OpenFlexure Microscope Software Documentation

Bath Open Instrumentation Group

Sep 21, 2021

Contents:

1 Quickstart **1**
- 1.1 Install . . . 1
- 1.2 Usage . . . 1
- 1.3 Managing the server . . . 2

2 Web Application interface **3**
- 2.1 Navigate pane . . . 3
- 2.2 Capture pane . . . 4
- 2.3 Settings pane . . . 7
- 2.4 Gallery pane . . . 8
- 2.5 Interface structure . . . 8

3 Microscope settings **11**
- 3.1 Microscope settings file . . . 11

4 Microscope class **13**

5 Camera Class **15**
- 5.1 Raspberry Pi Streaming Camera . . . 15
- 5.2 Base Streaming Camera . . . 18
- 5.3 Capture Object . . . 19
- 5.4 Capturing from a camera object . . . 20

6 Stage Class **23**
- 6.1 Sangaboard Microscope Stage . . . 23
- 6.2 Base Microscope Stage . . . 24

7 Developing API Extensions **27**
- 7.1 Introduction . . . 27
- 7.2 Basic extension structure . . . 28
- 7.3 Adding web API views . . . 29
- 7.4 Marshaling data . . . 31
- 7.5 Thing Properties . . . 35
- 7.6 Thing Actions . . . 39
- 7.7 Threads and Locks . . . 42
- 7.8 OpenFlexure eV GUI . . . 46
- 7.9 Lifecycle Hooks . . . 52

8 HTTP API **53**
8.1 Live documentation 53

9 Indices and tables **55**

Python Module Index **57**

Index **59**

CHAPTER 1

Quickstart

1.1 Install

1.1.1 Stable installation

The OpenFlexure Microscope software is designed to be run on the embedded Raspberry Pi, in an OpenFlexure Microscope. For most users, our pre-built Raspbian SD card image. is the easiest way to get started. This SD card image is based on Raspberry Pi OS and includes both the microscope server and OpenFlexure Connect. A desktop shortcut will directly start OpenFlexure Connect if you are using the Raspberry Pi directly, or the microscope can be controlled over the network with its default hostname *raspberrypi.local*.

1.1.2 Manual installation

To install the server on a Raspberry Pi without using the pre-built OpenFlexure Raspbian image, or to install the server on a different system (this is useful for development), follow the instructions in the README file at the top level of the project's repository.

1.2 Usage

The easiest way to use the microscope is through OpenFlexure Connect, our cross-platform application that handles discovering and connecting to the microscope. It is detailed on the instruction page on our website including a download link. OpenFlexure Connect is pre-installed on the full SD card image (not the "lite" image, as this does not have support for a graphical desktop).

If you know the hostname or IP address of your microscope, you can also connect to the same interface using a web browser by entering *http://microscope.local:5000/* as the address. *microscope.local* is the default hostname of the microscope if you use our pre-built SD card image. If you know the IP address or have customised the hostname, you can use that instead. Note that the hostname is announced via mDNS, which is usually reliable if the microscope is connected via a network cable directly to your client computer but may not work if both devices are connected to

a more complicated network. As support for mDNS varies between operating systems, OpenFlexure Connect often detects microscopes even if you cannot resolve the microscope using the mDNS hostname.

Whether you connect with OpenFlexure Connect, or through a web browser, the web application interface is the same. See the "web application" section of this manual for more details.

1.3 Managing the server

Managing the server through the installer script's CLI is documented on our website.

This includes starting the server as a background service, as well as starting a development server with real-time debug logging.

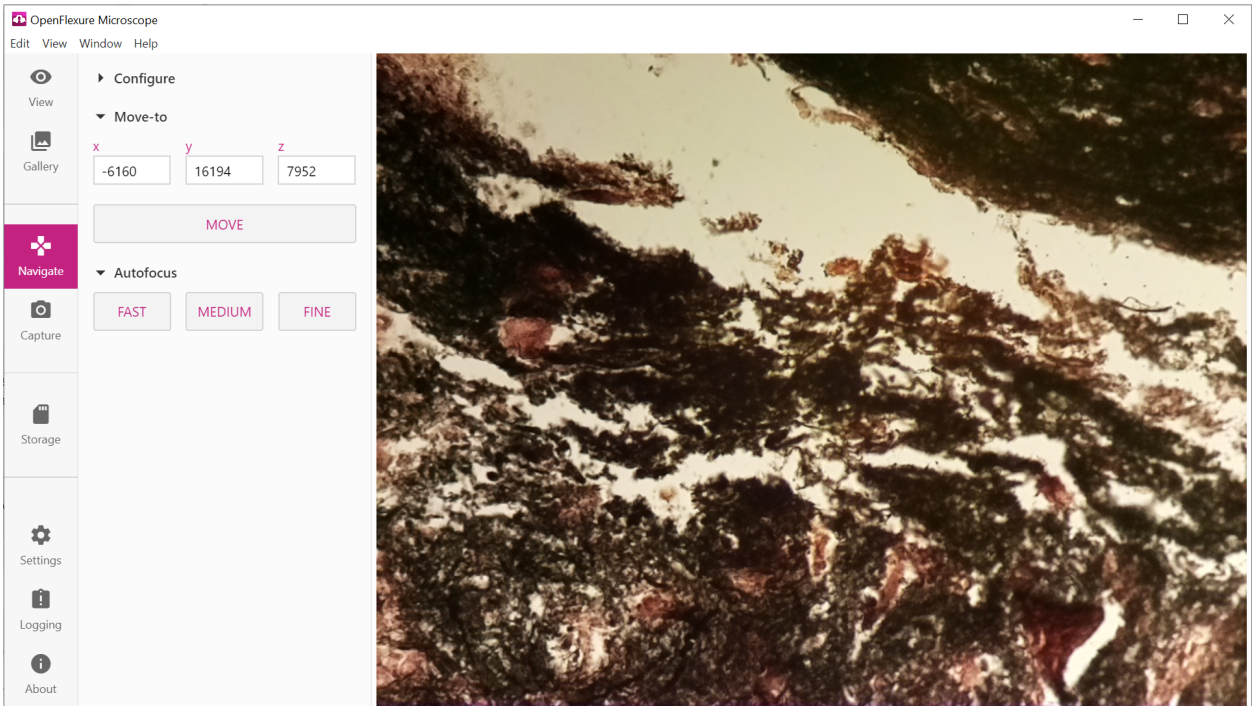
CHAPTER 2

Web Application interface

2.1 Navigate pane

The navigate pane displays the current stage position. Editing the values for X, Y, and Z and then hitting "enter" or clicking "move" will move the stage to the specified coordinates. Using the arrow keys (or page up/down) when not editing a text box will also move the stage, and the step size used can be set in the "configure" section at the top of the pane (which is collapsed by default). Using the mouse scroll wheel on the image will also move in Z, using the same configured step size. Double clicking on the image will bring the point clicked to the centre of the field of view, if the camera-stage mapping has been calibrated (see *Settings pane*).

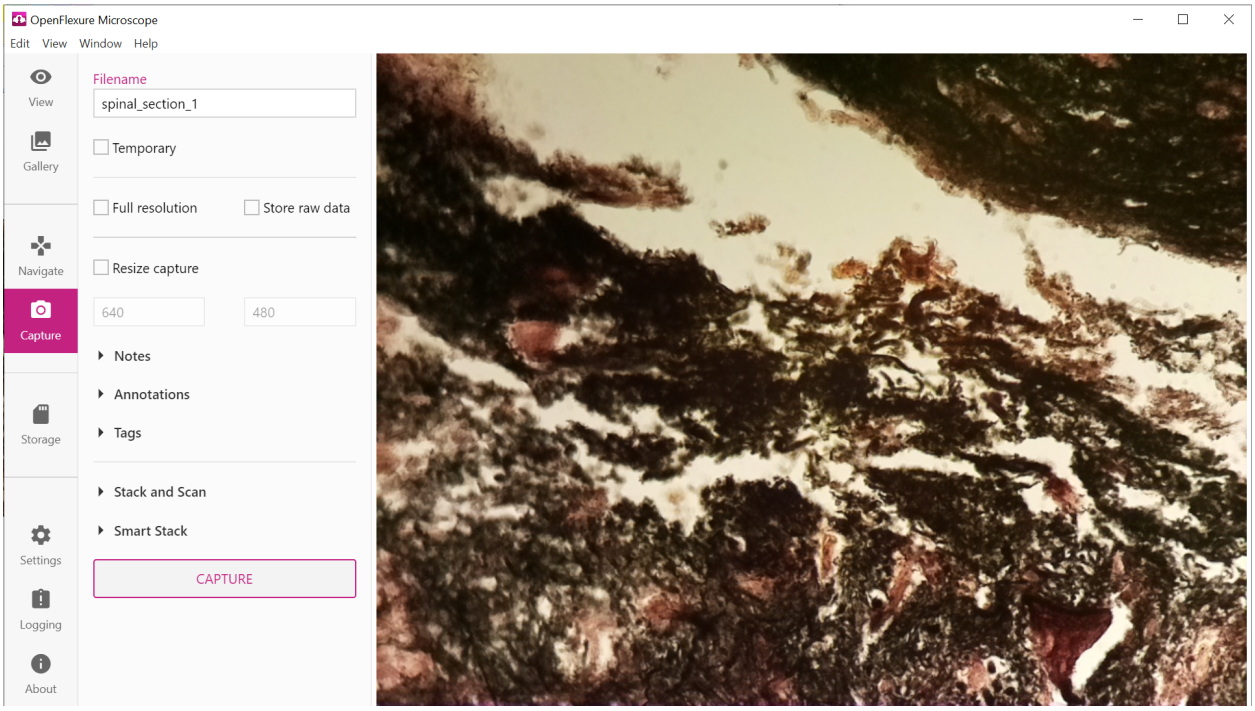
Autofocus can also be run from the navigate pane, by clicking the "fast", "medium" or "fine" buttons. Fast autofocus moves the stage up and down in a continuous motion, using the size of images in the MJPEG stream from the camera to determine the sharpest point. This is usually both faster and more accurate than the other methods, however the other two options use a different metric, and stop the stage for each measurement. This can lead to them being more reliable in some circumstances.



2.2 Capture pane

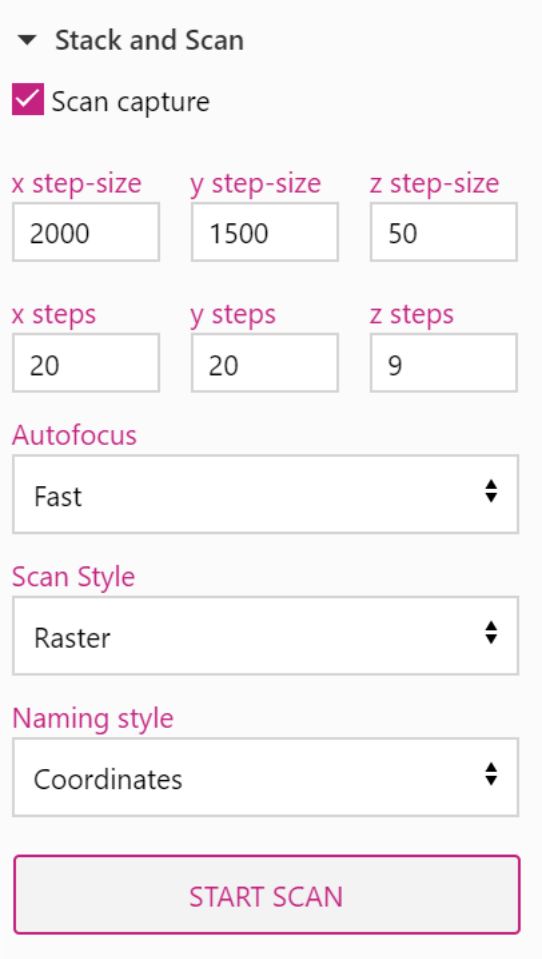
The capture pane allows images to be acquired through the interface. By default, a single image is captured to the microscope's internal SD card, and can be downloaded from the gallery. Various settings are available to control the resolution of the image captured: the "full resolution" checkbox will save the image at native resolution, and the "store raw data" checkbox saves raw pixel data as an EXIF annotation.

