

---

# **OpenFlexure Microscope Software Documentation**

**Bath Open Instrumentation Group**

**Jul 06, 2020**



---

## Contents:

---

<b>1</b>	<b>Quickstart</b>	<b>1</b>
1.1	Install . . . . .	1
1.2	Managing the server . . . . .	1
<b>2</b>	<b>Microscope settings</b>	<b>3</b>
2.1	Microscope settings file . . . . .	3
<b>3</b>	<b>Microscope class</b>	<b>5</b>
<b>4</b>	<b>Camera Class</b>	<b>7</b>
4.1	Raspberry Pi Streaming Camera . . . . .	7
4.2	Base Streaming Camera . . . . .	10
4.3	Capture Object . . . . .	11
4.4	Capturing from a camera object . . . . .	12
<b>5</b>	<b>Stage Class</b>	<b>13</b>
5.1	Sangaboard Microscope Stage . . . . .	13
5.2	Base Microscope Stage . . . . .	14
<b>6</b>	<b>Developing API Extensions</b>	<b>17</b>
6.1	Introduction . . . . .	17
6.2	Basic extension structure . . . . .	18
6.3	Adding web API views . . . . .	20
6.4	Marshaling data . . . . .	22
6.5	Thing Properties . . . . .	26
6.6	Thing Actions . . . . .	30
6.7	Tasks and Locks . . . . .	33
6.8	OpenFlexure eV GUI . . . . .	36
6.9	Lifecycle Hooks . . . . .	42
<b>7</b>	<b>HTTP API</b>	<b>45</b>
7.1	Live documentation . . . . .	45
<b>8</b>	<b>Indices and tables</b>	<b>47</b>
	<b>Python Module Index</b>	<b>49</b>
	<b>Index</b>	<b>51</b>



## 1.1 Install

### 1.1.1 Stable installation

For most users, the OpenFlexure Microscope software should be installed using our [pre-built Raspbian SD card image](#).

### 1.1.2 Manual installation

For offline (i.e. no real microscope connected) development, a basic development server can run on any system.

- (Recommended) create and activate a virtual environment
- Install non-python dependencies with `sudo apt-get install libatlas-base-dev libjasper-dev libjpeg-dev`
- Install [Poetry](#), clone this repo, and `poetry install` from inside the repo.

## 1.2 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.



---

Microscope settings

---

## 2.1 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 = {}*)

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

```
class openflexure_microscope.config.JSONEncoder (*, skipkeys=False, ensure_ascii=True,
                                                check_circular=True,          al-
                                                low_nan=True,          sort_keys=False,
                                                indent=None,          separators=None,
                                                default=None)
```

A custom JSON encoder, with type conversions for PiCamera fractions, Numpy integers, and Numpy arrays

**default** (o, markers=None)

Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    return JSONEncoder.default(self, o)
```

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 nonexistent.

**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 3

---

### 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.OpenflexureSettings.  
                                                    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.

**fov = None**

Microscope field-of-view in stage motor steps

**lock = None**

Composite lock for locking both camera and stage

**Type** labthings.lock.CompositeLock

**camera = None**

Currently connected camera object

**stage = None**

Currently connected stage object

**close()**

Shut down the microscope hardware.

**setup(configuration)**

Attach microscope components based on initially passed configuration file

**has\_real\_stage()** → bool

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

**has\_real\_camera()**

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*)

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** ()

Read cachable bits of microscope metadata. Currently ID, settings, and configuration can be cached

### **get\_metadata** (*cache\_key=None*)

Read microscope metadata, with partial caching

### 4.1 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.

```
    camera = None
```

Picamera object

```
        Type picamera.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

**picamera\_lst\_path = None**

Path of .npy lens shading table file

**Type** str

**configuration**

The current camera configuration.

**state**

The current read-only camera state.

**initialisation()**

Run any initialisation code when the frame iterator starts.

**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 `picamera.PiCamera`: object
- **pause\_for\_effect** (bool) – Pause tactically to reduce risk of timing issues

**set\_zoom(zoom\_value: float = 1.0) → None**

Change the camera zoom, handling re-centering and scaling.

**start\_preview(fullscreen=True, window=None)**

Start the on board GPU camera preview.

**stop\_preview()**

Stop the on board GPU camera preview.

