IntelliHAUS

Intelligent Home AUtomation System

ComS 309, Spring 2016
Eric Middleton, Sam Oswalt, David Wehr

Description

A smart home framework that allows devices within a user's house to communicate with a central server. Communication involves uploading data from sensors and receiving commands to perform actions.

Users can also define rules to automate behavior of their devices, using data from their sensors, time of day, or day of the week.

The home is manageable from a web portal that is available globally, allowing the user to view data from their sensors, send commands to actuators, and create rules.

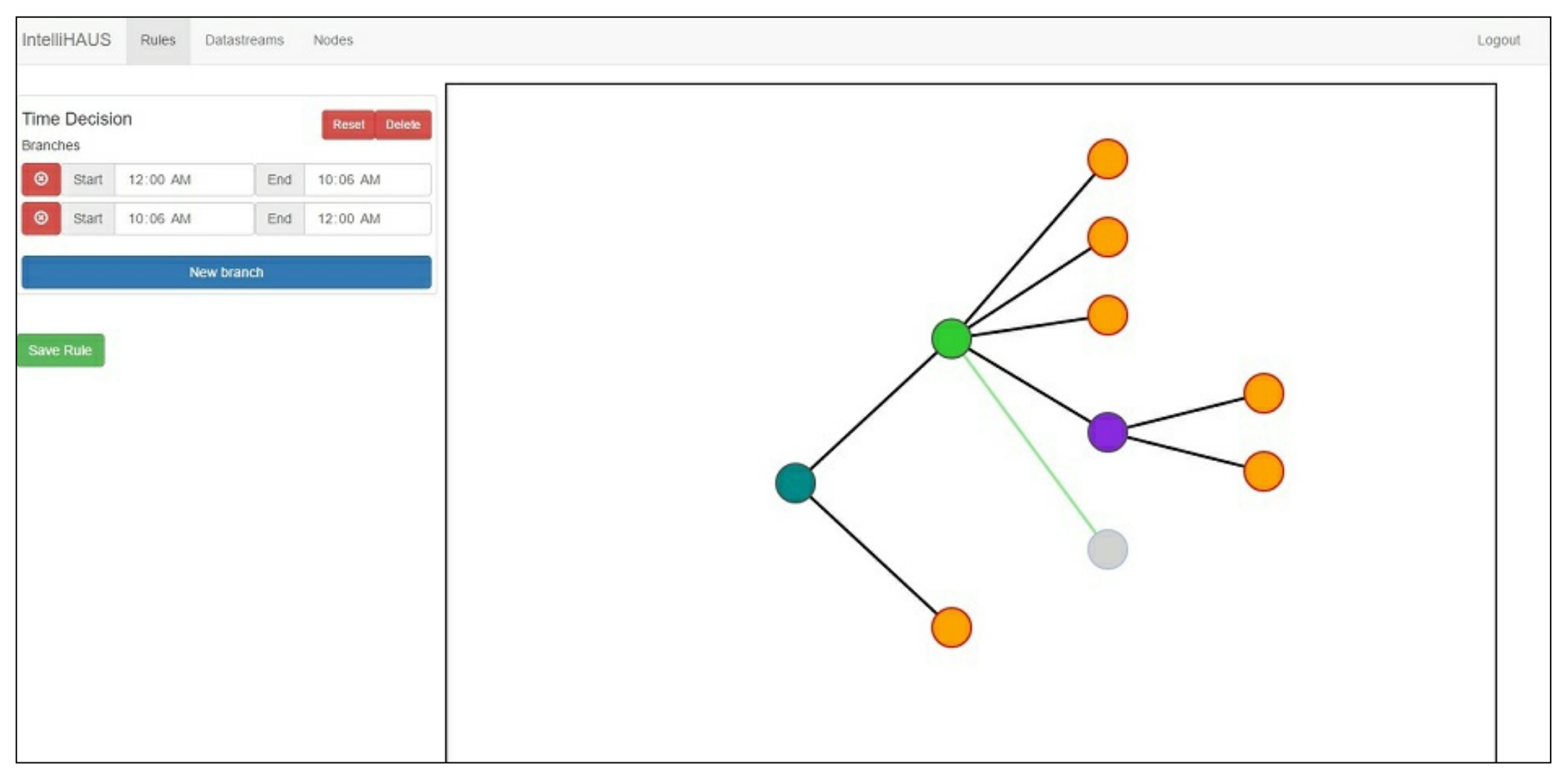
Actors

- User (owner of home)
- Device within home

User Interfaces

Rule Editor

- Sidebar (add/edit branches)
- Tree Interface (visualize tree)