Due to the underlying `picamera` library, images are always saved as JPEG files, and raw data is simply appended to the file for later extraction. If an image is saved with raw data, the JPEG image is still the processed, compressed version; an external tool must be used to extract and process the raw Bayer data.



It is also possible to acquire a grid of images for stitching together into a mosaic, by expanding the "Stack and Scan" section and enabling the "Scan capture" checkbox. Scans are 3D by default (i.e. a mosaic of images in X and Y, with a Z stack at each position) but 2D or linear scans can be performed by setting the number of steps in the unused axes to 1. This allows XY mosaics or Z stacks to be performed.

When scanning is enabled, the "capture" button is replaced by a "start scan" button, which will start the scan and display a progress indicator until scanning has finished.



The step sizes (in motor steps) for each axis specify the number of motor steps to move between images, then the "steps" fields specify the number of images to acquire along each axis. If "steps" is set to 1 for any axis, no scanning happens along that axis.

The scan routine will move through XY coordinates, and at each XY position will optionally run an autofocus routine, then acquire either a single image or a Z stack depending on the value of "z steps". The autofocus options correspond to those available in the "move" pane, allowing "fast" autofocus, or conventional autofocus with coarse, medium, or fine steps to be used. Selecting "none" disables autofocus. The vast majority of the time, "fast" autofocus is both quicker and more accurate than the other methods.

Various scan patterns are available for XY scanning. Raster scanning is the default, which scans columns (i.e. from lowest to highest Y coordinate) and works from low to high X coordinates as the "slow" scan axis. Snake scanning reverses every other column, such that there is a smaller distance from the end of one column to the beginning of the next. This can be helpful if the sample is not perfectly flat, as it avoids losing focus between columns. Finally, "spiral" scanning starts by taking an image at the current position, then works outwards in concentric squares. Spiral scans use the "x steps" value to set the number of rings, and ignore the "y steps" value.

Images acquired during the scan will be saved to a folder on the Raspberry Pi. They can be named according to their coordinates in the scan (default) or numbered sequentially (in case the latter is easier to process).

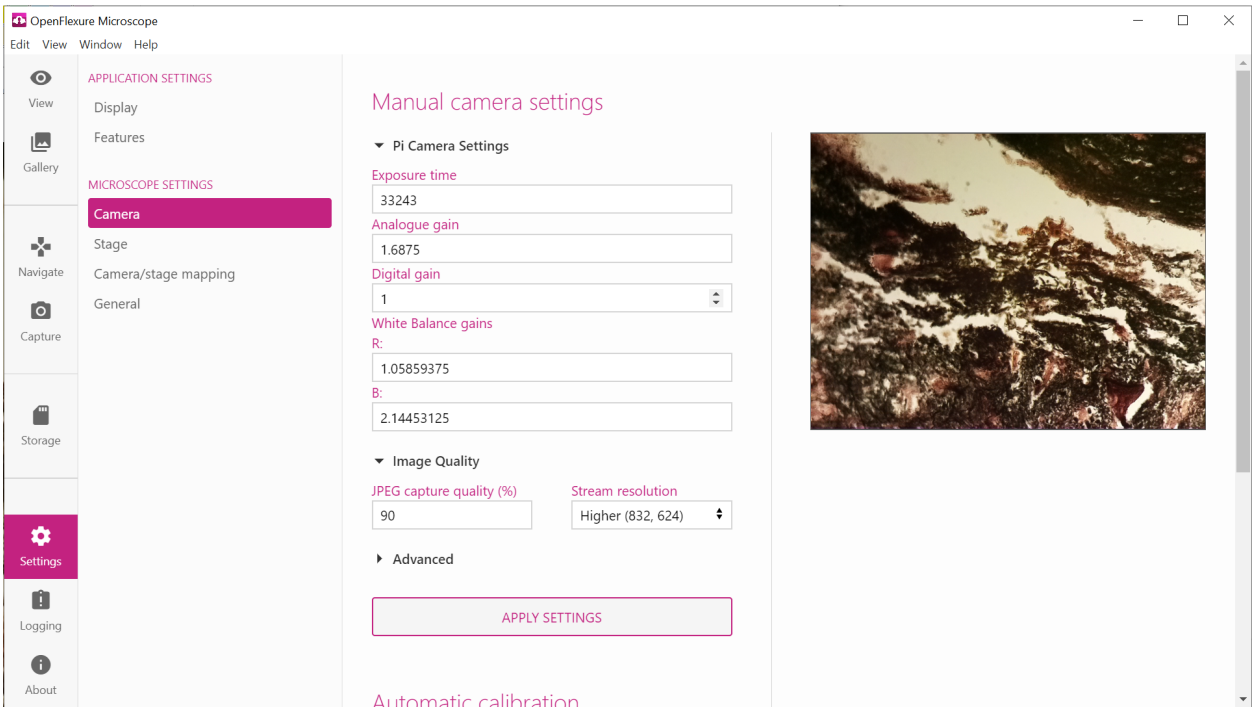
To retrieve images acquired during a scan, or captured individually, you can use the *Gallery pane*.

2.3 Settings pane

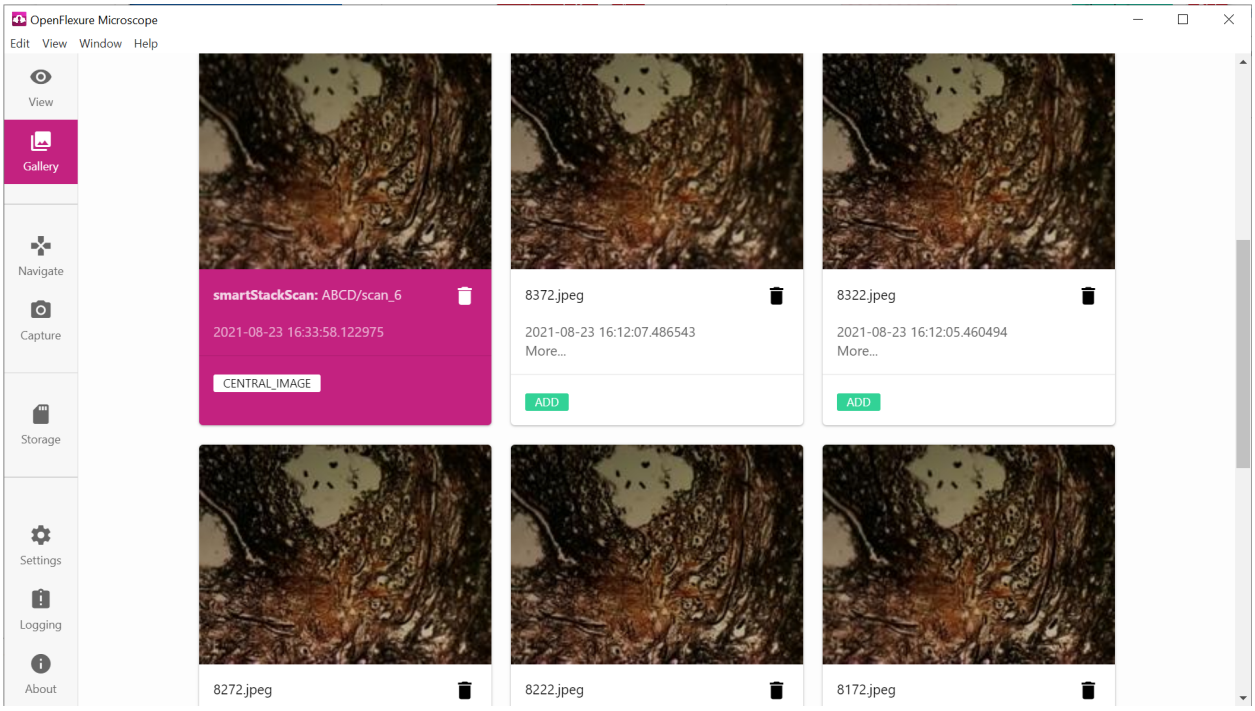
The settings for the microscope are gathered together into a settings pane, which is further subdivided into sections. This page does not provide an exhaustive list, but a few of the notable controls are:

- Adjusting exposure time and gain of the camera, including automatic adjustment.
- Automatic white balance and flat-field correction for the camera.
- Enabling or disabling certain features of the software (e.g. ImJoy integration).
- Enabling or disabling the video stream (this allows the native low-latency preview on the Raspberry Pi to be used instead).
- Calibrating the relationship between stage coordinates and pixel coordinates in the video stream, allowing click-to-move to function.

Important calibration tasks (in particular camera settings adjustment and click-to-move calibration) will be prompted in a "wizard" dialogue when the software is first run, to help first time users set up their microscope. All of the auto calibration routines are also available from the settings pane.



2.4 Gallery pane



The gallery displays all the images currently stored on the microscope. Scans are grouped together into folders. Clicking on an image will display it in a "lightbox" view that allows scrolling through all images in the current view. When an image is displayed in the lightbox view, it may be right-clicked to download it. Multiple images can also be downloaded as a zip archive.

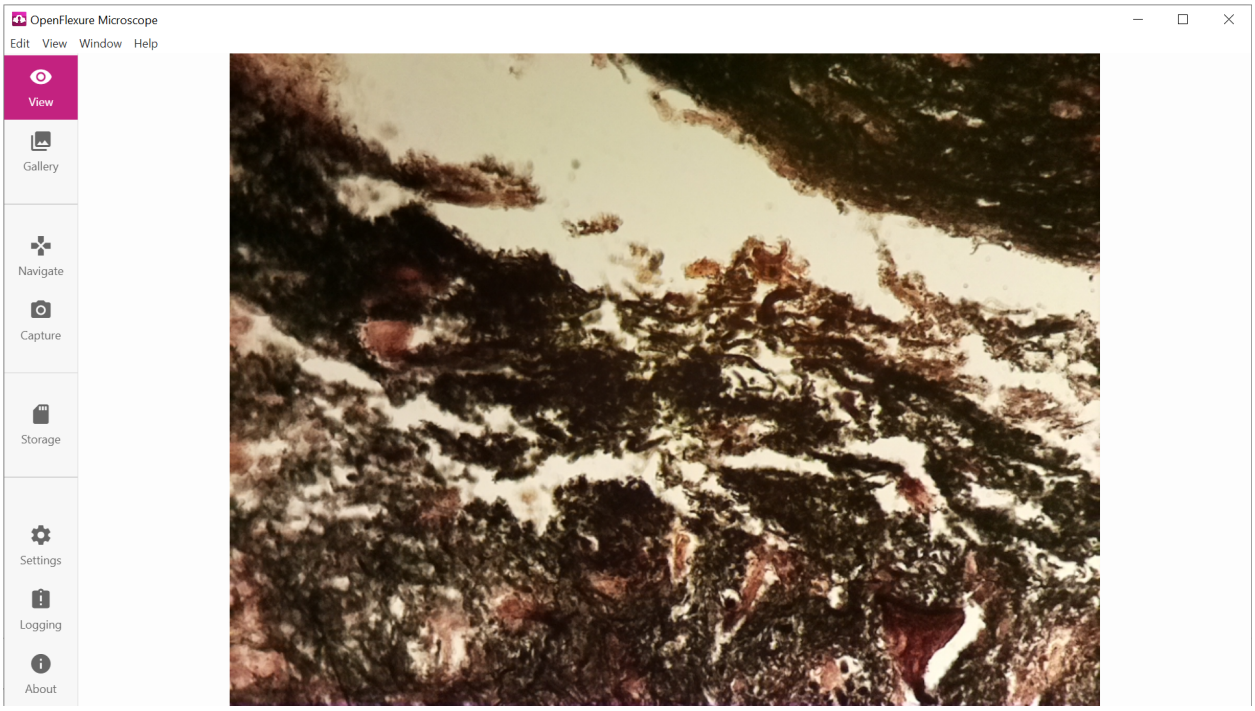
Bulk transfer of images is often easier using SCP, and images are stored by default in `/var/openflexure/data/micrographs/` on the Raspberry Pi.

