UNIVERSITY OF MICHIGAN

Flask Video Streaming Project Information

Global Challenges

Version 1.0

UNIVERSITY OF MICHIGAN

Table of Contents

UNIVERSITY OF MICHIGAN

Revision History

Version	Changes
1.0	Initial Version

1. Introduction

This document explains how all the different parts of Flask Video Streaming work[1]. FIrst there is a walk-through of how to setup and run the project, followed by an in depth look at how all the different parts of the project work. At the end are some examples for how to add things to the project.

Flask Video Streaming is based on the work of Miguel Grinberg[2].

2. Installation Notes

When it comes to installing Flask Video Streaming, checkout the *requirements.txt* file. This lists all the python libraries the project needs, and most should be as easy to install as:

```
$ pip install <package name>
```

There will be a couple that may need to be installed by hand depending on the platform Flask Video Streaming is going to run on. If you are setting this up to run on a Raspberry Pi running Raspbian, you will (as of 8.0 Stretch) need to compile OpenCV and rpi_281x[3] as they are not prebuilt for the ARM platform. Also note that if you are going to use the rpi_281x library, you are going to need to run the server as root, so make sure that OpenCV is not installed inside a virtual environment.

3. Setup and Running

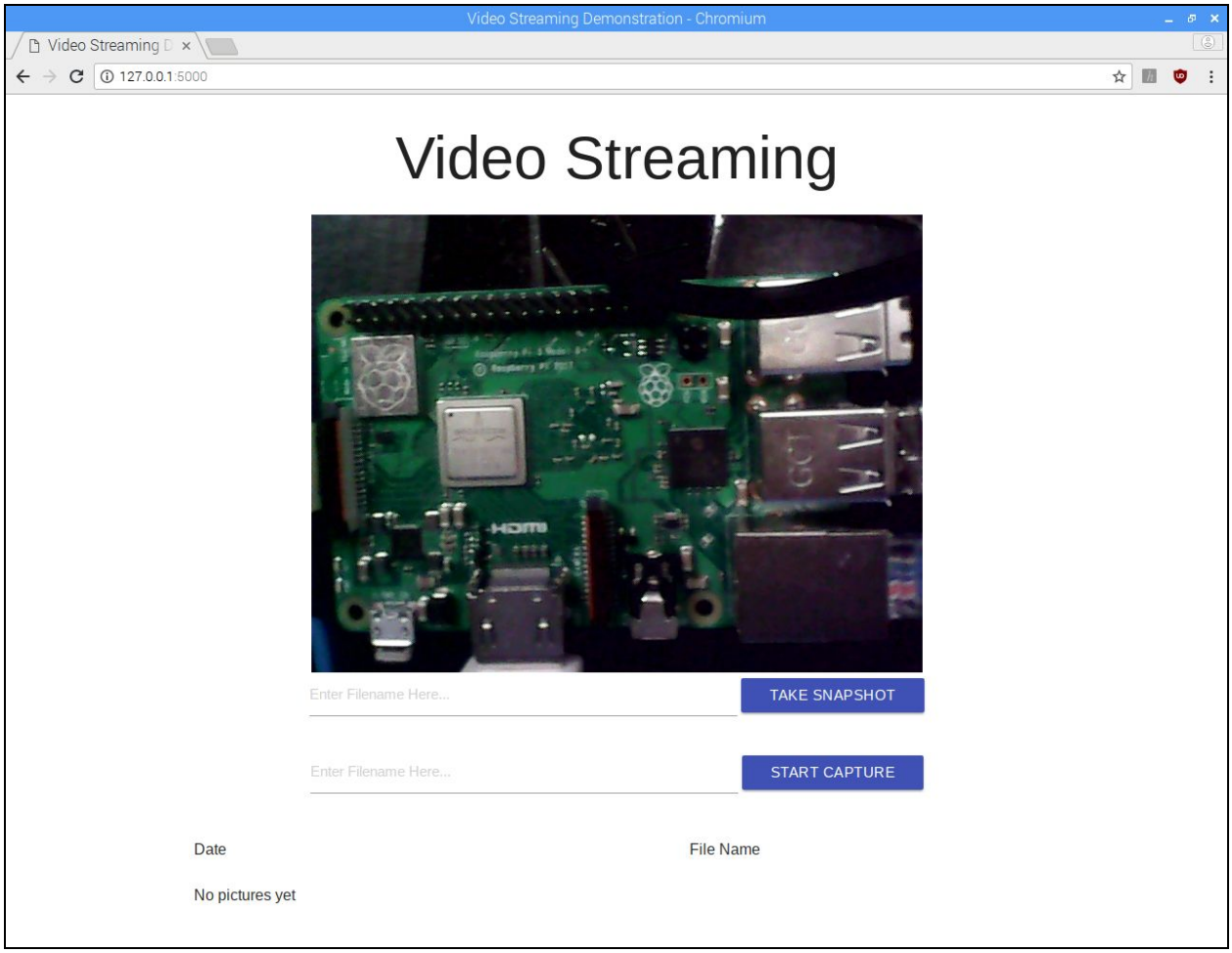
To get the Flask Video Streaming server running, it is as simple as just running the *flaskstart* script like so:

```
$ ./flaskstart.sh
```