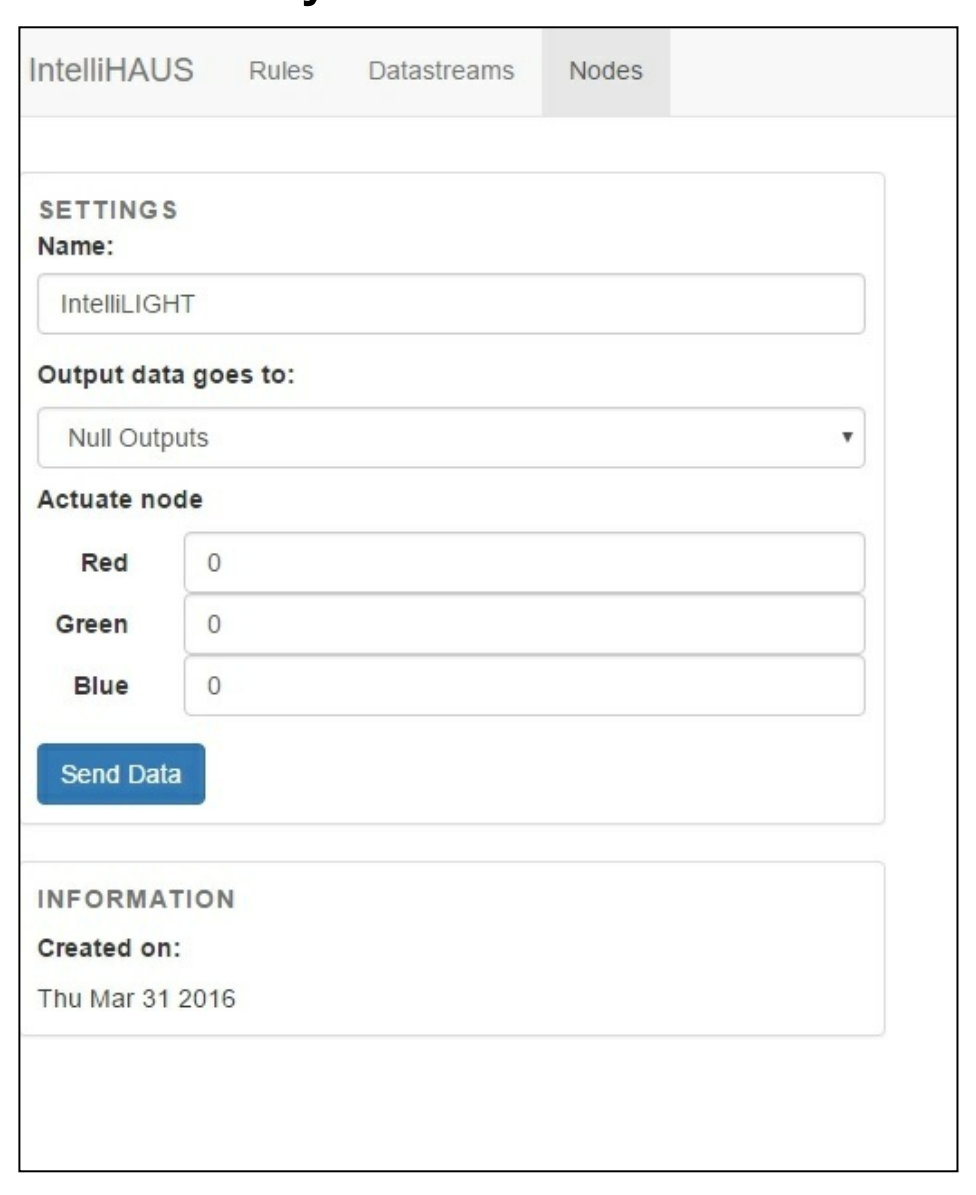


Time Decision
Branches

Start	12:00 AM	End	10:06 AM
Start	10:06 AM	End	12:00 AM

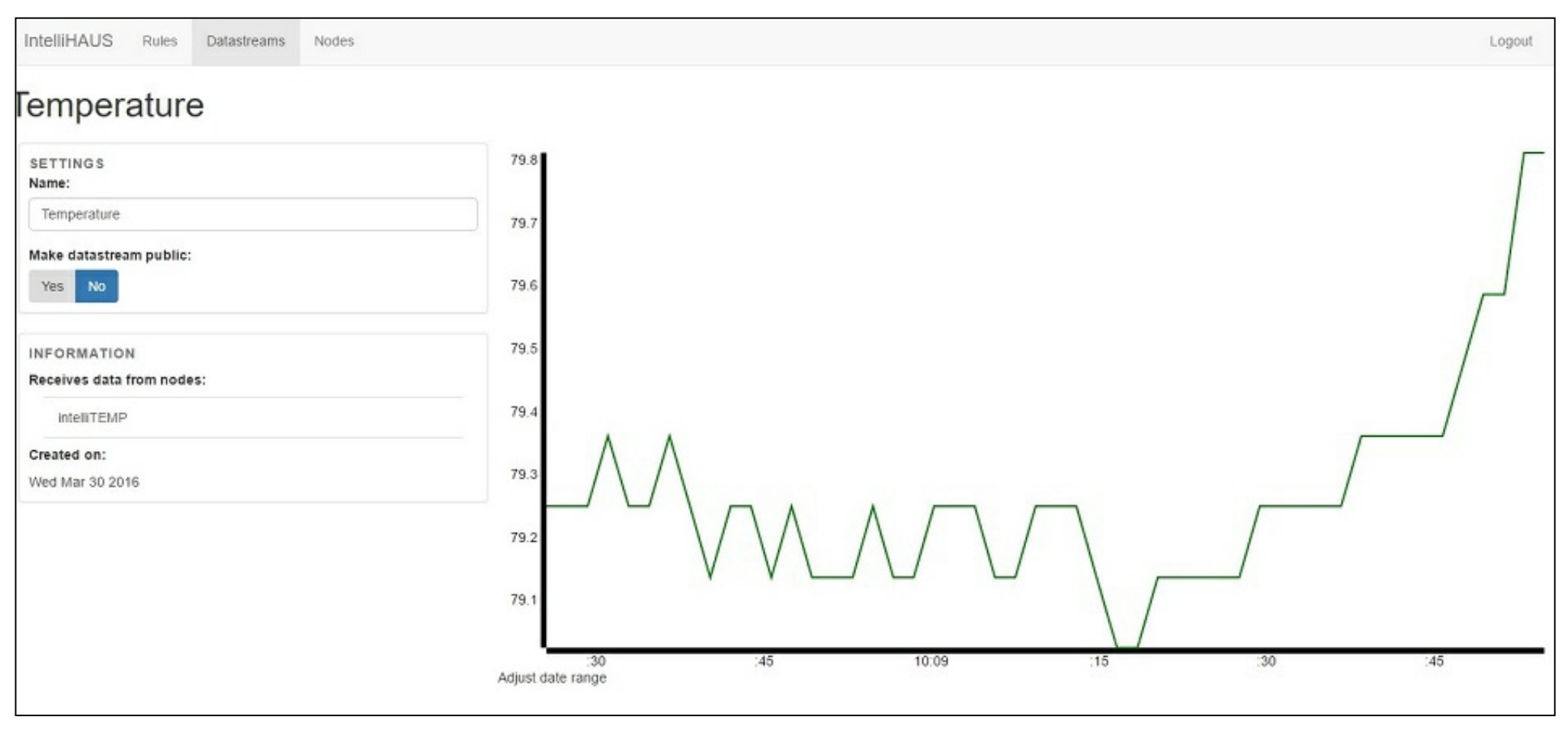
Node

- View specific node information
- Manually actuate nodes



SETTINGS
Name: IntelliLIGHT
Output data goes to: Null Outputs
Actuate node
Red: 0
Green: 0
Blue: 0

