System Design Specification

Senslify

By: Christen Ford

V 1.0.0

Table of Contents

[Introduction 3](#_Toc75229523)

[Architecture Diagram 3](#_Toc75229524)

[Activity Diagrams 3](#_Toc75229525)

[Client/Server Diagrams 4](#_Toc75229526)

[Gateway/Server Diagrams 9](#_Toc75229527)

[Sequence Diagrams 11](#_Toc75229528)

[State Diagrams 20](#_Toc75229529)

[Client/Server Protocol 20](#_Toc75229530)

[Gateway/Server Protocol 28](#_Toc75229531)

[Structure Diagrams 31](#_Toc75229532)

[Wireframe Diagrams 32](#_Toc75229533)

# Introduction

This document details the UML diagrams that compose the design of the Senslify web application. This document will attempt to be as thorough as possible but due to the agile nature of the project, it may not reflect the entirety of the actualized design as implemented by the project members.

# Architecture Diagram

This Architecture Diagram section details the overall architecture of the Senslify system. The system itself utilizes a layered design where each layer is solely responsible for the actions of it’s own layer. Furthermore, each layer receives messages from the layer above and sends messages to the layer below it. Two non-consecutive layers never directly interact with each other.

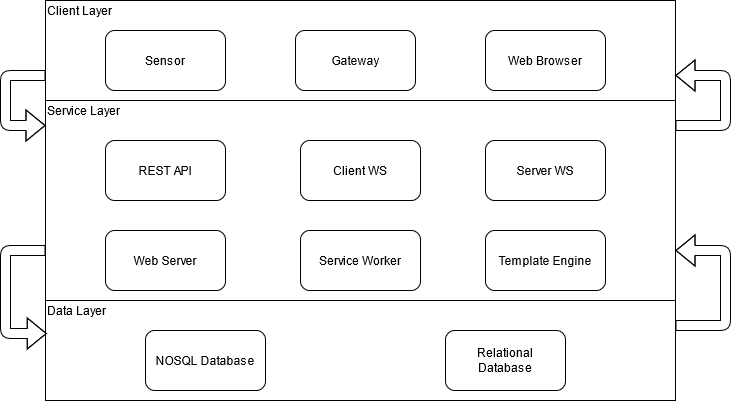


Figure : Senslify System Architecture

# Activity Diagrams

The Activity Diagrams section details all of the UML activity diagrams that detail parameterized behaviors as a coordinated flow of actions. Per the official UML documentation

The **flow of execution** is modeled as activity nodes connected by activity edges. A node can be the execution of a subordinate behavior, such as an arithmetic computation, a call to an operation, or manipulation of object contents. Activity nodes also include flow of control constructs, such as synchronization, decision, and concurrency control. Activities may form invocation hierarchies invoking other activities, ultimately resolving to individual actions. In an object-oriented model, activities are usually invoked indirectly as methods bound to operations that are directly invoked.

Each activity is composed of activity nodes which could represent an action, control, or object. Actions within an activity may represent occurrences of primitive functions, invocations of behavior, communication actions, or manipulation of objects. Furthermore, each activity may be called directly or indirectly by another activity via a call operation action. Behaviors may have pre-conditions that indicate conditions necessary for the action to take place as well as post-conditions that indicate what state(s) the activity could be in after the behavior has completed executing.

## Client/Server Diagrams

This section contains the activity diagrams detailing the interactions between the client and server. These diagrams utilize swimlanes and <<external>> tags to make it clear what component responsibilities are as well as what the system is not responsible for.

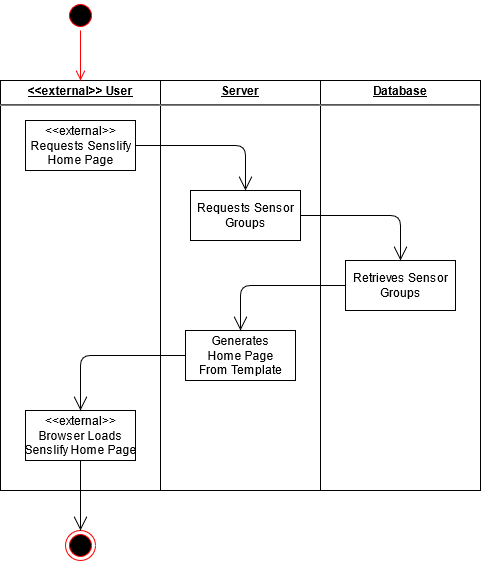


Figure : Requesting the Home Page

This activity diagram details the sequence of activities the system follows in response to an external user making a request to the index (or home page) url for the Senslify web application. The activity sequence is initialized by the user requesting the home page. This request is received by the server which then requests all sensor groups from the database. Once the database returns the list of sensor groups to the server, the HTML document for the home page is generated by the server from a template. This document is then sent in response to the users request and is then rendered by the users web browser upon arrival.

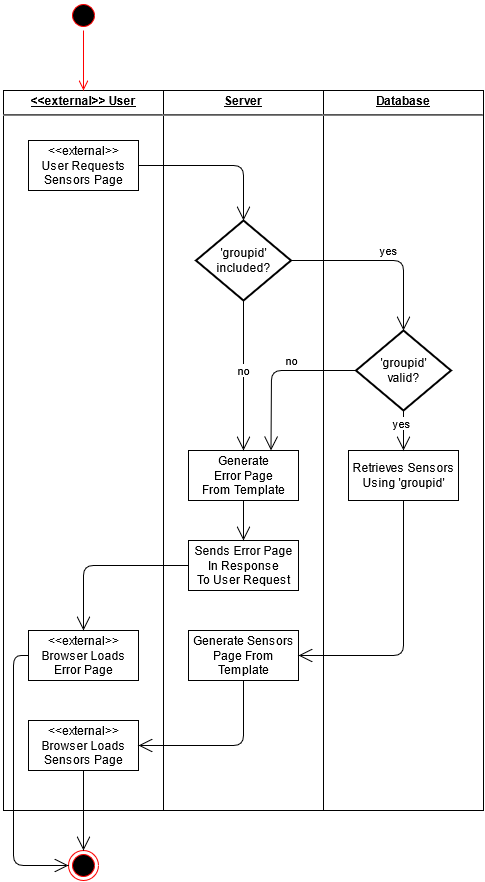


Figure : Requesting the Sensor Listings Page

This diagram details the sequence of activities the system follows when receiving a user request for the sensor listing page. The server will first validate that the user included a group identifier in their request. If the group identifier is included, the server will pass the group identifier to the database which will determine if it is valid or not. Otherwise, if the group identifier was not included in the request, the server will generate an HTML error page and send that in a response to the user to be rendered by their browser. If the database determines the group identifier is valid, it will then generate the list of sensor identifiers corresponding to the group identifier and return that to the server. Otherwise, if the group identifier is not valid, the database will notify the server which will generate an HTML error page and send that in a response to the user to be rendered by their browser. Upon receiving the sensor identifiers from the database, the server will generate the HTML sensors page from a template and send that in a response to the user to be rendered by their browser.

