ESP32 Y AWS IOT

Comunicación MQTT

ABSTRACT

Guía para establecer una comunicación MQTT entre microcontroladores ESP-32 y AWS IoT usando el entorno ESP-IDF y VS Code.

Federico Cañete

Contenido

Configuración de entorno para Windows 10 y Visual Studio Code 3
Introducción 3
Requisitos de software 3
Requisitos de hardware 3
Configuración de Visual Studio Code 3
Programando proyectos de ejemplo 7
Example Projects 7
Compilación y programación 10
Cómo conectar ESP32 con AWS IoT usando ESP-IDF 15
Requisitos 15
Introducción 15
AWS IoT: Alta del dispositivo y generación de certificados 15
AWS IoT: Creación y asignación de políticas 22
Política sin restricciones 23
Política con acciones definidas 24
Asignación de política 27
ESP32: Proyecto MQTT con Autenticación Mutua 29
Código de ejemplo "ESP-MQTT SSL Sample Application" 29
Configuración de parámetros de conexión y broker MQTT 31
ESP32: Probando el código 38
Fuentes 41

Configuración de entorno para Windows 10 y Visual Studio Code

Introducción

En esta sección se verá cómo configurar un entorno para poder compilar proyectos y bajarlos al microcontrolador ESP32 utilizando el framework ESP-IDF y el editor Visual Studio Code

Para poder programar los microcontroladores ESP32 en cualquiera de sus versiones, el fabricante Espressif dispone del framework de software llamado ESP-IDF (Espressif IoT Development Framework). Se trata de un conjunto de herramientas y APIs que pueden descargarse libremente desde su repositorio público.

Para facilitar el uso de este framework, el fabricante publicó y mantiene actualizada una extensión de Visual Studio Code, que permite integrar realizar todas las funcionalidades del entorno, de una forma amigable para el usuario. En este instructivo se trabajará con esta extensión y se mostrará cómo programar al microcontrolador los ejemplos provistos en el framework.

Requisitos de software

- Visual Studio Code
- Python 3.XX
- Git para Windows

Requisitos de hardware

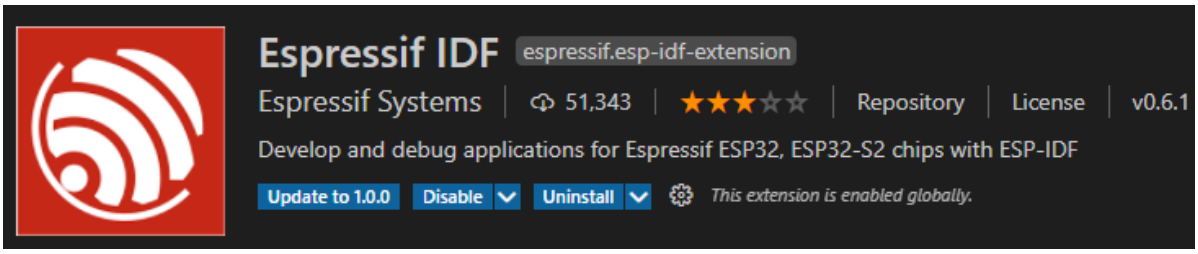
Se requiere el microcontrolador ESP32 en cualquiera de sus versiones. Es necesario poder programarlo vía puerto serie (UART).

Configuración de Visual Studio Code

Para empezar, debemos tener las siguientes extensiones en Visual Studio Code:

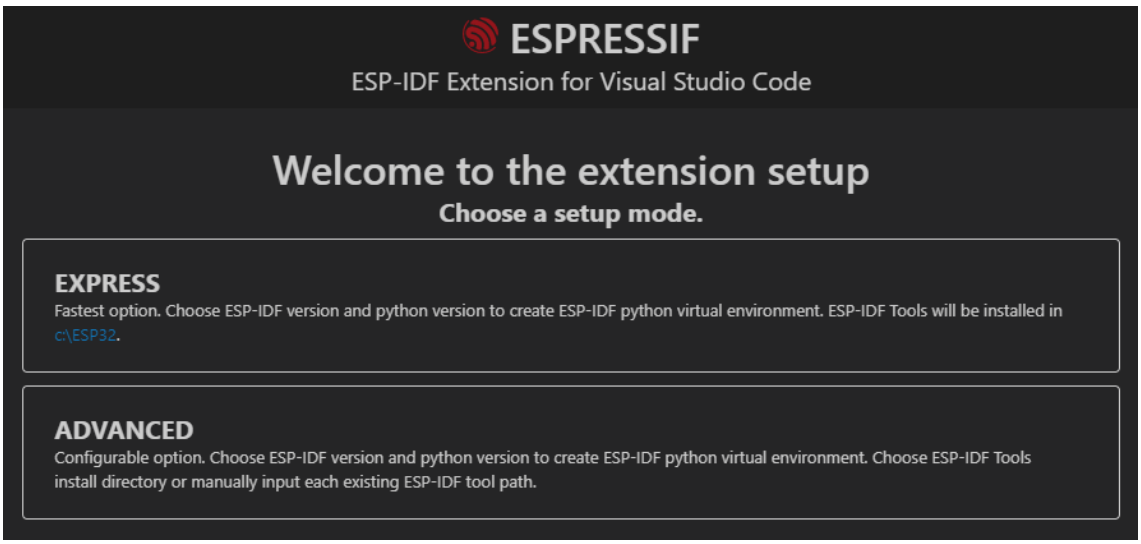
✓ C/C++

- ✓ Espressif IDF



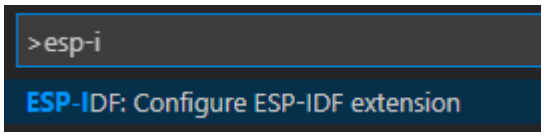
Es muy importante que la versión sea la **0.6.1** dado que, al momento de redactar este documento, las últimas versiones presentan algunos fallos que no tienen aún solución. Se sugiere además desactivar las actualizaciones automáticas de la extensión para evitar inconvenientes.

Al instalarla, aparecerá la siguiente pantalla:

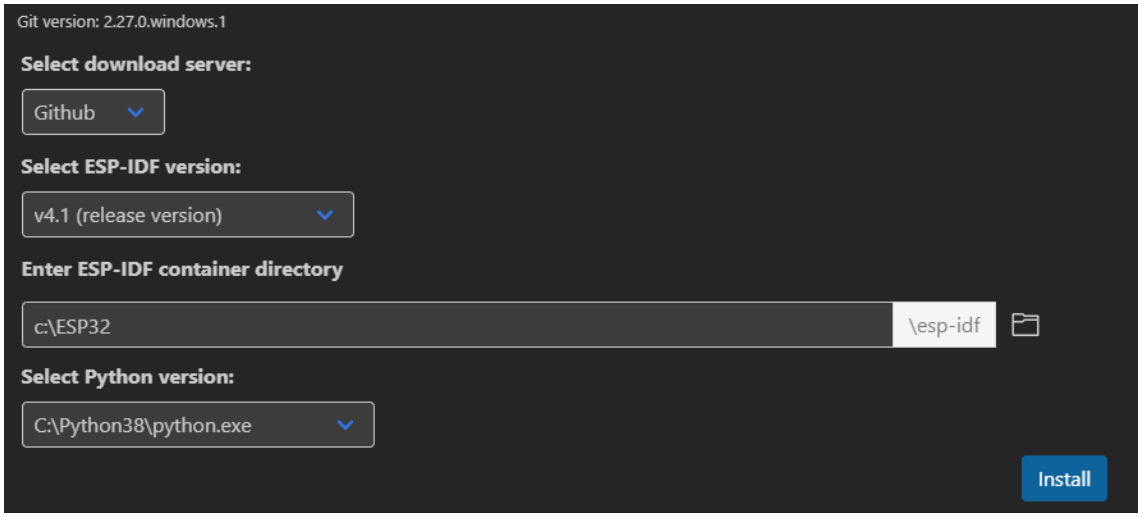


Se trata de un asistente que nos ayudará a configurar el entorno para usar las herramientas que provee el fabricante.

En caso de que no aparezca la ventana de *Onboarding*, abrir la Paleta de Comandos (CTRL+SHIFT+P) y escribir "Configure ESP-IDF extension"



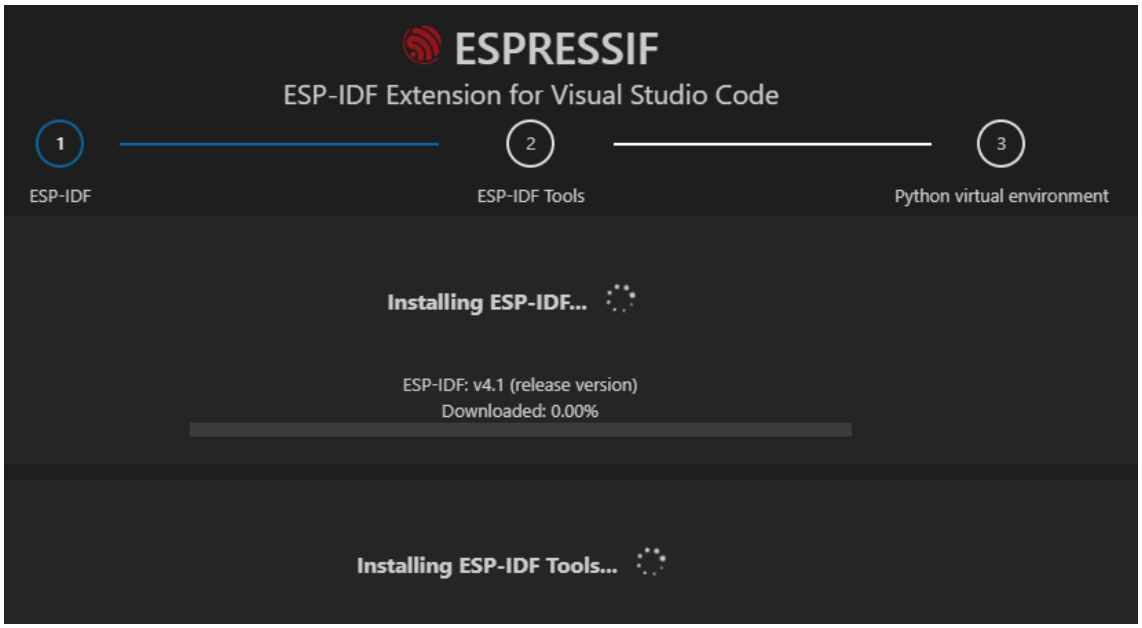
- Seleccionar la instalación *Express*. Aparecerá la siguiente ventana:

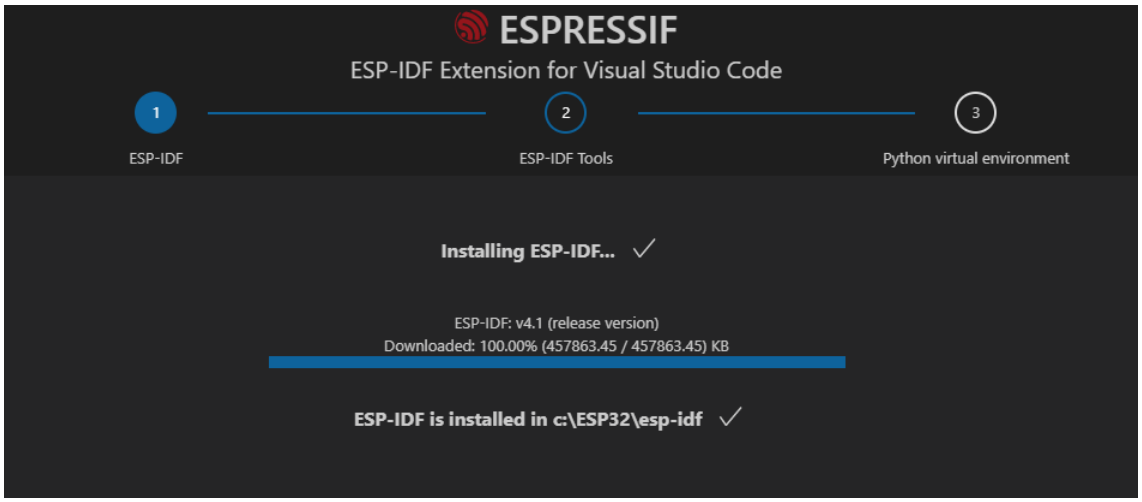


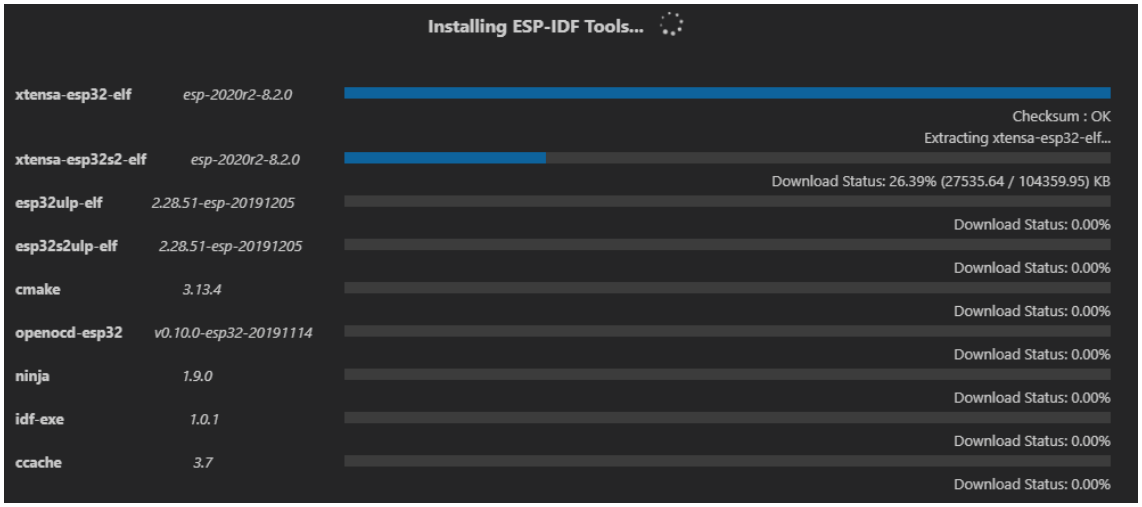
Automáticamente la interfaz buscará dónde se encuentran instalados Git y Python. Debemos seleccionar el directorio donde se descargará e instalará el framework ESP-IDF. Se sugiere para ello crear una carpeta "ESP32" en C:

`C:\ESP32`

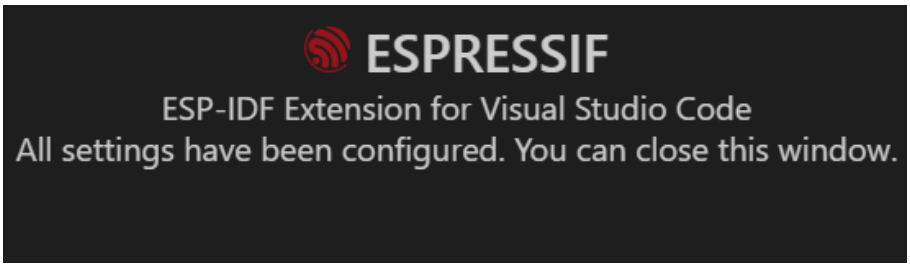
- Tocamos "Install" y la interfaz comenzará automáticamente con la instalación:







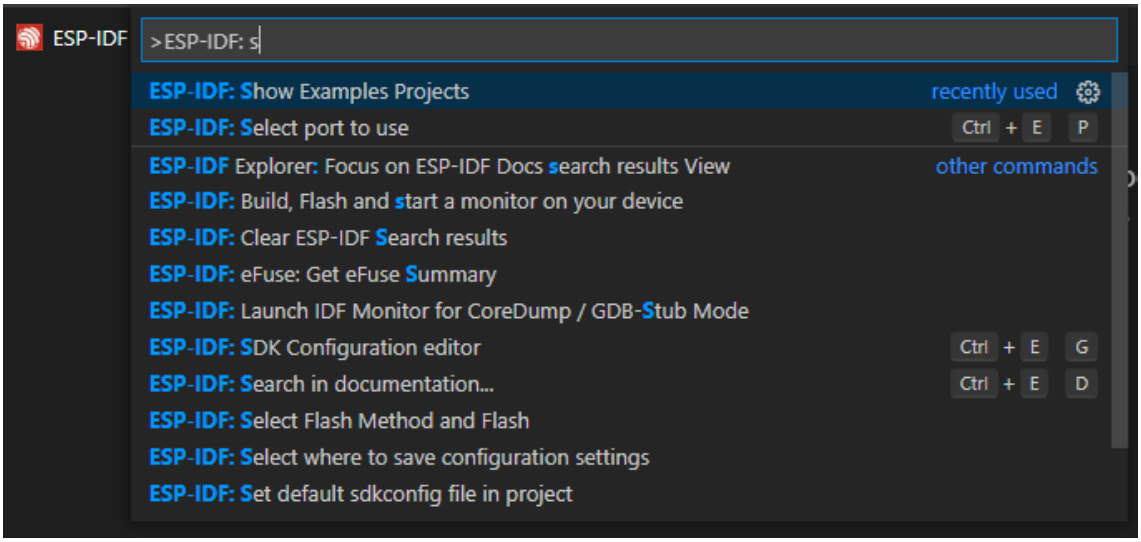
Y al final del proceso se muestra el mensaje de confirmación:



Programando proyectos de ejemplo

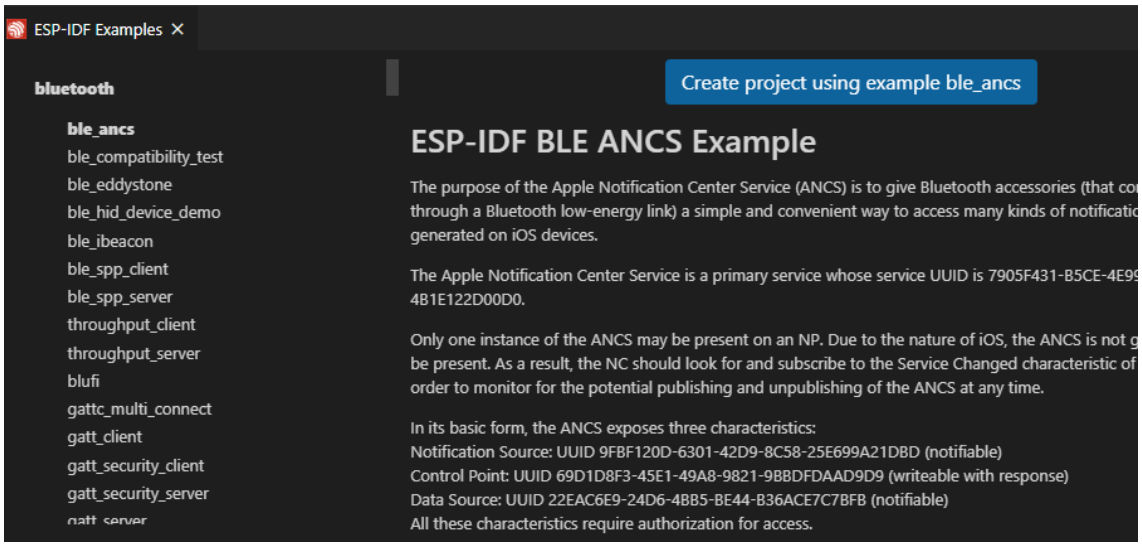
Example Projects

Luego de la instalación, abrimos la Paleta de Comandos con CTRL+SHIFT+P y tipeamos "ESP-IDF Show Examples Projects"

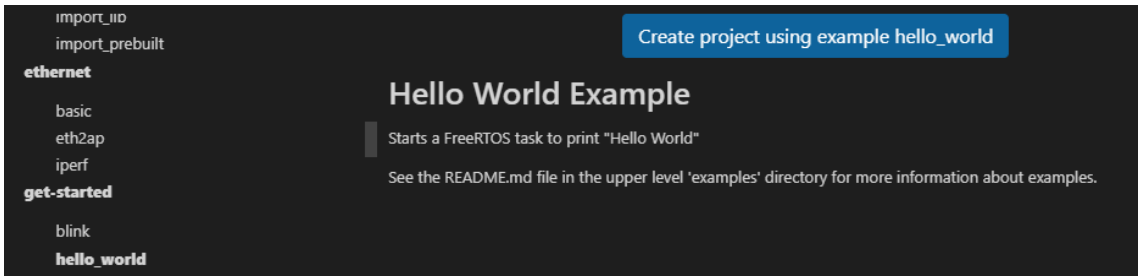


Y nos pedirá que seleccionemos el framework a usar. Usamos el que acabamos de instalar.

A continuación, se muestra una ventana con cada uno de los proyectos de ejemplo que vienen con el framework. Están ordenados por categoría y en cada uno se dispone de una descripción de lo que hace.

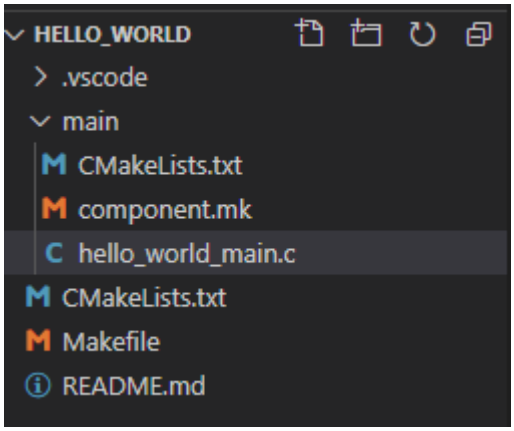


Para este instructivo, vamos a seleccionar el proyecto "Hello World", en la sección *get-started*.

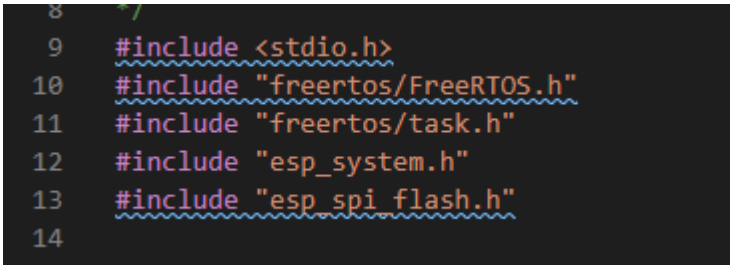


Tocamos en "Create Project" y elegimos una carpeta donde guardar el proyecto.

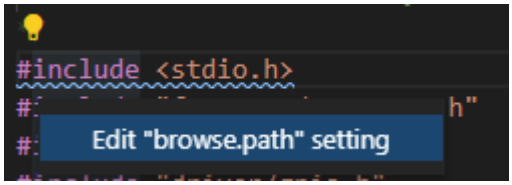
A continuación, se abrirá el proyecto. Podemos editarlo entrando en hello_world_main.c



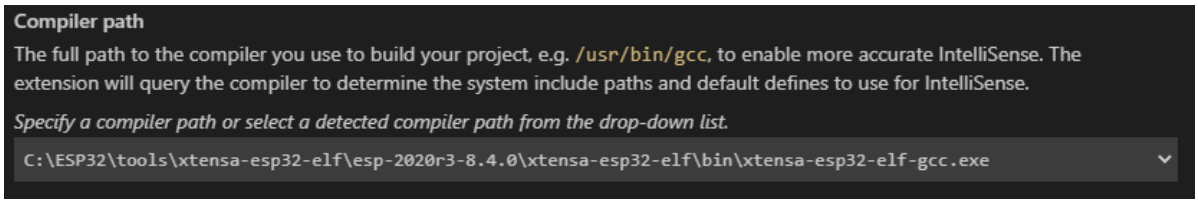
Al abrirlo, se podrá ver cómo algunos headers no han podido ser incluídos:



Debemos solucionar esto haciendo unos cambios en IntelliSense. Para ello, acercamos el mouse al los include y al aparecer una lamparita, hacemos click y elegimos "Edit 'browse.path setting"



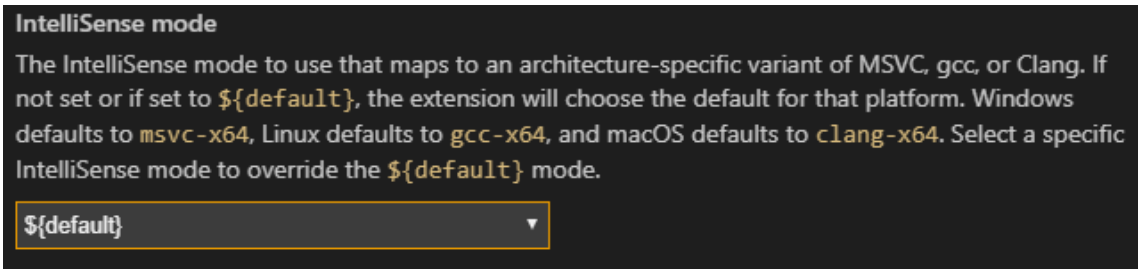
Se nos abrirá la configuración de IntelliSense. Debemos ir a *Compiler Path*.



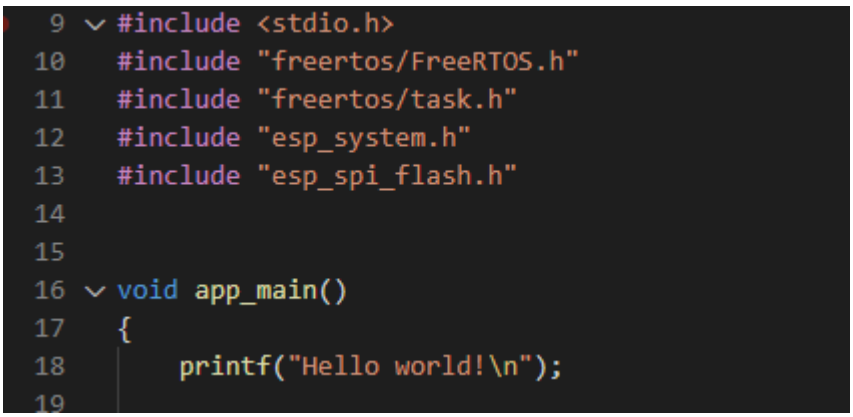
Acá debemos incluir la ruta del compilador GCC, el cual se encuentra en[1]:

C:\ESP32\tools\xtensa-esp32-elf\esp-2020r2-8.2.0\xtensa-esp32-elf\bin\xtensa-esp32-elf-gcc.exe

Por último, en *IntelliSense mode*, setear el modo a default:



De esta forma, si volvemos al archivo main, veremos cómo los errores fueron corregidos:



[1] *Notar que una versión distinta de xtensa cambiará el nombre de la ruta*

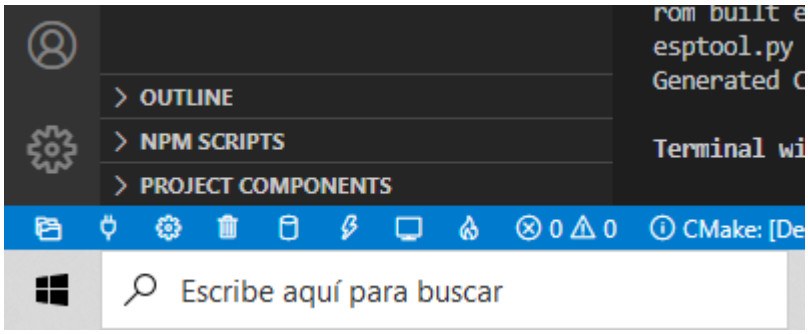
Compilación y programación

Ahora vamos a llevar el programa que abrimos al microcontrolador. En mi caso le modifiqué el texto del saludo:

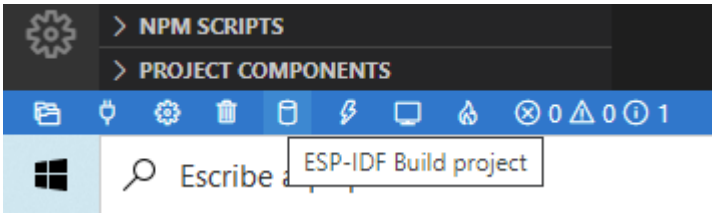
```
printf("Hola desde Visual Studio Code!\n");
```

Podemos compilar el proyecto de la siguiente manera:

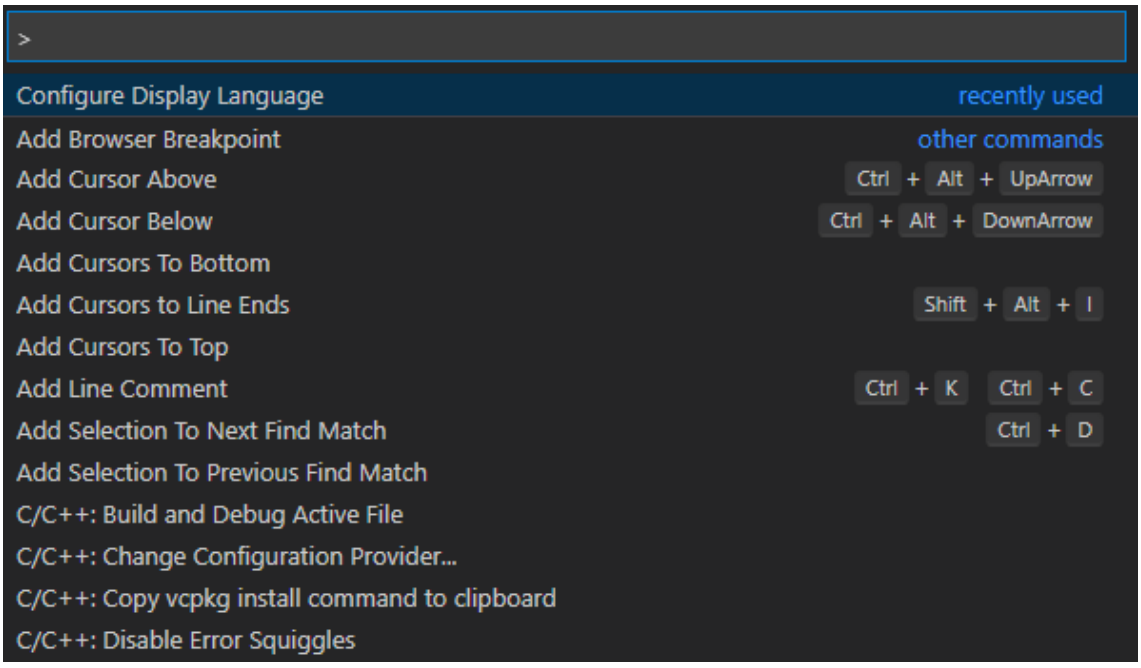
En de la barra de estado, se tienen diferentes íconos de abajo a la izquierda:



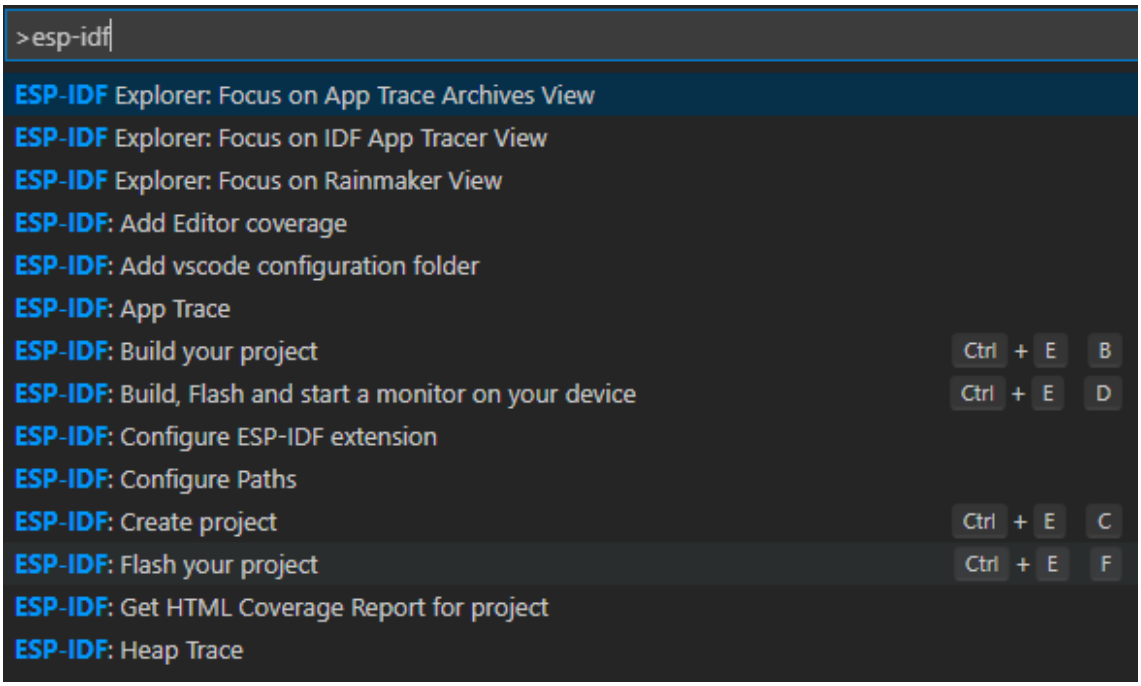
Tocamos el quinto, que es *Build Project*



Otra opción es tocando CTRL+SHIFT+P y se nos aparecerá una barra de búsqueda de comandos.

