COMP2014 – Systems Engineering II Group 7

RoboHome – Home Automation Project

Michael Boon, Chi Chiu (Terrence), Li Quan Khoo, Kinga Mrugala and Jerrine Soh

April 2013

Contents

1 Introduction

1.1 Project Team

Our project team consisted of Michael Boon, Chi Chiu (Terrence), Li Quan Khoo, Kinga Mrugala and Jerrine Soh. Throughout the process, our supervisor was Prof. Steve Hailes.

1.2 Project Outline

Following on from COMP2013, our system aims to manage a number of sensors around a building. The system should monitor and control these sensors whilst providing a user with a way to interact with the house and automatically react to events around the house.

Unfortunately, home automation is traditionally rather expensive and difficult to set up. The closest thing to an industry standard are X10 devices and the majority of newly released accessible hardware works strictly with hardware of its own kind which forces users to stick to a particular brand.

Our project aims to make home automation more accessible to users by providing them with a simple way to connect any number of devices together with our platform regardless of the brand of hardware. This will allow users to buy affordable hardware from places such as Amazon and Maplin and integrate it with their home whilst also allowing hobbyists to create their own hardware using devices such as Arduino. We achieve this by specifying an API for our system, which means we can write a number of "middle layers" between our system and proprietary hardware, which will be discussed in more detail later. We also aim to give the user control over the rules to automate the house by and graph information we gather for them. The project is primarily aimed at assisted living for the elderly and disabled but is also for home automation enthusiasts and anyone wishing to automate their home.

In COMP2013, we created a proof of concept for a home automation system. This was very basic and consisted of a very limited number of sensors working exclusively with the .NET Gadgeteer hardware. We decided to take what we had learnt from this experience and redesign the entire system from scratch with scalability in mind.

The scope of our project does not include physical security due to time and hardware constraints. For example, whilst we do provide security for the system with a login, we do not aim to control any door locks, CCTV cameras and other security systems around the house. We believe that home security should be treated as a different area to automation as far as our project is concerned.

This report will cover our updated research and design from COMP2013. It will then go into detail about our new implementation and testing strategy before evaluating our progress.

2 Updated Information and Research

2.1 Competitor Analysis

As part of our literature review, we researched existing solutions to spot opportunities and take advantage of missing features.

Table 2.1 – Competitor Analysis

	Home Seer	Power Home	ActiveHome Pro	Open Remote	Open Source Automation
Remote access	Yes	Yes	Yes	Yes	Yes
Customizable UI	Yes	Yes	Yes	Yes	Yes
Timer triggering	Yes	Yes	Yes	Yes	Yes
Manual triggering	Yes	Yes	Yes	Yes	No data
Event scripts / macro	Yes	Yes	Yes	Yes	Yes
Voice commands	Yes	No	No	No data	Yes
Media centre support	Yes	No	No	Yes	No data
Security system support	Yes	No	No	Yes	Yes
OS support					
Windows 8	Yes	No	No	No data	Yes
Windows 7	Yes	No	No	Yes	Yes
Windows Vista	Yes	Yes	No	Yes	Yes
Windows XP	Yes	Yes	Yes	Yes	Yes
iOS	Yes	No	No	Yes	No
Linux	No data	No	No	Yes	No
Android	Yes	No	No	Yes	No
Windows Phone	No	No	No	No data	No
Major protocol support					
X10	Yes	Yes	Yes	Yes	Yes
Insteon	Yes	Yes	No	Yes	Yes
Z-Wave	Yes	No	No	Yes	Yes
HVAC	Yes	No	No	No	No
Infrared	Yes	Yes	No	No	Yes

2.1.1 HomeSeer

- Closed source
- Supports a wide range of communication protocols
- Expensive; hardware costs $200 - $600 and to control from iOS or Android devices a separate controller costing $500 - $3000 is necessary.
- Modular UI; the user downloads components based on what he/she needs.

2.1.2 PowerHome

- Closed source
- Paying $100 buys the software, the SDK, and the license to modify the source code, but modified source cannot be shared
- Good scripting support – in any language supported by Windows Script Host
- Poor presentation/UI

2.1.3 ActiveHome Pro

- Closed source
- Only supports Windows XP and X10 protocol
- Poorly designed UI and getting the SDK requires registration for no reason

2.1.4 OpenRemote

- Partly open source (the free components)
- Designed for professional system designers rather than users
- Scripts and UI interface with their service which translates generic commands into platform-specific ones
- Software available for free, advanced features such as automated device recognition requires payment. A developer wishing to add functionality to the platform can pay a one-off fee and make the additions open source, or subscribe and keep the additions to themselves

2.1.5 Open Source Automation

- Open source
- Modular plugins required apart from standard installation. Plugins add support for different protocols and devices
- Still very early in development, not an install-and-run solution

2.1.6 Opportunities

None of the platforms above offers the following features:

1. Machine learning and suggestion
2. Mass analysis
3. Central control

2.2 Hardware research

There are many options for hardware and software available, as shown above. We wanted to make our system use different hardware components from various brands, to make the system platform agnostic. Below is some of the hardware that we researched.

2.2.1 Arduino

Arduino is a prototyping platform that supports many different hardware components. The components can be programmed using C and C++ and there is an Arduino IDE. Arduino's main advantage over other systems is the fact that it is widely supported by a large community. However, it is only a prototyping platform and might not be as stable as other solutions. It is good for some of the less common hardware that we had to build, such as the door modules, since there is a lot of support for the platform.

2.2.2 Gadgeteer

Like Arduino, Gadgeteer is a prototyping platform; however, since it is much younger, it has less hardware support and a much smaller community. On the other hand, it is much simpler to use. Due to our experience gained with it when making our proof of concept, we decided to continue using it.

2.2.3 ZigBee devices

ZigBee is a specification for high-level communication protocols, much like X10. Rather than Wi-Fi connections, they are connected via a mesh network. ZigBee is targeted at applications that require a low data rate, long battery life, and secure networking. Devices using ZigBee are usually cheaper, simpler and requiring less battery than devices that use Wi-Fi. However, these devices are low level, and it would be very hard to build the required components ourselves, especially since we have little experience with electronics.

2.2.4 X10 devices

The X10 standard is designed specifically for home automation. These devices communicate using a radio protocol, which can be both wired and wireless. Currently, this is a very widely used standard in many home automation devices; however, such devices are often severely overpriced and therefore not ideal for our aims.

2.2.5 LightwaveRF

This is a proprietary hardware and software company. They provide the user with fully functional home automation system, for controlling light, power and heating. They also provide an Open API, so anyone can use their hardware. Because of the easy setup, this was a good solution for our system.

2.2.6 WeMo

Like LightwaveRF, WeMo is a proprietary hardware created by Belkin with an iPhone app. It does not have an open API; however, it can be used without the system. It was a good solution for controlling plugs, since it meant we did not need to setup anything using the mains power.

3 Requirements and Use Cases

3.1 Requirements

1. The system should be able to change the state of plug sockets, blinds, doors and lights
2. The system should read the state from light sensors, motion sensors and buttons
3. The system should provide the capability for the user to define his own rules that control the house
4. The system should have a central server interacting with the hardware components
5. The system should have support changing items states using voice recognition
6. The system should support multiple brands of hardware through a unified system
7. The system should store data of user's rooms and items in a database
8. The system should install the updates automatically when there is a newer version
9. The system should have a cloud service for remote access and updates
10. The system should provide security for both the cloud service and API when accessed outside of the local network according to a whitelist of email addresses
11. The system should be able to measure the user's energy usage and store the data in a database
12. The system should allow users to add or remove rooms and items into the house structure through a web interface
13. The system should be able to view the state of the rooms or items in the web interface
14. The system should allow the user to change state of items using the UI
15. The system should have an open API for developers to use

3.2 Use Cases

Use Case	**Login**
Brief Description	User logs in to their account
Primary Actors	User
Secondary Actors	Server
Preconditions	None
Main Flow	1. The use case starts when the user opens the home page 2. The system logs in the user with an OpenID account 3. The system matches the e-mail address with the database or creates a new account for the user
Post conditions	User is logged in
Alternative flows	None

Use Case	Add New Room
Brief Description	User adds a room in their system
Primary Actors	User
Secondary Actors	Server
Preconditions	The user has to be logged in
Main Flow	1. The use case starts when the user chooses to add a new room 2. The user enters the name of the room 3. The system checks if the name exists 3.1. If it doesn't exist, a new room is added to the system 3.2. If it already exists, an error message is displayed 4. System updates the room structure file
Post conditions	New room was added
Alternative flows	None

Use Case	Update released
Brief Description	A new update is released
Primary Actors	Server
Secondary Actors	User
Preconditions	Already updated
Main Flow	1. The system checks for update every month from the cloud 2. If there is a new version, displays notification for updates 3. If the user accepts the update, the system will download the 4. Update and install it
Post conditions	The system will be updated
Alternative flows	None

Use Case	Send Command to Hardware
Brief Description	A command is sent to some hardware
Primary Actors	Server
Secondary Actors	Hardware
Preconditions	None
Main Flow	1. The use case starts when the server needs to send a command to some hardware 2. The server checks what type of hardware is attached to the item calling the method by checking the house structure data 3. The server calls the appropriate middle layer for the hardware type 4. The middle layer translates the API call into something recognisable by the hardware and sends it to the hardware 5. The middle layer receives the response, translates it into the correct representation for the server and returns it to the server
Post conditions	None
Alternative flows	NoResponse

Use Case	**State of a sensor changed**
Brief Description	A sensor or a button recognises that its state has changed
Primary Actors	Motion Module
Secondary Actors	Server
Preconditions	A state of a sensor has changed
Main Flow	1. The use case starts when a sensor recognises that its state has changed 2. The module sends a request to the server, informing of the changed state 3. The server checks what the user has specified should be the result from this action against the cached rules list 4. The server calls the appropriate methods according to the users settings
Post conditions	None
Alternative flows	None

Use Case	**Open/Close Blinds/Doors**
Brief Description	The blinds are opened or closed
Primary Actors	Server
Secondary Actors	Blinds
Preconditions	None
Main Flow	1. The use case starts when the server needs to open or close the blinds/doors 2. The open() or close() method are called 3. The server checks the status of the blinds 1.1. If the blinds are already in the correct state the server takes no further action 1.2. Otherwise, the server sends the appropriate command to the blinds
Post conditions	The blinds are in the correct state
Alternative flows	NoResponse, NegativeResponse

Use Case	**Lights/Plugs On/Off**
Brief Description	Switching the lights/plugs on/off
Primary Actors	User, Server
Secondary Actors	Hardware
Preconditions	None
Main Flow	1. The action is triggered when appropriate method is called by the server or the user 2. The trigger or the UI sends a command to the server 3. The server checks the status of the lights/plugs, and makes sure that they are in the opposite state 4. The server sends a command to the light/plug switch to turn on/off the lights/plugs 5. The light/plug switch sends a positive response back to the server
Post conditions	The lights/plugs are on/off
Alternative flows	NoResponse, NegativeResponse

Alternative flow	NoResponse
Brief Description	There is no response from the object
Primary Actors	Server
Secondary Actors	Hardware
Preconditions	No response after a command
Main Flow	1. The Alternative Flow starts after point 3 of the Main Flow 2. The object does not send back a response to the server 3. The Server tries to send the command again if there is still no response, the server tries ping the object 3.1. If there is no response the server sends a "Connection Error" message to the Technical Support Team 3.2. If there is a response the server sends a "No Action Error" to the Technical Support Team
Post conditions	The Technical Support Team is notified about the error
Alternative flows	None

Alternative flow	NegativeResponse
Brief Description	There is no response from the object
Primary Actors	Server
Secondary Actors	Hardware
Preconditions	No response after object is called
Main Flow	1. The Alternative Flow starts after point 3 of the Main Flow 2. The object sends a negative response 3. If there is a response the server sends a "Unknown Error" to the Technical Support Team
Post conditions	The Technical Support Team is notified about the error
Alternative flows	None

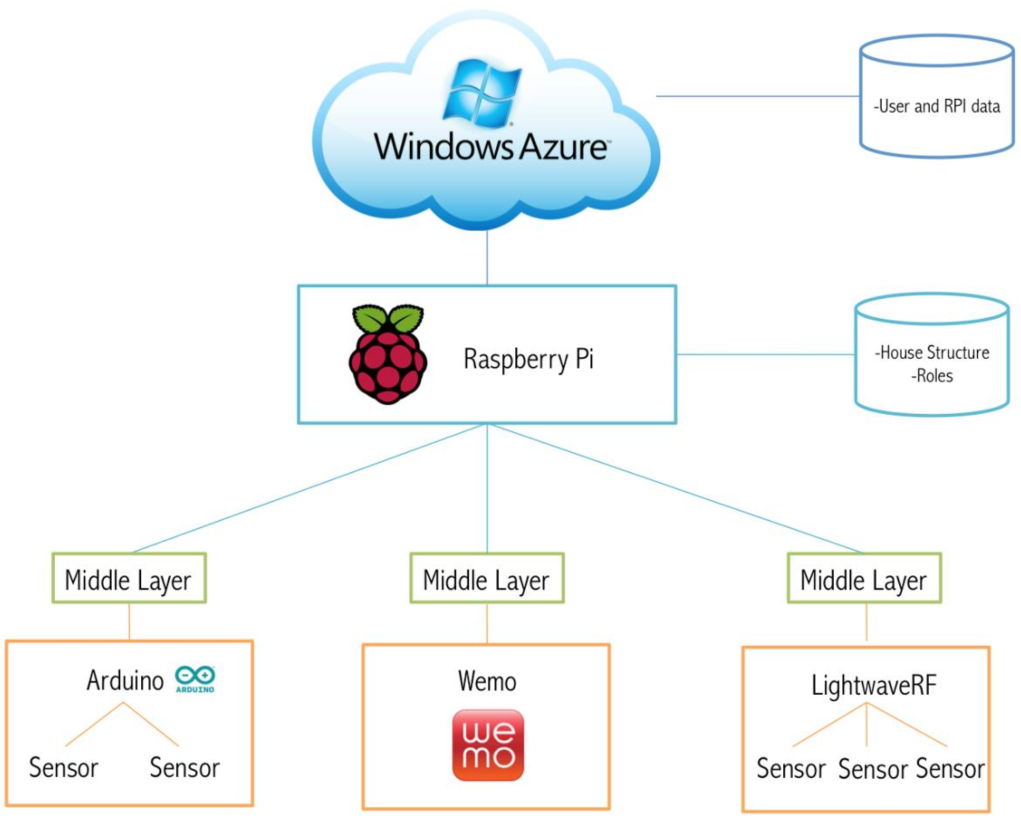
4 Design

4.1 Overview of High-Level Design

We designed the system in such a way so that it is platform independent and that anyone is able to make a user interface or application for it. This is why we focused on making a clear and concise RESTful API throughout the system and making the system as extendable as possible.

Our system is designed in a modular way and each module is made in such a way that they would be easily replaced if someone chooses to do so. On Figure 4.1 the general overview of the design is shown.

