



Università degli Studi di Camerino

SCUOLA DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica (Classe L-31)

Analisi del framework openHAB e sviluppo di un binding per un sistema IOT

Laureando
Geremia Pompei

Matricola 105333

Relatore
Michele Loreti

A handwritten signature in black ink, appearing to read "Michele Loreti".

Indice

1	Introduzione	11
1.1	Motivazione	11
1.2	Obiettivi	11
1.3	Struttura della Tesi	11
2	Struttura di openHAB	13
2.1	Concetti Principali	13
2.1.1	Binding	13
2.1.2	Things	15
2.1.3	Channels	15
2.1.4	Items	17
2.1.5	Rules	19
2.1.6	Sitemap	23
2.2	Modello Semantico	24
2.3	Persistence	26
2.4	openHAB REST API	28
3	MQTT Binding	31
3.1	Protocollo MQTT	31
3.2	openHAB Binding	33
3.2.1	Configurazione bridge	34
3.2.2	Things supportate	35
3.2.3	Configurazione channel	35
3.2.4	Rule actions	37
4	Configurazione dell'MQTT Binding per un sistema IOT	39
4.1	Smart Garden	39
4.1.1	Sviluppo sistema IOT	41
4.1.2	Broker MQTT	47
4.1.3	MQTT Binding del sistema IOT	47
4.2	Configurazione openHAB	48
4.2.1	MQTT Binding di openHAB	48
4.2.2	Creazione Sitemap	57
5	Conclusioni	59

Elenco dei codici

2.1	Import	19
2.2	Variable Declarations	19
2.3	Rules	19
2.4	Event-based Triggers	20
2.5	Member of Triggers	20
2.6	Time-based Triggers	21
2.7	System-based Triggers	21
2.8	Thing-based Triggers	21
2.9	Channel-based Triggers	21
2.10	Esempio Script	22
2.11	Esempio State Type	22
2.12	Esempio Codice di una Sitemap	23
3.1	Esempio Rule Action Istanza	37
3.2	Esempio Rule Action Pubblicazione	37
4.1	Soil.py	44
4.2	Pump.py	44
4.3	MQTTClient.py	45
4.4	main.py	45
4.5	Sitemap Smart Garden Code	57

Elenco delle figure

2.1	openHAB Logo	14
2.2	Transizioni di Stato	15
2.3	Esempio Group Item	18
2.4	Esempio di View di una Sitemap	24
2.5	Relazioni Ontologiche	25
2.6	Esempio di Modello	27
2.7	API Token Login	29
3.1	MQTT Logo	32
3.2	MQTT Model Structure	32
3.3	MQTT Connection	33
4.1	Soil Moisture Sensore	40
4.2	Water Pump	40
4.3	Raspberry Pi	41
4.4	Relé	42
4.5	Alimentatore da 12 Volt	42
4.6	Struttura Smart Garden	43
4.7	Crea Utente Amministratore	49
4.8	Impostazione Lingua Regione e Fuso orario	49
4.9	Impostazione Posizione	50
4.10	Selezionare Add-ons	51
4.11	Schermata Impostazioni	51
4.12	Schermata Things	52
4.13	Schermata Scelta Binding	52
4.14	Schermata aggiunta Thing	52
4.15	Configurazione MQTT Broker	53
4.16	Configurazione MQTT Thing	54
4.17	MQTT Thing	54
4.18	Schermata Channels	55
4.19	Schermata Add Channel	56
4.20	Schermata Link Channel to Item	56
4.21	Items Configurati	56
4.22	Sitemap Smart Garden UI Web	57

4.23 Sitemap Smart Garden UI App	58
5.1 Smart Garden di fianco	60
5.2 Smart Garden pompa dell'acqua	61
5.3 Smart Garden sensore di umiditá e erogazione dell'acqua	62
5.4 Smart Garden dall'alto	63

Elenco delle tabelle

2.1	Concetti	14
2.2	Things states	16
2.3	Items	17
2.4	EnumTypes	18
2.5	Tipi di Elementi di una Sitemap	25
4.1	Smart Garden CHannels	55

1. Introduzione

OpenHAB sta per *Open Automation Bus* ed è una piattaforma di automazione domestica. Come dice il nome è *open source* ed il suo punto di forza è l'indipendenza dalla tecnologia; infatti i dispositivi intelligenti possono connettersi a prescindere dalla marca o dal tipo di device. Tale software installato in un dispositivo che funge da server permette di controllare l'intera casa intelligente. Inoltre *openHAB* fornisce un approccio comune alle regole di automazione dell'intero sistema così da rendere più immediato e semplice l'utilizzo. Punti di forza sono anche la flessibilità nell'integrazione di molti prodotti smart e l'alto grado personalizzazione.

1.1 Motivazione

Il motivo che porta allo studio e approfondimento di *openHAB* è soprattutto la sua caratteristica ad essere molto versatile alla configurazione con diversi dispositivi. Ciò permette quindi di abbinarlo in diversi contesti con elementi totalmente differenti tra loro. In questo caso verrà configurato per dialogare con un sistema IOT totalmente sviluppato da zero. Avendo come scopo principale quello di essere dinamico sarà un gioco da ragazzi controllare dispositivi sviluppati autonomamente in maniera semplice ed efficace tramite *openHAB*.

1.2 Obiettivi

L'obiettivo di questa tesi è studiare e approfondire openHAB per poi configurarlo e andarlo ad usare con un piccolo sistema IOT chiamato *Smart Garden* per il controllo del proprio giardino in maniera centralizzate insieme agli altri sistemi High Tech disponibili nella propria abitazione. Ciò permetterà in un futuro momento anche l'interazione di tale sistema IOT con altri configurando regole così da rendere la propria casa il più smart e comoda possibile.

1.3 Struttura della Tesi

La Tesi inizierà con l'analisi del framework entrando nei dettagli della propria logica interna definendo i vari elementi e l'interazione con essi. Si approfondiranno quindi aspetti legati alla modellazione base di *openHAB* e come questa può essere applicata alla domotica casalinga oltre alla comunicazione con il framework dall'esterno tramite un protocollo di rete.

Successivamente si approfondirà lo studio sul *protocollo MQTT* riguardo a come funziona e quali sono gli elementi al suo interno. Ciò sarà utile per la sezione successiva

relativa all'associazione di *openHAB* tramite tale protocollo. Qui verrá spiegato come viene vista l'associazione nel framework e come puó essere configurata in base alle tipologie di elementi messi giá a disposizione da esso.

Infine si parlerá del sistema IOT sviluppato e di come questo interagisce con *openHAB*. Si approfondirá l'idea di base e lo sviluppo dietro lo *Smart Garden* e la sua associazione con il protocollo MQTT. In seguito verrá approfondita la configurazione di *openHAB* per dialogare con il sistema IOT.

2. Struttura di openHAB

Per cominciare ad utilizzare *openHAB* bisogna eseguire dei passaggi iniziali

1. possedere un **computer** sul quale eseguire il software. Vi sono molti device disponibili per eseguire openHAB. Il più minimale e famoso è sicuramente **Raspberry Pi** perché integra perfettamente **openHABian**, ovvero una distribuzione *Linux* completamente auto-configurata per soddisfare le esigenze degli utenti openHAB. Essa contiene l'immagine completa della scheda SD già preconfigurata e lo strumento di configurazione openHABian. Un altro dispositivo molto diffuso sul quale eseguire *openHAB* è **DiskStation di Synology**, ovvero un server NAS. Tale approccio è meno economico ma migliore se già si possiede tale dispositivo. OpenHAB può essere eseguito anche su un normale **computer domestico** anche se è preferibile utilizzare un sistema dedicato.
2. installare *openHAB* e i vari **software** per farlo funzionare. *OpenHAB* è stato scritto in *java*, dunque dipende dalla *Java Virtual Machine*. La JVM è disponibile di default nei nuovi dispositivi, in caso contrario bisogna scaricarla. La versione attualmente supportata è la **JVM Java 11** e **Azul Zulu** è la piattaforma consigliata per *openHAB*. Il software openHAB è disponibile per diverse versioni **macOS** e **Windows** e molte distribuzioni **Linux** come *Ubuntu*, *Raspbian*, ... Dopo aver installato il software vi sono passi aggiuntivi da fare come configurare una rete condivisa per editare in maniera agevole i file di configurazione. Per tale scopo viene utilizzato *Linux Samba Share* lato server, mentre lato client *Visual Studio Code* con *openHAB VS Code Extension*.
3. far interagire i vari **dispositivi smart** con il server *openHAB*. Un dispositivo intelligente è un dispositivo elettronico, generalmente connesso ad altri dispositivi o reti tramite diversi protocolli. *OpenHAB* permette di collegarsi direttamente a loro e gestirli in maniera semplice e automatizzata.

2.1 Concetti Principali

I concetti principali sono rappresentati nella Tabella 2.1.

2.1.1 Binding

Sono pacchetti software installati dall'utente che hanno lo scopo di stabilire una connessione tra il dispositivo e la thing. Essi traducono tutti i comandi tra *openHAB* e la *thing*. I binding sono elencati nella sezione *Add-on* della documentazione ufficiale. Per ogni *binding* vengono fornite istruzioni dettagliate ed esempi che includono una guida



Figura 2.1: openHAB Logo

Concetto	Descrizione
Binding	sono i componenti openHAB che forniscono l'interfaccia per interagire elettronicamente con i dispositivi
Things	sono la prima rappresentazione generata da openHAB dei dispositivi
Channels	sono la connessione openHAB tra “Things” e “Items”
Items	sono la rappresentazione generata da openHAB delle informazioni sui dispositivi
Rules	vengono eseguite azioni automatiche (nella forma più semplice: se “questo” accade, openHAB farà “quello”)
Sitemap	interfaccia utente (sito web) generata da openHAB che presenta le informazioni e consente le interazioni

Tabella 2.1: Concetti

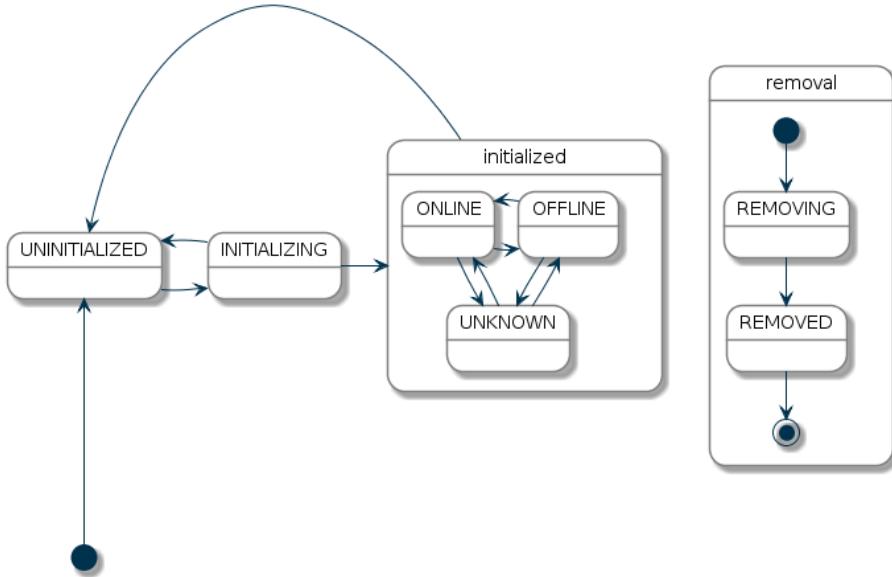


Figura 2.2: Transizioni di Stato

alla configurazione dell'associazione, l'elenco delle *things* supportate e i *channels* che le *things* forniscono.

2.1.2 Things

Sono le entità che possono essere fisicamente connesse al sistema e che potenzialmente forniscono più di una funzionalità. Esse non sono identificabili come device ma come il web service che questi che queste possiedono per essere controllate. Le *things* sono rilevanti per i processi di setup e configurazione ma non a runtime. Esse possono avere sia proprietà facoltative che obbligatorie (come indirizzo ip, access token, ...).

Un tipo particolare di *thing* è un **bridge**. Essi sono *thing* che devono essere aggiunte al sistema per aver accesso ad altre *thing*.

Poiché molte *things* possono essere scoperte automaticamente, sono disponibili meccanismi speciali che si occupano di tale gestione.

Ogni *thing* ha uno stato che aiuta a identificare possibili problemi con il dispositivo o il servizio. Possiamo vedere l'elenco degli stati alla Tabella 2.2.

Gli stati UNINITIALIZED, INITIALIZING, REMOVING e REMOVED sono impostati dal framework mentre UNKNOWN, ONLINE e OFFLINE sono assegnati dal binding. Sono rappresentate le transizioni tra i vari stati alla Figura 2.2.

2.1.3 Channels

Vengono forniti dalle *things* e rappresentano le diverse loro funzioni. Dove la *thing* è un'entità fisica o la fonte di informazione, il *channel* è una funzione concreta di questa. Una lampadina fisica(*thing*) potrebbe avere un canale della temperatura del colore(*channel 1*) e un canale del colore(*channel 2*), che forniscono entrambi la funzionalità dell'unica *thing* della lampadina al sistema. Per le fonti di informazione, la *thing* potrebbe essere il tempo locale con informazioni da un servizio web con diversi *channels* come temperatura, pressione e umidità. I *channels* sono collegati agli *items* e questi

State	Descrizione
UNINITIALIZED	Stato iniziale di una <i>thing</i> quando viene aggiunta o il framework viene avviato. Questo stato viene assegnato anche se il processo di inizializzazione non è riuscito o l'associazione non è disponibile. I comandi inviati ai <i>channels</i> non verranno elaborati.
INITIALIZING	Stato che viene assegnato mentre l'associazione inizializza la <i>thing</i> . Dipende dall'associazione quanto tempo impiega il processo di inizializzazione. I comandi inviati ai <i>channels</i> non verranno elaborati.
UNKNOWN	Il device è completamente inizializzato ma, a causa della natura del dispositivo/servizio rappresentato, non è ancora in grado di dire realmente se la <i>thing</i> è ONLINE o OFFLINE. Pertanto, la <i>thing</i> potrebbe già funzionare correttamente e potrebbe o non potrebbe elaborare i comandi. Ma il framework è autorizzato a inviare comandi, perché alcuni dispositivi basati su radio possono andare ONLINE se viene inviato loro un comando. Il conduttore dovrebbe aver cura di passare la <i>thing</i> a ONLINE o OFFLINE il prima possibile.
ONLINE	Si presume che il dispositivo/servizio rappresentato da una <i>thing</i> funzioni correttamente e possa elaborare i comandi.
OFFLINE	Si presume che il dispositivo/servizio rappresentato da una <i>thing</i> non funzioni correttamente e potrebbe non elaborare i comandi. Ma il framework è autorizzato a inviare comandi, perché alcuni dispositivi basati su radio potrebbero tornare a ONLINE se viene inviato loro un comando.
REMOVING	Il dispositivo/servizio rappresentato da una <i>thing</i> dovrebbe essere rimosso, ma l'associazione non ha ancora confermato l'eliminazione. Alcune associazioni devono comunicare con il dispositivo per annullarne l'associazione dal sistema. Probabilmente la <i>thing</i> non funziona e i comandi non possono essere elaborati.
REMOVED	Stato indicante che il dispositivo/servizio rappresentato da una <i>thing</i> è stato rimosso dal sistema esterno dopo che la REMOVING è stata avviata dal framework. Di solito tale stato è uno stato intermedio perché la <i>thing</i> viene rimossa dal database dopo che esso è stato assegnato.