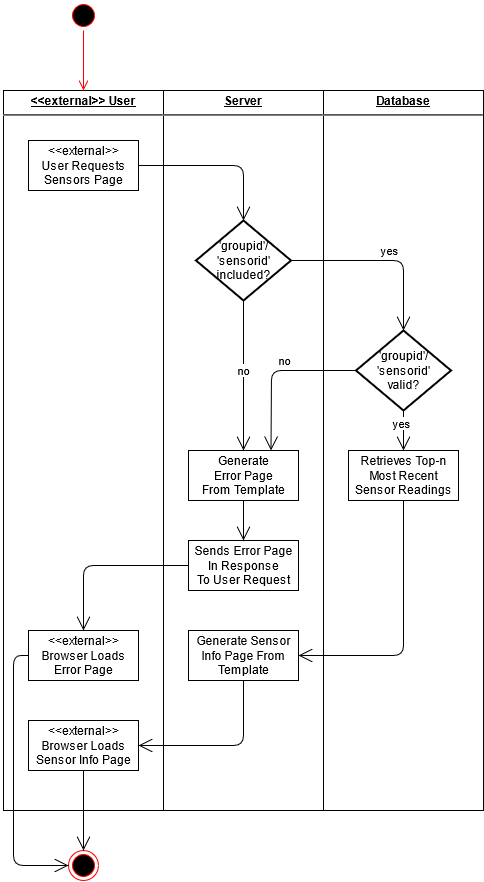


Figure : Requesting the Sensor Information Page

This diagram details the sequence of activities the system follows when receiving a user request for the sensor information page. The server will first validate that the user included a group identifier and sensor identifier in their request. If the group identifier and sensor identifier are both included, the server will pass the group identifier and sensor identifier to the database which will determine if they are both valid. Otherwise, if the group identifier or sensor identifier were not included in the request, the server will generate an HTML error page and send that in a response to the user to be rendered by their browser. If the database determines the group identifier and sensor identifier are both valid, it will then generate a list containing the top-n most recent sensor readings corresponding to the group identifier and sensor identifier and return this list to the server. Otherwise, if the group identifier or sensor identifier are not valid, the database will notify the server which will generate an HTML error page and send that in a response to the user to be rendered by their browser. Upon receiving the sensor readings from the database, the server will generate the HTML sensors information page from a template and send that in a response to the user to be rendered by their browser.

## Gateway/Server Diagrams

This section contains the activity diagrams that detail the interactions between the gateway (or other HTTP-enabled devices) and the web server. These diagrams utilize swimlanes and <<external>> tags to make it clear what component responsibilities are as well as what the system is not responsible for.

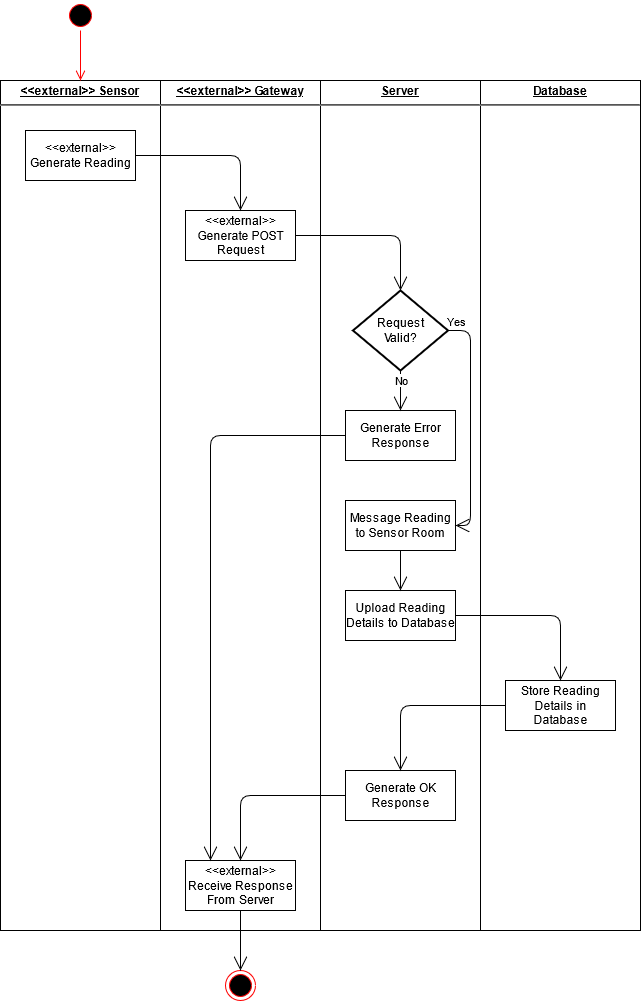


Figure : Uploading a Sensor Reading

This diagram details the sequence of activities the system undertakes when a gateway (or other HTTP-enabled device) uploads a sensor reading to the web server. This diagram showcases the typical use case whereby sensor data is first sent to an HTTP-enabled gateway. The system works equally well if data is uploaded directly from an HTTP-enable sensor platform. In both cases, this activity is initialized when a sensor generates a reading. This reading is then sent to the gateway which bundles the sensor reading into a HTTP POST request. This request is posted to the server using a well-known and defined route. Upon receiving a POST request, the server will validate the request ensuring that (1) the request contains a JSON object inside it’s body and (2) the JSON object contains the required fields.

