# Python-LabThings

**Joel Collins**

**Aug 23, 2021**

# CONTENTS:

CHAPTER

ONE

# CORE CONCEPTS

LabThings is rooted in the W3C Web of Things standards. Using IP networking in labs is not itself new, though perhaps under-used. However lack of proper standardisation has stiffled widespread adoption. LabThings, rather than try to introduce new competing standards, uses the architecture and terminology introduced by the W3C Web of Things. A full description of the core architecture can be found in the Web of Things (WoT) Architecture document. However, a brief outline of core terminology is given below.

## 1.1 Web Thing

A Web Thing is defined as "an abstraction of a physical or a virtual entity whose metadata and interfaces are described by a WoT Thing description." For LabThings this corresponds to an individual lab instrument that exposes functionality available to the user via a web API, and that functionality is described by a Thing Description.

LabThings automatically generates a Thing Description as you add functionality. The functionality you add falls into one of three categories of "interaction affordance": Properties, Actions, and Events.

## 1.2 Properties

A Property is defined as "an Interaction Affordance that exposes the state of the Thing. The state exposed by a Property MUST be retrievable (readable). Optionally, the state exposed by a Property MAY be updated (writeable)."

As a rule of thumb, any attribute of your device that can be quickly read, or optionally written, should be a Property. For example, simple device settings, or one-shot measurements such as temperature.

## 1.3 Actions

An Action is defined as "an Interaction Affordance that allows to invoke a function of the Thing. An Action MAY manipulate state that is not directly exposed (cf. Properties), manipulate multiple Properties at a time, or manipulate Properties based on internal logic (e.g., toggle). Invoking an Action MAY also trigger a process on the Thing that manipulates state (including physical state through actuators) over time."

The key point here is that Actions are typically more complex in functionality than simply setting or getting a property. For example, they can set multiple properties simultaneously (for example, auto-exposing a camera), or they can manipulate the state of the Thing over time, for example starting a long-running data acquisition.

Python-LabThings automatically handles offloading Actions into background threads where appropriate. The Action has a short timeout to complete quickly and respond to the API request, after which time a `201 - Started` response is sent to the client with information on how to check the state of the Action at a later time. This is particularly useful

for long-running data acquisitions, as it allows users to start an acquisition and get immediate confirmation that it has started, after which point they can poll for changes in status.

## 1.4 Events

An event "describes an event source that pushes data asynchronously from the Thing to the Consumer. Here not state, but state transitions (i.e., events) are communicated. Events MAY be triggered through conditions that are not exposed as Properties."

Common examples are notifying clients when a Property is changed, or when an Action starts or finishes. However, Thing developers can introduce new Events such as warnings, status messages, and logs. For example, a device may emit an events when the internal temperature gets too high, or when an interlock is tripped. This Event can then be pushed to both users AND other Things, allowing automtic response to external conditions.

A good example of this might be having Things automatically pause data-acquisition Actions upon detection of an overheat or interlock Event from another Thing.

CHAPTER

TWO

# QUICKSTART

The easiest way to get started with Python-LabThings is via the *labthings.create_app()* function, and the *labthings.LabThing* builder methods.

We will assume that for basic usage you already have some basic instrument control code. In our example, this is in the form of a `PretendSpectrometer` class, which will generate some data like your instrument control code might. Our `PretendSpectrometer` class has a `data` attribute which quickly returns a spectrum, an `x_range` attribute which determines the range of data we'll return, an `integration_time` attribute for cleaning up our signal, and a slow `average_data(n)` method to average `n` individual data measurements.

Building an API from this class requires a few extra considerations. In order to tell our API what data to expect from users, we need to construct a schema for each of our interactions. This schema simply maps variable names to JSON-compatible types, and is made simple via the *labthings.fields* module.

For properties, the input and output MUST be formatted the same, and so a single `schema` argument handles both. For actions, the input parameters and output response may be different. In this case, we can pass a `schema` argument to format the output, and an `args` argument to specify the input parameters,

An example Lab Thing built from our `PretendSpectrometer` class, complete with schemas, might look like:

```
import time

from labthings import ActionView, PropertyView, create_app, fields, find_component, op
from labthings.example_components import PretendSpectrometer
from labthings.json import encode_json

"""
Class for our lab component functionality. This could include serial communication,
equipment API calls, network requests, or a "virtual" device as seen here.
"""



"""
Create a view to view and change our integration_time value,
and register is as a Thing property
"""



# Wrap in a semantic annotation to autmatically set schema and args
class DenoiseProperty(PropertyView):
    """Value of integration_time"""

    schema = fields.Int(required=True, minimum=100, maximum=500)
```

(continues on next page)

(continued from previous page)

```
    semtype = "LevelProperty"

    @op.readproperty
    def get(self):
        # When a GET request is made, we'll find our attached component
        my_component = find_component("org.labthings.example.mycomponent")
        return my_component.integration_time

    @op.writeproperty
    def put(self, new_property_value):
        # Find our attached component
        my_component = find_component("org.labthings.example.mycomponent")

        # Apply the new value
        my_component.integration_time = new_property_value

        return my_component.integration_time


"""
Create a view to quickly get some noisy data, and register is as a Thing property
"""


class QuickDataProperty(PropertyView):
    """Show the current data value"""

    # Marshal the response as a list of floats
    schema = fields.List(fields.Float())

    @op.readproperty
    def get(self):
        # Find our attached component
        my_component = find_component("org.labthings.example.mycomponent")
        return my_component.data


"""
Create a view to start an averaged measurement, and register is as a Thing action
"""


class MeasurementAction(ActionView):
    # Expect JSON parameters in the request body.
    # Pass to post function as dictionary argument.
    args = {
        "averages": fields.Integer(
            missing=20, example=20, description="Number of data sets to average over",
        )
    }
    # Marshal the response as a list of numbers
    schema = fields.List(fields.Number)
```

(continues on next page)

(continued from previous page)

```
    # Main function to handle POST requests
    @op.invokeaction
    def post(self, args):
        """Start an averaged measurement"""

        # Find our attached component
        my_component = find_component("org.labthings.example.mycomponent")

        # Get arguments and start a background task
        n_averages = args.get("averages")

        # Return the task information
        return my_component.average_data(n_averages)


# Create LabThings Flask app
app, labthing = create_app(
    __name__,
    title="My Lab Device API",
    description="Test LabThing-based API",
    version="0.1.0",
)

# Attach an instance of our component
# Usually a Python object controlling some piece of hardware
my_spectrometer = PretendSpectrometer()
labthing.add_component(my_spectrometer, "org.labthings.example.mycomponent")


# Add routes for the API views we created
labthing.add_view(DenoiseProperty, "/integration_time")
labthing.add_view(QuickDataProperty, "/quick-data")
labthing.add_view(MeasurementAction, "/actions/measure")


# Start the app
if __name__ == "__main__":
    from labthings import Server

    Server(app).run()
```

Once started, the app will build and serve a full web API, and generate the following Thing Description:

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        "https://iot.mozilla.org/schemas/"
    ],
    "id": "http://127.0.0.1:7486/",
    "base": "http://127.0.0.1:7486/",
    "title": "My PretendSpectrometer API",
```

(continues on next page)

(continued from previous page)


```
    "description": "LabThing API for PretendSpectrometer",
    "properties": {
        "pretendSpectrometerData": {
            "title": "PretendSpectrometer_data",
            "description": "A single-shot measurement",
            "readOnly": true,
            "links": [{
                "href": "/properties/PretendSpectrometer/data"
            }],
            "forms": [{
                "op": "readproperty",
                "htv:methodName": "GET",
                "href": "/properties/PretendSpectrometer/data",
                "contentType": "application/json"
            }],
            "type": "array",
            "items": {
                "type": "number",
                "format": "decimal"
            }
        },
        "pretendSpectrometerMagicDenoise": {
            "title": "PretendSpectrometer_magic_denoise",
            "description": "Single-shot integration time",
            "links": [{
                "href": "/properties/PretendSpectrometer/magic_denoise"
            }],
            "forms": [{
                    "op": "readproperty",
                    "htv:methodName": "GET",
                    "href": "/properties/PretendSpectrometer/magic_denoise",
                    "contentType": "application/json"
                },
                {
                    "op": "writeproperty",
                    "htv:methodName": "PUT",
                    "href": "/properties/PretendSpectrometer/magic_denoise",
                    "contentType": "application/json"
                }
            ],
            "type": "number",
            "format": "integer",
            "min": 100,
            "max": 500,
            "example": 200
        }
    },
    "actions": {
        "averageDataAction": {
            "title": "average_data_action",
            "description": "Take an averaged measurement",
            "links": [{
```


(continues on next page)

(continued from previous page)

