Senslify Developer Guide

By Christen Ford

Version 1.0.0

Table of Contents

[1.0 Introduction and Background 4](#_Toc75353935)

[1.1 Introduction 4](#_Toc75353936)

[1.2 Objectives 5](#_Toc75353937)

[1.3 Risks 6](#_Toc75353938)

[2.0 Architecture 7](#_Toc75353939)

[2.1 Software Architecture 7](#_Toc75353940)

[2.2 Database Design 8](#_Toc75353941)

[2.3 Class Level Design 11](#_Toc75353942)

[2.3.1 \_\_init\_\_.py 12](#_Toc75353943)

[2.3.2 db.py 13](#_Toc75353944)

[2.3.3 errors.py 13](#_Toc75353945)

[2.3.4 filters.py 13](#_Toc75353946)

[2.3.5 index.py 13](#_Toc75353947)

[2.3.6 rest.py 14](#_Toc75353948)

[2.3.7 sensors.py 14](#_Toc75353949)

[2.3.8 sockets.py 14](#_Toc75353950)

[2.3.9 verify.py 14](#_Toc75353951)

[2.3.10 templates/base.jinja2 14](#_Toc75353952)

[2.3.11 templates/sensors/index.jinja2 15](#_Toc75353953)

[2.3.12 templates/sensors/info.jinja2 15](#_Toc75353954)

[2.4.13 templates/sensors/sensors.jinja2 15](#_Toc75353955)

[2.3.14 tools/btlemon.py 16](#_Toc75353956)

[2.3.15 tools/xlsx2tsv.py 16](#_Toc75353957)

[2.4 Software Interfaces 16](#_Toc75353958)

[3.0 Conclusion 18](#_Toc75353959)

[3.1 Remarks on Implementation 18](#_Toc75353960)

[3.2 Possible Future Improvements 19](#_Toc75353961)

# Introduction and Background

## 1.1 Introduction

Senslify is sensor reading visualization software. Senslify was created as part of a sponsored project from the IOT Collaborative between Case Western Reserve University (CWRU) and Cleveland State University (CSU) and involved Dr. Phillip Feng and Devendra Waikul from CWRU and Dr. Haodang Wang and Christen Ford from CSU. This software was motivated by a need for Devendra to have a way to visualize sensor data in real-time for his Master’s thesis. The overall goal of the Masters thesis was to produce Bluetooth low energy wireless sensor devices that could remotely send and receive information to a wireless gateway. Such devices already existed but they did not leverage the Bluetooth low energy protocol and therefore consumed more power than necessary. This gateway would then send and receive messages to the Senslify software stack which would store the content of these messages (sensor readings) and broadcast them to connected clients so they could view them in real-time.

While the overall implementation of Senslify is only ~3000 LOC, much thought went into its analysis and design with those two tasks taking up the bulk of the time allotted to the project. To support scalability such that Senslify can be used in the setting it was intended (metropolitan areas, county-wide, and even state-wide), it was determined early on in the requirements engineering process that the entire system must be asynchronous. To that end, I had to find or design an asynchronous software stack that supported the goals of the project. The software stack that comprises the Senslify system is detailed in the Software Interfaces section (section 2.4).

Senslify was designed for several different types of users: a normal web browser user, a sensor/gateway, and a non-traditional HTTP API. Most users of the Senslify system will either be accessing the system through a web browser (to view, download, or get the stats for sensor data), or they will be accessing the system through a sensor or gateway to upload sensor data. A minority of users may access the system through the REST API designed into the system via a non-traditional HTTP client or API. In this case, this type of user only has read access to the information stored by Senslify but has additional commands that may still prove useful that are not currently integrated into the web browser interface (such as getting statistics on groups of sensors).

The purpose of this developer guide is to provide maintainers of the Senslify software with a unified location to lookup important design information regarding Senslify. This developer guide includes information regarding the architecture of Senslify, a maintenance guide, and remarks on the implementation as well as suggestions for future improvement of the software. If at any point, this guide fails to provide information needed by the maintainer, the reader is free to utilize any of the other major deliverables of the Senslify software that provide more fine-grained details as to the design and implementation of the software.