Note: It is important to note that with this design, the server does not perform any validation of input data. It is assumed that the external gateway or sensor will validate/verify the input data before it is sent to the server. Accommodating this feature into system with such dynamic data will only serve to slow the web server down.

If the request is invalid, the server will generate an HTML error response with status code 403 and send this back to the gateway for processing. Otherwise, the reading is sent to the sensor room corresponding to the sensor and group the reading originated from. The sensor reading is then sent to the database where it records the reading as well as the miscellaneous information that came with it. Lastly, the server generates an HTTP OK response with status code 200. This response is sent to the gateway for processing.

# Sequence Diagrams

This section contains the sequence diagrams that detail the flow of messages and responses between the various components of the system. In some cases, these messages are just plain English phrases to represent actions that a component would take at that time period in the sequence diagram. There are also messages that have been genericized as they are part of components that are external to the system. Note that it was the decision of the team to plan for integration of the RDBMS and features that depend on it (i.e. migration/downloading). As such, it is incorporated into the following sequence diagrams, however the final system that results from this project will not actually utilize it. The features that use the RDBMS should be easy enough to integrate into the system by future maintainers will little effort as the system itself was designed to be modularly self-contained.

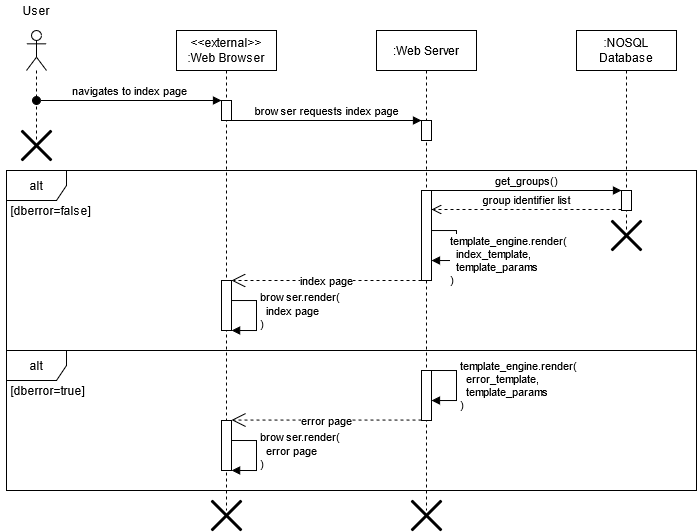


Figure : Viewing Groups

This sequence diagram shows the series of messages and responses the various components in the system make to allow the user to view the home page.

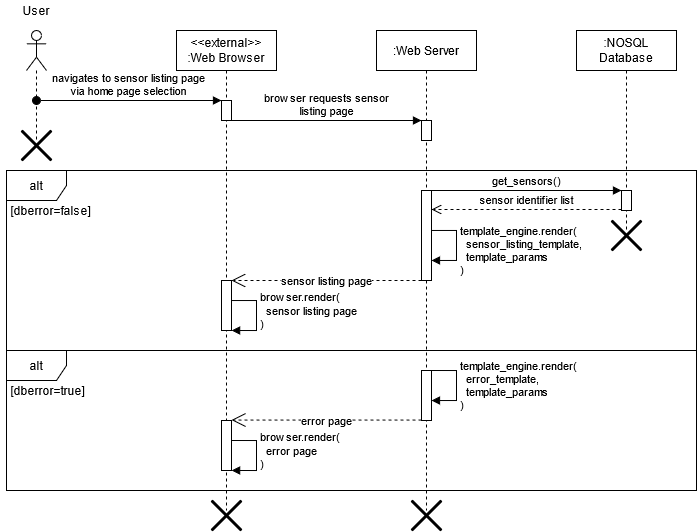


Figure : Viewing Sensors

This sequence diagram shows the series of messages and responses the various components in the system make to allow the user to view the sensors listing page for a specific group of sensors.

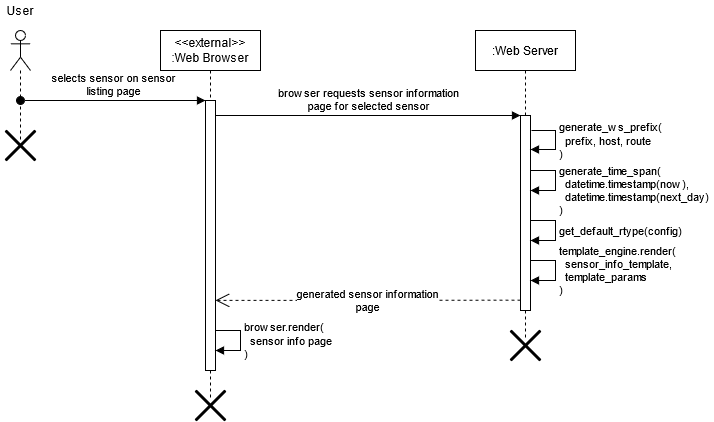


Figure : Viewing Sensor Information

This sequence diagram shows the series of messages and responses the various components in the system make to allow the user to view the sensors information page for a specific sensor and group. This page allows the user to request statistics/download readings for a given sensor and monitor sensor readings in real-time.

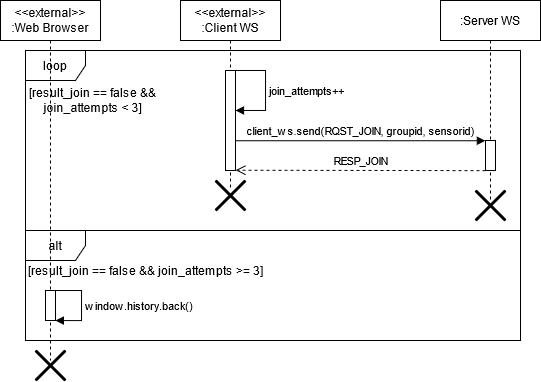


Figure : Subscribing to Sensor Readings

This sequence diagram shows the series of messages and responses the various components in the system make to allow a web client/browser to connect to the server and receive sensor readings when the publisher broadcasts a new sensor reading to subscribed clients.

