



# SenseBerry

Proyecto centrado en el Internet de las cosas



FRANCISCO JAVIER RUIZ-TOLEDO GALLEGO  
Desarrollo de Aplicaciones Web Curso 2015-2016  
05/06/2016

## Licencia

**Esta obra está bajo una licencia Reconocimiento-Compartir bajo la misma licencia 3.0 España de Creative Commons. Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-sa/3.0/es/> o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.**



Ciclo Formativo Grado Superior  
Desarrollo de Aplicaciones Web  
I.E.S Virgen de la Paz  
Francisco Javier Ruiz-Toledo Gallego  
Alcobendas (Madrid)  
4 de Junio de 2016

# Índice

---

1. Justificación del proyecto y objetivos	3
2. Introducción	4
2.1 Internet Of Things – IoT	4
2.2 MQTT (Message Queue Telemetry Transport)	5
3. Metodología y desarrollo	11
4. Resultados y discusión	29
5. Conclusiones	30
6. Bibliografía	31
7. Anexo	32
7.1 Frameworks	32
7.2 Acceso a Raspberry desde múltiples localizaciones	33
7.3 Salidas por consola de la aplicación	36
7.4 Quality of Service (QoS)	39
7.5 Mensajes MQTT	40
7.6 Sintaxis completa Mosquitto	41
7.7 Glosario de términos	42

## 1. Justificación del proyecto y objetivos

En este documento se expone el trabajo realizado para el módulo de proyecto del CFGS en el ciclo de Desarrollo de Aplicaciones Web.

El proyecto elegido fue “Proyecto centrado en el internet de las cosas ([IOT](#))”. Se escogió este proyecto por la importancia que el internet de las cosas está adquiriendo en el mundo actual, donde internet ya es accesible desde prácticamente cualquier localización y la relación de los estilos de vida con la tecnología es cada vez más estrecha.

En este proyecto se persigue una aproximación al [IOT](#) (Internet Of Things), en el cual se conectará un equipo ([Raspberry Pi](#)) con una serie de sensores incorporados en éste ([SenseHat](#)) y estará constantemente enviando información mediante el protocolo [MQTT](#) (Message Queuing Telemetry Transport), para que cualquier cliente pueda recibir la información enviada, en este caso renderizada en una página web de acceso público.

La importancia del proyecto no es tanto la aplicación en sí, sino el estudio de las tecnologías y protocolos que se usarán para llevarlo a cabo, dado que la utilidad es mucho mayor que la usada en esta ocasión.

El objetivo es saber, en todo momento, las condiciones climáticas (temperatura, presión y humedad) de una localización concreta a través de los métodos comentados anteriormente.

## 2. Introducción

### 2.1 *Internet Of Things – IoT*

Es un concepto donde se conectan objetos cotidianos y de uso diario con Internet, ya sea para su gestión a distancia o para recoger la información que éstos puedan enviar.

El concepto de internet de las cosas lo propuso Kevin Ashton en el Auto-ID Center del MIT en 1999, donde se realizaban investigaciones en el campo de la identificación por radiofrecuencia en red (**RFID** - **R**adio **F**requency **ID**entification) y tecnología de sensores.

En la actualidad, éste concepto es bastante novedoso en cuanto al uso particular, pero en el mundo de la industria se lleva utilizando durante años, donde se puede llevar un estricto control del estado de la maquinaria, dado que se almacena toda la información que las máquinas envían, pudiendo llegar a evitar averías en el futuro que paren la cadena de producción mediante los datos recopilados.

Su uso es prácticamente ilimitado. Es aplicable a cualquier ámbito, desde el **transporte** (un avión que manda información sobre altitud, velocidad, trayectoria, estado de los motores..., un coche que haya sufrido un accidente puede contactar inmediatamente con un equipo de emergencias, se puede saber la localización del autobús/metro/tren a tiempo real...), **sanidad** (hacen posible los diagnósticos precoces de enfermedades que pueden resultar mortales), hasta los objetos más cotidianos, como pueden ser los **electrodomésticos**, **iluminación** de los hogares, **riegos** automatizados...

## 2.2 MQTT (Message Queue Telemetry Transport)

- *Definición*

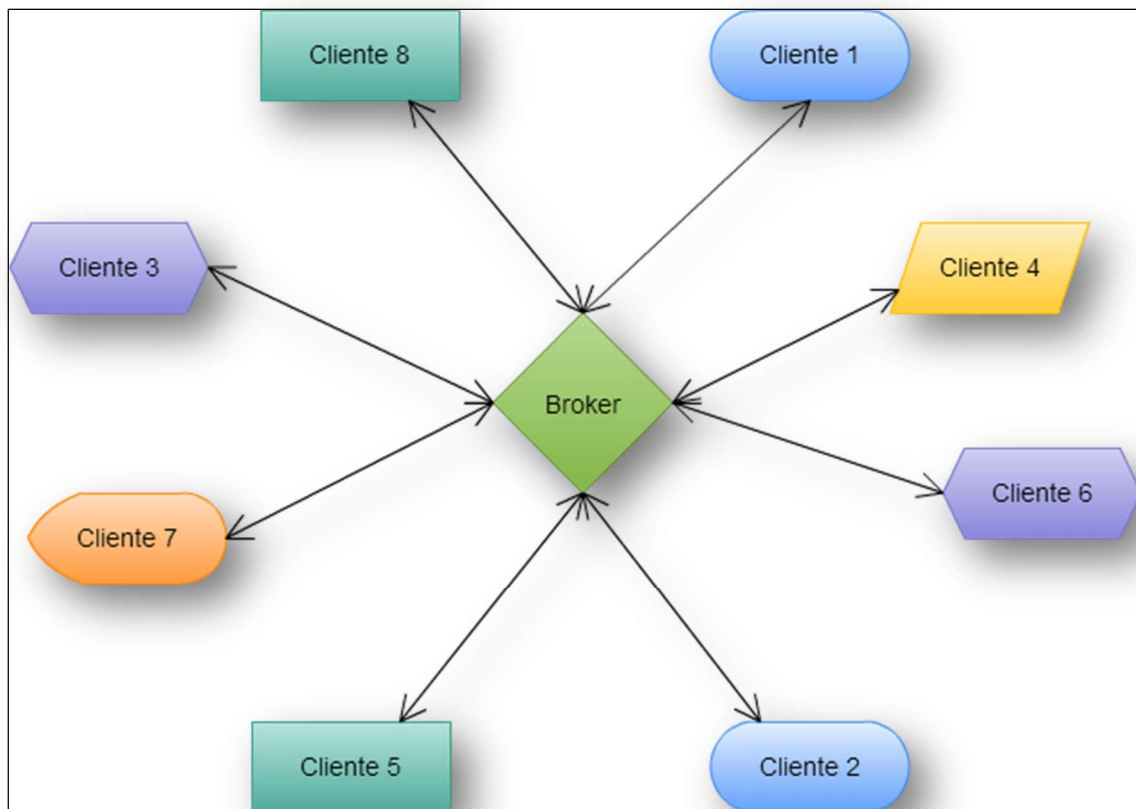
Es un protocolo usado para la comunicación M2M (Machine to Machine) en el Internet de las cosas, debido al poco ancho de banda que utiliza, y puede ser usado en la mayoría de dispositivos con pocos recursos.

Por ejemplo, la aplicación de Facebook Messenger hace uso de esta tecnología.

- *Arquitectura*

La arquitectura de MQTT está construida con topología de estrella, donde el nodo central es el **servidor** o **broker** (ver figura). Es el encargado de distribuir los mensajes entre los clientes conectados. El broker aguanta hasta 10.000 clientes conectados.

Topología MQTT



- *Conexión*

Los mensajes MQTT son enviados a través de **TCP/IP** hasta la versión 3.1.1, donde además de **TCP/IP**, pueden ir a través de **TLS** y/o **Websockets**.

Los puertos **8883** y **1883** de **TCP** han sido registrados por la **IANA** (Internet Assigned Numbers Authority) para las conexiones **TLS** y **NO TLS** de **MQTT**.

- *Mensajes*

Los pasos y mensajes desde que un cliente quiere conectarse hasta su desconexión son los siguientes:

- 1) **CONNECT**: El cliente solicita una conexión al servidor.
- 2) **CONNACK**: El servidor responde reconociendo la conexión.
- 3) **DISCONNECT**: El cliente se desconecta del broker.

Una vez establecida la conexión, los clientes tienen que asegurarse de que ésta sigue activa, y lo realizan de la siguiente manera:

- 1) **PINGREQ**: El cliente manda un paquete periódicamente al servidor.
- 2) **PINGRESP**: El servidor responde al cliente, asegurando que la conexión sigue activa.

Los siguientes no tienen por qué llevar un orden:

- 1) **SUBSCRIBE**: Un cliente solicita una suscripción a un tema o **topic**.
- 2) **SUBACK**: El servidor manda una respuesta reconociendo la suscripción.
- 3) **PUBLISH**: Publicación de un mensaje, puede publicarlo tanto un cliente como el broker.
- 4) **PUBACK**: El que recibe el mensaje (ya sea cliente o servidor), responde reconociendo su recepción (menos QoS 0).