## 1.2 Objectives

As stated earlier, Senslify is a sensor reading visualization tool. Senslify is real-time. When a sensor or gateway uploads sensor data to the software, this data is simultaneously uploaded to the Senslify database and is additionally broadcast to clients. Senlsify explicitly provides the following functionality:

* Users can interact with sensor data stored or received by the system including:
  + Receiving and viewing sensor data in real-time via the chart and reading list.
  + Requesting sensor reading downloads for a specific period of time.
  + Requesting sensor statistics for a specific period of time.
  + Be notified when a sensor reading is received that is outside the normal range of sensor readings for that sensor.
* Sensors/gateways can upload sensor readings to the system.
* Users can add groups and sensors to the system.
* APIs/HTTP clients can leverage the REST API to accomplish the following:
  + Download sensor readings for a given period of time.
  + Request sensor statistics for a single sensor for a specific period of time.
  + Request sensor statistics for a group of sensors for a given period of time.
* Web server administrators can configure various options within the system itself either via a configuration file or through system initialization including:
  + The number of attempts a client can attempt to establish a web socket connection.
  + Whether debugging is enabled or not.
  + The maximum number of readings displayed on user web browser clients.
  + The version of Bootstrap, chartjs, and jquery to use.
  + The locale used for date formatting.
  + The port of the server.
  + The connection string used by the NoSQL database.
  + The username and password used to connect to the NoSQL database.
  + Whether to continue using the existing NoSQL database or to create a new database.
  + The default reading type users are subscribed to when they view a sensor.

## 1.3 Risks

This section details the risks faced by the current implementation of the software. The goal is to detail potential issues with the software and potential risks it faces.

* The web server is implemented using the Python ‘aiohttp’ package which largely depends on the Python ‘asyncio’ package. While making the web server and corresponding client code entirely asynchronous greatly enables scalability, it comes with a few drawbacks.
  + `asyncio` is not at stable release, it is still in beta. This means serious portions of the API can change in effect causing software that depends on it to need to be rewritten. This should not be a problem for Senslify as Senslify uses the `aiohttp` API as the backend for the web server, which is stable release software. While `aiohttp` may need to internally change to account for changes in `asyncio`, these changes should not change the outward facing portions of the `aiohttp` api.
  + Async/Await in Javascript is an evolving standard. Code written utilizing it one day may not work years, months, or even weeks down the line. While Mozilla and the other major Internet companies working with ECMA International on the Javascript standard are doing their best to ensure compatibility with older version of Async/Await, there is always the possibility that future versions of Javascript may break the client side protocol implementation. Future maintainers of the client side code therefore need to stay up-to-date with changes made to these specific features of the Javascript language.
* The web server temporarily caches results for sensors, groups, readings, etc… from the database before passing them to the jinja2 templating engine. This may not pose an issue for smaller datasets within the Senslify system, but as the database grows and more sensors are added to the system, the runtime memory consumption of the current model will likely grow exponentially w.r.t. the number of sensors/gateways sending data to the system.
  + This is because jinja2 templates do not (without experimental code) support the consumption of async generator functions (which is what all of the function calls in the database layer return when you ask them for data). To account for this without introducing experimental and potentially dangerous code to the jinja2 templating engine, I instead wrote the intermediary code that generates the template parameters in such a way that entire generator is consumed into a single template parameter (a Python list – hence where the memory consumption comes from) that is then passed to the template.
  + Until jinja2 templates support consumption of async generator, it is probably in the best interest of the server administrator to make it so that data is paginated from the database instead (at least for large amounts of data). This feature was not implemented into the system but is often times used in situations such as this to prevent runtime memory consumption explosion.