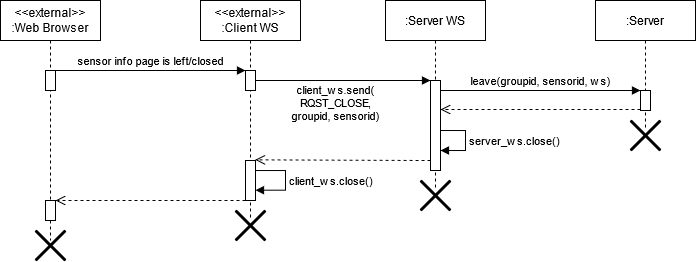


Figure : Unsubscribing from Sensor Readings

This sequence diagram shows the series of messages and responses the various components in the system make to allow a web client to unsubscribe from receiving sensor readings.

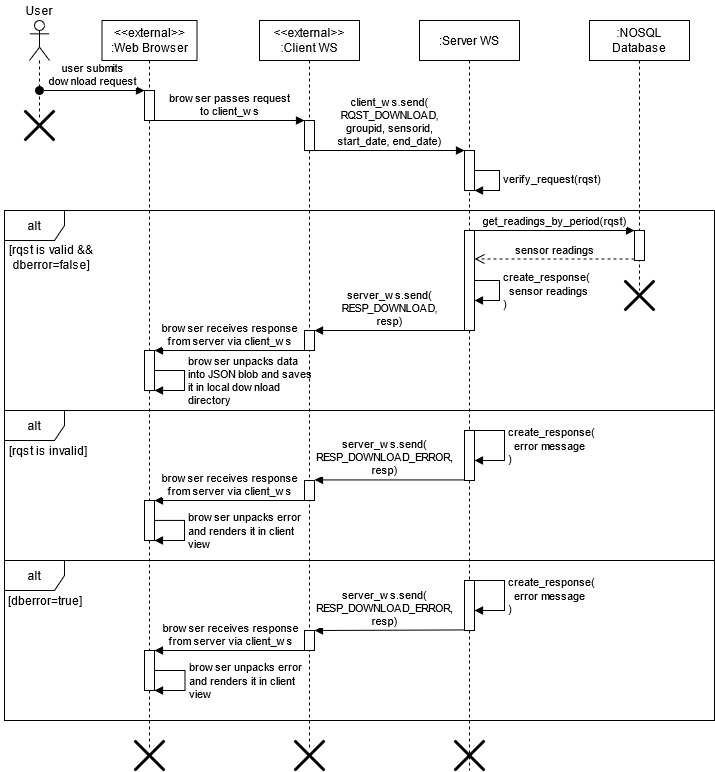


Figure 11: Downloading Sensor Readings

This sequence diagram shows the series of messages and response the various components in the system make to allow a web client to download readings for a specific sensor over a user-provided time period.

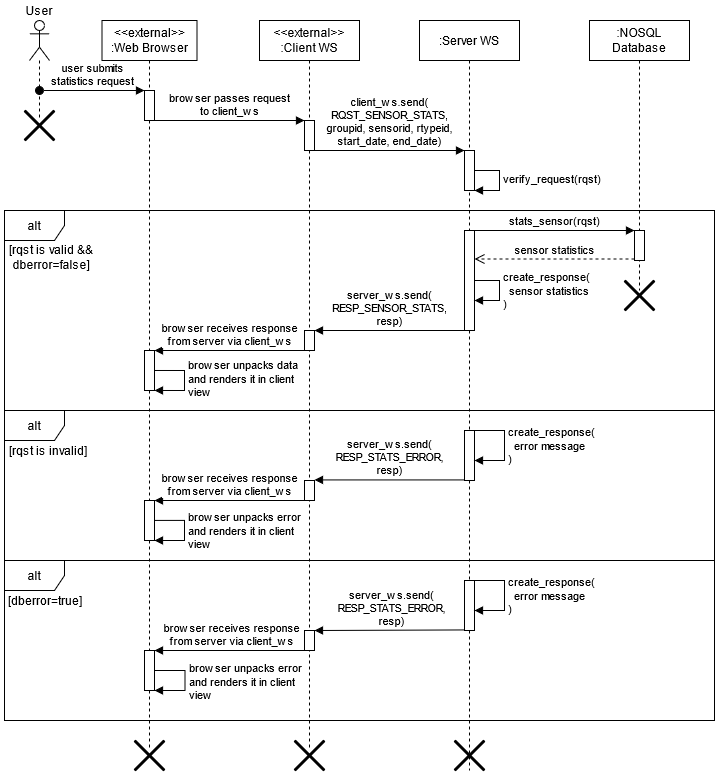


Figure : Requesting Sensor Statistics

This sequence diagram shows the series of messages and responses that allows the user to request statistics on sensor readings for a given time period.

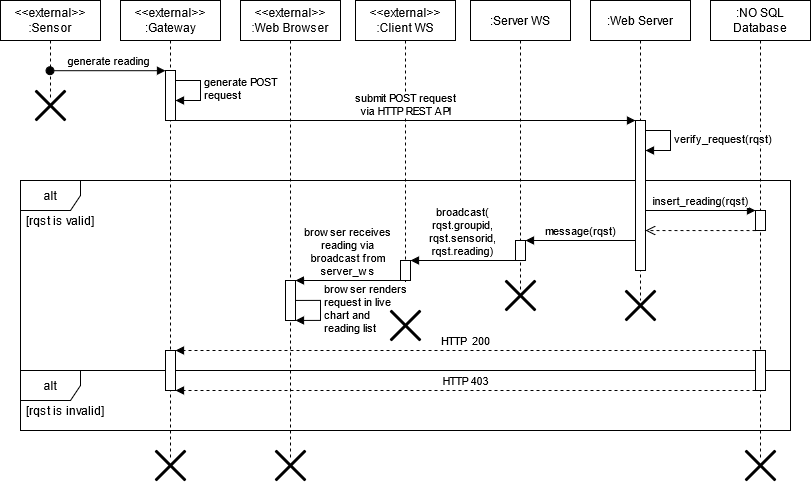


Figure : Uploading a Sensor Reading

This sequence diagram shows the series of messages and responses that allows a sensor to upload a reading to the system. This is one of the most complicated sequence diagrams featured in this document as it involves every single component in the system.

