**Software Documentation**

360-Degree Photography Project

*April 2019*

*Version 0.1*

|  |  |  |
| --- | --- | --- |
| Date | Version Number | Document Changes |
| 4/8/2019 | 0.1 | Initial Draft |
|  |  |  |
|  |  |  |

[**Introduction**](#_feexrc5jwow6) **4**

[Structure](#_5p98yai1t9jg) 4

[**Tower Software (Back-end)**](#_kswc1283kzoo) **5**

[server.py](#_yk6nez7vzeju) 5

[run()](#_tjye27c2pcnf) 5

[command\_receiver(dictionary/map)](#_4sx2mqqgry59) 5

[thread\_gc()](#_12x6czvbugya) 6

[Server Commands](#_8p20oftf3bky) 6

[camera\_control.py](#_gqnjpe43jhky) 7

[kill\_gphoto2\_process()](#_n7lh3prz6p5q) 7

[get\_ports()](#_e4i5xek5nzpp) 7

[organize\_file\_system()](#_r88y8637cuj8) 7

[cancel\_capture()](#_wli27pm54y1u) 7

[capture\_images()](#_mq6xyvjlvsa1) 7

[communication/blob.py](#_imjbpnr5v4et) 8

[communication/bus.py](#_9g7hlgwb7ws9) 8

[communication/serverutil.py](#_k7i91v3swgfe) 8

[**Communication to the Front-end**](#_rui34xwhbbw7) **9**

[How it works](#_5ie3resmpz1w) 9

[**React Application (Front-end)**](#_jbxtpwan652x) **9**

[**Out-of-Scope Proposals**](#_q0rq3alufboo) **10**

[Device Selection](#_dqtp3h45q5zc) 10

# Introduction

This document is aimed to assist in learning about the architecture and logic in the backend of the 360° Product Photography project. The backend is comprised of a Python application that automates the setup and capture of photography.

## Structure

Controller/

├── src/

| ├── communication/

| | ├── \_\_init\_\_.py

| | ├── blob.py

| | ├── bus.py

| | └── serverutil.py

| ├── \_\_init\_\_.py

| ├── camera\_control.py

| └── server.py

├── test/

| ├── test\_camera\_control.py

| ├── test\_camera\_integration.py

| └── test\_communication.py

├── .env

└── README.md

The main driver program that we are looking for is server.py. This maintains the connection to Azure as well as using camera\_control.py to control the cameras. Some of the more important aspects of the project that we need are located until the communication/ subfolder, which has respective files for the Azure Blob Storage, Azure Storage Bus, and general utility commands that helps out around the project.

The bottom folder, test/, has three files that ensure the correctness of the application.

At root level, there is a readme that details more about how to run the application, and a “dotenv” file that will be used for establishing connections to Azure.

# Tower Software (Back-end)

The backend is comprised of a Python application that is covered within five Python files, as detailed in the previous section.

## server.py

The driver of the project.

When ran, it will compile together information that is needed to fulfill all of the connections necessary. It will load up the “dotenv” file that is located at the root of Controller/ and will use the environment variables declared around the project.

### run()

This function compiles together the communication pieces and hooks up to the camera controller module. First, it attempts a connection to the blob service, then to the message bus, then makes the controller object for use later. If either of the first two fail, then the server will automatically halt. The function ends up creating a couple of new threads for the project: one to continually watch for commands coming in, and another to collect “dead” threads. It was decided on that this was the best route so the server can handle multiple incoming requests and provide responsiveness to the front-end.

### command\_receiver(dictionary/map)

The command receiver takes in one argument, which is a list of the commands that the front end may use in order to manipulate the controller. It has a sub-function inside of it called fire\_thread(function, dict) which creates threads and stores them into a global array for the thread cleaner to find and clean after the thread is done executing.