Figure 4.1 - High Level Design



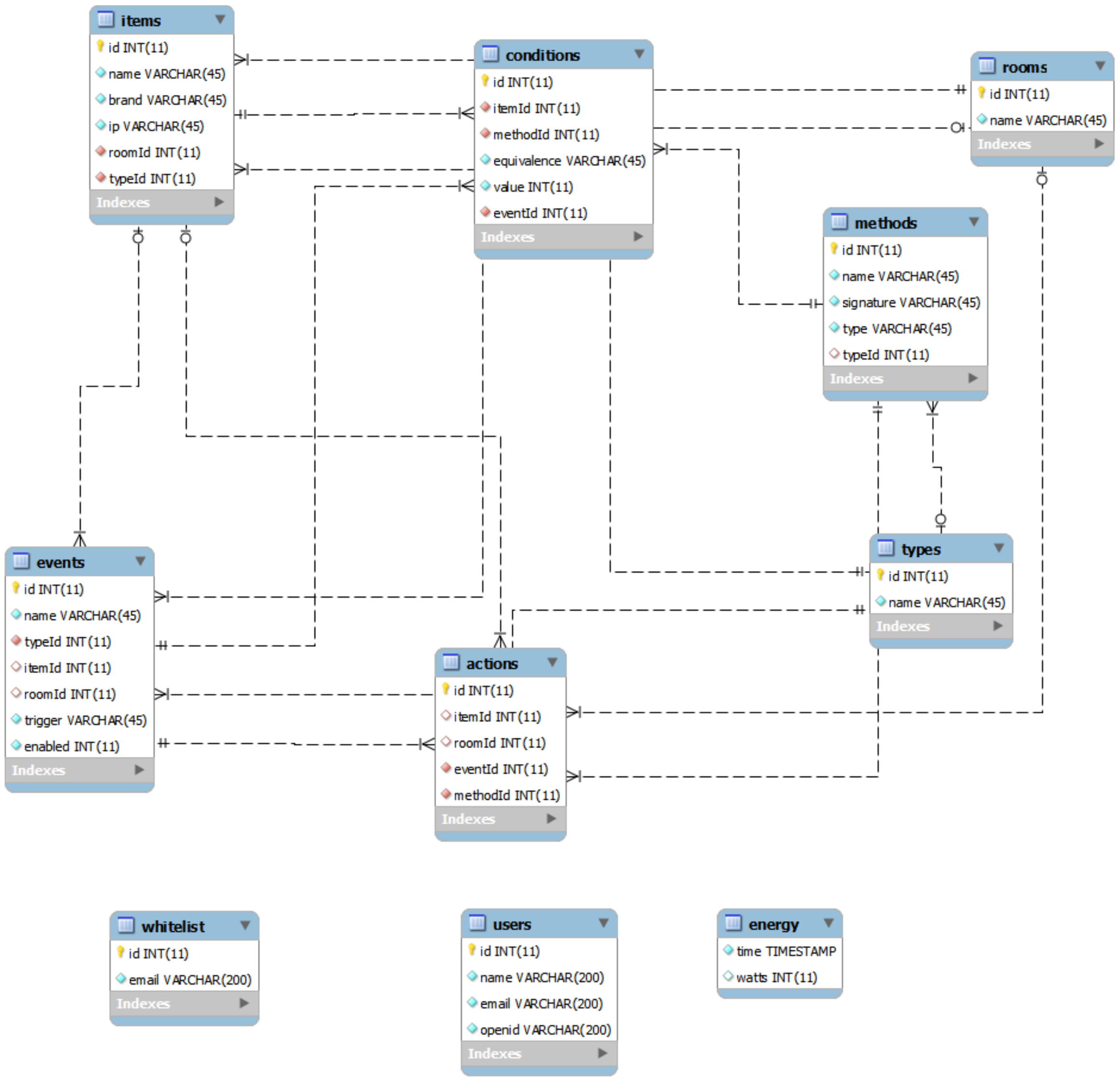
The system is designed to be very flexible and allow any developer to modify relevant parts without needing to deal with changes occurring in the other areas because of that. We thought that this could be a strength of our system and thus one of the main features that we implemented is the middle layers. Thanks to them, anyone can add support to a new brand of hardware without changing anything outside of this interface.

All of the middle layers are managed centrally via Python on the Raspberry Pi. It also integrates a Flask server, which hosts our UI. Here also we wanted to make it as flexible as possible, therefore we developed an open API, so that anyone can make a UI for the system.

4.2 Database Designs

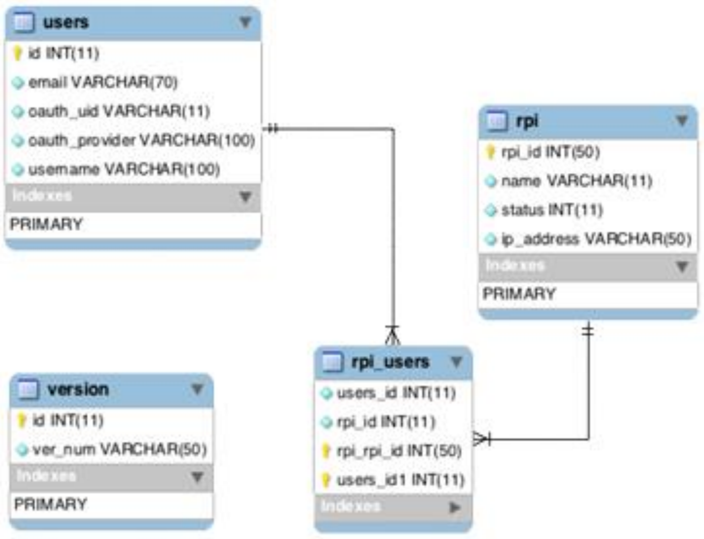
4.2.1 Raspberry Pi Database

Figure 4.2 - Database on Raspberry Pi



4.2.2 Cloud Database

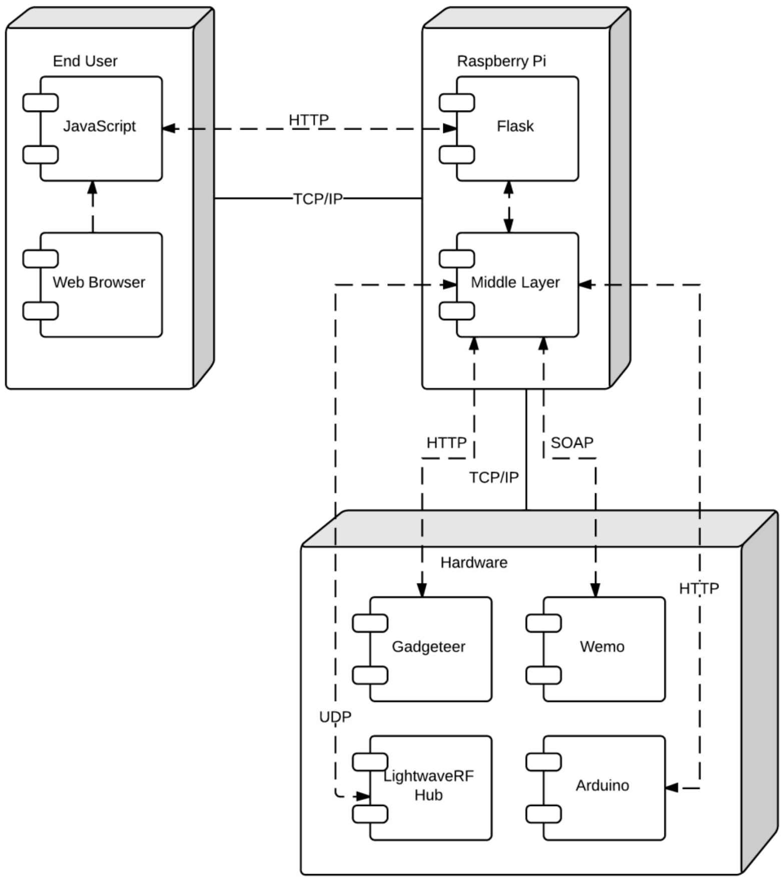
Figure 4.3 - Cloud Database Design



4.3 Components Description

Figure 4.4 shows the major components and communication protocols used to communicate between them.

Figure 4.4 - Components of the system



4.3.1 User Interface

The system is designed in such a way that anyone can make a user interface for it on any platform. It can for example be a web interface or a mobile app. The only requirement is that it has to be able to send HTTP requests to the server. The whole process is done through a RESTful API, which can be seen in Appendix B. The responses from the server are in JSON format.

The current system has a web interface that can be loaded on phones and tablets. Our native user interface is run on the Raspberry Pi.

4.3.2 Cloud Service

Because the Raspberry Pi server is run locally, we also made a cloud with user accounts for storing the user’s Raspberry Pies and their IP addresses. That way the user can name them and in case they have several homes, they can manage and access them all remotely. This service communicates with the Raspberry Pi via HTTP requests.

4.3.3 The Raspberry Pi

The central system runs on the Raspberry Pi, which acts as a server. The server listens for RESTful HTTP requests from the user interface and then, depending on the request, acts on the database/hardware or sends back requested information using JSON. The system is designed to be event driven to ease network load due to polling.

One feature that is worth noting is the middle layer system we implemented. Each type of hardware component in the system has its own middle layer, which means that the rest of the code is uniform regardless of the type and brand of the hardware. We believe this is a very useful feature, since it allows the system to be extended easily.

4.3.4 The Hardware

The server on the Raspberry Pi is using a variety of methods to communicate with the hardware components. Because we want to include many different devices from different brands, such as Gadgeteer, Arduino, WeMo and LightwaveRF, there is no uniform way to communicate with them all. If we have the choice, we try to stick to an event driven system, thus using HTTP requests from and to the hardware device. For proprietary components, this is rarely possible and the communication is done in a way the company that made the hardware designed it to be. Therefore, for LightwaveRF we are using UDP to communicate with the central Lightwave hub, which then uses its own protocol to communicate with particular hardware components and WeMo communicates through SOAP using XML.

5 Implementation

5.1 User interface

5.1.1 Overview

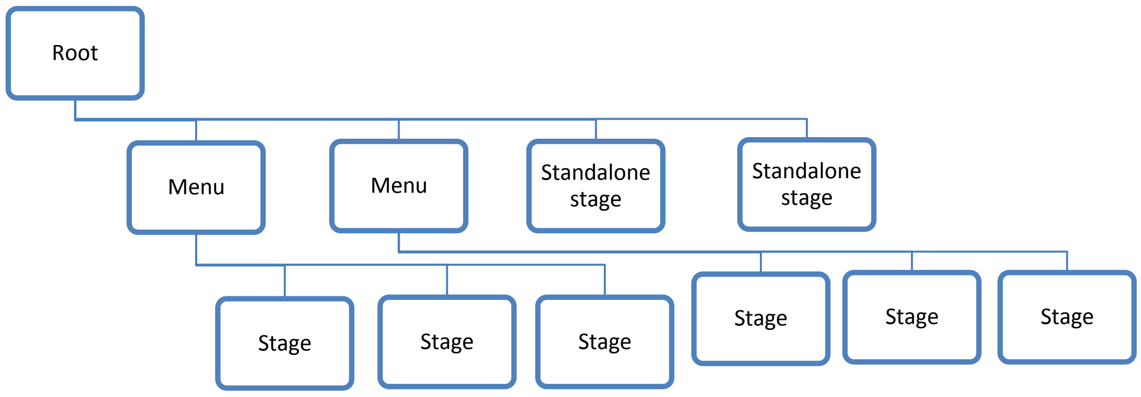
We started in COMP2013 with a straightforward and lightweight implementation – just JavaScript with jQuery, and that design was carried forward into COMP2014. Ideally, if we knew how much the scope would have grown at the time of this report, we might have been better off building the UI based on a heavyweight framework such as Google's closure library and made use of their built-in event handlers and so forth. That would have made the code more extensible if someone else were to take it and add unrelated features or completely rebuild it.

The UI consists of a single AJAX-driven page. The skeleton of this page is defined in terms of Jinja2 templates, which the Flask server would interpret and serve. As with any AJAX-driven website, we wanted to implement AJAX history to improve it at one point, but we abandoned the idea in favour of getting the UI ready for all our required features. That design takes away little in terms of user experience since the hierarchy is, by design, highly branched and the user is supposed to find the right place within 2 clicks.

5.1.2 Hierarchy Design

Since the UI technically only has one page, the different parts of the UI are defined in terms of Stages. These stages are custom JavaScript objects defining how the UI is supposed to be defined and constructed based on information received from the server.

Figure 5.1 - Design of the UI

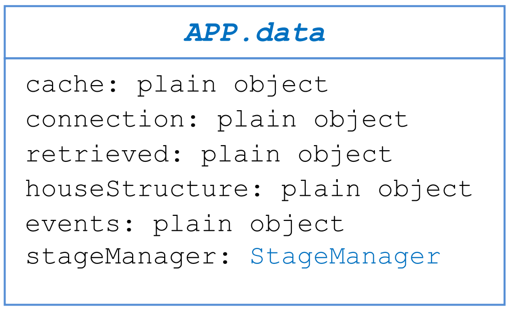


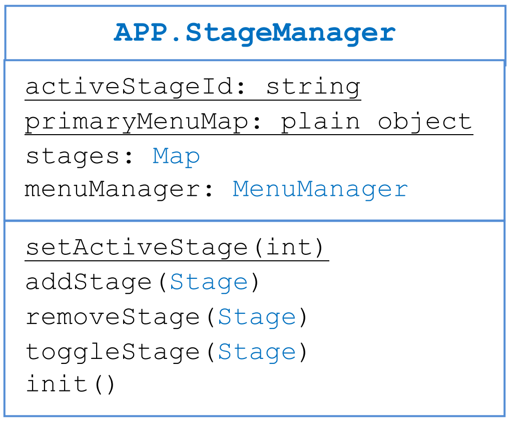
5.1.3 Main Classes and Initialisation

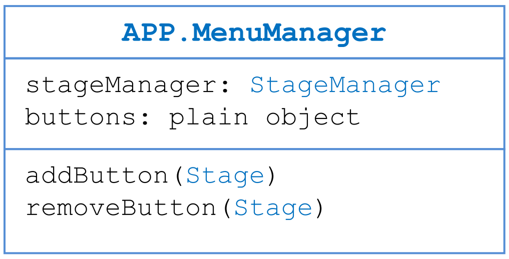
When the document is ready, the script does four main things. It initializes an instance of the `APP.StageManager` class and calls its `init()` method, and runs the methods defined within the self-explanatory static classes `APP.clock`, `APP.windowResizeListener`, and `APP.resizer`. This starts up the clock and does whatever resizing cannot be handled via CSS, e.g. adjust the font-size in response to screen width changes.