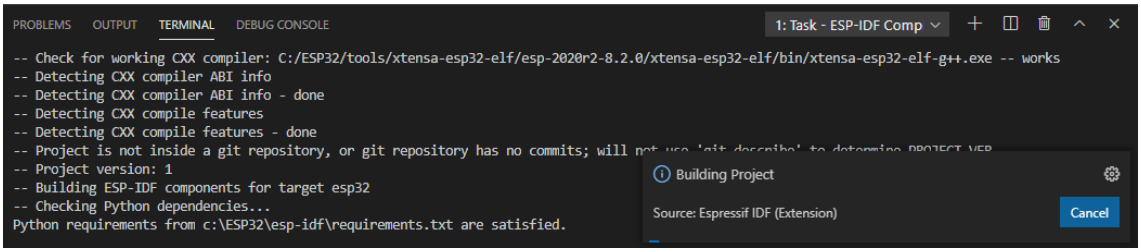


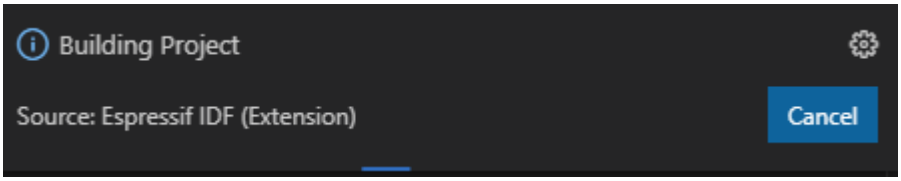
Allí buscamos “esp-idf” y nos aparecerán todos los comandos disponibles.



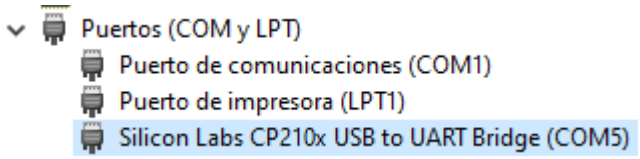
Seleccionamos el comando *build* de la lista o tecleamos su atajo (CTRL+E B)

Nos aparecerá una notificación en la parte inferior. En la primera compilación el proceso puede demorar algunos minutos.



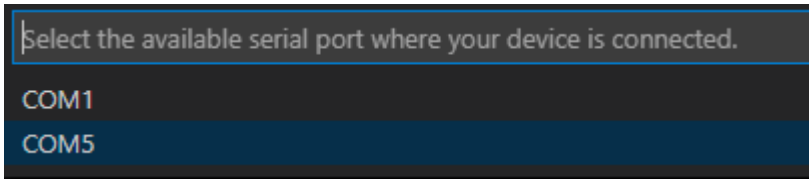


Una vez compilado el proyecto, debemos indicarle al IDE en qué puerto se conecta nuestro ESP32. En este momento debemos conectar el microcontrolador a la PC y saber qué puerto serial utiliza. Podemos ayudarnos con el administrador de dispositivos de Windows. En mi caso es el puerto COM5

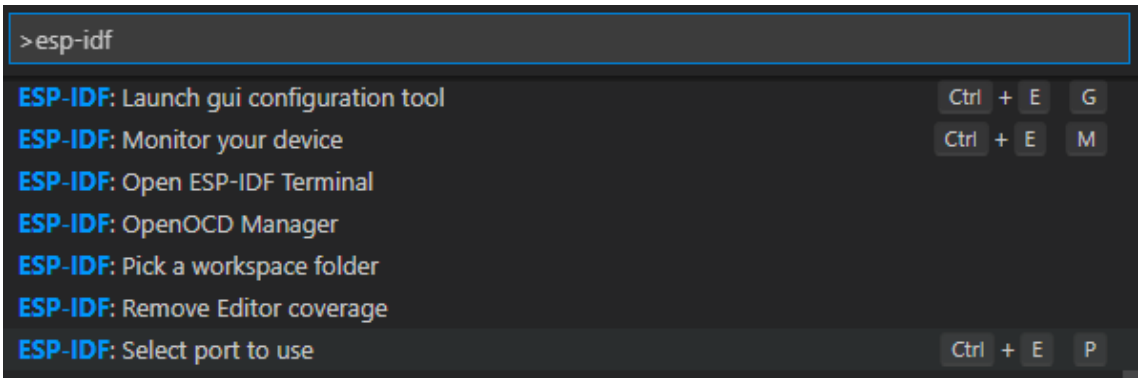


Tocamos el segundo ícono de la barra de estado:

Y seleccionamos el puerto correspondiente

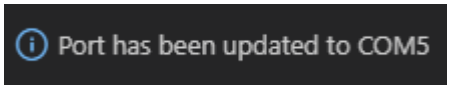


Otra opción es a través del comando *Select port to use*

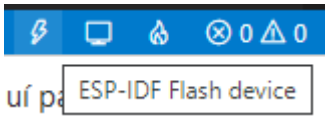


Seleccionamos el comando y luego seleccionamos el puerto:

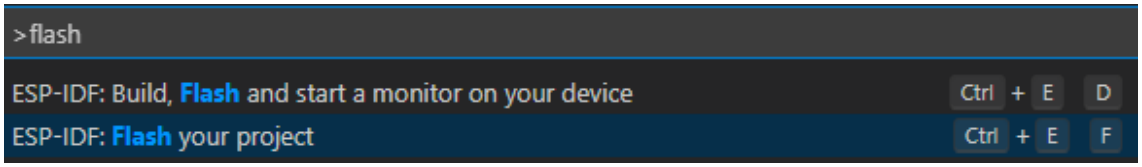
Nos aparecerá inmediatamente una notificación



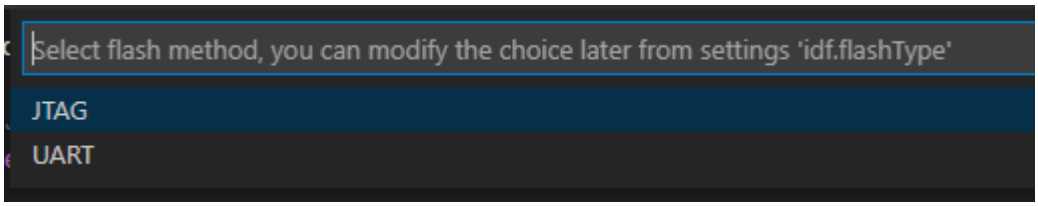
Ya podemos entonces iniciar el flasheo. Usamos para ello el sexto icono comando



O usando el comando *Flash your Project*: CTRL+E F



Se pedirá indicar con qué método programar. Seleccionamos UART.



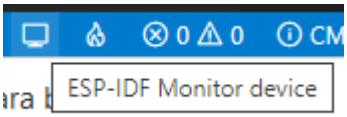
Nos aparecerá el aviso de que se programó correctamente

Y en la terminal veremos lo siguiente

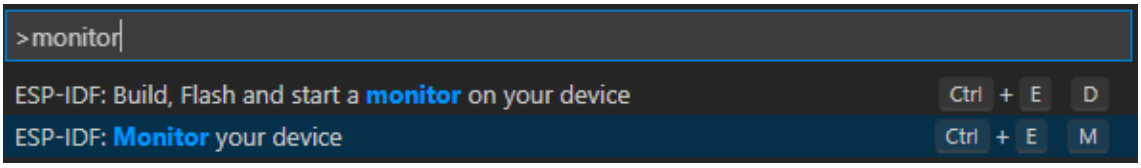
```
Wrote 147488 bytes (76825 compressed) at 0x00010000 in 6.8 seconds (effective 17
2.6 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

Por último, podemos abrir un monitor serial buscando el ícono:



O usando el comando *Monitor your device*



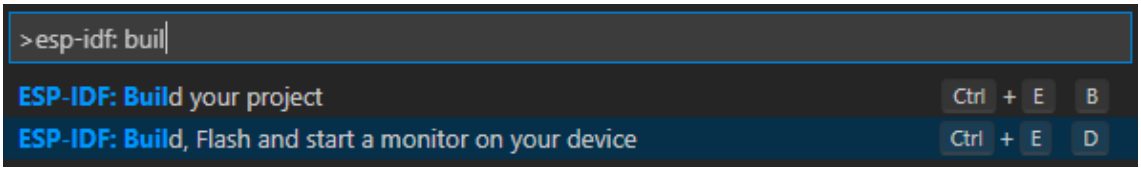
Y se abrirá el monitor serial.

Automáticamente veremos la comunicación del microcontrolador y cómo se ejecuta el programa descargado.