Saving of images to external storage is possible - this can be configured using the "autostorage" plugin, which currently displays an SD card icon in the navigation bar at the left of the screen.

The main graphical interface for the OpenFlexure Microscope is implemented as a web application, which allows it to be accessed either through OpenFlexure Connect or a web browser. See the *Quickstart* page for connection instructions. A "tour" should guide users through the interface when they connect for the first time, and introduce the key interface elements.

2.5 Interface structure

The main interface has a tab bar on the left, which allows the operator to select different controls. By default, the "view" pane does not show any additional controls, and a video feed from the camera fills the window.



Selecting one of the tab icons on the left will bring up the corresponding interface. Most of the pages display the image on the right hand side, and add additional controls between the image display and the tab bar.

CHAPTER 3

Microscope settings

3.1 Microscope settings file

Microscope settings are made persistent via a microscope settings file. By default, this file exists at `~/.openflexure/microscope_settings.json`.

The class *openflexure_microscope.config.OpenflexureSettingsFile* provides functionality for loading a JSON-format settings file as a Python dictionary, and merging changed settings back into the file.

The default settings are loaded by the *openflexure_microscope.config.user_settings*, which can be imported anywhere in the microscope server application to allow reading and writing of persistent settings.

class openflexure_microscope.config.**OpenflexureSettingsFile**(*path: str*, *defaults: dict = None*)

An object to handle expansion, conversion, and saving of the microscope configuration.

Parameters

- **config_path** (*str*) – Path to the config JSON file (None falls back to default location)
- **expand** (*bool*) – Expand paths to valid auxillary config files.

load() → dict

Loads settings from a file on-disk.

save(*config: dict*, *backup: bool = True*)

Save settings to a file on-disk.

Parameters

- **config** (*dict*) – Dictionary of new settings
- **backup** (*bool*) – Back up previous settings file

merge(*config: dict*) → dict

Merge settings dictionary with settings loaded from file on-disk.

Parameters **config** (*dict*) – Dictionary of new settings

openflexure_microscope.config.**load_json_file**(*config_path*) → dict
Open a .json config file

Parameters **config_path** (*str*) – Path to the config JSON file. If *None*, defaults to *DEFAULT_CONFIG_PATH*

openflexure_microscope.config.**save_json_file**(*config_path: str*, *config_dict: dict*)
Save a .json config file

Parameters

- **config_dict** (*dict*) – Dictionary of config data to save.
- **config_path** (*str*) – Path to the config JSON file.

openflexure_microscope.config.**create_file**(*config_path*)
Creates an empty file, and all folder structure currently nonexistant.

Parameters **config_path** – Path to the (possibly) new file

openflexure_microscope.config.**initialise_file**(*config_path*, *populate: str = '{}\n'*)
Check if a file exists, and if not, create it and optionally populate it with content

Parameters

- **config_path** (*str*) – Path to the file.
- **populate** (*str*) – String to dump to the file, if it is being newly created

openflexure_microscope.config.**user_settings = <openflexure_microscope.config.OpenflexureSet**
Default user settings object

openflexure_microscope.config.**user_configuration = <openflexure_microscope.config.Openflexu**
Default user settings object

CHAPTER 4

Microscope class

The main microscope class handles microscope settings, passing these between the settings file and their appropriate components (camera, stage), basic metadata about the current device status, and interfacing with the separate camera and stage components. Defines a microscope object, binding a camera and stage with basic functionality.

class openflexure_microscope.microscope.**Microscope**(*settings=<openflexure_microscope.config.OpenflexureSetti object>*, *configuration=<openflexure_microscope.config.OpenflexureSettings object>*)

A basic microscope object.

The camera and stage objects may already be initialised, and can be passed as arguments.

lock = None
Composite lock for locking both camera and stage

Type labthings.CompositeLock

camera = None
Currently connected camera object

stage = None
Currently connected stage object

close()
Shut down the microscope hardware.

setup(*configuration: dict*)
Attach microscope components based on initially passed configuration file

set_stage(*configuration: Optional[dict] = None*, *stage_type: Optional[str] = None*)
Set or change the stage geometry

has_real_stage() → bool
Check if a real (non-mock) stage is currently attached.

has_real_camera() → bool
Check if a real (non-mock) camera is currently attached.

state
Dictionary of the basic microscope state.

Returns Dictionary containing complete microscope state

Return type dict

update_settings(*settings: dict*)
Applies a settings dictionary to the microscope. Missing parameters will be left untouched.

read_settings(*full: bool = True*) → dict
Get an updated settings dictionary.

Reads current attributes and properties from connected hardware, then merges those with the currently saved settings.

This is to ensure that settings for currently disconnected hardware don't get removed from the settings file.

save_settings()
Merges the current settings back to disk

force_get_metadata() → dict
Read cachable bits of microscope metadata. Currently ID, settings, and configuration can be cached

get_metadata(*cache_key: Optional[str] = None*)
Read microscope metadata, with partial caching

CHAPTER 5

Camera Class

5.1 Raspberry Pi Streaming Camera

Raspberry Pi camera implementation of the PiCameraStreamer class.

NOTES:

Still port used for image capture. Preview port reserved for onboard GPU preview.

Video port:

- Splitter port 0: Image capture (if *use_video_port* == *True*)
- Splitter port 1: Streaming frames
- Splitter port 2: Video capture
- Splitter port 3: [Currently unused]

PiCameraStreamer streams at video_resolution

Camera capture resolution set to stream_resolution in frames()

Video port uses that resolution for everything. If a different resolution is specified for video capture, this is handled by the resizer.

Still capture (if use_video_port == False) uses pause_stream to temporarily increase the capture resolution.

class openflexure_microscope.camera.pi.**PiCameraStreamer**

Raspberry Pi camera implementation of PiCameraStreamer.

picamera = None

Attached Picamera object

Type picamerax.PiCamera

image_resolution = None

Resolution for image captures

Type tuple

stream_resolution = None
Resolution for stream and video captures

Type tuple

numpy_resolution = None
Resolution for numpy array captures

Type tuple

jpeg_quality = None
JPEG quality

Type int

mjpeg_quality = None
MJPEG quality

Type int

mjpeg_bitrate = None
MJPEG quality

Type int

configuration
The current camera configuration.

state
The current read-only camera state.

close()
Close the Raspberry Pi PiCameraStreamer.

read_settings() → dict
Return config dictionary of the PiCameraStreamer.

update_settings(*config: dict*)
Write a config dictionary to the PiCameraStreamer config.

The passed dictionary may contain other parameters not relevant to camera config. Eg. Passing a general config file will work fine.

Parameters **config** (*dict*) – Dictionary of config parameters.

apply_picamera_settings(*settings_dict: dict*, *pause_for_effect: bool = True*)

Parameters

- **settings_dict** (*dict*) – Dictionary of properties to apply to the picamerax.PiCamera: object
- **pause_for_effect** (*bool*) – Pause tactically to reduce risk of timing issues

set_zoom(*zoom_value: Union[float, int] = 1.0*) → None
Change the camera zoom, handling re-centering and scaling.

start_preview(*fullscreen: bool = True*, *window: Tuple[int, int, int, int] = None*)
Start the on board GPU camera preview.

stop_preview()
Stop the on board GPU camera preview.

start_recording(*output: Union[str, BinaryIO], fmt: str = 'h264', quality: int = 15*)
Start recording.

Start a new video recording, writing to a output object.

Parameters

- **output** – String or file-like object to write capture data to
- **fmt** (*str*) – Format of the capture.
- **quality** (*int*) – Video recording quality.

Returns Target object.

Return type output_object (str/BytesIO)

stop_recording()
Stop the last started video recording on splitter port 2.

start_stream() → None
Sets the camera resolution to the video/stream resolution, and starts recording if the stream should be active.

stop_stream() → None
Sets the camera resolution to the still-image resolution, and stops recording if the stream is active.

Parameters **splitter_port** (*int*) – Splitter port to stop recording on

capture(*output: Union[str, BinaryIO], fmt: str = 'jpeg', use_video_port: bool = False, resize: Tuple[int, int] = None, bayer: bool = True, thumbnail: Tuple[int, int, int] = None*)
Capture a still image to a StreamObject.

Defaults to JPEG format. Target object can be overridden for development purposes.

Parameters

- **output** – String or file-like object to write capture data to
- **fmt** – Format of the capture.
- **use_video_port** – Capture from the video port used for streaming. Lower resolution, faster.
- **resize** – Resize the captured image.
- **bayer** – Store raw bayer data in capture
- **thumbnail** – Dimensions and quality (x, y, quality) of a thumbnail to generate, if supported

Returns Target object.

Return type output_object (str/BytesIO)

array(*use_video_port: bool = True*) → numpy.ndarray
Capture an uncompressed still RGB image to a Numpy array.

Parameters

- **use_video_port** (*bool*) – Capture from the video port used for streaming. Lower resolution, faster.
- **resize** (*(int, int)*) – Resize the captured image.

Returns Output array of capture

Return type output_array (np.ndarray)

5.2 Base Streaming Camera

class openflexure_microscope.camera.base.**TrackerFrame**(*size*, *time*)

size
Alias for field number 0

time
Alias for field number 1

class openflexure_microscope.camera.base.**FrameStream**(**args*, ***kwargs*)
A file-like object used to analyse and stream MJPEG frames.

Instead of analysing a load of real MJPEG frames after they've been stored in a BytesIO stream, we tell the camera to write frames to this class instead.

We then do analysis as the frames are written, and discard old frames as each new frame is written.

start_tracking()
Start tracking frame sizes

stop_tracking()
Stop tracking frame sizes

reset_tracking()
Empty the array of tracked frame sizes

write(*s*)
Write a new frame to the FrameStream. Does a few things: 1. If tracking frame size, store the size in self.frames 2. Rewind and truncate the stream (delete previous frame) 3. Store the new frame image 4. Set the new_frame event

getvalue() → bytes
Clear tne new_frame event and return frame data

getframe() → bytes
Wait for a new frame to be available, then return it

class openflexure_microscope.camera.base.**BaseCamera**
Base implementation of StreamingCamera.

lock = None
Access lock for the camera

Type labthings.StrictLock

stream = None
Streaming and analysis frame buffer

Type *FrameStream*

configuration
The current camera configuration.

state
The current read-only camera state.

start_stream()
Ensure the frame stream is actively running

stop_stream()
Stop the active stream, if possible

update_settings(*config: dict*)
Update settings from a config dictionary

read_settings() → dict
Return the current settings as a dictionary

capture(*output: Union[str, BinaryIO], fmt: str = 'jpeg', use_video_port: bool = False, resize: Optional[Tuple[int, int]] = None, bayer: bool = True, thumbnail: Optional[Tuple[int, int, int]] = None*)
Perform a basic capture to output

Parameters

- **output** – String or file-like object to write capture data to
- **fmt** – Format of the capture.
- **use_video_port** – Capture from the video port used for streaming. Lower resolution, faster.
- **resize** – Resize the captured image.
- **bayer** – Store raw bayer data in capture
- **thumbnail** – Dimensions and quality (x, y, quality) of a thumbnail to generate, if supported

start_worker(**_) → bool
Start the background camera thread if it isn't running yet.

get_frame() → bytes
Return the current camera frame.

Just an alias of self.stream.getframe()

close()
Close the BaseCamera and all attached StreamObjects.

5.3 Capture Object

By default, all image and video capture data are stored to instances of `openflexure_microscope.captures.CaptureObject`. This class mostly wraps up complexity associated with moving data between disk and memory.