- 5) **UNSUBSCRIBE**: El cliente solicita la cancelación de la suscripción de un **topic** al que previamente estaba suscrito.
- 6) **UNSUBACK**: El servidor manda una respuesta reconociendo la cancelación de la suscripción.

- *Envío de mensajes*

Al realizar una publicación de un mensaje, ésta puede llevar distintos niveles de **QoS** (Quality of Service):

**QoS 0**: El mensaje es enviado una vez, o no se envía. La recepción no es reconocida (no hay **PUBACK**). El mensaje no es guardado, y se perderá si el cliente se desconecta o el servidor falla. Es la manera más rápida de transferencia. Se le suele denominar enviar y olvidar (**fire and forget**).

**QoS 1**: Es el tipo de transferencia por defecto. El mensaje es entregado, al menos, una vez. Si el remitente no recibe el **PUBACK**, el mensaje es enviado de nuevo con un indicador **DUP** (Duplicate) hasta que reciba el **PUBACK**. El receptor puede recibir varias veces el mismo mensaje, y será procesado tantas veces como lo reciba. El mensaje es guardado localmente por el remitente y el receptor hasta que es procesado. El mensaje se borra del receptor una vez es procesado. Si el receptor es el broker, el mensaje es publicado a sus suscriptores, si es el cliente, el mensaje es enviado a la aplicación que los recoge. Una vez el mensaje es borrado, el receptor envía la notificación de recepción al remitente. El mensaje es borrado del remitente cuando recibe esta notificación.

**QoS 2**: El mensaje es enviado sólo una vez. El mensaje se guarda localmente tanto por el remitente como el receptor hasta que es procesado. Es la manera más segura de envío, pero también la más lenta. Necesita, al menos, 2 pares de transmisiones entre el remitente y el receptor antes de borrar el mensaje. El mensaje puede ser procesado por el receptor tras la primera transmisión. En el primer par de transmisiones, el remitente envía el mensaje y obtiene



la notificación del receptor del guardado del mensaje (**PUBREC** - Publish Received), guardando éste último una referencia al paquete (única para cada paquete) hasta que envíe la notificación **PUBCOMP** (Publish Complete), de esta manera se evita procesar el mensaje de nuevo. Una vez ha recibido el mensaje **PUBREC**, el remitente puede descartar el mensaje inicial, dado que ya sabe que el receptor ha obtenido el mensaje correctamente. Si no obtuviera esta notificación, el mensaje es enviado de nuevo con un indicador **DUP** hasta que reciba la notificación.

En el segundo par de transmisiones, y tras haber recibido el **PUBREC**, el remitente notifica al receptor que puede terminar de procesar el mensaje (**PUBREL** - Publish Release). Una vez recibido el **PUBREL**, el receptor descarta todos los datos guardados y responde con un paquete **PUBCOMP**.

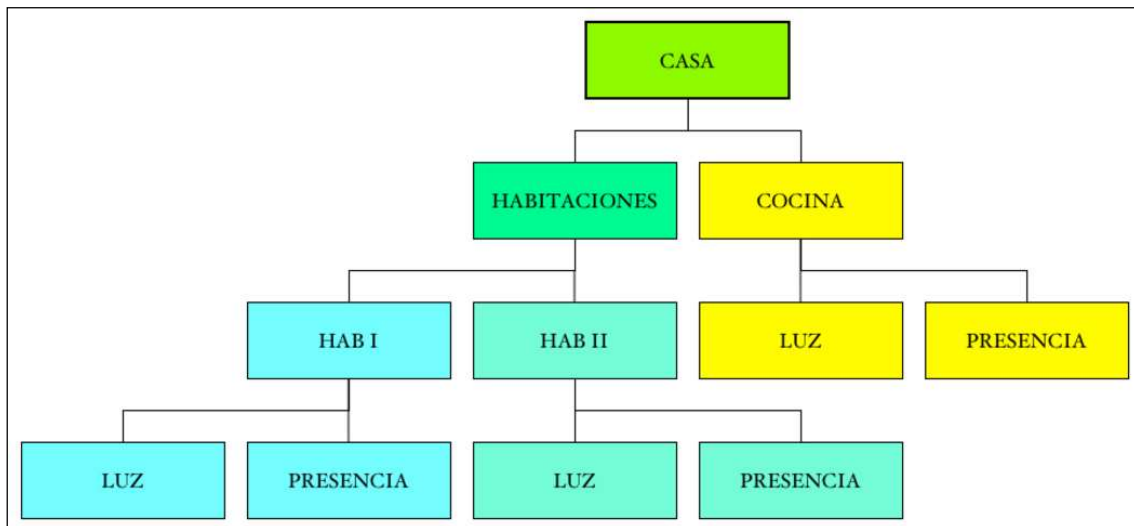
Tras el paquete **PUBCOMP**, la transferencia ha sido finalizada con éxito, y ambas partes tienen conocimiento de ello.

- *Arquitectura MQTT*

➤ TEMAS O TOPICS

Los clientes se conectan a diferentes **temas** o “**topics**”, según la información que quieran recibir. De esta manera, un cliente puede estar conectado a los mensajes de uno o varios **topics** (ver figura).

Ejemplo topics MQTT



Por lo general, los topics se organizan de manera jerárquica, siendo organizados por plantas, salas (en el caso de un edificio) o máquinas y sensores (en el caso de la industria).

En la [figura de ejemplo](#), hay varios sensores (luz y presencia) distribuidos en una casa con 2 habitaciones y una cocina.

Cada uno de los topics se puede organizar de la siguiente manera:

- Hab1

Luz: **casa/habitaciones/hab1/luz**

Presencia: **casa/habitaciones/hab1/presencia**

- Hab2

Luz: **casa/habitaciones/hab2/luz**

Presencia: **casa/habitaciones/hab2/presencia**

- Cocina

Luz: **casa/cocina/luz**

Presencia: **casa/cocina/presencia**

También podemos suscribirnos a varios topics de una sola vez con la siguiente notación:

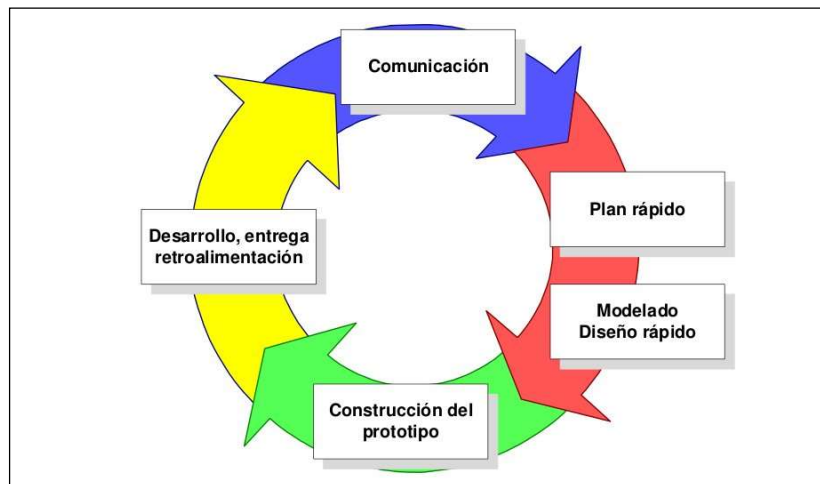
**casa/#** Suscripción a todos los temas de la casa

**casa/cocina/#** Suscripción a todos los temas de la cocina

El '#' es semejante al asterisco, se usa como comodín, y comprende todos los hijos desde el nivel deseado.

### 3. Metodología y desarrollo

Para este proyecto, se ha elegido una metodología de prototipado, dado que no se conocían las tecnologías a usar.



Esta metodología consiste en desarrollar pequeños módulos que vayan presentando la funcionalidad básica de la aplicación, agregando de manera consecutiva pequeñas funcionalidades que vayan completando los objetivos del proyecto.

Por una parte, se empezó investigando el funcionamiento de distintos frameworks (ver anexo) para ver sus ventajas e inconvenientes respecto al proyecto elegido.

Una vez elegido el framework, se investigó el sistema más conveniente y fiel a la idea del **Internet de las cosas**, siendo elegido el envío y recepción de mensajes **MQTT**.

Tras configurar el dispositivo de recogida de datos (**Raspberry Pi**), implementar el envío de datos a través de **MQTT** programado con **Python**.



*Raspberry, Raspbian y Python*

Antes de empezar el proyecto, es necesario configurar la **Raspberry Pi** para poder trabajar con el dispositivo.

El primer paso es instalar un sistema operativo, en este caso **Raspbian**, el cual está basado en el S.O. **Debian Jessie**.

Necesitamos una **tarjeta SD** de, al menos, 2GB, dado que Raspbian ocupa 1.3 GB aproximadamente.

Descargamos [Raspbian](#) desde la página de RaspBerry y extraemos el archivo **.img** para grabar la imagen en la tarjeta **SD**.

Para grabar la imagen, según el sistema operativo usado, se seguirán unos pasos. En este caso, al usar Microsoft Windows, usaremos la herramienta [Win32DiskImager](#).

Una vez instalado **Raspbian**, tenemos ya instalado el software básico que vamos a utilizar, entre ellos un editor de texto y el intérprete de Python.

