sensortank Documentation

Release 0.0

Jacob Kittley-Davies

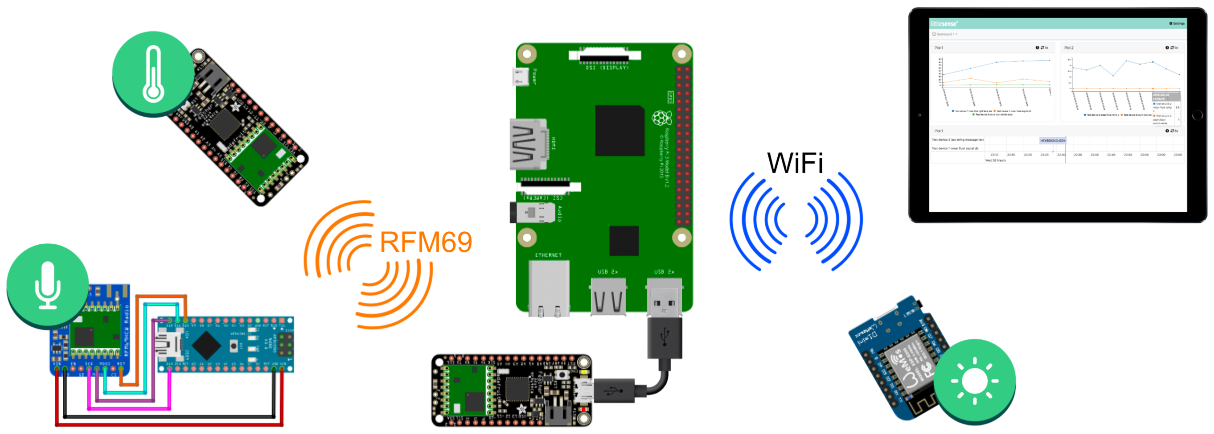
Mar 31, 2018

Contents

Note: This project has just been released so it may contain bugs. If you spot one then please report it as an issue on Github. We want to make this project awesome and to do that we need your help!

So you want to deploy a bunch of sensors and store the data they produce locally? Well you have come to the right place. Little Sense was developed as a foundation on which novice and experienced programmers can develope bespoke sensor network applciations. Out of the box Little Sense has all the code you need to sense, record and visualise data from any number of sensors. In addition the code has been specifically designed and written to make editing the layout and functionality easy.

At the heart of this project (and its creators) is the Raspberry Pi. The Pi acts as a server, listening for data over WiFi and Serial ports, storing the data it receives in a timeseries optimised database (InfluxDB). Radio receivers (such as RFM69) can be connected to the Raspberry Pi via USB allowing for low energy sensor devices to communicated via any radio or protocal (there's a bunch of examples in the repository). The Pi also provides a local website on which the recieved data can be inspected and the system monitored and debugged.



CHAPTER 1

Features

- All the latest tech: Python 3, InfluxDB, Flask. . .
- Simple setup and updates using Fabric automation scripts.
- Example sensor device code.
- Communication via various Radios and WiFi.
- Designed to be simple to expand and modify.
- Well documented.

CHAPTER 2

Contents

2.1 Getting Started

2.1.1 Installation

The fastest way to get this code up and running on a Raspberry Pi is with Fabric. Fabric is a tool which automates deployment of code and installation of dependancies on remote machines. In order to do this you must first setup the system on your local machine. N.B. We actually employ a fork of Fabric which is Python 3 compatable.

Local Install

1. Create a Python 3.5+ virtual environment (we recommend using Anaconda as a tool for managing local environments. If you have Anaconda installed you can call `` `conda create --name myenv` `` to create an environment called "myenv".
2. Step in to the environment with `` `conda activate myenv` `` (or equivelent).
3. Navigate to the directory where you cloned/downloaded this project and enter the server subfolder e.g `` `cd /Users/me/sensorStore/server` ``
4. Install the local requirements: `` `pip install -r requirements/local.txt` ``
5. Edit the "config/settings.py" as the comments instruct in the file.

If you don't want to run the system locally, you can skip ahead to the next section. However if you to make any modifications it is recommened that you complete the following steps so you can test before deployment.

6. Install InfluxDB. Visit InfluxDB and follow the instructions relativant to your operating system.
7. Export an environment variable: `` `export LOCAL=1` `` to tell the webserver it is running on a local machine (this will launch it in Debug mode).
8. To test the system run: `` `python webapp.py` `` and the Flask debug server should start.
9. Navigating to http://localhost:5000/ in your favorite browser and you should see the Little Sense web interface.

Prepare yor Raspberry PI

There are a number of operating system for the PI, however we recommend NOOBs. It may use a bit more disk space, but it if far easier for beginners and does all the drive expansion stuff for you.

1. Follow the instructions at https://www.raspberrypi.org/ to install NOOBs.
2. Exit the desktop to command line mode.
3. Run `` `sudo raspi-config` `` to enter configuration mode.

First we are going to configure the pi so it boots to the command line and not the desktop. This will save system resources.

1. Select "Boot Options" from the
2. Select "Desktop / CLI"
3. Select "Console"
4. The menu should return to the main menu.

Next we are going to change the host name. This allows the Pi to be accessed via <HOST_NAME>.local, which is much easier than entering the IP Address.

1. Select "Network Options"
2. Select "Hostname"
3. Read the instructions and hit ok
4. Enter a new hostname.

Now we want to enable a few service, namely SSH access so we can connect over the network and SPI so we can communicated via the SPI bus.

1. Select "Interface options"
2. Select "SSH" and answer "Yes" to enable this service.
3. Hit "ok" to return to the main menu.
4. Select "Interface options" again
5. Select "SPI" and answer "Yes" to enable this service.
6. Hit "ok" to return to the main menu.
7. Select "Finish" and "OK" to reboot.

Remote (Raspberry PI) First Install

Once you have prepared your Pi:

1. Activated your local virtual environment e.g. `` `conda activate myenv` ``.
2. Navigate to your local project folder and enter the server subfolder.
3. Run `` `fab -H littlesense.local init` `` to setup the Pi for the first time.
4. Enter your username and password. If you have specifed a SSH public key path in the settings, this will be the only time the Pi will ask you.