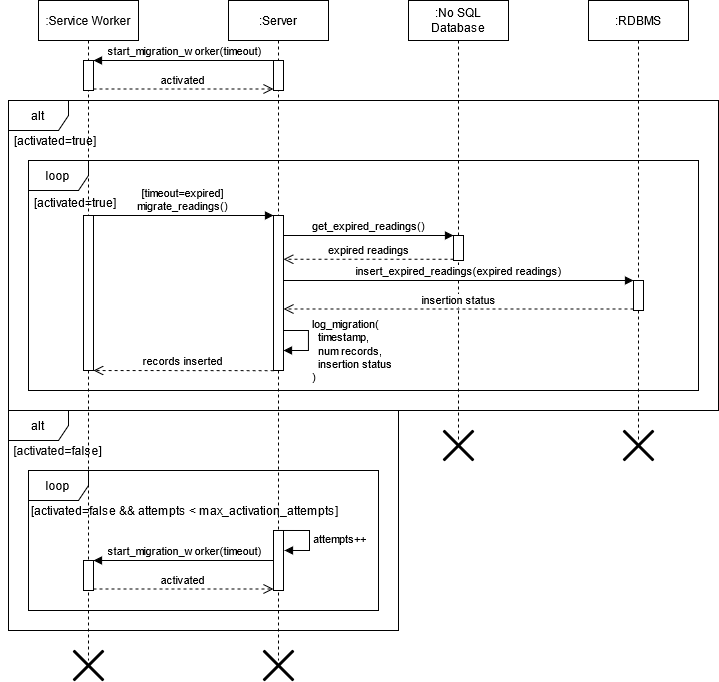


Figure : Migrating Expired Sensor Readings

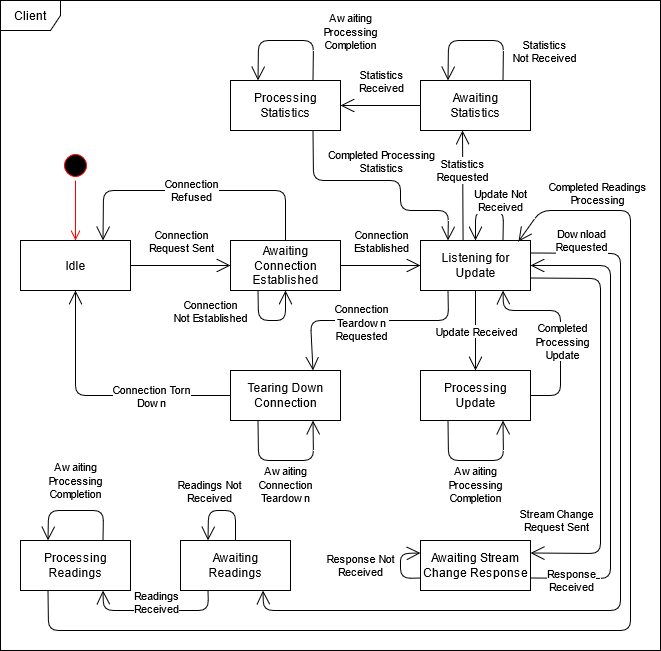
This sequence diagram shows the series of messages and responses the system makes to allow a service worker to migrate data from the NOSQL database to the relational database. This feature as discussed between the team members will not be implemented in the system that is presented as part of the IOT Collaborative or Devendra’s Masters Thesis (primarily to keep things simple while Devendra is demoing the project to his thesis committee). This feature will be intentionally left for future maintainers of this software. Future maintainers of this software should utilize this sequence diagram when referencing how this feature should be implemented as well as understanding at a high-level what components are involved and how they communicate with each other.

# State Diagrams

State machine diagrams are a type of behavior diagram that shows how a system behaves given a specific set of finite state transitions. State Machine diagrams are used frequently in Operating Systems and Computer Networking to show state transitions in applications such as process scheduling and network protocols. Behavior is modelled via the diagram using a set of interconnected states where each connection represents a state transition. State machines themselves may exist within the context of a larger system in which they are periodically triggered via specific call triggers, or they may exist separately to perform some independent processing. Each state in a state machine may define an associated behavioral feature associated with the system that owns that state machine. In this case, the parameters for the state machine must match the parameters for the behavioral feature to access the behavioral feature parameters from within the state machine.

## Client/Server Protocol

This section contains the state machine diagrams for the client/server protocol. This protocol is utilized to establish and maintain WebSocket connections between the web server and connected clients as well as to allow them to exchange information. The state machine protocol for the server is implemented in the client code while the state machine protocol for the web server is implemented in the web server code. Note that the server state machine is from the context of a single client WebSocket connection. In actuality, the server maintains many WebSocket connections to different clients simultaneously.

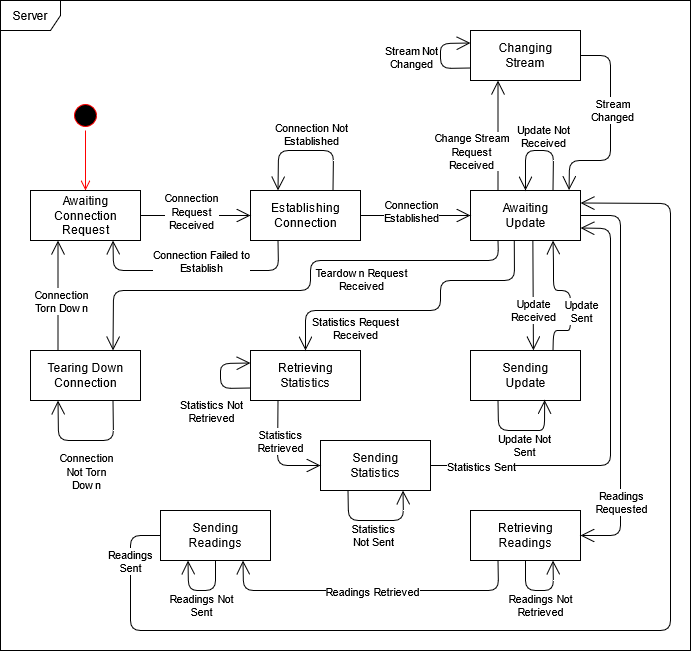


The client is required to make HTTP WebSocket requests where the payload is a JSON object containing a ‘cmd’ value followed by the parameters for the command. Valid commands are given by the following table. This table includes the command name, parameter list and descriptions, as well as a description of the command itself.