```
I (293) spi_flash: flash io: dio
I (302) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
Hola desde Visual Studio Code!
This is ESP32 chip with 2 CPU cores, WiFi/BT/BLE, silicon revision 1, 4MB external flash
Restarting in 10 seconds...
Restarting in 9 seconds...
Restarting in 8 seconds...
Restarting in 7 seconds...
```

Incluso es posible con un solo comando compilar, programar e iniciar el monitor:

O bien:



Cómo conectar ESP32 con AWS IoT usando ESP-IDF

En esta sección se verá cómo establecer una conexión MQTT entre un ESP32 y el servicio AWS IoT, haciendo uso del framework ESP-IDF de Espressif y el editor VS Code.

Requisitos

- Visual Studio Code con la extensión ESP-IDF en su versión 0.61
- Git
- Cuenta en Amazon Web Services

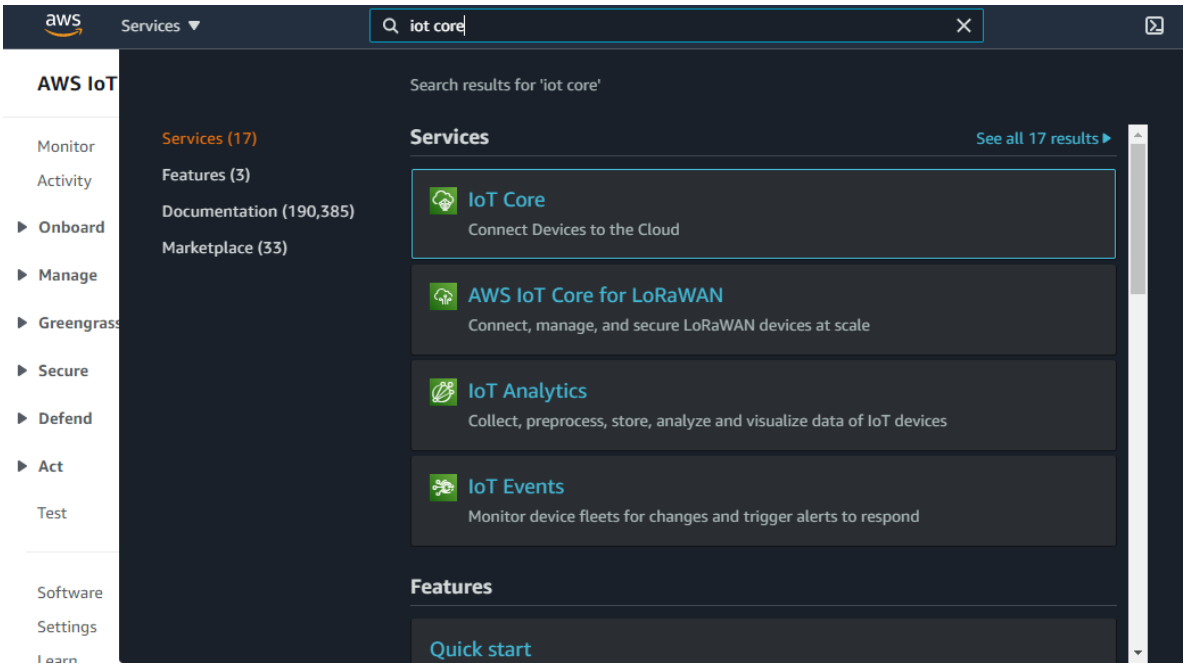
Introducción

El framework ESP-IDF de Espressif ofrece APIs que permiten implementar conexiones MQTT sobre distintos protocolos como TCP, SSL, TLS y Websocket. Para poder conectarnos a los servicios de AWS necesitaremos implementar MQTT sobre SSL usando *mutual authentication*. Usaremos para ello uno de los códigos de ejemplo disponibles en el framework, adaptándolo a los requerimientos de seguridad de AWS IoT.

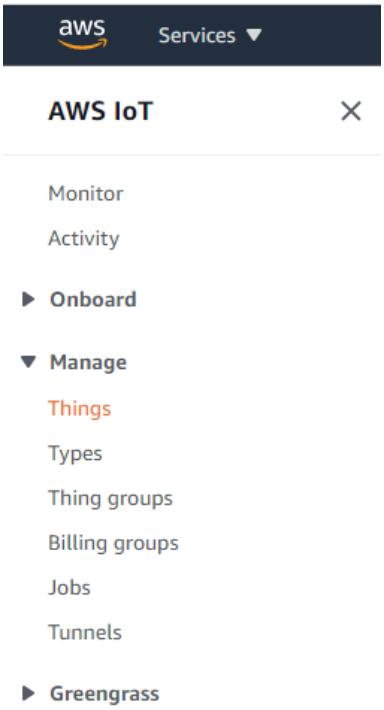
Debemos en primer lugar configurar nuestro dispositivo desde Amazon, generando los certificados y *policies* correspondientes para que la conexión sea exitosa. Una vez terminada esta etapa, nos encargaremos de editar nuestro proyecto y programarlo en el microcontrolador para implementar finalmente la comunicación.

AWS IoT: Alta del dispositivo y generación de certificados

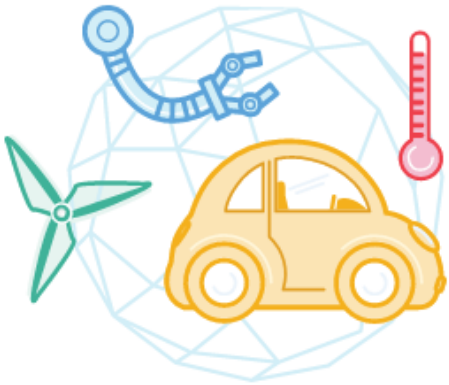
Para empezar, debemos iniciar sesión en nuestra cuenta de AWS y desde la consola dirigirnos a "Iot Core"



Procedemos a dar de alta a nuestro dispositivo, llamado aquí *Thing*. Para ello, nos dirigimos al panel izquierdo y dentro del menú *Manage* seleccionamos *Things*.



A continuación, seleccionamos *Register a thing:*

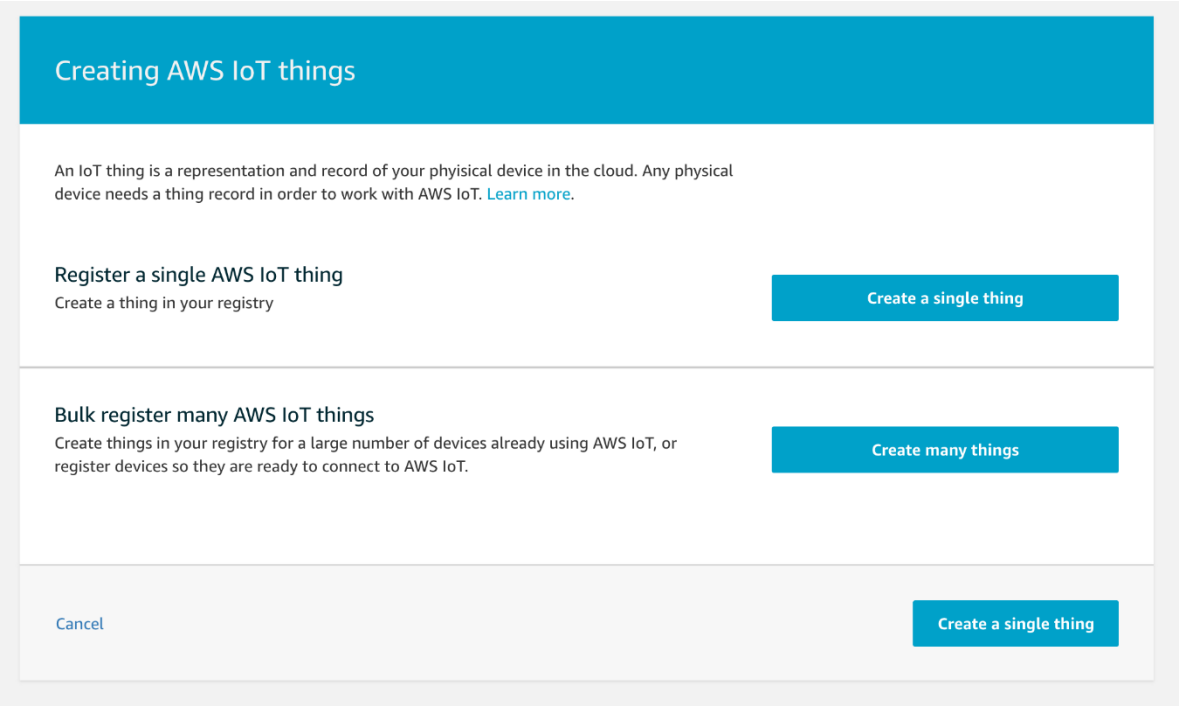


You don't have any things yet

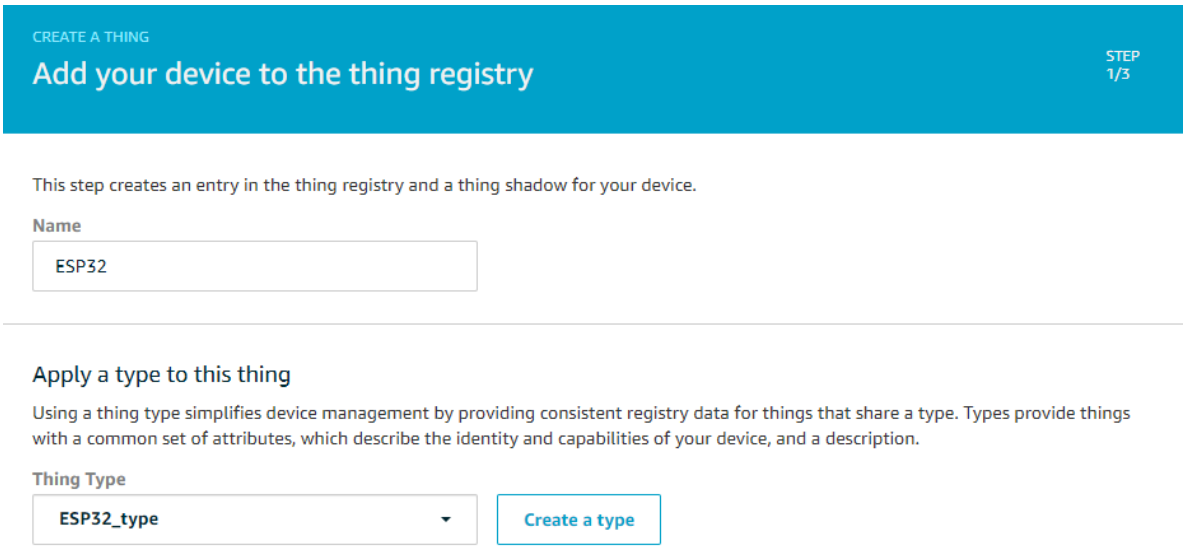
A thing is the representation of a device in the cloud.

Learn more | Register a thing

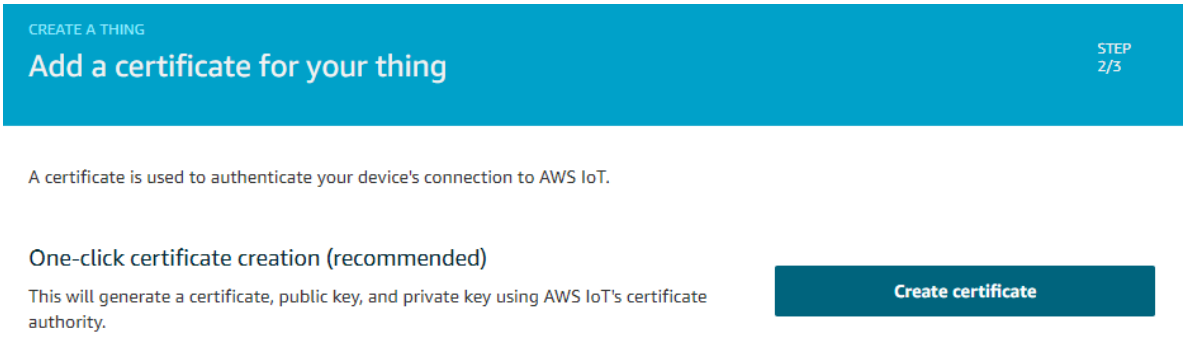
Como en este caso sólo necesitamos dar de alta a nuestro microcontrolador, seleccionamos *Create a single thing:*



En esta sección debemos especificar un nombre para nuestro dispositivo (***Thing Name***). Es importante tener en cuenta que **necesitaremos este nombre más adelante.** Podemos además crear o aplicar un tipo de dispositivo (thing type), aunque no es necesario.

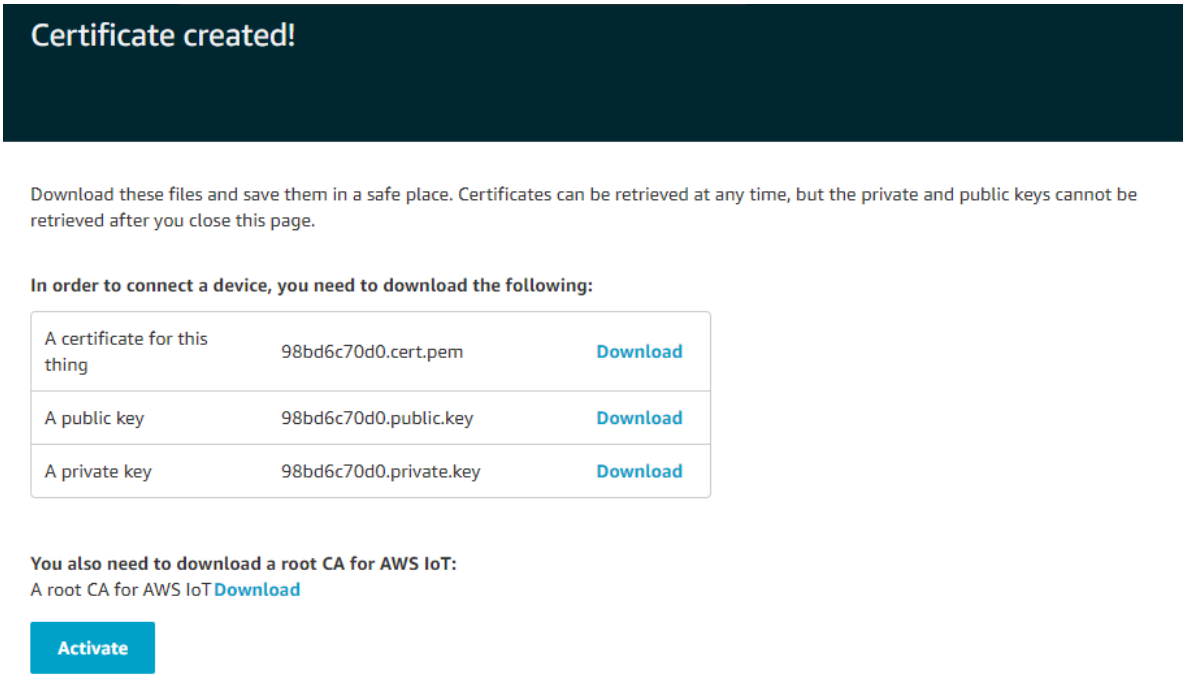


A continuación, debemos crear un certificado de autenticación para nuestro dispositivo. Elegimos la opción "One-click certificate creation" y tocamos *Create certificate*.



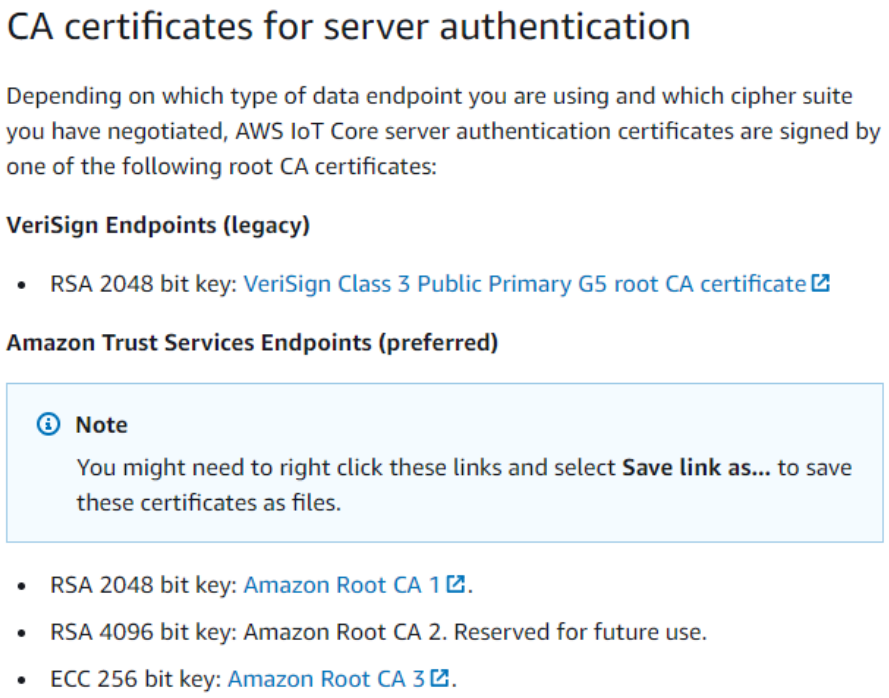
Se generarán entonces tres archivos:

- El certificado del dispositivo
- La clave pública
- La clave privada



Es muy importante descargar los tres archivos en esta etapa. Las claves no podrán descargarse en otro momento, por lo que, si no se lo hace en esta etapa, no se podrá autenticar el dispositivo y, por ende, no podrá entablarse la conexión. **Debemos guardar estos archivos** en un lugar seguro y tenerlos a mano, ya que **se necesitarán para configurar el ESP32 en la próxima sección.**

Por otro lado, también necesitaremos un certificado CA para autenticación con el servidor. Siguiendo el link en *Download*, nos llevará a una nueva página donde podremos descargar dicho certificado.



CA certificates for server authentication

Depending on which type of data endpoint you are using and which cipher suite you have negotiated, AWS IoT Core server authentication certificates are signed by one of the following root CA certificates:

VeriSign Endpoints (legacy)