Será necesario conectar la **RaspBerry** a un monitor (HDMI), y agregar un teclado y un ratón para dejar configurada la IP fija y el acceso desde **SSH** en red local.

El siguiente paso no es necesario, pero por comodidad se configurará el router de casa para tener conexión desde cualquier parte con Internet accesible. Para más detalles, consultar anexo.

Para este Proyecto, se han probado una serie de frameworks para decidir cuál era el más apropiado.

Uno de los más completos era Django, pero no ofrecía cambios a tiempo real, sino que había que recargar el documento para mostrar los nuevos datos.



Al final se optó por Meteor, un framework de desarrollo en **NodeJS** con reactividad en los documentos, es decir, cada vez que llegan datos nuevos se renderizan en pantalla de manera inmediata. Al estar montado con NodeJS, todo el código va en Javascript, y es accesible tanto desde el cliente como desde el servidor. Usa como base de datos **MongoDB**, facilitando la inserción y búsqueda de datos desde la aplicación.

Usa, como sistema reactivo en las vistas, **Blaze**, sistema que ya conocíamos de otros proyectos realizados con **Laravel**.

Lo primero es descargar el paquete de meteor. Navegamos hasta la página oficial de **Meteor**, en la sección “Install”:

<https://www.meteor.com/install>

Como la aplicación va a ser desarrollada en Linux, nos vale con poner el siguiente comando:

```
$ > curl https://install.meteor.com/ | sh
```

Una vez instalado el paquete, podemos crear una aplicación con el comando

```
$ > meteor create <nombre aplicación>
```

Cuando se haya creado, tendremos la siguiente estructura de carpetas y archivos:

<b>client/</b>	# código del cliente. HTML, JS y CSS irían aquí.
<b>server/</b>	# código del servidor, que el cliente no puede ver ni ejecutar.
<b>imports/</b>	# todo lo incluido no se carga en ningún sitio, hace falta importarlo.
<b>package.json</b>	# fichero necesario para la descarga de dependencias del proyecto.
<b>.meteor</b>	# ficheros arquitectura Meteor.

También nos instalará los siguientes módulos por defecto:

```
jrt@jrtCasa:~/sensor/sensores$ meteor list
autopublish          1.0.7 (For prototyping only) Publish the entire database to all clients
blaze-html-templates 1.0.4 Compile HTML templates into reactive UI with Meteor Blaze
ecmascript            0.4.3 Compiler plugin that supports ES2015+ in all .js files
es5-shim              4.5.10 Shims and polyfills to improve ECMAScript 5 support
insecure             1.0.7 (For prototyping only) Allow all database writes from the client
jquery               1.11.8 Manipulate the DOM using CSS selectors
meteor-base          1.0.4 Packages that every Meteor app needs
mobile-experience    1.0.4 Packages for a great mobile user experience
mongo                1.1.7 Adaptor for using MongoDB and Minimongo over DDP
perak:mqtt-collection 1.0.4 IoT for Meteor - send/receive MQTT messages via collections
reactive-var         1.0.9 Reactive variable
standard-minifier-css 1.0.6 Standard css minifier used with Meteor apps by default.
standard-minifier-js  1.0.6 Standard javascript minifiers used with Meteor apps by default.
tracker              1.0.13 Dependency tracker to allow reactive callbacks
```

No es necesario instalar **NodeJS** ni **NPM**, se instalan como dependencia del proyecto.

Ya podemos navegar hasta la raíz de nuestro proyecto

```
$ > cd <nombre aplicación>
```

Una vez dentro del proyecto, podemos lanzar la aplicación de prueba que viene hecha:

```
$ > meteor
```

Cargará todos los módulos necesarios e iniciará la parte del servidor, cuando acabe lanza el siguiente mensaje:

```
jrt@jrtCasa:~/mqttPrueba/Sensores$ meteor
[[[[[ ~/mqttPrueba/Sensores ]]]]]

=> Started proxy.
=> Started MongoDB.
=> Started your app.

=> App running at: http://localhost:3000/
```

Si introducimos la dirección en un navegador, ya podremos interactuar con la aplicación que Meteor dispone al instalarlo, que no es más que un botón con la función de incrementar un contador que se renderiza en pantalla.



Para entender mejor el funcionamiento de este framework, se siguió el tutorial disponible en <https://www.meteor.com/tutorials/blaze/creating-an-app>, donde se crea una aplicación que permite introducir tareas que se guardan en base de datos (**MongoDB**) y se renderizan automáticamente en la página, siendo disponibles para todos los usuarios conectados. Trae la lógica de login de usuarios y privacidad de mensajes, pero no será necesario en el proyecto que vamos a realizar.

Los cambios en el cliente son inmediatamente desplegados sin necesidad de reinicio del servidor. Por otro lado, los cambios en el código de servidor requieren un reinicio de servidor, acción que se realiza automáticamente.

Una vez comprobado el funcionamiento del framework (**Meteor**), podemos empezar a investigar el funcionamiento de los mensajes **MQTT**.



La primera prueba fue realizada con **Mosquitto**, un broker de mensajes de código abierto para el sistema operativo (**Linux**).

Lo primero, instalar el cliente (como administrador):

```
$ > sudo apt-get install mosquitto
```

Si el anterior comando no funcionase, habría que descargar la versión elegida desde la página de descargas de **Mosquitto**.



Para recibir los mensajes en otro equipo mediante **Mosquitto**, es necesario instalar el software de nuevo.

Una vez instalado en ambos dispositivos, ya podemos empezar a enviar desde un equipo y recibir en otro con los siguientes comandos:

**Para enviar:**

```
$ > mosquitto_pub [-h IP DEL EQUIPO] [-p PUERTO] -t TEMA -m MENSAJE
```

```
jrt@jrtCasa:~/mqttPrueba/Sensores$ mosquitto_pub -t "Sensores" -m "mensaje temperatura"
```

```
jrt@jrtCasa:~/mqttPrueba/Sensores$ mosquitto_pub -t "Sensores" -m "prueba envío de mensajes mqtt"
```

Como este ejemplo se ejecuta de manera local, no hace falta poner dirección de host ni puerto.

Si abrimos otra consola de manera simultánea e introducimos el comando de suscripción al tema indicado (“**Sensores**”), podremos recibir todo el tráfico de mensajes (ver suscripción).

**Para recibir:**

```
$ > mosquitto_sub [-h IP DEL EQUIPO] [-p PUERTO] -t TEMA
```

```
jrt@jrtCasa:~$ mosquitto_sub -t "Sensores"
mensaje temperatura
prueba envío de mensajes mqtt
```

Para ver los mensajes con todas sus opciones, consultar [anexo](#).


Tras lograr el envío en red local, se probará el envío de nuevo, pero esta vez a través del broker.





El broker elegido fue [CloudMQTT](#), ya que permite un plan gratuito de hasta 10 conexiones simultáneas y ofrece muchas facilidades para la conexión.


Tras dar de alta la cuenta, nos dará los datos para conectarse al broker:


CloudMQTT Console

 Overview

 Websocket UI

 Server log

 Statistics

 Restart


Instance info

Server	m21.cloudmqtt.com
User	lkvezrhe
Password	Qx5YthbMM3RP
Port	13438
SSL Port	23438
Websockets Port (TLS only)	33438
Connection limit	10

Una vez tengamos la cuenta configurada, ya podemos usar el broker para empezar a enviar y recibir mensajes.



Para la conexión con el broker y las primeras pruebas, se usará el cliente websocket [HiveMQ](#).

 **HiveMQ** Websockets Client Showcase

**Connection** ⌵

Host  
broker.mqttdashboard.com

Port  
8000

ClientID  
clientId-qkT1jN6eY

Connect

Username

Password

Keep Alive  
60

SSL  
☐

Clean Session  
☒

Last-Will Topic

Last-Will QoS  
0

Last-Will Retain  
☐

Last-Will Message

Publish ⌵


Subscriptions ⌵

Messages ⌵

Para probar el funcionamiento, podemos publicar los mensajes con **Mosquitto**, enviando los mensajes desde éste al broker.