INFORMATION
Created on: Thu Mar 31 2016



Temperature

SETTINGS
Name: Temperature
Make datastream public: Yes [✓ No]

INFORMATION
Receives data from nodes: intelliTEMP
Created on: Wed Mar 30 2016

Datastreams

- View specifc datastream graph
- Make datastream public/private

Server

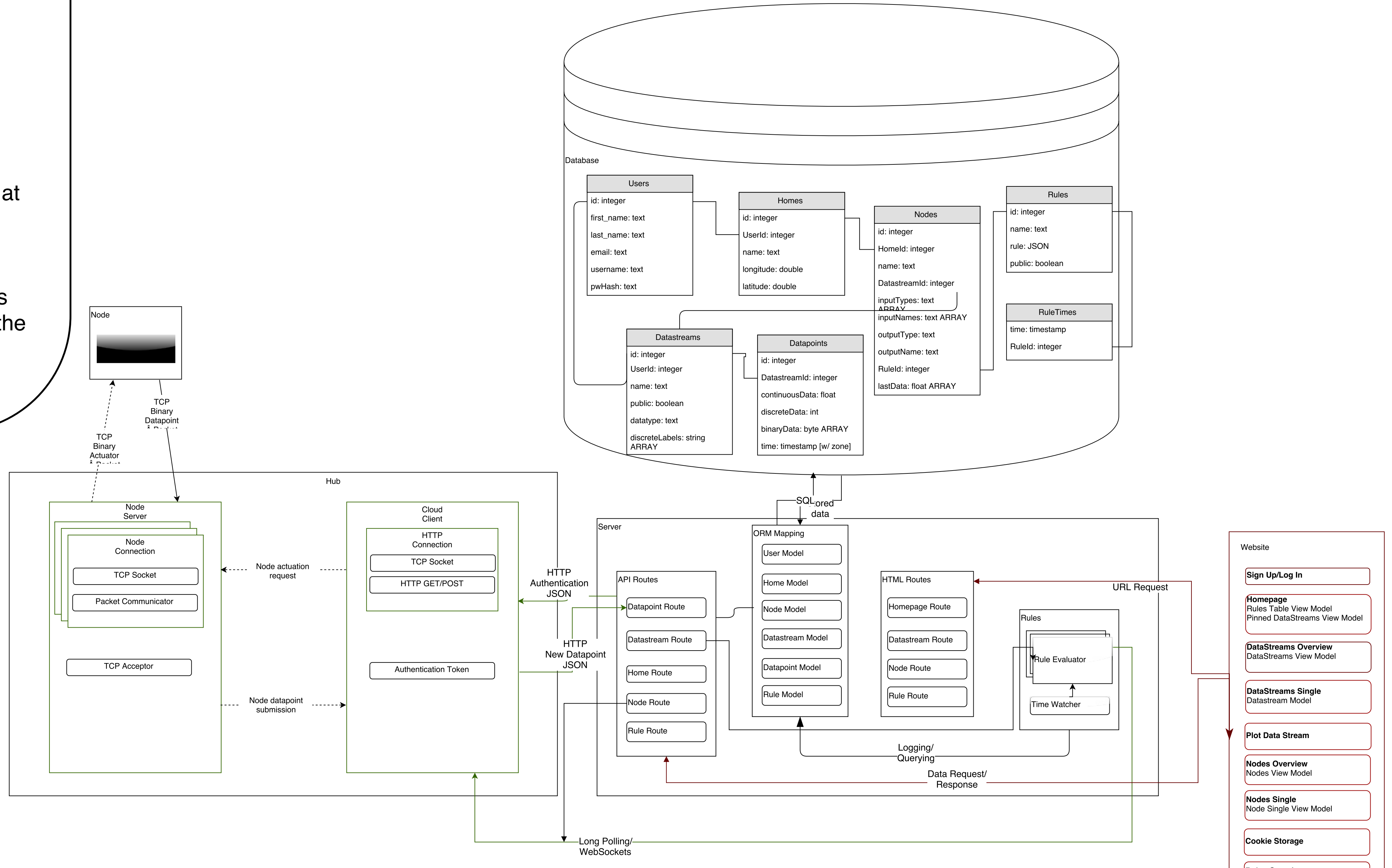
Interfaces
The server communicates via a REST API with the hub and the website. The API is listed below.