# 2.0 Architecture

## 2.1 Software Architecture

Like many of the projects I have worked on in the past, Senslify uses a layered architecture. The architecture diagram for Senslify is shown below. Note that Senslify was defined as a API-first system. Any data access to the system that falls outside of using the web browser to view the information stored by the system is mediated either through a REST API or through several well-defined protocols (see the Design Specification for the specifics regarding these protocols).

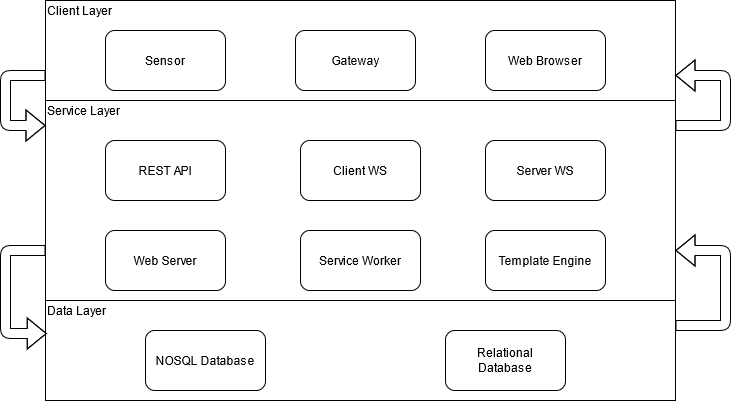


Figure 1: Senslify System Architecture

The Client Layer is responsible for allowing the user to interact with the system. This layer is outside of the control of the system and as such all components defined within this system are considered **external** to the rest of the system. Their access to the rest of the system is mediated through the Service Layer. The Service Worker is designed to be spun off by the Web Server once the Web Server is initialized (though this did not make it into the final implementation per the clients decision). The Template Engine is utilized by the Web Server to dynamically generate web pages to return to the user. The Service Layer then interacts directly with Data Layer to store and retrieve sensor readings, statistics, groups and sensors.

## 2.2 Database Design

Senslify was designed to utilize two databases. Even though the initial version of the software only utilizes the NoSQL database at the behest of the client, several of the core features were designed around using a secondary RDBMS as permanent storage for older sensor readings. The design of the NoSQL and RDBMS databases adhere to the following schema. Note that in the case of the RDBMS database, the schema only consists of the Readings table. The other three tables are stored exclusively in the NoSQL database as the RDBMS is meant to store sensor readings long-term as an archival database while the NoSQL database holds more recent (an thus, more likely to be accessed) readings.

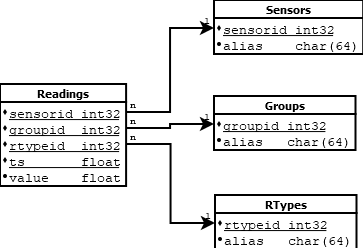


Figure 2: Senslify Entity Relationship Diagram

A similar design schema can be utilized if future maintainers of the software want to shift away from using NoSQL entirely. The reason NoSQL is utilized in the current iteration of the Senslify system was to ensure that many different types of sensors (which could foreseeably produce different data schemas) did not have to restrict the way they format their data to adhere to the design of the database. This means that the overall implementation of the Senslify system had to be more complex to accommodate for it, but it allowed for a much more diverse range of sensors to be deployed and be compatible with the system. Following is a description of the tables presented in the ERD as well as their relations.

|  |  |  |
| --- | --- | --- |
| **Description of the Tables Present in the Senslify ERD** | | |
| **Table Name** | **Related To** | **Description** |
| Groups | N/A | Holds sensor group information including the group identifier and the plain-English alias for each group |
| Readings | Groups, RTypes, Sensors | Holds sensor reading information including the sensor identifier, group identifier, reading type identifier, timestamp, and value. This table is related to the Groups table via the group identifier, the RTypes table via the reading type identifier, and the Sensors table via the sensor identifier. |
| RTypes | N/A | Holds reading type information including the reading type identifier and the plain-English alias for each group. |
| Sensors | N/A | Holds sensor information including the sensor identifier and the plain-English alias for each sensor. |