The receiver sits in a forever-loop that will continuously listen for input in the queue it is told to listen to (specified by BACKEND\_QUEUE\_NAME at the root level below imports.) After receiving a message, then it will go through a couple rounds of verification to determine if it is a valid request. Such as: is the message not blank, and is the message a command that is specified in the commands that were passed? If it passed, it will fire off a new thread using the sub-function specified above, and will go back to listen for another command.

### thread\_gc()

A simple collector that loops through the active commands list and purges any thread that is complete. Does this operation every 5 seconds.

### Server Commands

The project comes with a few pre-defined commands that the front-end can use to control the cameras and images. They are: spree60, cancel (or “stop”, same functionality), upload, status, and reset.

spree60 sets up and begins the camera taking process. The front-end will send extra properties known as “custom properties” within their messages, which will tell the server how many images they would like to take. If it is a preview, then it communicates to the controller to take only one image from each camera. If it is a full capture, then it tells the controller that we need to do an entire rotation of images. Anything taken in the capture\_images() function is yielded back as a tuple, one is the path to the lower photo and the other is a path to the upper photo. This helps send the image directly to the front-end immediately after the capture. If the variable yielded was a number, then it will update the front-end on any progress that has passed. Otherwise, we are told that the process is over, and will let the front-end know that it is fine to stop listening for more pictures to come in.

cancel is a simple command that will tell the controller (if it is actively taking images) to stop taking images, and start canceling any threaded process that is running by setting an Event object.

upload will accept a couple custom properties from the front-end: lists of images to delete from the lower and upper cameras, respectively. It will run through each folder and delete the images that were specified in the lists. After that, we move every photo to a separate folder, to\_azure, located where the photos are generally at, so that the person setting up products for taking images can immediately start another session of images while it is uploading to the blob. After moving to the new folder, we tell the blob object to take that directory and upload everything into its respective blob.

status just returns what is happening in the server currently. Not implemented in the front-end.

reset nukes everything and starts from the beginning. Useful if the user does something bad and can not get back to what they were doing. Or, something happens in the project that we have not found yet.

## camera\_control.py

The module for controlling the 2 connected DSLR cameras. Contains common Exceptions thrown by the cameras during capture. Also contains the Controller class, a class designed to be instantiated from other python files. This creates an instance of a controller that allows many useful functions for controlling the cameras.

### kill\_gphoto2\_process()

This function is used to kill any gphoto2 processes currently running on the tower. When cameras are plugged in, a process for gphoto2 automatically starts for each camera that is connected. When a gphoto2 process is already running for a camera and another process attempts to use the camera, an error occurs. To prevent this error, this function gets the list of any gphoto2 process currently running and kills it to allow the rest of the code to access the cameras.

### get\_ports()

A function to use a gphoto2 subprocess “--auto-detect”, which returns a list of any usb ports of the tower that have cameras connected. This is used to parse the usb ports out of the process output for the cameras so that they can be controlled separately.

### organize\_file\_system()

A function that ensures that all directories needed to store files exist and are empty before starting the capture process. These directories include: “photos”, “photos/upper”, and “photos/lower”.

### cancel\_capture()

A setter for setting a class-owned boolean cancel to true to let any other threads capturing images with the Controller object know that it should stop its process and return.

### capture\_images()

The main function for capturing images with the DSLR cameras. This function calls the other functions above to ensure that the tower is ready to start the capture process. It calls kill\_gphoto2\_process, organize\_file\_system, and get\_ports before setting/checking the function inputs for camera capture. The function is given two inputs, duration and interval. If the input duration is null, then the duration is set to 1 to take a preview image from each camera. Two threads are created to handle sending the photos to the front end to decouple the capturing and sending of images. The main thread captures images while the current time passed is less than the duration. The two created threads wait for filenames to be added to a queue so that they can send the files with those names. The interval input allows the front end to specify the amount of time in seconds between each image capture. Increasing this time can potentially lead to more consistent photo numbers for each product. This function also catches any errors encountered by the cameras and reports them to the front end to be displayed. Overall, this function controls camera capture for duration seconds with interval seconds between each capture.

