UNIVERSITY OF MICHIGAN

Flask Video Streaming Project Information

Global Challenges

Version 1.1

UNIVERSITY OF MICHIGAN

Table of Contents

Revision History

Version	**Changes**
1.0	Initial Version
1.1	State at Matt handoff. Adding generic payloads, and changed external devices.

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

Before running the server for the first time, you should check to make sure the configuration file for the server is set up properly. This file should be named `config.txt`, and an example of what goes in this file can be seen in the `config.txt.example`. The important items in this file are the `camera`, `light`, and `userkey`. All the other values are optional, or tied to having a flag set. Setting the camera variable to match the kind of physical camera used with the server is critical, otherwise the server will be unable to stream any video. OpenCV supports many USB cameras, though when cameras have special functionality, there may be values for those specific cameras (like the LiquidLens setting for cameras that can adjust their focus). If the server has a way to control a light, this variable should be set to true, and then `lighttype` should be defined to either be neopixel (which is depreciated) or trinket

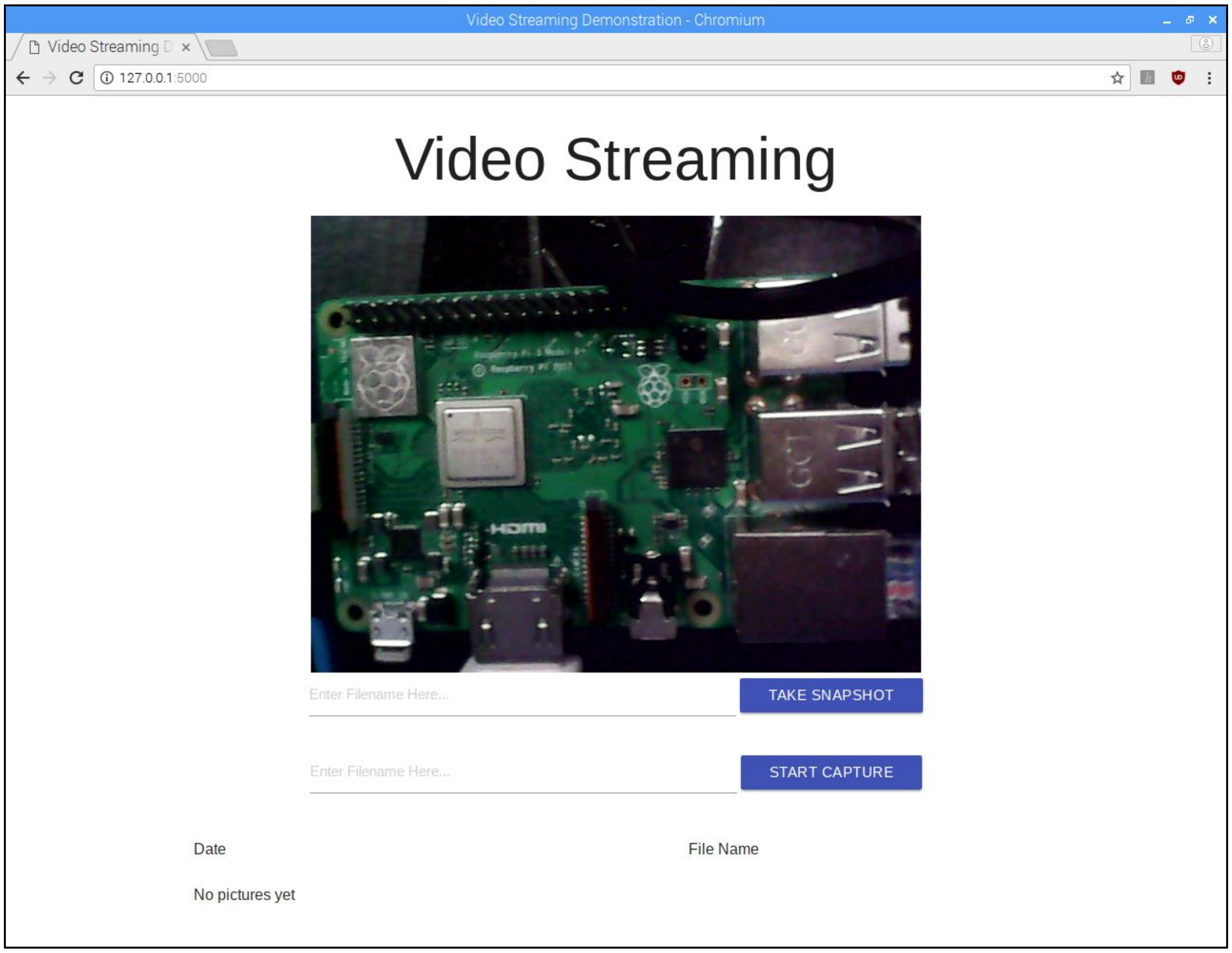
[1] For an in depth explanation of the API, see Flask Video Streaming API
[2] A link to Miguel Grinberg's Github repo can be found in the Appendix
[3] rpi_281x is no longer used, but if needed more information about rpi_281x can be found in the Appendix

(for a controller connected over serial). The last key setting is `userkey`, which is a string that allows for special access to server functions.