The class also includes some convenience features such as handling metadata tags and file names, and generating image thumbnails. Additionally, the class handles storing capture metadata to Exif tags in supported formats.

Below are details of available methods and attributes.

class openflexure_microscope.captures.capture.**CaptureObject**(*filepath: str*)
File-like object used to store and process on-disk capture data, and metadata. Serves to simplify modifying properties of on-disk capture data.

id = None
Unique capture ID

Type str

split_file_path(*filepath: str*)
Take a full file path, and split it into separated class properties.

Parameters **filepath** (*str*) – String of the full file path, including file format extension

exists
Check if capture data file exists on disk.

put_tags(*tags: List[str]*)
Add a new tag to the `tags` list attribute.

Parameters **tags** (*list*) – List of tags to be added

delete_tag(*tag: str*)
Remove a tag from the `tags` list attribute, if it exists.

Parameters **tag** (*str*) – Tag to be removed

put_annotations(*data: Dict[str, str]*)
Merge annotations from a passed dictionary into the capture metadata, and saves.

Parameters **data** (*dict*) – Dictionary of metadata to be added

put_metadata(*data: dict*)
Merge root metadata from a passed dictionary into the capture metadata, and saves.

Parameters **data** (*dict*) – Dictionary of metadata to be added

put_and_save(*tags: Optional[List[str]] = None*, *annotations: Optional[Dict[str, str]] = None*, *dataset: Optional[Dict[str, str]] = None*, *metadata: Optional[dict] = None*)
Batch-write tags, metadata, and annotations in a single disk operation

metadata
Create basic metadata dictionary from basic capture data, and any added custom metadata and tags.

data
Return a BytesIO object of the capture data.

binary
Return a byte string of the capture data.

thumbnail
Returns a thumbnail of the capture data, for supported image formats.

save()
Write stream to file, and save/update metadata file

delete() → bool
If the StreamObject has been saved, delete the file.

5.4 Capturing from a camera object

In the cases of both a Raspberry Pi Streaming Camera, and a Mock Camera (attached if no real camera can be found), the camera's `capture` method takes as it's first positional argument either a string describing a file path to save to, or any Python file-like object.

The `openflexure_microscope.camera.capture.CaptureObject` class works by providing a file path string, but adds additional functionality around storing and retreiving EXIF metadata in compatible files.

If, for your application, you do not require this functionality, you can pass a simple string or file-like object. For example, to take an image that will be stored in-memory, processed rapidly, and then discarded, you could use a BytesIO stream:

```
import io
from PIL import Image
...
```

(continues on next page)

(continued from previous page)

```
with microscope.camera.lock, io.BytesIO() as stream:

    microscope.camera.capture(
        stream,
        use_video_port=True,
        bayer=False,
    )

    stream.seek(0)
    image = Image.open(stream)
```

CHAPTER 6

Stage Class

6.1 Sangaboard Microscope Stage

class openflexure_microscope.stage.sanga.**SangaStage**(*port=None*, ***kwargs*)

Sangaboard v0.2 and v0.3 powered Stage object

Parameters **port** (*str*) – Serial port on which to open communication

board

Parent Sangaboard object.

Type openflexure_microscope.stage.sangaboard.Sangaboard

_backlash

3-element (element-per-axis) list of backlash compensation in steps.

Type list

state

The general state dictionary of the board.

configuration

The general stage configuration.

n_axes

The number of axes this stage has.

position

The current position, as a list

backlash

The distance used for backlash compensation. Software backlash compensation is enabled by setting this property to a value other than *None*. The value can either be an array-like object (list, tuple, or numpy array) with one element for each axis, or a single integer if all axes are the same. The property will always return an array with the same length as the number of axes. The backlash compensation algorithm is fairly basic - it ensures that we always approach a point from the same direction. For each axis that's moving, the direction of motion is compared with `backlash`. If the direction is opposite, then the stage will

overshoot by the amount in `-backlash[i]` and then move back by `backlash[i]`. This is computed per-axis, so if some axes are moving in the same direction as `backlash`, they won't do two moves.

update_settings(*config: dict*)
Update settings from a config dictionary

read_settings() → dict
Return the current settings as a dictionary

move_rel(*displacement: Union[int, Tuple[int, int, int], numpy.ndarray], axis: Optional[typing_extensions.Literal['x', 'y', 'z']][x, y, z]] = None, backlash: bool = True*)
Make a relative move, optionally correcting for backlash. displacement: integer or array/list of 3 integers axis: None (for 3-axis moves) or one of 'x','y','z' backlash: (default: True) whether to correct for backlash.

Backlash Correction: This backlash correction strategy ensures we're always approaching the end point from the same direction, while minimising the amount of extra motion. It's a good option if you're scanning in a line, for example, as it will kick in when moving to the start of the line, but not for each point on the line. For each axis where we're moving in the *opposite* direction to self.backlash, we deliberately overshoot:

move_abs(*final: Union[Tuple[int, int, int], numpy.ndarray], **kwargs*)
Make an absolute move to a position

zero_position()
Set the current position to zero

close()
Cleanly close communication with the stage

release_motors()
De-energise the stepper motor coils

class openflexure_microscope.stage.sanga.**SangaDeltaStage**(*port: Optional[str] = None, flex_h: int = 80, flex_a: int = 50, flex_b: int = 50, camera_angle: float = 0, **kwargs*)

position
The current position, as a list

move_rel(*displacement: Union[int, Tuple[int, int, int], numpy.ndarray], axis: Optional[typing_extensions.Literal['x', 'y', 'z']][x, y, z]] = None, backlash: bool = True*)
Make a relative move, optionally correcting for backlash. displacement: integer or array/list of 3 integers axis: None (for 3-axis moves) or one of 'x','y','z' backlash: (default: True) whether to correct for backlash.

Backlash Correction: This backlash correction strategy ensures we're always approaching the end point from the same direction, while minimising the amount of extra motion. It's a good option if you're scanning in a line, for example, as it will kick in when moving to the start of the line, but not for each point on the line. For each axis where we're moving in the *opposite* direction to self.backlash, we deliberately overshoot:

move_abs(*final: Union[Tuple[int, int, int], numpy.ndarray], **kwargs*)
Make an absolute move to a position

6.2 Base Microscope Stage

class openflexure_microscope.stage.base.**BaseStage**

lock
Strict lock controlling thread access to camera hardware

Type labthings.StrictLock

update_settings(*config: dict*)
Update settings from a config dictionary

read_settings()
Return the current settings as a dictionary

state
The general state dictionary of the board.

configuration
The general stage configuration.

n_axes
The number of axes this stage has.

position
The current position, as a list

backlash
Get the distance used for backlash compensation.

move_rel(*displacement: Union[int, Tuple[int, int, int]], axis: Optional[typing_extensions.Literal['x', 'y', 'z']][x, y, z]] = None, backlash: bool = True*)
Make a relative move, optionally correcting for backlash. displacement: integer or array/list of 3 integers backlash: (default: True) whether to correct for backlash.

move_abs(*final: Tuple[int, int, int], **kwargs*)
Make an absolute move to a position

zero_position()
Set the current position to zero

close()
Cleanly close communication with the stage

scan_linear(*rel_positions: List[Tuple[int, int, int]], backlash: bool = True, return_to_start: bool = True*)
Scan through a list of (relative) positions (generator fn) rel_positions should be an nx3-element array (or list of 3 element arrays). Positions should be relative to the starting position - not a list of relative moves. backlash argument is passed to move_rel if return_to_start is True (default) we return to the starting position after a successful scan. NB we always attempt to return to the starting position if an exception occurs during the scan..

scan_z(*dz: List[int], **kwargs*)
Scan through a list of (relative) z positions (generator fn) This function takes a 1D numpy array of Z positions, relative to the position at the start of the scan, and converts it into an array of 3D positions with x=y=0. This, along with all the keyword arguments, is then passed to `scan_linear`.

CHAPTER 7

Developing API Extensions

7.1 Introduction

Extensions allow functionality to be added to the OpenFlexure Microscope web API without having to modify the base code. They have full access to the `openflexure_microscope.Microscope` object, including direct access to any attached *`openflexure_microscope.camera.base.BaseCamera`* and `openflexure_stage.stage.OpenFlexureStage` objects. This also allows access to the `picamerax.PiCamera` object.

Extensions can either be loaded from a single Python file, or as a Python package installed to the environment being used.

7.1.1 Single-file extensions

For adding simple functionality, such as a few basic functions and API routes, a single Python file can be loaded as a extension. This Python file must contain all of your extension objects, and be located in the applications extensions directory (by default `/var/openflexure/extensions/microscope_extensions`).

7.1.2 Package extensions

Generally, for adding anything other than very simple functionality, extensions should be written as package distributions. This has the advantage of allowing relative imports, so functionality can be easily split over several files. For example, class definitions associated with API routes can be separated from class definitions associated with the microscope extension.

Your module must be a folder within the extensions folder (by default `/var/openflexure/extensions/microscope_extensions`), and include a top-level `__init__.py` file which includes (or imports) all of your extension classes, and includes them in a global constant *LABTHINGS_EXTENSIONS* list.

For example, if your extension classes are defined in a file `my_extension.py`, your adjascent `__init__.py` file may look like:

```
from .my_extension import MyExtensionClass, MyOtherExtensionClass

LABTHINGS_EXTENSIONS = (MyExtensionClass, MyOtherExtensionClass)
```

In order to enable a globally installed, packaged extension, create a file in the applications extensions directory (by default `/var/openflexure/extensions/microscope_extensions`) which imports your extension object(s) from your module.

7.2 Basic extension structure

An extension starts as a subclass of `labthings.extensions.BaseExtension`. Each extension is described by a single `BaseExtension` instance, containing any number of methods, API views, and additional hardware components.

You will build your extension by subclassing `labthings.extensions.BaseExtension`, and adding the class to a top-level *LABTHINGS_EXTENSIONS* list.

In order to access the currently running microscope object, use the `labthings.find_component()` function, with the argument `"org.openflexure.microscope"`. Likewise, any new components attached by other extensions can be found using their full name, as above.

A simple extension file, with no API views but application-available methods may look like:

```
from labthings import find_component
from labthings.extensions import BaseExtension


# Create the extension class
class MyExtension(BaseExtension):
    def __init__(self):
        # Superclass init function
        super().__init__("com.myname.myextension", version="0.0.0")

    def identify(self):
        """
        Demonstrate access to Microscope.camera, and Microscope.stage
        """
        microscope = find_component("org.openflexure.microscope")

        response = (
            f"My name is {microscope.name}. "
            f"My parent camera is {microscope.camera}, "
            f"and my parent stage is {microscope.stage}."
        )

        return response

    def rename(self, new_name):
        """
        Rename the microscope
        """

        microscope = find_component("org.openflexure.microscope")

        microscope.name = new_name
        microscope.save_settings()
```

(continues on next page)

(continued from previous page)

```
LABTHINGS_EXTENSIONS = (MyExtension,)
```

Once this extension is loaded, any other extensions will have access to your methods:

```
from labthings import find_extension

def test_extension_method():
    # Find your extension. Returns None if it hasn't been found.
    my_found_extension = find_extension("com.myname.myextension")

    # Call a function from your extension
    if my_found_extension:
        my_found_extension.identify()
```

7.3 Adding web API views

7.3.1 Key terminology

API View (or View)

"A view function is the code you write to respond to requests to your application [. . .] For RESTful APIs it's especially helpful to execute a different function for each HTTP method. With the [View class] you can easily do that. Each HTTP method maps to a function with the same name (just in lowercase)" - Flask documentation

7.3.2 Introduction

Extensions can create views to expose extension functionality via the web API. Creating API views for your extension is strongly recommended, as this is the primary way we encourage interaction with the microscope device.

As with most HTTP APIs, we make use of basic HTTP request methods. GET requests return data without modifying any state. POST requests completely replace data with data passed as request arguments. PUT requests update data with new data passed as request arguments. DELETE requests delete a particular object from the server. Your API views need not implement all of these methods.