- RSA 2048 bit key: VeriSign Class 3 Public Primary G5 root CA certificate

Amazon Trust Services Endpoints (preferred)

Note

You might need to right click these links and select **Save link as...** to save these certificates as files.

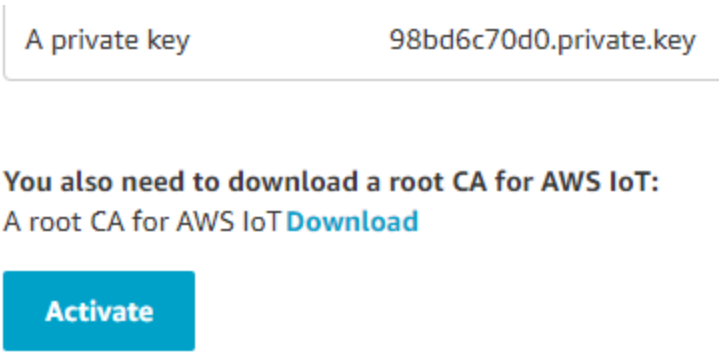
- RSA 2048 bit key: Amazon Root CA 1.
- RSA 4096 bit key: Amazon Root CA 2. Reserved for future use.
- ECC 256 bit key: Amazon Root CA 3.

Podemos elegir cualquier opción. En mi caso, opté por la primera opción (Amazon Root CA 1). Se nos abrirá el archivo en otra pestaña:

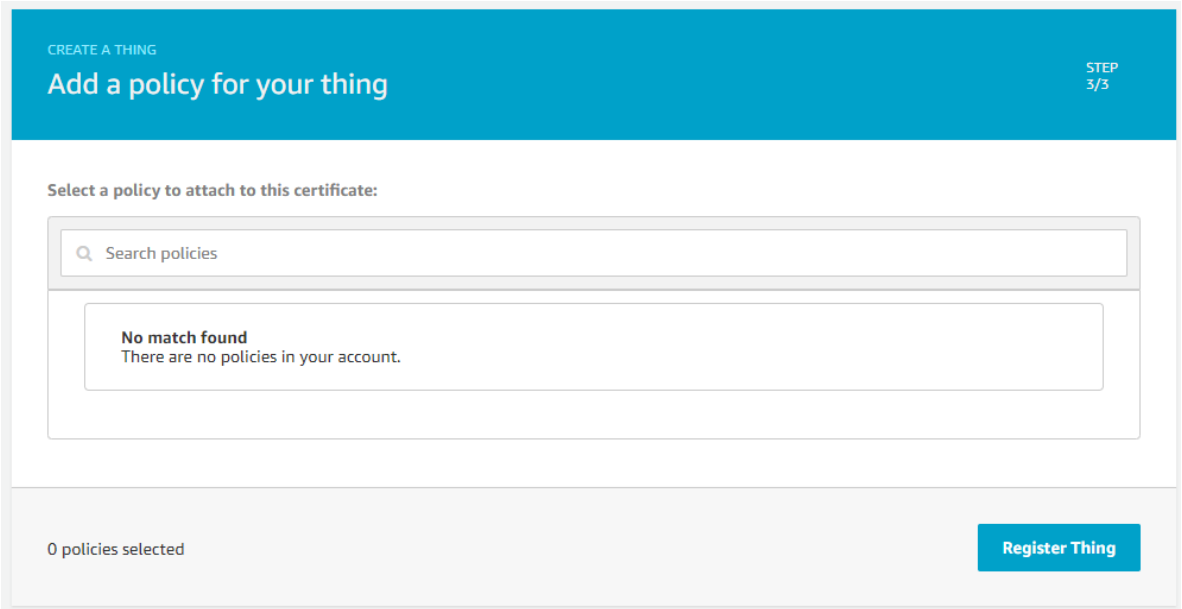
```
-----BEGIN CERTIFICATE-----
MIIDQTCCAimgAwIBAgITBmyfz5m/jAo54vB4ikPmljZbyjANBgkqhkiG9w0BAQsF
ADA5MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6
b24gUm9vdCBDQSAxMB4XDTE1MDUyNjAwMDAwMFoXDTM4MDExNzAwMDAwMFowOTEL
MAkGA1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJv
b3QgQ0EgMTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALJ4gHHKeNXj
ca9HgFB0fW7Y14h29Jlo91ghYPl0hAEvrAIthtOgQ3pOsqTQNroBvo3bSMgHFzZM
9O6II8c+6zf1tRn4SWiw3te5djgdYZ6k/oI2peVKVuRF4fn9tBb6dNqcmzU5L/qw
IFAGbHrQgLKm+a/sRxmPUDgH3KKHOVj4utWp+UhnMJbulHheb4mjUcAwhmahRWa6
VOujw5H5SNz/0egwLX0tdHA114gk957EWW67c4cX8jJGKLhD+rcdqsq08p8kDi1L
93FcXmn/6pUCyziKrlA4b9v7LWIbxcceVOF34GfID5yHI9Y/QCB/IIDEgEw+OyQm
jgSubJrIqg0CAwEAAaNCMEAwDwYDVR0TAQH/BAUwAwEB/zAOBgNVHQ8BAf8EBAMC
AYYwHQYDVR0OBBYEFIQYzIU07LwMlJQuCFmcx7IQTgoIMA0GCSqGSIb3DQEBCwUA
A4IBAQCY8jdaQZChGsV2USggNiMOruYou6r4lK5IpDB/G/wkjUu0yKGX9rbxenDI
U5PMCCjjmCXPI6T53iHTfIUJrU6adTrCC2qJeHZERxhlbI1Bjjt/msv0tadQ1wUs
N+gDS63pYaACbvXy8MWy7Vu33PqUXHeeE6V/Uq2V8viTO96LXFvKWlJbYK8U90vv
o/ufQJVtMVT8QtPHRh8jrdkPSHCa2XV4cdFyQzR1bldZwgJcJmApzyMZFo6IQ6XU
5MsI+yMRQ+hDKXJioaldXgjUkK642M4UwtBV8ob2xJNDd2ZhwLnoQdeXeGADbkpy
rqXRfboQnoZsG4q5WTP468SQvvG5
-----END CERTIFICATE-----
```

Debemos copiar el texto y guardarlo en un archivo **.pem**:

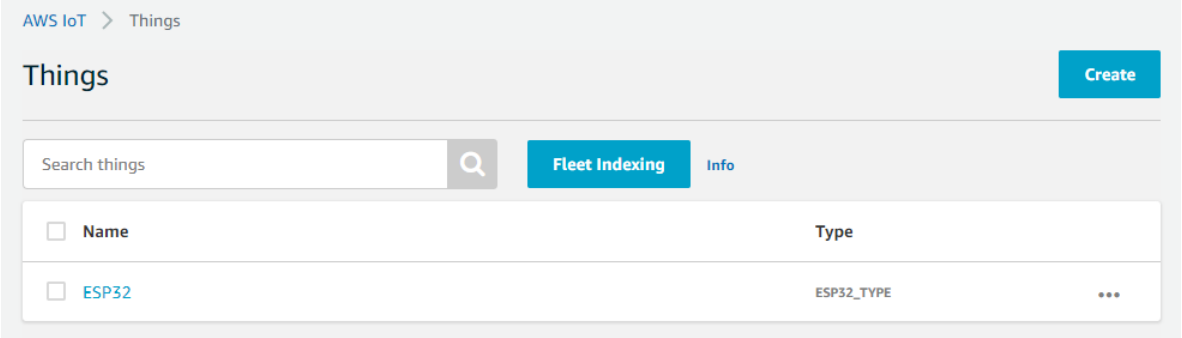
Para finalizar, volviendo a la ventana de AWS IoT, debemos tocar *Activate*.



Por último, la interfaz nos pedirá que le asignemos alguna *policy* a nuestro dispositivo. Omitimos este paso.



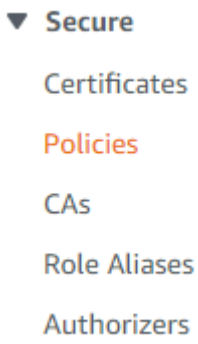
Como resultado, aparece nuestra *Thing* en nuestra lista de dispositivos:

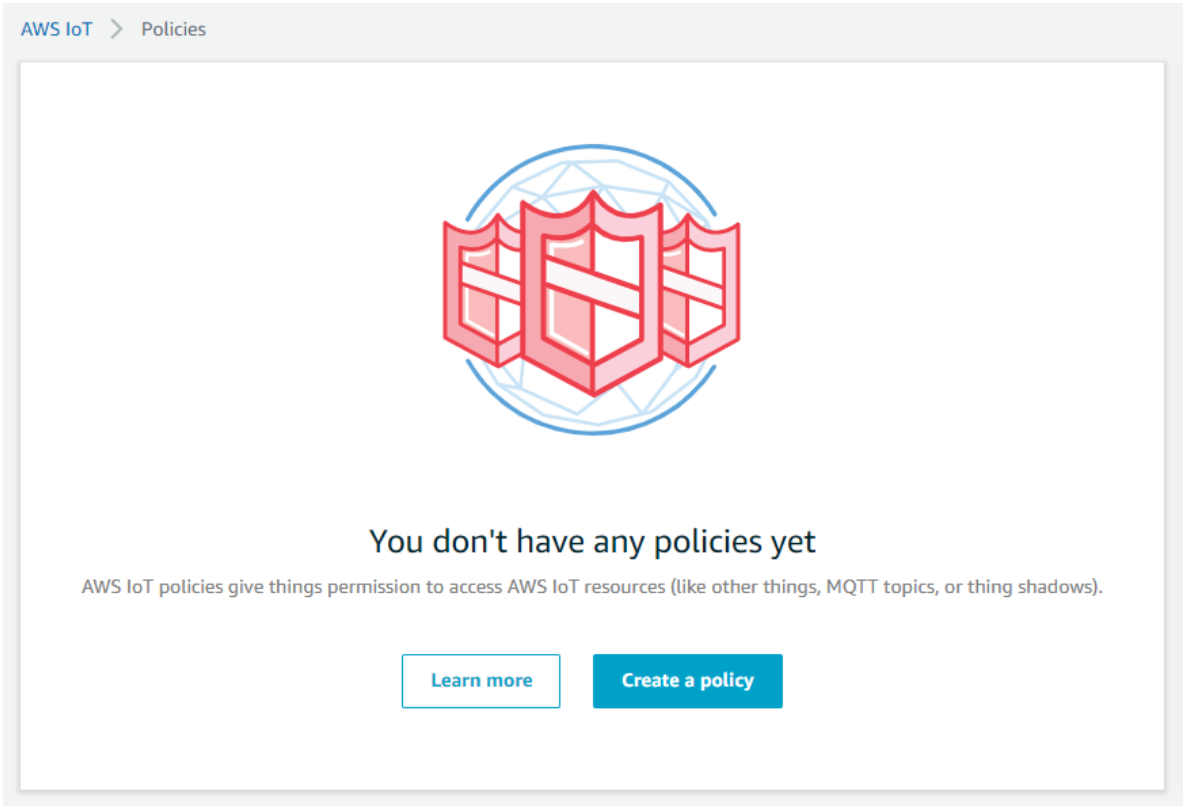


AWS IoT: Creación y asignación de políticas

Ahora debemos crear una política o *policy* que describa la forma en la que vamos a comunicarnos con el Gateway de AWS IoT. Podemos incluso crear una política general y aplicarla a un conjunto de dispositivos al mismo tiempo.

Para crear una nueva política debemos dirigirnos a *Secure – Policies* en el panel izquierdo.





Tocamos *Create a policy*.

A efectos de tener una política que nos permita operar con mucha flexibilidad en nuestra comunicación, en principio crearemos una que nos permita todo. La idea es no perder mucho tiempo limitando en los topics a los que el ESP32 puede publicar y a los que se puede suscribir, para correr el ejemplo rápidamente. Sin embargo, se

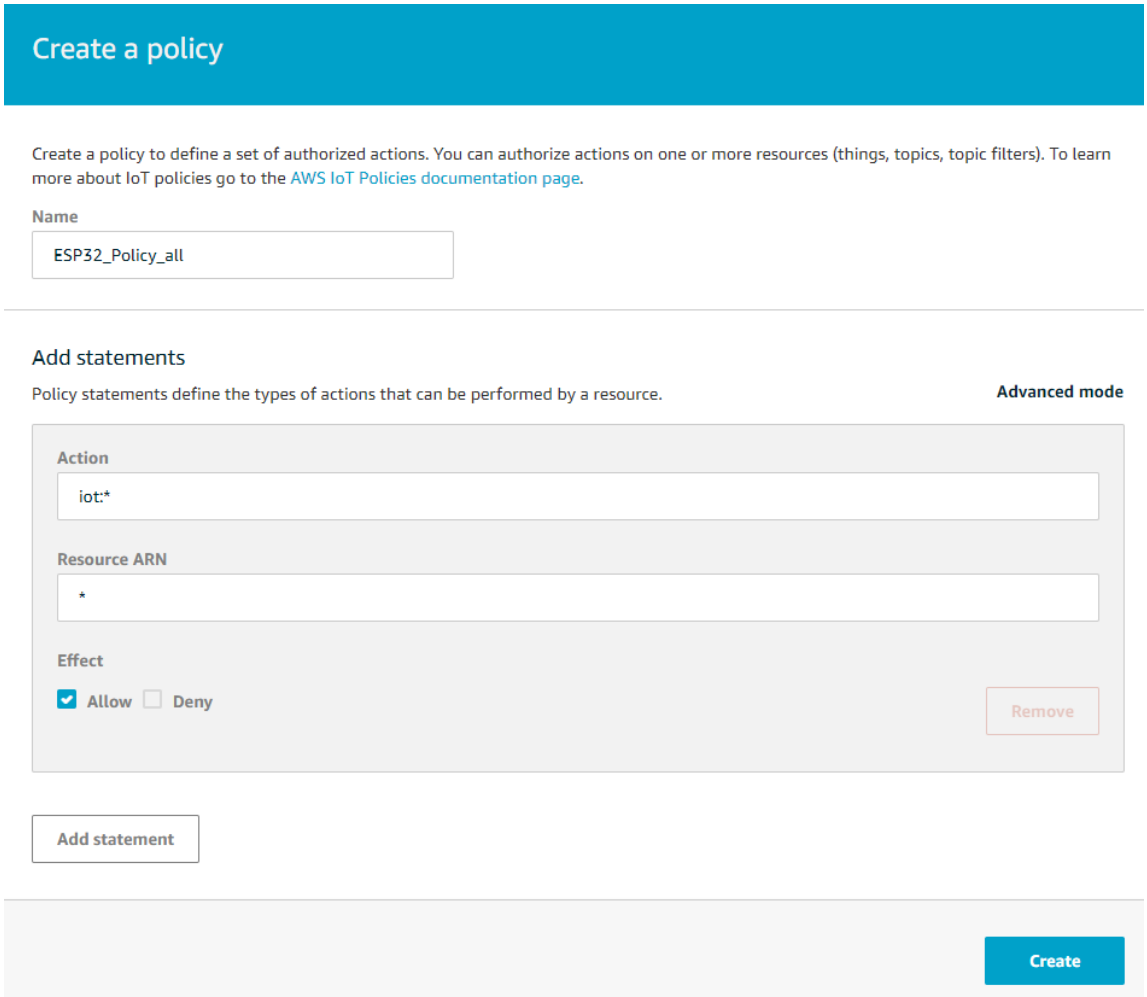
mostrará también cuál es la estructura de una política típica y bien definida para este caso de uso.

Política sin restricciones

Debemos completar los campos de la siguiente manera:

- **Name:** *ESP_Policy_all*
  El nombre es arbitrario, pero nos sirve para recordarnos rápidamente el nivel de permisividad de la política.
- **Action:** *iot:**
- **Resource ARN:** *

Y debemos tocar la opción *Allow*. Finalizamos con *Create*.



Con esta configuración ya podemos continuar con la asignación de políticas al dispositivo. La siguiente sección, donde se detalla la estructura de una política general, es opcional.

Política con acciones definidas

A continuación, se describirá brevemente la estructura de una política general para limitar acciones permitidas y el acceso a los recursos.

Una política es un documento JSON que se compone de dos elementos: la versión y las declaraciones o *statements*:

```
{
  "Version": "[FECHA]",
  "Statement":[LISTA DE AUTORIZACIONES]
}
```

La versión consiste únicamente de la fecha de creación y los *statements* son cada una de las acciones permitidas o no permitidas, con sus correspondientes ARNs (Amazon Resource Names).

Cada uno de los *statements* se compone de los siguientes campos:

- **Effect:** *Allow* ó *Deny*
- **Action:** *iot:Connect, iot:Publish, iot:Subscribe* ó *iot:Receive,* entre otros.
- **Resource:** *arn.* Su estructura es la siguiente:

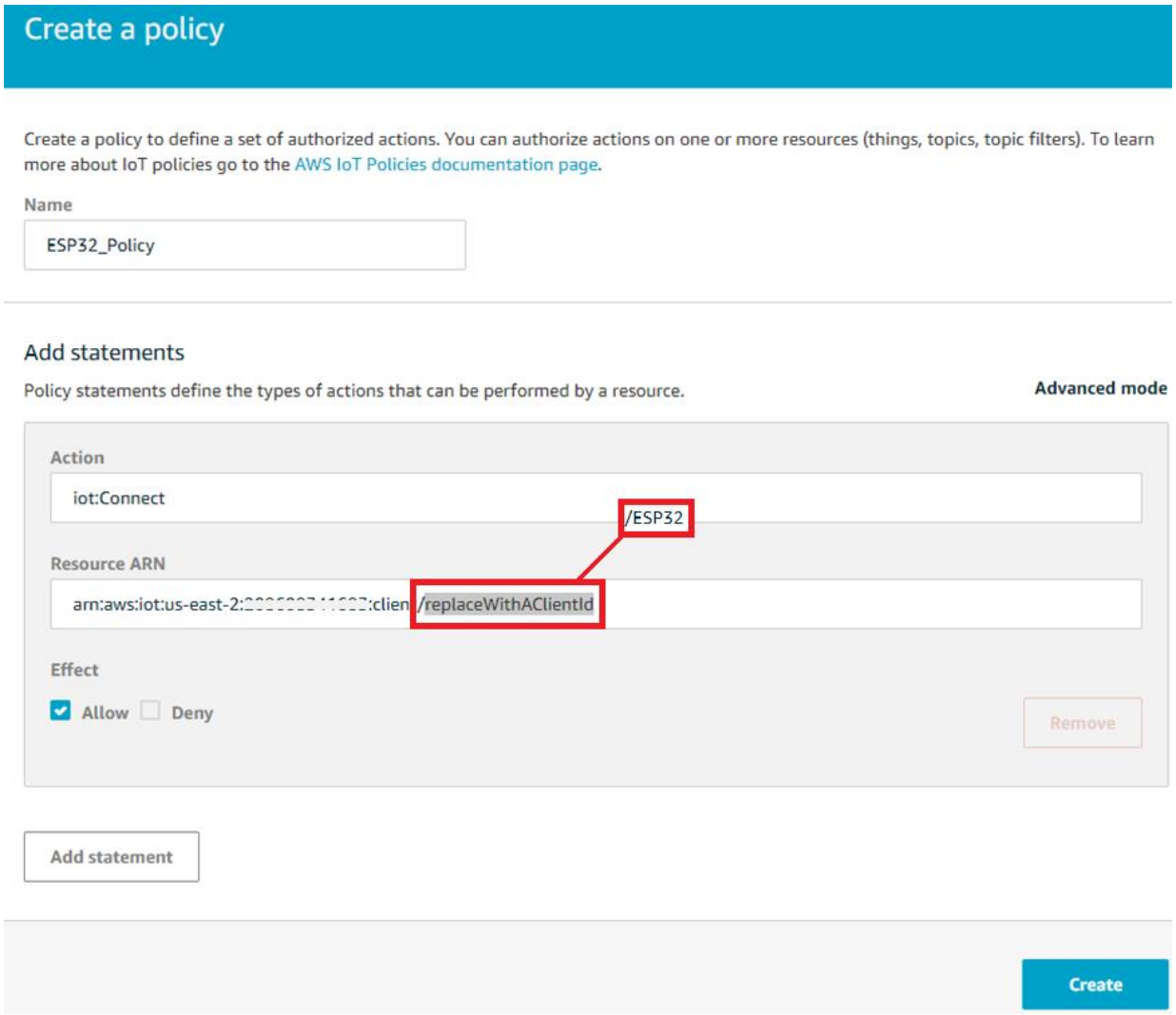
```
arn:aws:iot:[region]:[AWS account ID]:[resource type]/[resource ID]
```

Todos estos campos se generan automáticamente al crear la política en modo básico. Es importante saber que las declaraciones están ordenadas por prioridad, siendo la primera de mayor prioridad.

Procedemos entonces a crear la política:

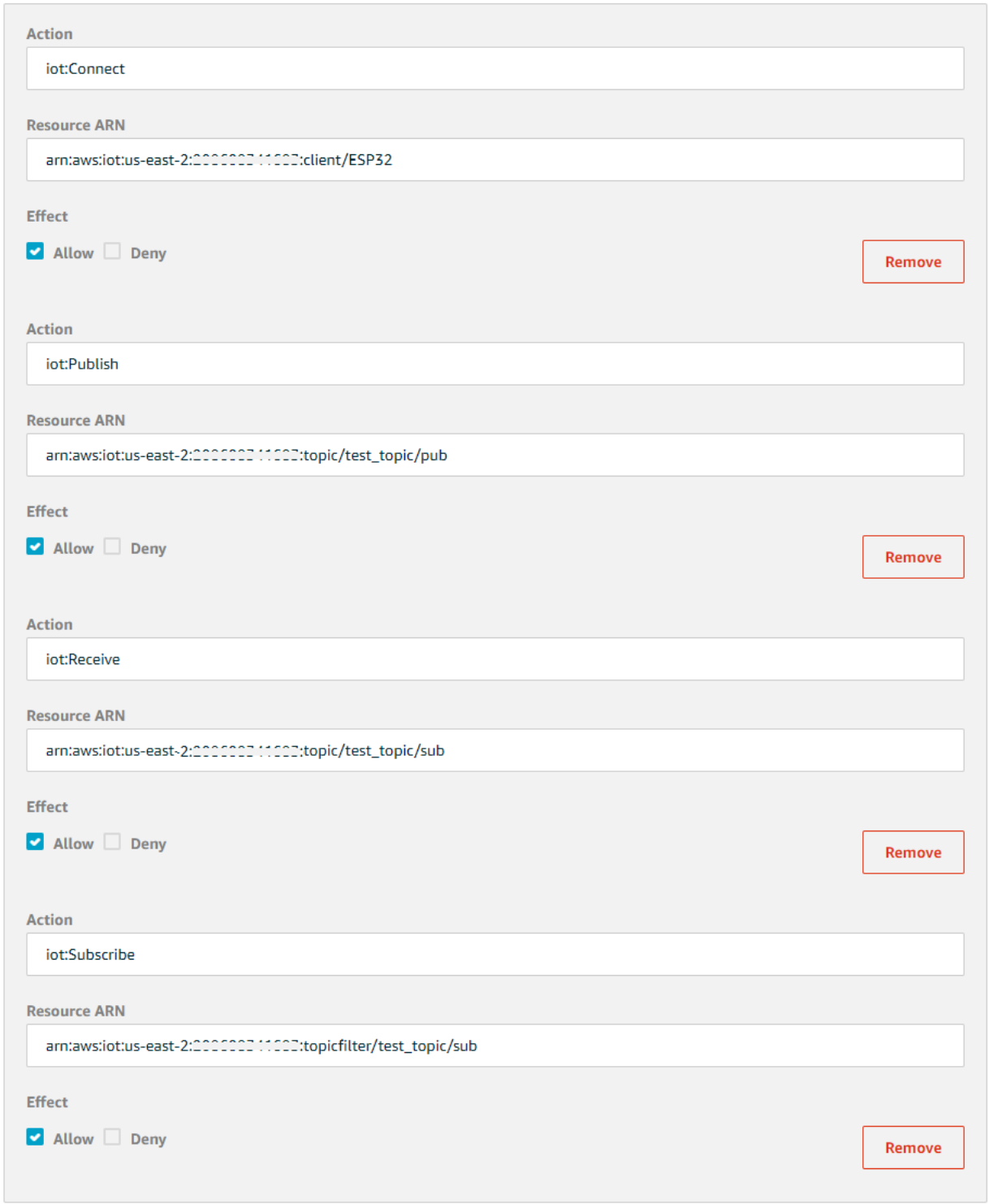
Como antes, completamos el campo de nombre, y luego los de acción y ARN.

- **Name:** *ESP_Policy*
- **Action:** *iot:Connect*
- **Resource ARN:** Aquí debemos dejar el ARN que nos autocompleta la interfaz, pero debemos cambiarle la última parte. Debemos introducir el nombre del dispositivo que dimos de alta en la anterior etapa. En este caso, el nombre es "**ESP32**"



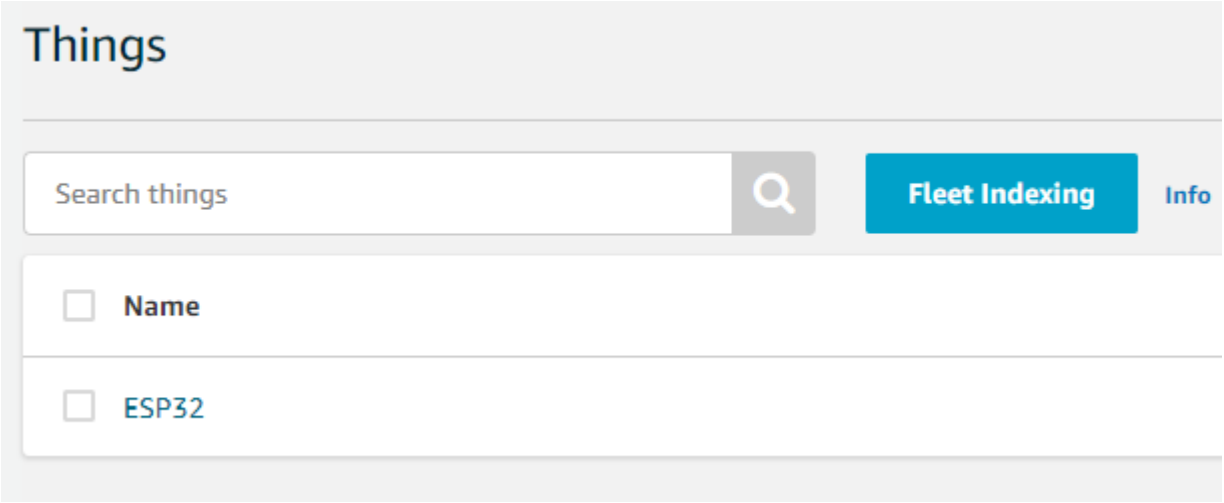
Debemos dejar la casilla *Allow* marcada y luego tocar *Add statement* para seguir con cada una de las restantes autorizaciones:

Usaremos el topic **`test_topic/pub`** para hacer las publicaciones y nos suscribiremos al topic **`test_topic/sub`**. **Estos mismos topics serán los que fijaremos a la hora de programar el ESP32.**

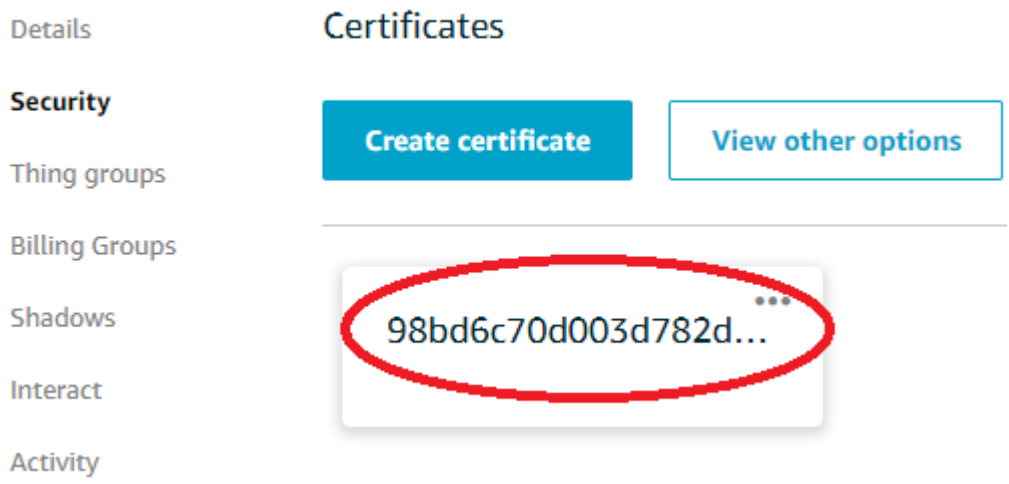

Action
iot:Connect
Resource ARN
arn:aws:iot:us-east-2:[redacted]:client/ESP32
Effect
Allow
Deny
Remove
Action
iot:Publish
Resource ARN
arn:aws:iot:us-east-2:[redacted]:topic/test_topic/pub
Effect
Allow
Deny
Remove
Action
iot:Receive
Resource ARN
arn:aws:iot:us-east-2:[redacted]:topic/test_topic/sub
Effect
Allow
Deny
Remove
Action
iot:Subscribe
Resource ARN
arn:aws:iot:us-east-2:[redacted]:topicfilter/test_topic/sub
Effect
Allow
Deny
Remove

Asignación de política

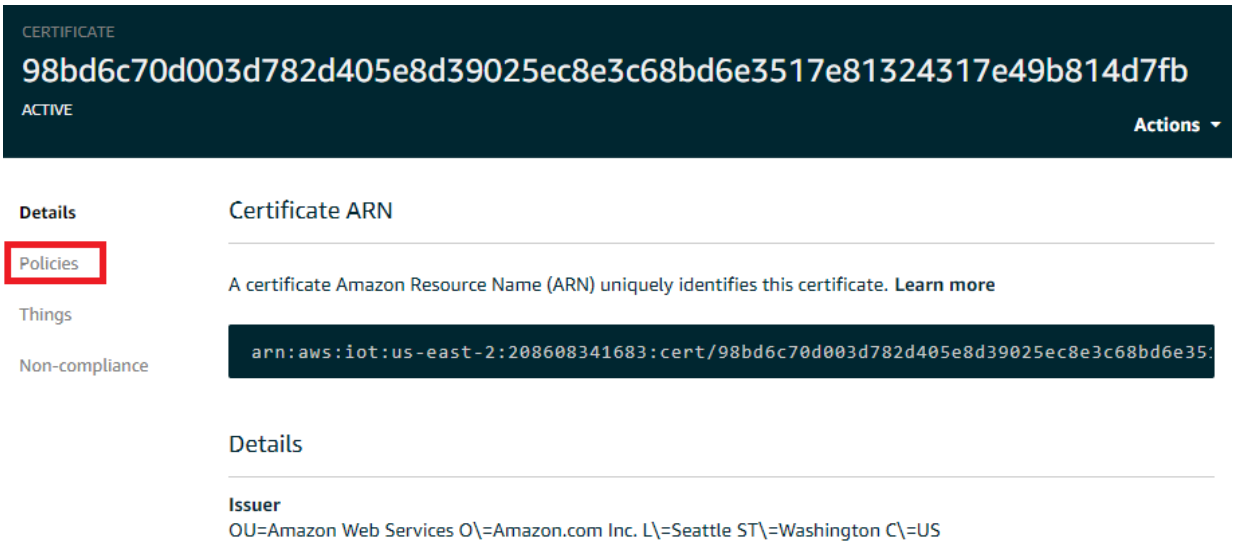
Finalmente, debemos asignarle la política recién creada a nuestro dispositivo. Para ello, nos dirigimos a Manage – Things y hacemos click en nuestro dispositivo.



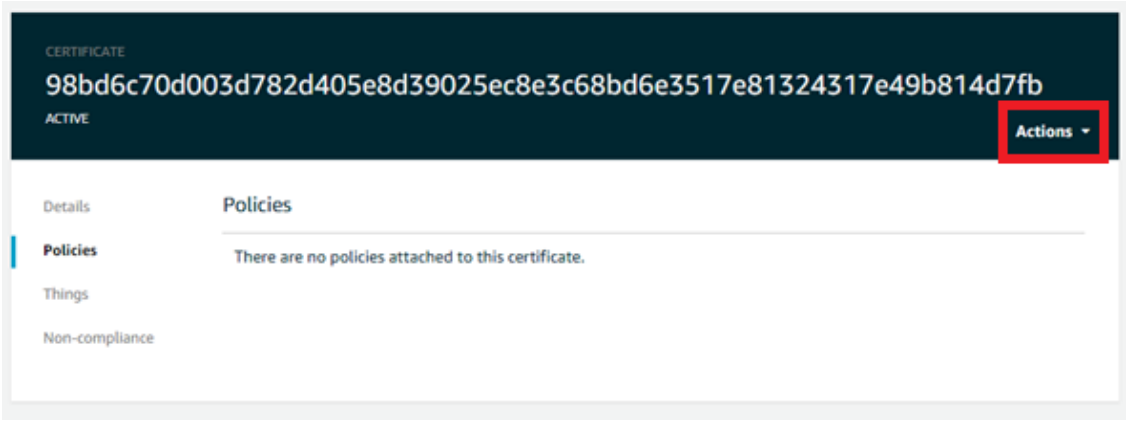
Debemos seleccionar *Security* y luego hacer click en certificado creado al principio.



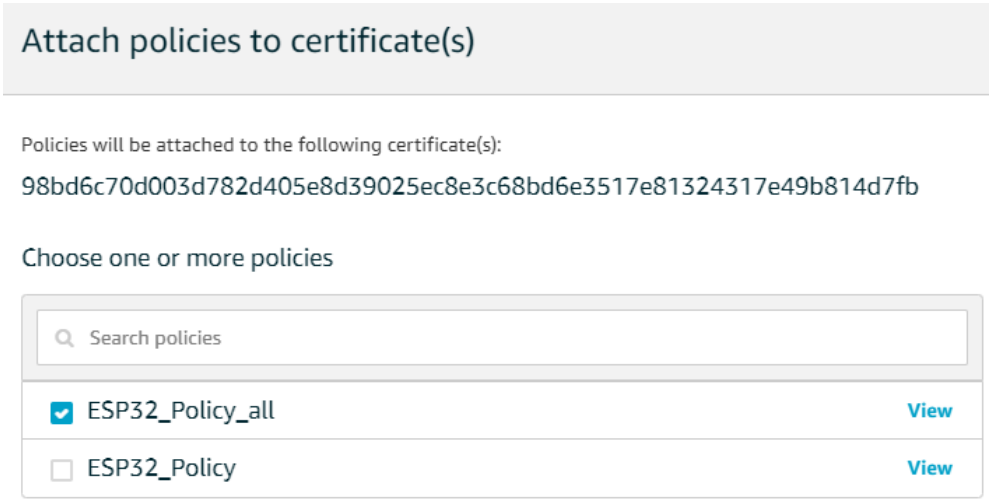
A continuación, nos dirigimos a *Policies*



Debemos entonces adjuntar una política tocando en *Actions – Attach Policy*



Después, debemos seleccionar qué política aplicar.



Tocamos *Attach* y con esto finalizamos la configuración desde el lado de AWS IoT.

ESP32: Proyecto MQTT con Autenticación Mutua

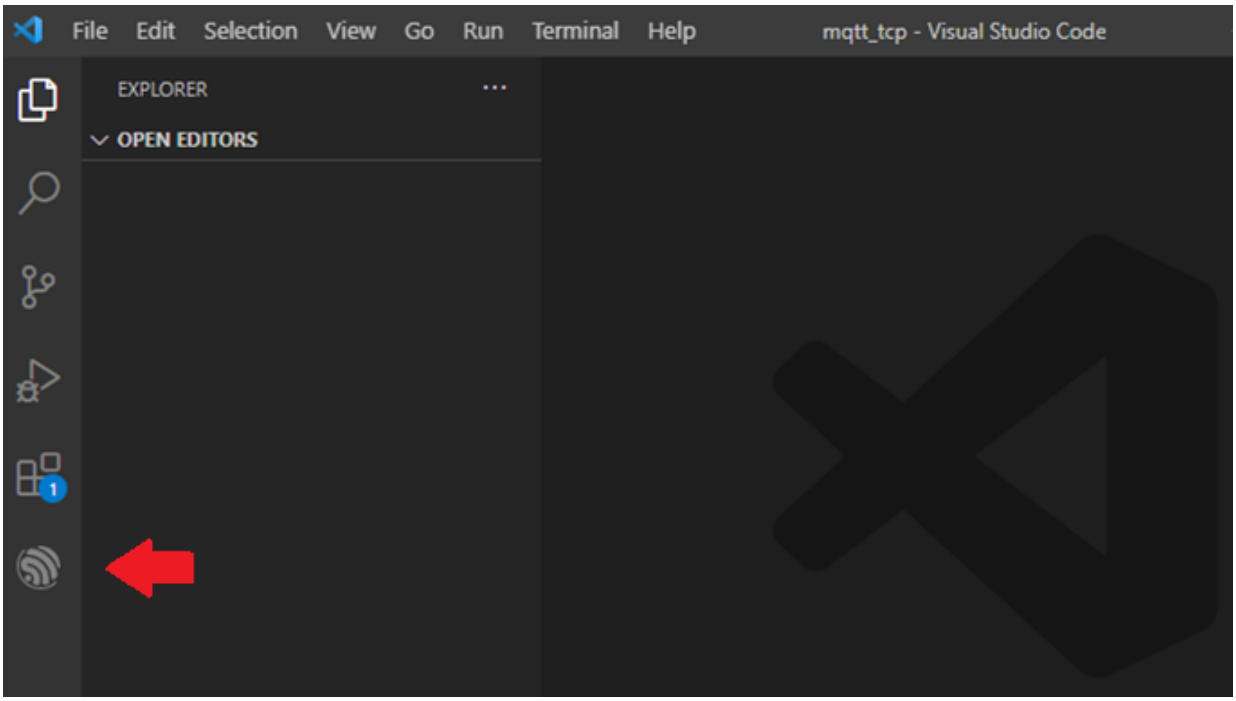
A continuación, veremos cómo aprovechar los códigos de ejemplo que incorpora el framework ESP-IDF de Espressif para lograr una comunicación MQTT sobre SSL con AWS IoT. Para ello usaremos VS Code como editor y la extensión ESP-IDF.

Es importante destacar que, para este instructivo, usaremos la versión 0.61 de la extensión mencionada y además la versión de ESP-IDF será la 4.3 (para versiones anteriores, el proyecto que queremos importar no está completo, por lo que no lo podremos usar).