|  |  |  |
| --- | --- | --- |
| **Command ID** | **Command** | **Description** |
| CLIENT\_CMD\_01 | RQST\_JOIN | Requests a WebSocket connection be established between the client and webserver. |
| CLIENT\_CMD\_02 | RQST\_CLOSE | Requests that the existing WebSocket connection between the client and web server be closed/torn down. |
| CLIENT\_CMD\_03 | RQST\_SENSOR\_STATS | Requests statistics for the sensor being viewed. |
| CLIENT\_CMD\_04 | RQST\_STREAM | Requests that the server switch the sensor reading type for sensor readings being sent to the web client. |
| CLIENT\_CMD\_05 | RQST\_DOWNLOAD | Requests the downloading of sensor readings for the current sensor. |

|  |  |  |
| --- | --- | --- |
| **Command ID** | **Parameter** | **Description** |
| CLIENT\_CMD\_01 | groupid | The group identifier corresponding to the group to listen for updates from. |
| CLIENT\_CMD\_01 | sensorid | The sensor identifier corresponding to the sensor to listen for updates from. |
| CLIENT\_CMD\_02 | groupid | The group identifier corresponding to the group the client is listening for updates on. |
| CLIENT\_CMD\_02 | sensorid | The sensor identifier corresponding to the sensor the client is listening for updates on. |
| CLIENT\_CMD\_03 | groupid | The group identifier corresponding to the sensor to the group the client is listening for updates on. |
| CLIENT\_CMD\_03 | sensorid | The sensor identifier corresponding to the sensor the client is listening for updates on. |
| CLIENT\_CMD\_03 | rtypeid | The reading type identifier corresponding to the sensor reading to retrieve. |
| CLIENT\_CMD\_03 | start\_date | The start date timestamp. |
| CLIENT\_CMD\_03 | end\_date | The end date timestamp. |
| CLIENT\_CMD\_04 | sensorid | The sensorid corresponding to the sensor to lookup statistics for. |
| CLIENT\_CMD\_04 | groupid | The groupid corresponding to the group to lookup statistics for. |
| CLIENT\_CMD\_04 | rtypeid | The reading type identifier corresponding to the sensor reading to retrieve. |
| CLIENT\_CMD\_05 | sensorid | The sensorid corresponding to the sensor to download readings for. |
| CLIENT\_CMD\_05 | groupid | The groupid corresponding to the group the sensor belongs to. |
| CLIENT\_CMD\_05 | start\_date | The UNIX timestamp representing the start of the download window. |
| CLIENT\_CMD\_05 | end\_date | The UNIX timestamp representing the end of the download window. |

Packets sent by connected clients to the server are expected to adhere to the prior specifications. Data is to be sent in the request body in an HTML-safe JSON serialized format.



The server listens for commands from connected clients and either responds with the following commands or sends the following commands to connected clients.

|  |  |  |
| --- | --- | --- |
| **Command ID** | **Command** | **Description** |
| SERVER\_CMD\_01 | RESP\_JOIN | Triggered in response to receiving a RQST\_JOIN command from a connected client. This command attempts to add the client to the list of listening clients for the given group identifier and sensor identifier. The server will send back a boolean value indicating whether this was successful or not. If this was unsuccessful, the client will resend the join request up to MAX\_JOIN\_ATTEMPTS times (where MAX\_JOIN\_ATTEMPTS) is configured by the server administrator. If the client exceeds this number of attempts, the client will be automatically redirected back to the sensors listing page by their web browser. |
| SERVER\_CMD\_02 | RESP\_READING | Triggered in response to receiving a sensor update over a specific web server endpoint from a sensor or any other device that sends updates using the well-defined protocol and packet format specified in this design documentation. This command simultaneously stores the sensor reading in the database and broadcasts the reading to connected clients that are listening for sensor readings on the given groupid and sensorid. |
| SERVER\_CMD\_03 | RESP\_STREAM | Triggered in response to receiving a RQST\_STREAM command from a connected client. This command updates the server to the specified rtypeid for the given groupid and sensorid effectively changing what type of sensor reading the client is receiving updates for and can request statistics for. |
| SERVER\_CMD\_04 | RESP\_SENSOR\_STATS | Triggered in response to receiving a RQST\_SENSOR\_STATS command from a connected client. This command invokes the database to determine the minimum, maximum, and average value for a given time period for a given groupid and sensor id. This information is then sent back to the requesting client. |
| SERVER\_CMD\_05 | RESP\_DOWNLOAD | Triggered in response to receiving a RQST\_DOWNLOAD command from a connected client. Downloads the sensor readings for a given sensorid and groupid within a given start\_date and end\_date. The end\_date is intended to be exclusive. |
| SERVER\_CMD\_06 | RESP\_ERROR | Triggered when the web server encounters a generic error, usually caused by a malformed request or database error. Sends a response to the client WebSocket containing an ‘error’ field that can be rendered in the clients viewport. |
| SERVER\_CMD\_07 | RESP\_STATS\_ERROR | Triggered when the web server encounters an error while retrieving statistics for a specific sensor. Sends a response to the client WebSocket containing an ‘error’ field that can be rendered in the clients viewport. This should be treated separately by the client from a generic error. |
| SERVER\_CMD\_08 | RESP\_DOWNLOAD\_ERROR | Triggered when the web server encounters an error while preparing the sensor reading download for the client. Sends a response to the client WebSocket containing an ‘error’ field that can be rendered in the clients viewport. This should be treated separately by the client from a generic error. |