Continuing our example on the previous page, and discussed below, adding API views may look like:

```
from labthings import fields, find_component
from labthings.extensions import BaseExtension
from labthings.views import View


# Create the extension class
class MyExtension(BaseExtension):
    def __init__(self):
        # Superclass init function
        super().__init__("com.myname.myextension", version="0.0.0")

        # Add our API Views (defined below MyExtension)
        self.add_view(ExampleIdentifyView, "/identify")
        self.add_view(ExampleRenameView, "/rename")
```

(continues on next page)

(continued from previous page)

```
    def identify(self, microscope):
        """
        Demonstrate access to Microscope.camera, and Microscope.stage
        """
        response = (
            f"My name is {microscope.name}. "
            f"My parent camera is {microscope.camera}, "
            f"and my parent stage is {microscope.stage}."
        )

        return response

    def rename(self, microscope, new_name):
        """
        Rename the microscope
        """
        microscope.name = new_name
        microscope.save_settings()


## Extension views
class ExampleIdentifyView(View):
    def get(self):
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Return our identify function's output
        return self.extension.identify(microscope)


class ExampleRenameView(View):
    # Expect a request parameter called "name", which is a string.
    # Passed to the argument "args".
    args = fields.String(required=True, example="My Example Microscope")

    def post(self, args):
        # Look for our new name in the request body
        new_name = args

        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Pass microscope and new name to our rename function
        self.extension.rename(microscope, new_name)

        # Return our identify function's output
        return self.extension.identify(microscope)


LABTHINGS_EXTENSIONS = (MyExtension,)
```

Note that we are now passing our microscope object as an argument to our API methods. Finding the microscope component is performed by the API view at request-time, and passed onto the functions.

Your extension functions can be accessed from within an API View by using `self.extension`. Once your view has been added to your extension, this will point to the extension object, allowing your API views to use your extension

functionality.

In this case, our extension will have two new API views at */identify* and */rename*. The */identify* view only accepts GET requests, and the */rename* view only accepts POST requests.

Request arguments

For POST and PUT requests, data usually needs to be provided to the view in order to perform its function. In this example, our `rename` view requires a new microscope name to be passed. We make use of the `args` class attribute to provide this functionality.

`args` defines the type of data expected in the request body. In this example, we use `String` type data. The arguments of `fields.String` allow us to provide additional information, such as the parameter being required, and example values to appear in API documentation.

Adding additional fields, and the meaning of the field types, will be discussed further in the next section.

When a POST request is made to our API view, the server converts the body of the request into a `String`, and passes it as a positional argument to our `post` function.

Swagger documentation

At this point, it is useful to introduce the automatically generated Swagger documentation. From any web browser, go to `http://microscope.local/api/v2/docs/swagger-ui` (or replace `microscope.local` with your microscope's IP address if `microscope.local` doesn't work for your system).

This page uses SwaggerUI to provide visual, interactive API documentation. Find your extensions URL in the documentation under the `extensions` group. Basic documentation about the parameters required for your POST method should be visible, as well as an interactive example filled out with the example request given in the view `schema`.

7.4 Marshaling data

7.4.1 Introduction

The OpenFlexure Microscope Server makes use of the Marshmallow library for both response and argument marshaling. From the Marshmallow documentation:

> **marshmallow** is an ORM/ODM/framework-agnostic library for converting complex datatypes, such as objects, to and from native Python datatypes.
>
> In short, marshmallow schemas can be used to:
>
> - **Validate** input data.
> - **Deserialize** input data to app-level objects.
> - **Serialize** app-level objects to primitive Python types. The serialized objects can then be rendered to standard formats such as JSON for use in an HTTP API.

When developing extensions, you are encouraged to make use of your View `schema` and `args` class attributes to handle serialisation of your API responses, and parsing of request parameters respectively.

Schemas and fields

A **field** describes the data type of a single parameter, as well as any other properties of that parameter for use in parsing, and documentation. For example, a String-type field, with a default value in case no actual value is passed, and extra documentation, may look like:

```
fields.String(required=False, missing="Default value", example="Example value")
```

A **schema** is a collection of keys and fields describing how an object should be serialized/deserialized. Schemas can be created in several ways, either by creating a `Schema` class, or by passing a dictionary of key-field pairs. Both methods will be discussed in the following examples.

Argument parsing

In the previous section we saw how to use fields and `args` to get simple arguments from requests, in which a single parameter is required. By making use of Marshmallow schemas, and the Webargs library, we can allow for more complex requests containing many parameters of different types. The parsed request parameters are then passed to the view function as a positional argument (as before), in the form of a dictionary.

For example, if you are creating an API route, in which you expect parameters `name`, `age`, and optionally, `job`, your schema class may look like:

```
from labthings.schema import Schema
from labthings import fields

class UserSchema(Schema):
    name = fields.String(required=True)
    age = fields.Integer(required=True)
    job = fields.String(required=False, missing="Unknown")
```

To inform your POST method to expect these arguments, use the `args` class attribute:

```
class MyView(View):
    args = UserSchema()

    def post(self, args):
    ..
```

Alternatively, if your schema is only used in a single location, it may be simpler to create a dictionary schema only where it is used, for example:

```
class MyView(View):
    args = {
        "name": fields.String(required=True),
        "age": fields.Integer(required=True),
        "job": fields.String(required=False, missing="Unknown")
    }

    def post(self, args):
        ...
```

A compatible request body, in JSON format, may look like:

```
{
    "name": "John Doe",
    "age": 45,
```

(continues on next page)

(continued from previous page)

```
    "job": "Python developer"
}
```

This JSON data is the parsed, converted into a Python dictionary, and passed as an argument. Retreiving the data from within your view function may therefore look like:

```
class MyView(View):
    args = {
        "name": fields.String(required=True),
        "age": fields.Integer(required=True),
        "job": fields.String(required=False, missing="Unknown")
    }

    def post(self, args):
        name = args.get("name")  # Returns "John Doe", type str
        age = args.get("age")  # Returns 45, type int
        job = args.get("job")  # Returns "Python developer", type str
```

Object serialization

Schemas can also be used to format our data so that it is suitable for an API response. Our API expects JSON formatted data both in, and out. It is therefore important that your API views respond with valid JSON where possible.

Continuing with our example in the previous pages, we will enhance our `identify` method to provide more, better formatted information about our current microscope.

We start by creating a schema to describe how to serialise a `openflexure_microscope.Microscope` object.

```
# Define which properties of a Microscope object we care about,
# and what types they should be converted to
class MicroscopeIdentifySchema(Schema):
    name = fields.String()  # Microscopes name
    id = fields.UUID()  # Microscopes unique ID
    state = fields.Dict()  # Status dictionary
    camera = fields.String()  # Camera object (represented as a string)
    stage = fields.String()  # Stage object (represented as a string)
```

We use this new schema in our `identify` view like so:

```
class ExampleIdentifyView(View):
    # Format our returned object using MicroscopeIdentifySchema
    schema = MicroscopeIdentifySchema()

    def get(self):
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Return our microscope object,
        # let schema handle formatting the output
        return microscope
```

Note that our `get` method now returns the `openflexure_microscope.Microscope` object itself. No formatting is done by the function, it is entirely handled by the view class, and its *schema* attribute. Additionally, since we defined our schema as a class, it can be re-used elsewhere.

For our `rename` view, we will use a simpler schema for our input arguments, defined by a dictionary (since we are

only expecting a single parameter in, and it will likely not be re-used elsewhere). Our response, however, will use our `MicroscopeIdentifySchema` class. This means that the *response* of our `identify` and `rename` views will be identically formatted.

Our `rename` view class may now look like:

```
class ExampleRenameView(View):
    # Format our returned object using MicroscopeIdentifySchema
    schema = MicroscopeIdentifySchema()
    # Expect a request parameter called "name", which is a string. Pass to argument
↪"args".
    args = {"name": fields.String(required=True, example="My Example Microscope")}

    def post(self, args):
        # Look for our "name" parameter in the request arguments
        new_name = args.get("name")

        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Pass microscope and new name to our rename function
        rename(microscope, new_name)

        # Return our microscope object,
        # let schema handle formatting the output
        return microscope
```

Complete example

Combining both of these into our example extension, we now have:

```
from labthings import Schema, fields, find_component
from labthings.extensions import BaseExtension
from labthings.views import View


# Create the extension class
class MyExtension(BaseExtension):
    def __init__(self):
        # Superclass init function
        super().__init__("com.myname.myextension", version="0.0.0")

        # Add our API Views (defined below MyExtension)
        self.add_view(ExampleIdentifyView, "/identify")
        self.add_view(ExampleRenameView, "/rename")

    def rename(self, microscope, new_name):
        """
        Rename the microscope
        """
        microscope.name = new_name
        microscope.save_settings()


# Define which properties of a Microscope object we care about,
# and what types they should be converted to
class MicroscopeIdentifySchema(Schema):
```

(continues on next page)

(continued from previous page)

```
    name = fields.String()  # Microscopes name
    id = fields.UUID()  # Microscopes unique ID
    state = fields.Dict()  # Status dictionary
    camera = fields.String()  # Camera object (represented as a string)
    stage = fields.String()  # Stage object (represented as a string)


## Extension views
class ExampleIdentifyView(View):
    # Format our returned object using MicroscopeIdentifySchema
    schema = MicroscopeIdentifySchema()

    def get(self):
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Return our microscope object,
        # let schema handle formatting the output
        return microscope


class ExampleRenameView(View):
    # Format our returned object using MicroscopeIdentifySchema
    schema = MicroscopeIdentifySchema()
    # Expect a request parameter called "name", which is a string. Pass to argument
↪"args".
    args = {"name": fields.String(required=True, example="My Example Microscope")}

    def post(self, args):
        # Look for our "name" parameter in the request arguments
        new_name = args.get("name")

        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Pass microscope and new name to our rename function
        self.extension.rename(microscope, new_name)

        # Return our microscope object,
        # let schema handle formatting the output
        return microscope


LABTHINGS_EXTENSIONS = (MyExtension,)
```

7.5 Thing Properties

7.5.1 Introduction

As well as generating Swagger documentation, the server will generate a draft W3C Thing Description . This description allows the microscope's features to be understood in a common "Web of Things" language.

Thing Properties "expose state of the Thing. This state can then be retrieved (read) and optionally updated (write)." For the microscope, this includes the current read-only state, such as if the microscope has real camera or stage hardware

attached, as well as read-write states like camera settings, and the microscope name.

The property description for a view will be generated automatically from your available view methods, any schema decorators used, and any docstrings added to the view.

7.5.2 Defining Thing Properties

In order to register a view as a Thing property, we use the `PropertyView` class, like so:

```
# Since we only have a GET method here, it'll register as a read-only property
class ExampleIdentifyView(PropertyView):
    # Format our returned object using MicroscopeIdentifySchema
    schema = MicroscopeIdentifySchema()

    def get(self):
        """
        Show identifying information about the current microscope object
        """
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Return our microscope object,
        # let schemah handle formatting the output
        return microscope
```

7.5.3 Property schema

For read-write properties, it is best practice for the expected request arguments, and the views responses, to follow the same format. In this way, by looking at the response of a GET request, one can know the type of data expected in by a PUT request.

For example, if your GET request returns the JSON:

```
{
    "name": "John Doe",
    "age": 45,
    "job": "Python developer"
}
```

and your property supports PUT requests (for updating data), then a valid PUT request could contain the data:

```
{
    "age": 46,
    "job": "Landscape gardener"
}
```

This request would update the property, such that a GET request would *now* return:

```
{
    "name": "John Doe",
    "age": 46,
    "job": "Landscape gardener"
}
```

In Property Views the `schema` class attribute acts as the schema for both marshalling responses *and* parsing arguments. This is because property requests and responses should be identically formatted.