Table 1: Table Descriptions from ERD

Following is a presentation of Senslify’s data dictionary:

|  |  |  |
| --- | --- | --- |
| **Data Dictionary for Senslify** | | |
| **Name** | **Data Type** | **Description** |
| alias | 64-byte character array | The plain-English alias for the associated identifier. |
| groupid | 32-bit positive integer | The group identifier used in conjunction with the sensor identifier that is used to uniquely identify a sensor. The group identifier uniquely identifies a group of sensors. |
| rtypeid | 32-bit positive integer | The reading type identifier that identifies what type of measurement was taken by a sensors. This identifier is always associated with a timestamp (ts) and value (val). |
| sensorid | 32-bit positive integer | The sensor identifier used in conjunction with the group identifier that is sued to uniquely identify a sensor. The sensor identifier uniquely identifies a sensor within a group of sensors. |
| ts | 64-bit positive integer | The UNIX timestamp representing when the timestamp was taken. |
| val | 64-bit double precision floating point | The value of the sensor reading taken by the senor. |

Table 2: Data Dictionary for Senslify

## 2.3 Class Level Design

Senslify uses several classes to do its job. The initial class diagram is shown below. This class diagram has not been updated to reflect future iterations of the software but can still be consulted as a starting point if any problems arise with the software.