Ponemos los datos que **CloudMQTT** nos proporciona y enviamos un mensaje, confirmando que los mensajes llegan hasta el broker:

```
mosquitto_pub -h m21.cloudmqtt.com -p 13438 -u lkvezh -P Qx5YthbMM3RP -t "Sensores" -m "prueba envío de mensajes mqtt"
```

 **HiveMQ** Websockets Client Showcase

**Connection** ⌵ connected

**Publish** ⌵

Topic  
testtopic/1

QoS  
0

Retain  
☐

Publish

Message

**Subscriptions** ⌵

Add New Topic Subscription

Qos: 2

Sensores

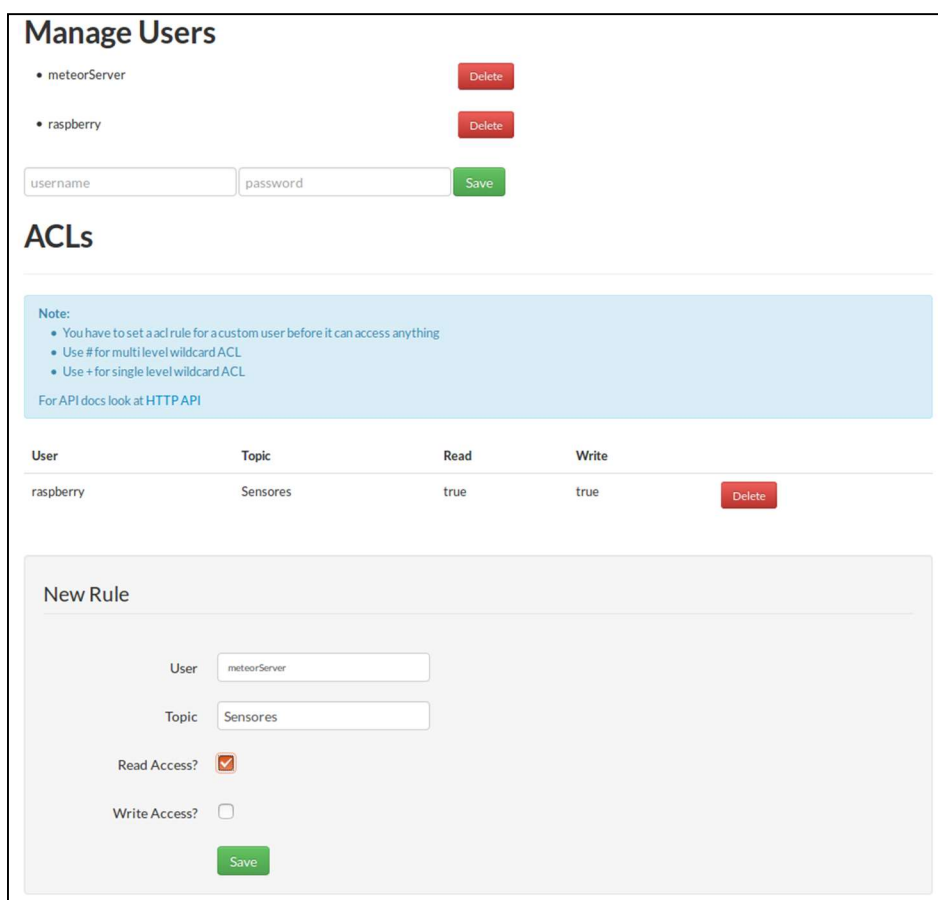
**Messages** ⌵

2016-06-04 16:57:13 Topic: Sensores Qos: 0

prueba envío de mensajes mqtt

Para esto, hemos introducido los datos que **CloudMQTT** nos proporcionó, y podemos tener el cliente Websocket **HiveMQ** a la escucha de los mensajes recibidos. Tan sólo hay que agregar la suscripción al tema “Sensores”, y podemos ver cómo recibe todo lo que enviamos desde nuestro dispositivo.

Podemos crear usuarios en nuestro broker con permisos específicos para cada tema, en este caso crearemos un cliente para la RaspBerry con permiso de escritura sobre el tema Sensores, y otro para el servidor de Meteor con permisos de lectura.

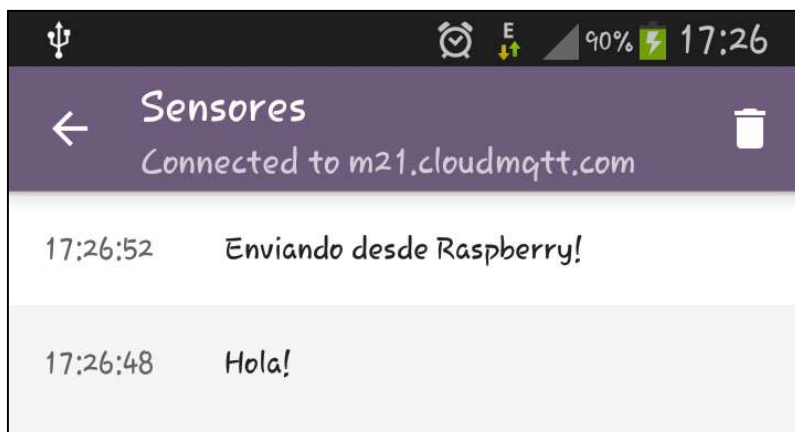


The screenshot shows the 'Manage Users' interface of CloudMQTT. It lists two users: 'meteorServer' and 'raspberry', each with a 'Delete' button. Below this is a form to add a new user with fields for 'username' and 'password', and a 'Save' button. The 'ACLs' section contains a note about setting ACL rules and a table showing the permissions for the 'raspberry' user on the 'Sensores' topic. The table indicates 'true' for both 'Read' and 'Write' permissions. Below the table is a 'New Rule' form with fields for 'User' (set to 'meteorServer') and 'Topic' (set to 'Sensores'). It also has checkboxes for 'Read Access?' (checked) and 'Write Access?' (unchecked), and a 'Save' button.

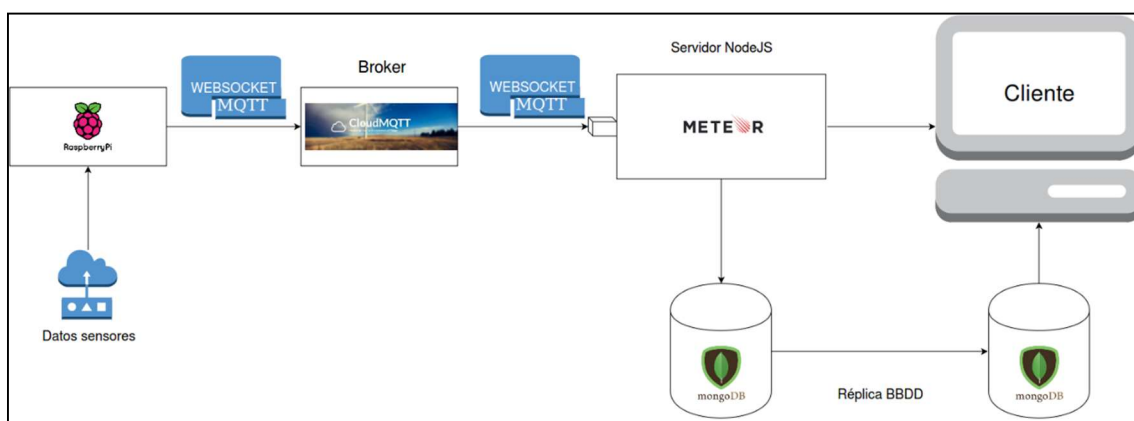
User	Topic	Read	Write
raspberry	Sensores	true	true

Una vez conseguido el envío de mensajes entre máquinas (red local e internet), el siguiente paso es tener un broker online que reciba los datos desde la **RaspBerry** y la envíe a todos los clientes conectados en ese momento.

```
$ mosquitto_pub -h m21.cloudmqtt.com -p 13438 -t "Sensores" -m "Hola!" -u lkvezrhe -P Qx5YthbMM3RP
$ mosquitto_pub -h m21.cloudmqtt.com -p 13438 -t "Sensores" -m "Enviando desde Rasperry!" -u lkvezrhe -P Qx5YthbMM3RP
```



Como ya conocemos el framework, y hemos visto el funcionamiento de mensajes **MQTT**, podemos realizar el boceto de la arquitectura de nuestra aplicación.



Como vemos, la Raspberry recoge los datos de sus sensores, y los envía al broker a través de **MQTT**.

El broker recoge estos datos, y los pone a disposición del servidor una vez éste último arranque.

Se ha decidido que sea el servidor el que recoja los datos debido a, primero, la limitación de 10 clientes que **CloudMQTT** nos impone, y segundo, para hacer uso de la reactividad que **Meteor** nos ofrece.

En esta primera versión, cada dato que llega al servidor se introduce en **MongoDB**, donde luego se accederá para mostrar los datos en las vistas. En futuras revisiones, no todos los datos serán almacenados en la base de datos, sino que se guardarán en sesión, y solo se introducirían a base de datos una vez cada hora, siguiendo la hora del servidor.

La base de datos “principal” estará en el servidor, pero nos valdremos del paquete **autopublish** incluido en **Meteor** para replicar esta base de datos en el cliente. Cualquier cambio que se haga sobre una, se realizará también sobre la otra (en nuestro caso la única modificada será la del servidor **Meteor**).

Una vez comprobado el correcto funcionamiento del broker, y teniendo clara la estructura y flujo de datos de nuestra aplicación, se puede empezar a hacer pruebas con el servidor **Meteor**.

Usaremos el paquete de **NPM MQTT**. Para hacer uso de éste, necesitamos importarlo en el servidor con el siguiente comando:

```
$ > meteor npm install mqtt
```

En el archivo `server/main.js` introduciremos la siguiente línea:

```
var mqtt = require('mqtt');
```

De esta manera, podremos hacer uso de la funcionalidad que `mqtt` nos proporciona, pudiendo así conectarnos al broker cuando el servidor sea iniciado.