```
                "href": "/actions/PretendSpectrometer/average_data"
            }],
            "forms": [{
                "op": "invokeaction",
                "htv:methodName": "POST",
                "href": "/actions/PretendSpectrometer/average_data",
                "contentType": "application/json"
            }],
            "input": {
                "type": "object",
                "properties": {
                    "n": {
                        "type": "number",
                        "format": "integer",
                        "default": 5,
                        "description": "Number of averages to take",
                        "example": 5
                    }
                }
            }
        }
    },
    "links": [],
    "securityDefinitions": {},
    "security": "nosec_sc"
}
```

For completeness of the examples, our `PretendSpectrometer` class code is:

```
import random
import math
import time

class PretendSpectrometer:
    def __init__(self):
        self.x_range = range(-100, 100)
        self.integration_time = 200

    def make_spectrum(self, x, mu=0.0, sigma=25.0):
        """
        Generate a noisy gaussian function (to act as some pretend data)

        Our noise is inversely proportional to self.integration_time
        """
        x = float(x - mu) / sigma
        return (
            math.exp(-x * x / 2.0) / math.sqrt(2.0 * math.pi) / sigma
            + (1 / self.integration_time) * random.random()
        )

    @property
    def data(self):
```

(continues on next page)

(continued from previous page)

```
        """Return a 1D data trace."""
        time.sleep(self.integration_time / 1000)
        return [self.make_spectrum(x) for x in self.x_range]

    def average_data(self, n: int):
        """Average n-sets of data. Emulates a measurement that may take a while."""
        summed_data = self.data

        for _ in range(n):
            summed_data = [summed_data[i] + el for i, el in enumerate(self.data)]
            time.sleep(0.25)

        summed_data = [i / n for i in summed_data]

        return summed_data
```

CHAPTER

# THREE

# BASIC USAGE

## 3.1 App, LabThing, and Server

Python LabThings works as a Flask extension, and so we introduce two key objects: the `flask.Flask` app, and the *labthings.LabThing* object. The *labthings.LabThing* object is our main entrypoint for the Flask application, and all LabThings functionality is added via this object.

In order to enable threaded actions the app should be served using the *labthings.Server* class. Other production servers such as Gevent can be used, however this will require monkey-patching and has not been comprehensively tested.

### 3.1.1 Create app

The *labthings.create_app()* function automatically creates a Flask app object, enables up cross-origin resource sharing, and initialises a *labthings.LabThing* instance on the app. The function returns both in a tuple.

labthings.**create_app**(*import_name*, *prefix: str = ''*, *title: str = ''*, *description: str = ''*, *types: Optional[list] = None*, *version: str = '0.0.0'*, *external_links: bool = True*, *handle_errors: bool = True*, *handle_cors: bool = True*, *flask_kwargs: Optional[dict] = None*)

Quick-create a LabThings-enabled Flask app

**Parameters**

- **import_name** – Flask import name. Usually `__name__`.
- **prefix** (*str*) – URL prefix for all LabThings views. Defaults to "/api".
- **title** (*str*) – Title/name of the LabThings Thing.
- **description** (*str*) – Brief description of the LabThings Thing.
- **version** (*str*) – Version number/code of the Thing. Defaults to "0.0.0".
- **handle_errors** (*bool*) – Use the LabThings error handler, to JSON format internal exceptions. Defaults to True.
- **handle_cors** (*bool*) – Automatically enable CORS on all LabThings views. Defaults to True.
- **flask_kwargs** (*dict*) – Keyword arguments to pass to the Flask instance.
- **prefix** – str: (Default value = "")
- **title** – str: (Default value = "")
- **description** – str: (Default value = "")
- **types** – list: (Default value = None)

- **version** – str: (Default value = "0.0.0")
- **external_links** – bool: Use external links in Thing Description where possible
- **handle_errors** – bool: (Default value = True)
- **handle_cors** – bool: (Default value = True)
- **flask_kwargs** – dict: (Default value = None)

**Returns** (Flask app object, LabThings object)

ALternatively, the app and *labthings.LabThing* objects can be initialised and attached separately, for example:

```
from flask import Flask
from labthings import LabThing

app = Flask(__name__)
labthing = LabThing(app)
```

### 3.1.2 LabThing

The LabThing object is our main entrypoint, and handles creating API views, managing background actions, tracking logs, and generating API documentation.

**class** labthings.**LabThing**(*app: Optional[flask.app.Flask] = None*, *id_: Optional[str] = None*, *prefix: str = ''*, *title: str = ''*, *description: str = ''*, *version: str = '0.0.0'*, *types: Optional[List[str]] = None*, *format_flask_exceptions: bool = True*, *external_links: bool = True*, *json_encoder=<class 'labthings.json.encoder.LabThingsJSONEncoder'>*)

The main entry point for the application. You need to initialize it with a Flask Application:

```
>>> app = Flask(__name__)
>>> labthing = labthings.LabThing(app)
```

Alternatively, you can use *init_app()* to set the Flask application after it has been constructed.

**Parameters**

- **app** (*flask.Flask*) – the Flask application object
- **prefix** (*str*) – Prefix all routes with a value, eg v1 or 2010-04-01
- **title** (*str*) – Human-readable title of the Thing
- **description** (*str*) – Human-readable description of the Thing
- **version** (*str*) – Version number of the Thing
- **types** (*list of str*) – List of Thing types, used by clients to filter discovered Things
- **format_flask_exceptions** (*bool*) – JSON format all exception responses
- **external_links** (*bool*) – Use external links in Thing Description where possible
- **json_encoder** – JSON encoder class for the app

### 3.1.3 Views

Thing interaction affordances are created using Views. Two main View types correspond to properties and actions.

**class** labthings.**PropertyView**(**args*, ***kwargs*)

**class** labthings.**ActionView**(**args*, ***kwargs*)

### 3.1.4 Server

The integrated server actually handles 3 distinct server functions: WSGI HTTP requests, and registering mDNS records for automatic Thing discovery. It is therefore strongly suggested you use the builtin server.

**Important notes:**

The integrated server will spawn a new native thread *per-connection*. This will only function well in situations where few (<50) simultaneous connections are expected, such as local Web of Things devices. Do not use this server in any public web app where many connections are expected. It is designed exclusively with low-traffic LAN access in mind.

**class** labthings.**Server**(*app*, *host='0.0.0.0'*, *port=7485*, *debug=False*, *zeroconf=True*)

Combined WSGI+mDNS server.

**Parameters**

- **host** (*string*) – Host IP address. Defaults to 0.0.0.0.
- **port** (*int*) – Host port. Defaults to 7485.
- **debug** (*bool*) – Enable server debug mode. Defaults to False.
- **zeroconf** (*bool*) – Enable the zeroconf (mDNS) server. Defaults to True.

**run**(*host=None*, *port=None*, *debug=None*, *zeroconf=None*)

Starts the server allowing for runtime parameters. Designed to immitate the old Flask app.run style of starting an app

**Parameters**

- **host** (*string*) – Host IP address. Defaults to 0.0.0.0.
- **port** (*int*) – Host port. Defaults to 7485.
- **debug** (*bool*) – Enable server debug mode. Defaults to False.
- **zeroconf** (*bool*) – Enable the zeroconf (mDNS) server. Defaults to True.

**start**()

Start the server and register mDNS records

## 3.2 HTTP API Structure

*Documentation to be written*

## 3.3 Marshalling and Serialising views

### 3.3.1 Introduction

LabThings 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.

Marshalling schemas are used by LabThings to document the data types of properties, as well as the structure and types of Action arguments and return values. They allow arbitrary Python objects to be returned as serialized JSON, and ensure that input arguments are properly formated before being passed to your Python functions.

From our quickstart example, we use schemas for our *integration_time* property to inform LabThings that both responses *and* requests to the API should be integer formatted. Additional information about range, example values, and units can be added to the schema field.

```
labthing.build_property(
    my_spectrometer,  # Python object
    "integration_time",  # Objects attribute name
    description="Single-shot integration time",
    schema=fields.Int(min=100, max=500, example=200, unit="microsecond")
)
```

Actions require separate schemas for input and output, since the action return data is likely in a different format to the input arguments. In our quickstart example, our *schema* argument informs LabThings that the action return value should be a list of numbers. Meanwhile, our *args* argument informs LabThings that requests to start the action should include an attribute called *n*, which should be an integer. Human-readable descriptions, examples, and default values can be added to the args field.