The init process can take quite a while - it will install large packages like Python 3.5 and update the OS. Don't worry deploying changes is much faster!

Remote Updates

When you want to update the Pi with changes that you have make locally (and tested!):

1. Make sure you are using the virtual environment and in the server folder.
2. Run the command: `` `fab -H littlesense.local deploy` ``.

2.1.2 Server Customisation (Optional)

On this page we detail some of the server configurations you may want to change from standard.

Creating an SSH-Key

To create an ssh key open command prompt on your local machine and:

1. `` `ssh-keygen -t rsa -b 4096 -C "your_email@example.com" ``
2. When asked where to save it, just use the default by hitting enter.
3. Next you will be asked to enter a passphrase, again just hit "enter".
4. Check the files have been created:
   - On Mac: Run `` `ls ~/.ssh` ``
   - On Windows: Run `` `dir /c/Users/<YOUR_USERNAME>/` ``
   - On Linux: Run `` `ls ~/.ssh` ``

You should see the following files: "id_rsa" and "id_rsa.pub" among others. Important! "id_rsa" is private, do NOT copy this anywhwere! The "id_rsa.pub" file is one you transfer to other machines.

5. Set the PUBLIC_SSH_KEY variable in the settings file to "/Users/<YOUR_USERNAME>/.ssh/id_rsa.pub" for Mac and Linux users and "/c/Users/<YOUR_USERNAME>/" for Windows.

If you have not run the initial intall on the Pi yet then stop here. The SSH key will be added when the install begins. If you have already initialised the PI then run the following command: `` `fab add_ssh_key -H littlesense.local` ``

Changing the .local Address

If you have used the NOOBs installation of Raspbian then you can find the Raspberry Pi using http://raspberrypi.local/ rather than entering the IP Address. This is nice, but maybe you want something more related with your project. To change the host name i.e. raspberrypi, do the following:

Using raspi-config

1. Run `` `sudo raspi-config` ``
2. Select "Network Options"
3. Select "Hostname"
4. Read the instructions and hit ok
5. Enter a new hostname.

Command line

1. Connect to the Pi over SSH
2. Run `` `sudo nano /etc/hosts` ``
3. Change the "raspberrypi" part of the last entry (`` `127.0.1.1 raspberrypi` ``) to the name of your choice. Make sure you conform to URL conventions; don't use spaces, etc.
4. Run `` `sudo nano /etc/nginx/sites-enabled/<ROOT_NAME>` `` where ROOT_NAME is the ROOT_NAME you specified in the settings file.
5. Change any occurances of "raspberrypi" to the new name. Save and exit.
6. Run `` `sudo /etc/init.d/hostname.sh` `` to update the PI
7. Run `` `shutdown -r now` `` to reboot the Pi.
8. Test your <new_name>.local in the browser once the Pi has rebooted.

Reduce GPU memory

If you have no intention of using the Raspberry Pi's desktop environment then you can reduce the amount of memory allocated to the GPU. This frees up memory for other applications and may improve preformance.

1. Run `` `sudo raspi-config` ``
2. Select "Advanced options"
3. Select "Memory Split"
4. Set to "16"
5. Select "OK"
6. Select "Finish" and the Pi may ask to reboot.

2.1.3 Video Overview

This is a 10 minute video which walks you through the setup process from installing NOOBs on your Pi to having the server ready to collect data.

2.2 Getting Data

One of the most important things once the system is setup is to receive data from your sensor devices. Devices can transmit information to Little Sense in a number of ways. The simplest is by making http requests to the RESTful API over a wired of WiFi connection. However Little Sense also supports the connection of transceivers to the serial port. We suggest you start by generating some test data, just to make sure everything is working.

2.2.1 Generating Test Data

The `` `transceiver.py` `` script, which you will learn more about later, can generate test data which is designed to fit with the example Dashboards. Use the following command to create data every few seconds until the script is terminated.

```
`python transceiver.py --test`
```

2.2.2 Sending Data via WiFi

The simplest is to send data to Little Sense is via the RESTful API. Any device on the same local network as the server can make requests. See the *RESTful API Documentation* for full details of its functionality.

Arduino Example

Here is a quick example of how you might send data to the server. This code was tested on an Adafruit Feather esp8266.

```
#include <ESP8266WiFi.h> // https://github.com/esp8266/Arduino
#include <WiFiClient.h>  // https://github.com/esp8266/Arduino
#include <ArduinoJson.h> // https://arduinojson.org

// Wifi Credentials
String ssid = "YOUR WIFI NAME";
String pass = "YOUR PASSWORD";

// IP Address of LittleSense server
const char* host = "192.168.1.188";

// Make sure this is different for every device
String device_id = "test_device_1";

// Setup
ADC_MODE(ADC_VCC);
WiFiClient client;
void setup() {
    delay(2000);
    Serial.begin(115200);
    Serial.println("Little Sense WiFi Example");
    pinMode(LED_BUILTIN, OUTPUT);
}

// Loop
void loop() {
if (wifiConnection()) {
    String url = "/api/readings/";
    String pload = payload();
    String reply = httpPut(url, pload);
    delay(5000);
}

// Payload - Data to send
String payload() {
    StaticJsonBuffer<200> jsonBuffer;
    JsonObject& root = jsonBuffer.createObject();
    root["device_id"] = device_id;
    root["dtype"] = "float";
    root["field"] = "temperature";
    root["unit"] = "C";
    root["value"] = random(0,450) / 10;
    root["utc"] = "NOW";
    char JSONmessage[300];
    root.printTo(JSONmessage);
    return String(JSONmessage);
}
```

(continues on next page)