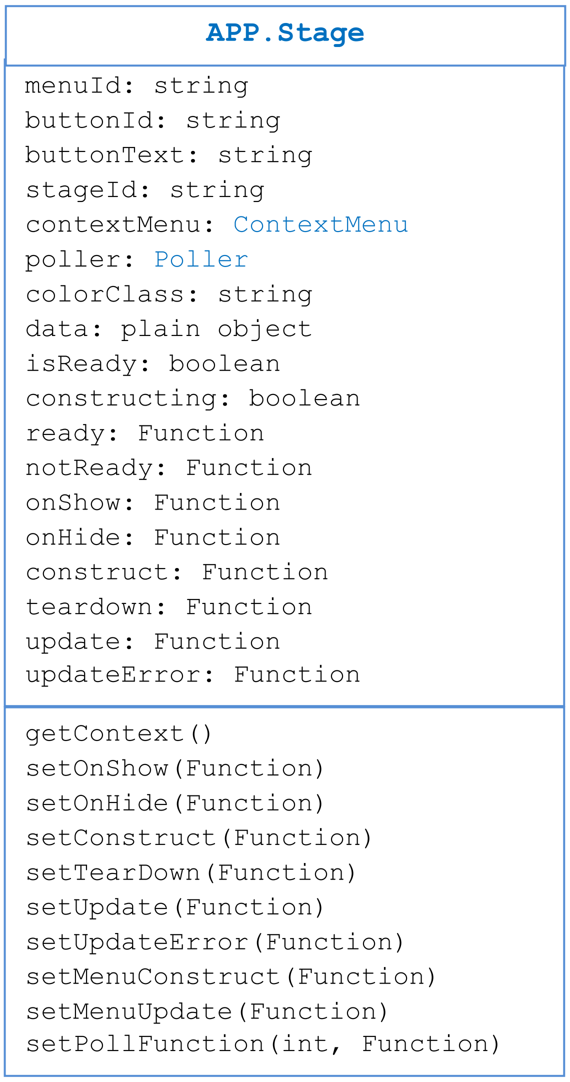
`APP.StageManager` is responsible for adding/removing stages to the UI and toggling whichever one should be active at any given time. It also contains an instance of the `APP.MenuManager` class that initializes the menu and handles which buttons and menus should be corresponding to the active stage.

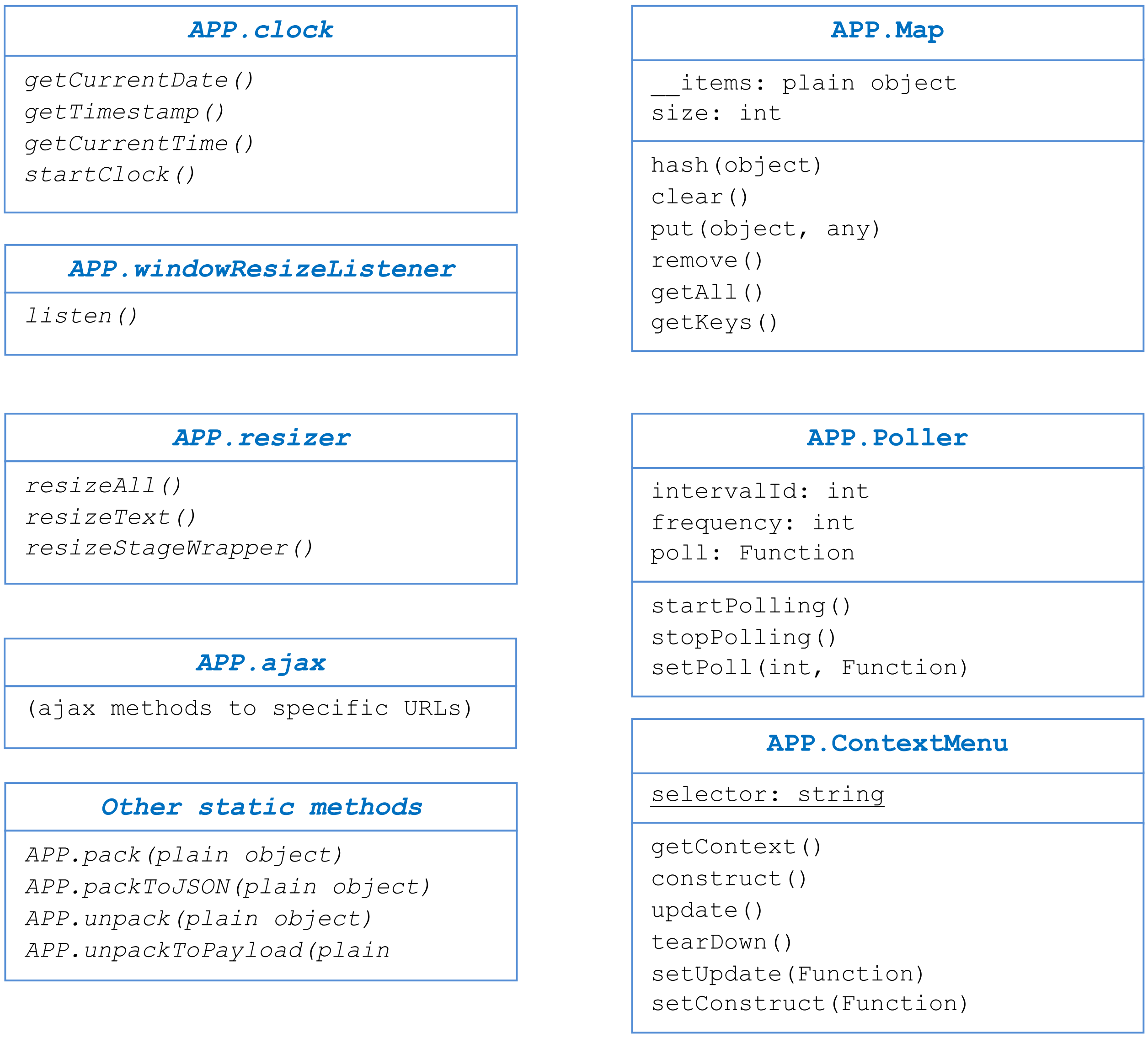
The `APP.Stage` class defines a stage object, which `APP.StageManager` can use. Since most of the functionality of a stage depends on what it is supposed to do, `Stage` has a few methods that specify some default behaviour for all stages that cannot be overridden, for example, what to do when the button for that stage is clicked. It then offers several callbacks for the programmer to specify, so a function can be passed in to be called when, say, the `onHide()` method is called for a particular stage.











The other classes not fully outlined above handle the UI components of specific stages. These are:

- `APP.ItemTypeDisplay`
- `APP.ItemDisplay`
- `APP.ECARuleManager`
- `APP.ECARuleDisplay`
- `APP.ECANewRuleDisplay`
- `APP.ECAEventDisplay`
- `APP.ECAConditionManager`
- `APP.ECAConditionDisplay`
- `APP.ECANewConditionDisplay`
- `APP.ECAActionManager`
- `APP.ECAActionDisplay`
- `APP.ECANewActionDisplay`

5.2 Python Backend

5.2.1 Flask

The entire system is run on a RaspberryPi using Python, therefore choosing a Python based server was an obvious choice. Flask was one of the many servers that we researched. The others included Django and CherryPy. We chose Flask because it is lightweight, has a nice debugger and unittesting tool and RESTful request dispatching. The fact that RESTful support was native and easy to use was very important to our project since our entire API is REST.

The Flask server hosts the UI and listens to RESTful requests, which are then delegated to methods in different classes of the system. There are many requests that can be made to the API, all of which are found in the documentation in Appendix B. An example request that can be sent to the flask server is:

```
<ip>/version/0.1/rooms/1/items/1/open/
```

This request is used to call method “open” on item “1” in room “1”. For example, open a door in the lounge. This would be caught in flask with the rule:

```
@app.route('/version/<string:version>/rooms/<int:roomId>/items/<int:
itemId>/<string:cmd>/', methods=['PUT'])
```

The method for handling this request is:

```
def rooms_roomId_items_itemId_cmd(version, roomId, itemId, cmd):
    if g.user is None and not isIpOnLocalNetwork():
        return redirect(url_for('login'))

    args = request.args.to_dict()
    if('test' in args):
        return parrot(request)

    if request.method == 'PUT':
        # Command item
        house.addToQueue(int(roomId), int(itemId), cmd)
        return jsonify(pack('success'))
```

This method has three sections, each of which is important to our system. The first part checks if the user sending the request is logged in or is on local network. If that is not the case, the site will redirect them to the login page, which prompts them to give the permission to share the information from Google OpenID.

The second part is used for integration testing done with QUnit. This sends back a `[200] OK` response back to QUnit if the given rule is supported by the API. This can be used by any developer that wants to use our API to communicate with the system.

The third part checks if the request is of the correct type. This particular example supports only PUT requests, however some of our other methods support several kinds, which can all be seen in the API documentation. This adds the command to the priority queue, explained below, and returns a `[200] OK` response.

The security of our system is an important factor. We chose Open ID because that way we can make our system secure, without worrying about storing passwords. We are using a whitelist to store the e-mails of users that can access our system. The first user that logs in is automatically added to the whitelist, any further users that want to use the service need to be added to the whitelist by one of the already registered users. This way we can assure that only authorised users can access the system.

Authorisation is not necessary if a user is on the local network, since we are assuming that anyone accessing the local network is a resident in the house. This way we can keep the need to login to a minimum, improving the user's experience. This is checked by the `isIpOnLocalNetwork()` function.

5.2.2 House System

The House System is made up of the House class and the Room class. The House class handles most of the methods from the Flask server. It is the main class for handling actions on the house and its components. The House class has a list of rooms and a list of events.

The functionality of the House class includes:

- Adding, editing and deleting rooms, items and events from the system
- Getting the state of each item, room and the entire house
- Handling API commands such as /version, /version/<version number>/state and /version/<version number>/structure
- Adding actions to the priority queue

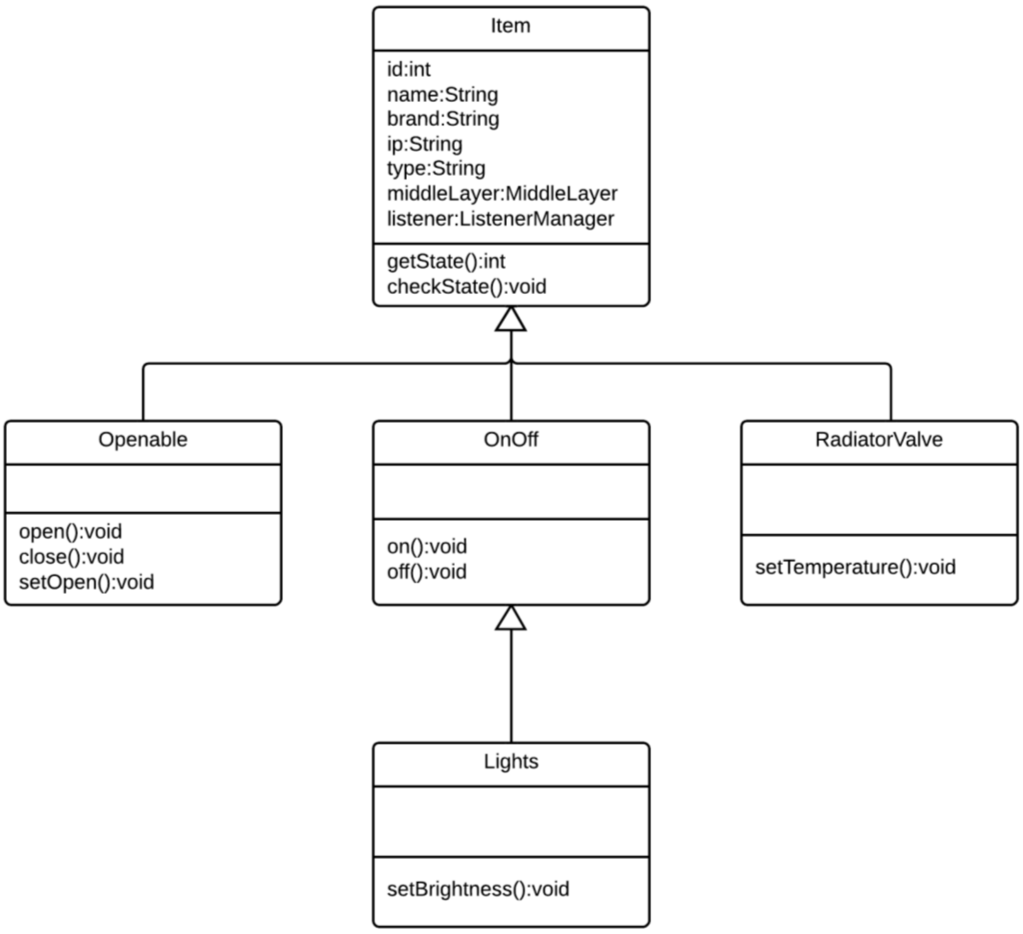
The system is designed to be always operating, however because sometimes unpredictable events occur, such as power failure, we store all the data the user configured on setup (as explained in the User Manual, Appendix A) in a database. On start-up, the House class restores everything from the database into memory.

The queue is made to store the requests from the users and execute them according to their priority. We made this in case that there are too many requests for the system to handle at once, because of either slow internet or request overflow. We chose priority over the timestamp, because some of the requests may be done slower than others. For example, opening the door should happen instantly whereas switching the light on can be done a bit later.

The Room class contains methods that act on a particular room. This class deals with adding items and returning the structure and state of a given room. Its instance variables include the ID of the room, the name of the room and a list of items in that room.

The Item class is a class that controls the behaviour of a particular item. Its main role is to check that the given item supports the required action and then it should transfer the command to the appropriate middle layer. Each item has an id, name, brand, IP, type, reference to a middle layer and reference to a listener. The item class has many subclasses and each subclass has different behaviours, as shown in the class diagram in Figure 5.2.

Figure 5.2 - Class Diagram of Item Objects



All items support a `getState()` method and a `stateChanged()` method. The `stateChanged()` method is very important, since this is what is used to make our system event driven. When the state changes it notifies the listener, which should react to it accordingly. All the methods in the subclasses and `getState()` method act in the same way, by sending the action to the middle layer. For example, `setOpen()`:

```
def setOpen(self, percentage):
        return self.middleLayer.send('setOpen', percentage)
```

5.2.3 Database

The database is created through a script on the Raspberry Pi rather than through the Python code. Python only adds, deletes and edits the elements in the tables.

In Python, the connection to the database is made using the MySQLdb library. [1] Our implementation is made using several classes. First, there is a DatabaseHelper class for the simple database methods, such as execute query, add entry, remove entry, update entry and retrieve data. Then there are classes for each of the tables, which have more specified functions. This is all contained within the Database class, which interacts and coordinates all the classes and is the only one accessed by the rest of the system.

[1] https://pypi.python.org/pypi/MySQL-python/1.2.3

5.2.4 Events and Rules

Users are able to define rules so that they can automate their house. Events and rules are handled using an Event Condition Action method. When a state change occurs, the system checks the list of events to see if any of the rules are eligible to be triggered. If this is the case, it checks that all of the conditions pass and if they do it will process the actions associated with that rule. For example:

Event: *Door A* is opened
Condition: *Light sensor A* detects no light
Action: Turn on all of the lights in the lounge

This is one of the simplest of cases. If someone opens the door and it is dark then the lights in the lounge will turn on. Note that the lights will not turn on just because it is dark - the door needs to be opened first. Additionally, if the door is left open it will not trigger this rule.

Whenever a state change occurs in the house, the `ListenerManager` is informed by the Item object. `ListenerManager` allows us to register methods to be called upon a state change so that we can easily expand the functionality and number of features reacting to state changes in the future. An IP address and an event name (e.g. “opened”) are passed on, which is enough to identify what has happened and where.