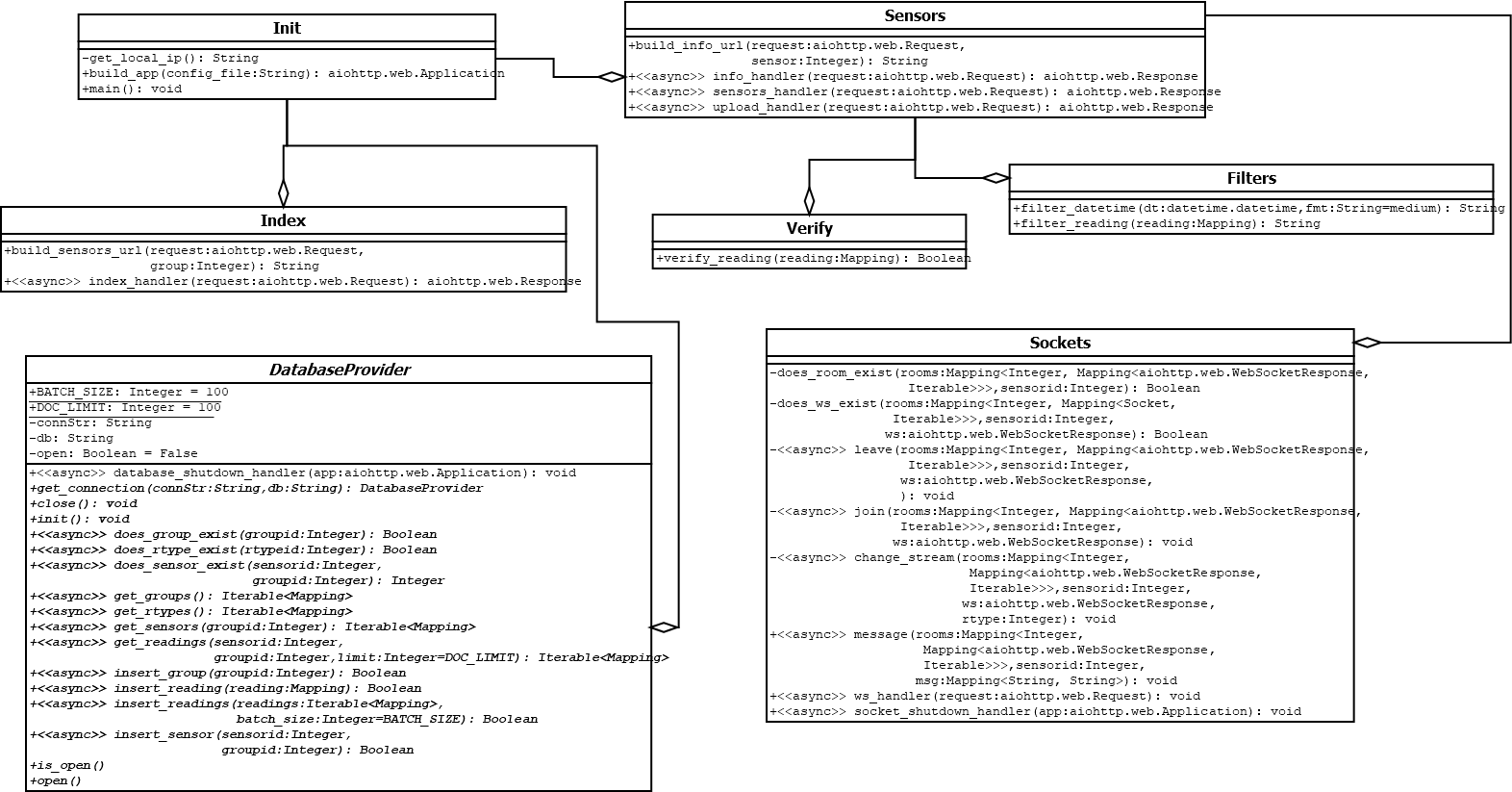


Figure 3: Senslify Class Diagram

Note that his diagram is not comprehensive. There are several functions missing from it, that were since added in future iterations of the software. One important facet of this implementation is that almost every function implemented as part of the Senslify software package is asynchronous. This was done to ensure scalability to eventually county-wide and state-wide sensor deployments. This greatly increases the complexity of the software and future maintainers must be aware of how the asynchronous paradigm works, especially as it relates to the Python programming language (the main language used to implement the Senslify system). Note that each module in the Senslify system has a single, well-defined purpose and set of responsibilities. No single module has more than one purpose and all related functionality to serving that single purpose is implemented entirely in that module. This should make debugging or adding additional features to the system far easier for future maintainers of the software.

### 2.3.1 \_\_init\_\_.py

This module contains the code needed to initialize the web server. It is responsible for properly initialize external software interfaces, setting up routes, reading the configuration file, opening the database connections, as well as starting the web server. When the user types `senslify` from the command line, the main() function in this file is what will run as a result.

### 2.3.2 db.py

This module contains all of the database code needed to allow the system to interact with the NoSQL database. It also defines a generic wrapper that future maintainers should extend and implement to provide access to other databases. Senslify utilizes the MongoDB database as the NoSQL component. As a reminder, the RDBMS features of Senslify were not implemented per a discussion with the client. Utilizing the generic adapter provided by this class, it should be trivial to do so, then integrate it with the rest of the system.

### 2.3.3 errors.py

This module contains definitions for custom exceptions that the system might raise under operation. This module also contains functions for generating errors as well as generating traceback strings for debugging purposes. Note that the web server administrator must specify that debugging is turned on in the web servers configuration file in order to see detailed debugging information through the web interface, otherwise, maintainers and testers will only see the generic error messages that normal system users see.

### 2.3.4 filters.py

This module contains functions for formatting dates, timestamps and sensor readings. Date and timestamp formatting is ISO compliant. These functions are used by the web server to render web pages where their respectively formatted components are in a human readable and well-understood format.

### 2.3.5 index.py

This module contains functions for generating and rendering the home page. This is the page first visited by a user of the Senslify web application. This web page contains a listing of all of the groups currently in the Senslify system that have at least one sensor registered with them. Note that due to async compatibility issues with jinja2, it is (and still may be) impossible to pass async generator functions directly to jinja2 templates. This means that all groups in the database are aggregated on the fly in the template generation function, temporarily consuming more RAM than necessary. Because each group is represented by a 32-bit positive integer, this should not be an issue, even for a large number of groups, but future maintainers should be aware that this is a point for refactoring once jinja2 allows templates to consume async generator functions.