Modules
The server is arranged these modules:

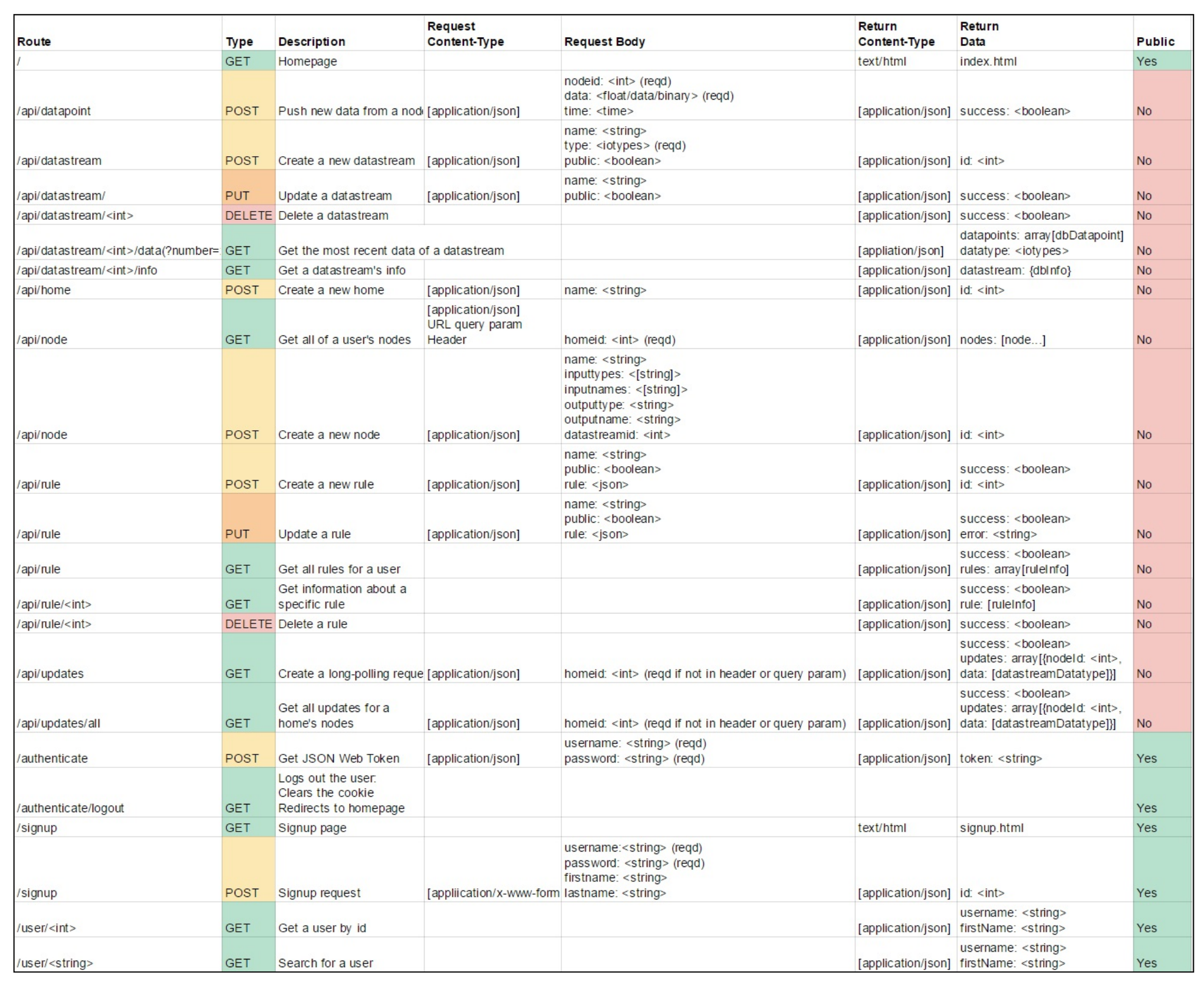
- Models - Database models for Sequelize, a node.js ORM
- Rule Evaluation - Standalone module that can evalute rules
- Routes - Separates the portions of the server dedicated to handling different requests and responses
- API routes - Routes related to API calls
- HTML routes - Routes related to serving the website
- Public routes - Routes accessible without authentication

Design Decisions
Our server is written in node.js, since it needs to handle many simultaneous connections at a time, including long-poll requests, which require the server to handle connections in an asynchronous fashion.