We will implement the `schema` attribute in our `ExampleRenameView` view from our previous example:

```
# We can use a single schema as the input and output will be formatted identically
# Eg. We always expect a "name" string argument, and always return a "name" string␣
↪attribute
class ExampleRenameView(PropertyView):
    schema = {"name": fields.String(required=True, example="My Example Microscope")}

    def get(self):
        """
        Show the current microscope name
        """
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        return microscope

    def post(self, args):
        """
        Change the current microscope name
        """
        # Look for our "name" parameter in the request arguments
        new_name = args.get("name")

        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Pass microscope and new name to our rename function
        rename(microscope, new_name)

        # Return our microscope object,
        # let schema handle formatting the output
        return microscope
```

7.5.4 Complete example

Combining these into our example extension, we now have:

```
from labthings import Schema, fields, find_component
from labthings.extensions import BaseExtension
from labthings.views import PropertyView


# Create the extension class
class MyExtension(BaseExtension):
    def __init__(self):
        # Superclass init function
        super().__init__("com.myname.myextension", version="0.0.0")

        # Add our API Views (defined below MyExtension)
        self.add_view(ExampleIdentifyView, "/identify")
        self.add_view(ExampleRenameView, "/rename")

    def rename(self, microscope, new_name):
        """
        Rename the microscope
```

(continues on next page)

(continued from previous page)

```
        """
        microscope.name = new_name
        microscope.save_settings()


# Define which properties of a Microscope object we care about,
# and what types they should be converted to
class MicroscopeIdentifySchema(Schema):
    name = fields.String()  # Microscopes name
    id = fields.UUID()  # Microscopes unique ID
    state = fields.Dict()  # Status dictionary
    camera = fields.String()  # Camera object (represented as a string)
    stage = fields.String()  # Stage object (represented as a string)


## Extension viewss

# Since we only have a GET method here, it'll register as a read-only property
class ExampleIdentifyView(PropertyView):
    # Format our returned object using MicroscopeIdentifySchema
    schema = MicroscopeIdentifySchema()

    def get(self):
        """
        Show identifying information about the current microscope object
        """
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Return our microscope object,
        # let schema handle formatting the output
        return microscope


# We can use a single schema as the input and output will be formatted identically
# Eg. We always expect a "name" string argument, and always return a "name" string␣
↪attribute
class ExampleRenameView(PropertyView):
    schema = {"name": fields.String(required=True, example="My Example Microscope")}

    def get(self):
        """
        Show the current microscope name
        """
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        return microscope

    def post(self, args):
        """
        Change the current microscope name
        """
        # Look for our "name" parameter in the request arguments
        new_name = args.get("name")

        # Find our microscope component
```

(continues on next page)

(continued from previous page)

```
        microscope = find_component("org.openflexure.microscope")

        # Pass microscope and new name to our rename function
        self.extension.rename(microscope, new_name)

        # Return our microscope object,
        # let schema handle formatting the output
        return microscope


LABTHINGS_EXTENSIONS = (MyExtension,)
```

7.6 Thing Actions

7.6.1 Introduction

As well as properties, the OpenFlexure Microscope Server also supports Thing Actions. Thing Actions "invoke a function of the Thing, which manipulates state (e.g., toggling a lamp on or off) or triggers a process on the Thing (e.g., dim a lamp over time)." For the microscope, this would include moving the stage or taking a capture. Both of these require internal logic, and cannot be performed by changing a simple property.

Actions should be *triggered* with POST requests *only*. Ideally, a view corresponding to an action should only support POST requests.

Like properties, we use a special view class to identify a view as an action: `ActionView`. For example, a view to perform a "quick-capture" action may look like:

```
class QuickCaptureAPI(ActionView):
    """
    Take an image capture and return it without saving
    """
    # Expect a "use_video_port" boolean, which defaults to True if none is given
    args = {"use_video_port": fields.Boolean(missing=True)}

    # Our success response (200) returns an image (image/jpeg mimetype)
    responses = {
        200: {
            "content": { "image/jpeg": {} },
        }
    }

    def post(self, args):
        """
        Take a non-persistant image capture.
        """
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Open a BytesIO stream to be destroyed once request has returned
        with io.BytesIO() as stream:

            # Capture to our stream object
            microscope.camera.capture(stream, use_video_port=args.get("use_video_port
↪"))
```

(continues on next page)

(continued from previous page)

```
            # Rewind the stream
            stream.seek(0)

            # Return our image data using Flasks send_file function
            return send_file(io.BytesIO(stream.read()), mimetype="image/jpeg")
```

In this example, we are also making use of the `responses` attribute, to document that our successful response (HTTP code 200) will return data with a mimetype `image/jpeg`, as well as `args` to accept optional parameters with POST requests.

7.6.2 Complete example

Adding this new view into our example extension, we now have:

```
import io  # Used in our capture action

from flask import send_file  # Used to send images from our server
from labthings import Schema, fields, find_component
from labthings.extensions import BaseExtension
from labthings.views import ActionView, PropertyView


# Create the extension class
class MyExtension(BaseExtension):
    def __init__(self):
        # Superclass init function
        super().__init__("com.myname.myextension", version="0.0.0")

        # Add our API Views (defined below MyExtension)
        self.add_view(ExampleIdentifyView, "/identify")
        self.add_view(ExampleRenameView, "/rename")

    def rename(self, microscope, new_name):
        """
        Rename the microscope
        """
        microscope.name = new_name
        microscope.save_settings()


# Define which properties of a Microscope object we care about,
# and what types they should be converted to
class MicroscopeIdentifySchema(Schema):
    name = fields.String()  # Microscopes name
    id = fields.UUID()  # Microscopes unique ID
    state = fields.Dict()  # Status dictionary
    camera = fields.String()  # Camera object (represented as a string)
    stage = fields.String()  # Stage object (represented as a string)


## Extension views

# Since we only have a GET method here, it'll register as a read-only property
class ExampleIdentifyView(PropertyView):
```

(continues on next page)

(continued from previous page)

```
    # Format our returned object using MicroscopeIdentifySchema
    schema = MicroscopeIdentifySchema()

    def get(self):
        """
        Show identifying information about the current microscope object
        """
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Return our microscope object,
        # let schema handle formatting the output
        return microscope


# We can use a single schema as the input and output will be formatted identically
# Eg. We always expect a "name" string argument, and always return a "name" string␣
↪attribute
class ExampleRenameView(PropertyView):
    schema = {"name": fields.String(required=True, example="My Example Microscope")}

    def get(self):
        """
        Show the current microscope name
        """
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        return microscope

    def post(self, args):
        """
        Change the current microscope name
        """
        # Look for our "name" parameter in the request arguments
        new_name = args.get("name")

        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Pass microscope and new name to our rename function
        self.extension.rename(microscope, new_name)

        # Return our microscope object,
        # let schema handle formatting the output
        return microscope


class QuickCaptureAPI(ActionView):
    """
    Take an image capture and return it without saving
    """

    # Expect a "use_video_port" boolean, which defaults to True if none is given
    args = {"use_video_port": fields.Boolean(missing=True)}

    # Our success response (200) returns an image (image/jpeg mimetype)
```

(continues on next page)

(continued from previous page)

```
    responses = {200: {"content": {"image/jpeg": {}}}}

    def post(self, args):
        """
        Take a non-persistant image capture.
        """
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Open a BytesIO stream to be destroyed once request has returned
        with io.BytesIO() as stream:

            # Capture to our stream object
            microscope.camera.capture(stream, use_video_port=args.get("use_video_port
↪"))

            # Rewind the stream
            stream.seek(0)

            # Return our image data using Flasks send_file function
            return send_file(io.BytesIO(stream.read()), mimetype="image/jpeg")


LABTHINGS_EXTENSIONS = (MyExtension,)
```

7.7 Threads and Locks

7.7.1 Introduction

Some actions in your extension may perform tasks that take a long time (compared to the expected response time of a web request). For example, if you were to implement a timelapse feature, this inherently runs over a long time.

This introduces a couple of problems. Firstly, a request that triggers a long function will, by default, block the Python interpreter for the duration of the function. This usually causes the connection to timeout, and the response will never be revieved.

Similarly, if your functionality takes a long time, it may be possible for other requests to interfere with your function. For example, in our hypothetical timelapse extension, while the timelapse is running, another user could open a connection and start moving the stage around, ruining the timelapse.

We get around these issues by making use of action threads, and component locks.

7.7.2 Action threads

Action threads are introduced to manage long-running functions in a way that does not block HTTP requests. Any API Action will automatically run as a background thread.

Internally, the `labthings.LabThing` object stores a list of all requested actions, and their states. This state stores the running status of the action (if itis idle, running, error, or success), information about the start and end times, a unique ID, and, upon completion, the return value of the long-running function.

By using threads, a function can be started in the background, and it's return value fetched at a later time once it has reported success. If a long-running action is started by some client, it should note the ID returned in the action state JSON, and use this to periodically check on the status of that particular action.

API routes have been created to allow checking the state of all actions (GET `/actions`), a particular action by ID (GET `/actions/<action_id>`), and stopping or removing individual actions (DELETE `/actions/<action_id>`).

All actions will return a serialized representation of the action state when your POST request returns. If the action completes within a default timeout period (usually 1 second) then the completed action representation will be returned. If the action is still running after this timeout period, the "in-progress" action representation will be returned. The final output value can then be retrieved at a later time.

Most users will not need to create instances of this class. Instead, they will be created automatically when a function is started by an API Action view.

An example of a long running task may look like:

```
...
from labthings import ActionView

class SlowAPI(ActionView):
    def post(self):
        # Return the task object.
        return long_running_function(function_argument_1, function_argument_2)
```

After some time, once the task has completed, it could be retreived using:

```
...
from labthings import current_labthing

def get_result(action_id):
    matching_action = current_labthing().actions.get(task_id)
    return matching_action.state
```

or by making GET requests to the `http://microscope.local/api/v2/tasks/<task_id>` view.

Accessing the current action instance

Every time a user requests your action, a new `labthings.actions.ActionThread` instance is created to hold the state of your action. This object holds return values, errors, action progress and status, and handles action cancellation.

In some cases, your action function will need to access the currently running `labthings.actions.ActionThread` instance. The `labthings.current_action()` function will return the currently running `labthings.actions.ActionThread` instance if it's called from within an `ActonThread`, and will return `None` if running outside of an *ActionThread*.

Handling action cancellation

Users always have the option to stop an action while it's running. Your action function has the option to support an elegant cancellation by watching for cancellation requests on the running `labthings.actions.ActionThread` instance.

The `labthings.current_action().stopped` attribute will return `True` if the Action has been requested to stop, and `False` otherwise. If your action runs a loop, this can be checked at each iteration, and used to return early if the action has been stopped.

If a stop request is sent and your action does not return within a timeout (by default 5 seconds), then the thread will be forcefully terminated. This is to ensure that actions can be stopped even if they have become stuck, or would otherwise

take an unexpected amount of time. However, every effort should be made to handle action cancellation elegantly from within the action.

An example of elegant action cancellation is included in the example later on this page.

The `ActionView.default_stop_timeout` class attribute can be used to increase or descrease the forced cancellation timeout. Developers should carefully consider how long their action should take to elegantly stop, and avoid abusing this timeout override to simply prevent forceful cancelltion.

Updating action progress

Some applications such as OpenFlexure eV are able to display progress bars showing the progress of an action thread. Implementing progress updates in your extension is made easy with the `labthings.update_action_progress()` function. This function takes a single argument, which is the action progress as an integer percent (0 - 100).

If your long running function was started within a background thread, this function will update the state of the corresponding action thread object. If your function is called outside of a long-running task (e.g. by another extension, directly), then this function will silently do nothing.

An example of task progress is included in the example later on this page.

7.7.3 Component Locks

Locks have been implemented to solve a distinct issue, most obvious when considering long-running actions. During a long action such as a tile-scan or autofocus, it is absolutely necesarry to block any completing interaction with the microscope hardware. For example, even if the stage is not actively moving (for example during a capture phase within a tile scan), another user should not be able to move the microscope, interrupting the action. Thread locks act to prevent this.