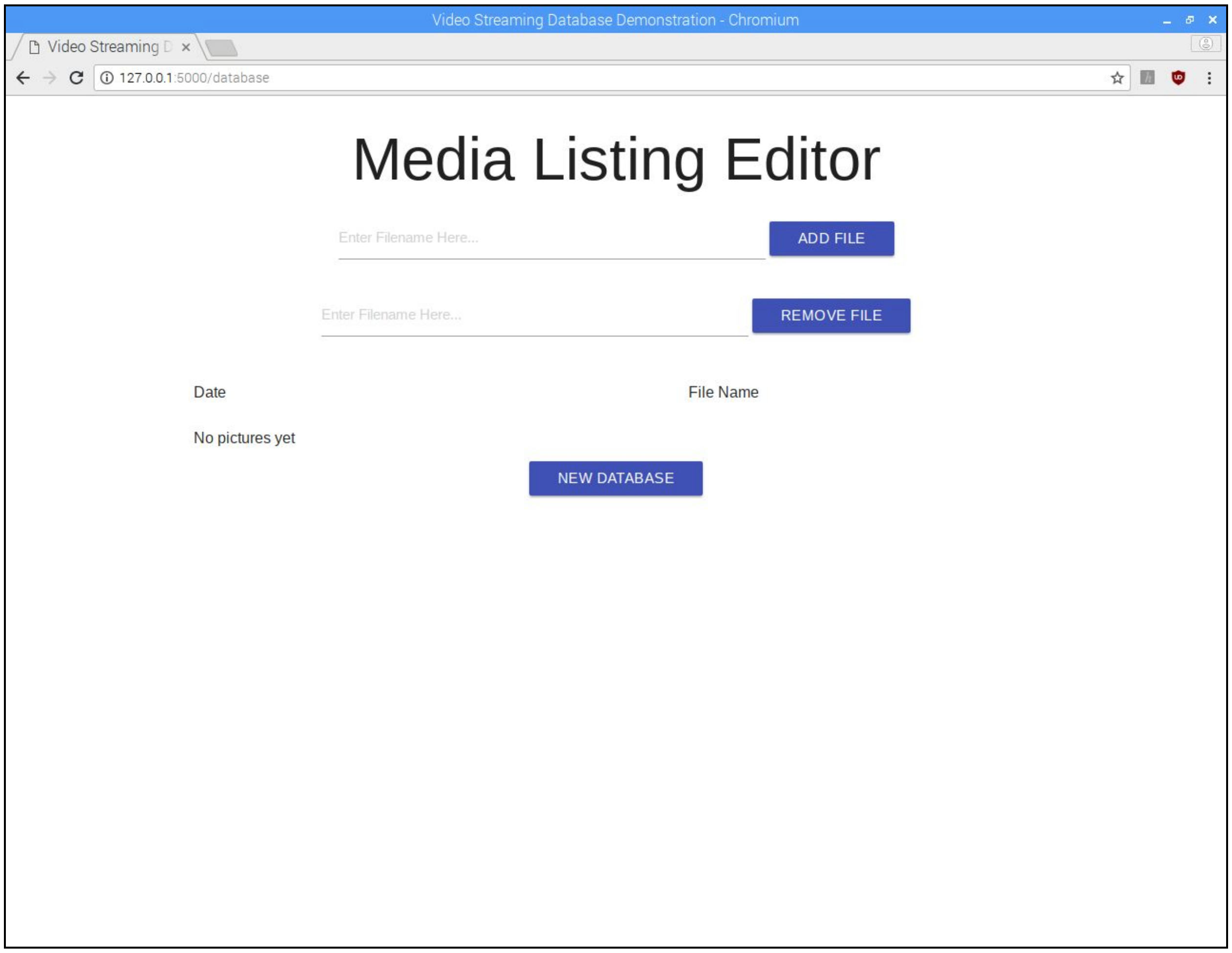
To get the Flask Video Streaming server running, it is as simple as just running the *flaskstart* script like so:

