button but the server is never interrupted in the yield loop

#4 Miguel Grinberg said 2 years ago

@Tony: Yes, that should not be a problem as long as your cameras can be accessed. If you use OpenCV, you have to pass an index when you open the camera, that allows you to select the camera that you want, or open multiple ones at the same time. To serve multiple cameras from the same application you will have to modify my code to expose multiple camera endpoints.

#5 Miguel Grinberg said 2 years ago

@Jose Luis: Sure, the video writer is a good option to write the stream locally. Pausing the generator is not a good idea, because the client may timeout waiting for data and close the connection. If you want to create the illusion of the stream being paused, I suggest you keep sending the same frame over and over.

#6 David said 2 years ago

Hello,
I tried to do it with my Laptop camera but when I enter the Website I just see the Headline and no Video. Have you any Idee what I could do
Thanks

#7 Miguel Grinberg said 2 years ago

@David: I don't really know how to help you. You need to use a camera driver that works for your camera. If the OpenCV driver that I have created does not work with your camera then you need to find how to control the camera in Python, and then write a driver for that, similar to the OpenCV or Raspberry Pi ones that I have in the repo.

#8 Mabs said 2 years ago

Hi Miguel,
Excellent blog!
I am using the openCV class, and its working perfectly so far!
I am trying to make an an interface which allows a user to select a different VideoCapture() index, or give a file path to a video they want to stream.
I tried using a session variable to store user preference but since the

frame() method is a staticmethod, it doesn't like it when sessions are used.

How would you recommend implementing such a feature to the above code?

#9 Miguel Grinberg said 2 years ago

@Mabs: I just updated the opencv Camera class with a set_video_source() method. You can call it before streaming starts as follows:

```
Camera.set_video_source(1)
```

Whatever you pass as argument is then passed to the VideoCapture class, so you can use a device number or a filename. I hope this helps!

#10 Defcon said 2 years ago

Is there a way to display multiple streams on one page?

Something similar to this:

```


```

I have tried to do this, but it only shows whichever img src is first. I'm guessing its because it never gets to the second one? Is there a way around this?

Many Thanks

#11 Cosmin said 2 years ago

Hi Miguel,

Great article!

I setup the video server on a Raspberry Pi and was able to see the streaming on a network PC in Chrome and Firefox. However, in IE and Edge, the video is not displayed. Do you have any idea what could be the problem?

Thanks,
Cosmin

#12 Miguel Grinberg said 2 years ago

@Defcon: Are you using a multithreaded web server? If you are using Flask's web server with default settings, it can only handle one connection at a time, so your second stream will not work.

#13 Miguel Grinberg said 2 years ago

@Cosmin: My understanding is that both Edge and IE support Motion-JPEG natively, so I do not know what can be the problem. I will test on a Windows machine when I get a chance.

#14 Ben said 2 years ago

I'm a little confused as to why we're using `` rather than `<video>` to do this. `<video>` would seem to automatically support audio as well and allow the use of things like `<track>`. Is it just to eliminate worries about converting things to mp4/theo/webM from whatever the source is or experiment with an alternate method of doing things?

I'm currently writing a rather involved Flask app that adds a bunch of metadata to each frame which will likely be rendered in a `<canvas>`. Your method here looks interesting for that because there's an obvious moment to add code while processing each frame server side, but looking at other sites made `<video>` with a `<track kind="metadata">` attractive for the metadata and audio. I'm going to have a variety of sources including continuous-streaming ones like security cams so just straight hosting a bunch of mp4 files like youtube doesn't work. Have I missed something?

#15 Miguel Grinberg said 2 years ago

@Ben: Have you read my previous articles on video streaming? Generating a video stream in a format that the `<video>` tag can play on the fly with Python is hard. The Motion-JPEG format that I'm using here has the advantage that is easy to generate, please with fairly low latency, and works on pretty much any browser. You can certainly do mp4/webm/etc. but then you will have to have a more complex encoding strategy on the server, maybe based on ffmpeg or one of the libraries from that project.

#16 Ben said 2 years ago

Yes I read the first post. 'Generating a video stream in a format that `<video>` can play with Python is hard' Is the answer I was looking for. The good news is that it's looking like in at least most cases I'll just be able to play the original video and generate `<track>` data without needing to actually re-encode anything. Thanks for your help!

This site has been the first or second google result for a lot of my questions. I'm also in Portland so /wave from across the street! Thanks again, I appreciate it a lot!

#17 Peter Jackson said 2 years ago

This is not really relate to your example, it works great on chrome.

But the video image does not display on the Edge or internet explorer browsers.

Anyone have a fix?

#18 nicolas said 2 years ago

Many Thanks Miguel for this review !

I test it with opencv (python 3.62 / OCV 3.3 / FFMPEG 09/2010 / Windows10)

It's work great with iframe (on html) and so we can change on live !
brigness / sharpness ... we can put date/ time ... (no time to go as motion but ...) xD

Very good job !

For the record, opencv is not the good library I think .. so I do the record directly via ffmpeg (import subprocess as sp ...) and I creaye mp4 video file, can read them on all browsers with <video> tag ...
I will finalise on raspberry 3 with picamera and tell you (will test on Ubuntu also)

For Edge / IE ...

I test in canvas (as image from video_feed) / in div with css (image background video_feed) no sucess '(

You can read directly video live, via VLC ..

Thus ++ (from Paris)

#19 Islam Saad said 2 years ago

Many thanks for your great article and efforts.

Please, could you give me a hint how to control the frame rate as i use RaspberryPI camera?