The system has a `reactToEvent()` method, which is subscribed to `ListenerManager`, and a list of event objects, which in turn has lists of condition and action objects inside. Once notified of a state change, the system checks to see if any of the user-defined events should be triggered. It does this by matching the IP with an item and then checking which events contain both a matching trigger name and that item. It makes a list of possible events with these matches and then proceeds to evaluate the rules within.

An issue to consider is whether to select only the first match or continue to select all matches. We wanted to make the system as flexible as possible and therefore chose to try to match as many events as possible, which raised the new issue of conflicting events. For example, it is possible that a user could define two rules such that they both have the same event and conditions but one opens a door whereas the other closes that same door. We handled this by adding a method called `isConflictWithOtherActions()` to the action class, which takes a list of the matched actions so far and will only return false if there are no conflicting actions. If any single action within an event conflicts with an existing action then the entire event will be discarded. This allows us to match as many events as possible whilst still restricting unusual behaviour from the system. The code below shows the system removing conflicts from the events to be evaluated.

```
for event in possibleEvents:
    eventItemsActedOn = []
    eventMatch = True
    for action in event.actions:
        if action.isConflictWithOtherActions(itemsActedOn +
eventItemsActedOn):
            eventMatch = False
            break
        else:
            eventItemsActedOn.append(action.getItemsActedOn())
    if eventMatch == True:
        itemsActedOn.extend(eventItemsActedOn)
        events.append(event)
```

The system does not currently backtrack on conflicts, which would be a possible improvement for the future. For example, with the rules given above the system would match the first rule and ignore the second but it is possible that the condition check on the first rule would fail and so it would be desirable to backtrack and match the second rule. It would be time consuming to implement this since such conflicts are very rare and only occur with badly defined rules.

5.2.5 Plugins

Our system is open source and therefore developers are able to make changes and additions as they see fit. However, we also wanted to provide the option to develop plugins so that users can upload these to an existing system without the hassle of downloading another developer's forked version. Developer documentation on the specifics of writing a plugin is provided both online and in Appendix B.

Plugins are written in Python and must extend the Plugin class that we provide. This includes methods such as `setup()` and `teardown()`, which are called at the appropriate times. When a user uploads a plugin zip file the system will extract it and place it in the correct folder. It then looks for subclasses of Plugin and loads them into the system.

Each plugin has a unique name that is used to access it from the URL …/plugins/name. When a request for this URL is received by the server, the system will request a stream of HTML from the plugin. It is also possible to extend the path further and it will be passed onto the plugin so that different responses can be offered.

Plugins also have access to the current list of items, rooms and events in addition to the queue to process state changes within the house. By default, plugins are also subscribed to `ListenerManager` so they are notified when the state changes. This means that it is possible to add large amounts of functionality through a plugin, which gives a developer almost as much freedom as forking the repository and changing the existing code.

The largest issue with plugins currently is that they perhaps give too much power to developers. Unlike large commercial projects, which would limit the use of certain functionalities and perhaps even have their own libraries to be called on, our system simply blindly runs Python code providing it extends our Plugin class. This means that it is possible to write malicious code in a plugin and that a badly written plugin could interfere with our existing code or even crash the system. Whilst this is a concern, it is a much larger task and out of the scope of the project. For the time being, developers should be trusted.

5.2.6 Middle Layers

The middle layers are an important part of how we managed to make the system device independent and allow for easy addition of new hardware. When an item is initialised, it creates an object which subclasses `MiddleLayer` according to its brand. Each middle layer contains specific instructions for interacting with hardware and must implement all methods required by the item types it supports. For example, if we have a new brand of hardware that supports doors it must have open and close methods. More details on specific middle layers can be found in their respective hardware sections.

Although it is possible for some of the hardware brands such as Gadgeteer and Arduino to be truly event driven, it is extremely difficult to make the proprietary hardware do this since it would involve custom firmware. To solve this problem, the `MiddleLayer` class spawns a new thread upon creation, which continuously polls the state of the hardware every 2 seconds using the `checkForStateChange()` method and acts upon changes in state. If there is a change then it calls the `stateChanged()` method of the item in another new thread and then continues to poll. The thread that calls `stateChanged()` then notifies the listeners and executes any results of this change of state. This layer of abstraction allows the rest of the system to act as if all hardware was event driven.

```
def checkForStateChange(self):
    while True:
        realState = self.checkState()
        if realState != self.state:
            self.state = realState
            t = threading.Thread(target=self.item.stateChanged,
args=[realState])
            t.daemon = True
            t.start()
        time.sleep(2)
```

5.2.7 Automatic Updates

The Raspberry Pi is able to automatically check for and install updates. When a request is sent to `…/update`, the system will start the update process. To avoid abuse of this feature it will first check if there is actually a new version before downloading anything new. It does this by comparing its current version to the one provided by the cloud system[2].

If there is a new version, it tries to retrieve a zip folder from our servers. In order to replace the older files with the newly downloaded zip folder, the current process needs to be killed. The system gets the process ID by using the method `getpid()` and writes it to a file for later use by a shell script (runUpdate.sh) which will be called. The script then unzips the file, kill the process listed in the file, replace the old files and finally restart the server.

[2] http://robohome.co.uk/version.php

Initially deleting the old files caused issues when developing since our recent changes were not committed and downloaded in the new version. We solved this by creating a backup folder, "webOLD", as part of the process. We rename the "web" folder to "webOLD" and move the new folder, "comp2014-master/web", into the web folder. Below is a sample of this code.

```
echo "\n Delete previous backup...\n"
rm -rf ./webOLD

echo "\n Replace files... \n"
mv ./web ./webOLD
mv ./comp2014-master/web ./web

echo "\n Deleting files...\n"
rm -rf ./github.zip
rm -rf ./comp2014-master
```

5.2.8 IP Management

The cloud service must always have the correct Raspberry Pi IP address. The cloud service requires an ID to change the IP so before anything else the system must know what its ID is. We implemented a method to first check if the .txt file (id_RPi.txt) exists and if there is not it creates one with the ID obtained from the cloud service. The system then polls its IP address to check if it has changed.

To see if a Raspberry Pi's ID has already been saved in a text file we check the size of the file. There are two possible cases:

1. If the size is zero the file has just been created and has no ID in it. In this case, we get the ID returned from `‘http://robohome.co.uk/ip/rpi_id.php?ip=’ + my_ip`. The ID is then written to the empty file.
2. If the size of the file is not zero then open the existing text file, read the ID which has been stored in it and send a request to update the IP if it is different

Here is some example code for handling the file:

```
if((os.stat('./id_RPi.txt').st_size) == 0):
                get_ID = urllib2.urlopen('http://robohome.co.uk/ip/
rpi_id.php?ip=' + my_ip).read()
                my_id = get_ID.strip()

                f = open('./id_RPi.txt', 'w')
                f.write(my_id)
                f.close()

else:
                f = open('./id_RPi.txt', 'r')
                my_id = f.read()
                f.close()

urllib2.urlopen('http://robohome.co.uk/ip/ip_update.php?id=' + my_id
+ '&ip=' + my_ip)
```

5.2.9 Energy Usage Data

In addition to displaying the current energy usage, which is limited by our use of binary states, we wanted to provide the user with a way to view continuous data, including the history, with a graph. The LightwaveRF energy monitor returned continuous data anyway which was being converted to a High/Low state to work with the existing interfaces. We used this to our advantage and stored the current continuous state in memory before conversion so that the object knew its precise state internally. Then, upon initialisation of a LightwaveRF object that was an energy monitor, a thread would be spawned to store this data in the database periodically.

Finally, we added the necessary API methods for the web interface, or any other component interacting with our system, to retrieve this energy usage data between a pair of timestamps. Whilst the current user interface simply requests all of the data, the option for specifying only partial data was given for software such as the mobile apps with slow connections.

5.3 Cloud Service

The cloud service[3] is hosted on Window Azure. Its main purpose is to allow users to manage multiple Raspberry Pis and forward them to the correct IP to allow for users not having static IP addresses. In addition to this, it allows Raspberry Pis to check for the latest version of the software and download it. The website was built using Twitter's Bootstrap framework and PHP. It uses some library code to allow users to login using OpenID.

When a user adds a Raspberry Pi, they specify an IP address for it. The server then checks to see if this IP address is already in use elsewhere on the system. If it already exists in the database (e.g. the case where many users in an office have added the same Raspberry Pi to different accounts), the system names it appropriately and does not create a new entry in the database. If it is a new IP then a new entry is added and it is assigned a new ID that the Raspberry Pi will collect later.

The cloud service also allows Raspberry Pis to automatically update their IP address in the event that they have changed. It does this by changing the correct database entry when a request is sent to a specific URL. Currently there is little security with this feature and it would be possible to maliciously guess the correct URL for a Raspberry Pi and change its IP to a phishing site. This would be a priority for further development.

5.4 Mobile apps

Our UI is made to be available through any web browser. However, for ease of use we also decided to make mobile apps to manage greater functionality. We made Android and Windows Phone apps.

The Android app shows the web UI in a web view, and provides additional settings, such as changing the address of the page. It also adds voice recognition feature, described below.

The Windows Phone app parses JSON from the Flask Server on the Raspberry Pi and shows it in a readable format, using json2csharp[4]. Thanks to this, the user does not need to load up the UI, which means less data usage on a phone.

[3] http://robohome.co.uk
[4] http://json2csharp.com

5.5 Voice recognition

One of the additional features of the system is voice recognition. Because of the simplicity of the Google API and the fact that we were planning to do an android app anyway, we decided to make use of the voice recognition API for Android.

The Android app provides a button for voice recognition. The commands for voice recognition have to be in the format `<room name> <item name> <action>`, for example "Lounge Door Open"

This command is then parsed and the room name and item name are changed into IDs for the REST commands to work. To achieve this we are getting the data from `…/version/<version>/state` and parsing the result (shown in Appendix B).

To improve the accuracy of the result we are using the Google API voice recognition function, which returns not only the best match, but also other possible matches. With this, each of the matches is compared to the list of rooms and items, and if a match is found the correct RESTful command is sent to the server.

5.6 Hardware

5.6.1 .NET Gadgeteer

The system currently supports Gadgeteer light sensor, button and PIR modules that are very similar in implementation and will be discussed together. They start by connecting to the Wi-Fi network using the provided libraries and then start a web server that can return the state of the hardware. Upon a state change, they will inform the server by sending a request to the URL …/gadgeteer/state/stateNumber. When the server receives a request of this type it will check the IP it was sent from and change the state of that item accordingly.

We also greatly improved the networking of the Gadgeteer modules from our proof of concept in COMP2013 and switched them to using a wireless connection. The web server library was changed to a Gadgeteer specific one rather than the .NET Microframework one we used previously. With the exception of this and the integration with middle layers, they remain largely the same as our proof of concept.

5.6.2 LightwaveRF

For lighting control and energy monitoring, we used LightwaveRF hardware. This hardware works by communicating wirelessly with a central hub, which connects to the local network. It is a good example of the need for the middle layers since it shows how different the methods for interacting with some of the proprietary hardware can be. Additionally, the middle layer listens for UDP messages broadcasting the current energy usage.

The central hub assigns each piece of hardware a device ID and then forwards commands using its own wireless protocol according to the commands sent to it. The result of this is that IP's cannot be unique within our system; all LightwaveRF hardware will have the same IP since the individual devices are not actually on the local network. We fixed this problem by requiring that users specify the IP for LightwaveRF hardware in the format "192.168.0.1:RX:DX" where X is replaced by the room and device numbers registered with the central hub.

We were given access to the API for the central hub which contains command IDs to be sent using UDP. It is also possible to add a message to display on the hub's screen; these were left null since the intention is for the hub to be stored out of sight. The following is an example command:

```
def sendToWiFiLink(self, roomId, deviceId, commandId, messageTop,
messageBottom):
    self.sock.sendto("533!R%sD%sF%s|||" % (roomId, deviceId,
commandId), (self.ip, 9760))
```

This command first requests access to the hub (by prefixing "533!"), which the user will need to confirm manually on the hardware for the first use, then specifies the device and finally gives a command ID according to the API.

5.6.3 WeMo

Belkin's WeMo hardware is a good example of how we support hardware that is easily available for users. We were able to find existing Python code[5] to communicate with the WeMo hardware but we did spend some time stripping out a large number of unnecessary code so that we were left with just what our system required to turn it off and on and check the state.

We further edited this code to create a helper class for the middle layer to import since the resulting code was still rather large and added the ability for multiple connections to different WeMos around the house. This class contains the `start()` method which takes the IP of the WeMo and calls the appropriate library methods and is called during initialisation of the middle layer.

5.6.4 Arduino Wi-Fi Library

Where Arduino hardware is concerned, all communication is done via the Wi-Fi Arduino Shield[6]. The pre-defined functions that can be used with the Arduino Shield are basic, therefore to make implementation of different modules easier we decided to make an additional library to suit our needs.

For communicating with the Arduino modules we are using a RESTful API, similar, but a lot simpler, to the open API we are using for communicating to the Raspberry Pi. For example, a door module would implement commands:

```
<ip>/state
<ip>/open
<ip>/close
```

(Where `<ip>` is the IP of the module)

The default commands only include reading characters from the request and writing characters back to the response. Therefore, for our library, we implemented a RESTful parser for parsing the requests and delegating them to the appropriate methods and a JSON generator for the response. It also connects to the network and listens for requests. The RESTful parser is based upon code from Jotschi's Blog[7].

[5] http://www.issackelly.com/blog/2012/08/04/wemo-api-hacking/
[6] http://arduino.cc/en/Main/ArduinoWiFiShield
[7] http://www.jotschi.de/Technik/2012/04/21/arduino-ethernet-shield-simple-rest-api-example.html

The library connects to the specified network (provided in the constructor) and listens for requests. Once there is a request, it parses it and returns the command. Using the door example above, it would return “state”, “open” or “close”. Then it sends back a JSON response with the specified result. For example for state, it would be `{state : 0}.`

We wanted to make the Wi-Fi library not only listen for requests, but also send a request back to the server when the state changes, however this was not possible because of hardware constraints. The Wi-Fi Shield cannot cope with serving a server and listening to a different server at the same time.

5.6.5 Motor Shield

The Wi-Fi Shield shares pin 12 with the Motor Shield, which is why for the hardware that needs both (door and blinds modules) we had to adjust the Motor Shield. We connected it to pin 9 using some additional wires. Finally, we changed `#define MOTORLATCH 12` to `#define MOTORLATCH 9` in motors.h.