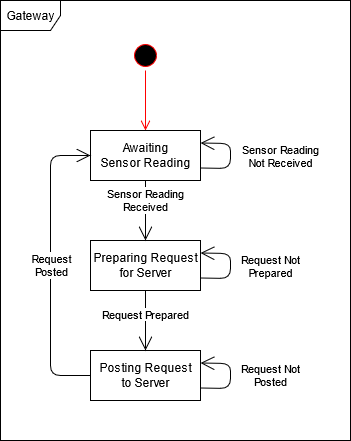
|  |  |  |
| --- | --- | --- |
| **Command ID** | **Parameter** | **Description** |
| SERVER\_CMD\_01 | join\_result | A boolean value indicating whether the server was able to add the client to the list of listening clients for the indicated group identifier and sensor identifier. |
| SERVER\_CMD\_02 | readings | A list of mappings where each mapping contains a reading type identifier, UNIX timestamp, value, and a human readable string for the sensor reading indexed by rtypeid, ts, val, and rstring respectively. |
| SERVER\_CMD\_03 | readings | A list of mappings where each mapping contains a reading type identifier, UNIX timestamp, value, and a human readable string for the sensor reading indexed by rtypeid, ts, val, and rstring respectively. |
| SERVER\_CMD\_04 | min | The minimum sensor reading for a given start\_date, end\_date, groupid, and sensorid. |
| SERVER\_CMD\_04 | max | The maximum sensor reading for a given start\_date, end\_date, groupid, and sensorid. |
| SERVER\_CMD\_04 | avg | The average sensor reading for a given start\_date, end\_date, groupid, and sensorid. |
| SERVER\_CMD\_05 | data | The sensor reading data for the given groupid, sensorid, start\_date, and end\_date. This will always be a stringified JSON array. |
| SERVER\_CMD\_06 | error | The error message generated by the server. |
| SERVER\_CMD\_07 | error | The error message generated by the server. Should be treated differently from a generic error message. |
| SERVER\_CMD\_08 | error | The error message generated by the server. Should be treated differently from a generic error message. |

Packets sent by the server to connected clients are expected to adhere to the prior specifications. Data is to be sent in the response body in an HTML-safe JSON serialized format. Note that if at least one client is connected to the web server and is listening for updates for a sensor, the web server will always be in the “Awaiting Update” state for that sensor. These state machine diagrams are designed from the perspective of a single client. The web server, however, can be connected to many clients simultaneously.

## Gateway/Server Protocol

This section contains the state machine diagrams for the gateway/server protocol. This protocol is utilized to upload sensor data from the gateway to the web server. The state machine protocol for the server will be implemented in the client code while the state machine protocol for the web server will be implemented in the web server code. Note that the server must define an HTTP POST endpoint for gateways or other HTTP enabled devices to upload sensor readings to. Likewise, gateways or other HTTP enable devices that wish to upload sensor readings to the server must correctly format an HTTP POST request and send it to the endpoint specified by the servers configuration.

The gateway (or any device capable of making HTTP POST requests such as an HTTP enabled sensor) follows the protocol defined by the following state machine diagram.



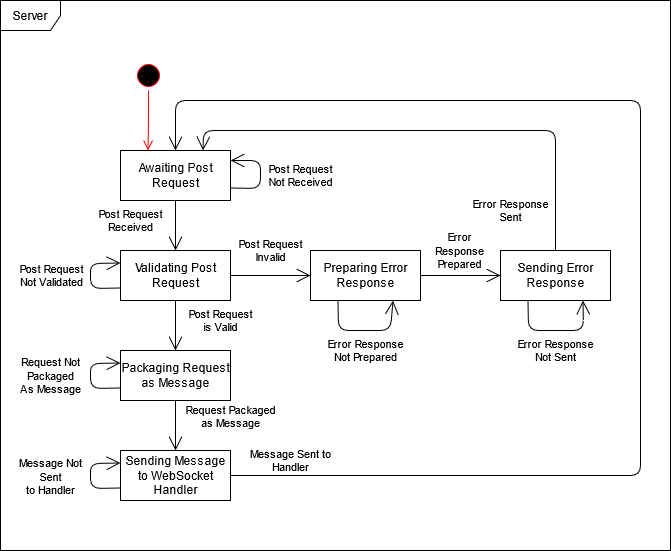
Future versions of the protocol, specifically for gateways, will incorporate batching so multiple sensor readings may be posted to the server at once.

HTTP POST requests generated by the client are required to package their data into an HTML-compatible serialized JSON object with the following parameters.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Data Type** | **Description** |
| sensorid | 8-bit unsigned integer | The sensor identifier that uniquely identifies a sensor within a group of sensor. |
| groupid | 8-bit unsigned integer | The group identifier that identifies what group a sensor belongs to. |
| rtypeid | 8-bit unsigned integer | The reading type identifier that identifies what type of sensor reading `val` corresponds to. |
| ts | 64-bit unsigned integer | The UNIX timestamp corresponding to when the sensor reading was taken. |
| val | 64-bit signed floating point | The value corresponding to the sensor reading taken by the sensor. |

The JSON object containing the above parameters must be transmitted in the body of the POST request.

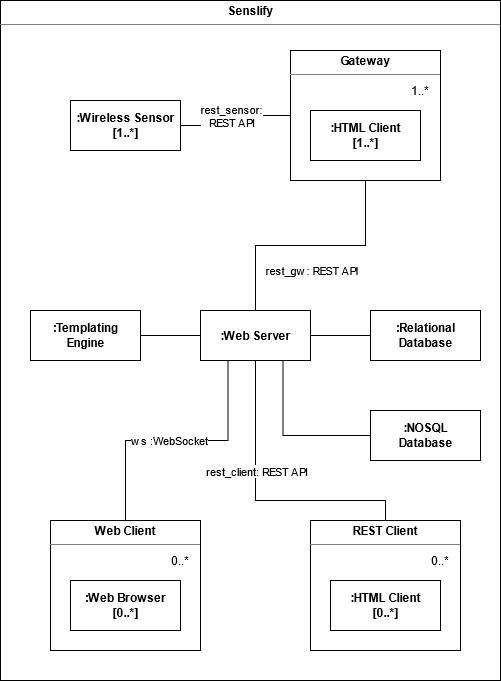
The server follows the protocol defined by the following state machine diagram when receiving HTTP POST requests from a gateway or other HTTP enabled device.



If the server encounters an error while validating a POST request, it shall return an HTTP 403 error along with an appropriate error message in the body of the response.