(continued from previous page)

```
// WiFi
boolean wifiConnection() {
    WiFi.begin(ssid.c_str(), pass.c_str());
    int count = 0;
    Serial.print("Waiting for Wi-Fi connection");
    while ( count < 20 ) {
        if (WiFi.status() == WL_CONNECTED) {
        Serial.println("");
        Serial.println("WiFi connected");
        Serial.println("IP address: ");
        Serial.println(WiFi.localIP());
        return (true);
        }
        delay(500);
        Serial.print(".");
        count++;
    }
    Serial.println("Timed out.");
    return false;
}

String httpPut(String url, String data) {
    if (client.connect(host, 80)) {
        Serial.println("Sending...");
        client.println("PUT " + url + " HTTP/1.1");
        client.println("Host: " + (String)host);
        client.println("User-Agent: ESP8266/1.0");
        client.println("Connection: close");
        client.println("Content-Type: application/json;");
        client.print("Content-Length: ");
        client.println(data.length());
        client.println();
        client.println(data);
        delay(10);
        // Wait foir reply
        unsigned long timeout = millis();
        while (client.available() == 0) {
        if (millis() - timeout > 5000) {
            Serial.println("!!! Client Timeout !!!");
            client.stop();
            break;
        }
        }
        // Read all the lines of the reply from server and print them to Serial
        while (client.available()) {
        String line = client.readStringUntil('\r');
        Serial.print(line);
        }
        // Close connection
        Serial.println();
        Serial.println("closing connection");
    }
    else {
        // Return error if failed to get response
        return "ERROR - No reply from host";
    }
```

(continues on next page)

(continued from previous page)

```
}
```

2.2.3 Sending Data via the Serial Port

In order for Little Sense to monitor the Serial ports of the Raspberry Pi, the server needs to run a background service.

The Background Service

On initial setup Little Sense creates a background task which keeps `` `transceiver.py` `` running constantly. This script listens to communications send via the serial port. In this way you can attach any radio transceiver via USB and relay messages to Little Sense. When you make a change and redeploy using the fabric command, this background service gets rebooted automatically. However if you want to control it manually then you can use the following commands (when SSH'd into the Pi).

`` `sudo systemctl status transceiver` `` - What is the current status of the transceiver `` `sudo systemctl stop transceiver` `` - Stop the service `` `sudo systemctl start transceiver` `` - Start the service `` `sudo systemctl restart transceiver` `` - Restart the service

Manually run the service

You can run the `` `transceiver.py` `` manually by calling `` `python transceiver.py --verbose` ``. We strongly recommend you stop the background service first, to save confusion. The –verbose flag tells the transceiver to script to print out debug information.

Attaching Radio Receivers

Tutorial coming soon.

2.2.4 Common Issues

In this section we make not of a few things which may trip you up when trying to connect devices to Little Sense.

Connecting to the RESTful API

Are you trying to connect to a .local address?

Some devices and software libraries cannot automatically discover .local addresses. Instead try the IP address of the Little Sense server. You may also need to check that the Nginx configuration of your Little Sense server allows the IP address (see below).

Getting a 444 response?

Check that the Nginx configuration allows the IP address. Loging to the Little Sense server via SSH and type `` `sudo nano /etc/nginx/sites-available/littlesense` ``. Make sure the IP address is listed as below.

```
# Block all names not in list i.e. prevent HTTP_HOST errors
if ($host !~* ^(192.168.1.100|littlesense.local)$) {{
    return 444;
}}
```

Note: You can through any number of key:value pairs at the server (along with a UTC time string and a unique sender device id) and the server will record them without question. You don't even need to register devices first!. However, Little Sense will only keep the data of unregistered devices for a limited time. Devices can easily be registered via the web interface and once registered their data will be kept forever.

2.2.5 Sensing Devices

Example sensor code for various harware platforms such as Arduinoscan be found in the repository here: https://github.com/jkittley/LittleSense/tree/master/device

2.3 Customising Little Sense

Now you are all setup, you can make Little Sense your own by adding a little style.

2.3.1 Change the look and feel

Making Little Sense your own couldn't be easier. There are a few things you need to know before you get started.

1. `` `static/img/` `` holds all the images.
2. `` `templates/` `` is where all the HTML, CSS and Javascript lives. The CSS and Javascript needed by each is either included in the template itself or in its parent. For experienced programmers this may appear as we have gone mad, but we chose to structure things this was so it is really obvious to novices what effects what.
3. `` `templates/base.html` `` is extended by all other templates. For more information about how to manipulate the templates see http://flask.pocoo.org/docs/templating/.

2.3.2 Dashboards

Dashboards are just a collections of plots and can be defined in a Dashboard JSON files. These files can be found in the `` `data/dashboards` `` directory. Little Sense automatically scans this folder and adds new dashboards to the menu. Below we dicuss how dashboard files are constructed.

At the top level dashboards have 3 sections: A *header*, a *footer* and an array of *plots*. The header and footer can either be "null" as per the footer in the example below, or it can be an object with a title and text attribute, see header in example.

```
{
    "header": {
        "title": "Some title",
        "text": "Some descriptive text"
    },
    "footer": null,
    "plots": [ "-- See below --" ]
}
```

Warning: JSON does not like trailing commas (unlike Python) and is very strict! Curious Concept have a nice validation tool which will happlily point out all your mistakes.

Plots

Each item in the plots array must be an object like the one in the example below.

```
"plots": [
    {
        "title": "Plot 1",
        "width": 6,
        "help": "Some help text for the viewer",
        "metrics": [ '-- See below --' ],
        "time": {},
        "type": "timeseries",
        "refresh_interval_seconds": 5
    },
    ...
]
```

title The plot title

width The plot width. A vaulue from 1 to 12

help Help text for the user

metrics See below

time See below

type The type of plot. Must be one of the following: "timeseries", more soon.

refresh_interval_seconds Number of seconds between each refresh of the plot. Can be set to null to disable automatic refresh.

Time

The time attribute must be an object and be structured as follows:

```
"time": {
    "start": null,
    "end": null,
    "reading_interval_seconds": 10,
    "period": {
        "days": 0,
        "hours": 0,
        "minutes": 5
    },
    "fill": "none"
}
```

start The start of the time period to be displayed. If null the default will be the end minus the period.

end The end of the time period to be displayed. If null then the current time is used and will auto update to the current time each refresh.

reading_interval_seconds Group readings into intervals of this many seconds.

period A period of time messured in days, hours and minutes. If start and end are set this is ignored.

fill If “none” then reading_intervals which contain no readings will be skipped. If “null” then reading_intervals with no readings return a null value.