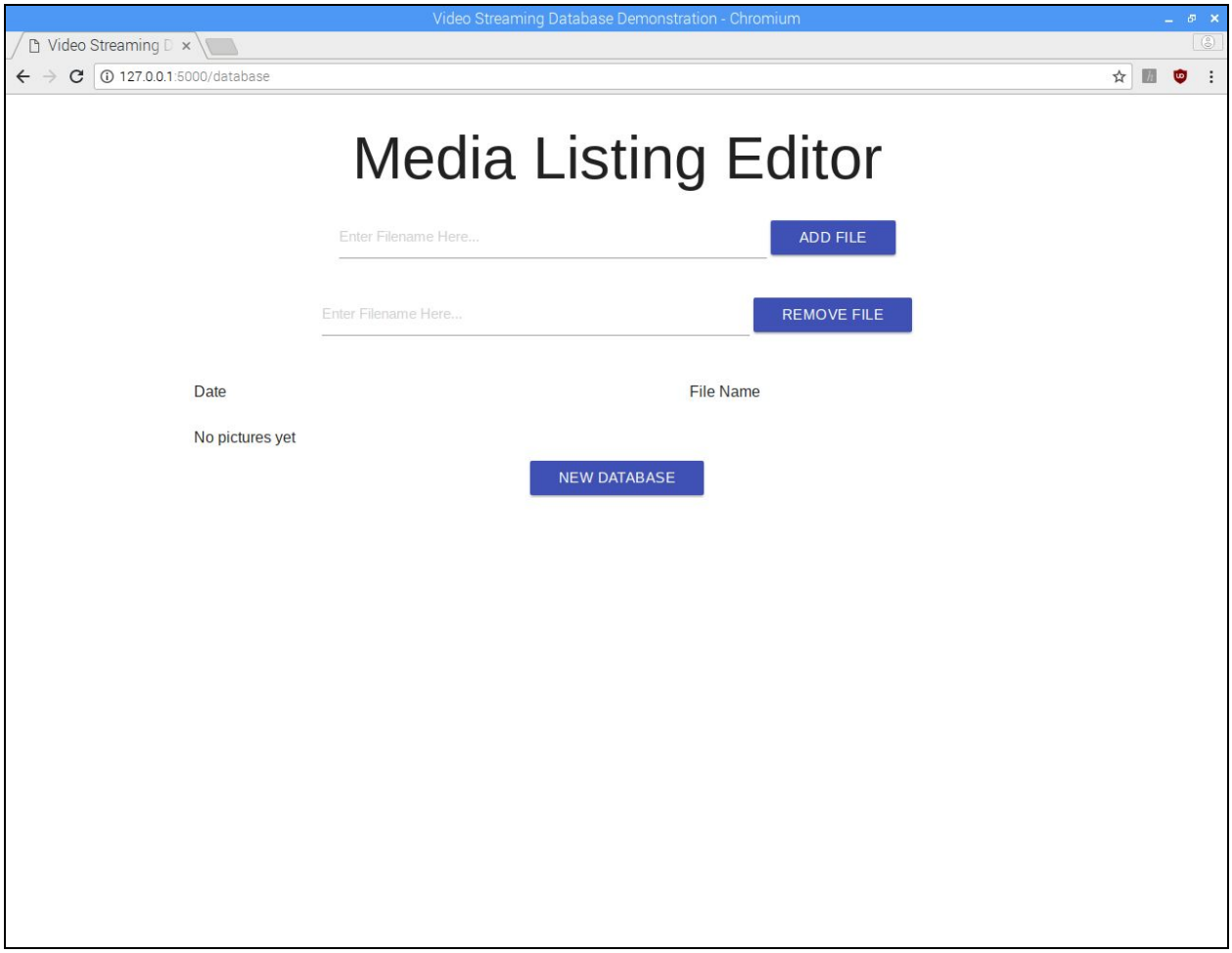
The terminal will then be blank as the server runs, with messages from the camera showing up, all other server information is logged to a file. To connect to the server, load up a web browser (any except IE) and go to 127.0.0.1:5000 and you should see a live feed from the camera.