#20 Miguel Grinberg said 2 years ago

@Islam: As I mention in the article, you can slow down the frame rate by adding a sleep call in the loop that grabs frames.

#21 Jug said 2 years ago

Hi Miguel, thanks for the article

Which version of gunicorn did you use for this review? When i start the the app in gunicorn 19.7.1 it raises the error:

```
[2017-10-07 10:56:36 +0000] [14201] [ERROR] Error handling
request /video_feed
Traceback (most recent call last):
  File "/home/pi/.virtualenvs/cv/local/lib/python2.7/site-packages
/gunicorn/workers/gthread.py", line 284, in handle
    keepalive = self.handle_request(req, conn)
  File "/home/pi/.virtualenvs/cv/local/lib/python2.7/site-packages
/gunicorn/workers/gthread.py", line 338, in handle_request
    for item in respiter:
  File "/home/pi/.virtualenvs/cv/local/lib/python2.7/site-packages
/werkzeug/wsgi.py", line 704, in __next__
    return self._next()
  File "/home/pi/.virtualenvs/cv/local/lib/python2.7/site-packages
/werkzeug/wrappers.py", line 81, in _iter_encoded
    for item in iterable:
  File "/home/pi/videostream.py", line 61, in gen
    yield (b' --frame\r\n' + b'Content-Type: image/jpeg\r\n\r\n' + frame +
b'\r\n')
TypeError: cannot concatenate 'str' and 'NoneType' objects
```

This happens because the function is trying to concatenate a string with the variable 'frame' which happens to be None, but only when i try to use gunicorn

Here's the chunk of code that raises the error:

```
@app.route('/video')
def video():
    return render_template('video.html')

def gen(camera):
    while True:
        frame = camera.get_frame()
        yield (b' --frame\r\n' + b'Content-Type: image/jpeg\r\n\r\n' +
frame + b'\r\n')
```

'frame' shouldn't be None

If i run the app from the command line:

```
$ python videostream.py
```

everything works fine.

I think it's a problem of compatibility between flask and gunicorn but i don't know how to fix it

#22 Miguel Grinberg said 2 years ago

@Jug: I'm also using 19.7.1 here and don't see this problem. The line numbers in your stack trace do not agree with the code I have in my repository, so you don't seem to be using a clean version of the code as I published it. Can you clone my repository and try with that code?

#23 Islam Saad said 2 years ago

@Miguel: I can increase frame rate to 25 fps by add command in
Campera_pi:
camera.framerate=25 (Is that right?)

Finally, I now face on PI terminal [socket error processing request] if
client stopped the connection. It's good if we can find a solution for
that.

#24 Miguel Grinberg said 2 years ago

@Islam: You can freely configure the camera settings according to
the picamera module, sure.

The socket error is benign, it just means that the server was in the
middle of writing a frame and the client went away. This isn't an
error in this application, it is an error that the web server isn't
handling, so the problem is not here.

#25 Islam Saad said 2 years ago

Hi Miguel

I see your great effort by streaming MJPEG.

I'm using Raspberry Pi.

I hope for using h.264 video port by picamera command:

start_recording(stream,'h264',video_port=True) but i can't image
how can i display this video using flask in video<tag> so how can i
use generator function get.frames?

I don't understand how could i do that, and your help is appreciated!

-
- [««](#)
 - [«](#)
 - [»»](#)

- [»](#)

Leave a Comment

Name

Email

Comment

Captcha

☐

I'm not a robot

reCAPTCHA
[Privacy](#) - [Terms](#)

Submit

The New Flask Mega-Tutorial

My [Kickstarter](#) project was a big success! Thank you to everyone who contributed to it!

If you would you like to support my work on this tutorial and on this blog and as a reward have access to the complete tutorial nicely structured as an ebook and a set of videos, you can now order it from my [Courses](#) site.

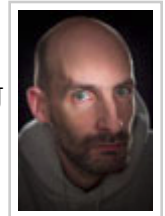


About Miguel

Welcome to my blog!


I'm a software engineer, photographer and filmmaker, currently living in Drogheda, Ireland.

You can also find me on [Facebook](#), [Google+](#), [LinkedIn](#), [Github](#) and [Twitter](#).



Thank you for visiting!

Categories

-  [AWS](#) (1)
-  [Arduino](#) (7)
-  [Authentication](#) (6)
-  [Blog](#) (1)
-  [C++](#) (5)
-  [Cloud](#) (8)
-  [Database](#) (13)
-  [Docker](#) (1)
-  [Filmmaking](#) (6)
-  [Flask](#) (81)
-  [Games](#) (1)

-  [HTML5](#) (1)
-  [Heroku](#) (1)
-  [IoT](#) (8)
-  [JavaScript](#) (8)
-  [MicroPython](#) (8)
-  [Microservices](#) (2)
-  [Movie Reviews](#) (5)
-  [Node.js](#) (1)
-  [OpenStack](#) (1)
-  [Personal](#) (3)
-  [Photography](#) (7)
-  [Product Reviews](#) (2)
-  [Programming](#) (111)
-  [Project Management](#) (1)
-  [Python](#) (107)
-  [REST](#) (6)
-  [Rackspace](#) (1)
-  [Raspberry Pi](#) (7)
-  [Robotics](#) (6)
-  [Security](#) (10)
-  [Video](#) (5)
-  [Webcast](#) (3)
-  [Windows](#) (1)

© 2012-2020 by Miguel Grinberg. All rights reserved. [Questions?](#)