Tabella 2.2: Things states

Nome Item	Descrizione	Tipo di Comando
Color	Informazione sul colore(RGB)	OnOff, IncreaseDecrease, Percent, HSB
Contact	Stato di memorizzazione dell'articolo	OpenClosed
DateTime	Memorizzazione data e ora	-
Dimmer	Valore percentuale del dimmer	OnOff, IncreaseDecrease, Percent
Group	Oggetto che ne raccoglie altri	-
Image	Contiene i dati binari dell'immagine	-
Location	Memorizza le coordinate GPS	Point
Number	Memorizza valori nel formato numero e può avere una dimensione opzionale per il suffisso	Decimal
Number:<dimension>	Come Number ma con un'informazione addizionale per l'unità supportata	Quantity
Player	Permette di controllare i players come i riproduttori musicali	PlayPause, NextPrevious, RewindFastforward
Rollershutter	Tipicamente utilizzati per le tapparelle	UpDown, StopMove, Percent
String	Memorizza testi	String
Switch	Usati tipicamente per le luci	OnOff

Tabella 2.3: Items

collegamenti sono il collante tra livello fisico e virtuale. Stabilito il collegamento una *thing* reagisce agli eventi inviati per un *item* collegato ad uno dei suoi *channels* e a sua volta una *thing* può inviare eventi attraverso gli *item* sfruttando uno dei suoi *channel*.

2.1.4 Items

OpenHAB applica una forte separazione tra mondo fisico (*Things*) e mondo virtuale (*Items*). Gli *Items* rappresentano in particolarità le funzionalità usate dall'applicazione. Essi hanno uno *stato* e vengono usati attraverso degli *eventi*. Gli *Items* disponibili sono rappresentati alla Tabella 2.3.

Gli *items group* possono contenere *items* o *groups* al loro interno. In questo modo a livello di interfaccia utente viene vista una sola voce per il *group* che poi può fornire in dettaglio la vista di ogni singolo membro. Il raggruppamento ciclico non è vietato ma fortemente sconsigliato. Un esempio rappresentante la configurazione di un gruppo è alla Figura 2.3.

Group items possono derivare il proprio stato dai loro membri; per fare ciò i *group* devono essere costruiti a partire da un *base item* e una *group function*. Nel calcolare lo stato la *group function* attraversa i membri di gruppi e sottogruppi e se trova un *item* che definisce lui stesso uno stato allora viene preso questo come stato del gruppo.

Il *formato* dei vari *items* sarà



Figura 2.3: Esempio Group Item

Type	Valore supportato
IncreaseDecreaseType	INCREASE, DECREASE
NextPreviousType	NEXT, PREVIOUS
OnOffType	ON, OFF
OpenClosedType	OPEN, CLOSED
PlayPauseType	PLAY, PAUSE
RewindFastforwardType	REWIND, FASTFORWARD
StopMoveType	STOP, MOVE
UpDownType	UP, DOWN

Tabella 2.4: EnumTypes

- **StringType:** normali java strings
- **DateTimeType:** java SimpleDateFormat.parse() usando come matching pattern yyyy-MM-dd'T'HH:mm:ss.SSSZ, yyyy-MM-dd'T'HH:mm:ss.SSSz, yyyy-MM-dd'T'HH:mm:ss.SSSX, yyyy-MM-dd'T'HH:mm:ssz, yyyy-MM-dd'T'HH:mm:ss
- **DecimalType, PercentType:** java BigDecimal e java PercentType con valori tra 0 e 100
- **QuantityType:** valore numerico più un unità numerica
- **HSBType:** tre valori numerici separati dalla virgola che corrispondono a *Hue* (tonalità) compreso tra 0 e 360, *Saturation* (saturazione) compresa tra 0 e 100 e *Brightness* (luminosità) compresa tra 0 e 100
- **PointType:** sono tre valori numerici separati dalla virgola che indicano *latitudine*, *longitudine* e *altitudine*. I primi due vengono rappresentati in gradi mentre il terzo in metri.
- **EnumTypes:** rappresentati alla Tabella 2.4

A volte è necessario allegare informazioni aggiuntive agli elementi per determinati casi d'uso. Potrebbe trattarsi di un'applicazione che necessita di alcuni suggerimenti per rendere gli *items* in modo generico, o un'integrazione con assistenti vocali, o qualsiasi altro servizio che accede agli *items* e ha bisogno di comprenderne il significato. Per questo ci sono gli *items metadata* che possono essere allegati agli *item* utilizzando namespaces. Ogni metadata è una entry che ha un valore principale o una coppia opzionale di chiave/valore (esempio: Switch MyFan "My Fan" homekit="Fan.v2", alexa="Fan" [type="oscillating", speedSteps=3]).

2.1.5 Rules

Servono per automatizzare dei processi. Ogni regola invoca uno script che esegue dei tasks. Le *rules* sono nella cartella **\$OPENHAB_CONF/rules**. Vi sono degli esempi dai quali partire nel demo file chiamato *demo.rules*. Un file può contenere diverse regole che condividono lo stesso contesto di esecuzione. Le regole possono essere manipolate anche da UI. Ciò è possibile tramite l'accesso al web server tramite browser e andando nella sezione Settings/Rules. Qui tramite l'icona + è possibile aggiungere una nuova regola inserendo un *nome* e impostando un *trigger*. In seguito bisogna aggiungere un'azione che verrà scaturita dell'attivazione del trigger. Questa è lo script che andremo ad aggiungere. È possibile inserire o un ECMAScript in javascript o un Rule DSL in un linguaggio proprietario di openHAB. L'estensione *openHAB VS Code Extension* aiuta il programmatore a scrivere in modo corretto le *rules* per Rule DSL.

Il *rule file* è strutturato in

- Imports
- Variable Declarations
- Rules

Imports sono come in java; infatti non fanno altro che importare delle librerie per usarle all'interno del file. Esempio al Codice 2.1.

```
1 import java.net.URI
```

Codice 2.1: Import

Variable Declarations sono variabili usate dalle *rules* del file. Esse possono avere o no un valore iniziale e possono essere modificabili o read-only. Esempio al Codice 2.2.

```
1 // a variable with an initial value. Note that the variable type is
   automatically inferred
2 var counter = 0
3
4 // a read-only value, again the type is automatically inferred
5 val msg = "This is a message"
6
7 // an uninitialized variable where we have to provide the type (as it cannot
   be inferred from an initial value)
8 var Number x
```

Codice 2.2: Variable Declarations

Rules contengono la lista di regole e la loro sintassi deve essere come quella dell'esempio al Codice 2.3

```
1 rule "<RULE_NAME>"
2 when
3     <TRIGGER_CONDITION> [or <TRIGGER_CONDITION2> [or ...]]
4 then
5     <SCRIPT_BLOCK>
6 end
```

Codice 2.3: Rules

- <RULE_NAME>: ogni regola deve avere un nome univoco e possibilmente con un significato associato
- <TRIGGER_CONDITION>: sono le condizioni tramite le quali vengono applicate le regole. Una regola viene eseguita quando si verifica la sua condizione. Possono essere aggiunte più condizioni separate dalla parola chiave *or*
- <SCRIPT_BLOCK>: corrisponde alla logica che viene eseguita quando una certa condizione diventa vera

Triggers

Vi sono differenti tipi di *trigger* che possono essere inseriti

- **Item**(-Event)-based triggers: reagiscono agli eventi sul bus degli eventi openHAB, ovvero reagiscono a comandi o aggiornamenti di stato degli *Items*
- **Member of**(-Event)-based triggers: reagiscono agli eventi sul bus degli eventi openHAB per gli *Items* che fanno parte del gruppo fornito
- **Time**-based triggers: reagiscono ad una certa ora preimpostata
- **System**-based triggers: reagiscono ad un certo stato del sistema
- **Thing**-based triggers: reagiscono ad uno stato di una *thing*, per esempio il cambio tra online e offline

Event-based Triggers : È possibile ascoltare i comandi per un *item* specifico, sugli aggiornamenti di stato. Può essere scelto se catturare solo un comando/stato specifico o uno qualsiasi. La sintassi per questi casi è al Codice 2.4.

```
1 Item <item> received command [<command>]
2 Item <item> received update [<state>]
3 Item <item> changed [from <state>] [to <state>]
```

Codice 2.4: Event-based Triggers

Member of Triggers : come event-based triggers ma dei membri di un gruppo dato. Le variabili implicite vengono popolate dall'*item* che causa l'evento. La variabile *triggerItem* viene popolata dall'*item* che ha causato il trigger della *rule*. Esempio al Codice 2.5. I *member of triggers* lavorano con i membri diretti di un gruppo e non con gruppi nidificati.

```
1 Member of <group> received command [<command>]
2 Member of <group> received update [<state>]
3 Member of <group> changed [from <state>] [to <state>]
```

Codice 2.5: Member of Triggers

Time-based Triggers : si possono usare espressioni definite per le ore o espressioni cronologiche. I campi di un'espressione cronologica possono essere 6 o 7:

1. Seconds

2. Minutes
3. Hours
4. Day-of-Month
5. Month
6. Day-of-week
7. Year (opzionale)

Un esempio è al Codice 2.6.

```
1 Time is midnight
2 Time is noon
3 Time cron "<cron expression>"
```

Codice 2.6: Time-based Triggers

System-based Triggers : si dovrebbe usare System started trigger per inizializzare i valori allo startup se non lo sono. Esempio al Codice 2.7.

```
1 rule "Speedtest init"
2 when
3     System started
4 then
5     createTimer(now.plusSeconds(30), [
6         if (Speedtest_Summary.state == NULL || Speedtest_Summary.state == "")
7             ) Speedtest_Summary.postUpdate("unknown")
7     ])
8 end
```

Codice 2.7: System-based Triggers

Thing-based Triggers : si possono catturare eventi di cambiamento o aggiornamento di uno stato generato da una particolare *thing*. Si possono catturare uno specifico aggiornamento/cambiamento di stato o qualsiasi. La sintassi è mostrata con un esempio al Codice 2.8.

```
1 Thing <thingUID> received update [<status>]
2 Thing <thingUID> changed [from <status>] [to <status>]
```

Codice 2.8: Thing-based Triggers

Channel-based Triggers alcuni add-ons forniscono dei *trigger channels*. Un *trigger channel* fornisce informazioni su eventi discreti e non continui. Si può reagire ad uno specifico o ad ogni trigger che il *channel* fornisce. Vi è l'esempio al Codice 2.9.

```
1 Channel "<triggerChannel>" triggered [<triggerEvent>]
```

Codice 2.9: Channel-based Triggers

Scripts

Il linguaggio delle espressioni usato negli *scripts* è lo stesso usato nel linguaggio Xtend. La sintassi è simile a java ma ci sono molte caratteristiche che permettono di scrivere il codice in modo più conciso (come nell'uso delle collections). Non c'è bisogno che gli *scripts* vengano compilati ma possono essere interpretati. OpenHAB fornisce accesso a

- tutti gli *items* definiti, accessibili dal proprio nome
- tutti gli *stati/comandi* enumerati (ON, OFF, DOWN, ...)
- tutte le *standard actions*

Tramite la combinazione di questi possono essere creati *scripts* come l'esempio al Codice 2.10

```
1 if (Temperature.state < 20) {  
2     Heating.sendCommand(ON)  
3 }
```

Codice 2.10: Esempio Script

Le *Rules* vengono usate per manipolare lo stato o il valore di un *item*. Due comandi possono essere

- MyItem.postUpdate(<new_state>) - cambia lo stato di un *item* senza causare azioni implicite.
- MyItem.sendCommand(<new_state>) - cambia lo stato di un *item* e scatena i trigger di potenziali altre azioni.

Spesso nelle *rules* è necessario manipolare lo stato di un *item*. Ogni *item* in openHAB ha uno stato accessibile tramite l'attributo *state* (esempio: MyItem.state). Per usare uno stato di un *item* è necessario conoscere il *tipo* di stato e come convertire questo in valori di tipi primitivi con i quali possono essere fatte operazioni. Vengono divisi i tipi di comando e i tipi di stato. Per semplicità è possibile aggiungere la parola “*type*” alla fine del comando per capire il tipo di stato come all'esempio al Codice 2.11.

```
1 MyColorItem.sendCommand(ON) // --> OnOffType  
2 MyColorItem.sendCommand(INCREASE) // --> IncreaseDecreaseType  
3 MyColorItem.sendCommand(new PercentType(50)) // --> PercentType  
4 MyColorItem.sendCommand(new HSBTyPe(new DecimalType(123), new PercentType  
    (45), new PercentType(67))) // --> HSBTyPe
```

Codice 2.11: Esempio State Type

I *group* possono essere dichiarati con qualsiasi *item type* e lo stato interno del gruppo sarà di quel tipo (esempio: Group:Switch sarà OnOffType). Ciascun tipo di stato fornisce dei metodi utili per conversioni e calcoli. Variabili predefinite sono disponibili e possono essere utilizzate nelle regole. Esse sono

- **receivedCommand** - implicitamente disponibile in ogni regola che ha almeno un command event trigger
- **previousState** - implicitamente disponibile in ogni regola che ha almeno uno status change event trigger

- **newState** - implicitamente disponibile in ogni regola che ha almeno uno status update o status change event trigger
- **triggeringItemName** - implicitamente disponibile in ogni regola che ha almeno uno status update, uno status change o un command event trigger
- **triggeringItem** - implicitamente disponibile in ogni regola che ha un “Member of” trigger
- **receivedEvent** - implicitamente disponibile in ogni regola che ha un channel-based trigger

2.1.6 Sitemap

In openHAB *things* e *items* rappresentano gli oggetti fisici e logici della domotica dell’utente. Le *sitemaps* vengono utilizzate per selezionare e preparare questi elementi al fine di comporre una presentazione orientata all’utente di questa configurazione per varie interfacce utente.

Le *sitemaps* sono file di testo con estensione .sitemap e vengono archiviate nella cartella **\$OPENHAB_CONF/sitemaps**. Vi è di default un file demo denominato *demo.sitemap* da cui partire per configurarla in base alle proprie esigenze. Vi è un esempio del codice al Codice 2.12 e quello che viene prodotto dalla Figura 2.4.

```

1 sitemap demo label="My home automation" {
2     Frame label="Date" {
3         Text item=Date
4     }
5     Frame label="Demo" {
6         Switch item=Lights icon="light"
7         Text item=LR_Temperature label="Livingroom [%.1f C]"
8         Group item=Heating
9         Text item=LR_Multimedia_Summary label="Multimedia [%s]" icon="video"
10        {
11            Selection item=LR_TV_Channel mappings=[0="off", 1="DasErste", 2=
12                "BBC One", 3="Cartoon Network"]
13            Slider item=LR_TV_Volume
14        }
15    }
16 }
```

Codice 2.12: Esempio Codice di una Sitemap

Elements le *sitemaps* sono composte da vari elementi. L’esempio contiene come elementi *Frame*, *Text* e *Switch*. Gli elementi presentano informazioni, consentono l’interazione e sono altamente configurabili in base allo stato del sistema. Una riga di definizione dell’elemento della *sitemap* produce un elemento dell’interfaccia utente corrispondente. Come mostrato nell’esempio, ogni elemento genera un testo descrittivo accanto a un’icona sul lato sinistro e uno stato e/o elementi di interazione sulla destra.

Parameters è possibile configurare un determinato set di parametri per personalizzare la presentazione di un elemento. Nell’esempio mostrato *item* e *label* sono parametri. Quasi tutti i parametri sono opzionali, alcuni sono tuttavia necessari per ottenere un’interfaccia utente significativa. Per evitare righe di definizione degli elementi molto lunghe o non strutturate, i parametri possono essere suddivisi in più righe di codice.