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

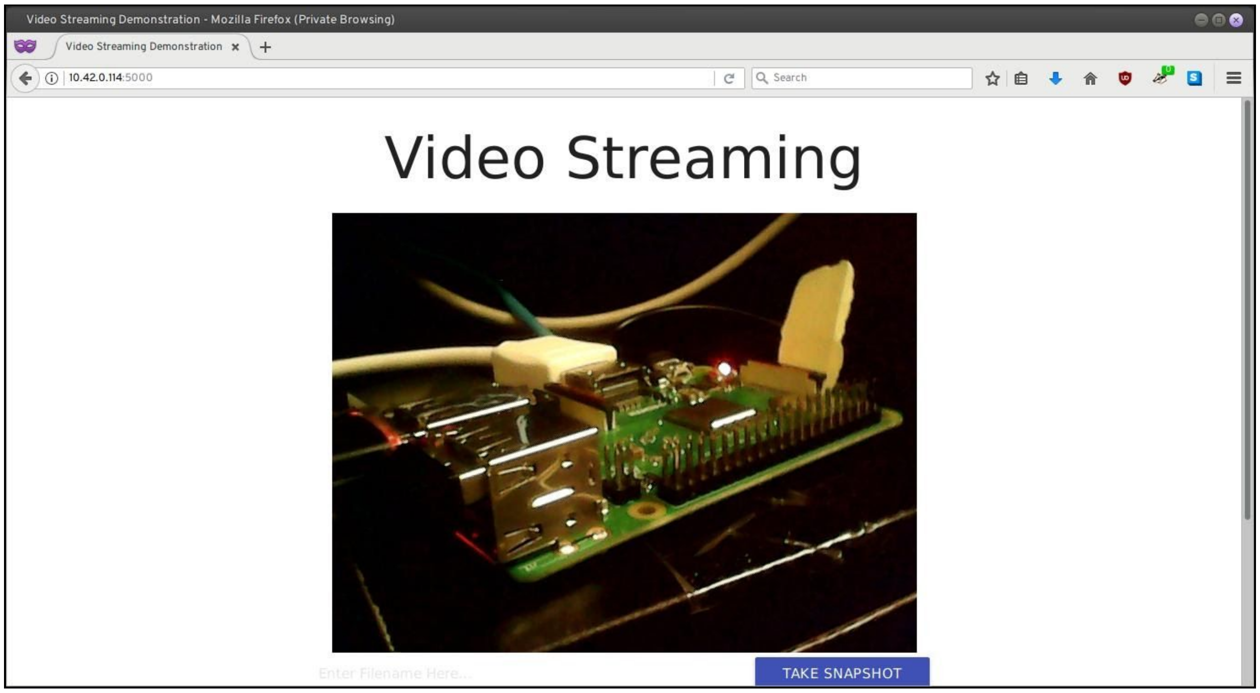
The terminal will then be blank as the server runs, with messages from the camera showing up, all other server information is logged to a file. To connect to the server, load up a web browser (any except IE) and go to 127.0.0.1:5000 and you should see a live feed from the camera.



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.


Video Streaming Demonstration - Mozilla Firefox (Private Browsing)
Video Streaming Demonstration
10.42.0.114:5000
Search
Video Streaming
Enter Filename Here...
TAKE SNAPSHOT

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 responds 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.

f. 'Generic' Payload

The 'generic' payload is to help make it easier to quickly add functionality to the camera class, without having to define new routes on the flask server. The idea is that from a single route (in this case it is `/manual_video` on the flask server), any arguments that were given to the server would then be passed to this section where actions for those arguments can be acted on. From the route on the flask server, everything is passed to `video_payload()` which then decides what to do with the information passed to it. As many of the camera functions may be taking place on other threads or processes, this provides a common place for flags or variables to be set for those other threads and processes to use.

As an example to how this would be used, the ability to take a snapshot has been added to `video_payload()`. The first step is to check and see if the payload the user wishes to use matches the name for the functionality we want to provide. So in this case we check for `payload` to match ‘snapshot’. Any other arguments that were also given to the flask server are then available as part of the `args` variable (think of it as a python dictionary). Since the user may not have provided a filename to save the picture as, we generate a default filename before checking `args` if they did. We can then just call the `take_snapshot()` function as if the user had made `/snapshot` request to the flask server.