5.6.6 Door Module

Moving a door automatically is a complicated task. Our key use case for the door module is:

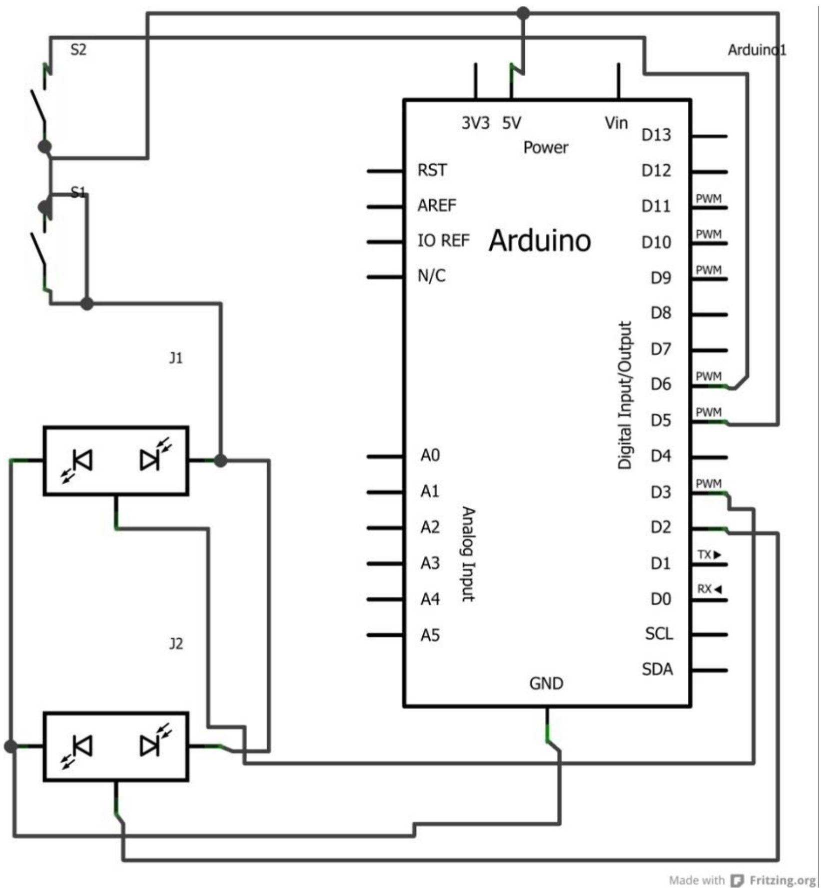
1. A user approaches the door
2. The door detects the user and opens the door
3. The user passes through the door
4. The door closes

This short use case proved to be much more problematic than we first thought. The hardware used in the door module is:

1. Arduino Mega
2. 2 PIR sensors to detect when someone is close to the door
3. 2 Magnetic switches to detect when the door is open
4. 1 Magnet
5. Stepper motor
6. Wi-Fi Shield
7. Motor Shield

Figure 5.3 shows the schematic for the module. For simplicity reasons this does not include the Motor Shield and Wi-Fi Shield.

Figure 5.3 - Door Module Diagram



The general idea for the module operation:

1. The motion sensor detects the motion
2. The motor is started
3. Once the door reaches the reed switch on the wall, the motor stops
4. Wait 30 seconds
5. The motor is started
6. Once the door reaches the reed switch on the frame of the door the motor stops
7. The module goes back into the “listening” stage awaiting movement

The placement of reed switches, motion sensors and magnet is crucial for the accurate execution of this module. The module needs to know which reed switch is signalling that the door is opened and which is signalling that the door is closed. In addition, the magnet has to be placed in such a way that it touches the appropriate reed switch when the door is fully open or closed.

The main problem was the motion sensor. We did not want it to detect too much motion; therefore, we bought a short distance, which is not very sensitive. On the software side, we included timeouts, so that it does not recognise the same motion as two motions.

To communicate with the server it uses the Wi-Fi library previously explained. It supports the “open”, “close” and “state” commands. This is used in addition to the automatic opening and closing on detected motion. To open the door the following code is used:

```
void openDoor() {
  while(currentOpen!=1) {
   motor.run();
   currentOpen = readReedSwitch(reedOpenPin);
   }
   motor.stop()
}
```

A similar snippet is used for closing the door. The state is checked by reading the states of the two reed switches. At the moment if the door is in an undefined state (e.g. someone manually left it half opened) it is shown as closed.

5.6.7 Arduino Blinds Module

The Arduino blinds module is similar to the door module, in a sense that it uses both the Motor Shield and the Wi-Fi Shield. It is also similar in principal, however here we do not have the motion sensors. This module has no automated behaviour, unlike the door module. In addition to this, the Hall Effect sensors replaced the reed switches.

The code concept for the code is also similar. The motor is started when the user clicks the button and stopped when the desired position is reached i.e. when the other Hall Effect sensor detects a magnetic field.

The motor requires a 12V power supply, which needs to be supplied externally rather than through the Arduino since it will only handle 5V. Initially, the motor shield had issues with overheating after a short period. This was because the H-Bridge was not designed for a 12V supply and so the default H-Bridge was replaced with a new[8] one that was capable of this.

The schematic and product are shown below. The motor and Wi-Fi shields are shown in the image rather than the schematic.

[8] http://download.siliconexpert.com/pdfs/2008/04/20/urlc/txn/sn754410.pdf

Figure 5.4 - Arduino Blind Module Schematics

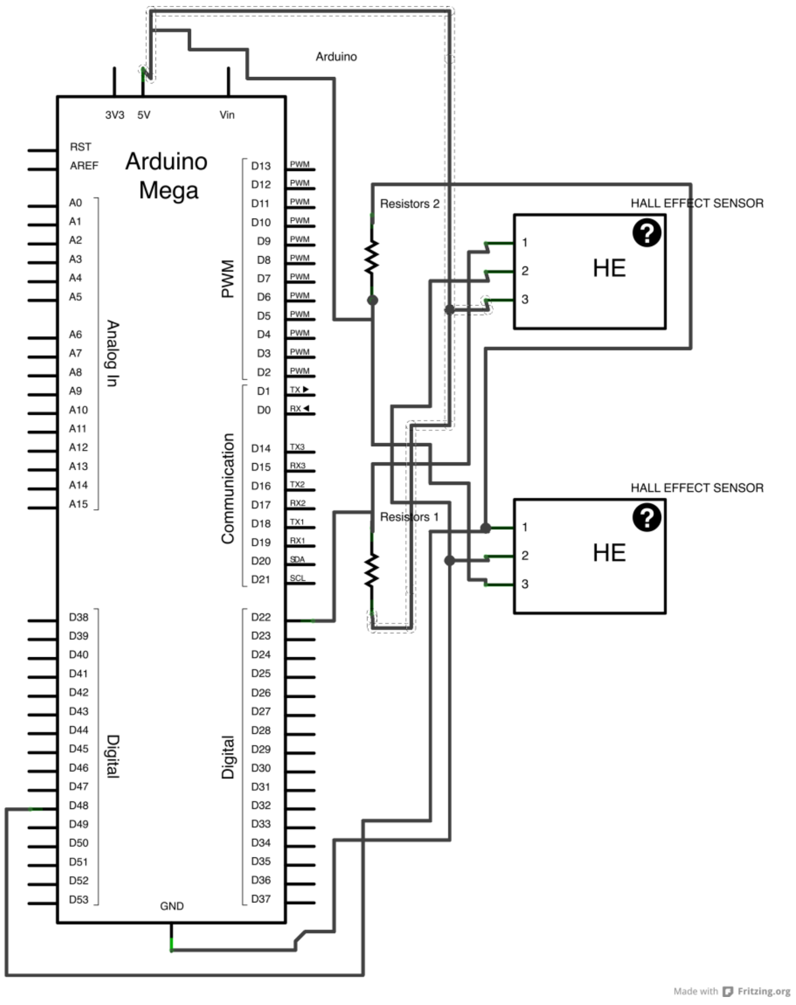
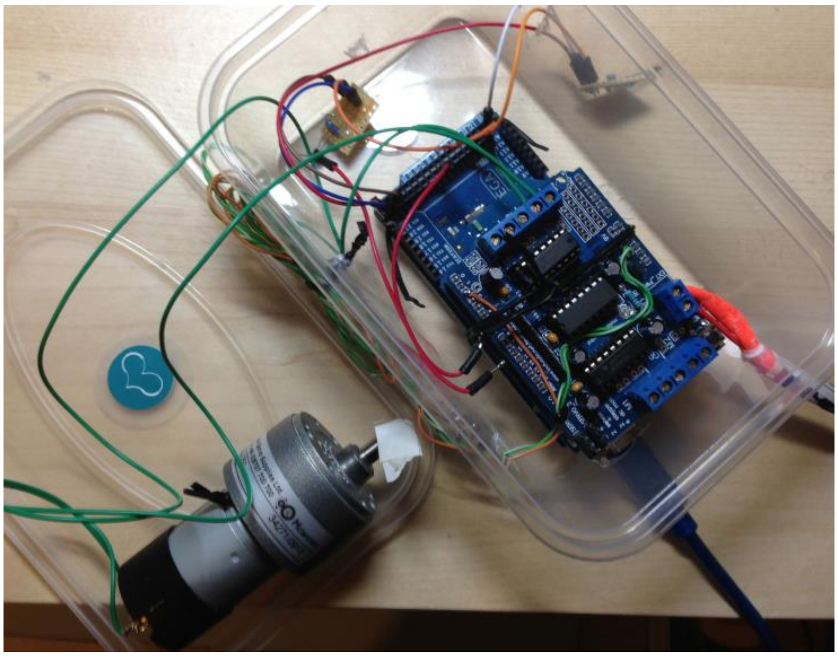


Figure 5.5 - Blinds Module



5.6.8 Basic Arduino Sensors (Button & PIR)

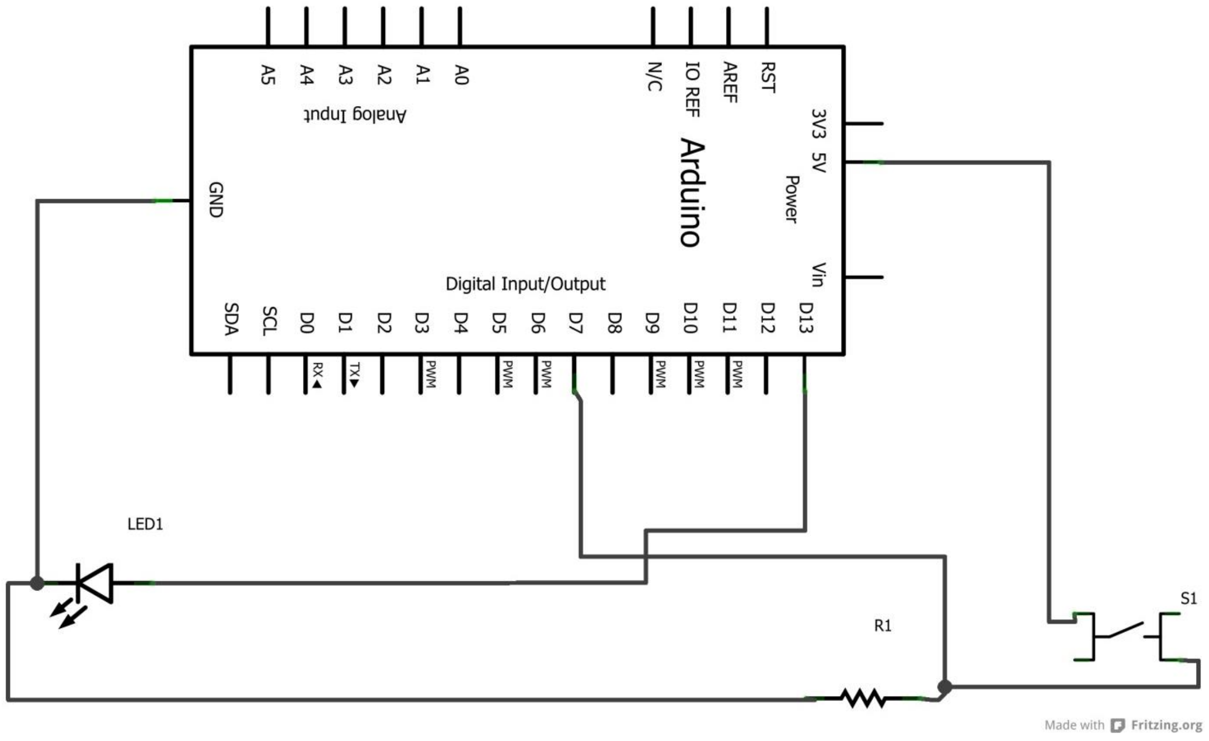
For the Arduino modules that were not as complex as the door opener, we used the following hardware:

- Arduino Uno board
- Arduino Wi-Fi shield
- Small resistors
- Button switch
- PIR sensor

After testing with breadboards, we moved onto making them permanent by soldering them. Both of the sensors use the WiFi library mentioned above to act as a server for returning the current state and make use of widely available example code[9] for sensors such as these.

Button

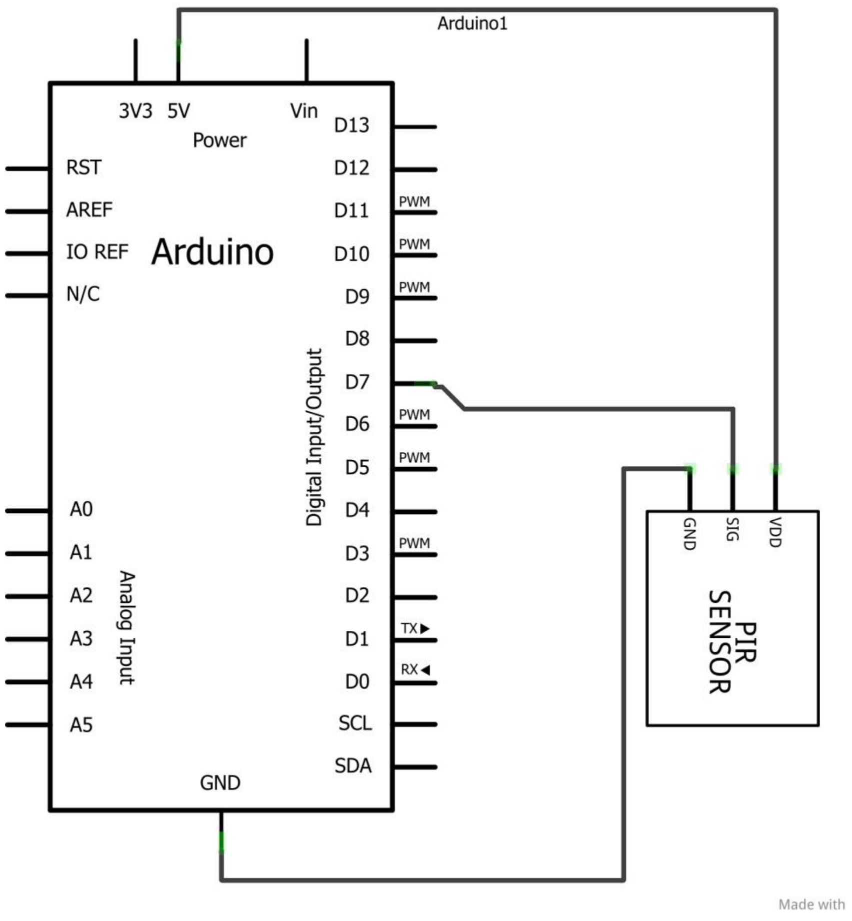
Figure 5.6 - Button Module



[9] http://playground.arduino.cc/Code/PIRsense , http://arduino.cc/en/tutorial/button

PIR Sensor

Figure 5.7 - PIR Module



6 Testing

6.1 Testing Strategy

Initially, we decided to make use of the test driven development. This meant that we wanted to write the tests, make code that passes them and eventually refactor the code for a better quality and optimisation. This also went along well with the agile practices that we tried to incorporate in our development process.