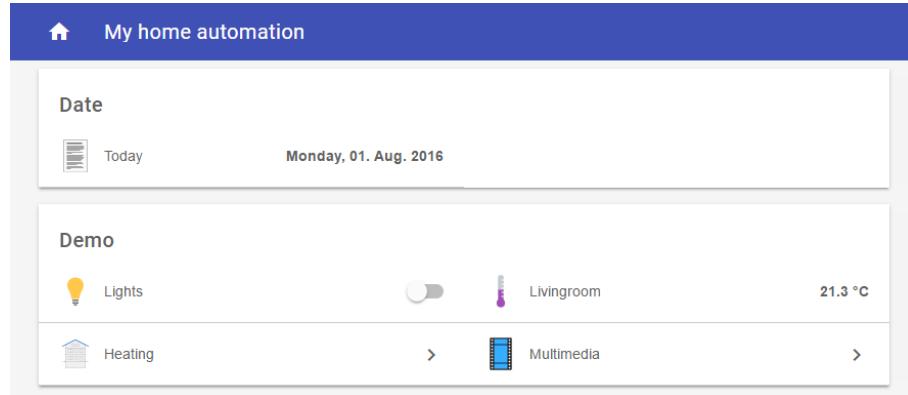


Figura 2.4: Esempio di View di una Sitemap

Blocks incapsulando gli elementi con parentesi graffe, è possibile annidare più elementi all'interno di altri. Il tipo di elemento *Frame* viene spesso utilizzato in combinazione con i blocchi di elementi. I frame vengono utilizzati per distinguere visivamente più elementi dello stesso argomento su una pagina dell'interfaccia. Quando si utilizzano blocchi di codice con altri tipi di elementi come *Text* o *Group*, questi elementi dell'interfaccia utente, oltre alla loro normale funzione, saranno collegamenti a una nuova vista, presentando gli elementi nidificati. Nell'esempio precedente, sono definiti più *Frame* e alcuni elementi non sono visibili nella vista principale ma sono accessibili dietro il loro elemento genitore. Questi sono indicati dall'icona di controllo “>” a destra di un elemento.

Dependencies una tipica *sitemap* contiene dozzine di singoli elementi. Lo stato del sistema e le possibili interazioni sono tuttavia spesso strettamente dipendenti. OpenHAB supporta queste dipendenze fornendo parametri per il comportamento dinamico.

L'elemento *sitemap* è obbligatorio nella definizione di una *sitemap*. Nell'esempio del Codice 2.12 alla riga 1 troviamo l'elemento *sitemap* con

- *sitemapname* ovvero *demo*. Deve essere lo stesso nome del file, quindi il file contenente tale codice deve chiamarsi *demo.sitemap*.
- *label* ovvero *My home automation*. Esso è il testo della *sitemap* mostrato a video.

Gli elementi che possono essere utilizzati in una *sitemap* possono essere quelli della Tabella 2.5.

2.2 Modello Semantico

Uno degli scopi di openHAB è quello di astrarre gli smart device vedendoli nel modo più generico possibile così da mantenere una logica semplice e facilmente gestibile con l'annessione di qualsiasi altro dispositivo. Proprio per questo vengono usati gli *items* che nascondono l'oggetto fisico. Essi sono le entità principali con le quali lavorano le API Rest di openHAB. I vari *items* vengono raccolti in un modello semantico che permette di organizzarli in maniera semplice ed intuitiva in modo tale da semplificare l'uso da parte dell'utente.

Element	Descrizione
Chart	Aggiunge un oggetto grafico rappresentante una serie temporale per i dati persistiti
Colorpicker	Permette l'utente di scegliere un colore da una color wheel
Default	Esegue il rendering di un elemento nella rappresentazione dell'interfaccia utente predefinita specificata dal tipo di elemento specificato
Frame	Stabilisce un'area contenente vari altri elementi della sitemap
Group	Concentra tutti gli elementi di un dato gruppo in un blocco annidato
Image	Rende un'immagine data da un URL
Mapview	Visualizza una mappa OSM basata su un determinato elemento posizione
Selection	Fornisce un menu a discesa o un popup modale che presenta i valori tra cui scegliere per un elemento
Setpoint	Rende un valore compreso tra i pulsanti di aumento e diminuzione
Slider	Presenta un valore in un cursore simile a una barra di avanzamento
Switch	Rende un oggetto come interruttore ON/OFF o multi-pulsante
Text	Rende un oggetto come testo
Video	Visualizza un flusso video, dato un URL
Webview	Visualizza il contenuto di una pagina web

Tabella 2.5: Tipi di Elementi di una Sitemap

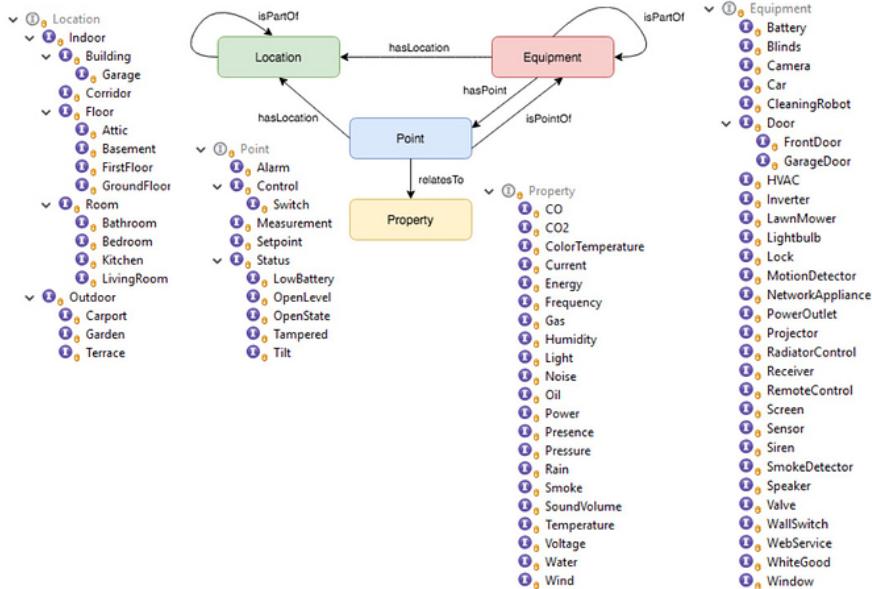


Figura 2.5: Relazioni Ontologiche

Come rappresentato all'esempio della Figura 2.5 vi sono 4 concetti fondamentali nel modello semantico

- **Location** è un *group item* che può contenere altre location, equipments e points e rappresenta una posizione fisica (edificio, stanza, ...). Una Location può essere membro di un solo Location Group
- **Equipment** è normalmente un *group item* che può contenere un equipment secondario e points. Un Equipment può essere membro diretto di una sola Location o un solo Equipment
- **Point** non è un gruppo, ma rappresenta qualsiasi altro tipo di elemento ed è solitamente collegato a un channel. Un Point può essere membro diretto di una sola Location o un solo Equipment
- **Property** è un tag aggiuntivo su un point item che indica che tipo di punto è (esempio: un termometro potrebbe essere un point di tipo misurazione con una property di tipo temperatura)

A Figura 2.6 vi è un esempio avanzato di modello.

2.3 Persistence

OpenHAB permette di tenere traccia degli stati storici degli *items* attraverso l'uso della persistenza. Con essa quindi si può

- Tracciare un grafico della lettura di un sensore di temperatura nel tempo
- Ripristinare un elemento allo stato che aveva prima della chiusura o del riavvio di openHAB
- Usare lo stato di un oggetto nel passato, o qualche aggregato dello stato di un oggetto nel passato (esempio: Media da un'ora fa) nelle regole di automazione

Per persistere i dati vengono forniti più database sia interni che esterni. Quelli esterni richiedono l'installazione e la configurazione su server separato.

Quando *persistence* salva gli stati degli *items*?

- quando un *item* cambia
- quando un *item* viene aggiornato
- quando un *item* riceve un comando
- ogni minuto se ha ricevuto un evento

Una strategia di persistenza è *restoreOnStartup* che permette di aggiornare gli *items* con gli stati salvati più recentemente. Può essere possibile creare nella cartella **\$OH_CONF/persistence** un file con estensione *.persist* per definire la strategia di persistenza.

Il database predefinito per la persistenza è *rrd4j* e viene fornito con strategie *everyChange*, *everyMinute* e *restoreOnStartup*. La comodità di questo database è che rimane sempre di una dimensione preimpostata e non c'è bisogno di pulirlo. Per eseguire tale logica rimpiazza i vecchi records con quelli nuovi. Questo approccio non funziona con tutti i tipi di *items*.

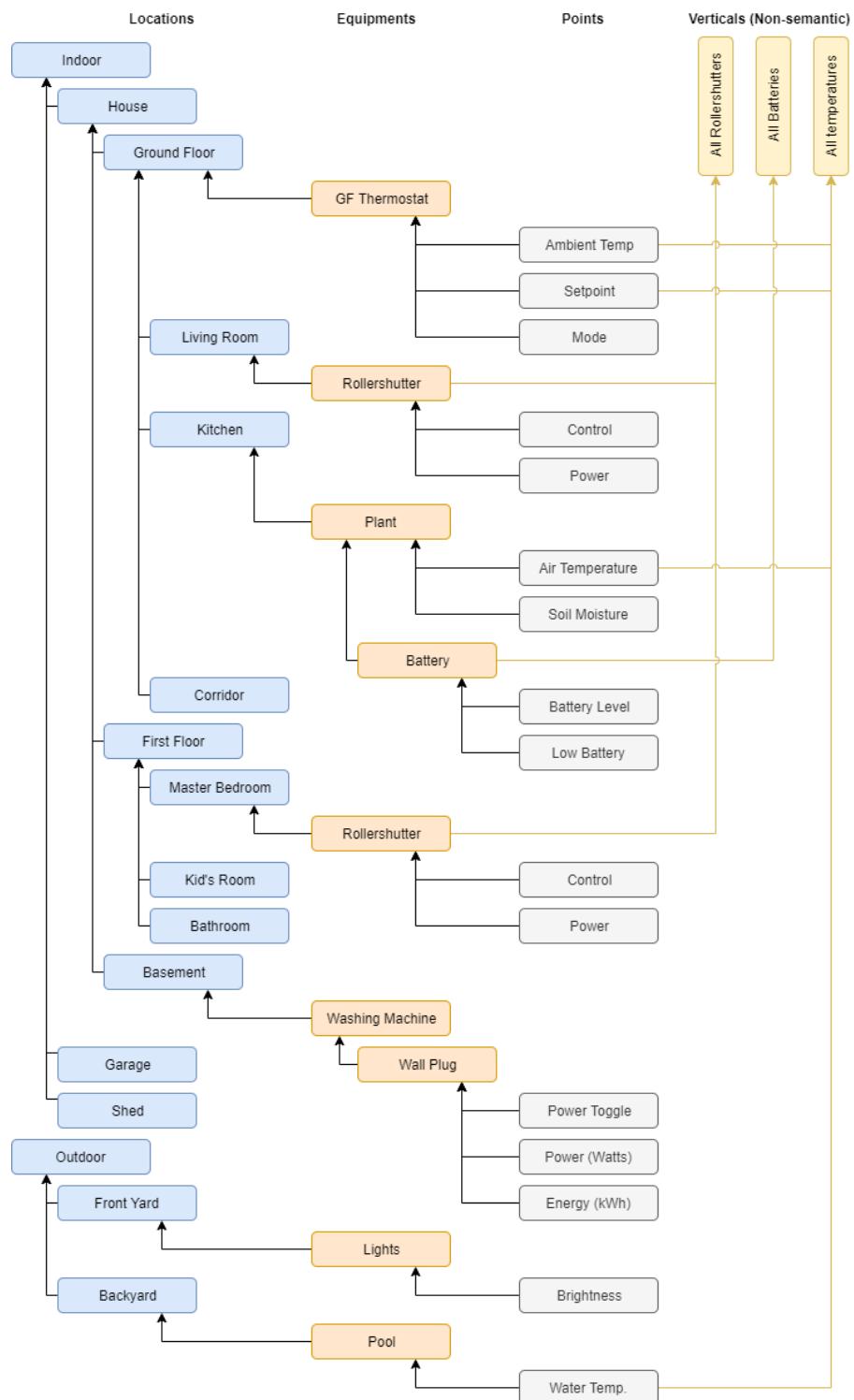


Figura 2.6: Esempio di Modello

2.4 openHAB REST API

Tramite le REST API di openHAB è possibile accedere alla piattaforma da altre applicazioni e manipolare i dati visti nella struttura. Ciò include l'accesso ad *Item*, *Things* e *Bindings* e la modifica dei vari stati. Le interazioni con l'API REST si basano su protocollo *http*. È possibile accedere tramite *internet* all'API REST ma a discapito della sicurezza, poiché in tale modo si espone openHAB a chiunque è in rete. Gli utenti infatti sono incoraggiati a garantire connessioni sicure e protette. *OpenHAB API REST* può non essere installata automaticamente ma una volta fatto è accessibile tramite la dashboard.

Esempi di applicazioni di REST API in openHAB sono

- Recupera i dati openHAB da applicazioni esterne
- Inietta dati e attiva eventi in openHAB da applicazioni esterne (esempio: alcuni rilevatori di movimento o telecamere di sorveglianza)
- Ispeziona bindings, things o items di openHAB, scopri gli states, i parametri o i problemi correnti
- Interagire con openHAB da altri programmi; molti linguaggi di programmazione e strumenti di automazione possono facilmente utilizzare l'API REST
- Utilizzo di software di terze parti sui telefoni cellulari, come tasker per aprire la porta del garage

Esempi di applicazione pratica tramite il tool *curl* che permette di fare chiamate REST al servizio server

- Comutazione My_ItemOFF mediante l'emissione di un HTTP POST richiesta

```
1 curl -X POST --header "Content-Type: text/plain" --header "Accept: application/json" -d "OFF" "http://{openHAB_IP}:8080/rest/items/My_Item"
```

- Impostare una voce di contatto My_Item su CHIUSO emettendo una richiesta http PUT a My_Item/state

```
1 curl -X PUT --header "Content-Type: text/plain" --header "Accept: application/json" -d "CLOSED" "http://{openHAB_IP}:8080/rest/items/My_Item/state"
```

- Recupero di un elenco di tutti gli items e groups mediante l'emissione di una richiesta GET

```
1 curl -X GET --header "Accept: application/json" "http://{openHAB_IP}:8080/rest/items?recursive=false"
```

- Recupero di un elenco di tutte le sitemap mediante l'emissione di una richiesta GET

```
1 curl -X GET --header "Accept: application/json" "http://{openHAB_IP}:8080/rest/sitemaps"
```

- Iscrizione agli eventi

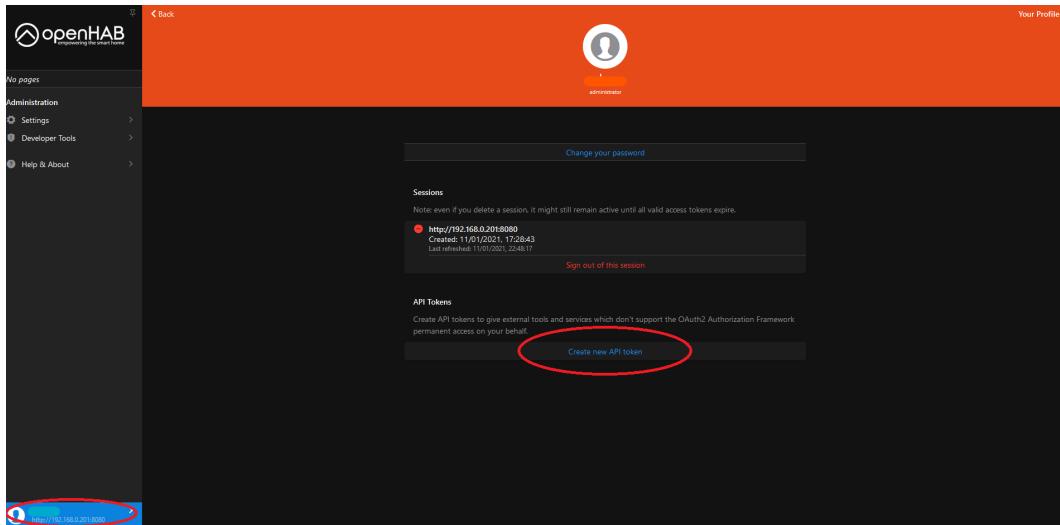


Figura 2.7: API Token Login

```

1 curl "http://{openHAB_IP}:8080/rest/events?topics=smarthome/things
      /{thingUID}/statuschanged"
2 curl "http://{openHAB_IP}:8080/rest/events?topics=smarthome/
      channels/{channelUID}/triggered"

```

OpenHAB supporta anche la protezione tramite password per contenuti sensibili come parti del modello semantico. I meccanismi forniti per tale scopo sono l'*basic authentication* e l'*OAuth authorization*. La maggior parte dei linguaggi di programmazione già supporta queste tecnologie. Per aggiungere utente e password nelle chiamate *curl* basta aggiungere `-u {USER_NAME}` e poi inserire la password richiesta. Per fare tutto con un singolo comando basta aggiungere `-u {USER_NAME:PASSWORD}`. Con l'API token basta aggiungere il comando `-u {API_TOKEN}`.