Metrics

Metrics represent the data to be displayed and how to present it.

```
"metrics": [
    {
        "name": "Device 1: Signal (dB)",
        "field_id": "float_signal_db",
        "device_id": "test_device_1",
        "aggrfunc": "mean"
    },
    ...
]
```

name A friendly name for the user to read

field_id The id of the field to be displayed

device_id The device from which the field should be obtained

aggrfunc The method of aggregating readings when grouped by intervals. The values should be set to one of the following depending on the data type of the field.

- Numeric: ‘count’, ‘mean’, ‘mode’, ‘median’, ‘sum’, ‘max’, ‘min’, ‘first’ or ‘last’.
- Booleans: ‘count’, ‘first’ or ‘last’ and
- Strings: ‘count’, ‘first’ or ‘last’.

Full example

```
{
    "header": null,
    "footer": null,
    "plots": [
        {
            "id": 1,
            "pos": 1,
            "title": "Plot 1",
            "width": 6,
            "help": "Some help text for the viewer",
            "metrics": [{
                "name": "Device 1: Signal (dB)",
                "field_id": "float_signal_db",
                "device_id": "test_device_1",
                "aggrfunc": "mean"
            },{
                "name": "Device 1: Max Light Level (Lux)",
                "field_id": "int_light_level_lux",
                "device_id": "test_device_1",
                "aggrfunc": "max"
            },{
                "name": "Device 2: Switch State (OnOff)",
```

(continues on next page)

(continued from previous page)

```
                "field_id": "bool_switch_state",
                "device_id": "test_device_2",
                "aggrfunc": "count"
            }],
            "time": {
                "start": null,
                "end": null,
                "reading_interval_seconds": 10,
                "period": {
                    "days": 0,
                    "hours": 0,
                    "minutes": 5
                },
                "fill": "none"
            },
            "type": "timeseries",
            "refresh_interval_seconds": 5
        },
        {
            "id": 2,
            "pos": 2,
            "title": "Plot 2",
            "width": 6,
            "help": "Some help text for the viewer",
            "metrics": [{
                "name": "Device 2: Temp (C)",
                "field_id": "float_temp_c",
                "device_id": "test_device_2",
                "agrfunc": "mean"
            },{
                "name": "Device 2: Switch State (OnOff)",
                "field_id": "bool_switch_state",
                "device_id": "test_device_2",
                "aggrfunc": "count"
            }],
            "time": {
                "start": null,
                "end": null,
                "reading_interval_seconds": 5,
                "period": {
                    "days": 0,
                    "hours": 0,
                    "minutes": 5
                },
                "fill": "none"
            },
            "type": "timeseries",
            "refresh_interval_seconds": 5
        }
        ]
}
```

2.3.3 Notes on editing CSS and JS

When you are looking at code someone else has written it can be very confusing and difficult to understand what impact the changes you make will have on other parts of the codebase whoch you may not have looked at yet. In Little

Sense, all the Javascript and CSS is stored alongside the HTML in the template files. This means that when you want to make a change all the elements are available in one place and mor importantly, you can be sure that the changes you make only impact the template you are editing and the templates which extend it.

In short, everything extends `` `templates/base.html` ``. System templates all extend `` `templates/system/base.html` `` which extends `` `templates/base.html` `` but also includes `` `templates/system/menu.html` ``.

Below is an inheritance diagram showing which templates extend what.

- **base.html**
  - dashboards/index.html
  - **system/base.html**
    - backup.html
    - configure.html
    - db.html
    - devices.html
    - index.html
    - logs.html
    - preview.html
    - serial.html
    - ungerister.html

2.3.4 Add new pages

Should you want to add a new page to the site you can create a new template in the 'tempaltes' folder. The web interface is build on [Flask](http://flask.pocoo.org/) and the internet is full of great tutorials on how to build [Flask](http://flask.pocoo.org/) websites, so rather than repeat others, we point you to [Flask](http://flask.pocoo.org/) as a starting point.

Querying readings from pages

Firstly we recommend adding dashboards rather than new pages. However if you need query the readings for some readon. Here is an example of how to do it. Details of what each field means can be found in the RESTfull API section.

```
$.ajax({
    dataType: "json",
    method: "POST",
    url: "/api/readings/get",
    data: {
        start: "2018-01-11T09:16:50+00:00",
        end: "2018-03-11T09:14:50+00:00",
        fill: "null",
        interval: reading_interval,
        metrics: JSON.stringify([{
                "name": "Device 10: Temp (C)",
                "field_id": "float_temp_c",
                "device_id": "test_device_10",
                "agrfunc": "mean"
```

(continues on next page)

(continued from previous page)

```
            },{
                "name": "Device 10: Light Level (Lux)",
                "field_id": "int_light_level_lux",
                "device_id": "test_device_4",
                "agrfunc": "max"
            }]),
        success: function(results) {},
        error:  function(results) {}
}
```

2.3.5 Periodic Tasks

Some times you want to run periodic tasks. Little Sense for example, periodically cleans the log files and deletes old backups. If you want to add your own periodic task then open up cronjobs.py in the server folder and add a new static method in the Maintenance class.

```
@staticmethod
def my_custom_task():
    # Your code here
    pass
```

Next we need to tell the server to add the task you have created to the cron background manager. Near the bottom of the Maintenance class there is a function called "update_crontab". At the bottom add the following code:

```
@staticmethod
def update_crontab():

    ...

    task = 'my_custom_task'
    job = my_cron.new(command='{venv}bin/python {script} --task {task}'.format(
        venv=settings.DIR_VENV,
        script=this_script,
        task=task)
    )
    job.setall('0 0 * * *')
    my_cron.write()
```

The section `` `job.setall('0 0 * * *')` `` is where you tell the cron service when and how ofter you want your task to run. See the [Python Crontab}(https://github.com/doctormo/python-crontab)_ for more information on how to set these values.

2.3.6 Utilities (utils package)

Python is an Object Orientated programing language and as such most of the core concepts in Little Sense are models as such. The "utils" package contains these objects Classes e.g. Dashboards and Devices. Much of the functionality is contained within these classes and is the best place to add new features which will then become avaiable system wide.

Warning: We advice great caution to novice developers, however do not be put off and breaking code like this is the best way to learn, but maybe keep a copy backed up just incase and document what you change.