### 2.3.6 rest.py

This module contains functions that implement the REST API. This API is used by non-traditional HTTP clients that want to request information from the system. This API allows users to request sensors, groups, sensor readings, sensors statistics for a single sensor or sensor group, and reading types. This API always returns data in the format of a stringified JSON object.

### 2.3.7 sensors.py

This module contains functions that generate and render the sensor listing and sensor information pages. It also defines functions that allow sensors/gateways to upload sensor readings.

### 2.3.8 sockets.py

This module contains all functionality related to the server-side handling of the web socket connection. It implements the server-side web socket-based protocol as defined in the design documentation and is the heart of the real-time environment maintained via the web socket connection. Any issues with the sensor information page can be traced directly back to this module, as such it si the most thoroughly tested module in the entire system. Adding additional commands to the web socket protocol is simple, however before doing so, the maintainer must ensure the command is well-defined, makes sense for the system, and is documented in the code base as well as the System Design Specification. I have included the draw.io documents used to generate the protocol state diagrams with this documentation.

### 2.3.9 verify.py

This module contains functions that are used to validate inputs sent to the various REST and web socket endpoints.

### 2.3.10 templates/base.jinja2

This template defines the base layout for the entire Senslify web application. Changes made to this template will be reflected in every page across the entire web application. There are jinja2 blocks for head, header, body, and footer content defined by this template that can be overridden in derived templates. If maintainers want to add additional functionality or web pages to the web application, please be aware of the blocks defined by this template and how to utilize them in your derived templates.

### 2.3.11 templates/sensors/index.jinja2

This template overrides the base.jinja2 template and acts as the template for the web applications home page. This template is rendered by the template engine after all groups have been retrieved from the database and is the first web page the user sees when they visit the web application.

### 2.3.12 templates/sensors/info.jinja2

This template overrides the base.jinja2 template and acts as the template for the web applications sensor information page. This template contains a fair deal of Javascript code that implements the client-side portion of the web socket protocol. It also provides Javascript code that automatically displays received error messages and automatically clears them after a delay. Likewise, when the web socket receives a reading via a server broadcast, the Javascript defined by this page will automatically render it in the appropriate HTML controls, modifying the controls as necessary. This is the most complicated template for the client-side implementation. Any errors that occur on the sensor information page can be traced directly back to this template. Note that debugging and testing this template is more arduous then debugging its partner code on the web server. Future maintainers therefore need to exercise caution when changing this template.

### 2.4.13 templates/sensors/sensors.jinja2

This template overrides the base.jinja2 template and acts as the template for the web applications sensor listing page. This template is rendered by the template engine after all the sensors for the user-supplied group have been retrieved from the database and is usually the second web page the user sees when they visit the web application. Note that if the user knows the group identifier for the sensor group they want to view as well as the proper format for the HTML request, they can skip the home page and view this page directly by properly formatting the URL for this template in their web browser.

### 2.3.14 tools/btlemon.py

This tool partially implements a Bluetooth low-energy client that listens for updates from a Bluetooth low-energy device and automatically retransmits these readings to the web server using the REST API. During development, Devendra and I had considered contracting an organization at CWRU that specializes in designing wireless gateways such as what would be used by this system. Their response indicated that it would take too long and we would not have full control over the gateway itself. These factors are what led to the design and partial implementation of this tool. The implementation was never actually finished, because Devendra just utilized the Postman software package to upload sensor readings using the API. This tool may or may not be worth completing by future maintainers and should be given low priority.

### 2.3.15 tools/xlsx2tsv.py

This tool provides functionality that will convert a modern Microsoft Excel workbook (\*.xlsx extension) to a Tab-Separated Values file. TSV was used instead of the more common CSV extension because despite being an IETF standard, CSV is not properly implemented in some major database systems (such as MS SQL Server). On the other hand, TSV has 100% compatibility with all major database systems. This tool should be utilized if the user has an Excel file containing sensor readings or other information that needs loaded into the Senslify database. This tool will automatically convert your Excel file into a format that is easily uploaded (using other, database-provided tools) to the Senslify database. This tool should require little to no maintenance as the software libraries it depends on are unlikely to change in the future.