As already mentioned, the main component of our system is the server running on the Raspberry Pi. This is why we decided that this part of the system has to be tested thoroughly and with great care. If the server breaks, the whole system would go down. We decided to use various methods, such as unit testing and integration testing to make sure that everything works.

The user interface is also tested in a similar way to the server, since this is also a very important feature of our system. Many code and UI elements are generated dynamically, which means that there are many variables to take into account.

The other component that is more difficult to test is the hardware, since most of the time it has to be physically checked if the hardware operates. For example, currently the system cannot tell when a lamp has died out. This is very hard to test and we decided it is unnecessary for our system.

We also decided to trust code that we got from external sources. Therefore, we do not test any of the libraries that we use or any other code that we sourced as not our own work. The motivation behind it is that most likely this code works and we usually use it for things like communicating with the database or the hardware, which in itself is hard to test.

Each feature that we or some other developer adds or changes on the system has to be checked and tested by us before it is released for the people to upload. This means that although anyone can change the code freely, we still can control the updating of the main system and maintain it functional at all times.

6.2 Types of Testing Used

6.2.1 Unit Testing

The system incorporates standard unit tests using mock data on all the components of the system, apart from the Hardware, which is very hard to test. We chose them because they are simple and quick to write and are good to use with Agile development.

6.2.2 Integration Testing

Integration testing is done between JavaScript and the server, which is where most of the API calls are made. The integration of other components is not tested automatically.

As it stands, the tests for the backend run through all of its output data and format. The frontend has tests for both its input and output in separate test suites and its output tests are done the same way as the Python ones are. The application is AJAX-driven, so the tests run through all of the methods in `APP.ajax` and check their output for a given set of arguments. If there is an untested method, then it shows up as an error as well.

For the input tests, we test the JSON data returned by a GET request to the URLs `/structure`, `/state`, and `/eca`. Since the data is dynamic, we loop through all of it and test every unit to make sure they comply with the specified API format and the expected data types – the fact that the number of tests do not remain the same if we change the input data was the reason for separating the input and output tests.

6.2.3 UI Testing

The UI is tested on all major browsers and mobile devices for compatibility.

6.2.4 Manual Testing

Our system involves a lot of communication with hardware, which is sometimes impossible to check automatically. For example, it is hard to test that a light bulb turns on when a button is pressed. Therefore, in our user manual we ask the users to check if each button in the UI does what it is supposed to, by manually switching items through the UI.

Also from time to time, it is advised to repeat this test, in case the IP of the item has changed (even though it should not, if the setup is done properly) or the item has broken. For example a light bulb died.

6.2.5 Lint Checks

One of the smaller tests we perform is lint checking. Whilst it doesn't directly cause errors it is very important to have consistent code since the project is open source and available for anyone to view. It also helps us to try and stay consistent with each other's coding styles. A checker called Pylint is used for this and the results can be viewed in Jenkins.

6.3 Testing tools used

6.3.1 Unittest

Unittest is the default Python library. This provides functionality for producing unit tests for python code and is very easy to use, since it is based on JUnit, which means that it's similar to Java unit testing.

Each test has to be inside a test class that extends `unittest.TestCase`. It provides the usual assertions, teardown and setup methods, whereas nose helps in finding the tests to be run, and runs them all automatically at once.

6.3.2 Nose

Nose is a Python plugin that helps in finding and running tests. It can be used in addition to the default and pre-built `unittest` library in Python. However, Nose helps in finding the tests to be run, and runs them all automatically at once. It also provides more support for testing exceptions and timed tests, which the `unittest` library's support is limited.

According to the specification, "nose collects tests automatically from python source files, directories and packages found in its working directory."[10] It checks if a function or class is a test by checking the names that contain "test". It also checks for any classes that are subclasses of `unittest.TestCase`.

[10] https://nose.readthedocs.org/en/latest/usage.html

6.3.3 QUnit

QUnit is used to write and run unit tests in JavaScript. Like all other unit test frameworks, it provides all the usual functions such as assertions.

To run unit tests using QUnit, all the tests should be saved into one file, here for example tests.js. The JavaScript to test should be in a different file, here master.js.

6.3.4 PHPUnit

PHPUnit is used for unit testing PHP. It supports all the usual test methods, such as assertions, setup and teardown. Each test class should extend the `PHPUnit_Framework_TestCase`. The tests are public methods expecting no parameters and prefixed with "test". Each test method should include one or more assertion methods to assert that an actual value matched the expected value (or that the value is wrong as expected). There are, also, many additional functions, such as marking tests as unimplemented or skipping tests.

6.4 Automation

Our system uses a continuous integration server called Jenkins to automate the testing process. When there is a commit on Git, a request is sent that triggers the testing. The current testing status can be viewed any time at http://robohome.cloudapp.net/job/Robohome/ .

The script below is used to automate the process:

```
rm -f ./master.zip
rm -rf ./comp2014-master/
wget https://github.com/michboon/comp2014/archive/master.zip
unzip master.zip
cd ./comp2014-master/web/RoboHome/

nosetests --with-xcoverage --with-xunit --cover-package=robohome --
cover-erase
pylint --disable=C0301 --disable=C0103 --disable=W0614 -f parseable
robohome/ | tee pylint.out

screen -p 0 -X -S flask kill

cd ../../Scripts
echo "drop database robohome;" | mysql -u root
mysql -u root < robohome.sql
cd ../web/RoboHome/robohome

screen -dmS flask
screen -S flask -p 0 -X stuff "python flaskServer.py > flaskOut.txt
"
```

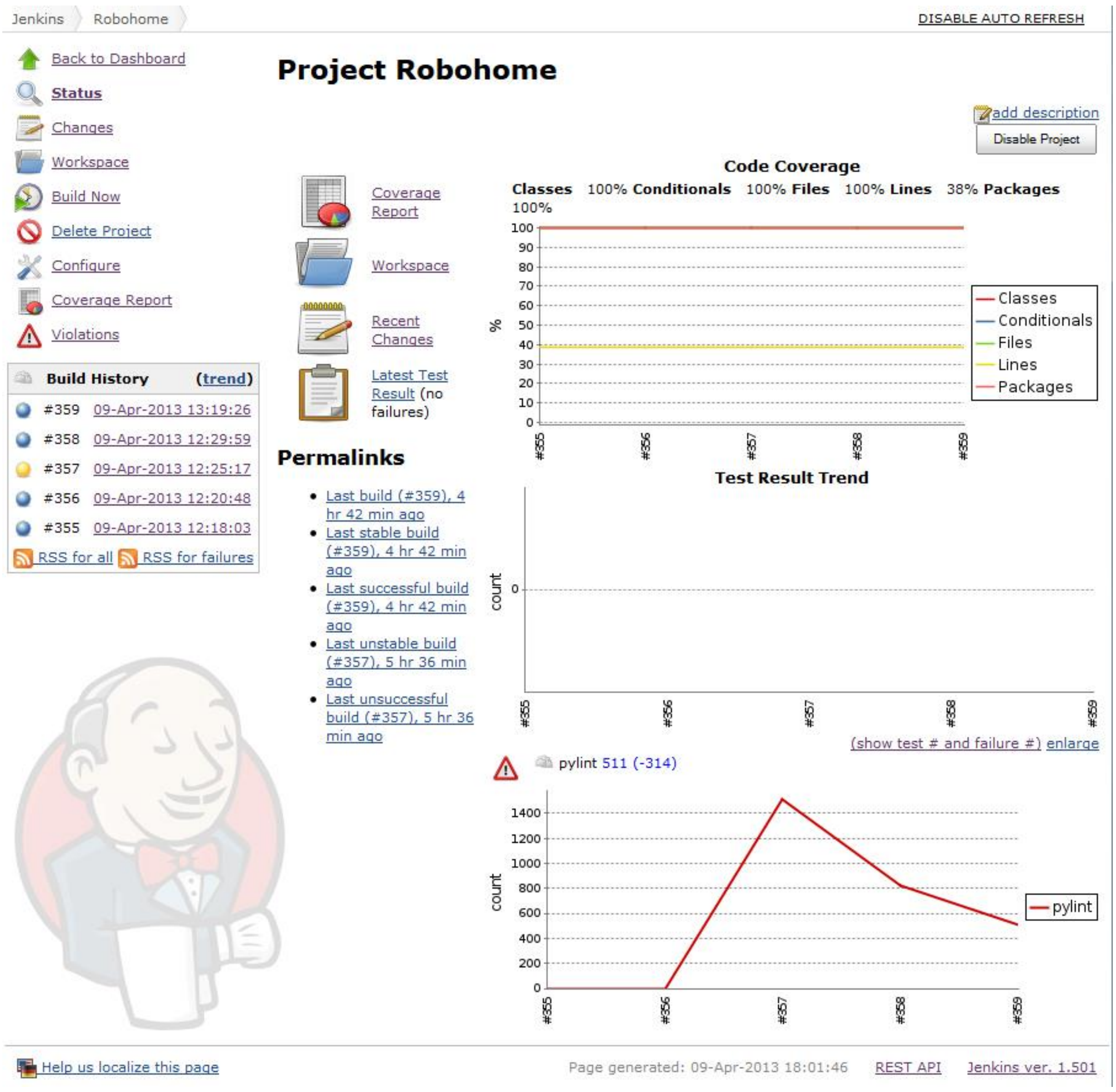
It firstly deletes any previous files in the Jenkins workspace and downloads the newest version from GitHub. It then runs the Python unit tests along with code coverage and lint checks. Once all the tests have passed, the database is filled with some mock data and the system is run. Therefore, developers are able to do any extra checks easily and run the JavaScript tests.

6.5 Code Coverage

As part of our Jenkins tests, code coverage reports are generated and graphed for easy viewing of our progression. We check for coverage of classes, conditionals and lines. For our project it is particularly hard to achieve high code coverage for some sections of code since they are very dependent on the hardware (namely the middle layer sections). There are also various sections of code copied from the internet, which we do not test (i.e. any code with a source listed in its documentation).

With that said, we believe that we have thoroughly covered all other functional areas of the code and all methods that contain logic are tested. Our latest reports tell us that classes and files are at 100% coverage and lines are at 38% (approximately 70% without hardware) coverage.

Figure 6.1 - Example Jenkins Output


Jenkins › Robohome

DISABLE AUTO REFRESH

Back to Dashboard
Status
Changes
Workspace
Build Now
Delete Project
Configure
Coverage Report
Violations

Build History (trend)

#359 09-Apr-2013 13:19:26
#358 09-Apr-2013 12:29:59
#357 09-Apr-2013 12:25:17
#356 09-Apr-2013 12:20:48
#355 09-Apr-2013 12:18:03

RSS for all RSS for failures

Project Robohome

add description
Disable Project

Coverage Report
Workspace
Recent Changes
Latest Test Result (no failures)

Permalinks

- Last build (#359), 4 hr 42 min ago
- Last stable build (#359), 4 hr 42 min ago
- Last successful build (#359), 4 hr 42 min ago
- Last unstable build (#357), 5 hr 36 min ago
- Last unsuccessful build (#357), 5 hr 36 min ago

Code Coverage

Classes 100% Conditionals 100% Files 100% Lines 38% Packages 100%

Test Result Trend

(show test # and failure #) enlarge

pylint 511 (-314)

Help us localize this page

Page generated: 09-Apr-2013 18:01:46 REST API Jenkins ver. 1.501

7 Evaluation and Conclusion

7.1 Summary of Deliverables

Our deliverables are as follows:

- Raspberry Pi OS image[11]
  - Stripped down Arch Linux installation from our proof of concept
  - Server code running as services
- Server code to manage the house
  - Manages items and rooms and reacts to events
  - Manages user logins with OpenID for security between local and remote access
  - Automatically updates when new versions are released and deploys the updates seamlessly
  - Support for plugins
- Web interface
  - Ability to manage items, rooms and rules through the API
  - Handles disconnections and other stability issues
  - Graphing of data
  - Fully utilises all features of the API
- Cloud service
  - Manages multiple Raspberry Pi's and multiple users
  - Uses OpenID for logging in for easy compatibility with the web interface
- Various hardware prototypes
  - Arduino door and blind openers, buttons and PIR sensors
  - Gadgeteer buttons, light sensors and PIR sensors
  - Belkin WeMo compatibility
  - LightwaveRF compatibility
- API, user manual and developer documentation
- Mobile apps
  - Windows Phone app
  - Android app with web view and voice recognition

7.2 Evaluation

We believe that our system meets our initial requirements and is fit for purpose. The purpose of our system, as described in the introduction, is to allow a user to manage a variety of hardware around the house. The system allows a user to add and remove items, control the state of the house, react to events and access the system securely through a cloud service, which are all important aspects of home automation. On the other hand, a large selection of supported hardware is not offered which is obviously also a very important aspect. This issue would always be the case for us because of the time constraints of the project; however, we have tackled it by instead providing a way for new supported hardware to be added very easily.

[11] Due to GitHub binary file restrictions, we were unable to host the image. Instead, setup scripts are provided which will turn a fresh Arch Linux install into the correct state. Given ample hosting, we would then host an image of this.

The system was shown to various potential users and clients over the course of the project. In general, they were impressed with the concept and hardware flexibility of the system. Unfortunately, we were told that the UI seemed slightly cluttered by a Microsoft Representative and that we should try to limit the number of options available to a user at once. However, we made some small UI adjustments and he approached us again after the final project presentations to commend us on our general improvement.

There was a strong focus on scalability from the start of our redesign after our proof of concept. It was very limited in terms of adding new sensors to the system let alone allowing users to manage multiple systems at once. With our new design we addressed these issues as a priority by managing items and rooms in the database and providing API methods to manage these as well as adding the cloud service to allow users to manage multiple versions of our system.

Next, we wanted to ensure that we designed the system in a modular way so that it would be possible to swap new components in and out, as we need to. We achieved this by making use of interfaces as much as possible and completely separating out the components of the system.

Finally, we wanted to design the system in a way that was flexible and stable. Since it is an embedded system for a user's home, it is very important that it handles unusual use cases appropriately and acts exactly as expected. To address this we made sure we handled various situations such as network, power and hardware failure. For example, the web interface relies on polling the house state from the server and if the server stops replying to requests, the interface shows an error to the user and displays the most recently known state of the house.

Following on from stability, we wanted to make our system efficient. This is important since a user does not want to wait around for an action such as a door opening for them. We handled efficiency with the priority queue system as discussed earlier. Apart from this, efficiency was less of a concern for us since each home has its own Raspberry Pi, which is more than capable enough of managing even the largest of houses. For the cloud service, the capacity would be limited by the server hardware but it is more than capable enough for its current use and can be easily upgraded in the future.

We believe the security of our system is adequate. Whilst we do take measures to ensure that a user's hardware is secure there are a number of ways to abuse the system. For example, it is possible to be the first user to access the system during setup and gain access. It is also possible to change the IP's on the cloud service and access the system, perhaps without permission, if you are on the local network. These small problems can be dealt with easily in further development.