---

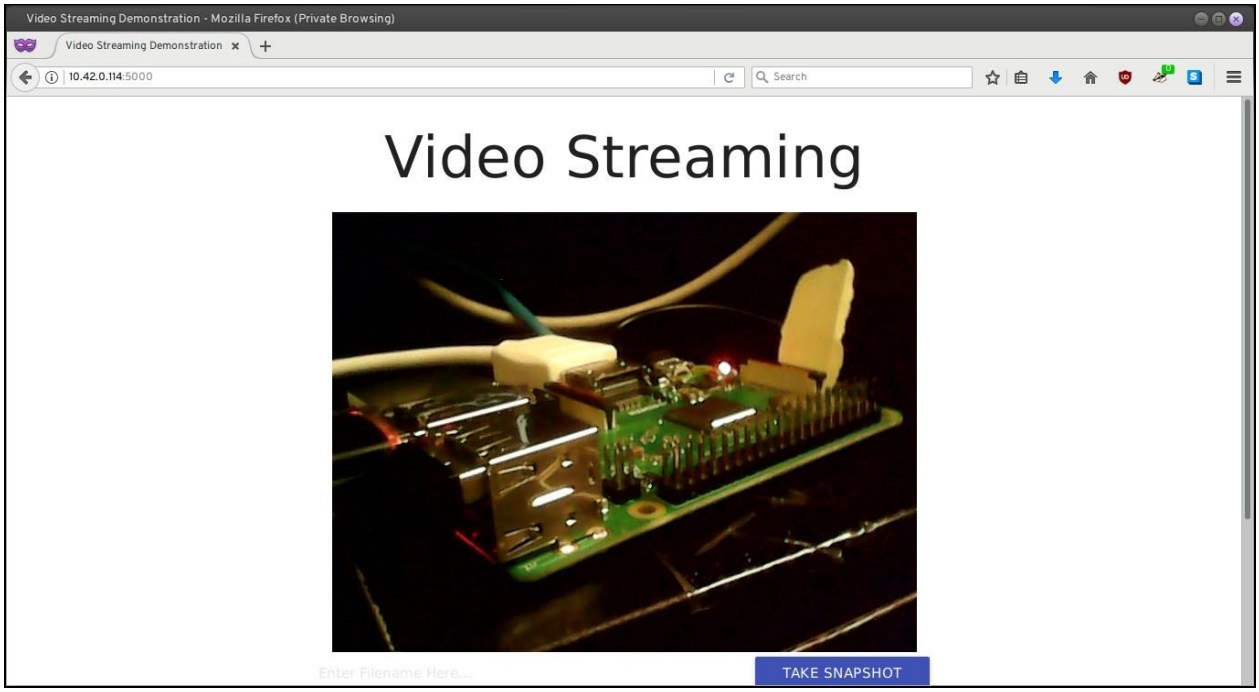
[1] For an in depth explanation of the API, see Flask Video Streaming API
[2] A link to Miguel Grinberg's Github repo can be found in the Appendix
[3] More information about rpi_281x can be found in the Appendix