Functions

get_InfluxDB()	Get an instance of the InfluxDB database
influx_connected()	Test connection to InfluxDB database

get_InfluxDB

`utils.`**`get_InfluxDB`**`()`
Get an instance of the InfluxDB database

influx_connected

`utils.`**`influx_connected`**`()`
Test connection to InfluxDB database

Classes

BackupManager()	This class manages all thing relating to backups.
DashBoards()	Dashboard Collection (Iterable)
Device(device_id, create_if_unknown)	Individual Device
Devices()	Devices Collection (Iterator)
Field(dtype, name, unit)	.
FieldValue(field, value)	Field Value
Logger()	Custom logger with InfluxDB backend
Metric(device, field, aggregation_function)	Data type for reading query metric.
Readings(readings, device_id, time_start, . . .)	Datatype for returned readings from readings query
SerialLogger	Log specifically designed for the serial data.
SerialTXQueue()	An interface between the web UI and serial interface.

BackupManager

`class` `utils.`**`BackupManager`**
Bases: `object`

This class manages all thing relating to backups.

Methods Summary

create(messurement, start, end)	Create a new backup locally.
delete_backup(filename)	Delete specified backup files.
get_backups()	List backup files which exist locally.
get_backups_folder()	Get the path to the local backup folder.

Methods Documentation

create (*messurement: str*, *start: str = None*, *end: str = None*) → str
Create a new backup locally.

Parameters

- **messurement** – The messurement type i.e. 'readings' or 'logs'.
- **start** – Datetime start of backup period (default: 1 hour ago).
- **end** – Datetime end of backup period (default: Now).

Returns Saved file name

Raises LookupError if no data to backup

delete_backup (*filename: str*)
Delete specified backup files.

Parameters **filename** – The name of the backup file to delete. This cannont be a path and must be in the backups folder.

Raises `FileNotFoundError`

get_backups () → List[Dict]
List backup files which exist locally.

Returns List of dictionarys containing filename, stem, dataset, start and end.

get_backups_folder () → unipath.path.Path
Get the path to the local backup folder. Automatically creates folder if does not exist.

Returns List of Paths

Return type path

DashBoards

class utils.**DashBoards**
Bases: `object`

Dashboard Collection (Iterable)

Methods Summary

`get`(**kwargs)	Get specific dashbaord.
`update`()	Reload dashboard from file i.

Methods Documentation

get (***kwargs*)
Get specific dashbaord.

Keyword Arguments **slug** – The slug assosiated with the dashboard.

Returns Dashboard is successful else None

update ()

Reload dashboard from file i.e. update cache

Device

class `utils.`**`Device`**(*device_id: str*, *create_if_unknown: bool = False*)

Bases: `object`

Individual Device

Parameters

- **`device_id`** – The unique ID of the device.
- **`create_if_unknown`** – If the device_id is not known then create a new Device.

Methods Summary

`add_reading`(utc, field_values, commlink_name)	Add a new reading for the device.
`as_dict`()	Device represented as a dictionary
`count`()	Count the number of readings for this device
`fields`(**kwargs)	List all fields which this device has at some time reported values for.
`get_commlinks`()	Get a list off commlinks which have provided data to this devices data
`get_dtypes`(category)	Get the most recent reading from this devices data.
`get_readings`(**kwargs)	Get the most recent reading from this devices data.
`is_registered`([setto])	Set/Get device registration status
`last_reading`()	Get the most recent reading from this devices data.
`set_name`(name)	Set a human readable name for the device.

Methods Documentation

`add_reading`(*utc: str*, *field_values*, *commlink_name: str*)

Add a new reading for the device.

Parameters

- **`utc`** – UTC timestamp as a string.
- **`field_values`** (*`list of utils.FieldValues`*) – List of FieldValue objects
- **`commlink_name`** – Name of the commulication link through which the reading was devivered

Raises

- *`utils.exceptions.InvalidUTCTimestamp`* – If UTC value is incorrectly formatted
- *`utils.exceptions.IllformedField`* – If field name does not conform to the prescribed pattern or conatins eronious values.

`as_dict`()

Device represented as a dictionary

Returns: Device represented as a dictionary:

```
{
    "device_id": (str) The device ID,
    "registered": (bool) Is this device registered,
    "fields": {
        see fields for details
    },
    "last_update": last update,
    "last_upd_keys": last update keys
}
```

count()
Count the number of readings for this device

Returns Number of readings for this device

Return type int

fields(***kwargs*)
List all fields which this device has at some time reported values for.

Keyword Arguments **only** (*list*) – List of device_ids to which results should be limited.

Returns List of Fields, each carrying information pertaining to the Field.

Return type list of utils.Field

get_commlinks()
Get a list off commlinks which have provided data to this devices data

Returns List of commlink names.

Return type list

get_dtypes(*category: str = None*)
Get the most recent reading from this devices data.

Parameters **category** (*str*) – What sub type of dtypes to return. Options: numbers, strings, booleans

Returns List of acceptable data types.

Return type list

get_readings(***kwargs*)
Get the most recent reading from this devices data.

Keyword Arguments

- **fill** (*str*) – A fill mode for intervals with no data. Options:: "none" don't fill and "null" fill intervals with null readings (defaults to none).
- **interval** (*int*) – Reading interval in seconds. Readings are groupded into intervals (defaults to 5s).
- **metrics** (*list of utils.Metrics*) – Limit fields to this lists field_ids (defaults to all)
- **start** (*arrow.arrow.Arrow*) – Datetime start of search period (defaults to 1 hour ago)
- **end** (*arrow.arrow.Arrow*) – Datetime end of search period (defaults to now)
- **limit** (*int*) – Limit the number of readings (Max = 5000, Min = 1).

Returns Readings matching query.

Return type *Readings*

`is_registered`(*setto=None*) → bool
Set/Get device registration status

Parameters **`setto`** (*`bool`*) – New registration status

Returns Current registration status

`last_reading`()
Get the most recent reading from this devices data.

Returns The most recent reading. last_upd_keys: The keys assosiated with the reading.

Return type last_upd

`set_name`(*name: str*)
Set a human readable name for the device.