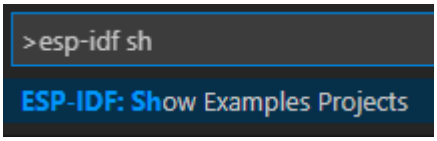
Código de ejemplo "ESP-MQTT SSL Sample Application"

Para nuestra conexión con AWS IoT usaremos un código de ejemplo usado para demostrar el uso de MQTT sobre SSL y un bróker mosquitto. Vamos a adaptarlo según lo configurado en Amazon.

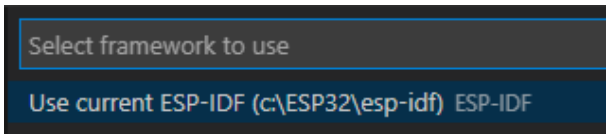
Primero, necesitamos abrir VS Code e inicializar la extensión haciendo click en su ícono:



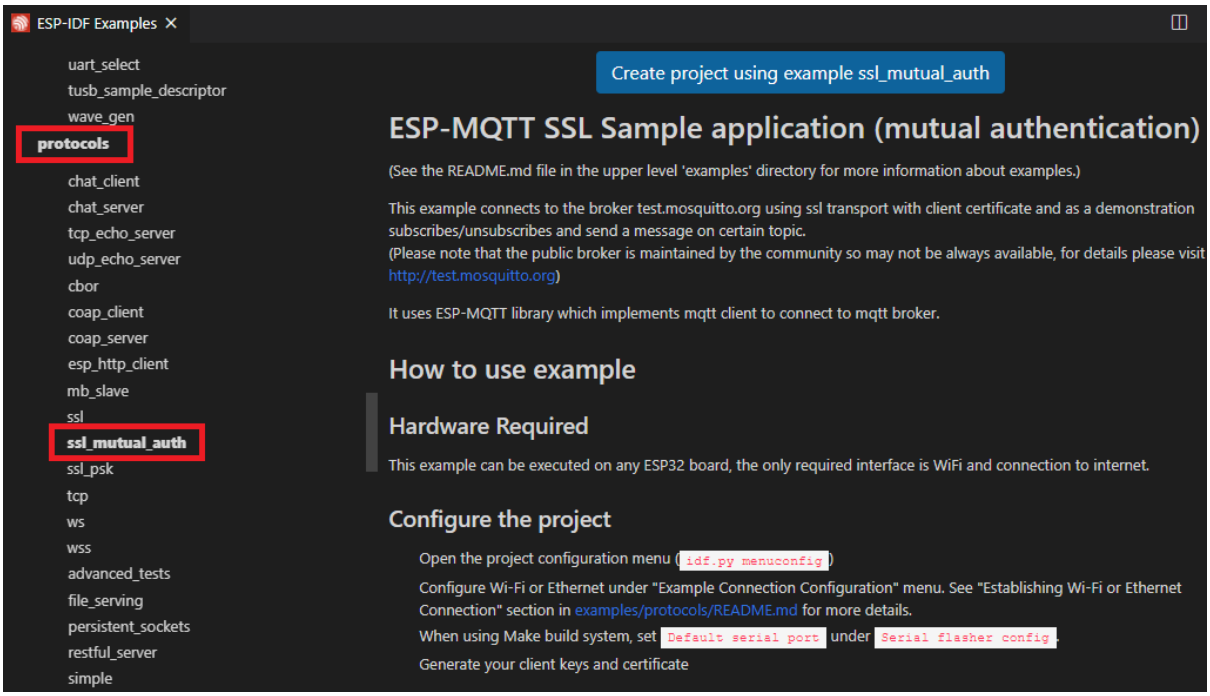
A continuación, debemos abrir los proyectos de ejemplo abriendo la paleta de comandos con CTRL+SHIFT+P y escribiendo *ESP-IDF show example projects:*



E indicamos que el framework a utilizar es el único que tenemos configurado:

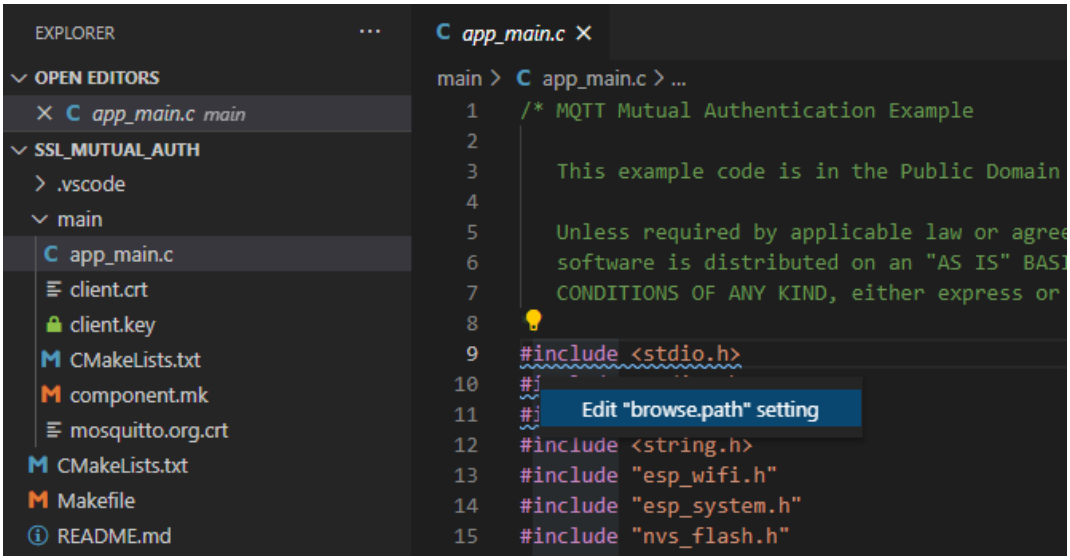


Luego, buscar el proyecto "ssl_mutual_auth" bajo la sección *Protocols*, en la parte izquierda:



Debemos tocar *Create project using example ssl_mutual_auth* y luego seleccionar una carpeta que usaremos como workspace.

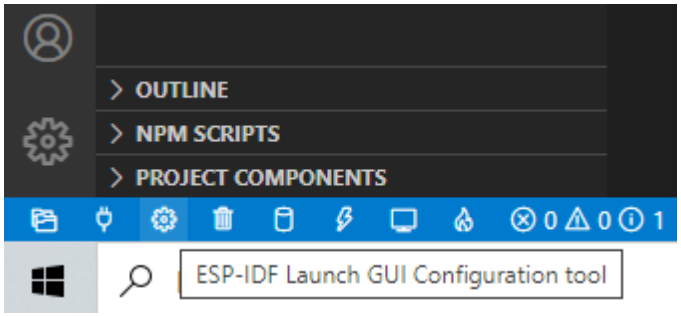
Una vez hecho esto, vamos al archivo `app_main.c` y resolvemos el problema de los include como se indicó anteriormente.



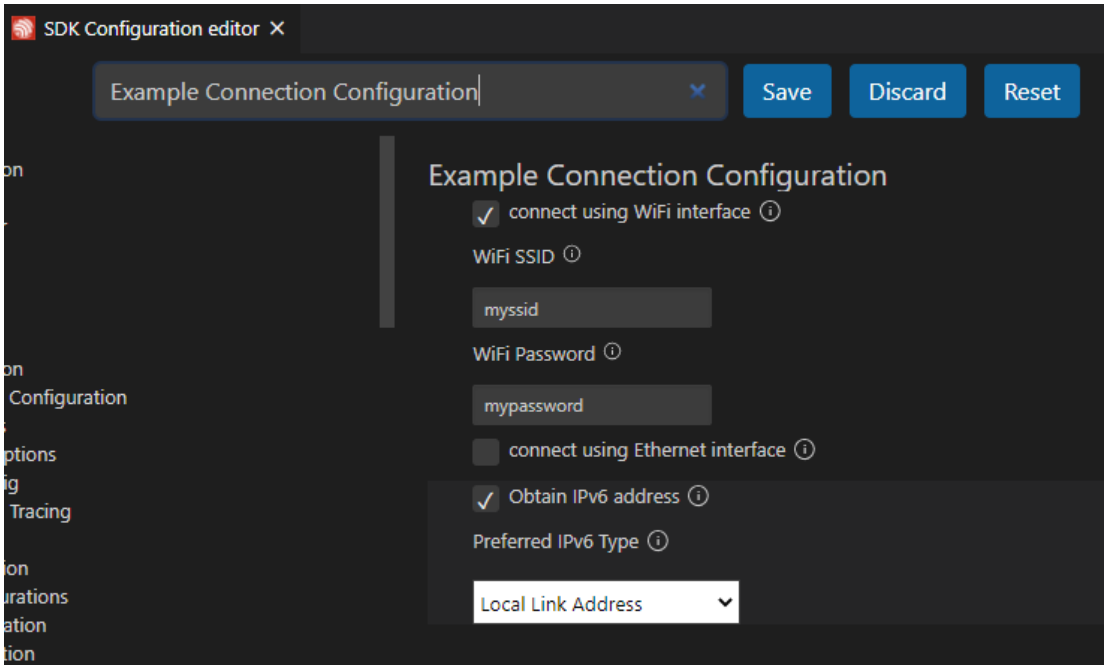
Configuración de parámetros de conexión y broker MQTT

Wi-Fi

En primer lugar, debemos configurar los parámetros de conexión a internet. Para ello, debemos ir a la parte inferior izquierda de la ventana y tocar el botón de *Launch GUI Configutarion tool.*



Buscamos *Example Connection Configuration*



Acá debemos configurar el SSID y la contraseña de la red WiFi que vayamos a usar para hacer nuestras pruebas. Al terminar, tocar *Save*.

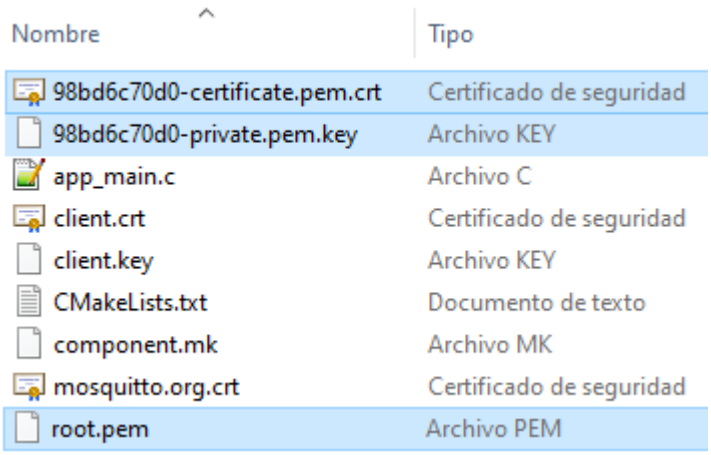
Credenciales

En este momento debemos usar los archivos antes descargados con los certificados y las claves para poder realizar la autenticación de nuestro ESP32 frente a AWS IoT.

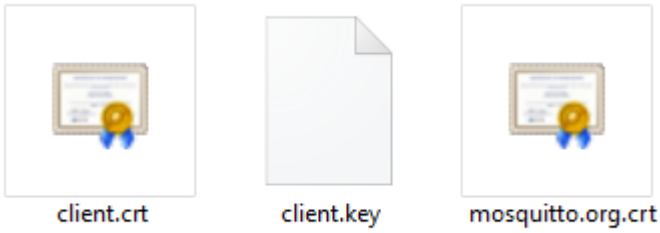
Vamos a necesitar tres archivos:

- El certificado del dispositivo: Archivo de nombre **`XXXX-certificate.pem.crt`**
- La clave privada del dispositivo: **`XXXX-private.pem.key`**
- El certificado CA de Amazon: **`root.pem`**

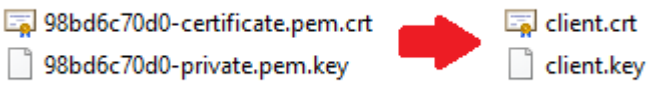
Debemos copiar estos archivos a la carpeta **main** de nuestro proyecto:



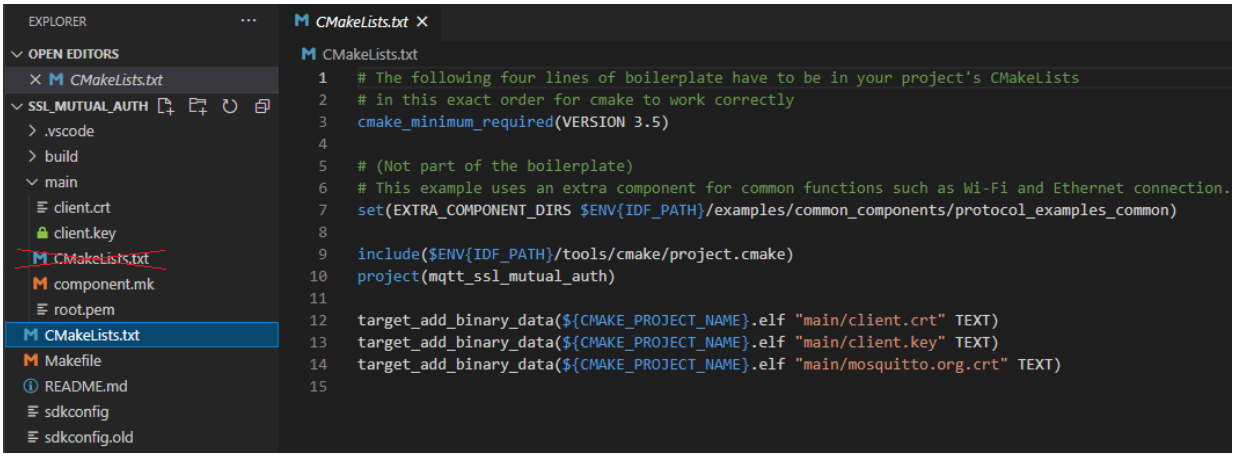
Notar que en la carpeta ya disponemos de unas claves y certificados propios del código de ejemplo. Vamos a reemplazarlas por los nuestros, por lo que podemos directamente **borrar** los archivos **`client.crt`**, **`client.key`** y **`mosquitto.org.crt`**



A continuación, **renombramos** los archivos de certificado del dispositivo y el de clave privada, por los nombres **`client.crt`** y **`client.key`**, respectivamente.



Como siguiente paso, debemos editar el archivo **CMakeLists.txt** que se encuentra en la carpeta principal del proyecto (no debe confundirse con el archivo del mismo nombre dentro de la carpeta main)



Aquí podemos ver que las sentencias **target_add_binary_data** se encargan de embeber los certificados y la clave privada en el código. Sólo debemos cambiar la última, para que pueda encontrar nuestro archivo de certificado CA **root.pem**.

Reemplazamos entonces "mosquitto.org.crt"