**start\_recording** (*output*, *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.

**stop\_stream\_recording** (*splitter\_port*: *int* = 1, *resolution*: *Tuple*[*int*, *int*] = *None*) → *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
- **resolution** ((*int*, *int*)) – Resolution to set the camera to, after stopping recording.

**start\_stream\_recording** (*splitter\_port*: *int* = 1, *resolution*: *Tuple*[*int*, *int*] = *None*) → *None*

Sets the camera resolution to the video/stream resolution, and starts recording if the stream should be active.

#### Parameters

- **splitter\_port** (*int*) – Splitter port to start recording on
- **resolution** ((*int*, *int*)) – Resolution to set the camera to, before starting recording. Defaults to *self.stream\_resolution*.

**capture** (*output*, *fmt*: *str* = 'jpeg', *use\_video\_port*: *bool* = *False*, *resize*: *Tuple*[*int*, *int*] = *None*, *bayer*: *bool* = *True*)

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** (*str*) – Format of the capture.
- **use\_video\_port** (*bool*) – Capture from the video port used for streaming. Lower resolution, faster.
- **resize** ((*int*, *int*)) – Resize the captured image.
- **bayer** (*bool*) – Store raw bayer data in capture

**Returns** Target object.

**Return type** output\_object (str/BytesIO)

**yuv** (*use\_video\_port*: *bool* = *True*, *resize*: *Tuple*[*int*, *int*] = *None*) → *numpy.ndarray*

Capture an uncompressed still YUV 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)

**array** (*use\_video\_port: bool = True, resize: Tuple[int, int] = None*) → 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)

**wait\_for\_camera** (*timeout=5*)

Wait for camera object, with 5 second timeout.

**frames** ()

Create generator that returns frames from the camera.

Records video from port 1 to a byte stream, and iterates sequential frames.

## 4.2 Base Streaming Camera

**class** openflexure\_microscope.camera.base.BaseCamera

Base implementation of StreamingCamera.

**configuration**

The current camera configuration.

**state**

The current read-only camera state.

**update\_settings** (*config: dict*)

Update settings from a config dictionary

**read\_settings** () → dict

Return the current settings as a dictionary

**close** ()

Close the BaseCamera and all attached StreamObjects.

**start\_worker** (*timeout: int = 5*) → bool

Start the background camera thread if it isn't running yet.

**stop\_worker** (*timeout: int = 5*) → bool

Flag worker thread for stop. Waits for thread close or timeout.

**get\_frame** ()

Return the current camera frame.

**frames** ()

Create generator that returns frames from the camera.

## 4.3 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.

`openflexure_microscope.captures.capture.pull_usercomment_dict(filepath)`

Reads UserComment Exif data from a file, and returns the contained bytes as a dictionary. :param filepath: Path to the Exif-containing file

`openflexure_microscope.captures.capture.capture_from_exif(path, exif_dict)`

Creates an instance of `CaptureObject` from a dictionary of capture information. This is used when reloading the API server, to restore captures created in the previous session.

### Parameters

- **path** (*str*) – Path to image file
- **exif\_dict** (*dict*) – Dictionary containing capture information

**class** `openflexure_microscope.captures.capture.CaptureObject(filepath)`

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*)

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*)

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*) → `None`

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*) → `None`

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: list = None, annotations: dict = None, metadata: dict = None*)

Batch-write tags, metadata, and annotations in a single disk operation

**save\_metadata()** → None

Save metadata to exif, if supported

**metadata**

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

**state**

Return a dictionary of objects full state, including metadata.

**data**

Return a byte string 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()** → None

Write stream to file, and save/update metadata file

**delete()** → bool

If the StreamObject has been saved, delete the file.

## 4.4 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 retrieving 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
...

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)
```

## 5.1 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: list, axis=None, backlash=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.

**move\_abs** (*final, \*\*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

## 5.2 Base Microscope Stage

```
class openflexure_microscope.stage.base.BaseStage
```

**lock**

Strict lock controlling thread access to camera hardware

**Type** `labthings.lock.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, backlash=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*, *\*\*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*, *backlash=True*, *return\_to\_start=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*, *\*\*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*.



---

## Developing API Extensions

---

### 6.1 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 `picamera.PiCamera` object.

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

#### 6.1.1 Single-file extensions

For adding simple functionality, such as a few basic functions and API routes, a single Python file can be loaded as an 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`).

#### 6.1.2 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 objects.

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.

## 6.2 Basic extension structure

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

In order to access the currently running microscope object, use the `labthings.server.find.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.server.extensions import BaseExtension
from labthings.server.find import find_component

def identify():
    """
    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(new_name):
    """
    Rename the microscope
    """

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

    microscope.name = new_name
    microscope.save_settings()

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

# Add methods to your extension
my_extension.add_method(identify, "identify")
my_extension.add_method(rename, "rename")
```