Parameters **`name`** – New human readable name for the device

Devices

class `utils.`**`Devices`**
Bases: `object`

Devices Collection (Iterator)

Methods Summary

`as_dict`(update)	Get Devices collection represented as a dictionary
`get`(device_id, create_if_unknown)	Get (or create) a specific Device
`purge`(**kwargs)	Purge (delete) databases of requested content
`stats`()	Generate device related statistics
`to_dict`(devices)	Converts list of Devices to list of dictionaries
`update`()	Refreshed cache of known devices from database

Methods Documentation

`as_dict`(*update: bool = False*) → dict
Get Devices collection represented as a dictionary

Parameters **`update`** – Update cache before returning.

Returns Dictionary containing three lists. All devices (all), Registered devices (registered) and Unregistered devices (unregistered)

Return type dict

`get`(*device_id: str*, *create_if_unknown: bool = False*)
Get (or create) a specific Device

Parameters **`device_id`** – The unique ID of the device.

Keyword Arguments **`create_if_unknown`** – If the device_id is not known then create a new Device.

Returns If device_id is found returns a Device. If no mathcing device found and create_if_unknown is False then returns None. However if create_if_unknown is True then return the newly created Device()

Return type *Device*

purge(***kwargs*)
Purge (delete) databases of requested content

Keyword Arguments

- **registered** (*bool*) – Purge registered devices.
- **unregistered** (*bool*) – Purge unregistered devices.
- **registry** (*bool*) – Purge known device registry.
- **start** (*str*) – DateTime string from when to purge data.
- **end** (*str*) – DateTime string until when to purge data.

Raises `ValueError` – If ill defined purge

stats()
Generate device related statistics

Returns total_readings, last_update, last_update_humanized, list of registered_devices

Return type dict

to_dict(*devices: list*)
Converts list of Devices to list of dictionaries

Parameters **devices** – List of Device() objects

Returns List of dictionaries, each representing a Device()

Return type list

update()
Refreshed cache of known devices from database

Field

class utils.**Field**(*dtype: str*, *name: str*, *unit: str*)
Bases: `object`

DATA_TYPES_NUMERIC
list – List of acceptable numeric data types

DATA_TYPES_ALPHANUMERIC
list – List of acceptable alphanumeric (string) data types

DATA_TYPES_BOOLEAN
list – List of acceptable boolean data types

DATA_TYPES
List of all acceptable data types

id
str – Uniquie field identifier e.g. float_light_level_lux

dtype
str – Type of data being added

name
str – The name of the field

unit
str – The unit of messurement

Attributes Summary

DATA_TYPES
DATA_TYPES_ALPHANUMERIC
DATA_TYPES_BOOLEAN
DATA_TYPES_NUMERIC
id
is_boolean
is_numeric
is_string

Methods Summary

as_dict()	Get representation as dictionary.
fromDict(dictionary)	Create Field() from dictionary.
fromString(field_string)	Create Field() from string e.

Attributes Documentation

DATA_TYPES = ['float', 'int', 'string', 'str', 'bool', 'boolean']

DATA_TYPES_ALPHANUMERIC = ['string', 'str']

DATA_TYPES_BOOLEAN = ['bool', 'boolean']

DATA_TYPES_NUMERIC = ['float', 'int']

id

is_boolean

is_numeric

is_string

Methods Documentation

as_dict()
Get representation as dictionary.

Returns Dictionary of core field values i.e. dtype, name, unit, value, is_numeric, is_boolean and is_string.

Return type dict

classmethod fromDict(*dictionary: dict*)
Create Field() from dictionary.

Parameters dictionary – A dictionary containing keys: dtype, name and unit.

classmethod fromString(*field_string*)
Create Field() from string e.g. float_light_level_lux.

FieldValue

class utils.**FieldValue**(*field*, *value: <built-in function any> = None*)
Bases: `object`

Field Value

field
(utils.Field): Field of messurement

value
any – The value of the reading

Logger

class utils.**Logger**
Bases: `object`

Custom logger with InfluxDB backend

Methods Summary

comms(msg, **kwargs)	Get list of categories and human readable names.
debug(msg, **kwargs)	See comms()
device(msg, **kwargs)	See comms()
error(msg, **kwargs)	See comms()
funcexec(msg, **kwargs)	See comms()
get_categories()	Get list of categories and human readable names.
interaction(msg, **kwargs)	See comms()
list_records(**kwargs)	List log records.
purge(**kwargs)	Purge (delete) log records.
stats()	Get statistics about the current logs

Methods Documentation

comms(*msg: str*, ***kwargs*)
Get list of categories and human readable names.

Parameters **msg** – Message to record

Keyword Arguments ***** – All keyworord arguments which are JSON serialisable are recorded as sublimental information.

debug(*msg: str*, ***kwargs*)
See comms()

device(*msg: str*, ***kwargs*)
See comms()

error(*msg: str*, ***kwargs*)
See comms()

funcexec(*msg: str*, ***kwargs*)
See comms()

get_categories()
Get list of categories and human readable names.

Returns List of tuples (category_id, fiendly name)

Return type list

interaction(*msg: str*, ***kwargs*)
See comms()

list_records(***kwargs*)
List log records.

Keyword Arguments

- **cat** (*str*) – Category
- **limit** (*int*) – Limit results
- **offset** (*int*) – Results offset
- **start** (*str*) – UTC Datetime formatted as string - Results period start
- **end** (*str*) – UTC Datetime formatted as string - Results period end
- **orderby** (*str*) – Formatted as field and order e.g. time ASC or time DESC

Returns

Dictionary of results and pagination:

```
{
    total: Total number of records,
    page_start: Page starts at result x,
    page_end: Page ends at result x,
    num_pages: Total number of pages,
    page_num: Current page number,
    results: List of results
}
```

Return type dict

purge(***kwargs*)
Purge (delete) log records.

Keyword Arguments

- **start** (*str*) – UTC Datetime formatted as string - Period start
- **end** (*str*) – UTC Datetime formatted as string - Period end
- **categories** (*list*) – List pf category ids to delete

stats()
Get statistics about the current logs

Returns Dictionary containing overall record ‘count’ and counts for each of the ‘categories’

Return type dict

Metric

class utils.**Metric**(*device*, *field*, *aggregation_function: str = None*)

Bases: `object`

Data type for reading query metric.

Attributes Summary

AGGR_FUNC_BOOLEAN
AGGR_FUNC_NUMERIC
AGGR_FUNC_STRING

Methods Summary

fromString(string)	Initialize from a string e.