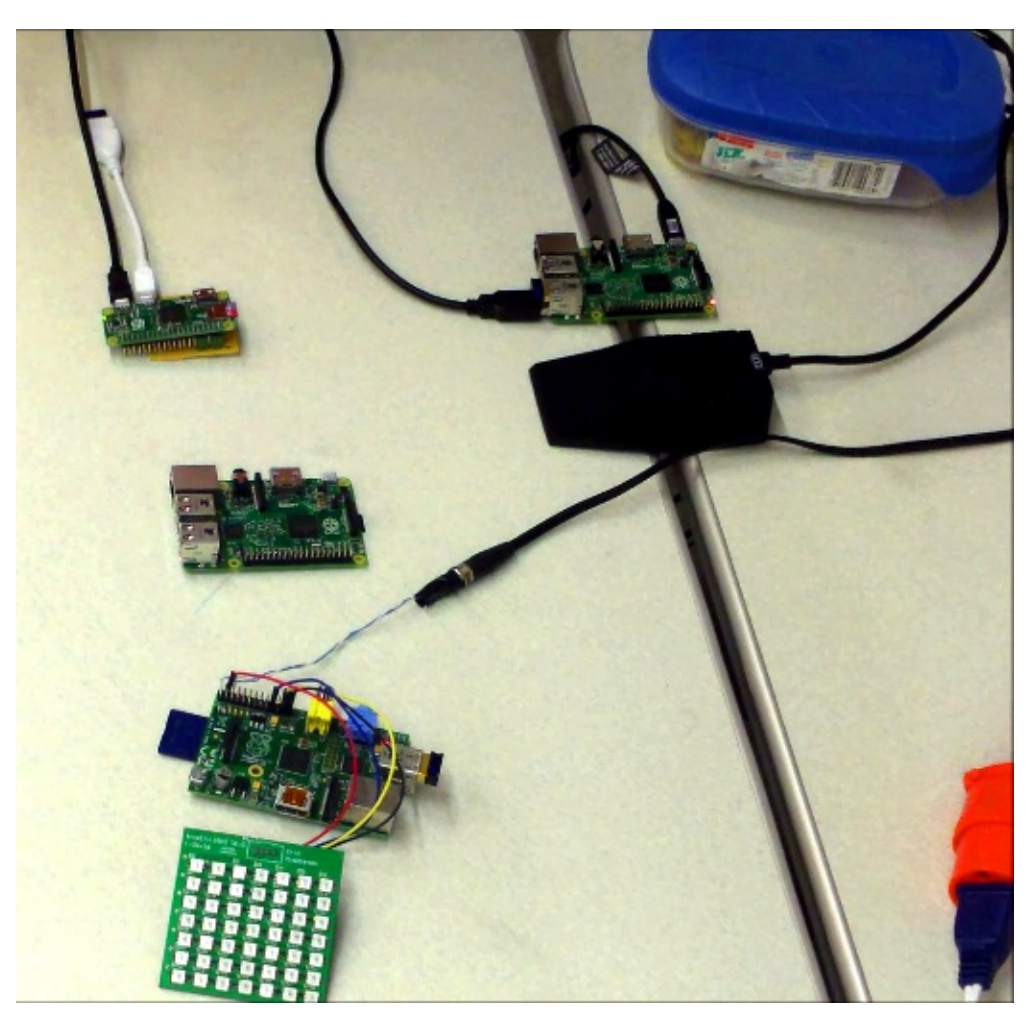
Since the rules will take a significant amount of the total processing time for the server, as they need to be re-evaluated often, separate worker processes are created to handle all the rule evaluation, which communicate with the main process via IPC.