En el archivo `server/main.js`, dentro de la función `Meteor.startUp()`, iniciaremos la lógica de conexión y suscripción al tema “Sensores”, que es donde la Raspberry mandará los datos recogidos por sus sensores.

```
var client = mqtt.connect('mqtt://m21.cloudmqtt.com', {  
  port: 13438,  
  clientId: "clientId-3ljf14u1aSasd",  
  username: "meteorServer",  
  password: "meteorserverpass",  
  clean: false,  
  insert: true,  
  sync: true,  
  readable: true  
});
```

La sintaxis del método de conexión es la siguiente:

```
mqtt.connect( host [, {opciones}]);
```

Recogeremos también cualquier error con la función:

```
client.on('error', function (err) {  
  console.log(err);  
});
```

Cualquier error que se produzca durante el intento de conexión con el broker, imprimirá su traza en la consola.

Para comprobar si se ha conectado, podemos sacar por la consola el contenido de la variable client, poniendo tras la anterior función:

```
console.log(client);
```

Vemos que nos da información sobre la conexión MQTT ([ver anexo](#)).

Una vez veamos que conecta, podemos establecer la suscripción al tema que recoge los datos de los sensores:

```
client.on('connect', function () {  
  client.subscribe("Sensores");  
});
```

Con la suscripción realizada, podemos recoger los mensajes enviados, y para comprobar que los datos son correctos, los escribiremos en la consola:

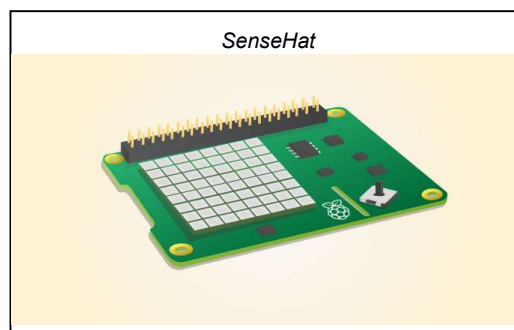
```
client.on("message", function (topic, message) {  
  console.log(JSON.parse(message.toString().replaceAll('"', '\\\"')));  
  //insertMessage(JSON.parse(message.toString().replaceAll('"', '\\\"')));  
});
```

Siendo la salida de la consola la siguiente:

```
120160604-18:48:30.540(2)? { temperature: 32, pressure: 932, humidity: 38 }
```

Podemos comprobar que el servidor ya se conecta al **broker**, y recoge los mensajes que el broker distribuye. Con esta funcionalidad acabada, podemos montar el script que lanzará la **Raspberry** con los datos de sus sensores.

Los sensores de la Raspberry son un accesorio de esta ([SenseHat](#)), y sus datos se recogen por **Python**. Tras un vistazo a la sintaxis de **Python** (en cualquier web, dado que el código a usar es muy básico), en la página oficial de Raspberry hay un pequeño [manual](#) de uso, y una referencia a la [API](#) completa.



Siguiendo este manual, podemos programar el script que mandará los datos de los sensores al broker:

Primero, hay que importar los paquetes que usaremos en el script

```
import paho.mqtt.publish as pub
from sense_hat import SenseHat
import time
```

En este script, vemos que en lugar de **MQTT.js** o **Mosquitto**, hemos usado [Paho](#) para mandar los mensajes al broker. Da igual cual usemos, siempre y cuando podamos comunicarnos con el broker y recoja los mensajes enviados.

Importamos también la API de SenseHat para sacar los datos de los sensores, e importamos time para poner un tiempo de demora entre mensaje y mensaje.

```
sense = SenseHat()
print("Sending data...")
```



Guardamos la referencia a la api de SenseHat como sense, y cuando se lance el script que escriba en consola “Sending data...”, para saber que el script está activo.

Recogemos los datos de cada sensor, formateando la salida para que ofrezca 1 decimal para cada dato, y realizamos la conexión al broker publicando toda la información.

La sintaxis de publicación con **Paho** es la siguiente:

```
paho.mqtt.publish(pub).single(“Topic”, “Mensaje”  
                                [, Host, Port  
                                [, auth = {User, Pass}]  
                                [, Id-Cliente]  
                                [, Tiempo de conexión]  
                                ])
```

```
while True:  
    data = "{ 'temperature': " + str(round(sense.get_temperature(),1)) +  
            ", 'pressure': " + str(round(sense.get_pressure(),1)) +  
            ", 'humidity': " + str(round(sense.get_humidity(),1)) + "}"  
    pub.single("Sensores", data, hostname="m21.cloudmqtt.com",  
              port=13438, auth = {'username':"raspberrypi", 'password':"raspberrypipass"},  
              client_id="clientId-RASPIuLaS", keepalive=600)  
  
    time.sleep(1)
```

*(La variable data será una sola línea, se ha estructurado en bloque para facilitar la lectura)*

Toda la lógica de recopilación de datos la envolvemos en un bucle infinito, que es repetido cada segundo (**time.sleep(1)**), de esta manera, la Raspberry estará constantemente enviando los datos de sus sensores.

Los mensajes son Strings (cadenas de texto), pero organizados de tal manera que se puedan traducir a JSON una vez lleguen al servidor.

Comprobamos, ya sin necesidad de mirar el **broker**, sino la salida a la consola de nuestro servidor **Meteor** ([mirar salida mensajes por consola](#)).

Lanzamos el script de la **Raspberry**:

```
pi@raspberrypi:~/mqtt$ python publish.py  
Sending data...
```

```
I20160604-19:31:56.629(2)? { temperature: 35.9, pressure: 0, humidity: 28.1 }  
I20160604-19:31:57.836(2)? { temperature: 36, pressure: 938.3, humidity: 28.8 }  
I20160604-19:31:58.913(2)? { temperature: 35.9, pressure: 938.3, humidity: 28.9 }  
I20160604-19:32:00.076(2)? { temperature: 35.8, pressure: 938.3, humidity: 28.5 }  
I20160604-19:32:01.170(2)? { temperature: 36, pressure: 938.3, humidity: 28.6 }  
I20160604-19:32:02.391(2)? { temperature: 35.9, pressure: 938.3, humidity: 28.7 }  
I20160604-19:32:03.546(2)? { temperature: 35.9, pressure: 938.3, humidity: 29.3 }  
I20160604-19:32:04.681(2)? { temperature: 35.9, pressure: 938.3, humidity: 29 }  
I20160604-19:32:05.782(2)? { temperature: 36, pressure: 938.3, humidity: 29.1 }  
I20160604-19:32:06.932(2)? { temperature: 36, pressure: 938.3, humidity: 28.9 }  
I20160604-19:32:08.074(2)? { temperature: 36, pressure: 938.3, humidity: 28.7 }
```

Y comprobamos los mensajes que está recibiendo el servidor

De esta manera comprobamos que el flujo de **Raspberry > Broker > Servidor** funciona correctamente, pudiendo empezar a montar el renderizado de los datos en la página.

Para ello, y como vamos a sacar los datos de la base de datos, hay que guardar los mensajes en **MongoDB**, para su posterior reutilización.

Crearemos una colección de datos en `imports/api/data.js`, que llevará el siguiente contenido:

```
import { Mongo } from 'meteor/mongo';  
  
export const Data = new Mongo.Collection('data');
```

Necesitaremos importar esta colección en el servidor, para poder agregarle los objetos que nos lleguen con los mensajes.

```
import { Mongo } from 'meteor/mongo';  
import { Data } from '../imports/api/data.js';
```

Una vez tenemos lo necesario, podemos empezar a guardar en base de datos todo lo que nos llegue.

```
client.on("message", function (topic, message) {  
  //console.log(JSON.parse(message.toString().replaceAll('"', "\\\"")));  
  insertMessage(JSON.parse(message.toString().replaceAll('"', "\\\"")));  
});
```

El mensaje llegará a la función **insertMessage** como **JSON**, para poder sacar sus variables y guardar los datos en **MongoDB**.

```
function insertMessage(message) {  
  var now = new Date(),  
      h = now.getHours().toString(),  
      m = now.getMinutes().toString(),  
      s = now.getSeconds().toString(),  
      d = now.getDate().toString(),  
      mo = now.getUTCMonth().toString(),  
      y = now.getUTCFullYear().toString();  
  h = h.length < 2 ? "0" + h : h;  
  m = m.length < 2 ? "0" + m : m;  
  s = s.length < 2 ? "0" + s : s;  
  d = d.length < 2 ? "0" + d : d;  
  mo = mo.length < 2 ? "0" + mo : mo;  
  
  now = d + "/" + mo + "/" + y + " - " + h + ":" + m + ":" + s;  
  
  Fiber(function () {  
    Data.insert({ temperature: message.temperature, pressure: message.pressure, humidity: message.humidity, time: now });  
  }).run();  
}
```

En la función **insertMessage**, formatearemos la fecha del servidor para mostrarla de manera más legible.