Attributes Documentation

AGGR_FUNC_BOOLEAN = ['count', 'first', 'last']

AGGR_FUNC_NUMERIC = ['mean', 'mode', 'median', 'count', 'sum', 'max', 'min', 'first',

AGGR_FUNC_STRING = ['count', 'first', 'last']

Methods Documentation

classmethod fromString(*string: str*)

Initialize from a string e.g. device_id,field_id,aggregation

Readings

class utils.**Readings**(*readings: list*, *device_id: str*, *time_start: str*, *time_end: str*, *time_interval: int*, *fillmode: str*, *query_fields: dict*)

Bases: `object`

Datatype for returned readings from readings query

Parameters

- **readings** – List of readings
- **device_id** – Device ID
- **time_start** – Readings earliest date and time
- **time_end** – Readings latest date and time
- **time_interval** – Readings interval in seconds
- **fillmode** – Reading fill mode
- **fields_data** – List of dicts each represnting a field

Methods Summary

all(**kwargs)	Return all readings as a dictionary:
merge_with(other)	Merge another Readings object into this one.
timestamps()	Get list of timestamps in results

Methods Documentation

all(***kwargs*)
Return all readings as a dictionary:

Keyword Arguments **format** (*str*) – How the readings should be formatted. Options: list ot by_time

Returns: Readings as a dictionary:

```
{
    "device_ids": (list) List of device ids which contributed to these␣
↪readings,
    "start": (str) Readings earliest date and time,
    "end": (str) Readings latest date and time,
    "count": (int) Numbner of readings,
    "fields": [
        see device.fields()
    ],
    "field_ids": (list) Field ids,
    'readings': Readings as either a list of dicts (list) or as a dict where␣
↪key=time and val=dict(field->val)
}
```

merge_with(*other*)
Merge another Readings object into this one.

Parameters **other** (*Reading*) – Readings objects to consume

timestamps()
Get list of timestamps in results

Returns List of string date time stamps

Return type list

SerialLogger

class utils.**SerialLogger**
Bases: `object`

Log specifically designed for the serial data.

Methods Summary

add_message(rxtx, message)	Add a serial message to the log.
rx(message)	Shortcut to add_message for RX

Continued on next page

Table 13 – continued from previous page

tail(nlines)	Return the tail of the logg file
tx(message)	Shortcut to add_message for TX

Methods Documentation

add_message (*rxtx: str*, *message: str*)
Add a serial message to the log.

Parameters

- **rxtx** – Was the message received (rx) or sent (tx)
- **message** – Message to record

rx (*message: str*)
Shortcut to add_message for RX

Parameters **message** – Message to record (as RX)

tail (*nlines: int = 15*)
Return the tail of the logg file

Parameters **nlines** – Number of lines to return

Returns lines seporated with newlines

Return type str

tx (*message: str*)
Shortcut to add_message for TX

Parameters **message** – Message to record (as TX)

SerialTXQueue

class utils.**SerialTXQueue**
Bases: object

An interface between the web UI and serial interface. Messages can be added to the queue which will then be sent by the background transeiver service.

Methods Summary

pop()	Return and remove the next message in the queue.
push(message)	Add a messages to the send queue.
remove(message_id)	Remove msg from TX queue

Methods Documentation

pop ()
Return and remove the next message in the queue.

Returns Dictionary containing the message and meta data i.e. doc_id - the unique id of the message in the queue, and the time it was created. Returns "None" if no messages in the queue.

Return type dict

push(*message: str*)
Add a messages to the send queue.

Parameters **message** – The message to be transmitted.

Returns A unique reference for the newly created message in the queue.

Return type int

remove(*message_id: int*)
Remove msg from TX queue

Parameters **message_id** – The unique id of the message in the queue to remove.

2.3.7 Exceptions (utils.exceptions)

The utils.exceptions module contains a number of bespoke exceptions. These were implemented just to make the reasons for failure more obvious.

Classes

IllformedField	Thrown when a field is not correctly formed
InvalidFieldDataType	Thrown when a field name contains an unkown data type
InvalidPacket	Thrown when the packet formatter cannot parse the incomming data
InvalidReadingsRequest	Thrown when a request for readings has an invalid format
InvalidUTCTimestamp	Thrown when the passed UTC string cannot be parsed
UnknownDevice	Raised when an unknown device id is used
UnknownFieldName	Thrown when a field name is not know to be assosiated with a device

IllformedField

exception utils.exceptions.**IllformedField**
Thrown when a field is not correctly formed

InvalidFieldDataType

exception utils.exceptions.**InvalidFieldDataType**
Thrown when a field name contains an unkown data type

InvalidPacket

exception utils.exceptions.**InvalidPacket**
Thrown when the packet formatter cannot parse the incomming data

InvalidReadingsRequest

exception utils.exceptions.**InvalidReadingsRequest**
Thrown when a request for readings has an invalid format

InvalidUTCTimestamp

exception utils.exceptions.**InvalidUTCTimestamp**
Thrown when the passed UTC string cannot be parsed

UnknownDevice

exception utils.exceptions.**UnknownDevice**
Raised when an unknown device id is used

UnknownFieldName

exception utils.exceptions.**UnknownFieldName**
Thrown when a field name is not know to be assosiated with a device

2.4 RESTful API Documentation

This section documents the RESTful API offered by Little Sense. Using the URI endpoints listed below you can add and examine data collected by sensor devices on your network.

2.4.1 Summary

Resource	Operation	Description
Add reading	*PUT /api/readings/*	Add a new reading.
Device	*GET /api/device/(string:device_id)*	Get individual device info.
Device Collection	*GET /api/devices*	Get collection of devices.
Readings Collection	*GET /api/readings/*	Get collection of readings.