È possibile creare un API Token dal proprio profilo accedendo con username da amministratore e password, andando sul profilo e premendo su Create new API token. Illustrazione a Figura 2.7.

3. MQTT Binding

Come già spiegato alla sezione 2.1.1 un *binding* permette di connettere un dispositivo fisico ad una *thing*. In questo caso specifico è stato sviluppato un *sistema IOT* per sperimentare l'associazione con openHAB. Per collegare il device ed il framework è stato utilizzato il protocollo di rete **MQTT**.

3.1 Protocollo MQTT

Message Queuing Telemetry Transport è un protocollo di rete leggero di tipo *publish-subscribe* che funziona su TCP/IP. MQTT è progettato per connessioni con postazioni remote in cui è richiesta un'impronta di codice ridotta o una larghezza di banda della rete limitata [Wik21].

Il protocollo definisce due tipi di entità di rete:

- **broker**
- **client**

Esse vengono illustrate a Figura 3.2.

broker server che riceve tutti i messaggi dai *client* e li instrada ai *client* di destinazione appropriati. Il *broker MQTT* è un software in esecuzione su un computer in locale o remoto. Di esso sono disponibili sia implementazioni open source che proprietarie. Il *broker* riceve solitamente messaggi da un *publisher* che distribuisce a più *subscriber*. Sia il *publisher* che il *subscriber* sono *client MQTT*. Il *broker* mantiene traccia di tutte le informazioni della sessione mentre i dispositivi si accendono e si spengono. Esse vengono chiamate *persistent sessions*.

client qualsiasi dispositivo che esegue una libreria MQTT per connettersi ad un broker MQTT in rete. Un *client* ha dialogo diretto solamente con un broker e non con altri *client*; il broker infatti fa da tramite tra lui e gli altri. I *client* possono *pubblicare* o *sottoscriversi* per inviare o ricevere dati.

topic le informazioni sono organizzate in gerarchie di argomenti chiamati *topic*. Quando un *client* vuole inviare un messaggio ad un altro, basta che lo invia al broker con lo stesso *topic* a cui è sottoscritto il *client*.

retained message Se viene inviato un messaggio al *broker* relativo ad un *topic* dove non vi sono *client* sottoscritti viene scartato. Ciò non viene fatto nel caso di *retained*



Figura 3.1: MQTT Logo

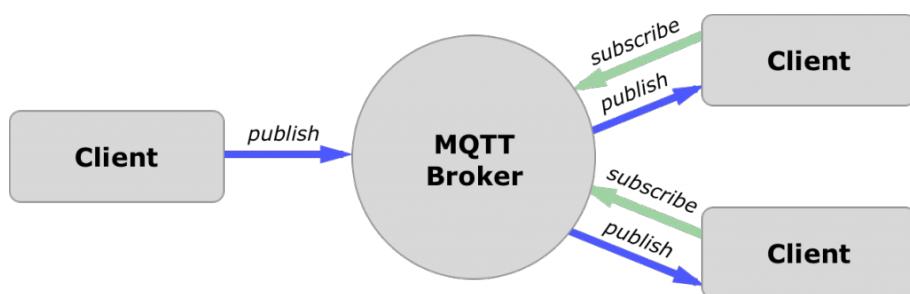


Figura 3.2: MQTT Model Structure

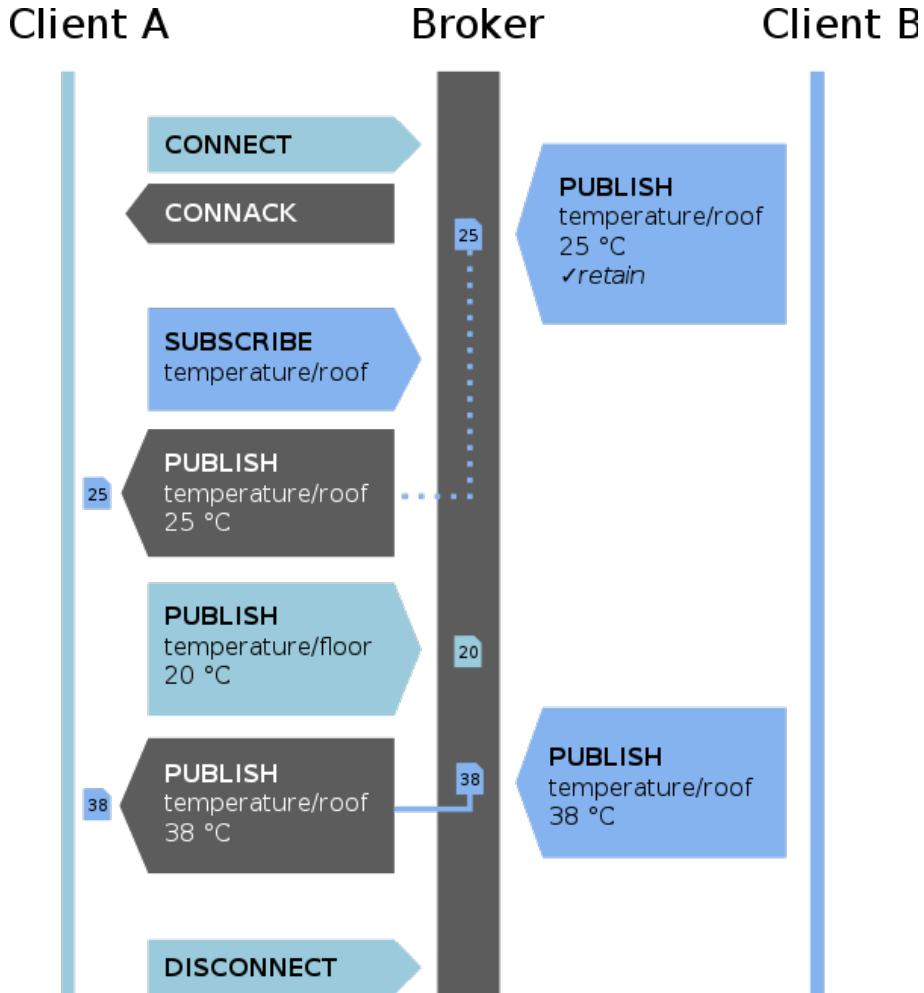


Figura 3.3: MQTT Connection

message. Esso è un normale messaggio MQTT con flag *retain* a true. In caso viene inviato un *retained message*, questo viene memorizzato insieme al rispettivo QoS. In tal caso quindi possono essere salvati gli ultimi messaggi nei propri *topic* così da notificare un subscriber degli ultimi dati inviati appena si sottoscrive.

message types sono utilizzati durante la connessione illustrata a Figura 3.3.

- **connect:** aspetta una connessione con il *broker* e crea un collegamento tra i nodi
- **disconnect:** attende la terminazione del lavoro da parte del *client* e la disconnessione della sessione TCP/IP.
- **publish:** ritorna il messaggio inviato da un *client MQTT* al broker immediatamente al thread dell'applicazione di ogni client sottoscritto allo stesso topic.

3.2 openHAB Binding

openHAB implementa un *binding* già configurato per il protocollo MQTT. Tale protocollo ha un'architettura client/server. Il server di MQTT, ovvero il *broker* NON viene

fornito nel binding di openHAB ma puó essere utilizzato qualsiasi altro software che fá da *broker MQTT* come *Mosquitto*. Il *binding openHAB* permette di configurare le connessioni ai broker tramite le *openHAB things*. L'associazione non permette però di collegare i *channels* ai *topic* o di eseguire il rilevamento automatico dei *topic* disponibili. I *bridge* supportati sono:

- **Broker:** *bridge* che rappresenta una connessione *broker MQTT* configurata e gestita da tale *binding*.
- **SystemBroker:** un *broker* configurato dal sistema non puó essere modificato da questo *binding* e verrá elencato come *broker* di sistema di sola lettura.

3.2.1 Configurazione bridge

- **host:** IP/hostname del broker MQTT. *Binding* che consente una connessione univoca con *host:porta*.
- **port:** se non ne viene fornita nessuna vengono utilizzate quelle di default che sono 1883 e 8883(SSL). *Binding* che consente una connessione univoca con *host:porta*.
- **secure:** se si vuole utilizzare una connessione sicura con il *broker* tramite TLS/SSL. Puó essere true o false. Di default é false.

Altri parametri da impostare possono essere:

- **qos:** indica qualitá del servizio che puó essere impostata a 0, 1 o 2 (default 0).
- **clientID:** ID client fisso che viene generato automaticamente.

I parametri di riconnessione sono:

- **reconnectTime:** tempo di riconnessione in millisecondi. Valore del tempo di attesa aspettato per una connessione dopo che ne viene persa una. Il valore predefinito é 60000 (60s).
- **keepAlive:** tempo in secondi utile per capire se una connessione al server é persa o no. IL valore di default 60s.

Puó essere impostato un LWT(testamento) MQTT:

- **lwtMessage:** messaggio del testamento. Default vuoto.
- **lwtTopic:** topic del testamento. Default vuoto.
- **lwt:** qos del testamento. Default 0.
- **lwtRetain:** conserva l'ultimo messaggio. True o false, false di default.

Inoltre si possono impostare i seguenti parametri opzionali:

- **username:** nome utente MQTT
- **password:** password MQTT

- **certificatepin**: se impostato dopo che è stata stabilita una connessione controlla il certificato. Se questo risulta non essere valido viene interrotta la connessione. Questa opzione aumenta la sicurezza.
- **publickeypin**: se impostata dopo che è stata stabilita una connessione controlla la chiave pubblica. Se questa risulta non essere valida viene interrotta la connessione. Questa opzione aumenta la sicurezza.
- **certificate**: hash del certificato utile per verificarne l'autenticità per mezzo di *certificatepin*.
- **publickey**: hash della chiave pubblica utile per verificarne l'autenticità per mezzo di *publickeypin*.
- **enableDiscovery**: di default sono abilitati i servizi di rilevamento sul broker. Questi possono essere controllati per mezzo di tale parametro.

3.2.2 Things supportate

Data la struttura molto generica di MQTT, il *binding* consente di aggiungere un numero arbitrario di *things MQTT* generiche. Su ogni *thing* possono essere aggiunti un numero arbitrario di *channels*.

Channels supportati

- **string**: invia o riceve del testo su un *topic MQTT*.
- **number**: invia o riceve un numero su un *topic MQTT*.
- **dimmer**: gestisce un valore percentuale.
- **contact**: rappresenta uno stato di OPEN/CLOSE.
- **switch**: invia o riceve uno stato ON/OFF su un *topic MQTT*.
- **color**: gestisce i valori di un colore nei formati HSB, RGB o xyY (x, y, luminosità..).
- **location**: gestisce una posizione.
- **image**: gestisce un'immagine binaria nei formati supportati da Java (bmp, jpg, png).
- **datetime**: gestisce un valore data/ora.
- **rollershutter**: gestisce una tapparella.

3.2.3 Configurazione channel

- **stateTopic**: *topic MQTT* che rappresenta lo stato della *thing*. Possono essere utilizzati caratteri jolly per recuperare lo stato da più *topic*.
- **trasformationPattern**: pattern di trasformazione opzionale applicato a tutti i valori di entrata.

- **trasformationPatternOut**: pattern di trasformazione opzionale applicato a tutti i valori di uscita.
- **commandTopic**: *topic* per inviare comandi in uscita. Se non viene impostato il *channel* é in modalità read-only.
- **formatBeforePublish**: utile per formattare un valore prima che viene pubblicato nel broker MQTT. Di default é utilizzato per il passaggio tra *channel* e *item*.
- **postCommand**: azione che permette di inviare ad altri elementi il valore ricevuto per un particolare *topic* MQTT.
- **retained**: verrá inviato il valore come messaggio MQTT mantenuto.
- **qos**: qos del *channel*.
- **trigger**: se true il *topic* dello stato aggiornerá un *channel* al suo posto.

string

- **allowStates**: elenco separato da virgole di stati consentiti.

number

- **min**: valore minimo
- **max**: valore massimo
- **step**: per canali aumenta o decrementa
- **unit**: unitá di misura

contact, switch

- **on**: un numero opzionale (come 1, 10) o una stringa (come “ON”/“Open”) riconosciuti come stato on/open.
- **off**: un numero opzionale (come 0, -10) o una stringa (come “OFF”/“Close”) riconosciuti come stato off/close.

color

- **color_mode**: stringa obbligatoria che definisce la rappresentazione del colore: “hsb”, “rgb”, “xyY”.
- **on**: una stringa facoltativa (come “BRIGHT”) riconoscita come on.
- **off**: una stringa facoltativa (come “DARK”) riconoscita come off.
- **onBrightness**: se collegato ad un *item* switch ed é acceso.

location

Puó essere collegato tale *channel* ad un *Location item*. Esso pubblicherá la posizione come elenco separato da virgole sul broker MQTT (esempio: “112,54,123”, ovvero latitudine, longitudine e altitudine).

image

Puó essere collegato tale *channel* ad un *Image item* ed é di tipo read-only. I valori devono essere in binario e del formato supportato da java (bmp, jpg, png).

datetime

Puó essere collegato tale *channel* ad un *DateTime item*. Pubblicherá la data/ora nel formato “aaaa-MM-gg’T’HH::mm”.

rollershutter

- **on**: stringa facoltativa come “Open” riconosciuta come UP.
- **off**: stringa facoltativa come “Close” riconosciuta come DOWN.
- **stop**: stringa facoltativa come “Stop” riconosciuta come STOP.

3.2.4 Rule actions

Tale *binding* include una regola di azione che consente di pubblicare messaggi MQTT dall'interno delle regole.

```
1 val mqttActions = getActions("mqtt", "mqtt:systemBroker:embedded-mqtt-broker")
```

Codice 3.1: Esempio Rule Action Istanza

Nel Codice 3.1 il primo parametro deve essere sempre mqtt mentre il secondo é il Thing UID del broker usato. Una volta recuperata l'istanza d'azione basta pubblicare con in comando publishMQTT(String topic, String value, Boolean retain) come nel Codice 3.2.

```
1 mqttActions.publishMQTT("mytopic", "myvalue", true)
```

Codice 3.2: Esempio Rule Action Pubblicazione

4. Configurazione dell'MQTT Binding per un sistema IOT

Per testare openHAB con un sistema IOT si é deciso di procedere allo sviluppo di uno *Smart Garden*, ovvero un sistema per la gestione automatizzata del proprio giardino. Di base si tratta di un sensore per la misurazione dell'umiditá del terreno e una pompa per l'attivazione dell'innaffiamento in caso di siccitá.

4.1 Smart Garden

Il sistema IOT quindi nello specifico é composto da:

- **Soil Moisture Sensor:** sensore di umiditá del terreno (Figura 4.1).
- **Water Pump:** pompa dell'acqua utile per irrigare alla sua attivazione (Figura 4.2).
- **Raspberry Pi:** unitá logica che permette di rendere dinamica l'interazione tra i sensori. È la centralina del sistema IOT (Figura 4.3).
- **Relé:** elemento utile per l'attivazione e disattivazione della pompa (Figura 4.4).
- **Alimentatore da 12 Volt:** alimentazione della pompa (Figura 4.5).