Usaremos **Fiber()**, necesaria debido a que **NodeJS** es asíncrono, y carga todo el código a la vez, en lugar de seguir un orden. Si no lo implementamos, el servidor lanzaría una excepción.

```
/home/jrt/mqttPrueba/Sensores/.meteor/local/build/programs/server/packages/meteor.js:1060  
  throw new Error("Meteor code must always run within a Fiber. " +  
    ^  
Error: Meteor code must always run within a Fiber. Try wrapping callbacks that you pass to non-Meteor libraries with Meteor.bindEnvironment.  
    at Object.Meteor._nodeCodeMustBeInFiber (packages/meteor/dynamics/nodejs.js:9:1)  
    at [object Object]._.extend.get (packages/meteor/dynamics/nodejs.js:21:1)  
    at Object.DDP.randomStream (packages/ddp-client/random-stream.js:5:1)  
    at [object Object].self._makeNewID (packages/mongo/collection.js:75:19)  
    at [object Object].insert (packages/mongo/collection.js:474:22)  
    at insertMessage (server/main.js:30:10)  
    at MqttClient.<anonymous> (server/main.js:61:9)  
    at MqttClient.emit (events.js:106:17)  
    at MqttClient._handlePublish (/home/jrt/mqttPrueba/Sensores/node_modules/mqtt/lib/client.js:765:12)  
    at MqttClient._handlePacket (/home/jrt/mqttPrueba/Sensores/node_modules/mqtt/lib/client.js:272:12)
```

Una vez guardados los datos en **MongoDB**, podemos mostrarlos en la vista.

En la carpeta **imports/ui/**, tenemos dos archivos:

### **body.html**

Será la plantilla para la vista de la página principal, aquí recibimos los datos para recibirlos en pantalla. Los módulos de **HTML** se cargan con la sintaxis

**{{ > nombrePlantilla }}**, y las plantillas se definen como

**<template name = "nombrePlantilla">**

```
<body>
  <div class="container">
    <table>
      <thead>
        <tr>
          <th>Fecha</th>
          <th>Temperatura</th>
          <th>Presión</th>
          <th>Humedad</th>
        </tr>
      </thead>
      <tbody>
        {{#each Data}} {{> sensorData}} {{/each}}
      </tbody>
    </table>
  </div>
</body>
<template name="sensorData">
  <tr>
    <td id="time">{{ time }}</td>
    <td><span id="temp">{{ temperature }}</span>°C</td>
    <td><span id="pressure">{{ pressure }}</span>Mb</td>
    <td><span id="humidity">{{ humidity }}</span>%</td>
  </tr>
</template>
```

### **body.js**

```
import { Template } from 'meteor/templating';
import { Data } from '../api/data.js';

import './body.html';

Template.body.helpers({

  Data() {
    return Data.find({}, { sort: { time: -1 }, limit: 1 });
  }

});
```

De aquí sacaremos los datos que se usarán en las plantillas. En este caso, haremos una búsqueda en **MongoDB** sacando todos los objetos limitados a 1, y el que tenga la fecha superior (el último insertado), de esta manera lograremos que la tabla se refresque con cada dato insertado.

## 4. Resultados y discusión

- Resultados

Al final se ha logrado la funcionabilidad principal deseada, pero no se ha llegado a cubrir todos los aspectos que desde un principio se propusieron.

Se ha logrado enviar los datos de los sensores desde la Raspberry hasta el servidor, y que éste los renderice en cliente, pero no de la manera deseada.

El principal problema que se tuvo en esta aplicación era el envío de los mensajes desde el servidor hasta el cliente, y el momento de insertarlo en base de datos.

Se pretendía que los mensajes llegaran al cliente, pero sólo guardando un dato por hora.

Para futuras evaluaciones de la aplicación, se investigaría y haría uso de **Meteor.methods**, donde se declaran métodos que tanto el servidor y el cliente podrían aprovechar, guardando con la frecuencia deseada.

Para evitar estos problemas, se decidió insertar cada dato en base de datos, para que luego el cliente hiciera uso de ellos.

Se incluiría también una serie de gráficos y filtros por mes, año o incluso por un rango entre los datos (temperatura de 20 a 30 grados), haciendo uso de las interfaces ofrecidas por <http://morrisjs.github.io/morris.js/>

- Escalabilidad de la aplicación

Esta aplicación, y las tecnologías que en ella han sido aplicadas, permite tener un pequeño servidor de sencillas prestaciones, puesto que la lógica y el peso de la aplicación reside en el envío y recepción de mensajes **MQTT**, y este protocolo se usa para la comunicación de pequeños dispositivos para su gestión externa.

## 5. Conclusiones

Se han investigado nuevas líneas de tecnología, frameworks y lenguajes distintos.

El uso de **MQTT** ha supuesto un verdadero descubrimiento, y ha sido de suma ayuda para entender la idea de **IoT** (Internet of Things), siendo aplicable a multitud de desarrollos debido a su facilidad de uso y su rendimiento frente a recursos usados.

El uso de la **Raspberry** y la investigación de este dispositivo han dado lugar a una inquietud por desarrollar proyectos mayores y más llamativos, dando más protagonismo a la aplicación que a la tecnología aplicada.

Esta aplicación ha permitido el estudio del lenguaje usado por la **Raspberry (Python)**, dando la posibilidad de dar el salto a otro tipo de desarrollos (plugins para software que funcionen bajo este lenguaje), además de ampliar el conocimiento sobre el desarrollo y sus diferentes ramas.

## 6. Bibliografía

### ➤ Información IoT y MQTT

- <http://internetofthingsagenda.techtarget.com>
- <https://www.ibm.com>
- <http://ovacen.com>
- <https://www.rti.com>
- <https://www.theguardian.com>
- <https://github.com/mqtt/mqtt.github.io/wiki>
- <https://geekytheory.com/que-es-mqtt/>
- <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>
- [http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN\\_spec\\_v1.2.pdf](http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf)

### ➤ Herramientas y utilidades

- |   |                           |
|---|---------------------------|
| • <a href="http://www.hivemq.com">http://www.hivemq.com</a>   | - Cliente Websocket       |
| • <a href="https://www.accenture.com">https://www.accenture.com</a>                                     | - Protocolo MQTT          |
| • <a href="https://www.cloudmqtt.com">https://www.cloudmqtt.com</a>                                     | - Broker de mensajes MQTT |
| • <a href="http://www.eclipse.org/paho/clients/python/">http://www.eclipse.org/paho/clients/python/</a> | - Envío de mensajes MQTT  |

### ➤ Tutoriales Meteor

- <https://jsjutsu.com/blog/2015/12/megatutorial-meteor-uno/>
- <https://www.meteor.com>



## 7. Anexo

### 7.1 Frameworks

En principio no es necesario usar un framework para el proyecto, pero siempre facilitan las operaciones.

La lista de frameworks disponibles en Python están en el siguiente enlace:

<https://wiki.python.org/moin/WebProgramming>

Django es el que más posibilidades van a ofrecer para el proyecto, pero también es uno de los más complejos.

Pyjs es otro framework donde el trabajo que podría realizar javascript en una aplicación lo realiza **Python**, de manera que nos podemos comunicar fácilmente con la Raspberry Pi.

Para pequeñas aplicaciones podemos usar otros frameworks como Bottle, Flask o Karrigell.

Bottle y Flask son los más accesibles en cuanto a creación de aplicaciones, por lo tanto los primeros proyectos se hicieron con estos frameworks.

<b>Django:</b>	<a href="https://docs.djangoproject.com/en/1.9/intro/tutorial01/">https://docs.djangoproject.com/en/1.9/intro/tutorial01/</a>
<b>Pyjs:</b>	<a href="http://pyjs.org/">http://pyjs.org/</a>
<b>Bottle:</b>	<a href="http://bottlepy.org/docs/dev/index.html">http://bottlepy.org/docs/dev/index.html</a>
<b>Flask:</b>	<a href="http://flask.pocoo.org/docs/0.10/">http://flask.pocoo.org/docs/0.10/</a>
<b>Karrigell:</b>	<a href="http://karrigell.sourceforge.net/en/pythoninsidehtml.html">http://karrigell.sourceforge.net/en/pythoninsidehtml.html</a>

Al final se eligió Meteor, no sólo por la facilidad de carga de plantillas, sino por la reactividad que lleva en sus documentos.

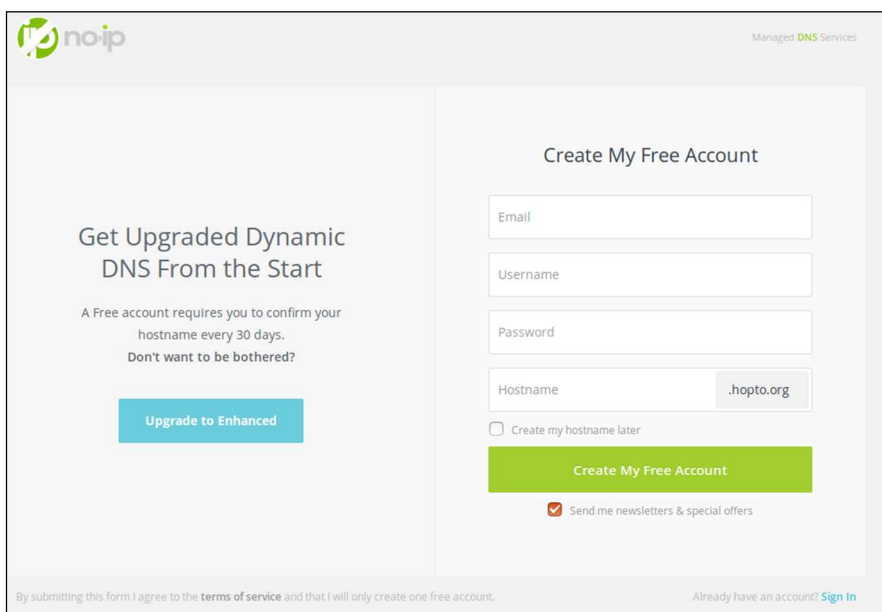
Está escrito sobre **NodeJS**, usando Javascript como lenguaje, y ambas tecnologías habían sido usadas con anterioridad, lo que nos facilitaría la inmersión al framework.