## 2.4 Software Interfaces

Senslify is a complex system of systems and utilizes many different subsystems to accomplish its goals. Senslify is built in Python3 and will not run on Python2 since `asyncio` does not exist in Python2 (at least in the state needed by Senslify). This software utilizes the following Python packages (installable from pip using setup.py or requirements.txt).

* aiodns – an `asyncio` compatible DNS resolver.
* aiohttp – a fully asynchronous http client/server framework.
* aiohttp-jinja2 – the official `jinja2` templating engine extension for `aiohttp`.
* cchardet – a high speed universal character encoding detector, binding for `uchardet`.
* click – alternative library to the Python standard library `argparse`, provides additional functionality on-top of `argparse`.
* gevent – high level networking library that provides a synchronous API on top of `asyncio` through `libev` or `libuv`, provides asynchronous compatiblity for the `pymongo` driver.
* pymongo – the official Python MongoDB driver. An alternative to `pymongo` that is actually `asyncio` compatible would be a library like `Tornado`.
* sphinx – automated documentation generation from docstrings, like `doxygen`.
* simplejson – extension library built on top of the Python standard JSON library, offers additional functionality on top of the standard library.

Likewise, the client also utilizes several JS and CSS libraries to perform it’s job. Note that all of these libraries are pulled down via CDN. They are not served to the client from the web server. These are listed here.

* Bootstrap – industry-standard CSS theming library developed by Twitter and released under an open-source license.
* Bootstrap-JS – the official Javacscript support libraries for the Bootstrap CSS theming library.
* ChartJS – open-source CSS/JS library that provides an API for rendering charts, graphs, plots etc… in the browser.
* ChartJS-Streaming – extension plugin for ChartJS that enables real-time streaming to ChartJS objects.
* DownloadJS – JS library that provides a cleaner interface for downloading data from a remote URL.
* JQuery – an extensive JS library that provides wrappers and additional functions on top of the DOM JS library.
* Moment.js – JS library that provides mechanisms for working with time and dates.
* Popper.js – JS/CSS library that provides enhanced tooltips and contextual popups via various DOM events.

Next, the tools section utilizes several dependencies that need installed manually via PIP or through a tool like anaconda. These are listed here.

* bluepy – Python library for working with Bluetooth Low-Energy devices that is based on the Bluez protocol.
* openpyxl – Python library for working with Excel documents from within Python.
* openpyxl\_utilities – extension library for openpyxl that adds additional utility functions to the library.

Lastly, Senslify itself utilizes several external software packages. They are listed here.

* MongoDB – A NoSQL collection-oriented database.
* Python3 – The third iteration of the Python programming language.

# 3.0 Conclusion

## 3.1 Remarks on Implementation

Senslify implements a layered architecture as seen in an earlier section. Each Python module in the accompanying source code has exactly one set of related responsibilities. There are clearly defined boundaries between these modules, and I have made every attempt to make sure each module adheres to that design. The most complicated facets of this software are that (1) it is fully asynchronous and (2) it utilizes the WebSocket API on the client and the WebSocket implementation provided by `aiohttp`. The first point makes debugging and testing Senslify quite difficult, however it greatly increases scalability. The second point directs Senslify to the future with a simple-to-use API.

The newness of this API though might cause some concerns for maintainers or developers that are not used to working with WebSockets. To that end, I point them to the IETF RFC 6455 document that details the underlying mechanisms that make up the WebSocket protocol. Both the client side API and the sever side API are bare-bones. They do not make any attempts to provide additional functionality on top of what is defined in the aforementioned RFC. As such, features implemented on the server such as channels/rooms for sensors were designed and implemented by hand. It may be suitable to probe available software packages on PyPi for a suitable extension library that provides the same functionality. While the code provided for the channel/room management is fully-tested and full-functional, I am certain there are better ways to handle it while adhering to the protocol I designed for the client/server WebSocket interaction.