Soil Moisture Sensore

Il *Soil Moisture Sensor* é un sensore che rileva l'umiditá del suolo. Esso ha una forma a ferro di cavallo dove alle due estremitá vi sono due contatti. Tra i contatti del sensore viene fatta passare una tensione, tanto maggiore è l'umiditá tra i due contatti, tanto minore sará la resistenza, permettendo alla corrente di fluire da un contatto verso l'altro.

Esso ha lo scopo di segnalare o meno la presenza di umiditá cosí da scatenare poi l'evento di start o stop della pompa dell'acqua. Il sensore viene collegato per mezzo di cavi ai pin della Raspberry Pi per inviare segnali di input.

Water Pump

La Pompa dell'Acqua permette di prendere l'acqua e dirigerla verso una direzione per l'irrigazione del proprio giardino. Essa ha la caratteristica di essere totalmente immersa nell'acqua e presenta nella parte superiore un buco nella quale viene infilato un tubo che sará poi collegato nell'altra estremitá ad un erogatore d'acqua utile ad innaffiare.

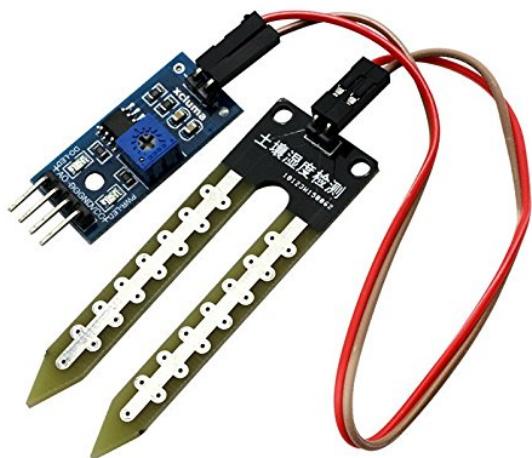


Figura 4.1: Soil Moisture Sensore



Figura 4.2: Water Pump



Figura 4.3: Raspberry Pi

La Pompa viene collegata tramite cavi ad un Relé per la sua attivazione ed all'alimentatore da 12 volt poiché la raspberry non può sostenere tale carico. Essa viene attivata in caso di siccità e spenta altrimenti.

Raspberry Pi

Centralina del sistema IOT. Essa permette di essere programmata per leggere e controllare i vari dispositivi che gli vengono collegati. Per collegarsi ad un sensore o dispositivo esterno vengono messi a disposizione dei pin ai quali vengono agganciati dei cavi.

La Raspberry viene collegata ai vari sensori e dispositivi per implementare la logica del sistema.

La Raspberry Pi è dotata di una scheda di rete tramite la quale comunicherà esternamente con il device in cui è installato openHAB.

Relé

Elemento che permette tramite il passaggio della corrente di attivarsi o disattivarsi. Esso è come un pulsante che si accende e si spegne al passaggio della corrente.

Utile nel sistema per attivare e disattivare la Pompa. La Raspberry quando vuole attivare/disattivare la Water Pump attiva/disattiva il Relé che a sua volta attiva/disattiva il circuito tra Pompa e la sua alimentazione.

Alimentatore da 12 Volt

Alimentazione utile per attivare la pompa. Essa non poteva funzionare con l'alimentazione fornita da Raspberry che arriva massimo a 5 volt.

L'alimentatore viene collegato al Relé e alla pompa.

È possibile visualizzare un semplice schema dei collegamenti a Figura 4.6.

4.1.1 Sviluppo sistema IOT

Il sistema IOT viene controllato tramite Raspberry Pi da un piccolo programma python. Esso ha accesso tramite la libreria RPi.GPIO ai pin per gestire i dispositivi o sensori agganciati. Il programma legge in input il valore prelevato dal sensore di umidità

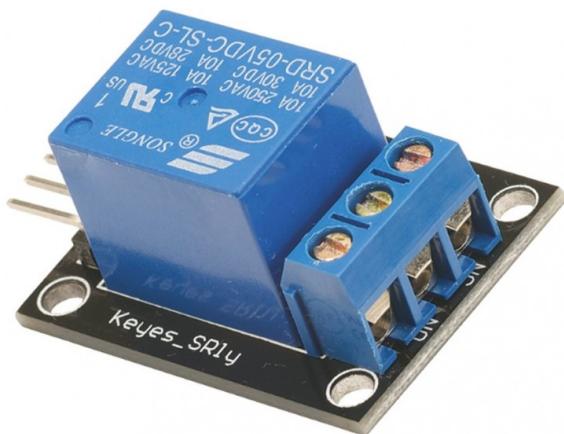


Figura 4.4: Relé



Figura 4.5: Alimentatore da 12 Volt

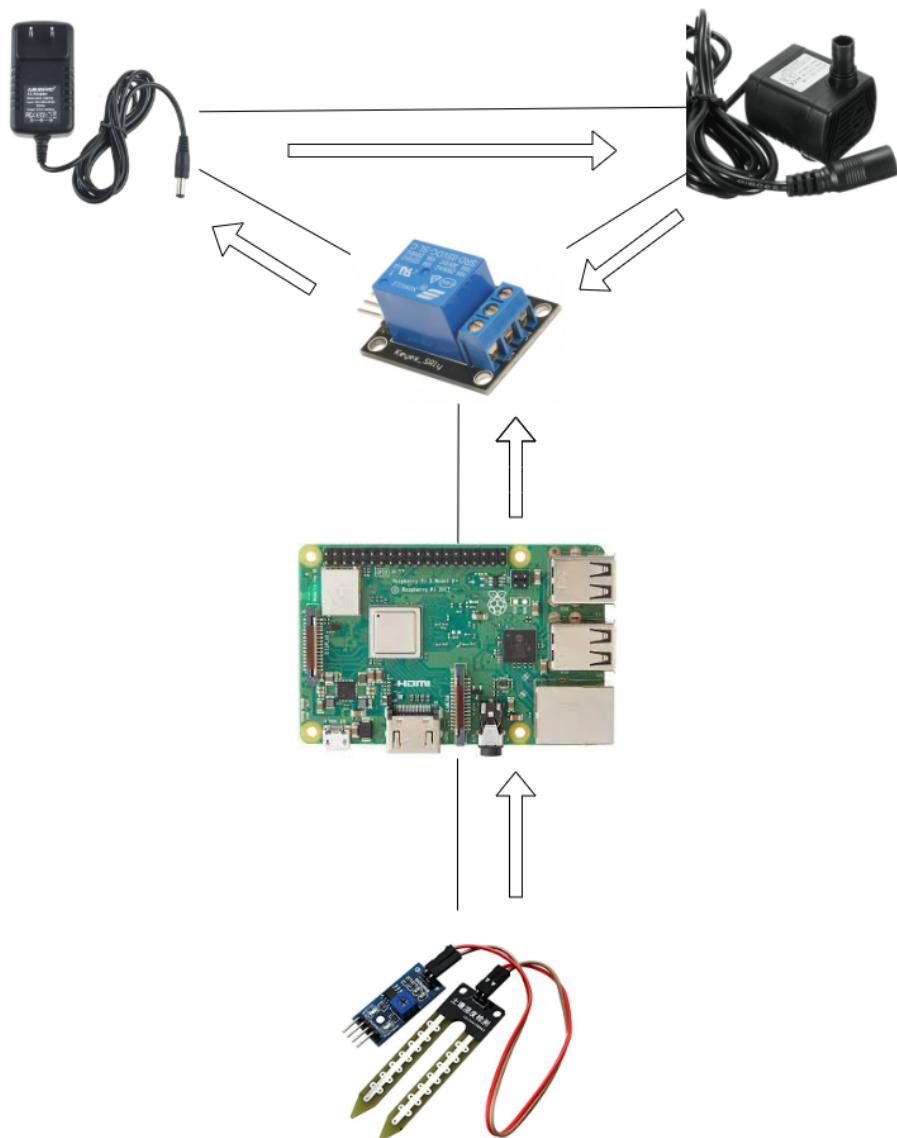


Figura 4.6: Struttura Smart Garden

mentre attiva e disattiva in output la pompa tramite il relé. Tutto ciò viene gestito tramite protocollo MQTT con libreria `paho.mqtt.client`. Tramite dei topic infatti si può:

- leggere il sensore di umidità
- attivare o disattivare la pompa manualmente
- attivare o disattivare il sistema in modalità automatica, ovvero sarà lui ad attivare o disattivare la pompa in caso non ci sia o ci sia un terreno umido.

Codice del sistema IOT

Il programma è composto da 4 file principali.

- **Soil.py**: contiene la classe per la creazione dell'oggetto capace di gestire il sensore di umidità del terreno.
- **Pump.py**: contiene la classe per la creazione dell'oggetto capace di gestire il relé per controllare la pompa dell'acqua.
- **MQTTClient.py**: contiene la classe per la creazione dell'oggetto capace di fare publish e subscribe in MQTT.
- **main.py**: file main da cui ha inizio il programma ed in cui è la logica di business.

```
1 import RPi.GPIO as GPIO
2
3 class Soil:
4
5     def __init__(self, channel):
6         self.channel = channel
7         GPIO.setmode(GPIO.BCM)
8         GPIO.setup(self.channel, GPIO.IN)
9
10    def status(self):
11        return GPIO.input(self.channel)
```

Codice 4.1: Soil.py

```
1 import RPi.GPIO as GPIO
2
3 class Pump:
4
5     def __init__(self, channel):
6         self.channel = channel
7         GPIO.setmode(GPIO.BCM)
8         GPIO.setup(self.channel, GPIO.OUT)
9
10    def on(self):
11        GPIO.output(self.channel, GPIO.HIGH)
12
13    def off(self):
14        GPIO.output(self.channel, GPIO.LOW)
```

Codice 4.2: Pump.py

```

1 import paho.mqtt.client as mqtt
2
3 class MQTTClient:
4
5     def __init__(self, ip, port):
6         self.ip = ip
7         self.port = port
8
9     def publish(self, topic, to_publish):
10        self.client = mqtt.Client()
11        self.client.connect(self.ip, self.port, 60)
12        self.client.publish(topic, to_publish, 0, True);
13        self.client.disconnect()
14
15    def subscribe(self, topic, on_message):
16        client = mqtt.Client()
17        client.connect(self.ip, self.port, 60)
18        client.on_connect = lambda client, userdata, flags, rc : self.
19            on_connect(client, userdata, flags, rc, topic)
20        client.on_message = on_message
21        client.loop_start()
22        return client
23
24    def on_connect(self, client, userdata, flags, rc, topic):
25        print("[ " + topic + "]: connected with result code " + str(rc))
26        client.subscribe(topic)

```

Codice 4.3: MQTTClient.py

```

1 import time
2 import threading
3
4 from MQTTClient import MQTTClient
5 from Soil import Soil
6 from Pump import Pump
7
8 import os
9 from os.path import join, dirname
10 from dotenv import load_dotenv
11
12 dotenv_path = join(dirname(__file__), '.env')
13 load_dotenv(dotenv_path)
14
15 # mqtt connection data
16 MQTT_IP = os.environ.get("MQTT_IP")
17 MQTT_PORT = (int)(os.environ.get("MQTT_PORT"))
18
19 # topics
20 TOPIC_HANDLE_PUMP = os.environ.get("TOPIC_HANDLE_PUMP")
21 TOPIC_HANDLE_AUTO = os.environ.get("TOPIC_HANDLE_AUTO")
22 TOPIC_STATUS_SOIL = os.environ.get("TOPIC_STATUS_SOIL")
23
24 # pins
25 PIN_PUMP = (int)(os.environ.get("PIN_PUMP"))
26 PIN_SOIL = (int)(os.environ.get("PIN_SOIL"))
27
28 # global variable to start and stop auto
29 stop_thread = True
30
31 mqtt_client = MQTTClient(MQTT_IP, MQTT_PORT)
32

```

```
33 soil = Soil(PIN_SOIL)
34 pump = Pump(PIN_PUMP)
35
36 # threading functions
37 def publish_pump(status):
38     global status_thread
39     if not stop_thread:
40         mqtt_client.publish(TOPIC_HANDLE_PUMP, status)
41
42 def callback():
43     if soil.status():
44         publish_pump('ON')
45         mqtt_client.publish(TOPIC_STATUS_SOIL, 'Water Not Detected')
46     else:
47         publish_pump('OFF')
48         mqtt_client.publish(TOPIC_STATUS_SOIL, 'Water Detected')
49
50 def loop():
51     while True:
52         callback()
53         time.sleep(1)
54
55 # on_message functions
56 def on_message_auto(client, userdata, msg):
57     global stop_thread
58     stop_thread = msg.payload.decode() != "ON"
59
60 def on_message_pump(client, userdata, msg):
61     global pump
62     if msg.payload.decode() == "ON":
63         pump.on()
64     else:
65         pump.off()
66
67 # main function
68 if __name__ == "__main__":
69     thread = threading.Thread(target=loop)
70     thread.start()
71     mqtt_client.subscribe(TOPIC_HANDLE_AUTO, on_message_auto)
72     mqtt_client.subscribe(TOPIC_HANDLE_PUMP, on_message_pump)
```

Codice 4.4: main.py

In breve alla partenza del programma vengono inizializzate le costanti prelevate dalle variabili d'ambiente relative a

- **MQTT_IP**: IP del client MQTT.
- **MQTT_PORT**: porta del client MQTT.
- **TOPIC_HANDLER_PUMP**: topic MQTT per gestire l'attivazione e la disattivazione manuale della pompa.
- **TOPIC_HANDLER_AUTO**: topic MQTT per gestire l'attivazione e la disattivazione della modalità automatica.
- **TOPIC_STATUS_SOIL**: topic MQTT per leggere lo stato del sensore di umidità del suolo.
- **PIN_PUMP**: pin di controllo della pompa.

- **PIN_SOIL:** pin per la lettura del valore dal sensore di umiditá.

Poi vengono inizializzati i vari oggetti relativi alla gestione sia dei sensori che del client MQTT. Oltre a questi viene inizializzato un flag chiamato `stop_thread` utile ad avviare e fermare l'esecuzione della modalità automatica.

Modalitá automatica Successivamente viene riportato lo sviluppo della modalità automatica, ovvero l'esecuzione di un thread che permette di mettere in un loop infinito una funzione capace di prelevare il dato dal sensore di umiditá e attivare o disattivare la pompa in caso di siccità o presenza di acqua. Il loop aspetta sempre un intervallo di un secondo. Inoltre la funzione viene eseguita solamente se la variabile globale `stop_thread` é False.

In realtá quindi l'esecuzione del thread serve per prelevare il dato in input dal sensore di umiditá e solo se il flag rispetta la condizione viene attivata o disattivata la pompa; quindi l'esecuzione automatica avviene per mezzo dell'attivazione del flag.

Oltre ad essere letti, i dati vengono anche notificati tramite un publish MQTT; ovvero vengono notificati sia i cambiamenti di stato del sensore di umiditá, sia l'accensione o spegnimento della pompa al topic della modalità manuale.

Esecuzione All'esecuzione del programma quindi viene fatto partire il thread per prelevare il dato dal sensore di umiditá con il flag per la modalità automatica disabilitato, dopodiché vengono inizializzati i subscriber MQTT per l'attivazione della pompa in modalità manuale e automatica.

4.1.2 Broker MQTT

Come *Broker MQTT* si é pensato di usare un software open source come **Mosquitto**. Esso puó essere scaricato tranquillamente su macchina linux tramite il comando

```
sudo apt install mosquitto
```

e si attiva subito. Esso viene attivato in automatico ogni volta che si riavvia la macchina che lo ospita.