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

```
from labthings.server.find 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")
```

(continues on next page)

(continued from previous page)

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

## 6.2.1 Subclassing BaseExtension

The syntax used above allows novice programmers to easily start building extensions, without having to deal with subclassing. However, for more complex extensions which require persistent state, subclassing `labthings.server.extensions.BaseExtension` is recommended.

The same simple extension as seen above can be written using subclassing:

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

# Create the extension class
class MyExtension(BaseExtension):
    def __init__(self):

        # Create some instance variable
        self.state_variable = "An example of a persistent instance variable"

        # Superclass init function
        super().__init__("com.mynamespace.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()

# Create your extension object
my_extension = MyExtension()
```

## 6.3 Adding web API views

### 6.3.1 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.server.extensions import BaseExtension
from labthings.server.find import find_component
from labthings.server.view import View

from labthings.server import fields

## Extension methods

def identify(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(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 identify(microscope)
```

(continues on next page)

(continued from previous page)

```

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
        rename(microscope, new_name)

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

## Create extension

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

# Add methods to your extension
my_extension.add_method(identify, "identify")
my_extension.add_method(rename, "rename")

# Add API views to your extension
my_extension.add_view(ExampleIdentifyView, "/identify")
my_extension.add_view(ExampleRenameView, "/rename")

```

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.

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/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`.

## 6.4 Marshaling data

### 6.4.1 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.server.schema import Schema
from labthings.server 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,
    "job": "Python developer"
}

```

This JSON data is the parsed, converted into a Python dictionary, and passed as an argument. Retrieving 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.server.extensions import BaseExtension
from labthings.server.find import find_component
from labthings.server.view import View

from labthings.server.schema import Schema
from labthings.server import fields

## Extension methods

# 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)

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

    microscope.name = new_name
    microscope.save_settings()

## 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")
```

(continues on next page)

(continued from previous page)

```

    # 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

## Create extension

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

# Add methods to your extension
my_extension.add_method(rename, "rename")

# Add API views to your extension
my_extension.add_view(ExampleIdentifyView, "/identify")
my_extension.add_view(ExampleRenameView, "/rename")

```

## 6.5 Thing Properties

### 6.5.1 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.

### 6.5.2 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")

```

(continues on next page)

(continued from previous page)

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

### 6.5.3 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):
```

(continues on next page)

(continued from previous page)

```

"""
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

```

### 6.5.4 Complete example

Combining these into our example extension, we now have:

```

from labthings.server.extensions import BaseExtension
from labthings.server.find import find_component
from labthings.server.view import View, PropertyView

from labthings.server.schema import Schema
from labthings.server import fields

## Extension methods

# 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)

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

    microscope.name = new_name
    microscope.save_settings()

## Extension views

# 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()

```

(continues on next page)

(continued from previous page)

```

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
        rename(microscope, new_name)

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

## Create extension

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

# Add methods to your extension
my_extension.add_method(rename, "rename")

# Add API views to your extension
my_extension.add_view(ExampleIdentifyView, "/identify")
my_extension.add_view(ExampleRenameView, "/rename")

```

## 6.6 Thing Actions

### 6.6.1 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_type": "image/jpeg"}
    }

    def post(self, args):
        """
        Take a non-persistent 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", True))

            # 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.

### 6.6.2 Complete example

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

```

from labthings.server.extensions import BaseExtension
from labthings.server.find import find_component
from labthings.server.view import View, ActionView, PropertyView

from labthings.server.schema import Schema
from labthings.server import fields

from flask import send_file # Used to send images from our server
import io # Used in our capture action

## Extension methods

# 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)

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

    microscope.name = new_name
    microscope.save_settings()

## Extension views

# 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")}

```

(continues on next page)

(continued from previous page)

```

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

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_type": "image/jpeg"}
    }

    def post(self, args):
        """
        Take a non-persistent 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)

```

(continues on next page)

(continued from previous page)

```

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

## Create extension

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

# Add methods to your extension
my_extension.add_method(rename, "rename")

# Add API views to your extension
my_extension.add_view(ExampleIdentifyView, "/identify")
my_extension.add_view(ExampleRenameView, "/rename")
my_extension.add_view(QuickCaptureAPI, "/quick-capture")

```

## 6.7 Tasks and Locks

### 6.7.1 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 reviewed.

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 background tasks, and component locks.