Speaking of which, I tested the WebSocket connection between the client and server for a whole (3) days in a laboratory environment. During which, I had a sensor repeatedly upload measurements to the server one a one (1) minute interval. The client never crashed or failed to receive a measurement, and the server logged each of them. The implementation of the uploading use case should work perfectly for gateways based on my testing, however it is untested in this environment. Though I suspect it will work the same way as with a sensor. Likewise, a planned feature was to be able to allow the server to receive batched uploads and handle this accordingly, however that did not make it into the final version of the implementation.

## 3.2 Possible Future Improvements

I have identified the following possible future improvements to the Senslify software.

1. Refactor the design of the group/sensor listing page, it is quite basic. At the very least, make it so that group/sensor aliases are displayed instead of raw identifiers.
2. Enhance an existing protocol or design a new protocol so groups/sensors can be automatically provisioned into the system as they interact with the system. This should already be happening informally, but a formal mechanism for it would be nice to have.
3. Test this implementation in an environment with at least 50 sensors. My laboratory tests were limited to only 5 sensors and the 50 sensor test is a hard requirement for the software to test scalability that was ignored for this implementation.
4. Refine the download functionality such that the user can pause/resume and cancel the download. Currently, the download functionality only lets the user initiate a download then wait for it’s completion. An easier way to do this would be to generate the readings file on the server, cache it for a limited amount of time in case others want to download it, then serve it using the official browser.downloads API so the browser can handle all of these features instead of the client API.
5. Implement the migration service worker. This requires two separate database to be running and reachable to function correctly. The NoSQL database (which stores recent readings) and the RDBMS (which stores migrated readings). This feature was ignored in this implementation because Devendra did not have a powerful enough computer to run the web server, MongoDB, and an RDBMS at the same time during his presentation of the software to his Masters committee. This could be house-built service that exists alongside the web server (as it is quite simple to implement) or it could be an external service provided by something such as RabbitMQ. Note that this does require implementing a second database API for the RDBMS component as this implementation only ships with the generic adapter and a MongoDB implementation of that adapter. Implementing the database API for and RDBMS component would be the most time-consuming aspect of this improvement but shouldn’t be difficult as the existing documentation in db.py is more than sufficient to guide the implementor.
6. While not explicitly stated in the requirements specification, Devendra talked about wanting the index page (the page that contains group listings) and the sensor listings page to contain information on groups and sensors at a glance. This would include just overview information such as how many groups there are or how many sensors there are in a group, how many sensors are online/offline, if there are any problems with any groups or sensors, etc… I could not reasonably accommodate this in the time I had to complete this project, but I would also like to see at least some of these features implemented.
7. There also needs to be a way to mediate who can ultimately create groups and sensors in the database. I think this mandates some sort of authentication. If the system had a way to create users, we could associate groups and sensors with users and only show each user the groups and sensors that are associated with their account. This should deal with several major security issues that exist in the current design. During the sensor provisioning process, you could just generate a token that’s a combination of the user identifier and group/sensor identifier to uniquely identify readings from that sensor (preventing issues such as reading forgery or sensor masquerading).
8. The system is already designed with security in mind for the database. In fact, when you start the system, it will ask you if there is a username/password required to access the database and if so you will be required to enter it. Currently though, these credentials are stored inside the DatabaseProvider object in plain text. Technically, you should only need the database credentials when initializing the database as the database connection should never be closed once the system is running. The better solution here would be to write them to a temporary secrets file that exists per the life of the web server and read them from there as necessary. There is name mangling that is performed on the username and password references so anyone performing memory attacks on the web server should have a harder time determining what fields the username and password are, but this is still a security risk that is best avoided with better design.