Now to set up the database for saving pictures and video, go to 127.0.0.1:5000/database and click on the "New Database" button. Now you are all set to save pictures and videos.



To use this, point the browser of the device you want to view your stream from to the IP address of the computer running the server with :5000 appended to the end.



4. app.py Breakdown

There are two phases that Flask Video Streaming goes through when app.py is run. The first is the setup phase, where things are loaded, and the runtime phase, where the server response to requests. The first thing we do in the setup phase is read in settings from config.txt, which should be filled out before running. Based on different settings in config.txt, we know what sort of hardware we are going to need to talk to. The next step is connecting to the database, which is hard coded to look for as media/media.db . After all this is done, we enter the second phase. In the runtime phase, the Flask[4] server is started which handles all the web requests that are aimed at the server. The different routes that we want to have a response to are defined like so:

```
@app.route('/something')
def do_something():
    # Code to do things
```

For each of the routes that are defined, an explanation of what they do can be found in Flask Video Streaming API. The most important part of any route is what it returns, which in most cases should be the HTTP code 200. Any time we need to notify the user that something didn't go right a *Response* object is made, which gives a short explanation and a HTTP status code[5].

[4] More information about Flask can be found in the Appendix
[5] Listing of HTTP status codes used in this project can be found in the Appendix.

5. Camera Breakdown

There are two different classes for using cameras, but only one will get used at a time. For any USB camera, *Camera* from camera_usb.py is, but when a Raspberry Pi style camera is used, *Camera* from camera_pi_cv.py is used. Flask Video Streaming picks the camera based on what is given in the config.txt file, but will default to a USB camera if nothing is specified. Both classes are very similar, so unless it says otherwise it can be assumed things work the same in both or they have the same methods available.

a. Setting Up Stream

As long as *Camera* from camera_pi_cv.py is given mode=0 when `get_frame()` is called, both classes prepare themselves in very similar ways. The first thing `get_frame()` does is initialize the object, which will start a seperate thread to read images off the camera if there isn't one running already, and then return a frame that has been pulled off the camera. The worker thread either uses OpenCV or the Raspberry Pi camera library to read images from the camera, and continues to do this in a loop until there hasn't been a request for a frame in the past 2 seconds. This thread also watches the values of some variables to know if there is anything it needs to do with the images it is pulling off the camera.

b. Taking a Snapshot

For both versions of *Camera*, the `take_snapshot()` method is exactly the same. They try to save a frame from the stream using OpenCV's `imgwrite()` with the filename given to `take_snapshot()`. If this works without any problems, `take_snapshot()` returns a status code of 200, otherwise it returns a status code of 500.

c. Recording Video

For both versions of *Camera*, the `take_video()` method is exactly the same. `take_video()` uses the value passed through status to decide what way to set the flags for the worker thread. `take_video()` uses the convention of False sets the flags to start a recording and True sets the flags to stop a recording as well as save off how long the recording was. When the flags are set to start recording, the worker thread starts placing a copy of each frame from the camera into a queue. This continues until the flags are set to stop recording, when the worker thread creates another thread that has the job of saving off the video. The worker thread gives the video saving thread a copy of the queue it just made, as well as the length of time it took to record that and number of frames in should have saved to the queue. At this point the worker thread creates a new queue for saving the next recording into and resets the time and number of frames. The

video saving thread then has the easy job of using OpenCV to write a video file using the queue of frames. Once it finishes writing the frames, this thread stops.