### 6.7.2 Background tasks

Tasks 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 task (that is, any View that subclasses *ActionView*).

Internally, the `tasks` submodule stores a list of all requested tasks, and their states. This state stores the running status of the task (if it is idle, running, error, or success), information about the start and end times, a unique task ID, and, upon completion, the return value of the long-running function.

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

API routes have been created to allow checking the state of all tasks (GET `/tasks`), a particular task by ID (GET `/tasks/<task_id>`), and terminating or removing individual tasks (DELETE `/tasks/<task_id>`).

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

An example of a long running task may look like:

```
...
from labthings.server.view 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 retrieved using:

```
...
from labthings import tasks

def get_result(task_id):
    matching_task = tasks.dict.get(task_id)
    return matching_task.state
```

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

### Updating task progress

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

If your long running function was started within a background task, this function will update the state of the corresponding task 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.

### 6.7.3 Component Locks

Locks have been implemented to solve a distinct issue, most obvious when considering long-running tasks. During a long task such as a tile-scan or autofocus, it is absolutely necessary 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 task. 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 attempts to move do not occur.

More importantly, however, tasks 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:` ...

### 6.7.4 Complete example

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

```

from labthings.server.extensions import BaseExtension
from labthings.server.find import find_component
from labthings.server.view import View, ActionView

from labthings.server.schema import Schema
from labthings.server import fields

from flask import send_file # Used to send images from our server
import io # Used in our capture action
import time # Used in our timelapse function

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

# Used to run our timelapse in a background thread
from labthings.core.tasks import update_task_progress

## Extension methods

def timelapse(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):
            # 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 anything if the function is running in a
            ↪ LabThings task)
            progress_pct = ((n + 1) / n_images) * 100 # Progress, in percent
            update_task_progress(progress_pct)

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

```

(continues on next page)

(continued from previous page)

```

## Extension views

class TimelapseAPI(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 timelapse(
            microscope, args.get("n_images"), args.get("t_between")
        )

## Create extension

# Create your extension object
my_extension = BaseExtension("com.mynamespace.timelapse-extension", version="0.0.0")

# Add methods to your extension
my_extension.add_method(timelapse, "timelapse")

# Add API views to your extension
my_extension.add_view(TimelapseAPI, "/timelapse")

```

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.

## 6.8 OpenFlexure eV GUI

### 6.8.1 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.

## 6.8.2 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.

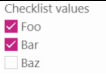



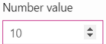
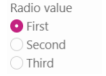
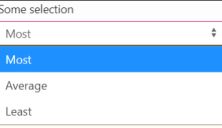


Upon form submission, the form data will be converted into a JSON object of key-value pairs, where the key is the components `name`, and the value is it's current value.

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.

### 6.8.3 Overview of components

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

## 6.8.4 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))
```

## 6.8.5 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))
```

## 6.8.6 Complete example

Adding a GUI to our previous timelapse example extension becomes:

```
from labthings.server.extensions import BaseExtension
from labthings.server.find import find_component
from labthings.server.view import View, ActionView

from labthings.server.schema import Schema
from labthings.server import fields

from flask import send_file # Used to send images from our server
import io # Used in our capture action
import time # Used in our timelapse function
```

(continues on next page)

(continued from previous page)

```

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

# Used to run our timelapse in a background thread
from labthings.core.tasks import update_task_progress

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

## Extension methods

def timelapse(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):
            # 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 anything if the function is running in a
↳LabThings task)
            progress_pct = ((n + 1) / n_images) * 100 # Progress, in percent
            update_task_progress(progress_pct)

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

## Extension views

class TimelapseAPI(ActionView):
    """
    Take a series of images in a timelapse, running as a background task
    """

```

(continues on next page)

(continued from previous page)

```

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")

    # Create and start "timelapse", running in a background task
    return timelapse(
        microscope, args.get("n_images"), args.get("t_between")
    )

## Extension GUI (OpenFlexure eV)
# Alternate form without any dynamic parts
extension_gui = {
    "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
                },
            ],
        },
    ],
}