## communication/blob.py

The blob file defines a BlobService object that establishes a connection through a connection string and has an upload function that takes a folder and sends everything off. There is a debug function that will list off every container within Azure and its respective blobs. Be careful if there are a lot of products uploaded… otherwise say goodbye to your console screen.

## communication/bus.py

The bus file defines a BusDriver object that provides a simple API for the server to communicate with. This handles the connection establishment as well as ensuring that the two queues required for use exist. It alleviates a lot of the uncleanliness that was in the project before moving to an object-oriented structure.

## communication/serverutil.py

This is your simple utility class that will help out with writing up code for the server.

# Communication to the Front-end

Communication within this project is facilitated through Spreetail’s Azure account, namely the Azure Message Service. All of the functionality needed for talking to the front-end is located in the Controller/communication/bus.py file.

## How it works

The server.py function run() holds the bulk of the setup process for starting this project up. The object called bus\_driver is instantiated, automatically creating the connection to the message bus. When the server would like to send a plain text message out to the front-end, all you need to do is call the bus\_driver’s send\_message function, which only needs a string (or bytearray… Azure Service Bus automatically converts to byte array if it’s a string.)

# React Application (Front-end)

The React application controls and communicates with the hardware tower through the azure service message bus. It is created using Spreetail components and styles. The application was built in the Visual Studio Code editor with prettier and eslint extensions. Eslint uses AirBnB eslint configurations.

## WebApp\root

##### .env

All of the Azure Service Messaging Bus secrets and keys are stored here. The settings for the capture duration and interval is also stored here. The capture duration is the number of seconds the camera will take pictures for. The interval is the seconds in between each camera shot.

##### App.js

The main part of the application that contains components and states passed to those components. Contains functions to handle sending messages up to the service message bus for capturing a preview and beginning the full process.

## root/src

### Components

#### Preview.js

This component is responsible for rendering the preview page and provides its functionality. It contains all preview event handlers (e.g. handle preview button click). All states that this view needs are wrapped in this component except the state that disables all buttons when the output view is busy (i.e. a process is running).

##### PreviewBox.js

This component is used inside of the preview component to render the preview images. It renders the two image boxes in the preview page. By default, the component renders a grey-colored-background image until the user clicks preview to capture two preview images.

#### Output.js

This component is used to display the output view which contains a grid list of images that have been taken. It also provides functionality that allows user to interact with these images. For example, user can select/unselect images, retake the images, cancel the process while images are being taken, and finally save images to the storage blob.

#### ImagesContainer.js

This components renders the images grid list that is displayed on the output view. Users can unselect/select images from this list. The component uses an images prop that is passed to it from App.js.

#### PreviewImages.js

This component is responsible for displaying the images inside the image container in a larger modal. The library that makes this possible is Lightbox. Lightbox takes in functions to decide what index it should be iterating through (given an array of images). These functions are stored inside the Output.js component and are passed in.

#### Modal.js

This component is responsible for displaying different type of modals. This happens by passing in variables (such as button texts, display message text, class names, etc.). It is also possible to decide what type of modal it is (such as a confirmation modal).

### Styles

The styles are split up into four separate files. Index, Global, Output, and Preview. The styles files are imported into the Index style file. Global handles any general styles (e.g. modal styles). The Output and Preview styles handle the styles for their respective components.

### Utilities

Contains helper file containing functions used throughout the application and in various components. This includes functions that create the service bus service by providing namespaces and keys from the environment variable file (.env), clearing the messaging queue before beginning any capture requests, handling non-numeric inputs for html input, and the polling system for pulling messages from the service message bus queue.

# Out-of-Scope Proposals

We got through enough in the project to complete what was needed, plus a little extra. However, there were some ideas that we had for the tower that we were unable to get to due to time constraints and blockers.

## Device Selection

The front-end would have held a screen that would allow the user to select the device they want to control. As mentioned above, the tower would create a couple of queues, one dedicated for receiving and one for sending. Having the ability to pick from an array of potential devices would allow for the use of multiple towers in this project.