## 7.2 Acceso a Raspberry desde múltiples localizaciones

Para tener acceso a la consola de la Raspberry desde cualquier localización, enrutaremos todo el tráfico entrante a la IP externa del router desde el puerto 3333 al puerto 22 de la dirección IP de la RaspBerry, para tener acceso por **SSH** desde cualquier parte con acceso a Internet.

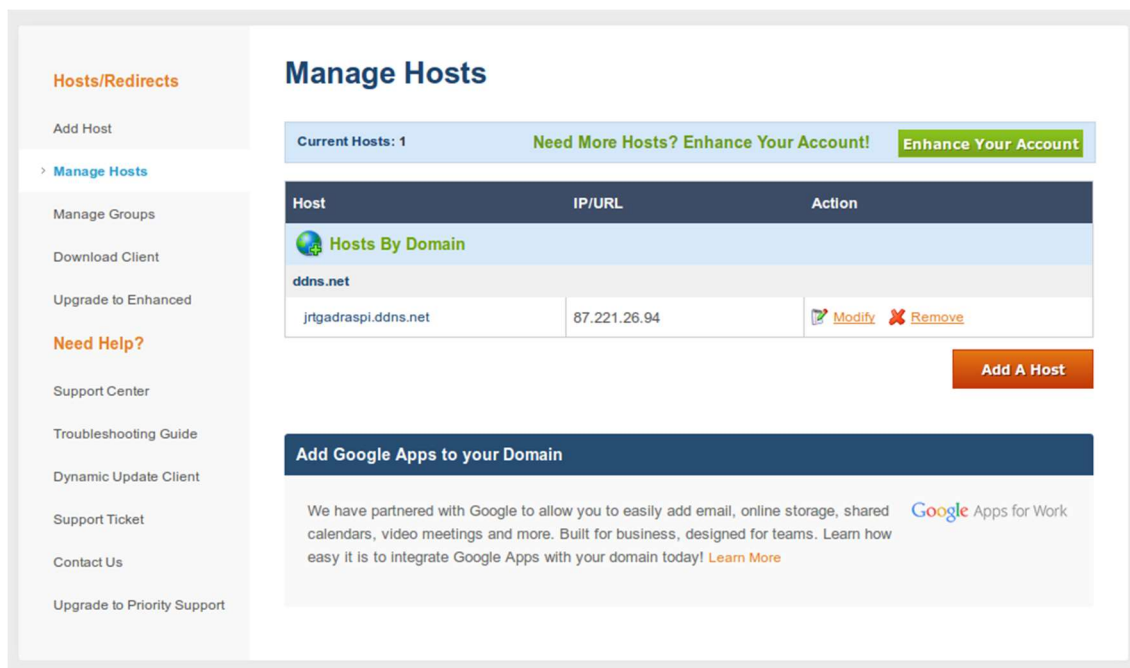
Para ello, necesitamos tener una IP externa fija para el router. Podemos utilizar un programa multiplataforma como es No-IP, que nos permite asignar un dominio a nuestra IP externa, y cada vez que cambie esta, cambiará la IP asignada al dominio.

Necesitaremos registrarnos en la página:



The screenshot shows the No-IP website's registration page. On the left, there is a section titled "Get Upgraded Dynamic DNS From the Start" with a note about the 30-day confirmation requirement for free accounts and a button to "Upgrade to Enhanced". On the right, under the heading "Create My Free Account", there is a form with fields for Email, Username, Password, and Hostname (with ".hopto.org" as a suggestion). There is a checkbox for "Create my hostname later" and a green "Create My Free Account" button. Below the button is a checked checkbox for "Send me newsletters & special offers". At the bottom, there is a disclaimer about terms of service and a link to "Sign In" for existing users.

Una vez registrados, creamos un dominio y le agregamos la IP indicada:

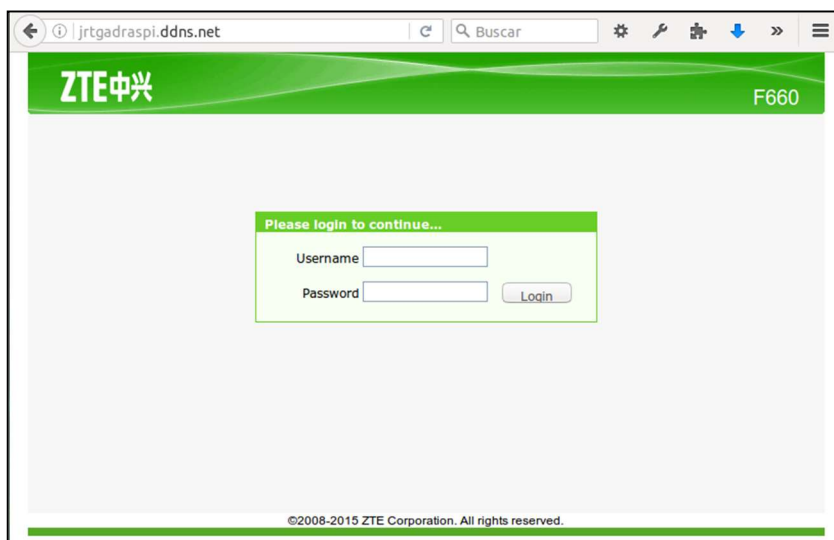


The screenshot shows a web interface for managing hosts. On the left is a sidebar with navigation links: Hosts/Redirects, Add Host, Manage Groups, Download Client, Upgrade to Enhanced, Need Help?, Support Center, Troubleshooting Guide, Dynamic Update Client, Support Ticket, Contact Us, and Upgrade to Priority Support. The main content area is titled 'Manage Hosts' and shows 'Current Hosts: 1'. It features a table with columns 'Host', 'IP/URL', and 'Action'. The table lists a host 'jrtgdraspi.ddns.net' with IP '87.221.26.94'. Below the table is a button 'Add A Host'. There is also a section 'Add Google Apps to your Domain' with a description and a 'Learn More' link.

Host	IP/URL	Action
jrtgdraspi.ddns.net	87.221.26.94	<a href="#">Modify</a> <a href="#">Remove</a>

Cuando lo tengamos enlazado, nos descargamos la interfaz ([ver descarga](#)), en este caso descargaremos el programa para Linux, que en lugar de una interfaz visual, es un daemon (proceso en segundo plano) que se puede configurar por consola.

Cuando esté bien configurado, si metemos el dominio, traducirá a nuestra IP, abriendo directamente la página de configuración del router:





Si en lugar de meter el dominio (jrtgadrspi.ddns.net) sin puerto, le agregamos el puerto de entrada, tendremos acceso directo a la Raspberry:

```
jrt@jrtCasa:~/mqttPrueba/Sensores$ ssh pi@jrtgadrspi.ddns.net -p 3333
The authenticity of host 'jrtgadrspi.ddns.net]:3333 ([87.221.26.94]:3333)' can't be established.
ECDSA key fingerprint is 86:6b:a1:c0:7d:bb:7f:65:a2:1c:c5:9a:08:f6:89:c3.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'jrtgadrspi.ddns.net]:3333,[87.221.26.94]:3333' (ECDSA) to the list of known hosts.
pi@jrtgadrspi.ddns.net's password:
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sat Jun  4 20:45:49 2016
pi@raspberrypi:~$
```

\$ > ssh pi@jrtgadrspi.ddns.net -p 3333

Nos conectamos con el usuario pi a través del dominio de NO-IP en su puerto 3333, y como está el tráfico redirigido, entra por el puerto 22 de la **Raspberry**.