The camera and stage both contain an instance of `labthings.lock.StrictLock`, named `lock`. Built-in functions such as capture and move will always acquire this lock for the duration of the function. This ensures that, for example, simultaneous attemps to move do not occur.

More importantly, however, threads can hold on to these locks for longer periods of time, blocking any other calls to the hardware.

Locks are acquired using context managers, i.e. `with component.lock: ...`

7.7.4 Complete example

Implementing both action threads and locks in a new timelapse extension may look like:

```
import time  # Used in our timelapse function

from labthings import current_action, fields, find_component, update_action_progress
from labthings.extensions import BaseExtension
from labthings.views import ActionView

# Used in our timelapse function
from openflexure_microscope.captures.capture_manager import generate_basename


# Create the extension class
class TimelapseExtension(BaseExtension):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        # Superclass init function
        super().__init__("org.openflexure.timelapse-extension", version="0.0.0")

        # Add our API views
        self.add_view(TimelapseAPIView, "/timelapse")

    def timelapse(self, microscope, n_images, t_between):
        """
        Save a set of images in a timelapse

        Args:
            microscope: Microscope object
            n_images (int): Number of images to take
            t_between (int/float): Time, in seconds, between sequential captures
        """
        base_file_name = generate_basename()
        folder = "TIMELAPSE_{}".format(base_file_name)

        # Take exclusive control over both the camera and stage
        with microscope.camera.lock, microscope.stage.lock:
            for n in range(n_images):
                # Elegantly handle action cancellation
                if current_action() and current_action().stopped:
                    return
                # Generate a filename
                filename = f"{base_file_name}_image{n}"
                # Create a file to save the image to
                output = microscope.camera.new_image(
                    filename=filename, folder=folder, temporary=False
                )

                # Capture
                microscope.camera.capture(output)

                # Add system metadata
                output.put_metadata(microscope.metadata, system=True)

                # Update task progress (only does anyting if the function is running␣
↪in a LabThings task)
                progress_pct = ((n + 1) / n_images) * 100  # Progress, in percent
                update_action_progress(progress_pct)

                # Wait for the specified time
                time.sleep(t_between)


## Extension views


class TimelapseAPIView(ActionView):
    """
    Take a series of images in a timelapse
    """

    args = {
        "n_images": fields.Integer(
```

(continues on next page)

(continued from previous page)

```
                required=True, example=5, description="Number of images"
            ),
            "t_between": fields.Number(
                missing=1, example=1, description="Time (seconds) between images"
            ),
        }

        def post(self, args):
            # Find our microscope component
            microscope = find_component("org.openflexure.microscope")

            # Start "timelapse"
            return self.extension.timelapse(
                microscope, args.get("n_images"), args.get("t_between")
            )


LABTHINGS_EXTENSIONS = (TimelapseExtension,)
```

Notice that even though we never use the stage here, our `timelapse` function still acquires the stage lock. This means that during the timelapse, no other user is able to move the stage, or take separate captures. Control of the microscope is handed exclusively to the thread that obtains the lock, which in this case is the thread spawned when handling the POST request.

7.8 OpenFlexure eV GUI

7.8.1 Introduction

The main client application for the OpenFlexure Microscope, OpenFlexure eV, can render simple GUIs (graphical user interfaces) for extensions.

We define our user interface by making use of the extensions general metadata, added using the `add_meta` function. This function adds arbitrary additional data to your extensions web API description, for example:

```
# Create your extension object
my_extension = BaseExtension("com.myname.myextension", version="0.0.0")

...

my_extension.add_meta("myKey", "My metadata value")
```

OpenFlexure eV will recognise the `gui` metadata key, and render properly structured descriptions of a GUI in the format described below. The `gui` data essentially describes HTML forms, which it is up to the client to render. The form is constructed by specifying a set of components, and their values.

Each component in the form has a `name` property, which must match up to a property your API route expects in JSON POST requests, and returns in JSON GET requests.

7.8.2 Structure of `gui`

Root level

The root of your `gui` dictionary expects 2 properties:

`icon` - The name of a Material Design icon to use for your plugin

`viewPanel` *(optional)* - Content to display to the right of the extension form. Either `stream` (default), `gallery`, or `settings`.

`forms` - An array of forms as described below

Form level

Your extension can contain multiple forms. For example, if your extension creates several API routes, you will need a separate form for each route.

Each form is described by a JSON object, with the following properties:

`name` - A human-readable name for the form

`route` - String of the corresponding API route. *Must* match a route defined in your `api_views` dictionary

`isTask` *(optional)* - Whether the client should treat your API route as a long-running task

`isCollapsible` *(optional)* - Whether the form can be collapsed into an accordion

`submitLabel` *(optional)* - String to place inside of the form's submit button

`schema` - List of dictionaries. Each dictionary element describes a form component.

`emitOnResponse` *(optional)* - OpenFlexure eV event to emit when a response is recieved from the extension (generally avoid unless you know you need this.)

Component level

Each form can (and probably should) contain multiple components. For example, if your API route expects several parameters in a POST requests, each parameter can be bound to a form component.

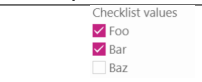
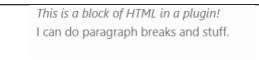
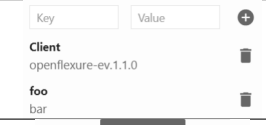
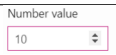
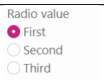
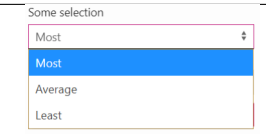
Upon form submission, the form data will be converted into a JSON object of key-value pairs, where the key is the components `name`, and the value is it's current value.

An overview of available components, and their properties, can be found below.

Arranging components

You can request that the client render several components in a horizontal grid by placing them in an array. You cannot nest arrays however. Each component in the array will be rendered with equal width as far as possible.

7.8.3 Overview of components

fieldType	Data type	Properties	Example
checkList	[str, str,...]	**name** (str) Unique name of the component **label** (str) Friendly label for the component **value** ([str, str,...]) List of selected options **options** ([str, str,...]) List of all options	
htmlBlock	N/A	**name** (str) Unique name of the component **label** (str) Friendly label for the component **content** (str) HTML string to be rendered	
keyvalList	dict	**name** (str) Unique name of the component **value** (dict) Dictionary of key-value pairs	
labelInput	str	**name** (str) Unique name of the component **label** (str) Friendly label for the component **value** (str) Value of the editable label text	
numberInput	int	**name** (str) Unique name of the component **label** (str) Friendly label for the component **value** (int) Value of the input **placeholder** (int) Placeholder value	
radioList	String	**name** (str) Unique name of the component **label** (str) Friendly label for the component **value** (str) Currently selected option **options** ([str, str,...]) List of all options	
selectList	str	**name** (str) Unique name of the component **label** (str) Friendly label for the component **value** (str) Currently selected option **options** ([str, str,...]) List of all options	
tagList	[str, str,...]	**name** (str) Unique name of the component **value** ([str, str,...]) List of tag strings	
textInput	str	**name** (str) Unique name of the component **label** (str) Friendly label for the component **value** (int) Value of the input **placeholder** (str) Placeholder value	

Note: Basic input types (`textInput`, `numberInput`) can also include additional attributes for HTML input elements inputs (e.g. `placeholder`, `required`, `min`, `max`). These additional attributes will be forwarded to the rendered HTML elements.

7.8.4 Building the GUI

Once you have a dictionary describing your GUI, use the `openflexure_microscope.api.utilities.gui.build_gui()` function to fill in and expand any information required to have it properly function. This function expands your `route` values to include your extensions full URI, and handles returning dynamic GUIs.

For example:

```
my_gui = {...}

# Create your extension object
my_extension = BaseExtension("com.myname.myextension", version="0.0.0")

...

my_extension.add_meta("gui", build_gui(my_gui, my_extension))
```

7.8.5 Dynamic GUIs

Instead of passing a static dictionary to `openflexure_microscope.api.utilities.gui.build_gui()`, you can instead pass a callable function which returns a dictionary. This function is then called every time a client requests a description of active extensions.

Using a callable has the advantage of allowing your extensions GUI to be updated as it is used. This could be as simple as changing `value` parameters of components (to show up-to-date default form values), but could be used to entirely change the GUI form as it is used, for example dynamically changing options in select boxes.

For example, this could take the form:

```
def create_dynamic_form():
    ...
    generated_form_dict = {...}
    return generated_form_dict

# Create your extension object
my_extension = BaseExtension("com.myname.myextension", version="0.0.0")

...

my_extension.add_meta("gui", build_gui(create_dynamic_form, my_extension))
```

7.8.6 Complete example

Adding a GUI to our previous timelapse example extension becomes:

```
import time  # Used in our timelapse function

from labthings import current_action, fields, find_component, update_action_progress
from labthings.extensions import BaseExtension
from labthings.views import ActionView

# Used to convert our GUI dictionary into a complete eV extension GUI
from openflexure_microscope.api.utilities.gui import build_gui

# Used in our timelapse function
from openflexure_microscope.captures.capture_manager import generate_basename


# Create the extension class
class TimelapseExtension(BaseExtension):
    def __init__(self):
        # Superclass init function
```

(continues on next page)

(continued from previous page)

```
        super().__init__("org.openflexure.timelapse-extension", version="0.0.0")

        # Add our API views
        self.add_view(TimelapseAPIView, "/timelapse")

        # Add our GUI description
        gui_description = {
            "icon": "timelapse",  # Name of an icon from https://material.io/
↪resources/icons/
            "forms": [  # List of forms. Each form is a collapsible accordion panel
                {
                    "name": "Start a timelapse",  # Form title
                    "route": "/timelapse",  # The URL rule (as given by "add_view")␣
↪of your submission view
                    "isTask": True,  # This forms submission starts a background task
                    "isCollapsible": False,  # This form cannot be collapsed into an␣
↪accordion
                    "submitLabel": "Start",  # Label for the form submit button
                    "schema": [  # List of dictionaries. Each element is a form␣
↪component.
                        {
                            "fieldType": "numberInput",
                            "name": "n_images",  # Name of the view arg this value␣
↪corresponds to
                            "label": "Number of images",
                            "min": 1,  # HTML number input attribute
                            "default": 5,  # HTML number input attribute
                        },
                        {
                            "fieldType": "numberInput",
                            "name": "t_between",
                            "label": "Time (seconds) between images",
                            "min": 0.1,  # HTML number input attribute
                            "step": 0.1,  # HTML number input attribute
                            "default": 1,  # HTML number input attribute
                        },
                    ],
                }
            ],
        }
        self.add_meta("gui", build_gui(gui_description, self))

    def timelapse(self, microscope, n_images, t_between):
        """
        Save a set of images in a timelapse

        Args:
            microscope: Microscope object
            n_images (int): Number of images to take
            t_between (int/float): Time, in seconds, between sequential captures
        """
        base_file_name = generate_basename()
        folder = "TIMELAPSE_{}".format(base_file_name)

        # Take exclusive control over both the camera and stage
        with microscope.camera.lock, microscope.stage.lock:
            for n in range(n_images):
```

(continues on next page)

(continued from previous page)

```
                # Elegantly handle action cancellation
                if current_action() and current_action().stopped:
                    return
                # Generate a filename
                filename = f"{base_file_name}_image{n}"
                # Create a file to save the image to
                output = microscope.camera.new_image(
                    filename=filename, folder=folder, temporary=False
                )

                # Capture
                microscope.camera.capture(output)

                # Add system metadata
                output.put_metadata(microscope.metadata, system=True)

                # Update task progress (only does anyting if the function is running␣
↪in a LabThings task)
                progress_pct = ((n + 1) / n_images) * 100  # Progress, in percent
                update_action_progress(progress_pct)

                # Wait for the specified time
                time.sleep(t_between)


## Extension views
class TimelapseAPIView(ActionView):
    """
    Take a series of images in a timelapse
    """

    args = {
        "n_images": fields.Integer(
            required=True, example=5, description="Number of images"
        ),
        "t_between": fields.Number(
            missing=1, example=1, description="Time (seconds) between images"
        ),
    }

    def post(self, args):
        # Find our microscope component
        microscope = find_component("org.openflexure.microscope")

        # Start "timelapse"
        return self.extension.timelapse(
            microscope, args.get("n_images"), args.get("t_between")
        )


LABTHINGS_EXTENSIONS = (TimelapseExtension,)
```

7.9 Lifecycle Hooks

7.9.1 Introduction

In some cases it is useful to have functions triggered by events in an extensions lifecycle. Currently two such lifecycle events can be used, `on_register`, and `on_component`.