## Create extension
# Create your extension object

```

(continues on next page)

(continued from previous page)

```
my_extension = BaseExtension("com.myname.timelapse-extension", version="0.0.0")

# Add methods to your extension
my_extension.add_method(timelapse, "timelapse")

# Add API views to your extension
my_extension.add_view(TimelapseAPI, "/timelapse")

# Add OpenFlexure eV GUI to your extension
my_extension.add_meta("gui", build_gui(extension_gui, my_extension))
```

## 6.9 Lifecycle Hooks

### 6.9.1 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`.

### 6.9.2 `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!")
```

### 6.9.3 `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
```

(continues on next page)

(continued from previous page)

```
# 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
```



### 7.1 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/swagger-ui`.

---

**Note:** We should have an online copy of the API SwaggerUI documentation up soon.

---



## CHAPTER 8

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### O

`openflexure_microscope.camera.base`, [10](#)  
`openflexure_microscope.camera.pi`, [7](#)  
`openflexure_microscope.captures.capture`,  
    [11](#)  
`openflexure_microscope.config`, [3](#)  
`openflexure_microscope.microscope`, [5](#)  
`openflexure_microscope.stage.base`, [14](#)  
`openflexure_microscope.stage.sanga`, [13](#)



## Symbols

`_backlash` (*openflexure\_microscope.stage.sanga.SangaStage attribute*), 13

## A

`apply_picamera_settings()` (*openflexure\_microscope.camera.pi.PiCameraStreamer method*), 8

`array()` (*openflexure\_microscope.camera.pi.PiCameraStreamer method*), 10

## B

`backlash` (*openflexure\_microscope.stage.base.BaseStage attribute*), 14

`backlash` (*openflexure\_microscope.stage.sanga.SangaStage attribute*), 13

`BaseCamera` (class in *openflexure\_microscope.camera.base*), 10

`BaseStage` (class in *openflexure\_microscope.stage.base*), 14

`binary` (*openflexure\_microscope.captures.capture.CaptureObject attribute*), 12

`board` (*openflexure\_microscope.stage.sanga.SangaStage attribute*), 13

## C

`camera` (*openflexure\_microscope.camera.pi.PiCameraStreamer attribute*), 7

`camera` (*openflexure\_microscope.microscope.Microscope attribute*), 5

`capture()` (*openflexure\_microscope.camera.pi.PiCameraStreamer method*), 9

`capture_from_exif()` (in module *openflexure\_microscope.captures.capture*), 11

`CaptureObject` (class in *openflexure\_microscope.captures.capture*), 11

`close()` (*openflexure\_microscope.camera.base.BaseCamera method*), 10

`close()` (*openflexure\_microscope.camera.pi.PiCameraStreamer method*), 8

`close()` (*openflexure\_microscope.microscope.Microscope method*), 5

`close()` (*openflexure\_microscope.stage.base.BaseStage method*), 15

`close()` (*openflexure\_microscope.stage.sanga.SangaStage method*), 14

`configuration` (*openflexure\_microscope.camera.base.BaseCamera attribute*), 10

`configuration` (*openflexure\_microscope.camera.pi.PiCameraStreamer attribute*), 8

`Configuration` (*openflexure\_microscope.stage.base.BaseStage attribute*), 14

`configuration` (*openflexure\_microscope.stage.sanga.SangaStage attribute*), 13

`create_file()` (in module *openflexure\_microscope.config*), 4

## D

`data` (*openflexure\_microscope.captures.capture.CaptureObject attribute*), 12

`default()` (*openflexure\_microscope.config.JSONEncoder method*), 4

`delete()` (*openflexure\_microscope.captures.capture.CaptureObject method*), 12

`delete_tag()` (*openflexure\_microscope.captures.capture.CaptureObject method*), 11

## E

`exists` (*openflexure\_microscope.captures.capture.CaptureObject attribute*), 11

## F

`force_get_metadata()` (*openflexure\_microscope.microscope.Microscope* method), 6

`fov` (*openflexure\_microscope.microscope.Microscope* attribute), 5

`frames()` (*openflexure\_microscope.camera.base.BaseCamera* method), 10