Often the payload that will be needed will be more complex, either because it is needing to change some OpenCV functionality to the camera stream or to do some other processing. All of these will follow the first basic steps of the snapshot example above, they need to check that the `payload` that is called matches the name of the functionality we are working with. Then they can look at `args` to get any other values that may be needed. From here though, things will change based on what and how the functionality is set up. For example, if there is some sort of OpenCV stuff that needs to be done with the camera stream, then this function should set some sort of variable in the *`Camera`* class that the camera `_thread()` will then access. The code to do whatever OpenCV thing will have to be added in the camera `_thread()`. On the other hand if the payload needs to do something else, such as do some analysis of image data, it may be best to have that run in a separate thread or process. In this case, the code in `video_payload()` should once again set some sort of variable(s) in the `Camera` class or elsewhere that can be accessed by the thread or process. The implementation of the thread or process will then be left to the reader as this is something that probably doesn’t already exist in the *`Camera`* class.

6. External Device Breakdown

There are currently two ways to control external devices, such as LEDs, from the server. The older method uses rpi_281x to control Neopixels directly, which is explained more under the ‘Deprecated’ heading, but is not recommended to use as it adds lots of extra processor overhead on a Raspberry Pi. The newer method uses an external device to directly control LEDs (like Neopixels) and/or other devices (like sensors). This is done by sending messages over a serial connection to an Adafruit ItsyBitsy on the Otoscope MD-03, though this protocol could use further development. This is an area that could use some cleanup work to make a better abstraction for what goes on.

a. Python (Flask Server) Side

Everything about talking with an external device is handled by the *`Led`* class in trinket.py. This class provides functions that the flask server can call for doing things such as turning the light on and off. Any messages from the external device are also accessible via the *`Led`* class from the flask server. The *`Led`* class talks to an external device via a serial connection, which on the Raspberry Pi is set up to be through the GPIO pins. The server can read in settings in the configuration file for setting up the serial connection, should the defaults not match what is to be used.

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()`. The *Color* class is provided in trinket.py.

b. Arduino Side

The Arduino code can all be found in the arduino directory of the project. A simple run down for how the code on the Arduino works follows. The Arduino has a simple serial connection to the Raspberry Pi, and inside the main program loop running on the Arduino it first checks if there is a serial message from the Raspberry Pi. If there is a serial message, then it treats it as information about what color to set the LEDs to (in this case they are Neopixel compatible). Then regardless of if there was a serial message from the Raspberry Pi, it sends a serial message to the Raspberry Pi of whatever the newest value is for the calculated heart rate.

Diving a bit deeper, the code to talk to the Raspberry Pi is done via a serial connection. Depending on the model of Arduino used, there may be multiple serial ports on the device itself. For the model we are using (an Adafruit ItsyBitsy), there are two serial ports, one for the USB connection and one for the RX and TX pins on the Arduino board. This allows for the Arduino to communicate with two separate devices, though in our case it does the same thing with reading color information and sending heart rate data to both serial connections. Any data read from the serial port is assumed to be color information encoded as a hex string (FFFFFFFF would be white, and 00000000 would be black aka off). The key thing about dealing with then sending this color information to the LEDs is knowing which type of LED is actually being used. As we have been using Neopixel style LEDs, they are controlled via the Adafruit

Neopixel library[6]. Make sure you install the correct Arduino code onto the Arduino based on which type of Neopixel is being used. The last bit about the code running on the Arduino is the most complicated, though should not need to be messed with unless the heart rate sensor is changed out. The heart rate sensor sends out a value based on how much light is reflected back from the skin and blood inside someone's body. This value spikes when the blood is pumped from the heart out to wherever the sensor is located, and then drops as blood leaves that area. The Arduino has interrupts set up to watch for these peaks and record the time between them. After a set amount of time, it then calculates a new value for the heart rate based on the number of spikes it saw. This value is what is sent back to over the serial connection.

c. ***Deprecated*** Raspberry Pi LED Breakdown

The following is retained from the previous version of the document. This all still works, though has been deprecated as it adds extra overhead and causes stuttering in the video stream so is no longer used.

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[7]. 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()`.

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

[6] Learn all about Neopixels from Adafruit here.
[7] It does this with the help of some additional libraries, see Appendix.

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 <center> </center> 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 `$('#something_button')` which ties this function to the button we made earlier as it has the id of something_button. The `on('click touch',` … makes it so if the button is clicked or touched on a touch screen then the function will trigger. We then set url_params to be the contents of the form we made (which has id="Form10"), which will get our value of 10. Then comes `$.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'[8] and don't bother with doing anything else. Other functions 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).

HTTP Status Codes:

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

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