```
labthing.build_action(
    my_spectrometer,  # Python object
    "average_data",  # Objects method name
    description="Take an averaged measurement",
    schema=fields.List(fields.Number()),
    args={  # How do we convert from the request input to function arguments?
        "n": fields.Int(description="Number of averages to take", example=5, default=5)
    },
)
```

### 3.3.2 Schemas

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 *labthings.Schema* class, or by passing a dictionary of key-field pairs.

Note that the *labthings.Schema* class is an alias of marshmallow.Schema, and the two can be used interchangeably.

Schemas are required for argument parsing. While Property views can be marshalled with a single field, arguments must be passed to the server as a JSON object, which gets mapped to a Python dictionary and passed to the Action method.

For example, a Python function of the form:

```
from typing import List

def my_function(quantity: int, name: str, organizations: List(str)):
    return quantity * len(organizations)
```

would require *args* of the form:

```
args = {
    "quantity": fields.Int()
    "name": fields.String()
    "organisation": fields.List(fields.String())
}
```

and a *schema* of *labthings.fields.Int*.

### 3.3.3 Fields

Most data types are represented by fields in the Marshmallow library. All Marshmallow fields are imported and available from the *labthings.fields* submodule, however any field can be imported from Marshmallow and used in LabThings schemas.

**class** labthings.fields.**AwareDateTime**(*format: Optional[str] = None*, ***, *default_timezone: Optional[datetime.tzinfo] = None*, ***kwargs*)

A formatted aware datetime string.

**Parameters**

- **format** – See *DateTime*.
- **default_timezone** – Used on deserialization. If *None*, naive datetimes are rejected. If not *None*, naive datetimes are set this timezone.
- **kwargs** – The same keyword arguments that *Field* receives.

New in version 3.0.0rc9.

**AWARENESS = 'aware'**

labthings.fields.**Bool**

alias of *marshmallow.fields.Boolean*

**class** labthings.fields.**Boolean**(***, *truthy: Optional[Set] = None*, *falsy: Optional[Set] = None*, ***kwargs*)

A boolean field.

**Parameters**

- **truthy** – Values that will (de)serialize to *True*. If an empty set, any non-falsy value will deserialize to *True*. If *None*, *marshmallow.fields.Boolean.truthy* will be used.
- **falsy** – Values that will (de)serialize to *False*. If *None*, *marshmallow.fields.Boolean.falsy* will be used.
- **kwargs** – The same keyword arguments that *Field* receives.

**default_error_messages = {'invalid': 'Not a valid boolean.'}**
Default error messages.

**falsy = {'false', 0, 'no', 'False', 'F', 'Off', 'No', 'FALSE', 'off', 'OFF', 'N', 'f', 'NO', 'n', '0'}**
Default falsy values.

**truthy = {'Yes', 1, 'True', '1', 'true', 'Y', 'On', 't', 'ON', 'y', 'yes', 'T', 'YES', 'TRUE', 'on'}**
Default truthy values.

**class** labthings.fields.**Bytes**(**, load_default: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Optional[Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]]]] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)
Marshmallow field for *bytes* objects

**class** labthings.fields.**Constant**(*constant: Any*, ***kwargs*)
A field that (de)serializes to a preset constant. If you only want the constant added for serialization or deserialization, you should use `dump_only=True` or `load_only=True` respectively.

**Parameters** **constant** – The constant to return for the field attribute.

New in version 2.0.0.

**class** labthings.fields.**Date**(*format: Optional[str] = None*, ***kwargs*)
ISO8601-formatted date string.

**Parameters**

- **format** – Either `"iso"` (for ISO8601) or a date format string. If *None*, defaults to "iso".
- **kwargs** – The same keyword arguments that *Field* receives.

**DEFAULT_FORMAT = 'iso'**

**DESERIALIZATION_FUNCS: *Dict*[str, Callable[[str], Any]] = {'iso': <function from_iso_date>, 'iso8601': <function from_iso_date>}**

**OBJ_TYPE = 'date'**

**SCHEMA_OPTS_VAR_NAME = 'dateformat'**

**SERIALIZATION_FUNCS: *Dict*[str, Callable[[Any], str]] = {'iso': <function to_iso_date>, 'iso8601': <function to_iso_date>}**

**default_error_messages = {'format': '"{input}" cannot be formatted as a date.', 'invalid': 'Not a valid date.'}**
Default error messages.

**class** labthings.fields.**DateTime**(*format: Optional[str] = None*, ***kwargs*)
A formatted datetime string.

Example: `'2014-12-22T03:12:58.019077+00:00'`

**Parameters**

- **format** – Either `"rfc"` (for RFC822), `"iso"` (for ISO8601), or a date format string. If *None*, defaults to "iso".
- **kwargs** – The same keyword arguments that *Field* receives.

Changed in version 3.0.0rc9: Does not modify timezone information on (de)serialization.

**DEFAULT_FORMAT = 'iso'**

**DESERIALIZATION_FUNCS: *Dict*[str, Callable[[str], Any]] = {'iso': <function from_iso_datetime>, 'iso8601': <function from_iso_datetime>, 'rfc': <function from_rfc>, 'rfc822': <function from_rfc>}**

**OBJ_TYPE = 'datetime'**

**SCHEMA_OPTS_VAR_NAME = 'datetimeformat'**

**SERIALIZATION_FUNCS: *Dict*[str, Callable[[Any], str]] = {'iso': <function isoformat>, 'iso8601': <function isoformat>, 'rfc': <function rfcformat>, 'rfc822': <function rfcformat>}**

**default_error_messages = {'format': '"{input}" cannot be formatted as a {obj_type}.', 'invalid': 'Not a valid {obj_type}.', 'invalid_awareness': 'Not a valid {awareness} {obj_type}.'}**

Default error messages.

**class** labthings.fields.**Decimal**(*places: Optional[int] = None*, *rounding: Optional[str] = None*, ***, *allow_nan: bool = False*, *as_string: bool = False*, ***kwargs*)

A field that (de)serializes to the Python `decimal.Decimal` type. It's safe to use when dealing with money values, percentages, ratios or other numbers where precision is critical.

**Warning:** This field serializes to a *decimal.Decimal* object by default. If you need to render your data as JSON, keep in mind that the *json* module from the standard library does not encode *decimal.Decimal*. Therefore, you must use a JSON library that can handle decimals, such as *simplejson*, or serialize to a string by passing `as_string=True`.

**Warning:** If a JSON *float* value is passed to this field for deserialization it will first be cast to its corresponding *string* value before being deserialized to a *decimal.Decimal* object. The default *__str__* implementation of the built-in Python *float* type may apply a destructive transformation upon its input data and therefore cannot be relied upon to preserve precision. To avoid this, you can instead pass a JSON *string* to be deserialized directly.

**Parameters**

- **places** – How many decimal places to quantize the value. If *None*, does not quantize the value.
- **rounding** – How to round the value during quantize, for example *decimal.ROUND_UP*. If *None*, uses the rounding value from the current thread's context.
- **allow_nan** – If *True*, *NaN*, *Infinity* and *-Infinity* are allowed, even though they are illegal according to the JSON specification.
- **as_string** – If *True*, serialize to a string instead of a Python *decimal.Decimal* type.

- **kwargs** – The same keyword arguments that *Number* receives.

New in version 1.2.0.

**default_error_messages = {'special': 'Special numeric values (nan or infinity) are not permitted.'}**
Default error messages.

**num_type**
alias of `decimal.Decimal`

**class** labthings.fields.**Dict**(*keys: Optional[Union[*marshmallow.fields.Field*, type]] = None*, *values: Optional[Union[*marshmallow.fields.Field*, type]] = None*, ***kwargs*)
A dict field. Supports dicts and dict-like objects. Extends Mapping with dict as the mapping_type.

Example:

```
numbers = fields.Dict(keys=fields.Str(), values=fields.Float())
```

**Parameters** **kwargs** – The same keyword arguments that *Mapping* receives.

New in version 2.1.0.

**mapping_type**
alias of `dict`

**class** labthings.fields.**Email**(**args*, ***kwargs*)
An email field.

**Parameters**

- **args** – The same positional arguments that *String* receives.
- **kwargs** – The same keyword arguments that *String* receives.

**default_error_messages = {'invalid': 'Not a valid email address.'}**
Default error messages.

**class** labthings.fields.**Field**(**, load_default: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Optional[Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]]]] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)
Basic field from which other fields should extend. It applies no formatting by default, and should only be used in cases where data does not need to be formatted before being serialized or deserialized. On error, the name of the field will be returned.

**Parameters**

- **dump_default** – If set, this value will be used during serialization if the input value is missing. If not set, the field will be excluded from the serialized output if the input value is missing. May be a value or a callable.
- **load_default** – Default deserialization value for the field if the field is not found in the input data. May be a value or a callable.
- **data_key** – The name of the dict key in the external representation, i.e. the input of *load* and the output of *dump*. If *None*, the key will match the name of the field.

- **attribute** – The name of the attribute to get the value from when serializing. If *None*, assumes the attribute has the same name as the field. Note: This should only be used for very specific use cases such as outputting multiple fields for a single attribute. In most cases, you should use `data_key` instead.
- **validate** – Validator or collection of validators that are called during deserialization. Validator takes a field's input value as its only parameter and returns a boolean. If it returns *False*, an `ValidationError` is raised.
- **required** – Raise a `ValidationError` if the field value is not supplied during deserialization.
- **allow_none** – Set this to *True* if *None* should be considered a valid value during validation/deserialization. If `missing=None` and `allow_none` is unset, will default to `True`. Otherwise, the default is `False`.
- **load_only** – If *True* skip this field during serialization, otherwise its value will be present in the serialized data.
- **dump_only** – If *True* skip this field during deserialization, otherwise its value will be present in the deserialized object. In the context of an HTTP API, this effectively marks the field as "read-only".
- **error_messages** (*dict*) – Overrides for *Field.default_error_messages*.
- **metadata** – Extra information to be stored as field metadata.

Changed in version 2.0.0: Removed *error* parameter. Use `error_messages` instead.

Changed in version 2.0.0: Added *allow_none* parameter, which makes validation/deserialization of *None* consistent across fields.

Changed in version 2.0.0: Added *load_only* and *dump_only* parameters, which allow field skipping during the (de)serialization process.

Changed in version 2.0.0: Added *missing* parameter, which indicates the value for a field if the field is not found during deserialization.

Changed in version 2.0.0: `default` value is only used if explicitly set. Otherwise, missing values inputs are excluded from serialized output.

Changed in version 3.0.0b8: Add `data_key` parameter for the specifying the key in the input and output data. This parameter replaced both `load_from` and `dump_to`.

**property context**
The context dictionary for the parent `Schema`.

**property default**

**default_error_messages = {'null': 'Field may not be null.', 'required': 'Missing data for required field.', 'validator_failed': 'Invalid value.'}**
Default error messages for various kinds of errors. The keys in this dictionary are passed to *Field.make_error*. The values are error messages passed to `marshmallow.exceptions.ValidationError`.

**deserialize**(*value: Any*, *attr: Optional[str] = None*, *data: Optional[*Mapping*[str, Any]] = None*, ***kwargs*)
Deserialize `value`.

**Parameters**

- **value** – The value to deserialize.
- **attr** – The attribute/key in *data* to deserialize.

- **data** – The raw input data passed to *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

**Raises** **ValidationError** – If an invalid value is passed or if a required value is missing.

**fail**(*key: str*, ***kwargs*)
Helper method that raises a *ValidationError* with an error message from `self.error_messages`.

Deprecated since version 3.0.0: Use *make_error <marshmallow.fields.Field.make_error>* instead.

**get_value**(*obj*, *attr*, *accessor=None*, *default=<marshmallow.missing>*)
Return the value for a given key from an object.

**Parameters**

- **obj** (*object*) – The object to get the value from.
- **attr** (*str*) – The attribute/key in *obj* to get the value from.
- **accessor** (*callable*) – A callable used to retrieve the value of *attr* from the object *obj*. Defaults to *marshmallow.utils.get_value*.

**make_error**(*key: str*, ***kwargs*) → marshmallow.exceptions.ValidationError
Helper method to make a *ValidationError* with an error message from `self.error_messages`.

**property missing**

**serialize**(*attr: str*, *obj: Any*, *accessor: Optional[Callable[[Any, str, Any], Any]] = None*, ***kwargs*)
Pulls the value for the given key from the object, applies the field's formatting and returns the result.

**Parameters**

- **attr** – The attribute/key to get from the object.
- **obj** – The object to access the attribute/key from.
- **accessor** – Function used to access values from `obj`.
- **kwargs** – Field-specific keyword arguments.

**class** labthings.fields.**Float**(*, *allow_nan: bool = False*, *as_string: bool = False*, ***kwargs*)
A double as an IEEE-754 double precision string.

**Parameters**

- **allow_nan** (*bool*) – If *True*, *NaN*, *Infinity* and *-Infinity* are allowed, even though they are illegal according to the JSON specification.
- **as_string** (*bool*) – If *True*, format the value as a string.
- **kwargs** – The same keyword arguments that *Number* receives.

**default_error_messages = {'special': 'Special numeric values (nan or infinity) are not permitted.'}**
Default error messages.

**num_type**
alias of `float`

**class** labthings.fields.**Function**(*serialize: Optional[Union[Callable[[Any], Any], Callable[[Any, Dict], Any]]] = None*, *deserialize: Optional[Union[Callable[[Any], Any], Callable[[Any, Dict], Any]]] = None*, ***kwargs*)
A field that takes the value returned by a function.

**Parameters**

- **serialize** – A callable from which to retrieve the value. The function must take a single argument `obj` which is the object to be serialized. It can also optionally take a `context` argument, which is a dictionary of context variables passed to the serializer. If no callable is provided then the `load_only` flag will be set to True.
- **deserialize** – A callable from which to retrieve the value. The function must take a single argument `value` which is the value to be deserialized. It can also optionally take a `context` argument, which is a dictionary of context variables passed to the deserializer. If no callable is provided then `value` will be passed through unchanged.

Changed in version 2.3.0: Deprecated `func` parameter in favor of `serialize`.

Changed in version 3.0.0a1: Removed `func` parameter.

labthings.fields.**Int**
alias of *marshmallow.fields.Integer*

**class** labthings.fields.**Integer**(*, *strict: bool = False*, ***kwargs*)
An integer field.

**Parameters**

- **strict** – If *True*, only integer types are valid. Otherwise, any value castable to *int* is valid.
- **kwargs** – The same keyword arguments that *Number* receives.

**default_error_messages = {'invalid': 'Not a valid integer.'}**
Default error messages.

**num_type**
alias of `int`

**class** labthings.fields.**List**(*cls_or_instance: Union[*marshmallow.fields.Field*, type]*, ***kwargs*)
A list field, composed with another *Field* class or instance.

Example:

```
numbers = fields.List(fields.Float())
```

**Parameters**

- **cls_or_instance** – A field class or instance.
- **kwargs** – The same keyword arguments that *Field* receives.

Changed in version 2.0.0: The `allow_none` parameter now applies to deserialization and has the same semantics as the other fields.

Changed in version 3.0.0rc9: Does not serialize scalar values to single-item lists.

**default_error_messages = {'invalid': 'Not a valid list.'}**
Default error messages.

**class** labthings.fields.**Mapping**(*keys: Optional[Union[*marshmallow.fields.Field*, type]] = None*, *values: Optional[Union[*marshmallow.fields.Field*, type]] = None*, ***kwargs*)
An abstract class for objects with key-value pairs.

**Parameters**

- **keys** – A field class or instance for dict keys.
- **values** – A field class or instance for dict values.
- **kwargs** – The same keyword arguments that *Field* receives.

**Note:** When the structure of nested data is not known, you may omit the *keys* and *values* arguments to prevent content validation.

New in version 3.0.0rc4.

**default_error_messages = {'invalid': 'Not a valid mapping type.'}**
Default error messages.

**mapping_type**
alias of `dict`

**class** labthings.fields.**Method**(*serialize: Optional[str] = None*, *deserialize: Optional[str] = None*, ***kwargs*)
A field that takes the value returned by a *Schema* method.

**Parameters**

- **serialize** (*str*) – The name of the Schema method from which to retrieve the value. The method must take an argument `obj` (in addition to self) that is the object to be serialized.
- **deserialize** (*str*) – Optional name of the Schema method for deserializing a value The method must take a single argument `value`, which is the value to deserialize.

Changed in version 2.0.0: Removed optional `context` parameter on methods. Use `self.context` instead.

Changed in version 2.3.0: Deprecated `method_name` parameter in favor of `serialize` and allow `serialize` to not be passed at all.

Changed in version 3.0.0: Removed `method_name` parameter.

**class** labthings.fields.**NaiveDateTime**(*format: Optional[str] = None*, ***, *timezone: Optional[datetime.timezone] = None*, ***kwargs*)
A formatted naive datetime string.

**Parameters**

- **format** – See *DateTime*.
- **timezone** – Used on deserialization. If *None*, aware datetimes are rejected. If not *None*, aware datetimes are converted to this timezone before their timezone information is removed.
- **kwargs** – The same keyword arguments that *Field* receives.

New in version 3.0.0rc9.

**AWARENESS = 'naive'**

**class** labthings.fields.**Nested**(*nested: Union[marshmallow.base.SchemaABC, type, str, Callable[[], marshmallow.base.SchemaABC]]*, ***, *dump_default: Any = <marshmallow.missing>*, *default: Any = <marshmallow.missing>*, *only: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *unknown: Optional[str] = None*, ***kwargs*)
Allows you to nest a Schema inside a field.

Examples:

```
class ChildSchema(Schema):
    id = fields.Str()
    name = fields.Str()
    # Use lambda functions when you need two-way nesting or self-nesting
```

(continues on next page)

(continued from previous page)

```
    parent = fields.Nested(lambda: ParentSchema(only=("id",)), dump_only=True)
    siblings = fields.List(fields.Nested(lambda: ChildSchema(only=("id", "name"))))

class ParentSchema(Schema):
    id = fields.Str()
    children = fields.List(
        fields.Nested(ChildSchema(only=("id", "parent", "siblings")))
    )
    spouse = fields.Nested(lambda: ParentSchema(only=("id",)))
```

When passing a *Schema <marshmallow.Schema>* instance as the first argument, the instance's `exclude`, `only`, and `many` attributes will be respected.

Therefore, when passing the `exclude`, `only`, or `many` arguments to *fields.Nested*, you should pass a *Schema <marshmallow.Schema>* class (not an instance) as the first argument.

```
# Yes
author = fields.Nested(UserSchema, only=('id', 'name'))

# No
author = fields.Nested(UserSchema(), only=('id', 'name'))
```

**Parameters**

- **nested** – *Schema* instance, class, class name (string), or callable that returns a *Schema* instance.
- **exclude** – A list or tuple of fields to exclude.
- **only** – A list or tuple of fields to marshal. If *None*, all fields are marshalled. This parameter takes precedence over `exclude`.
- **many** – Whether the field is a collection of objects.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **kwargs** – The same keyword arguments that *Field* receives.

**default_error_messages = {'type': 'Invalid type.'}**
Default error messages.

**property schema**
The nested Schema object.

Changed in version 1.0.0: Renamed from *serializer* to *schema*.

**class** labthings.fields.**Number**(*, *as_string: bool = False*, ***kwargs*)
Base class for number fields.

**Parameters**

- **as_string** (*bool*) – If *True*, format the serialized value as a string.
- **kwargs** – The same keyword arguments that *Field* receives.

**default_error_messages = {'invalid': 'Not a valid number.', 'too_large': 'Number too large.'}**
Default error messages.

**num_type**
alias of `float`

**class** labthings.fields.**Pluck**(*nested: Union[marshmallow.base.SchemaABC, type, str, Callable[[], marshmallow.base.SchemaABC]]*, *field_name: str*, ***kwargs*)
Allows you to replace nested data with one of the data's fields.

Example:

```
from marshmallow import Schema, fields

class ArtistSchema(Schema):
    id = fields.Int()
    name = fields.Str()

class AlbumSchema(Schema):
    artist = fields.Pluck(ArtistSchema, 'id')


in_data = {'artist': 42}
loaded = AlbumSchema().load(in_data)  # => {'artist': {'id': 42}}
dumped = AlbumSchema().dump(loaded)  # => {'artist': 42}
```

**Parameters**

- **nested** (Schema) – The Schema class or class name (string) to nest, or `"self"` to nest the Schema within itself.
- **field_name** (*str*) – The key to pluck a value from.
- **kwargs** – The same keyword arguments that *Nested* receives.

**class** labthings.fields.**Raw**(**, load_default: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Optional[Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]]]] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)
Field that applies no formatting.

labthings.fields.**Str**
alias of *marshmallow.fields.String*

**class** labthings.fields.**String**(**, load_default: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Optional[Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]]]] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)
A string field.

**Parameters** **kwargs** – The same keyword arguments that *Field* receives.

**default_error_messages = {'invalid': 'Not a valid string.', 'invalid_utf8': 'Not a valid utf-8 string.'}**
Default error messages.

**class** labthings.fields.**Time**(*format: Optional[str] = None*, ***kwargs*)
A formatted time string.

Example: `'03:12:58.019077'`

**Parameters**

- **format** – Either "iso" (for ISO8601) or a date format string. If *None*, defaults to "iso".
- **kwargs** – The same keyword arguments that *Field* receives.

**DEFAULT_FORMAT = 'iso'**

**DESERIALIZATION_FUNCS: *Dict*[str, Callable[[str], Any]] = {'iso': <function from_iso_time>, 'iso8601': <function from_iso_time>}**

**OBJ_TYPE = 'time'**

**SCHEMA_OPTS_VAR_NAME = 'timeformat'**

**SERIALIZATION_FUNCS: *Dict*[str, Callable[[Any], str]] = {'iso': <function to_iso_time>, 'iso8601': <function to_iso_time>}**

**class** labthings.fields.**TimeDelta**(*precision: str = 'seconds'*, ***kwargs*)
A field that (de)serializes a `datetime.timedelta` object to an integer and vice versa. The integer can represent the number of days, seconds or microseconds.

**Parameters**

- **precision** – Influences how the integer is interpreted during (de)serialization. Must be 'days', 'seconds', 'microseconds', 'milliseconds', 'minutes', 'hours' or 'weeks'.
- **kwargs** – The same keyword arguments that *Field* receives.

Changed in version 2.0.0: Always serializes to an integer value to avoid rounding errors. Add *precision* parameter.

**DAYS = 'days'**

**HOURS = 'hours'**

**MICROSECONDS = 'microseconds'**

**MILLISECONDS = 'milliseconds'**

**MINUTES = 'minutes'**

**SECONDS = 'seconds'**

**WEEKS = 'weeks'**

**default_error_messages = {'format': '{input!r} cannot be formatted as a timedelta.', 'invalid': 'Not a valid period of time.'}**
Default error messages.

**class** labthings.fields.**Tuple**(*tuple_fields*, **args*, ***kwargs*)
A tuple field, composed of a fixed number of other *Field* classes or instances

Example:

```
row = Tuple((fields.String(), fields.Integer(), fields.Float()))
```

**Note:** Because of the structured nature of *collections.namedtuple* and *typing.NamedTuple*, using a Schema within a Nested field for them is more appropriate than using a *Tuple* field.

**Parameters**

- **tuple_fields** (*Iterable[Field]*) – An iterable of field classes or instances.
- **kwargs** – The same keyword arguments that *Field* receives.

New in version 3.0.0rc4.

**default_error_messages = {'invalid': 'Not a valid tuple.'}**
Default error messages.

labthings.fields.**URL**
alias of *marshmallow.fields.Url*

**class** labthings.fields.**UUID**(**, load_default: Any = <marshmallow.missing>, missing: Any = <marshmallow.missing>, dump_default: Any = <marshmallow.missing>, default: Any = <marshmallow.missing>, data_key: Optional[str] = None, attribute: Optional[str] = None, validate: Optional[Union[Callable[[Any], Any], Iterable[Callable[[Any], Any]]]] = None, required: bool = False, allow_none: Optional[bool] = None, load_only: bool = False, dump_only: bool = False, error_messages: Optional[Dict[str, str]] = None, metadata: Optional[Mapping[str, Any]] = None, **additional_metadata*)

A UUID field.

**default_error_messages = {'invalid_uuid': 'Not a valid UUID.'}**
Default error messages.

**class** labthings.fields.**Url**(**, relative: bool = False, schemes: Optional[Union[Sequence[str], Set[str]]] = None, require_tld: bool = True, **kwargs*)

An URL field.

**Parameters**

- **default** – Default value for the field if the attribute is not set.
- **relative** – Whether to allow relative URLs.
- **require_tld** – Whether to reject non-FQDN hostnames.
- **schemes** – Valid schemes. By default, `http`, `https`, `ftp`, and `ftps` are allowed.
- **kwargs** – The same keyword arguments that *String* receives.

**default_error_messages = {'invalid': 'Not a valid URL.'}**
Default error messages.

## 3.4 Action threads

Many actions in your LabThing 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 action, 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.

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 terminating 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.

**class** labthings.actions.**ActionThread**(*action: str*, *target: Optional[Callable] = None*, *name: Optional[str] = None*, *args: Optional[Iterable[Any]] = None*, *kwargs: Optional[Dict[str, Any]] = None*, *daemon: bool = True*, *default_stop_timeout: int = 5*, *log_len: int = 100*, *http_error_lock: Optional[_thread.allocate_lock] = None*)

A native thread with extra functionality for tracking progress and thread termination.

Arguments: * *action* is the name of the action that's running * *target*, *name*, *args*, *kwargs* and *daemon* are passed to *threading.Thread*

(though the defualt for *daemon* is changed to *True*)

- *default_stop_timeout* specifies how long we wait for the *target* function to stop nicely (e.g. by checking the *stopping* Event )
- *log_len* gives the number of log entries before we start dumping them
- *http_error_lock* allows the calling thread to handle some errors initially. See below.

## Error propagation If the *target* function throws an Exception, by default this will result in: * The thread terminating * The Action's status being set to *error* * The exception appearing in the logs with a traceback * The exception being raised in the background thread. However, *HTTPException* subclasses are used in Flask/Werkzeug web apps to return HTTP status codes indicating specific errors, and so merit being handled differently.

Normally, when an Action is initiated, the thread handling the HTTP request does not return immediately - it waits for a short period to check whether the Action has completed or returned an error. If an HTTPError is raised in the Action thread before the initiating thread has sent an HTTP response, we **don't** want to propagate the error here, but instead want to re-raise it in the calling thread. This will then mean that the HTTP request is

answered with the appropriate error code, rather than returning a *201* code, along with a description of the task (showing that it was successfully started, but also showing that it subsequently failed with an error).

In order to activate this behaviour, we must pass in a *threading.Lock* object. This lock should already be acquired by the request-handling thread. If an error occurs, and this lock is acquired, the exception should not be re-raised until the calling thread has had the chance to deal with it.

**property cancelled: bool**
Alias of *stopped*

**property dead: bool**
Has the thread finished, by any means (return, exception, termination).

**property exception: Optional[Exception]**
The Exception that caused the action to fail.

**get**(*block: bool = True*, *timeout: Optional[int] = None*)
Start waiting for the task to finish before returning

**Parameters**

- **block** – (Default value = True)
- **timeout** – (Default value = None)

**property id: uuid.UUID**
UUID for the thread. Note this not the same as the native thread ident.

**property output: Any**
Return value of the Action function. If the Action is still running, returns None.

**run**()
Overrides default threading.Thread run() method

**property status: str**
Current running status of the thread.

| Status | Meaning |
|---|---|
| `pending` | Not yet started |
| `running` | Currently in-progress |
| `completed` | Finished without error |
| `cancelled` | Thread stopped after a cancel request |
| `error` | Exception occured in thread |

**stop**(*timeout=None*, *exception=<class 'labthings.actions.thread.ActionKilledException'>*) → bool
Sets the threads internal stopped event, waits for timeout seconds for the thread to stop nicely, then forcefully kills the thread.

**Parameters**

- **timeout** (*int*) – Time to wait before killing thread forecefully. Defaults to `self.default_stop_timeout`
- **exception** – (Default value = ActionKilledException)

**property stopped: bool**
Has the thread been cancelled

**terminate**(*exception=<class 'labthings.actions.thread.ActionKilledException'>*) → bool

**Parameters** **exception** – (Default value = ActionKilledException)

**Raises** **which** – should cause the thread to exit silently

**update_data**(*data: Dict[Any, Any]*)

**Parameters** **data** – dict:

**update_progress**(*progress: int*)
Update the progress of the ActionThread.

**Parameters** **progress** – int: Action progress, in percent (0-100)

### 3.4.1 Accessing the current action thread

A function running inside a `labthings.actions.ActionThread` is able to access the instance it is running in using the *labthings.current_action()* function. This allows the state of the Action to be modified freely.

labthings.**current_action**()
Return the ActionThread instance in which the caller is currently running.

If this function is called from outside an ActionThread, it will return None.

**Returns** `labthings.actions.ActionThread` – Currently running ActionThread.

### 3.4.2 Updating action progress

Some client applications may be able to display progress bars showing the progress of an action. Implementing progress updates in your actions 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 `labthings.actions.ActionThread`, this function will update the state of the corresponding `labthings.actions.ActionThread` instance. If your function is called outside of an `labthings.actions.ActionThread` (e.g. by some internal code, not the web API), then this function will silently do nothing.

labthings.**update_action_progress**(*progress: int*)
Update the progress of the ActionThread in which the caller is currently running.

If this function is called from outside an ActionThread, it will do nothing.

**Parameters** **progress** – int: Action progress, in percent (0-100)

## 3.5 Synchronisation objects

### 3.5.1 Locks

Locks have been implemented to solve a distinct issue, most obvious when considering action tasks. During a long task, it may be necesarry to block any completing interaction with the LabThing hardware.

The *labthings.StrictLock* class is a form of re-entrant lock. Once acquired by a thread, that thread can re-acquire the same lock. This means that other requests or actions will block, or timeout, but the action which acquired the lock is able to re-acquire it.

**class** labthings.**StrictLock**(*timeout: int = - 1*, *name: Optional[str] = None*)
Class that behaves like a Python RLock, but with stricter timeout conditions and custom exceptions.

**Parameters** **timeout** (*int*) – Time in seconds acquisition will wait before raising an exception

**acquire**(*blocking: bool = True*, *timeout=<object object>*, *_strict: bool = True*)

**Parameters**

- **blocking** – (Default value = True)
- **timeout** – (Default value = sentinel)
- **_strict** – (Default value = True)

A CompositeLock allows grouping multiple locks to be simultaneously acquired and released.

**class** labthings.**CompositeLock**(*locks*, *timeout: int = - 1*)

Class that behaves like a `labthings.core.lock.StrictLock`, but allows multiple locks to be acquired and released.

**Parameters**

- **locks** (*list*) – List of parent RLock objects
- **timeout** (*int*) – Time in seconds acquisition will wait before raising an exception

**acquire**(*blocking: bool = True*, *timeout=<object object>*)

**Parameters**

- **blocking** – (Default value = True)
- **timeout** – (Default value = sentinel)

### 3.5.2 Per-Client events

**class** labthings.**ClientEvent**

An event-signaller object with per-client setting and waiting.

A client can be any Greenlet or native Thread. This can be used, for example, to signal to clients that new data is available

**clear**() → bool

Clear frame event, once processed.

**set**(*timeout=5*)

Signal that a new frame is available.

**Parameters** **timeout** – (Default value = 5)

**wait**(*timeout: int = 5*)

Wait for the next data frame (invoked from each client's thread).

**Parameters** **timeout** – int: (Default value = 5)

CHAPTER

FOUR

# ADVANCED USAGE

## 4.1 Components

*Documentation to be written*

## 4.2 Data encoders

*Documentation to be written*

## 4.3 LabThing Extensions

*Documentation to be written*

CHAPTER

# FIVE

# API REFERENCE

**exception** labthings.**ActionKilledException**
Sibling of SystemExit, but specific to thread termination.

**class** labthings.**ClientEvent**
An event-signaller object with per-client setting and waiting.

A client can be any Greenlet or native Thread. This can be used, for example, to signal to clients that new data is available

**clear**() → bool
Clear frame event, once processed.

**set**(*timeout=5*)
Signal that a new frame is available.

**Parameters** **timeout** – (Default value = 5)

**wait**(*timeout: int = 5*)
Wait for the next data frame (invoked from each client's thread).

**Parameters** **timeout** – int: (Default value = 5)

**class** labthings.**CompositeLock**(*locks*, *timeout: int = - 1*)
Class that behaves like a `labthings.core.lock.StrictLock`, but allows multiple locks to be acquired and released.

**Parameters**

- **locks** (*list*) – List of parent RLock objects
- **timeout** (*int*) – Time in seconds acquisition will wait before raising an exception

**acquire**(*blocking: bool = True*, *timeout=<object object>*)

**Parameters**

- **blocking** – (Default value = True)
- **timeout** – (Default value = sentinel)

**class** labthings.**LabThing**(*app: Optional[flask.app.Flask] = None*, *id_: Optional[str] = None*, *prefix: str = ''*, *title: str = ''*, *description: str = ''*, *version: str = '0.0.0'*, *types: Optional[List[str]] = None*, *format_flask_exceptions: bool = True*, *external_links: bool = True*, *json_encoder=<class 'labthings.json.encoder.LabThingsJSONEncoder'>*)
The main entry point for the application. You need to initialize it with a Flask Application:

```
>>> app = Flask(__name__)
>>> labthing = labthings.LabThing(app)
```

Alternatively, you can use *init_app()* to set the Flask application after it has been constructed.

**Parameters**

- **app** (*flask.Flask*) – the Flask application object
- **prefix** (*str*) – Prefix all routes with a value, eg v1 or 2010-04-01
- **title** (*str*) – Human-readable title of the Thing
- **description** (*str*) – Human-readable description of the Thing
- **version** (*str*) – Version number of the Thing
- **types** (*list of str*) – List of Thing types, used by clients to filter discovered Things
- **format_flask_exceptions** (*bool*) – JSON format all exception responses
- **external_links** (*bool*) – Use external links in Thing Description where possible
- **json_encoder** – JSON encoder class for the app

**add_component**(*component_object*, *component_name: str*)

Add a component object to the LabThing, allowing it to be used by extensions and other views by name, rather than reference.

**Parameters**

- **device_object** – Component object
- **device_name** – str: Component name, used by extensions to find the object

**add_root_link**(*view: Type[labthings.views.View]*, *rel: str*, *kwargs=None*, *params=None*)

**Parameters**

- **view** –
- **rel** –
- **kwargs** – (Default value = None)
- **params** – (Default value = None)

**add_view**(*view: Type[labthings.views.View]*, **urls: str*, *endpoint: Optional[str] = None*, ***kwargs*)

Adds a view to the api.

**Parameters**

- **view** – View class
- **urls** (*str*) – one or more url routes to match for the resource, standard flask routing rules apply. Any url variables will be passed to the resource method as args.
- **endpoint** (*str*) – endpoint name (defaults to `Resource.__name__()` Can be used to reference this route in *fields.Url* fields
- **kwargs** – kwargs to be forwarded to the constructor of the view.

Additional keyword arguments not specified above will be passed as-is to `flask.Flask.add_url_rule()`.

Examples:

```
labthing.add_view(HelloWorld, '/', '/hello')
labthing.add_view(Foo, '/foo', endpoint="foo")
labthing.add_view(FooSpecial, '/special/foo', endpoint="foo")
```

**property description: str**
Human-readable description of the Thing

**emit**(*event_type: str*, *data: dict*)
Find a matching event type if one exists, and emit some data to it

**Parameters**

- **event_type** – str:
- **data** – dict:

**init_app**(*app*)
Initialize this class with the given `flask.Flask` application. :param app: the Flask application or blueprint object

**Examples::** labthing = LabThing() labthing.add_view(...) labthing.init_app(app)

**register_extension**(*extension_object: labthings.extensions.BaseExtension*)
Add an extension to the LabThing. This will add API views and lifecycle functions from the extension to the LabThing

**Parameters** **extension_object** (*labthings.extensions.BaseExtension*) – Extension instance

**property safe_title: str**
Lowercase title with no whitespace

**property title: str**
Human-readable title of the Thing

**url_for**(*view: Type[labthings.views.View]*, ***values*)
Generates a URL to the given resource. Works like `flask.url_for()`.

**Parameters**

- **view** –
- **values** –

**property version: str**
Version number of the Thing

**view**(**urls: str*, ***kwargs*)
Wraps a `labthings.View` class, adding it to the LabThing. Parameters are the same as *add_view()*.

Example:

```
app = Flask(__name__)
labthing = labthings.LabThing(app)

@labthing.view('/properties/my_property')
class Foo(labthings.views.PropertyView):
    schema = labthings.fields.String()

    def get(self):
        return 'Hello, World!'
```

class labthings.**Schema**(*, *only: Optional[Union[Sequence[str], Set[str]]] = None*, *exclude: Union[Sequence[str], Set[str]] = ()*, *many: bool = False*, *context: Optional[Dict] = None*, *load_only: Union[Sequence[str], Set[str]] = ()*, *dump_only: Union[Sequence[str], Set[str]] = ()*, *partial: Union[bool, Sequence[str], Set[str]] = False*, *unknown: Optional[str] = None*)

Base schema class with which to define custom schemas.

Example usage:

```
import datetime as dt
from dataclasses import dataclass

from marshmallow import Schema, fields


@dataclass
class Album:
    title: str
    release_date: dt.date


class AlbumSchema(Schema):
    title = fields.Str()
    release_date = fields.Date()


album = Album("Beggars Banquet", dt.date(1968, 12, 6))
schema = AlbumSchema()
data = schema.dump(album)
data  # {'release_date': '1968-12-06', 'title': 'Beggars Banquet'}
```

**Parameters**

- **only** – Whitelist of the declared fields to select when instantiating the Schema. If None, all fields are used. Nested fields can be represented with dot delimiters.
- **exclude** – Blacklist of the declared fields to exclude when instantiating the Schema. If a field appears in both *only* and *exclude*, it is not used. Nested fields can be represented with dot delimiters.
- **many** – Should be set to *True* if `obj` is a collection so that the object will be serialized to a list.
- **context** – Optional context passed to *fields.Method* and *fields.Function* fields.
- **load_only** – Fields to skip during serialization (write-only fields)
- **dump_only** – Fields to skip during deserialization (read-only fields)
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.

Changed in version 3.0.0: *prefix* parameter removed.

Changed in version 2.0.0: *__validators__*, *__preprocessors__*, and *__data_handlers__* are removed in favor of *marshmallow.decorators.validates_schema*, *marshmallow.decorators.pre_load* and *marshmallow.decorators.post_dump*. *__accessor__* and *__error_handler__* are deprecated. Implement the *handle_error* and *get_attribute* methods instead.

**class Meta**

Options object for a Schema.

Example usage:

```
class Meta:
    fields = ("id", "email", "date_created")
    exclude = ("password", "secret_attribute")
```

Available options:

- `fields`: Tuple or list of fields to include in the serialized result.
- **`additional`: Tuple or list of fields to include *in addition* to the** explicitly declared fields. `additional` and `fields` are mutually-exclusive options.
- **`include`: Dictionary of additional fields to include in the schema. It is** usually better to define fields as class variables, but you may need to use this option, e.g., if your fields are Python keywords. May be an *OrderedDict*.
- **`exclude`: Tuple or list of fields to exclude in the serialized result.** Nested fields can be represented with dot delimiters.
- `dateformat`: Default format for *Date* <*fields.Date*> fields.
- `datetimeformat`: Default format for *DateTime* <*fields.DateTime*> fields.
- `timeformat`: Default format for *Time* <*fields.Time*> fields.
- **`render_module`: Module to use for *loads* <*Schema.loads*> and *dumps* <*Schema.dumps*>.** Defaults to *json* from the standard library.
- **`ordered`: If *True*, order serialization output according to the** order in which fields were declared. Output of *Schema.dump* will be a *collections.OrderedDict*.
- **`index_errors`: If *True*, errors dictionaries will include the index** of invalid items in a collection.
- `load_only`: Tuple or list of fields to exclude from serialized results.
- `dump_only`: Tuple or list of fields to exclude from deserialization
- **`unknown`: Whether to exclude, include, or raise an error for unknown** fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*.
- **`register`: Whether to register the *Schema* with marshmallow's internal** class registry. Must be *True* if you intend to refer to this *Schema* by class name in *Nested* fields. Only set this to *False* when memory usage is critical. Defaults to *True*.

**OPTIONS_CLASS**

alias of `marshmallow.schema.SchemaOpts`

**dump**(*obj: Any*, *, *many: Optional[bool] = None*)

Serialize an object to native Python data types according to this Schema's fields.

**Parameters**

- **`obj`** – The object to serialize.
- **`many`** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

**Returns** Serialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

Changed in version 3.0.0rc9: Validation no longer occurs upon serialization.

**dumps**(*obj: Any*, **args*, *many: Optional[bool] = None*, ***kwargs*)
Same as *dump()*, except return a JSON-encoded string.

**Parameters**

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

**Returns** A `json` string

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the serialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if `obj` is invalid.

**error_messages: Dict[str, str] = {}**
Overrides for default schema-level error messages

**fields: Dict[str, *marshmallow.fields.Field*]**
Dictionary mapping field_names -> `Field` objects

**classmethod from_dict**(*fields: Dict[str, Union[*marshmallow.fields.Field*, type]]*, ***, *name: str = 'GeneratedSchema'*) → type
Generate a *Schema* class given a dictionary of fields.