Our high-level design has high cohesion and low coupling. Each component does a specific task; the Raspberry Pi manages the hardware and provides an interface to do this through, the cloud service redirects users and the middle layers interact with the hardware. In addition to this, each component interacts through interfaces, which achieves low coupling. The lower-level design of certain sections has slightly lower cohesion than could be hoped for in some places but still low coupling. This is because the Flask server needed an object to forward method calls onto and manage the house. Our answer to this was the House class, which turned out to handle many different tasks and has high cohesion.

We spent a considerable amount of time discussing our RESTful API and we believe that this has resulted in something that is easy to use for someone wishing to develop our system further. We have been sure to document the API thoroughly.

Finally, we think that we made the correct choice of programming languages and data formats. Python was an appropriate choice of language since it allowed us to make quick changes throughout development and load in new sections of code during runtime for updates and plugins. Additionally, it runs well on the Raspberry Pi. JSON was the natural choice for communicating between components since it represents states very well. We found jQuery to be sufficient for our purposes, but its nature as a general library meant that a lot more code had to be written, such as our own event handlers. Making use of an event or UI library such as node.js or Google's closure library might have saved use some time, although it would mean learning to use a new library or two.

In general, we are pleased with the progress of the project and the cooperation between the team. Communication was an issue that we had to overcome; sometimes we would forget to manage changes to the code properly. When problems like this occurred we did our best to fix them as soon as possible and by the end of the project there was definite improvement. Testing also helped us to overcome this and notice when recent changes had broken the code. We agreed as a team on the majority of the design however, on a low-level there are inevitable differences in style in some places. Throughout this project, we have learned the importance of both good communication and cooperation. In addition to this, we have seen the importance of deadlines and staying true to design decisions.

8 Appendix A – User Manual and Deployment Instructions

8.1 System Setup

8.1.1 Raspberry Pi OS

1. The latest stable version of the software is available from http://robohome.co.uk. Alternatively, the current development release can be found at https://github.com/michboon/comp2014
2. Download the ISO and use a tool such as Win32DiskImager (http://sourceforge.net/projects/win32diskimager/) to write the image to a SD card for the Raspberry Pi[12]
3. Connect the Raspberry Pi to your home router and a power source

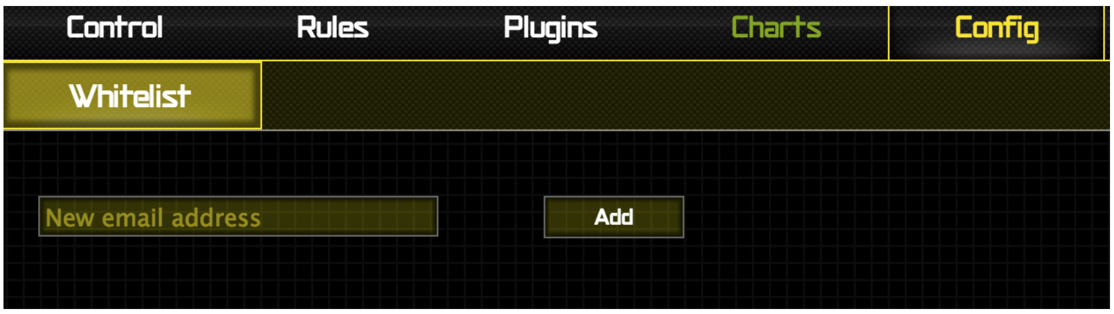
8.1.2 Router Configuration

For usage of the cloud service and remote access, you must forward ports on your router to port 9090 on the Raspberry Pi. Instructions specific to your brand of router are available at http://portforward.com/english/routers/port_forwarding/. For easy use, it is suggested that you forward incoming requests on port 80 to port 9090 on the Raspberry Pi.

It is recommended that each new module you add to the system be given a static IP on your local network. Please see the documentation for your router for instructions on how to do this. Afterwards, make a note of the IP of your Raspberry Pi.

8.1.3 Connecting to the system for the first time

1. Open a web browser on your local network and enter the address http://<RaspberryPiIP>:9090 and you will be taken to the start page on the Raspberry Pi. You can use this address whenever you are connected to your local network to access the system without any need for logging in
2. For remote access, you should use your public IP address (http://www.whatismyip.com/) with the correct port as configured for your router earlier
3. **IMPORTANT:** If you have forwarded your ports, you should access the system through your public IP as part of the setup. You will be asked to login with you Google account and automatically registered as the first user for the system. Any further users will need to be added to the whitelist before being allowed to register
4. New emails for OpenID registration can be added under Config -> Whitelist:



[12] Due to GitHub binary file restrictions, we were unable to host the image. Instead, setup scripts are provided which will turn a fresh Arch Linux install into the correct state. Given ample hosting, we would then host an image of this.

8.2 Hardware Setup

8.2.1 Adding new items

Once you have correctly configured new hardware, you can add it through the user interface. Under the “Control” tab, go to the tab for the room you wish to add it to and fill in the appropriate fields on the left side of the screen. Be sure to select the correct brand of hardware as this is an important part of how the system will interact with it.

Below are instructions for setting up our currently supported hardware. If you are using newer hardware with an updated version of the software then please see the relevant documentation for new hardware support.

8.2.2 Belkin WeMo

1. Connect the WeMo to your home network as described in the WeMo instructions at http://www.belkin.com/us/support-article?rnId=7073
2. Give each WeMo an IP address with your router

8.2.3 LightwaveRF

1. Connect the LightwwaveRF hub to your router
2. Go to http://www.lightwaverf.com/my-lightwave and set up the hub accordingly
3. Add any new LightwaveRF hardware to the hub as specified in their documentation
4. When a new piece of hardware is added make note of the Room ID and Device ID you give it in the manager
5. To add this into the RoboHome system you should use the IP of the hub followed by the Room ID and Device ID (e.g. 192.168.0.1:R1D1 for room 1, device 1)

8.2.4 Arduino and Gadgeteer

Software for Arduino and Gadgeteer modules should only be used by users who are familiar with these boards and is available from:

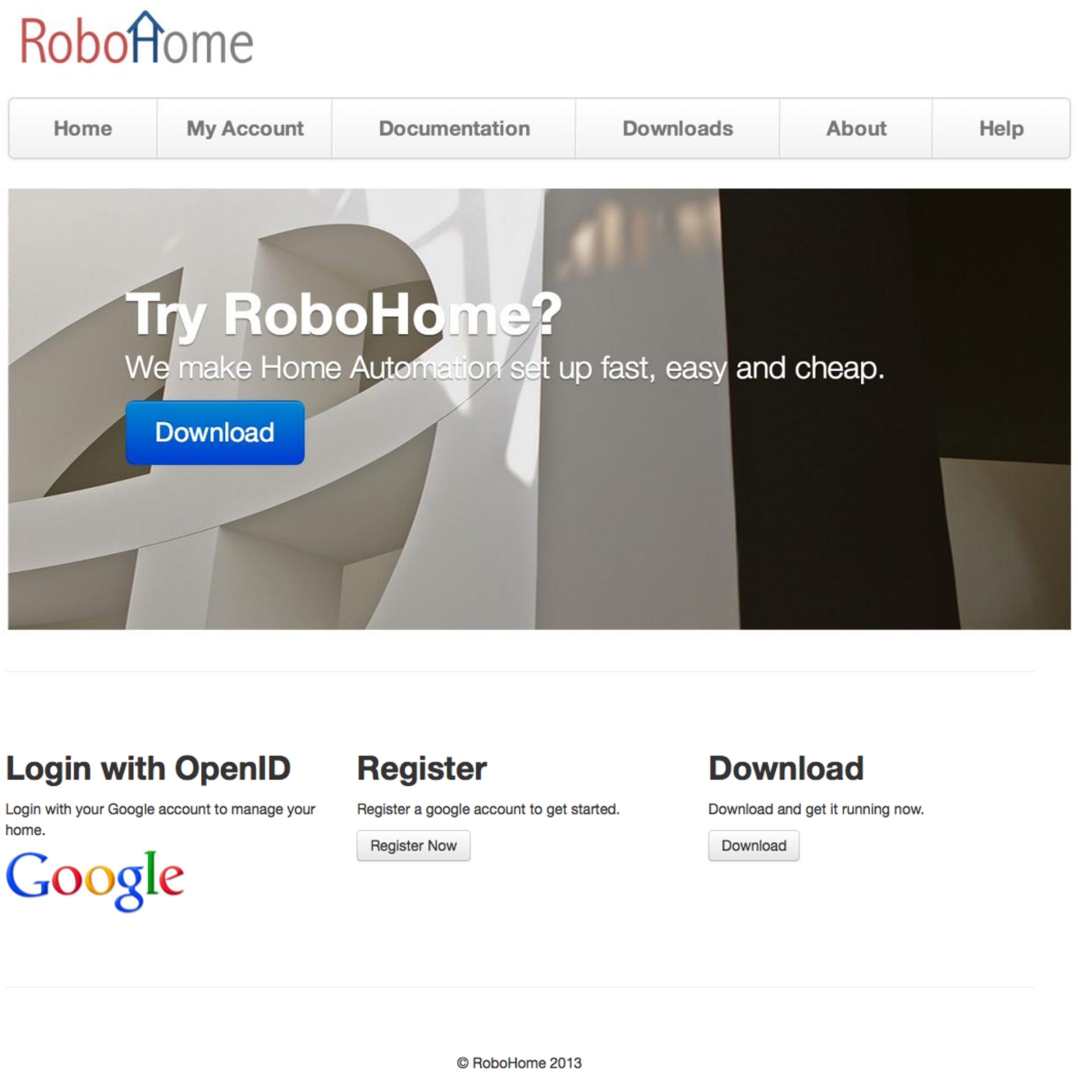
https://github.com/michboon/comp2014/tree/master/hardware

8.2.5 Testing

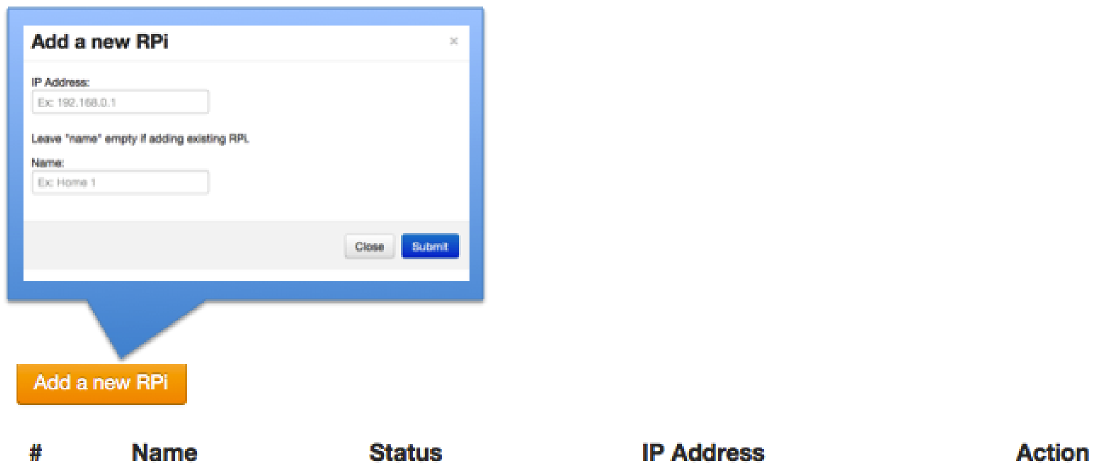
After all the items have been added, it is recommended that you check if everything is working. This can be simply done by clicking on each button and checking if the result is as expected. If it is not, please first check that the item is of the right type, brand and IP. If that is the case, then it is likely that there is a hardware fault. If after this there is still a problem, please report the issue to us.

8.3 Cloud Service

The RoboHome cloud service is accessible through http://robohome.co.uk


RoboHome
Home
My Account
Documentation
Downloads
About
Help
Try RoboHome?
We make Home Automation set up fast, easy and cheap.
Download
Login with OpenID
Login with your Google account to manage your home.
Google
Register
Register a google account to get started.
Register Now
Download
Download and get it running now.
Download
© RoboHome 2013

Once you have logged into an account, add a new Raspberry Pi by clicking on the orange button below. Then enter the IP address of the Raspberry Pi and give it a name.



Add a new RPi ×

IP Address:
Ex: 192.168.0.1

Leave "name" empty if adding existing RPi.

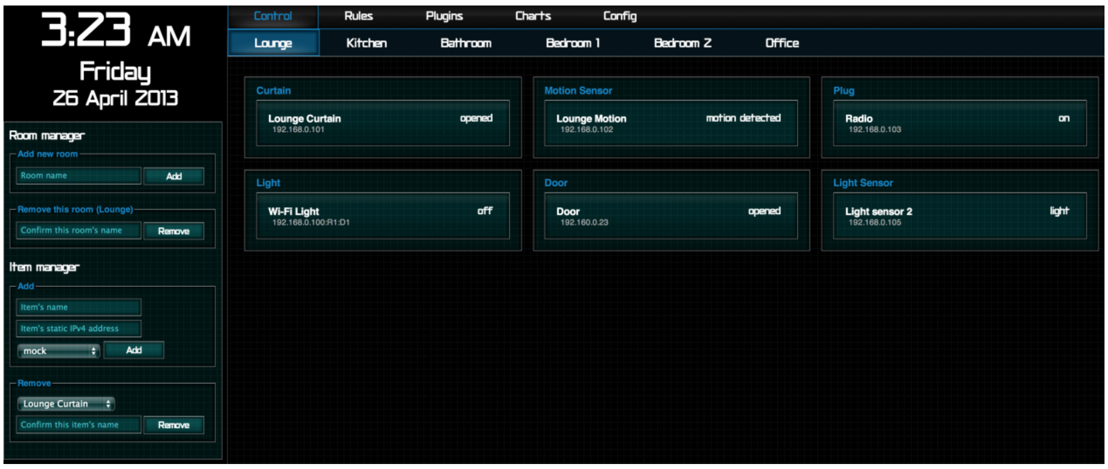
Name:
Ex: Home 1

Close Submit

Add a new RPi

#	Name	Status	IP Address	Action

8.4 The RoboHome Interface



3:23 AM
Friday
26 April 2013

Control | Rules | Plugins | Charts | Config

Lounge | Kitchen | Bathroom | Bedroom 1 | Bedroom 2 | Office

Room manager
Add new room: Room name — Add
Remove this room (Lounge): Confirm this room's name — Remove

Item manager
Add: Item's name; Item's static IPv4 address; mock — Add
Remove: Lounge Curtain; Confirm this item's name — Remove

Type	Item	Address	State
Curtain	Lounge Curtain	192.168.0.101	opened
Motion Sensor	Lounge Motion	192.168.0.102	motion detected
Plug	Radio	192.168.0.103	on
Light	Wi-Fi Light	192.168.0.100:R1:D1	off
Door	Door	192.160.0.23	opened
Light Sensor	Light sensor 2	192.168.0.105	light