d. Image Controls

For both versions of *Camera*, the two image control methods are exactly the same. These methods are `set_grayscale()`, which toggles giving the image a grayscale filter, and `drop_resolution()`, which toggles reducing the resolution of the image by half. Both methods work by changing some flags that the worker thread checks after reading an image off the camera. When the worker thread sees the flag for grayscale, it converts the image to grayscale, then if it sees the flag for smaller resolution, it shrinks the image by a factor of 2. After this, the worker thread then continues with normal operation.

e. Focus Controls

Currently only *Camera* from camera_usb.py has support for changing the focus and focal depth. There are no corresponding methods in *Camera* from camera_pi_cv.py, so trying to use these methods will result in an error.

There are three methods for controlling the focus and focal depth in *Camera*, those are `change_autofocus()`, `step_focus()`, and `set_focus()`. These three methods will all return a status value of 500 if the camera type is not set to 'LiquidLens', as that is the only camera we currently have that can change focus. Using `change_autofocus()` allows you to toggle between using the camera's autofocus feature and not, and it uses the convention of False turns it on and True turns it off. Both `step_focus()` and `set_focus()` turn off autofocus and change the current focal depth of the lens. `step_focus()` does this by only making incremental steps in or out, while `set_focus()` allows you to set the depth to a specific value. All of these commands use OpenCV to change their values.

6. Led Breakdown

To control lights attached to the device the server is running on there is a class called *Led*. Currently *Led* only supports controlling Neopixels attached to the GPIO pins of a Raspberry Pi[6]. To turn the lights on and off, there is the method `power_led()`. This will remember the color of the light before it was turned off. Changing the brightness is done with `set_brightness()`, and it looks for a value from 0 to 255. Since Neopixels can be set to any RGB combination, you can set a custom color with `set_color()` and build a *Color* object for it with `build_color()`.

[6] It does this with the help of some additional libraries, see Appendix.

7. Web Page Breakdown

There are two web pages that Flask Video Streaming servers, index.html (found at /) and database.html (found at /database). Index provides a live feed of the video stream, as well as controls, and a listing of the database. Database provides controls to manipulate the database. Both of these pages use similar techniques to send data back from the web page to the server, as well as providing information from the database to the web page.

The HTML files that describe the web pages contain tags for buttons, and each button tag is encapsulated inside a form tag. On their own, these buttons won't do anything when the user presses on them, so both web pages have a javascript file that gets sent and loaded with the web page. The javascript code is then run in the browser, so now when a button is pressed, the browser runs any code related to button presses. The javascript used in both web pages uses a library called JQuerry, which allows us to read and manipulate the HTML currently loaded in the browser. When a button is pressed or clicked, the javascript code with the name for that button is executed. In most cases, the code that is run reads information from the HTML, then makes a HTTP request to some API endpoint with that information. Then the javascript will edit values in the HTML, usually to change the state of a button, so it can use that for the next time it gets called.

When information needs to be added to the web page by the server before it gets sent out, the python code passes variables to `render_template()`. Then in the HTML, there is some scripting done in {% %} tags which gets filled in based on what was in the variables. An example of this is in the route for / which passes a list of all the entries in the database like so:

```
return render_template('index.html', pictures=entries)
```

Then in the index.html file, in the table for all the media information we have:

```
<tr>
{% for entry in pictures %}
</tr>
```

Which makes a new row of information for each item in the list of pictures.

8. Database Information

For keeping track of the pictures, video, and other data related to them, the project uses a SQLite database. The database is defined in the media_schema.sql, with fields for filename, date, and id (which is the primary key). The database itself lives in the media

folder as media.db, with all the pictures and videos saved. When Flask Video Streaming starts up, it tries to connect to an existing database, but if it can't find one, one will need to be created. By going to /database you can create a new database, as well as edit what is in it. There is also a python command line tool for doing all the same things called database_manager.py.