```
from marshmallow import Schema, fields

PersonSchema = Schema.from_dict({"name": fields.Str()})
print(PersonSchema().load({"name": "David"}))  # => {'name': 'David'}
```

Generated schemas are not added to the class registry and therefore cannot be referred to by name in *Nested* fields.

**Parameters**

- **fields** (*dict*) – Dictionary mapping field names to field instances.
- **name** (*str*) – Optional name for the class, which will appear in the `repr` for the class.

New in version 3.0.0.

**get_attribute**(*obj: Any*, *attr: str*, *default: Any*)
Defines how to pull values from an object to serialize.

New in version 2.0.0.

Changed in version 3.0.0a1: Changed position of `obj` and `attr`.

**handle_error**(*error: marshmallow.exceptions.ValidationError*, *data: Any*, ***, *many: bool*, ***kwargs*)
Custom error handler function for the schema.

**Parameters**

- **error** – The *ValidationError* raised during (de)serialization.
- **data** – The original input data.

- **many** – Value of `many` on dump or load.
- **partial** – Value of `partial` on load.

New in version 2.0.0.

Changed in version 3.0.0rc9: Receives *many* and *partial* (on deserialization) as keyword arguments.

**load**(*data: Union[Mapping[str, Any], Iterable[Mapping[str, Any]]]*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = None*)
Deserialize a data structure to an object defined by this Schema's fields.

**Parameters**

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

**Returns** Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

**loads**(*json_data: str*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*, *unknown: Optional[str] = None*, ***kwargs*)
Same as *load()*, except it takes a JSON string as input.

**Parameters**

- **json_data** – A JSON string of the data to deserialize.
- **many** – Whether to deserialize *obj* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

**Returns** Deserialized data

New in version 1.0.0.

Changed in version 3.0.0b7: This method returns the deserialized data rather than a `(data, errors)` duple. A `ValidationError` is raised if invalid data are passed.

**on_bind_field**(*field_name: str*, *field_obj:* marshmallow.fields.Field) → None
Hook to modify a field when it is bound to the *Schema*.

No-op by default.

**validate**(*data: Mapping*, ***, *many: Optional[bool] = None*, *partial: Optional[Union[bool, Sequence[str], Set[str]]] = None*) → Dict[str, List[str]]
Validate *data* against the schema, returning a dictionary of validation errors.

**Parameters**

- **data** – The data to validate.
- **many** – Whether to validate *data* as a collection. If *None*, the value for *self.many* is used.
- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.

**Returns** A dictionary of validation errors.

New in version 1.1.0.

class labthings.**Server**(*app*, *host='0.0.0.0'*, *port=7485*, *debug=False*, *zeroconf=True*)

Combined WSGI+mDNS server.

**Parameters**

- **host** (*string*) – Host IP address. Defaults to 0.0.0.0.
- **port** (*int*) – Host port. Defaults to 7485.
- **debug** (*bool*) – Enable server debug mode. Defaults to False.
- **zeroconf** (*bool*) – Enable the zeroconf (mDNS) server. Defaults to True.

**run**(*host=None*, *port=None*, *debug=None*, *zeroconf=None*)

Starts the server allowing for runtime parameters. Designed to immitate the old Flask app.run style of starting an app

**Parameters**

- **host** (*string*) – Host IP address. Defaults to 0.0.0.0.
- **port** (*int*) – Host port. Defaults to 7485.
- **debug** (*bool*) – Enable server debug mode. Defaults to False.
- **zeroconf** (*bool*) – Enable the zeroconf (mDNS) server. Defaults to True.

**start**()

Start the server and register mDNS records

class labthings.**StrictLock**(*timeout: int = - 1*, *name: Optional[str] = None*)

Class that behaves like a Python RLock, but with stricter timeout conditions and custom exceptions.

**Parameters** **timeout** (*int*) – Time in seconds acquisition will wait before raising an exception

**acquire**(*blocking: bool = True*, *timeout=<object object>*, *_strict: bool = True*)

**Parameters**

- **blocking** – (Default value = True)
- **timeout** – (Default value = sentinel)
- **_strict** – (Default value = True)

labthings.**create_app**(*import_name*, *prefix: str = ''*, *title: str = ''*, *description: str = ''*, *types: Optional[list] = None*, *version: str = '0.0.0'*, *external_links: bool = True*, *handle_errors: bool = True*, *handle_cors: bool = True*, *flask_kwargs: Optional[dict] = None*)

Quick-create a LabThings-enabled Flask app

**Parameters**

- **import_name** – Flask import name. Usually `__name__`.
- **prefix** (*str*) – URL prefix for all LabThings views. Defaults to "/api".

- **title** (*str*) – Title/name of the LabThings Thing.
- **description** (*str*) – Brief description of the LabThings Thing.
- **version** (*str*) – Version number/code of the Thing. Defaults to "0.0.0".
- **handle_errors** (*bool*) – Use the LabThings error handler, to JSON format internal exceptions. Defaults to True.
- **handle_cors** (*bool*) – Automatically enable CORS on all LabThings views. Defaults to True.
- **flask_kwargs** (*dict*) – Keyword arguments to pass to the Flask instance.
- **prefix** – str: (Default value = "")
- **title** – str: (Default value = "")
- **description** – str: (Default value = "")
- **types** – list: (Default value = None)
- **version** – str: (Default value = "0.0.0")
- **external_links** – bool: Use external links in Thing Description where possible
- **handle_errors** – bool: (Default value = True)
- **handle_cors** – bool: (Default value = True)
- **flask_kwargs** – dict: (Default value = None)

**Returns** (Flask app object, LabThings object)

labthings.**current_action**()
Return the ActionThread instance in which the caller is currently running.

If this function is called from outside an ActionThread, it will return None.

**Returns** labthings.actions.ActionThread – Currently running ActionThread.

labthings.**current_labthing**(*app=None*)
The LabThing instance handling current requests.

Searches for a valid LabThing extension attached to the current Flask context.

**Parameters** **app** – (Default value = None)

labthings.**find_component**(*component_name: str*, *labthing_instance=None*)
Find a particular LabThings Component registered to a LabThing instance

**Parameters**

- **component_name** (*str*) – Fully qualified name of the component
- **labthing_instance** (*optional*) – LabThing instance to search for the component. Defaults to current_labthing.

**Returns** Python object registered as a component, or *None* if not found

labthings.**find_extension**(*extension_name: str*, *labthing_instance=None*)
Find a particular LabThings Extension registered to a LabThing instance

**Parameters**

- **extension_name** (*str*) – Fully qualified name of the extension
- **labthing_instance** (*optional*) – LabThing instance to search for the extension. Defaults to current_labthing.

**Returns** LabThings Extension object, or *None* if not found

labthings.**registered_components**(*labthing_instance=None*)
Find all LabThings Components registered to a LabThing instance

**Parameters** **labthing_instance** (*optional*) – LabThing instance to search for extensions. Defaults to current_labthing.

**Returns** Python objects registered as LabThings components

**Return type** list

labthings.**registered_extensions**(*labthing_instance=None*)
Find all LabThings Extensions registered to a LabThing instance

**Parameters** **labthing_instance** (*optional*) – LabThing instance to search for extensions. Defaults to current_labthing.

**Returns** LabThing Extension objects

**Return type** list

labthings.**update_action_data**(*data: dict*)
Update the data of the ActionThread in which the caller is currently running.

If this function is called from outside an ActionThread, it will do nothing.

**Parameters** **data** – dict: Action data dictionary

labthings.**update_action_progress**(*progress: int*)
Update the progress of the ActionThread in which the caller is currently running.

If this function is called from outside an ActionThread, it will do nothing.

**Parameters** **progress** – int: Action progress, in percent (0-100)

CHAPTER

SIX

# INSTALLATION

```
pip install labthings
```

# PYTHON MODULE INDEX

# INDEX

## E



## F



## G



## H



## I



## L



## M



## N



## O



## P



## R



## S

## T



## U



## V



## W