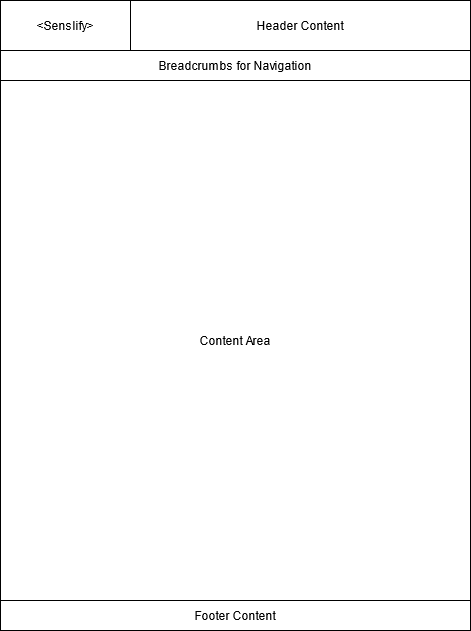
# Structure Diagrams

This section showcases the high-level structure diagram that details the individual systems that make up the overall system. Senslify is a system of systems. Each subsystem in the following diagram can be implemented or represent an entirely different system within it’s own category. For instance, the **Web Clients** subsystem can be represented by the following systems: Google Chrome, Mozilla Firefox, Microsoft Edge, etc… As long as the sub-systems that make up Senslify adhere to the requirements in the System Requirements Specification and the design in this document, the individual sub-systems can be any permutation of systems.

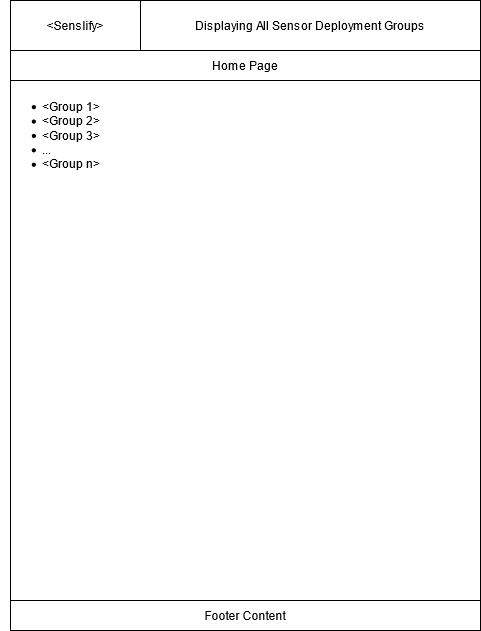


# Wireframe Diagrams

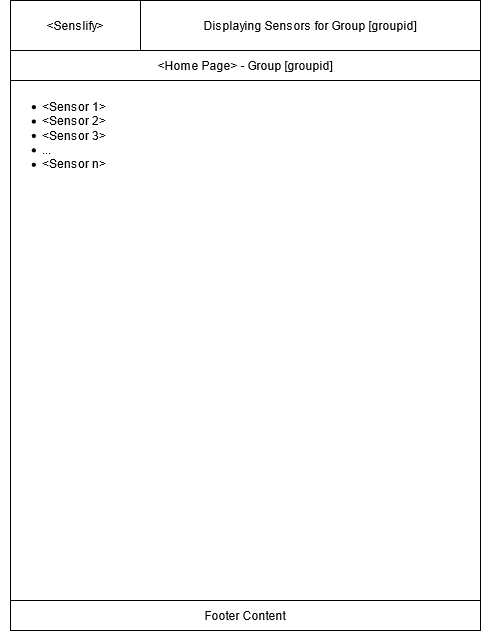
This section showcases the wireframe mockups used to design the templates that the system generates HTML from. Note that elements surrounded by carets (“<, ”>”) are to be treated as links while elements surrounded by brackets (“[“, “]”) are to be treated as template fillable elements (elements filled out by the template engine).



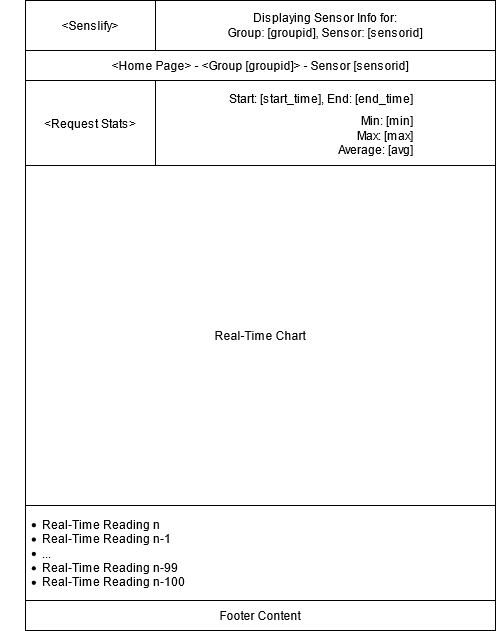
This wireframe shows the generic layout that all pages on the site are designed around. Please reference it as necessary when looking over the other wireframes in this document. Note that the Senslify link in the upper-left corner of the template is should always be present on every page of the site and can be used to navigate back to the home page at any time.



This wireframe shows the home page for the system. This page is displayed whenever a user visits the index page for the site and is generated from a template on-demand. Its sole responsibility is to serve links to the various sensor groups.



This wireframe shows the sensor listings page for the system. This page is displayed whenever a user visits the index page for the site and is generated from a template on-demand. Its sole responsibility is to serve links to the various sensors within a group.



This wireframe shows the sensor information page for the system. This page operates in real-time as indicated by the two real-time components shown on the wireframe. The first is a real-time chart that maps sensor readings on the y-axis to time on the x-axis. The second is a real-time log of the n (system administrator configurable) most recent sensor readings. Both the real-time chart and real-time log should update upon receiving a broadcast from the web server. If the real-time log is at max capacity, the oldest reading should be removed from the end of the log and the new reading should be added at the beginning of the log.

There is also a component on this template that allows the user to request statistics for the sensor being displayed. Clicking the <Request Stats> button should open up a modal prompt (without navigating to a new page) that allows the user to specify the start and end datetimes they want to request statistics on as well as a submit button. When the submit button is clicked, the prompt should close and submit the request to the web server. Assuming the request is valid, the web server will compute the statistics for the time period and send it back to the client. At this point, the client should fill in the [start\_time], [end\_time], [min], [max], and [avg] fields with the statistics information the server sends back as well as the datetime data the user has submitted. This prompt is simple enough to where it does not require a wireframe.