Ai client MQTT che gli si collegano basta dargli indirizzo IP e la porta (1883 di default). Esso puó essere installato in qualsiasi dispositivo interno alla rete. Nel caso specifico di questa sperimentazione é stato installato per comoditá nella Raspberry Pi dello *Smart Garden*.

4.1.3 MQTT Binding del sistema IOT

OpenHAB e lo *Smart Garden* quindi comunicano tramite protocollo MQTT.

In una prima fase é stato implementato uno sviluppo tramite protocollo REST ma dopo alcuni test é risultato essere molto pesante e inadatto per un sistema come questo. Esso infatti, non essendo di tipo publish/subscribe, obbligava openHAB a fare delle chiamate in loop per prelevare i dati. Tale approccio appesantisce di molto la rete nel caso le chiamate vengano fatte molto di frequente mentre nel caso contrario non garantisce un'esperienza real time.

Al contrario con MQTT entrambi i client (openHAB e lo Smart Garden) si sottoscrivono ai topic e notificano tramite questi i vari cambiamenti. I topic sono:

- **handle/pump:** per abilitare (inviando il messaggio “ON”) o disabilitare (inviando il messaggio “OFF”) la pompa dell’acqua manualmente.
- **handle/auto:** per abilitare (inviando il messaggio “ON”) o disabilitare (inviano-
do il messaggio “OFF”) la modalità di innaffiamento automatica. In tal caso,
come visto precedentemente tale attivazione scatena da parte dello *Smart Garden*
una pubblicazione del topic *handle/pump* una mutazione nel caso venga attivata
o no la pompa dell’acqua.
- **status/soil:** per visualizzare i cambiamenti di stato del sensore di umidità del
terreno. Nel caso rileva umidità verrà inviata dal publisher la stringa “**Water
detected**”, mentre “**Water not detected**” altrimenti”.

I client quindi comunicano tra loro per mezzo del broker MQTT *Mosquitto* installato
sulla Raspberry Pi del sistema IOT. I messaggi che vengono inviati da entrambi le parti
sono *retained*, così da mantenere l’ultimo stato inviato.

4.2 Configurazione openHAB

Dopo aver installato il framework tramite il sito ufficiale i primi passaggi per configu-
rarlo sono:

1. far partire il programma tramite l’eseguibile relativo al sistema operativo utili-
zato (windows: `start.bat`, linux/mac: `start.sh`)
2. accedere alla ui web tramite l’url `http://localhost:8080/`
3. creare l’account amministratore inserendo nome utente e password e premendo
su “Crea Account” (Figura 4.7)
4. impostare lingua regione e fuso orario e premere su “Inizia Installazione” (Figura
4.8)
5. impostare la posizione (Figura 4.9)
6. installare Add-on, in questo caso è stato selezionato il binding MQTT. Per fare ciò
basta premere la scritta “Seleziona gli add-on da installare” e selezionare **MQTT
Binding**. Finita tale procedura premere su “Installa 1 add-on” per continuare
(Figura 4.10)
7. premere su “Cominciamo”

4.2.1 MQTT Binding di openHAB

Eseguita la configurazione iniziale ora passiamo a configurare il Binding MQTT per
collegare openHAB allo Smart Garden. I passaggi sono i seguenti:

1. dalla schermata principale premere su “Impostazioni” e successivamente su “Things”
(Figura 4.11)
2. premere sul “+” in basso a destra per scegliere un binding (Figura 4.12)
3. scegliere il “MQTT binding” (Figura 4.13)