`frames()` (*openflexure\_microscope.camera.pi.PiCameraStreamer* method), 10

## G

`get_frame()` (*openflexure\_microscope.camera.base.BaseCamera* method), 10

`get_metadata()` (*openflexure\_microscope.microscope.Microscope* method), 6

## H

`has_real_camera()` (*openflexure\_microscope.microscope.Microscope* method), 5

`has_real_stage()` (*openflexure\_microscope.microscope.Microscope* method), 5

## I

`id` (*openflexure\_microscope.captures.capture.CaptureObject* attribute), 11

`image_resolution` (*openflexure\_microscope.camera.pi.PiCameraStreamer* attribute), 7

`initialisation()` (*openflexure\_microscope.camera.pi.PiCameraStreamer* method), 8

`initialise_file()` (in module *openflexure\_microscope.config*), 4

## J

`jpeg_quality` (*openflexure\_microscope.camera.pi.PiCameraStreamer* attribute), 8

`JSONEncoder` (class in *openflexure\_microscope.config*), 3

## L

`load()` (*openflexure\_microscope.config.OpenflexureSettingsFile* method), 3

`load_json_file()` (in module *openflexure\_microscope.config*), 4

`lock` (*openflexure\_microscope.microscope.Microscope* attribute), 5

`lock` (*openflexure\_microscope.stage.base.BaseStage* attribute), 14

## M

`merge()` (*openflexure\_microscope.config.OpenflexureSettingsFile* method), 3

`metadata` (*openflexure\_microscope.captures.capture.CaptureObject* attribute), 12

`Microscope` (class in *openflexure\_microscope.microscope*), 5

`mjpeg_quality` (*openflexure\_microscope.camera.pi.PiCameraStreamer* attribute), 8

`move_abs()` (*openflexure\_microscope.stage.base.BaseStage* method), 14

`move_abs()` (*openflexure\_microscope.stage.sanga.SangaStage* method), 14

`move_rel()` (*openflexure\_microscope.stage.base.BaseStage* method), 14

`move_rel()` (*openflexure\_microscope.stage.sanga.SangaStage* method), 14

## N

`n_axes` (*openflexure\_microscope.stage.base.BaseStage* attribute), 14

`n_axes` (*openflexure\_microscope.stage.sanga.SangaStage* attribute), 13

`numpy_resolution` (*openflexure\_microscope.camera.pi.PiCameraStreamer* attribute), 8

## O

`openflexure_microscope.camera.base` (module), 10

`openflexure_microscope.camera.pi` (module), 7

`openflexure_microscope.captures.capture` (module), 11

`openflexure_microscope.config` (module), 3

`openflexure_microscope.microscope` (module), 5

`openflexure_microscope.stage.base` (module), 14

`openflexure_microscope.stage.sanga` (module), 13

`OpenflexureSettingsFile` (class in *openflexure\_microscope.config*), 3

## P

`picamera_lst_path` (*openflex-*

<code>ure_microscope.camera.pi.PiCameraStreamer</code> (attribute), 8	<code>method</code> ), 11
<code>PiCameraStreamer</code> (class in <code>openflexure_microscope.camera.pi</code> ), 7	<code>save_settings()</code> ( <code>openflexure_microscope.microscope.Microscope</code> method), 6
<code>position</code> ( <code>openflexure_microscope.stage.base.BaseStage</code> attribute), 14	<code>scan_linear()</code> ( <code>openflexure_microscope.stage.base.BaseStage</code> method), 15
<code>position</code> ( <code>openflexure_microscope.stage.sanga.SangaStage</code> attribute), 13	<code>scan_z()</code> ( <code>openflexure_microscope.stage.base.BaseStage</code> method), 15
<code>pull_usercomment_dict()</code> (in module <code>openflexure_microscope.captures.capture</code> ), 11	<code>set_zoom()</code> ( <code>openflexure_microscope.camera.pi.PiCameraStreamer</code> method), 8
<code>put_and_save()</code> ( <code>openflexure_microscope.captures.capture.CaptureObject</code> method), 11	<code>setup()</code> ( <code>openflexure_microscope.microscope.Microscope</code> method), 5
<code>put_annotations()</code> ( <code>openflexure_microscope.captures.capture.CaptureObject</code> method), 11	<code>split_file_path()</code> ( <code>openflexure_microscope.captures.capture.CaptureObject</code> method), 11