2.4.2 API Details

GET /api/device/(string: *device_id*)
List Devices

Example request:

```
GET /api/device/test_device_1 HTTP/1.1
Host: littlesense.local
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "registered": false,
  "last_upd_keys": [
      "int_light_level_lux",
      "float_signal_db",
  ],
  "device_id": "test_device_1",
  "last_update": {
      "int_light_level_lux": 135,
      "float_signal_db": 3.6,
      "time": "2018-03-20T13:52:22Z",
  },
  "fields": [
      {
          "dtype": "int",
          "unit": "lux",
          "id": "int_light_level_lux",
          "is_string": false,
          "is_boolean": false,
          "name": "Light level",
          "is_numeric": true
      },
      {
          "dtype": "float",
          "unit": "db",
          "id": "float_signal_db",
          "is_string": false,
          "is_boolean": false,
          "name": "Signal",
          "is_numeric": true
      }
    ]
  }
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – device found
- 404 Not Found – device not found

GET /api/devices

List Devices

Example request:

```
GET /api/devices/ HTTP/1.1
Host: littlesense.local
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

[
  {
      "device_id": "test_device_1",
      "fields": [
          {
              "is_numeric": true,
              "unit": "db",
              "is_string": false,
              "dtype": "float",
              "id": "float_audio_level_db",
              "name": "Audio level",
              "is_boolean": false
          },
          {
              "is_numeric": true,
              "unit": "db",
              "is_string": false,
              "dtype": "float",
              "id": "float_signal_db",
              "name": "Signal",
              "is_boolean": false
          }
      ],
      "last_upd_keys": [
          "float_audio_level_db",
          "float_signal_db",
      ],
      "last_update": {
          "float_audio_level_db": 1.3,
          "float_signal_db": 4.5,
      },
      "registered": false
  },

]
```

Query Parameters

- **`filter`** – Filter results by “registered” or “unregistered

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – posts found

`PUT /api/readings/`

Add reading for device

Example request:

```
POST /api/readings HTTP/1.1
Host: littlesense.local
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
  "success": true
}
```

Query Parameters

- **device_id** – Device ID e.g. test_device_id
- **utc** – UTC timestamp string e.g. 2018-03-20T13:30:00Z. If absent or set to "NOW", the server will add a timestamp on receiving.
- **field** – Field ID e.g. float_light_level_lux
- **dtype** – Field data type e.g. float or int
- **unit** – Field messurement unit e.g. lux or C
- **value** – Value to record

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – query successful found
- 400 Bad Request – invalid request

GET /api/readings/

Query readings from one or more device

Example request:

```
GET /api/readings HTTP/1.1

?start=2018-03-20T13%3A30%3A00Z
&end=2018-03-20T13%3A35%3A00Z
&fill=none
&interval=5
&metric[]=test_device_2,float_temp_c,mean
&metric[]=test_device_2,bool_switch_state,count

Host: littlesense.local
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
{
  "device_ids": [
      "test_device_2"
  ],
  "count": 2,
  "readings": [
      {
      "test_device_2__count__bool_switch_state": 2,
      "test_device_2__mean__float_temp_c": 22.54,
      "time": "2018-03-20T13:34:35Z"
      },
      {
      "test_device_2__count__bool_switch_state": 2,
      "test_device_2__mean__float_temp_c": 31.25,
      "time": "2018-03-20T13:34:40Z"
      }
  ],
  "field_ids": [
      "test_device_2__count__bool_switch_state",
      "test_device_2__mean__float_temp_c"
  ],
  "fields": {
      "test_device_2__count__bool_switch_state": {
      "is_numeric": false,
      "dtype": "bool",
      "is_string": false,
      "id": "bool_switch_state",
      "name": "Unregistered: Count Switch",
      "is_boolean": true,
      "unit": "state"
      },
      "test_device_2__mean__float_temp_c": {
      "is_numeric": true,
      "dtype": "float",
      "is_string": false,
      "id": "float_temp_c",
      "name": "Unregistered: Mean Temp",
      "is_boolean": false,
      "unit": "c"
      }
  },
  "end": "2018-03-20 13:35:00+00:00",
  "start": "2018-03-20 13:30:00+00:00"
}
```

Query Parameters

- **`fill`** – Fill mode specifies what to do if there is no data for an interval. If set to "null" then an interval reading will be returned but with a null value. If "none" then no reading will be returned for the interval.
- **`interval`** – Interval must be an integer of seconds. The results will be grouped into intervals and the aggregation function applied.
- **`start`** – Start of reading period (inclusive)
- **`end`** – End of readings period (inclusive)

- **`limit`** – Limit the number of results
- **`format`** – Results format. Default is as a list. by_time = dict where key is timestamp and value is a dict of field:value pairs
- **`metrics[]`** – &metric[]=device_is,field_id,aggregation_function can use multiple &metric[]'s in call
- **`device_id`** – Optional device id (replaces need for metric)

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – query successful found
- 404 Not Found – device not found
- 400 Bad Request – invalid metrics

2.5 Design Notes

The server employs many technologies. Below we list the most prominant and link to their respective websites. If all of this is a bit confusing then don't worry, we only mention them here as a reference for anyone who is interested.

- InfluxDb (https://www.influxdata.com/) to store and inspect time series sensor data
- TinyDb (http://tinydb.readthedocs.io/) to store general information and settings.
- Flask (http://flask.pocoo.org/) to create the web interface
- Gunicorn (http://gunicorn.org/) and Nginx (https://www.nginx.com/) to serve the web interface

CHAPTER 3

About the code

The project code is designed for novice user to deploy and for novice programmers (e.g. academic researchers) to modify. As such we have chosen to use technologies which make the reading and adapting of the code less difficult, for example, we chose to use jQuery over more modern technologies such as React.js because the structure is more intuitive and there are fewer idependant technologies. In addition we have structured the project so that all relevant CSS and Javascript is contained within the tempalates. We chose to defy the convention of splitting code into seporate files so that it is obvious what code affects what and to maximise decoupling, thus minimising the chance of changes having unexpected consiquences elsewhere.

All the code you need can be found at: https://github.com/jkittley/LittleSense

CHAPTER 4

Why Little Sense

My mother always said should have a little sense and listen.

CHAPTER 5

Contribute

- Issue Tracker: https://github.com/jkittley/LittleSense/issues
- Source Code: https://github.com/jkittley/LittleSense

CHAPTER 6

Support

If you are having issues, please let us know by submitting an issue.

CHAPTER 7

License

The project is licensed under the BSD license.

CHAPTER 8

Index

- genindex

HTTP Routing Table

/api

Python Module Index

Index