Enable	Name	WAN Host Start IP Address	WAN Start Port	LAN Host Start Port	WAN Connection	Modify	Delete
	Protocol	WAN Host End IP Address	WAN End Port	LAN Host End Port	LAN Host Address		
✓	Raspi		3333	22	WANConnection		
	TCP		3333	22	192.168.1.89		

### 7.3 Salidas por consola de la aplicación

Salida console log 1

```
{ options:
  { protocol: 'mqtt',
    slashes: true,
    auth: null,
    host: 'm21.cloudmqtt.com',
    port: 13438,
    hostname: 'm21.cloudmqtt.com',
    hash: null,
    search: '',
    query: {},
    pathname: null,
    path: null,
    href: 'mqtt://m21.cloudmqtt.com',
    clientId: 'clientId-3IJf14ulaSasd',
    username: 'meteorServer',
    password: 'meteorserverpass',
    clean: false,
    insert: true,
    sync: true,
    readable: true,
    keepalive: 10,
    reschedulePings: true,
    protocolId: 'MQTT',
    protocolVersion: 4,
    reconnectPeriod: 1000,
    connectTimeout: 30000 },
  ... }
```

Salida console log 2

```
streamBuilder: [Function: wrapper],
outgoingStore: { _inflight: {} },
incomingStore: { _inflight: {} },
queueQoSZero: true,
pingTimer: null,
connected: false,
disconnecting: false,
queue: [],
connackTimer:
  { _idleTimeout: 30000,
    _idlePrev:
      { _idleNext: [Circular],
        _idlePrev: [Circular],
        msecs: 30000,
        ontimeout: [Function: listOnTimeout] },
    _idleNext:
      { _idleNext: [Circular],
        _idlePrev: [Circular],
        msecs: 30000,
        ontimeout: [Function: listOnTimeout] },
    _idleStart: 1465056374403,
    _monotonicStartTime: 8317253,
    _onTimeout: [Function],
    _repeat: false },
  reconnectTimer: null,
  nextId: 4916,
  outgoing: {},
  events:
    { connect: [ [Function], [Function], [Function] ],
      close: [ [Function], [Function], [Function] ] },
  domain: null,
```

Salida console log 3

```
events:
  { end: [Object],
    finish: [Object],
    _socketEnd: [Function: onSocketEnd],
    readable: [Function: pipeOnReadable],
    error: [Object],
    close: [Function],
    connect: [Object] },
_maxListeners: 1000,
_writableState:
  { highWaterMark: 16384,
    objectMode: false,
    needDrain: false,
    ending: false,
    ended: false,
    finished: false,
    decodeStrings: false,
    defaultEncoding: 'utf8',
    length: 60,
    writing: true,
    sync: false,
    bufferProcessing: false,
    onwrite: [Function],
    writecb: [Function],
    writelen: 60,
    buffer: [],
    errorEmitted: false },
```

Salida console log 4

```
_maxListeners: 10,
stream:
  { _connecting: true,
    _handle:
      { fd: -1,
        writeQueueSize: 0,
        owner: [Circular],
        onread: [Function: onread] },
    _readableState:
      { highWaterMark: 16384,
        buffer: [],
        length: 0,
        pipes: [Object],
        pipesCount: 1,
        flowing: true,
        ended: false,
        endEmitted: false,
        reading: false,
        calledRead: false,
        sync: true,
        needReadable: false,
        emittedReadable: false,
        readableListening: false,
        objectMode: false,
        defaultEncoding: 'utf8',
        ranOut: false,
        awaitDrain: 0,
        readingMore: false,
        decoder: null,
        encoding: null },
    readable: false,
    domain: null,
```

Salida console log 5

```
{ end: [Object],
  finish: [Object],
  _socketEnd: [Function: onSocketEnd],
  readable: [Function: pipeOnReadable],
  error: [Object],
  close: [Function],
  connect: [Object] },
_maxListeners: 1000,
_writableState:
  { highWaterMark: 16384,
    objectMode: false,
    needDrain: false,
    ending: false,
    ended: false,
    finished: false,
    decodeStrings: false,
    defaultEncoding: 'utf8',
    length: 60,
    writing: true,
    sync: false,
    bufferProcessing: false,
    onwrite: [Function],
    writecb: [Function],
    writelen: 60,
    buffer: [],
    errorEmitted: false },
writable: true,
allowHalfOpen: false,
onend: null,
destroyed: false,
bytesRead: 0,
_bytesDispatched: 0,
_pendingData: <Buffer 10 3a 00 04 4d 51 54 54>,
_pendingEncoding: 'buffer' }
```

## 7.4 Quality of Service (QoS)

Es la garantía de que los mensajes alcancen el objetivo (cliente o broker).

- **Retain-Flag** (Marca de conserva)

Determina que el mensaje será guardado por el broker para el tema específico como el último valor bueno. Los nuevos clientes que se suscriban a ese tema recibirán el último mensaje conservado justo después de suscribirse.

- **Payload** (Contenido)

Es el contenido del mensaje. Se puede enviar cualquier dato independientemente de la codificación.

- **Packet Identifier** (Identificador de paquete)

El identificador de paquete es único para identificar un mensaje en el flujo. Es relevante sólo para QoS superior a 0. Es responsabilidad de la arquitectura el asignar los identificadores.

- **DUP flag** (Marca de duplicado)

Indica que el mensaje ha sido duplicado y reenviado porque el receptor no ha reconocido el mensaje original (no ha enviado ACKNOWLEDGE). Relevante sólo para QoS mayor que 0.

- **Usar QoS 0 cuando ...**

- Tenemos una conexión estable entre remitente y receptor.

- No importa que algún mensaje se pierda de vez en cuando.

- **Usar QoS 1 cuando ...**

- Se necesita recibir cada mensaje.

- Se necesita una rápida entrega.



- **Usar QoS 2 cuando ...**

- Se necesita recibir cada mensaje exactamente una vez.

- No importa que exista una pequeña demora desde el envío hasta la recepción.



## 7.5 Mensajes MQTT

MQTT-Packet: <b>PUBLISH</b> 	MQTT-Packet: <b>SUBSCRIBE</b> 
contains: <b>packetId</b> (always 0 for qos 0)      Example 4314 <b>topicName</b> "topic/1" <b>qos</b> 1 <b>retainFlag</b> false <b>payload</b> "temperature:32.5" <b>dupFlag</b> false	contains: <b>packetId</b> Example 4312 <b>qos1</b> } (list of topic + qos)      1 <b>topic1</b> "topic/1" <b>qos2</b> }      0 <b>topic2</b> "topic/1" ...      ...

Se han reunido los paquetes más importantes, la lista completa de paquetes está en la figura siguiente:

Control packet	Direction of flow	Description
CONNECT	Client to Server	Client request to connect to Server
CONNACK	Server to Client	Connect acknowledgment
PUBLISH	Client to Server or Server to Client	Publish message
PUBACK	Client to Server or Server to Client	Publish acknowledgment
PUBREC	Client to Server or Server to Client	Publish received (assured delivery part 1)
PUBREL	Client to Server or Server to Client	Publish release (assured delivery part 2)
PUBCOMP	Client to Server or Server to Client	Publish complete (assured delivery part 3)
SUBSCRIBE	Client to Server	Client subscribe request
SUBACK	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	Client to Server	Unsubscribe request
UNSUBACK	Server to Client	Unsubscribe acknowledgment
PINGREQ	Client to Server	PING request
PINGRESP	Server to Client	PING response
DISCONNECT	Client to Server	Client is disconnecting

## 7.6 Sintaxis completa Mosquitto

A continuación se presenta la sintaxis completa de publicación de mensajes y suscripción a temas.

### ➤ PUBLISH

```
mosquitto_pub [-A bind_address] [-d] [-h hostname] [-i client_id] [-I client id prefix] [-k keepalive time] [-p port number] [-q message QoS] [--quiet] [-r] [-S] { -f file | -l | -m message | -n | -s } [ [-u username] [-P password] ] [ --will-topic topic [--will-payload payload] [--will-qos qos] [--will-retain] ] [ [ { --cafile file | --capath dir } [--cert file] [--key file] [--ciphers ciphers] [--tls-version version] [--insecure] ] | [ --psk hex-key --psk-identity identity [--ciphers ciphers] [--tls-version version] ] ] [--proxy socks-url] [-V protocol-version] -t message-topic
```

### ➤ SUBSCRIBE

```
mosquitto_sub [-A bind_address] [-c] [-C msg count] [-d] [-h hostname] [-i client_id] [-I client id prefix] [-k keepalive time] [-p port number] [-q message QoS] [-R] [-S] [-N] [--quiet] [-v] [ [-u username] [-P password] ] [ --will-topic topic [--will-payload payload] [--will-qos qos] [--will-retain] ] [ [ { --cafile file | --capath dir } [--cert file] [--key file] [--tls-version version] [--insecure] ] | [ --psk hex-key --psk-identity identity [--tls-version version] ] ] [--proxy socks-url] [-V protocol-version] [-T filter-out...] -t message-topic
```

## 7.7 Glosario de términos

- **CloudMQTT**: Broker gratuito usado para enviar los mensajes a los clientes.
- **DDP: Distributed Data Protocol**. Permite sincronizar MongoDB en servidor y cliente.
- **Framework**: Es una estructura conceptual y tecnológica de soporte definido, que puede servir de base para la organización y desarrollo de software.
- **HiveMQ**: Aplicación online para envío y recepción de mensajes MQTT.
- **IoT: Internet of Things**. Concepto en el que los objetos cotidianos están conectados.
- **M2M: (Machine to Machine)**. Es un concepto genérico que se refiere al intercambio de información o comunicación de datos entre dos máquinas remotas.
- **Meteor**: Framework desarrollado sobre **NodeJS** que utiliza Javascript como lenguaje.
- **MQTT: Message Queue Telemetry Transport**. Protocolo utilizado para el envío y recepción de mensajes por internet.
- **MongoDB**: es un sistema de base de datos **NoSQL** con sintaxis similar a **JSON**.
- **NodeJS**: es un entorno asíncrono de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript.
- **npm: Node Package Manager**. Gestor de dependencias de aplicaciones.
- **Python**: Es un lenguaje de programación interpretado.