REST API



Route	Type	Description	Request Content-Type	Request Body	Return Content-Type	Return Data	Public
/	GET	Homepage			text/html	index.html	Yes
/api/datapoint	POST	Push new data from a nod	[application/json]	nodeid: <int> (reqd) data: <float/data/binary> (reqd) time: <time>	[application/json]	success: <boolean>	No
/api/datastream	POST	Create a new datastream	[application/json]	name: <string> type: <iotypes> (reqd) public: <boolean>	[application/json]	id: <int>	No
/api/datastream/	PUT	Update a datastream	[application/json]	name: <string> public: <boolean>	[application/json]	success: <boolean>	No
/api/datastream/<int>	DELETE	Delete a datastream			[application/json]	success: <boolean>	No
/api/datastream/<int>/data(?number=	GET	Get the most recent data of a datastream			[appliation/json]	datapoints: array[dbDatapoint] datatype: <iotypes>	No
/api/datastream/<int>/info	GET	Get a datastream's info			[application/json]	datastream: {dbInfo}	No
/api/home	POST	Create a new home	[application/json]	name: <string>	[application/json]	id: <int>	No
/api/node	GET	Get all of a user's nodes	[application/json] URL query param Header	homeid: <int> (reqd)	[application/json]	nodes: [node...]	No
/api/node	POST	Create a new node	[application/json]	name: <string> inputtypes: <[string]> inputnames: <[string]> outputtype: <string> outputname: <string> datastreamid: <int>	[application/json]	id: <int>	No
/api/rule	POST	Create a new rule	[application/json]	name: <string> public: <boolean> rule: <json>	[application/json]	success: <boolean> id: <int>	No
/api/rule	PUT	Update a rule	[application/json]	name: <string> public: <boolean> rule: <json>	[application/json]	success: <boolean> error: <string>	No
/api/rule	GET	Get all rules for a user			[application/json]	success: <boolean> rules: array[ruleInfo]	No
/api/rule/<int>	GET	Get information about a specific rule			[application/json]	success: <boolean> rule: [ruleInfo]	No
/api/rule/<int>	DELETE	Delete a rule			[application/json]	success: <boolean>	No
/api/updates	GET	Create a long-polling reque	[application/json]	homeid: <int> (reqd if not in header or query param)	[application/json]	success: <boolean> updates: array[{nodeId: <int>, data: [datastreamDatatype]}]	No
/api/updates/all	GET	Get all updates for a home's nodes	[application/json]	homeid: <int> (reqd if not in header or query param)	[application/json]	success: <boolean> updates: array[{nodeId: <int>, data: [datastreamDatatype]}]	No
/authenticate	POST	Get JSON Web Token	[application/json]	username: <string> (reqd) password: <string> (reqd)	[application/json]	token: <string>	Yes
/authenticate/logout	GET	Logs out the user: Clears the cookie Redirects to homepage					Yes
/signup	GET	Signup page			text/html	signup.html	Yes
/signup	POST	Signup request	[appliication/x-www-form	username:<string> (reqd) password: <string> (reqd) firstname: <string> lastname: <string>	[application/json]	id: <int>	Yes
/user/<int>	GET	Get a user by id			[application/json]	username: <string> firstName: <string>	Yes
/user/<string>	GET	Search for a user			[application/json]	username: <string> firstName: <string>	Yes



Four of our demonstration nodes: connected outlet, voice recognition, smart light, and temperature sensor.

Hub

Interfaces
The hub uses these interfaces:

- HTTP to communicate to the server
- Binary protocol over TCP to communicate with the nodes

Modules
The hub is arranged into these modules:

- Server communicator
  - HTTP class
    - GET/POST
    - Request pipelining
    - Automatic reconnecting
  - Server class
    - Uses HTTP class to communicate with server
    - Maintains state information (e.g. current authentication token)
- Node Server class
  - TCP acceptor for arbitrary number of nodes
  - Separate thread for each TCP connection

Design Decisions
The hub is written in C++ with the boost::asio library for networking so that it can efficiently handle a large number of connections on a low-power computer.
Each TCP connection has its own thread, which allows for fast and efficient communication even with many devices connected.

Team

Members

- Sam Oswalt - CprE Senior
- Eric Middleton - CprE/EE Senior
- David Wehr - CprE Junior

Problems

- Reliability was the most difficult issue we had, since there are a lot of systems that need to connect with each other.
- We also had some difficulty getting all of the different threads in the hub to work together well.
- Team member dropped course, so we had to adjust our goals midway through the semester