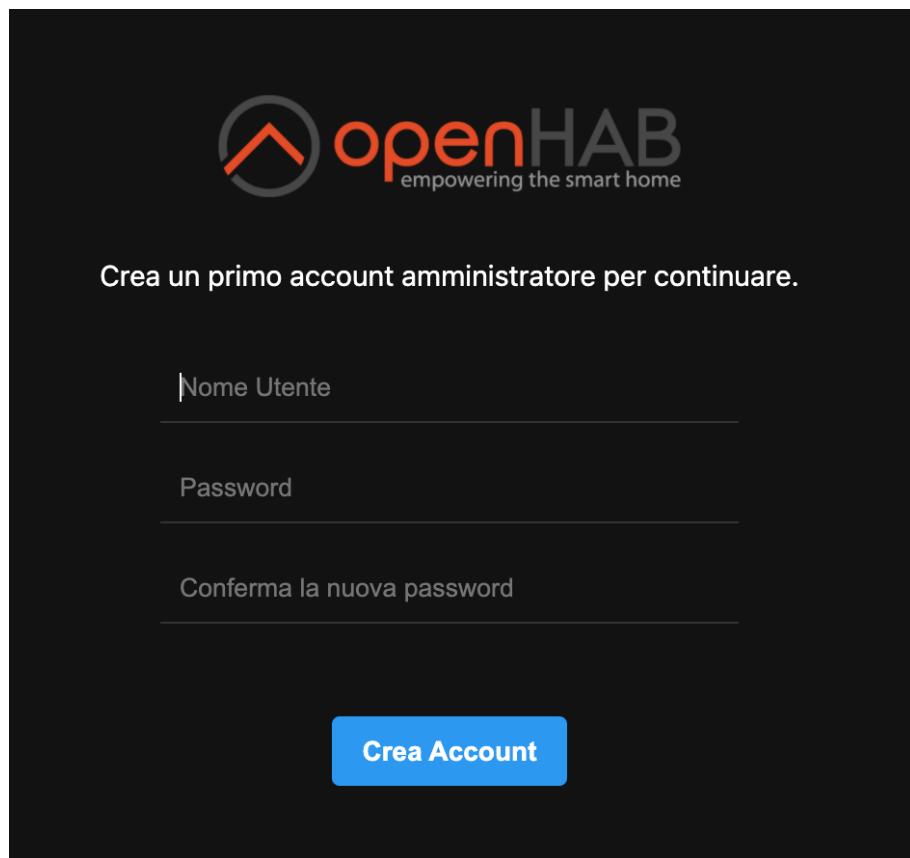


Figura 4.7: Crea Utente Amministratore

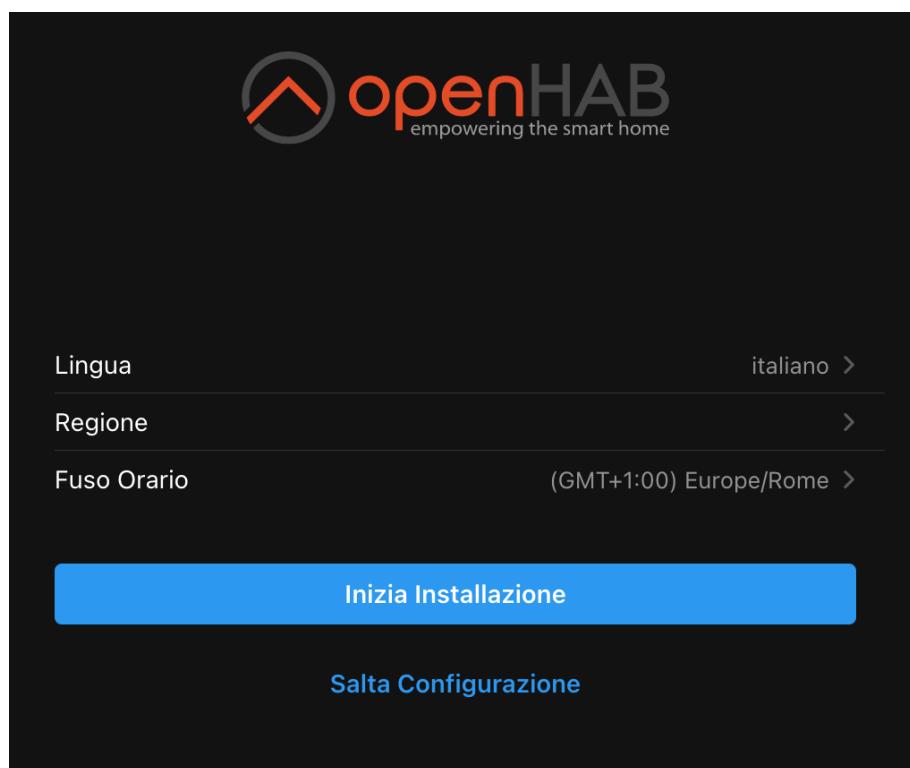


Figura 4.8: Impostazione Lingua Regione e Fuso orario



Figura 4.9: Impostazione Posizione

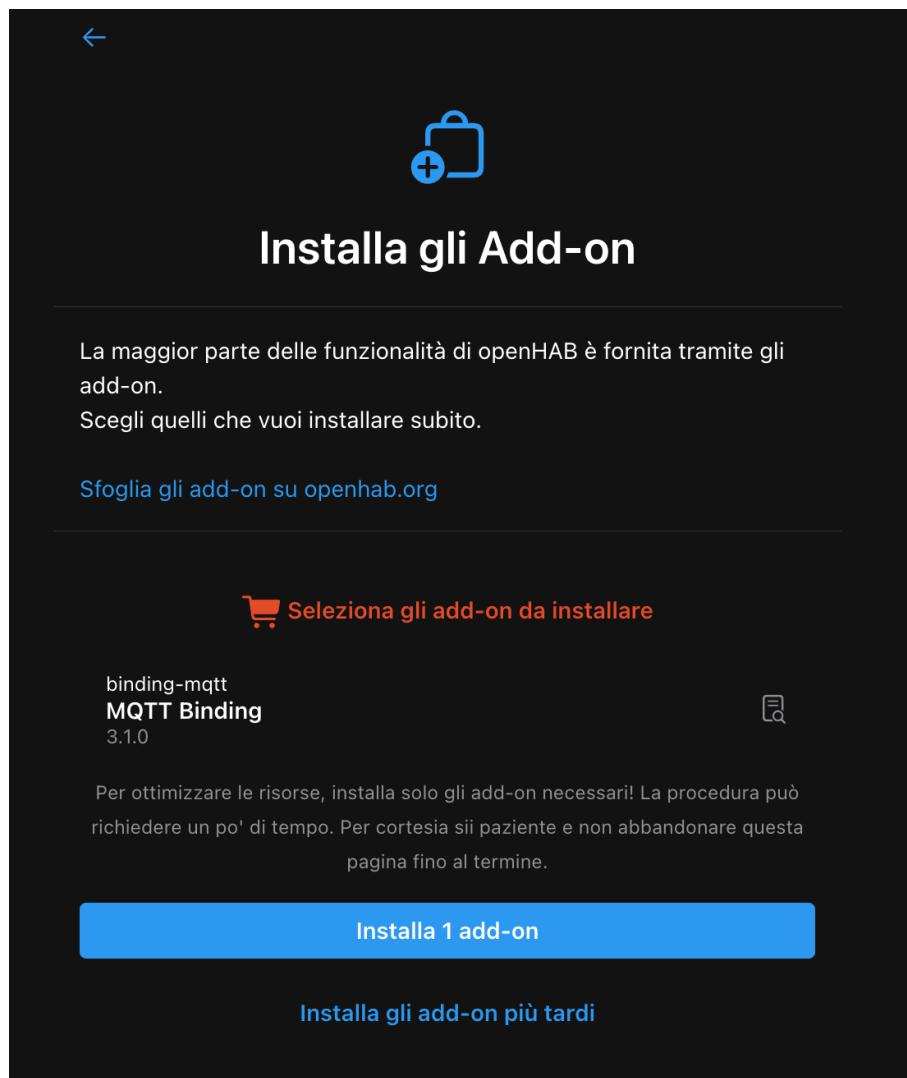


Figura 4.10: Selezionare Add-ons

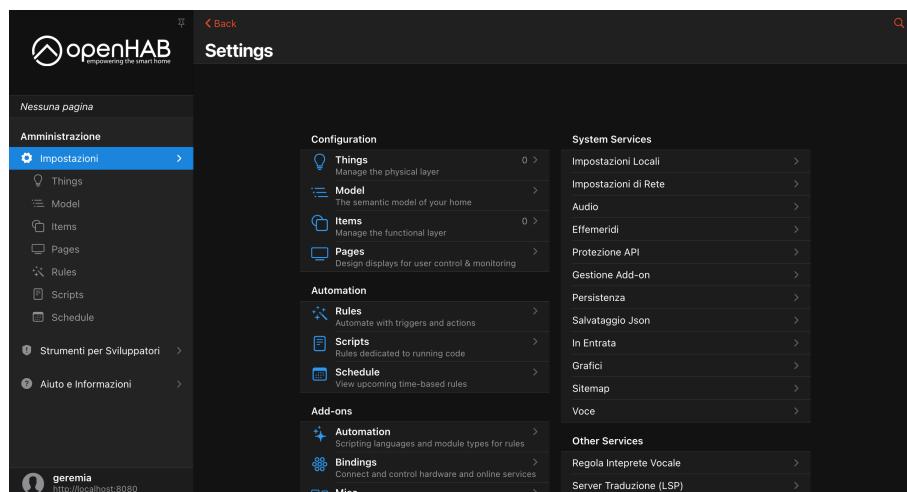


Figura 4.11: Schermata Impostazioni

Configurazione openHAB

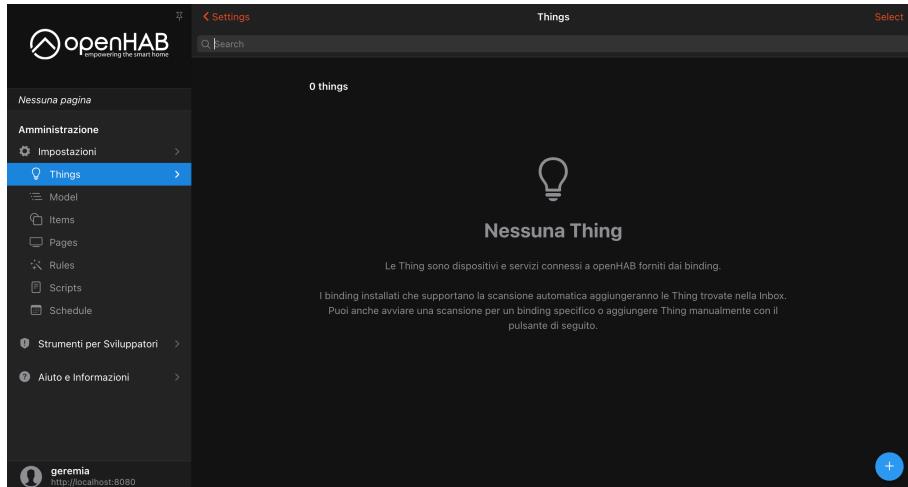


Figura 4.12: Schermata Things

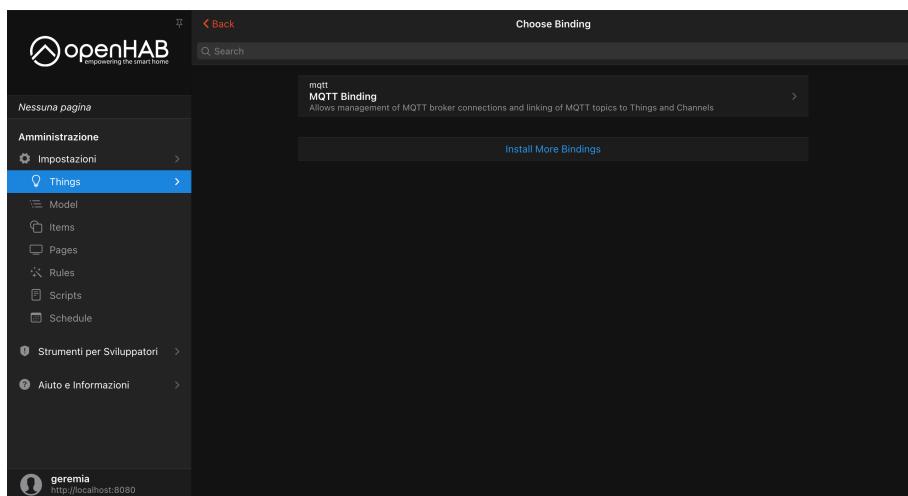


Figura 4.13: Schermata Scelta Binding

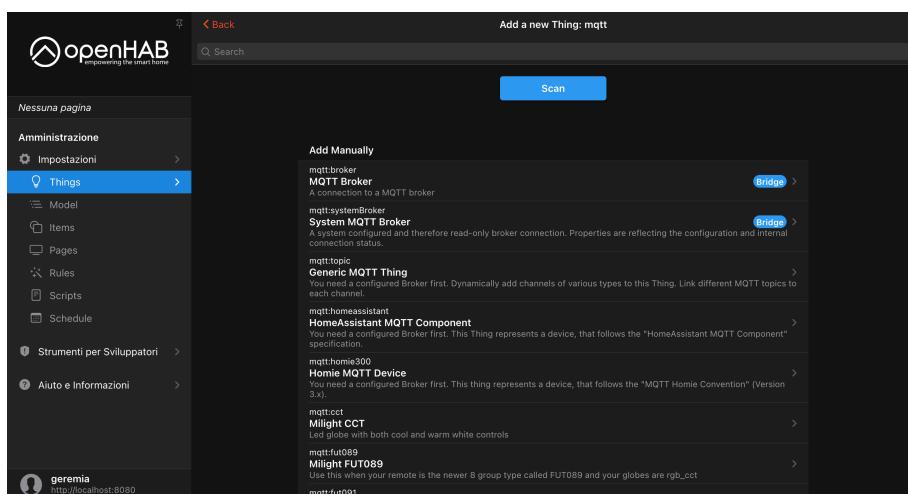


Figura 4.14: Schermata aggiunta Thing

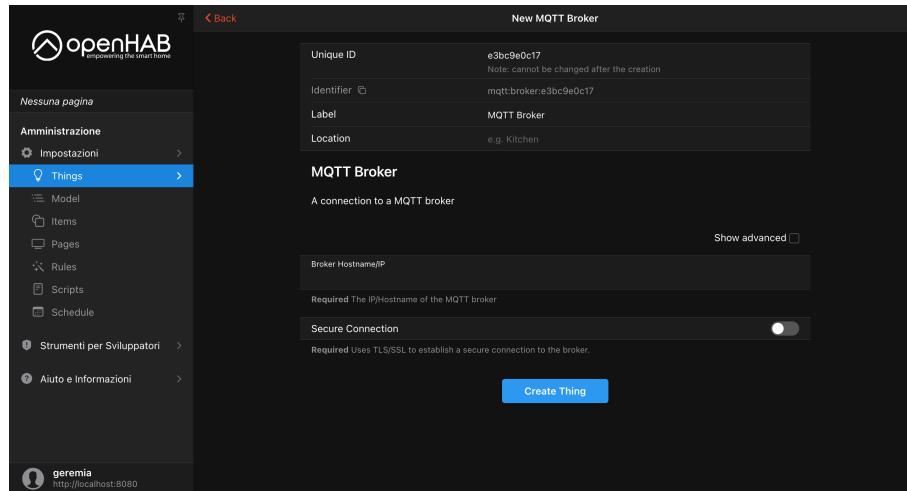


Figura 4.15: Configurazione MQTT Broker

4. aggiungere la thing “MQTT Broker” (Figura 4.14)
5. configurare *MQTT Broker* aggiungendo *Broker Hostname/IP* (IP del broker MQTT) e premendo sul pulsante “Create Thing” (in tal caso verrà aggiunto l’indirizzo IP locale della Raspberry Pi che ha già mosquitto installato). Appena eseguita tale operazione si ritornerà alla schermata precedente con la nuova thing aggiunta. In tale schermata è possibile visualizzare se lo stato del broker MQTT è ONLINE o OFFLINE (Figura 4.15)
6. aggiungere un ulteriore thing rieseguendo i passaggi 2 e 3 e selezionando questa volta la voce “Generic MQTT Thing”
7. si procede alla configurazione aggiungendo i campi:
 - **Label:** rinominarlo con un nome mnemonico così da rendere più semplice la sua ricerca futura (in questo caso “Samrt Garden”)
 - **Location:** aggiungere la posizione nella casa della nuova thing (in questo caso “Garden”)
 - **Bridge:** selezionare “MQTT Broker”, ovvero la thing aggiunta al passaggio 5

premere infine su “Create Thing” (Figura 4.16)

8. ritornati alla schermata Things, a questo punto bisogna aggiungere i Channels relativi ai topic MQTT all’interno della thing appena creata. Quindi premere sulla nuova thing (in questo caso chiamata “Smart Garden”) e successivamente su Channels. Nella schermata della thing è possibile visualizzare il suo status (Figura: 4.17)
9. nella schermata Channels è possibile aggiungere diversi canali relativi alla thing selezionata precedentemente. Per fare ciò basta premere sul pulsante “Add Channel” (Figura 4.18) e configurare il channel da aggiungere inserendo i campi richiesti:
 - **Channel Identifier:** id univoco del channel

Configurazione openHAB

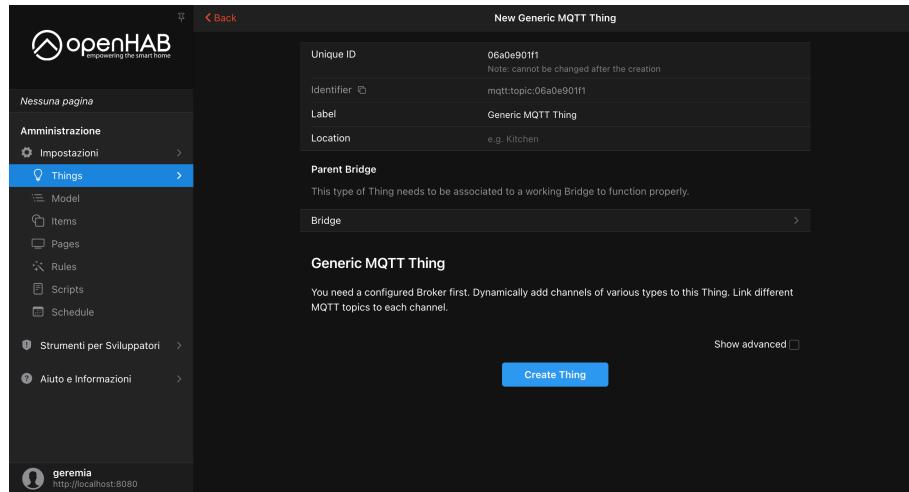


Figura 4.16: Configurazione MQTT Thing

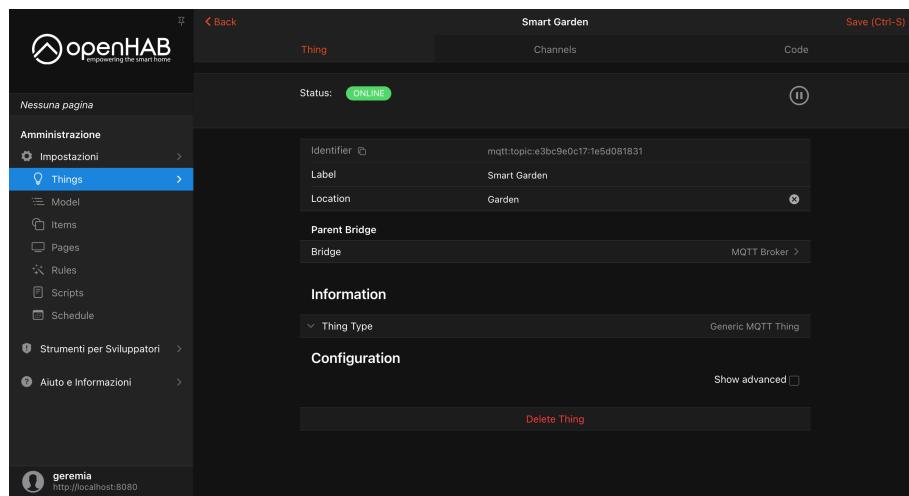


Figura 4.17: MQTT Thing

Identifier	Label	Type	Topic	On	Off
handle_pump	Handle Pump	On/Off Switch	handle/pump	ON	OFF
handle_auto	Handle Auto	On/Off Switch	handle/auto	ON	OFF
status_pump	Status Pump	Text Value	handle/pump	-	-
status_soil	Status Soil	Text Value	status/soil	-	-

Tabella 4.1: Smart Garden CHannels

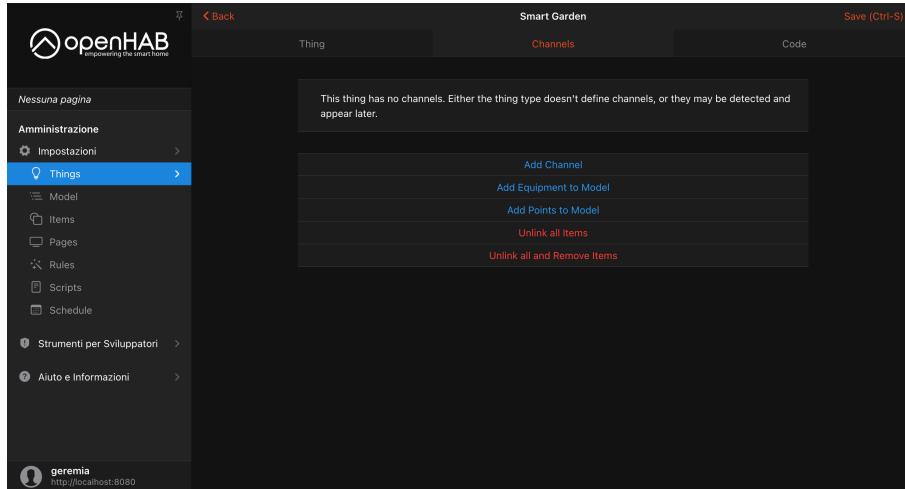


Figura 4.18: Schermata Channels

- **Label:** nome con cui verrá visualizzato a video il canale
- **Channel Type:** tipo del channel. Da questo usciranno fuori poi altri campi relativi alla sua configurazione

In questo caso sono stati aggiunti 4 channels relativi ad ogni topic(handle_pump e status_pump fanno riferimento allo stesso topic in due modi differenti: il primo serve per gestire la pompe in maniera manuale accendendola e spegnendola mentre il secondo per visualizzare il suo stato). Verranno riportati con le relative configurazioni alla Tabella 4.1. In quest’ultima i campi ON e OFF sono relativi solo al tipo “On/Off Switch” poiché corrispondono alle azioni di accensione e spegnimento. In tutti i channels puó essere attivata la voce sotto “Show advanced” chiamata “Retained” per abilitare i messaggi inviati ai relativi topic come mantenuti (Figura 4.19)

10. successivamente per ogni channels dovrebbe essere creato un item. Per fare ciò basta selezionare il channel e premere “Add Link to Item...”. Dalla schermata Channels quindi si passerá poi a quella “Link Channel to Item” in cui bisogna selezionare la voce “Create new Item”. A questo punto basta selezionare una Category (in questo caso “garden”) e premere “Link”. Viene ripetuto tale procedimento per ogni Channel (Figura 4.20)

A questo punto se andiamo alla schermata Items sotto la voce Impostazioni del menu principale possiamo vedere gli Items gi configuriati (Figura 4.21). Se entriamo in uno di essi ´ possibile sia visualizzare lo stato che, in caso di items collegati a channels dinamici come lo Switch, eseguire un azione per cambiarlo.