7.9.2 `on_register`

The `on_register` method can be used to have a function call as soon as the extension has been successfully registered to the microscope. For example:

```
class MyExtension(BaseExtension):
    def __init__(self):

        # Track if the extension has been registered
        self.registered = False

        # Add lifecycle hooks
        self.on_register(self.on_register_handler, args=(), kwargs={})

        # Superclass init function
        super().__init__("com.myname.myextension", version="0.0.0")

    def on_register_handler(self, *args, **kwargs):
        self.registered = True
        print("Extension has been registered!")
```

7.9.3 `on_component`

The `on_component` method can be used to have a function call as soon as a particular LabThings component has been added. This can be used, for example, to get information about the microscope instance as soon as it is available. For example:

```
class MyExtension(BaseExtension):
    def __init__(self):

        # Hold a reference to the microscope object as soon as it is available
        self.microscope = None

        # Add lifecycle hooks
        self.on_component("com.myname.myextension", self.on_microscope_handler)

        # Superclass init function
        super().__init__("org.openflexure.microscope", version="0.0.0")

    def on_microscope_handler(self, microscope_object):
        print("Microscope object has been found!")
        self.microscope = microscope_object
```

CHAPTER 8

HTTP API

8.1 Live documentation

Full, interactive Swagger documentation for your microscopes web API is available from the microscope itself. From any browser, go to `http://{your microscope IP address}/api/v2/docs/swagger-ui`.

The API is described in an OpenAPI description, available at `http://{your microscope IP address}/api/v2/docs/openapi.yaml`. It is also available from our build server. It can be conveniently viewed using Redoc's online preview.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

openflexure_microscope.camera.base, 18
openflexure_microscope.camera.pi, 15
openflexure_microscope.captures.capture, 19
openflexure_microscope.config, 11
openflexure_microscope.microscope, 13
openflexure_microscope.stage.base, 24
openflexure_microscope.stage.sanga, 23

Index

Symbols

_backlash (*openflexure_microscope.stage.sanga.SangaStage attribute*), 23

A

apply_picamera_settings() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 16
array() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 17

B

backlash (*openflexure_microscope.stage.base.BaseStage attribute*), 25
backlash (*openflexure_microscope.stage.sanga.SangaStage attribute*), 23
BaseCamera (*class in openflexure_microscope.camera.base*), 18
BaseStage (*class in openflexure_microscope.stage.base*), 24
binary (*openflexure_microscope.captures.capture.CaptureObject attribute*), 20
board (*openflexure_microscope.stage.sanga.SangaStage attribute*), 23

C

camera (*openflexure_microscope.microscope.Microscope attribute*), 13
capture() (*openflexure_microscope.camera.base.BaseCamera method*), 19
capture() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 17
CaptureObject (*class in openflexure_microscope.captures.capture*), 19
close() (*openflexure_microscope.camera.base.BaseCamera method*), 19
close() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 16
close() (*openflexure_microscope.microscope.Microscope method*), 13
close() (*openflexure_microscope.stage.base.BaseStage method*), 25
close() (*openflexure_microscope.stage.sanga.SangaStage method*), 24
configuration (*openflexure_microscope.camera.base.BaseCamera attribute*), 18
configuration (*openflexure_microscope.camera.pi.PiCameraStreamer attribute*), 16
configuration (*openflexure_microscope.stage.base.BaseStage attribute*), 25
configuration (*openflexure_microscope.stage.sanga.SangaStage attribute*), 23
create_file() (*in module openflexure_microscope.config*), 12

D

data (*openflexure_microscope.captures.capture.CaptureObject attribute*), 20
delete() (*openflexure_microscope.captures.capture.CaptureObject method*), 20
delete_tag() (*openflexure_microscope.captures.capture.CaptureObject method*), 20

E

exists (*openflexure_microscope.captures.capture.CaptureObject attribute*), 19

F

force_get_metadata() (*openflexure_microscope.microscope.Microscope method*), 14

FrameStream (*class in openflexure_microscope.camera.base*), 18

G

get_frame() (*openflexure_microscope.camera.base.BaseCamera method*), 19
get_metadata() (*openflexure_microscope.microscope.Microscope method*), 14
getframe() (*openflexure_microscope.camera.base.FrameStream method*), 18
getvalue() (*openflexure_microscope.camera.base.FrameStream method*), 18

H

has_real_camera() (*openflexure_microscope.microscope.Microscope method*), 13
has_real_stage() (*openflexure_microscope.microscope.Microscope method*), 13

I

id (*openflexure_microscope.captures.capture.CaptureObject attribute*), 19
image_resolution (*openflexure_microscope.camera.pi.PiCameraStreamer attribute*), 15
initialise_file() (*in module openflexure_microscope.config*), 12

J

jpeg_quality (*openflexure_microscope.camera.pi.PiCameraStreamer attribute*), 16

L

load() (*openflexure_microscope.config.OpenflexureSettingsFile method*), 11
load_json_file() (*in module openflexure_microscope.config*), 11
lock (*openflexure_microscope.camera.base.BaseCamera attribute*), 18
lock (*openflexure_microscope.microscope.Microscope attribute*), 13
lock (*openflexure_microscope.stage.base.BaseStage attribute*), 24

M

merge() (*openflexure_microscope.config.OpenflexureSettingsFile method*), 11
metadata (*openflexure_microscope.captures.capture.CaptureObject attribute*), 20
Microscope (*class in openflexure_microscope.microscope*), 13
mjpeg_bitrate (*openflexure_microscope.camera.pi.PiCameraStreamer attribute*), 16
mjpeg_quality (*openflexure_microscope.camera.pi.PiCameraStreamer attribute*), 16
move_abs() (*openflexure_microscope.stage.base.BaseStage method*), 25
move_abs() (*openflexure_microscope.stage.sanga.SangaDeltaStage method*), 24
move_abs() (*openflexure_microscope.stage.sanga.SangaStage method*), 24
move_rel() (*openflexure_microscope.stage.base.BaseStage method*), 25
move_rel() (*openflexure_microscope.stage.sanga.SangaDeltaStage method*), 24
move_rel() (*openflexure_microscope.stage.sanga.SangaStage method*), 24

N

n_axes (*openflexure_microscope.stage.base.BaseStage attribute*), 25
n_axes (*openflexure_microscope.stage.sanga.SangaStage attribute*), 23
numpy_resolution (*openflexure_microscope.camera.pi.PiCameraStreamer attribute*), 16

O

openflexure_microscope.camera.base (*module*), 18
openflexure_microscope.camera.pi (*module*), 15
openflexure_microscope.captures.capture (*module*), 19
openflexure_microscope.config (*module*), 11
openflexure_microscope.microscope (*module*), 13
openflexure_microscope.stage.base (*module*), 24
openflexure_microscope.stage.sanga (*module*), 23
OpenflexureSettingsFile (*class in openflexure_microscope.config*), 11

P

picamera (*openflexure_microscope.camera.pi.PiCameraStreamer attribute*), 15
PiCameraStreamer (*class in openflexure_microscope.camera.pi*), 15
position (*openflexure_microscope.stage.base.BaseStage attribute*), 25
position (*openflexure_microscope.stage.sanga.SangaDeltaStage attribute*), 24
position (*openflexure_microscope.stage.sanga.SangaStage attribute*), 23
put_and_save() (*openflexure_microscope.captures.capture.CaptureObject method*), 20
put_annotations() (*openflexure_microscope.captures.capture.CaptureObject method*), 20
put_metadata() (*openflexure_microscope.captures.capture.CaptureObject method*), 20
put_tags() (*openflexure_microscope.captures.capture.CaptureObject method*), 20

R

read_settings() (*openflexure_microscope.camera.base.BaseCamera method*), 19
read_settings() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 16
read_settings() (*openflexure_microscope.microscope.Microscope method*), 14
read_settings() (*openflexure_microscope.stage.base.BaseStage method*), 25
read_settings() (*openflexure_microscope.stage.sanga.SangaStage method*), 24
release_motors() (*openflexure_microscope.stage.sanga.SangaStage method*), 24
reset_tracking() (*openflexure_microscope.camera.base.FrameStream method*), 18

S

SangaDeltaStage (*class in openflexure_microscope.stage.sanga*), 24
SangaStage (*class in openflexure_microscope.stage.sanga*), 23
save() (*openflexure_microscope.captures.capture.CaptureObject method*), 20
save() (*openflexure_microscope.config.OpenflexureSettingsFile method*), 11
save_json_file() (*in module openflexure_microscope.config*), 12
save_settings() (*openflexure_microscope.microscope.Microscope method*), 14
scan_linear() (*openflexure_microscope.stage.base.BaseStage method*), 25
scan_z() (*openflexure_microscope.stage.base.BaseStage method*), 25
set_stage() (*openflexure_microscope.microscope.Microscope method*), 13
set_zoom() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 16
setup() (*openflexure_microscope.microscope.Microscope method*), 13
size (*openflexure_microscope.camera.base.TrackerFrame attribute*), 18
split_file_path() (*openflexure_microscope.captures.capture.CaptureObject method*), 19
stage (*openflexure_microscope.microscope.Microscope attribute*), 13
start_preview() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 16
start_recording() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 16
start_stream() (*openflexure_microscope.camera.base.BaseCamera method*), 18
start_stream() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 17
start_tracking() (*openflexure_microscope.camera.base.FrameStream method*), 18
start_worker() (*openflexure_microscope.camera.base.BaseCamera method*), 19
state (*openflexure_microscope.camera.base.BaseCamera attribute*), 18
state (*openflexure_microscope.camera.pi.PiCameraStreamer attribute*), 16
state (*openflexure_microscope.microscope.Microscope attribute*), 13
state (*openflexure_microscope.stage.base.BaseStage attribute*), 25
state (*openflexure_microscope.stage.sanga.SangaStage*

attribute), 23
stop_preview() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 16
stop_recording() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 17
stop_stream() (*openflexure_microscope.camera.base.BaseCamera method*), 18
stop_stream() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 17
stop_tracking() (*openflexure_microscope.camera.base.FrameStream method*), 18
stream (*openflexure_microscope.camera.base.BaseCamera attribute*), 18
stream_resolution (*openflexure_microscope.camera.pi.PiCameraStreamer attribute*), 15

T

thumbnail (*openflexure_microscope.captures.capture.CaptureObject attribute*), 20
time (*openflexure_microscope.camera.base.TrackerFrame attribute*), 18
TrackerFrame (*class in openflexure_microscope.camera.base*), 18

U

update_settings() (*openflexure_microscope.camera.base.BaseCamera method*), 18
update_settings() (*openflexure_microscope.camera.pi.PiCameraStreamer method*), 16
update_settings() (*openflexure_microscope.microscope.Microscope method*), 14
update_settings() (*openflexure_microscope.stage.base.BaseStage method*), 25
update_settings() (*openflexure_microscope.stage.sanga.SangaStage method*), 24
user_configuration (*in module openflexure_microscope.config*), 12
user_settings (*in module openflexure_microscope.config*), 12

W

write() (*openflexure_microscope.camera.base.FrameStream method*), 18

Z

zero_position() (*openflexure_microscope.stage.base.BaseStage method*), 25
zero_position() (*openflexure_microscope.stage.sanga.SangaStage method*), 24