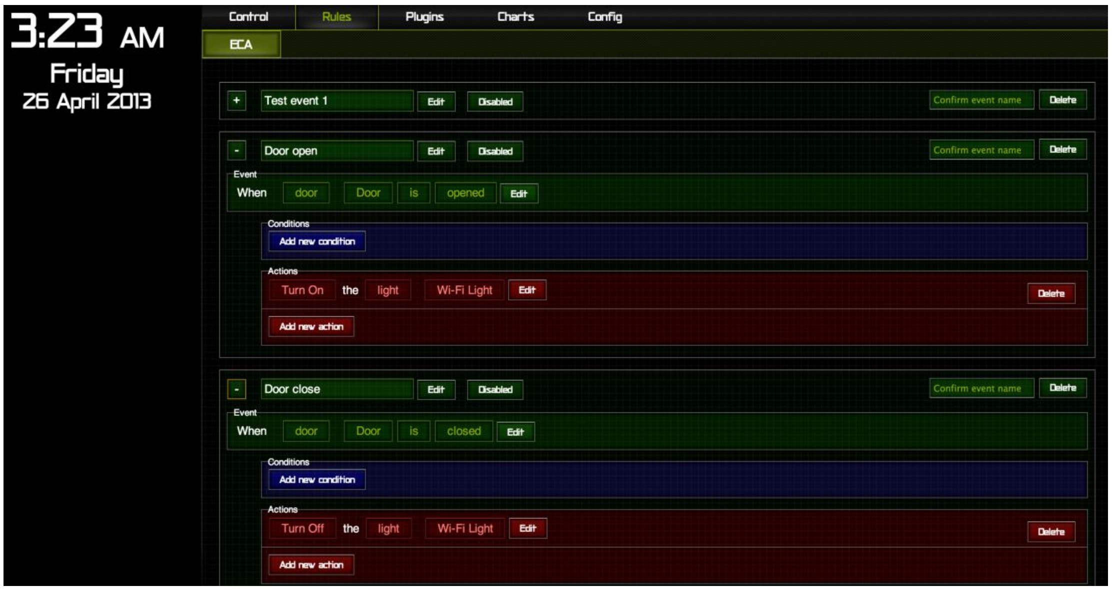
8.4.1 Adding and deleting rooms

1. Go to the "Control" tab
2. To add a new room, add a new name in the "Room Manager" on the left and click "Add"
3. To delete a room, go to the tab of that room and confirm its name to the left of the "Delete" button in the "Room Manager"

8.4.2 Rules

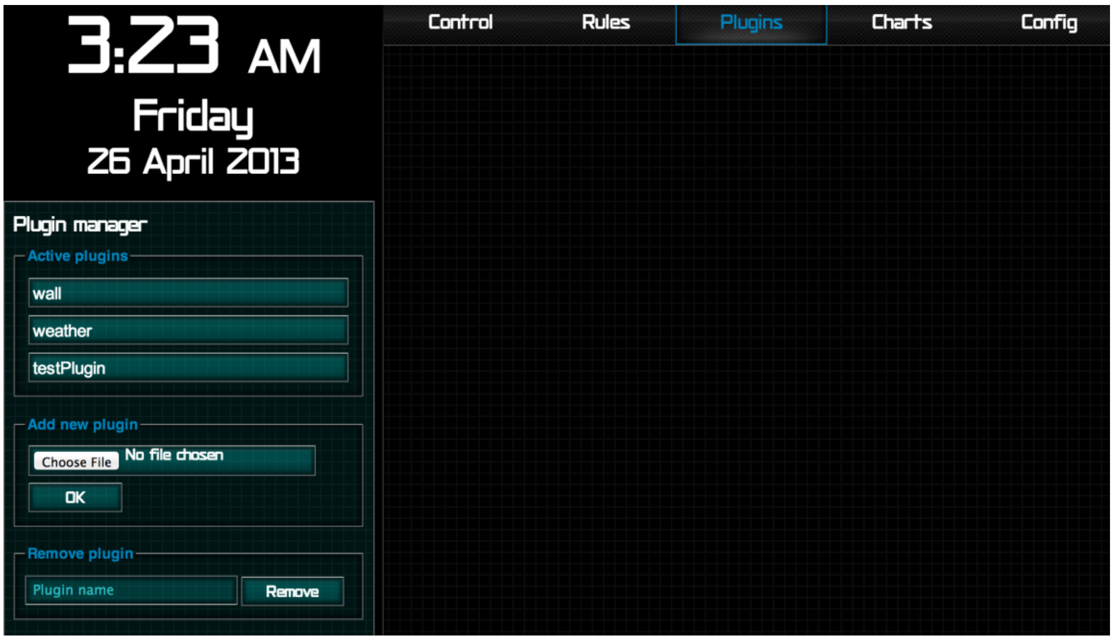
Rules can be added under the "Rules" tab. At the bottom of the page, you can create a new event by clicking the 'Add New Rule' button. After creating an event, add conditions and actions for further customisation.

If you wish to disable a rule, you can do so by clicking toggling the "Enabled" button. If you want to delete it permanently, then first type in the Event name and then click 'Delete' at the right hand side of the page.



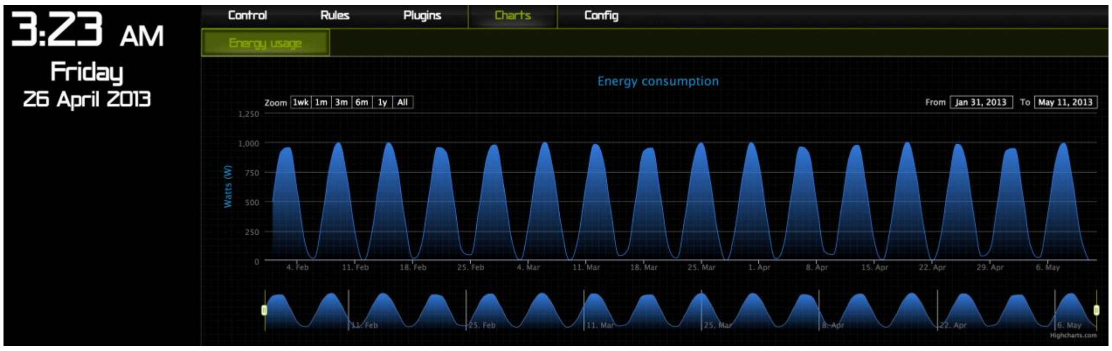
8.4.3 Plugins

Plugins can be uploaded and accessed through the "Plugins" tab. Please see the relevant documentation for installing and using a plugin. Some example plugins are included by default and can be deleted by confirming the plugin name and selecting "Delete".



8.4.4 Energy Usage

You can check your energy usage by looking in the "Charts" tab providing that you have an energy monitor installed and configured with the system.


3:23 AM
Friday
26 April 2013
Control
Rules
Plugins
Charts
Config
Energy usage
Energy consumption
Zoom 1wk 1m 3m 6m 1y All
From Jan 31, 2013 To May 11, 2013
Watts (W)
1,250
1,000
750
500
250
0
4. Feb
11. Feb
18. Feb
25. Feb
4. Mar
11. Mar
18. Mar
25. Mar
1. Apr
8. Apr
15. Apr
22. Apr
29. Apr
6. May
Highcharts.com

9 Appendix B – Developer Documentation

9.1 API

9.2 RESTful API Documentation

9.2.1 URL

```
http://0.0.0.0/version/(string:versionNo)/rooms/(int:roomId)/items
/(int:itemId)/(string:action)?(params)
```

Example

```
http://0.0.0.0/version/0.1/rooms/1/doors/1/open
```

9.2.2 Wrapper string

```
{statusCode: <statusCode>, content: {} }
```

9.2.3 /version

GET

Gets list of methods and equivalences associated with each device type

Response

```
content: {
   supportedTypes: {
        door: {
             name: 'Doors',
             isPassive: '<boolean true/false>',
             supportedBrands: [
                  'brandName-0',
                     'brandName-1'
                  ..
             ],
             methods: ['open', 'close'],
             states: [
                  { state: 0, name: 'open', method: 'open' },
                  { state: 1, name: 'closed', method: 'close' }
                  ]
             },
             window: {
                  ..
                  },
             lights: {
                  ..
                  },
                  ..
        }
   }
```

9.2.4 /state

GET

Get house structure

Response

```
content : {
   rooms: [
        {
          id: <roomId1 (int)>,
          name: <roomName (string)>,
          items: [
               {
                    id: <itemId (int)>,
                    name: <itemName (string)>,
                    itemType: <itemType (string)>,
                    brand: <itemBrand (string)>,
                    ip: <itemIP (string)>,
                    state: <state (int)>
               },
          ]
        },
        {
          ..
        },
   ]
}
```

9.2.5 /rooms

POST

Create new room with JSON content

Request

```
name = <roomName (string)>
```

Response

```
content: {
     id: <newRoomId (string)>
}
```

9.2.6 /rooms/(int: roomId)

PUT

Updates a specified room

Request

```
name = <newRoomName: string>
```

Response

```
content: 'success'
```

DELETE

Delete specified room

Response

```
content: 'success'
```

9.2.7 /rooms/(int: roomId)/items/

POST

Request

```
brand =  <brand (string)>,
ip = <ip (string)>,
name = <name (string)>,
itemType = <itemType (string)>
```

Response

```
content: {
	itemId: <itemId (int)> // new id allocated for new item
}
```

9.2.8 /rooms/(int: roomId)/items/(int: itemId)/events

GET

Gets a list of current rules

Response

```
content: {
  rules: [
       { // single rule
           ruleId: <ruleId (int)>, // rule id
           ruleName: <ruleName (string)>,
           enabled: <true/false (bool)>
           event: {
                // event
                id: <itemId / roomId (int) OR null if scope ==
house>,
                itemType: <itemType / time passes (string)>,
                scope: <scope (string) -- 'item', 'room', or
'house'>,
                equivalence: <gt/lt/eq (string)>,
                value: <number/bool/time (long) -- triggering
state>,
           },
           conditions: [ // per sensor basis
             {
                conditionId: <conditionId (int)>,
                // device
                itemId: <itemId (int)>,
                itemType: <itemType (string)>
                method: <function (noargs)>,
                equivalence: <gt/lt/eq (string)>,
                value: <number/bool/time (long)>
             },
             {
                ..
             }
           ],
           actions: [
             {
                actionId: <actionId (int)>,
                id: <itemId / roomId (int) OR null if scope ==
house>,
                id: <itemId / roomId (int) OR null if scope ==
house>,
                scope: <scope (string) -- 'room', 'item', or
'house'>
                method: <function (noargs, void)>
             },
             {
                ..
             }
           ]
      },
      { // 2nd rule
           ..
      }
  ]
}
```

POST

Create a new event

Request

```
ruleName = <ruleName (string)>
enabled = <true/false (boolean)>
id = <itemId / roomId OR null if scope == house>
itemType = <itemType (string)>
scope = <scope (string) -- 'item', 'room', or 'house'>
equivalence = <equivalence>
value = <value (int)>
```

Response

```
content: {
      ruleId: <ruleId (int)>
}
```

9.2.9 /events/(int: eventId)

PUT

Updates the event

Request

```
ruleName = <ruleName (string)>
enabled = <true/false (boolean)>
id = <itemId / roomId OR null if scope == house>
itemType = <itemType (string)>
scope = <scope (string) -- 'item', 'room', or 'house'>
equivalence = <equivalence>
value = <value (int)>
```

Response

```
content: 'success'
```

DELETE

Delete the event

Response

```
content: 'success'
```

9.2.10 /events/(int: eventId)/conditions/

POST

Add new condition to an event

Request

```
itemId =  <itemId (int)>,
equivalence = <gt/lt/eq (string)>,
value = <number/bool/time (long)>
```

Response

```
content: {
      conditionId: <newConditionId (int)>
}
```

9.2.11 /events/(int: eventId)/conditions/(int: conditionId)

PUT

Request

```
itemId = <itemId (int)>,
equivalence = <gt/lt/eq (string)>,
value = <number/bool/time (long)>
```

Response

```
content: 'success'
```

DELETE

Response

```
content: 'success'
```

9.2.12 /events/(int: eventId)/actions/

POST

Request

```
id = <itemId / roomId (int) OR null if scope == house>,
itemType = <itemType (string)>,
scope = <scope (string) -- 'room', 'item', or 'house'>
method = <function (noargs, void)>
```

Response

```
content: {
      actionId = <actionId (int)>
}
```

9.2.13 /events/(int: eventId)/actions/(int: actionId)

PUT

Request

```
id = <itemId / roomId (int) OR null if scope == house>,
itemType = <itemType (string)>,
scope = <scope (string) -- 'room', 'item', or 'house'>
method = <function (noargs, void)>
```

Response

```
content: 'success'
```

DELETE

Response

```
content: 'success'
```

9.2.14 /plugins

POST

Request

```
file = <file (file)>
```

GET

Response

```
content {
      plugins: [
            "plugin1",
            "plugin2",
            "plugin3",
            ..
      ]
}
```

9.2.15 /plugins/(string: pluginName)

DELETE

Response

```
content: 'success'
```

9.2.16 /whitelist

GET

Response

```
content {
      emails: [
            'email1@foo.com',
            'email2@bar.com',
            '...',
      ]
}
```

POST

Request

```
email = <newEmail@foo.com (string)>
```

Response

```
content: 'success'
```

DELETE

Request

```
email = <newEmail@foo.com (string)>
```

Response

```
content: 'success'
```

9.3 Plugin Creation

All plugins must implement the plugin interface, which contains the following methods.

setup(self, rooms, events, queue)

This method is called once upon loading or reloading plugins. It provides the plugin with access to the list of rooms and the items they contain (along with the user-defined rules) and the queue to add item commands to.

teardown(self)

This method is called once upon deleting or reloading a plugin.

getPage(self, path)

This method is called when a user opens an iframe to view the plugin in. The path argument is the remaining path in the URL after the plugins/pluginName. By default, this will be "None" in Python since the initial page loaded will be http://localhost:9090/plugins/pluginName but this allows developers to use links to any deeper paths needed so long as they handle the path.

notify(self, ip, trigger)

This method is called whenever the state of the house changes. Please see the ECA documentation for a more thorough description of the trigger parameters that will be passed through.

getName(self)

This method should return the name of the plugin, which should match the URL the user uses to access it. **NOTE: Until further development, plugin .zip folders should be named after this too.**

9.3.2 Example Plugin

This example plugin simply loads BBC weather whenever the user loads it. Note the change of file path in the imports so that the plugin is able to import the Plugin class and subclass it.

```
import os
parentdir =
os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(__fi
le__))))
os.sys.path.insert(0, parentdir)
from pluginManager import Plugin


class Weather(Plugin):
    def setup(self, rooms, events, queue):
        pass

    def getName(self):
        return "weather"

    def getPage(self, path):
        with open('./plugins/weather/page.html', 'r') as
content_file:
            return content_file.read()

    def teardown(self):
        pass

    def notify(self, ip, trigger):
        pass
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <link type="text/css" rel="stylesheet" href="../plugin.css"
/>
        </head>
    <body>
        <iframe style="width: 100%; height: 100%; border: none;"
src="http://www.bbc.co.uk/weather/"></iframe>
        </body>
</html>
```

9.3.3 Further Notes

- Any threads created in a plugin should be a daemon
- Plugins should add actions to the queue rather than directly on an item
- It is advised that any html returned by a plugin uses the CSS found at ./plugins/plugin.css