9. Example Additions

a. Adding a Route

Routes are specific URLs that the Flask server listens to and will give a response to based on what is defined. Flask makes adding a new route as easy as making a function, with a tiny bit of boilerplate before the function definition. The template for a route definition for a basic route is:

```
@app.route('/something')
def do_something():
    # Code to do things
```

Where 'something' is the name of the route, and `do_something()` is the function that will get called when 'something' is visited. The name of the function can be any valid python function name, and the route can be as long or short as desired. If you need to control the ways a route can be accessed, you can define the specific HTTP verbs that can be used on it like so:

```
@app.route('/something', methods=['POST'])
def do_something():
    # Code to do things
```

Here only if the request is a POST will the function `do_something()` be called, any other request will be given a 404.

The function that gets called when a route gets access is responsible for giving a proper response. For a route that needs to return a web page, the function should return the result of `render_template()`. When a specific response code is desired, return a *Response* object.

More information about routes can be found in the Flask documentation.

b. Adding a Button to the Web Page

There are two places for all the details for a new button to go, the first being the HTML file that will display the button, and the second being the javascript file that gets loaded with the HTML file. For this example we are going to add a button to

index.html, which already has a bunch of buttons on it that can be a good example of what to do. Our new button can go anywhere that is valid for a button (somewhere inside the <body></body> tags), but to keep it grouped with the other buttons on the screen, we will add it to the table of buttons.

Start by adding the following inside the <table> </table> tags:

```
<tr>
    <td>
        <center>

        </center>
    </td>
</tr>
```

This creates a new row and column in the table, where we will make our button. Since we are going to want our button to remember a value and be able to pull all that data out with javascript, we are going to need a little more than a simple button tag. We are going to wrap everything up in a form, and add an input for keeping track of our value. Inside the   tags add:

```
<form id="Form10">
    <input id="something" name="value" type="hidden"
value="10" />
    <button type="button" id="something_button" class="indigo
waves-effect waves-light btn" value="null">
        Title Text
    </button>
</form>
```

To break this down, the form has an id which should be unique (so check that Form10 isn't already being used). Both the input and button also have ids, which can be whatever we want but should also be unique. The name given to input will be what is used by the javascript to send the value of input back to the server in the URL. In this case it will send value=10, but it could be status=true or color=purple. The input's type and button's type and class define how they will appear on the web page when it gets loaded. Title Text is what is going to appear on the button when the page gets loaded.

Now to get this to do anything when the user interacts with the button, we need to edit the index.js file. In this simple example, we only want to send the value of the button when pressed. This can be easily done with the following:

```
$('#something_button').on('click touch', function() {
      url_params = $('#Form10').serializeArray();

      $.ajax({
            url: '/something',
            data: url_params,
            type: 'GET',
            contentType: 'application/json;charset=UTF-8',
            success: function(response) {
                  // Stuff you want to happen on a response
            },
            error: function(error) {
                  // Stuff you want to happen on an error
            }
      });
});
```

To break this down, we start with `$('#something_button')` which ties this function to the button we made earlier as it has the id of something_button. The `on('click touch', …` makes it so if the button is clicked or touched on a touch screen then the function will trigger. We then set url_params to being the contents of of the form we made (which has id="Form10"), which will get our value of 10. Then comes `$.ajax`, which allows us to send the server something but also listen for a response and act on it. In this example we only bother sending our data to the route '/something'[7] and don't bother with doing anything else. Other function in index.js show what can be done with success and error responses.

10. Appendix

For more information about the server API, see Flask Video Streaming API.

For more information about Miguel Grinberg's work see his Github repo.

More information about the Flask framework can be found here.

The *Led* class makes use of neopixel.py (which can be found in the git repository) and the rpi_281x library (rpi_281x Github).

[7] This route doesn't exist, but used for demonstration purposes. See Flask Video Streaming API for more information about what the server's API is.

HTTP Status Codes:

200	Everything is ok
401	Unauthorized
403	Forbidden
404	Page not found, or service not found
500	Internal server error