Configurazione openHAB

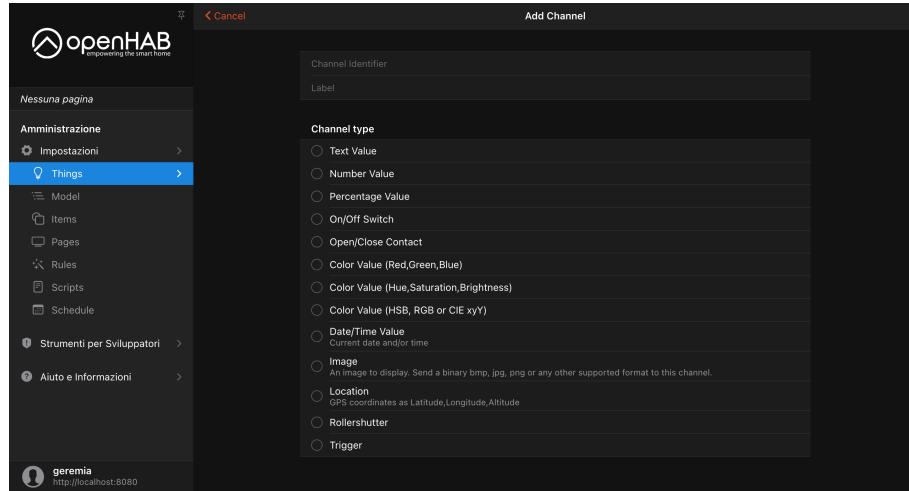


Figura 4.19: Schermata Add Channel

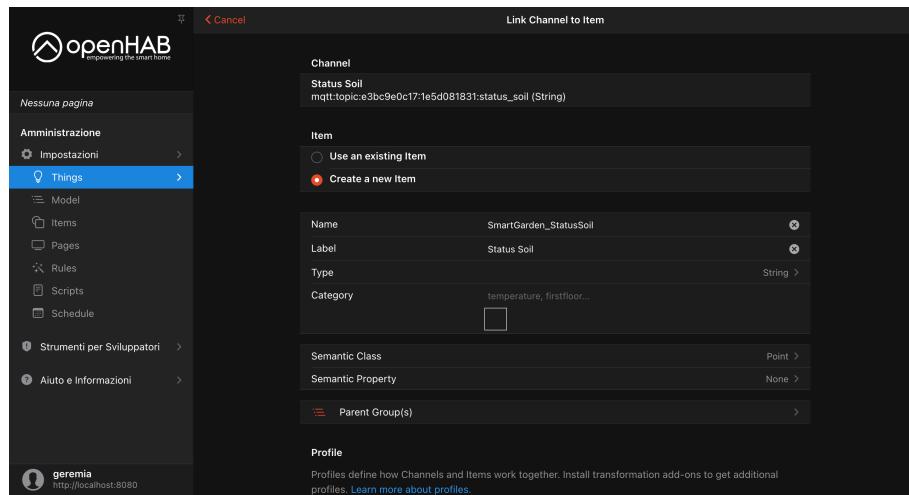


Figura 4.20: Schermata Link Channel to Item

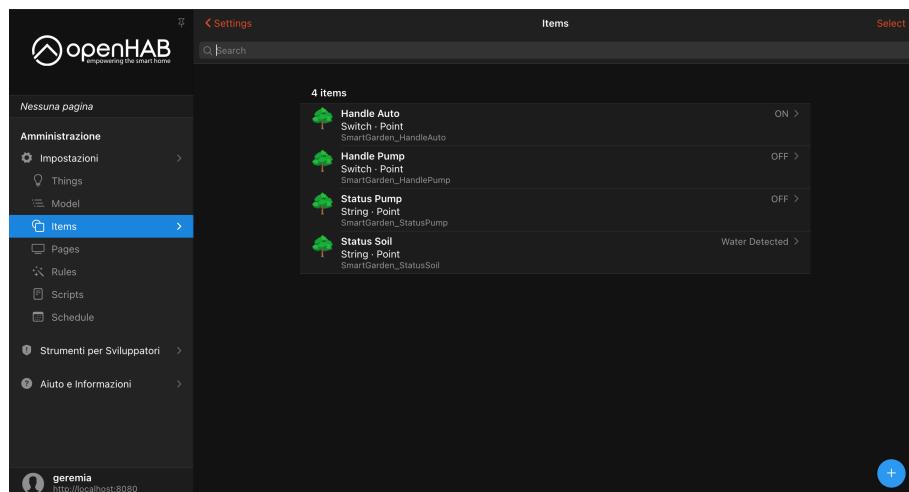


Figura 4.21: Items Configurati

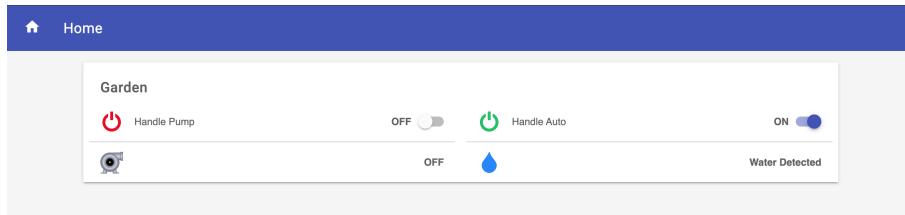


Figura 4.22: Sitemap Smart Garden UI Web

4.2.2 Creazione Sitemap

Per rendere il tutto più agevole e user friendly è possibile riportare tutto in una Sitemap. Per fare ciò bisogna agire all'interno della cartella “config” interna a sua volta alla folder principale del software openHAB. In questa cartella è possibile aggiungere i vari elementi di openHAB come items e things manualmente tramite file. A questo punto bisogna creare un file all'interno della cartella “sitemaps” nominato “defaultsitemap”. Questo è il file della Sitemap che andremo a creare. Il codice da scrivere all'interno sarà relativo agli Items aggiunti, infatti essi possono essere aggiunti tramite il loro identificativo che si trova nella propria schermata in alto. Possono essere aggiunte anche icone per rendere più immediata la schermata. In questo caso facciamo riferimento al Codice 4.5 dove troviamo il Frame “Garden” che contiene gli Items impostati precedentemente con relative icone e label (in caso di text value). È possibile poi visualizzare l'interfaccia web andando all'url <http://localhost:8080/basicui/app>. È possibile inoltre controllare ciò tramite smartphone installando l'applicazione di openHAB e configurandola aggiungendo indirizzo ip e porta del server openHAB interno alla propria rete locale (Figure 4.22 e 4.23).

```

1 sitemap default label="Home"
2 {
3     Frame label="Garden"
4     {
5         Switch item=SmartGarden_HandlePump icon="switch"
6         Switch item=SmartGarden_HandleAuto icon="switch"
7         Text item=SmartGarden_StatusPump label="[%s]" icon="pump"
8         Text item=SmartGarden_StatusSoil label="[%s]" icon="water"
9     }
10 }
```

Codice 4.5: Sitemap Smart Garden Code

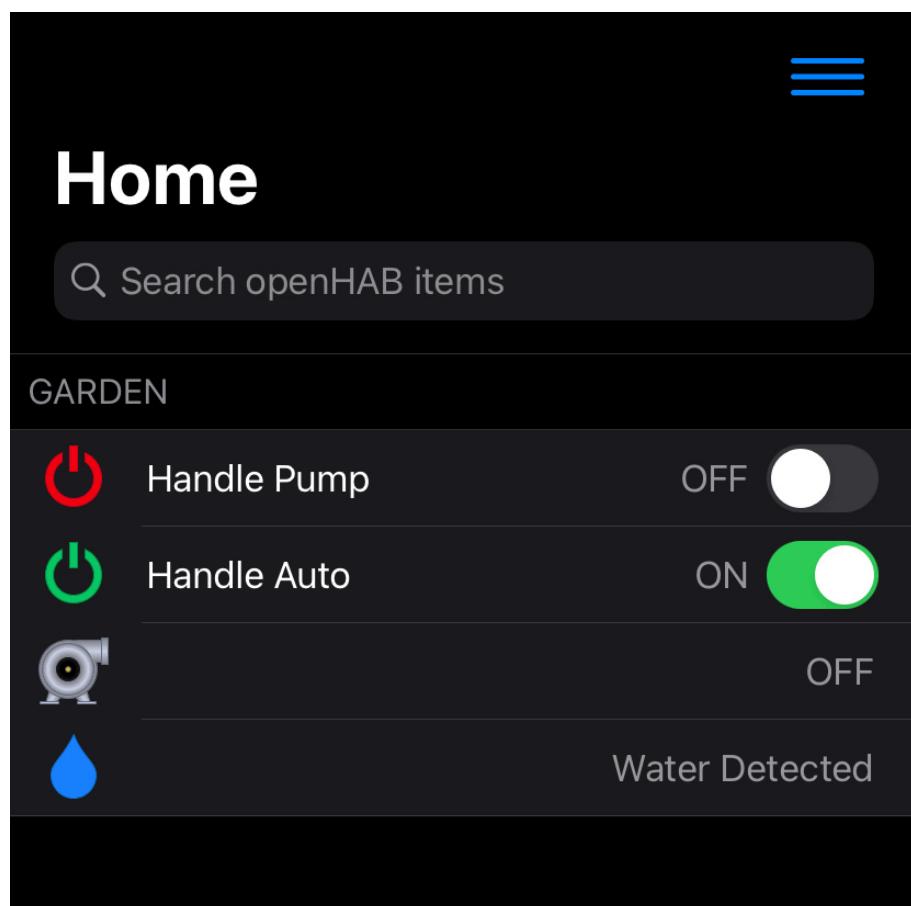


Figura 4.23: Sitemap Smart Garden UI App

5. Conclusioni

Arrivati a questo punto si può confermare la versatilità di *openHAB* al dialogo con qualsiasi tipo di dispositivo. Con lo sviluppo del sistema IOT e il collegamento di esso al software per la domotica si è riuscito a controllare l'irrigazione di un piccolo vaso di basilico. Esso quindi può essere tranquillamente gestito dal computer o dal proprio smartphone abilitando la modalità automatica o accendendo e spegnendo manualmente la pompa dell'acqua. Inoltre è possibile anche visualizzare i vari stati relativi al sensore di umidità e alla pompa dell'acqua. Possono essere visualizzati i risultati del lavoro alle Figure 5.1, 5.2, 5.3, 5.4.

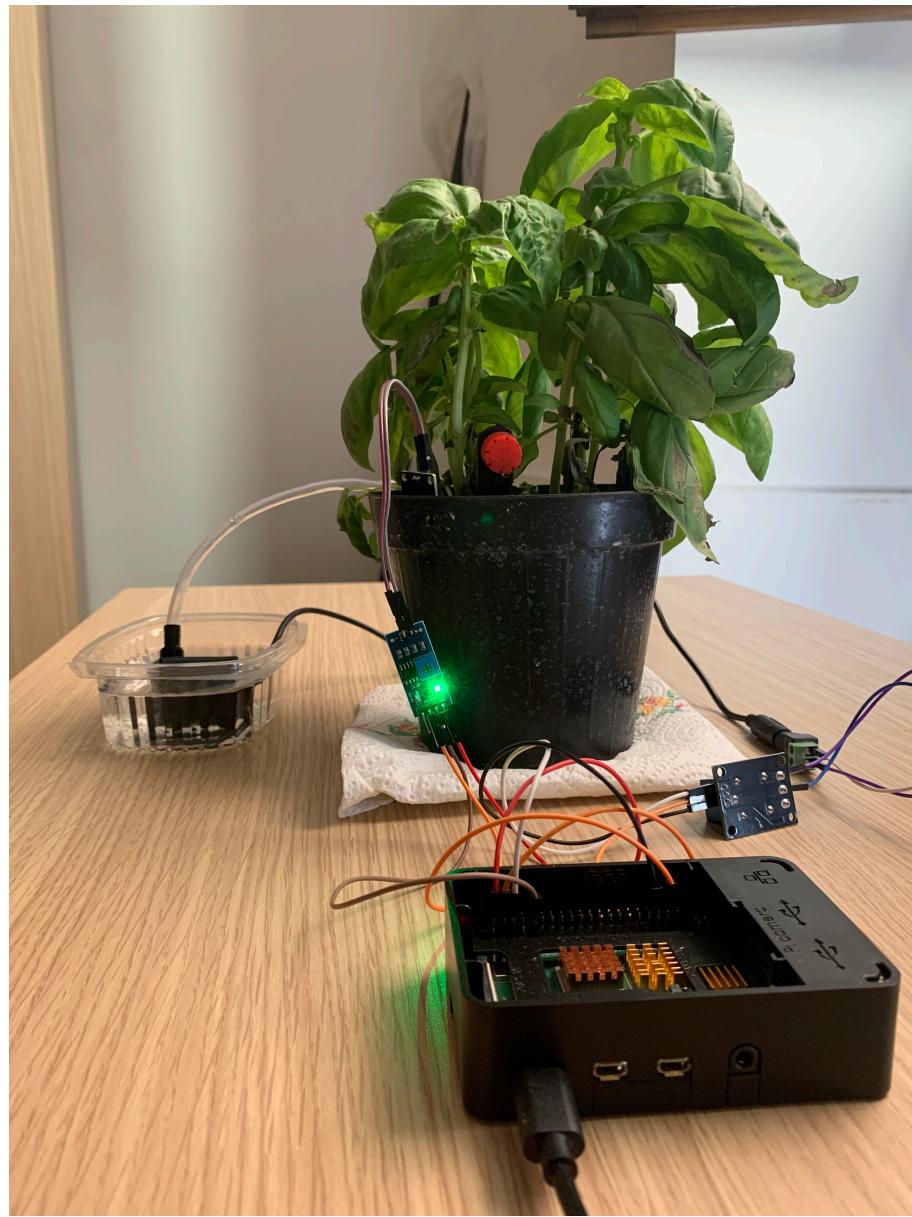


Figura 5.1: Smart Garden di fianco



Figura 5.2: Smart Garden pompa dell'acqua

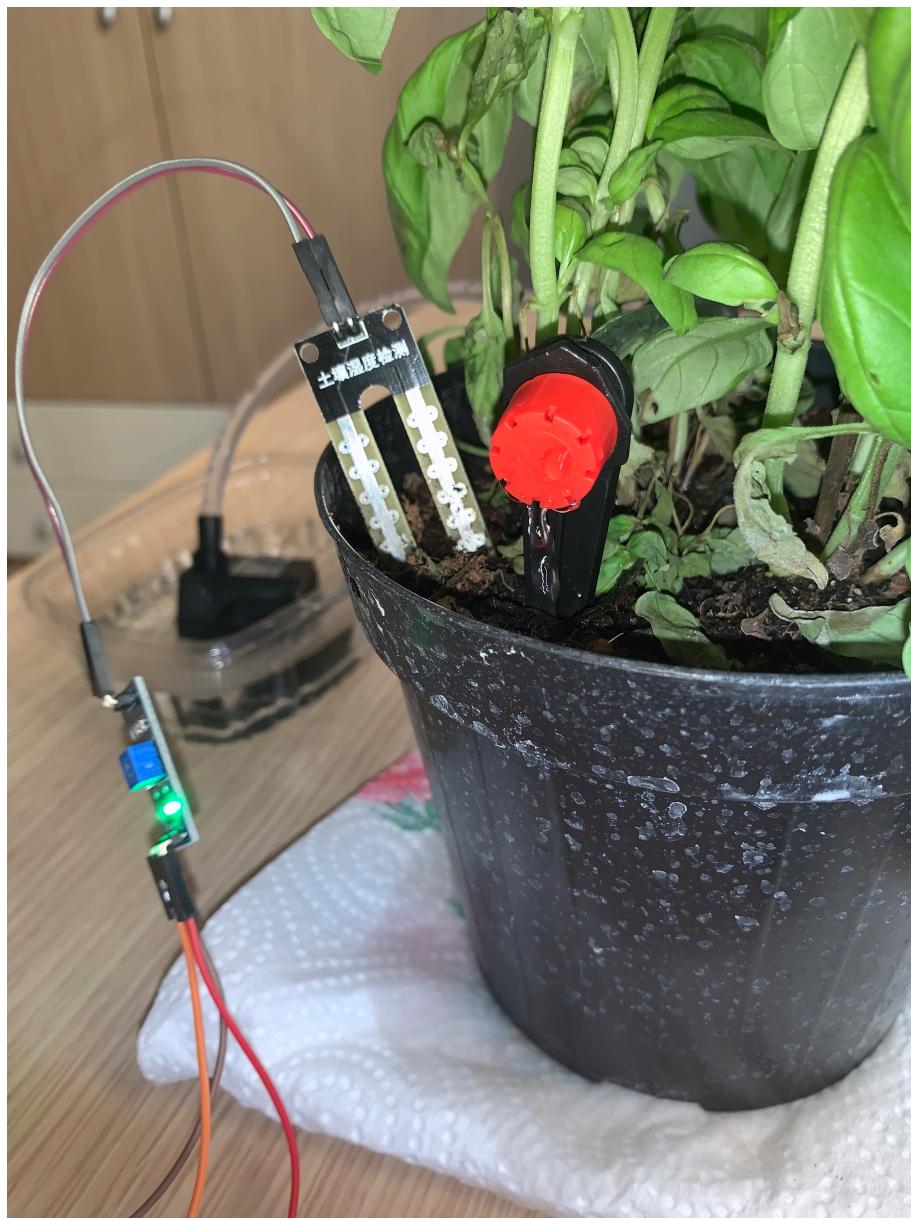


Figura 5.3: Smart Garden sensore di umidità e erogazione dell'acqua

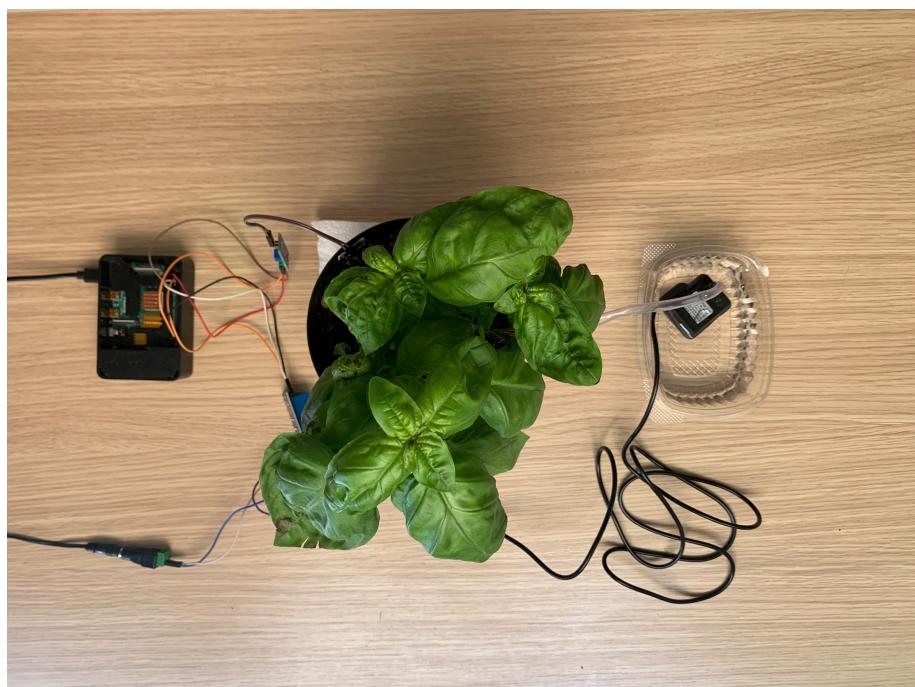


Figura 5.4: Smart Garden dall'alto

Bibliografia

- [Wik21] Wikipedia contributors. *MQTT — Wikipedia, The Free Encyclopedia*. [Online; accessed 3-July-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=MQTT&oldid=1023769301>.

Ringraziamenti

Il mio ringraziamento va al professor Michele Loreti per avermi sostenuto e guidato durante la tesi ed il percorso di formazione universitario e a Ivan Compagnucci, studente Unicam della Magistrale che mi ha aiutato a sviluppare il sistema IOT partendo da una base già sviluppata precedentemente da lui in arduino.