<code>put_metadata()</code> ( <code>openflexure_microscope.captures.capture.CaptureObject</code> method), 11	<code>stage</code> ( <code>openflexure_microscope.microscope.Microscope</code> attribute), 5
<code>put_tags()</code> ( <code>openflexure_microscope.captures.capture.CaptureObject</code> method), 11	<code>start_preview()</code> ( <code>openflexure_microscope.camera.pi.PiCameraStreamer</code> method), 8
<b>R</b>	<code>start_recording()</code> ( <code>openflexure_microscope.camera.pi.PiCameraStreamer</code> method), 8
<code>read_settings()</code> ( <code>openflexure_microscope.camera.base.BaseCamera</code> method), 10	<code>start_stream_recording()</code> ( <code>openflexure_microscope.camera.pi.PiCameraStreamer</code> method), 9
<code>read_settings()</code> ( <code>openflexure_microscope.camera.pi.PiCameraStreamer</code> method), 8	<code>start_worker()</code> ( <code>openflexure_microscope.camera.base.BaseCamera</code> method), 10
<code>read_settings()</code> ( <code>openflexure_microscope.microscope.Microscope</code> method), 6	<code>state</code> ( <code>openflexure_microscope.camera.base.BaseCamera</code> attribute), 10
<code>read_settings()</code> ( <code>openflexure_microscope.stage.base.BaseStage</code> method), 14	<code>state</code> ( <code>openflexure_microscope.camera.pi.PiCameraStreamer</code> attribute), 8
<code>read_settings()</code> ( <code>openflexure_microscope.stage.sanga.SangaStage</code> method), 14	<code>state</code> ( <code>openflexure_microscope.captures.capture.CaptureObject</code> attribute), 12
<code>release_motors()</code> ( <code>openflexure_microscope.stage.sanga.SangaStage</code> method), 14	<code>state</code> ( <code>openflexure_microscope.microscope.Microscope</code> attribute), 5
<b>S</b>	<code>state</code> ( <code>openflexure_microscope.stage.base.BaseStage</code> attribute), 14
<code>SangaStage</code> (class in <code>openflexure_microscope.stage.sanga</code> ), 13	<code>state</code> ( <code>openflexure_microscope.stage.sanga.SangaStage</code> attribute), 13
<code>save()</code> ( <code>openflexure_microscope.captures.capture.CaptureObject</code> method), 12	<code>stop_preview()</code> ( <code>openflexure_microscope.camera.pi.PiCameraStreamer</code> method), 8
<code>save()</code> ( <code>openflexure_microscope.config.OpenflexureSettingsFile</code> method), 3	<code>stop_recording()</code> ( <code>openflexure_microscope.camera.pi.PiCameraStreamer</code> method), 9
<code>save_json_file()</code> (in module <code>openflexure_microscope.config</code> ), 4	<code>stop_stream_recording()</code> ( <code>openflexure_microscope.camera.pi.PiCameraStreamer</code> method), 9
<code>save_metadata()</code> ( <code>openflexure_microscope.captures.capture.CaptureObject</code> method), 11	<code>stop_worker()</code> ( <code>openflexure_microscope.camera.base.BaseCamera</code> method), 10

*method*), 10  
stream\_resolution (openflex-  
ure\_microscope.camera.pi.PiCameraStreamer  
attribute), 7

## T

thumbnail (openflex-  
ure\_microscope.captures.capture.CaptureObject  
attribute), 12

## U

update\_settings() (openflex-  
ure\_microscope.camera.base.BaseCamera  
method), 10  
update\_settings() (openflex-  
ure\_microscope.camera.pi.PiCameraStreamer  
method), 8  
update\_settings() (openflex-  
ure\_microscope.microscope.Microscope  
method), 6  
update\_settings() (openflex-  
ure\_microscope.stage.base.BaseStage method),  
14  
update\_settings() (openflex-  
ure\_microscope.stage.sanga.SangaStage  
method), 14  
user\_configuration (in module openflex-  
ure\_microscope.config), 4  
user\_settings (in module openflex-  
ure\_microscope.config), 4

## W

wait\_for\_camera() (openflex-  
ure\_microscope.camera.pi.PiCameraStreamer  
method), 10

## Y

yuv() (openflexure\_microscope.camera.pi.PiCameraStreamer  
method), 9

## Z

zero\_position() (openflex-  
ure\_microscope.stage.base.BaseStage method),  
15  
zero\_position() (openflex-  
ure\_microscope.stage.sanga.SangaStage  
method), 14