```
target_add_binary_data(${CMAKE_PROJECT_NAME}.elf "main/mosquitto.org.crt" TEXT)
```

por "root.pem"

```
target_add_binary_data(${CMAKE_PROJECT_NAME}.elf "main/root.pem" TEXT)
```

Guardamos y nos dirigimos al archivo **app_main.c**

```
34  extern const uint8_t client_cert_pem_start[] asm("_binary_client_crt_start");
35  extern const uint8_t client_cert_pem_end[] asm("_binary_client_crt_end");
36  extern const uint8_t client_key_pem_start[] asm("_binary_client_key_start");
37  extern const uint8_t client_key_pem_end[] asm("_binary_client_key_end");
38  extern const uint8_t server_cert_pem_start[] asm("_binary_mosquitto_org_crt_start");
39  extern const uint8_t server_cert_pem_end[] asm("_binary_mosquitto_org_crt_end");
```

Debemos editar las líneas 38 y 39. Tenemos que reemplazar **mosquitto_org_crt** por **root_pem:**

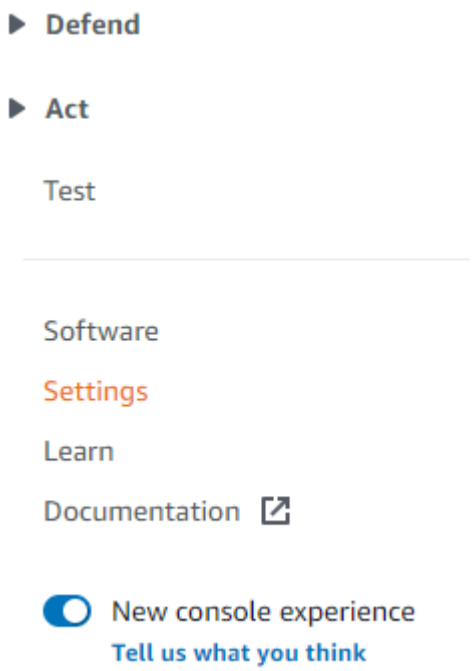
```
38  extern const uint8_t server_cert_pem_start[] asm("_binary_root_pem_start");
39  extern const uint8_t server_cert_pem_end[] asm("_binary_root_pem_end");
```

Parámetros de AWS IoT: Hostname, Port y Thing Name

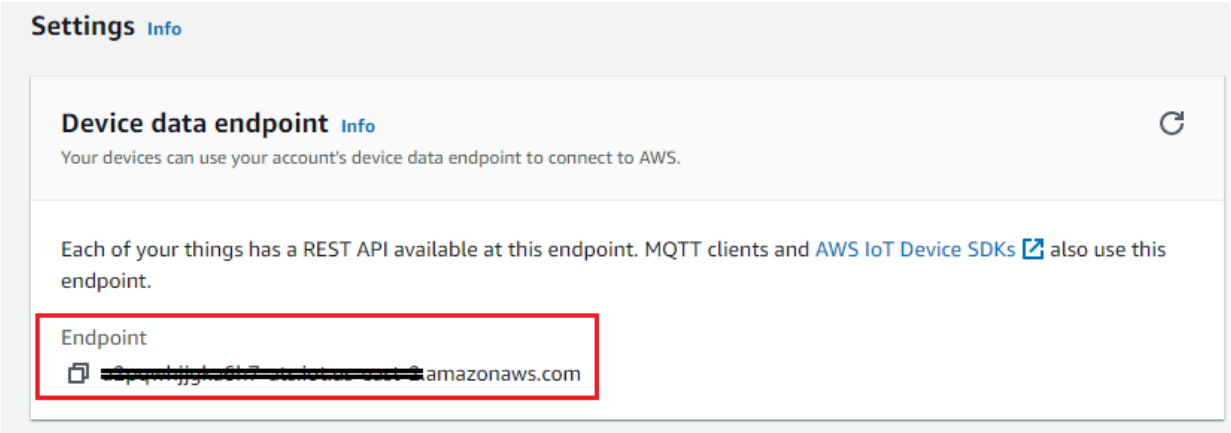
En este paso, debemos configurar los siguientes parámetros de AWS IoT en el ESP32:

- AWS IoT Hostname
- AWS IoT Port
- Thing Name
- Topics

Nos ocuparemos en primer lugar del **hostname** o ***Endpoint*** del lado de AWS. Podemos encontrarlo entrando a AWS IoT Core y dirigiéndonos a la sección *Settings* abajo a la izquierda:



Sólo debemos copiar la URL que nos indica bajo *Endpoint*



Volvemos al editor, y en el archivo **app_main.c** buscamos la línea 96:

```
93  static void mqtt_app_start(void)
94  {
95      const esp_mqtt_client_config_t mqtt_cfg = {
96          .uri = "mqtts://test.mosquitto.org:8884",
97          .client_cert_pem = (const char *)client_cert_pem_start,
98          .client_key_pem = (const char *)client_key_pem_start,
99          .cert_pem = (const char *)server_cert_pem_start,
100     };
```

Aquí debemos indicar en formato URI tanto la **dirección del *Endpoint*** de AWS como el **puerto** a utilizar. Por default, el puerto para MQTT en AWS IoT es el **8883**.

El formato debe ser el siguiente:

```
"mqtts://<URL_ENDPOINT>:<PUERTO>"
```

```
93  static void mqtt_app_start(void)
94  {
95      const esp_mqtt_client_config_t mqtt_cfg = {
96          .uri = "mqtts://[redacted].amazonaws.com:8883",
97          .client_cert_pem = (const char *)client_cert_pem_start,
98          .client_key_pem = (const char *)client_key_pem_start,
99          .cert_pem = (const char *)server_cert_pem_start,
100     };
```

El próximo parámetro que tenemos que setear es el ***Thing Name***.

En el entorno de ESP-IDF, este nombre se guarda en el campo **client_id** dentro de la estructura **mqtt_cfg.**

Simplemente agregamos la siguiente línea:

```
.client_id = "ESP32",
```

```
93  static void mqtt_app_start(void)
94  {
95      const esp_mqtt_client_config_t mqtt_cfg = {
96          .uri = "mqtts://[redacted].amazonaws.com:8883",
97          .client_cert_pem = (const char *)client_cert_pem_start,
98          .client_key_pem = (const char *)client_key_pem_start,
99          .cert_pem = (const char *)server_cert_pem_start,
100         .client_id = "ESP32",
101     };
```

Parámetros de AWS IoT: Topics

Finalmente, el último parámetro a configurar son los **topics**. Acá debemos prestar atención a qué tipo de política asignamos al dispositivo. Si le asignamos anteriormente una política que permite todo, en principio no necesitamos cambiar los topics que provee el ejemplo, ya que podrán ser usados de todas formas. En caso de haber definido topics para suscripción y publicación, debemos usar esos mismos topics en el código para que el ejemplo pueda funcionar.

Si miramos el archivo **`app_main.c`**, vemos que en la línea 41 comienza la función **`mqtt_event_handler_cb()`**, que es la que se encarga de manejar los eventos relacionados con la comunicación MQTT. La API de Espressif brinda diferentes funciones para suscribirse, desuscribirse y publicar a los topics.

```
41  static esp_err_t mqtt_event_handler_cb(esp_mqtt_event_handle_t event)
42  {
43      esp_mqtt_client_handle_t client = event->client;
44      int msg_id;
45      // your_context_t *context = event->context;
46      switch (event->event_id) {
47          case MQTT_EVENT_CONNECTED:
48              ESP_LOGI(TAG, "MQTT_EVENT_CONNECTED");
49              msg_id = esp_mqtt_client_subscribe(client, "/topic/qos0", 0);
50              ESP_LOGI(TAG, "sent subscribe successful, msg_id=%d", msg_id);
51
52              msg_id = esp_mqtt_client_subscribe(client, "/topic/qos1", 1);
53              ESP_LOGI(TAG, "sent subscribe successful, msg_id=%d", msg_id);
54
55              msg_id = esp_mqtt_client_unsubscribe(client, "/topic/qos1");
56              ESP_LOGI(TAG, "sent unsubscribe successful, msg_id=%d", msg_id);
57              break;
```

Analizando por ejemplo la función `esp_mqtt_client_publish()`, podemos ver que el topic es el segundo parámetro y de tipo cadena de caracteres.

int esp_mqtt_client_publish(esp_mqtt_client_handle_t *client*, *const* char **topic*, *const* char **data*, int *len*, int *qos*, int *retain*)

Client to send a publish message to the broker.

Return

message_id of the publish message (for QoS 0 message_id will always be zero) on success. -1 on failure.

Parameters

- `client`: **mqtt client handle**
- `topic`: **topic string**
- `data`: **payload string (set to NULL, sending empty payload message)**

- `len`: **data length, if set to 0, length is calculated from payload string**
- `qos`: **qos of publish message**
- `retain`: **retain flag**

Podemos entonces modificar las funciones para que se suscriban o publiquen a los topics que queramos. Para este ejemplo, usaremos los topics **`test_topic/sub`** y **`test_topic/pub`**.

El código modificado queda así:

```
41  static esp_err_t mqtt_event_handler_cb(esp_mqtt_event_handle_t event)
42  {
43      esp_mqtt_client_handle_t client = event->client;
44      int msg_id;
45      // your_context_t *context = event->context;
46      switch (event->event_id) {
47          case MQTT_EVENT_CONNECTED:
48              ESP_LOGI(TAG, "MQTT_EVENT_CONNECTED");
49              msg_id = esp_mqtt_client_subscribe(client, "test_topic/sub", 0);
50              ESP_LOGI(TAG, "sent subscribe successful, msg_id=%d", msg_id);
51
52          case MQTT_EVENT_DISCONNECTED:
53              ESP_LOGI(TAG, "MQTT_EVENT_DISCONNECTED");
54              break;
55
56          case MQTT_EVENT_SUBSCRIBED:
57              ESP_LOGI(TAG, "MQTT_EVENT_SUBSCRIBED, msg_id=%d", event->msg_id);
58              msg_id = esp_mqtt_client_publish(client, "test_topic/pub", "Hello from ESP32!", 0, 0, 0);
59              ESP_LOGI(TAG, "sent publish successful, msg_id=%d", msg_id);
60              break;
```

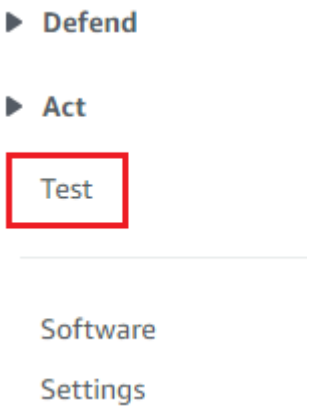
Con esto, ya podemos compilar el proyecto y bajarlo al microcontrolador para probarlo.

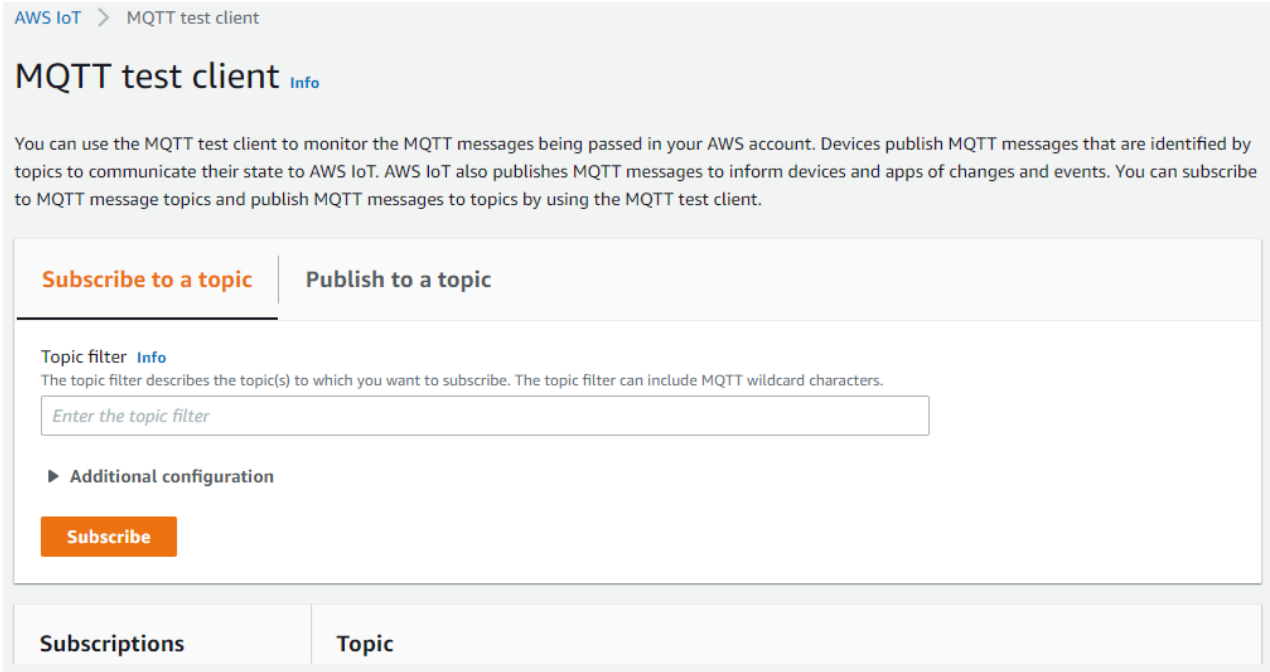
ESP32: Probando el código

Estamos ya en condiciones de probar la comunicación MQTT. Debemos compilar y grabar el código al microcontrolador, y luego abrir una terminal serie para leer los datos que enviaremos desde Amazon.

Antes de iniciar el proceso, es conveniente tener preparado el monitor MQTT desde Amazon.

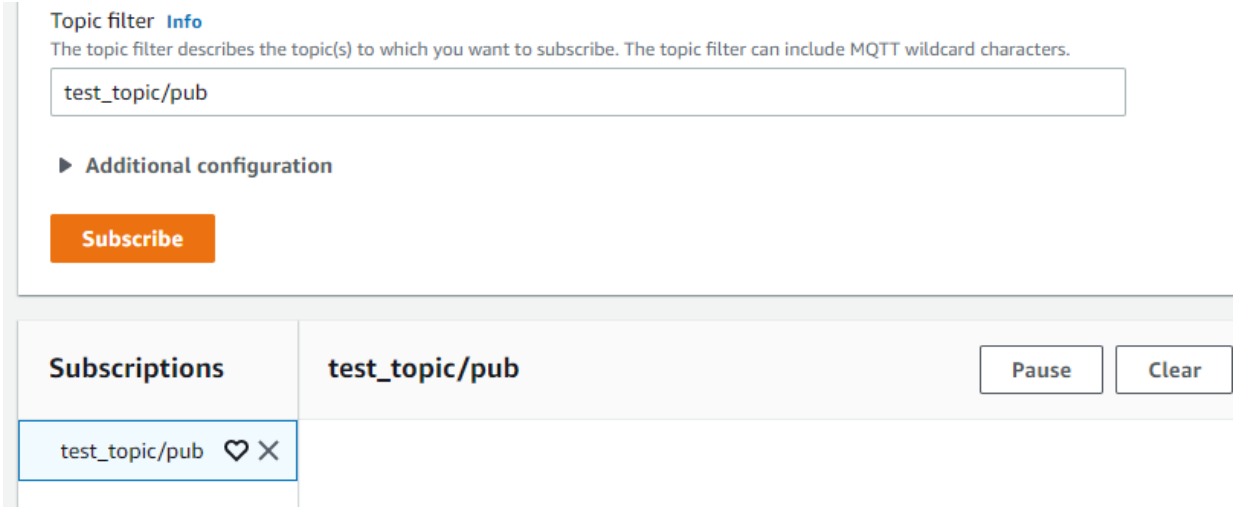
Para ello, nos dirigimos a la sección *Test*, en el panel izquierdo.





Primero vamos a subscribirnos al topic **`test_topic/pub`** escribiéndolo en *Topic Filter* y luego tocado *Subscribe*.

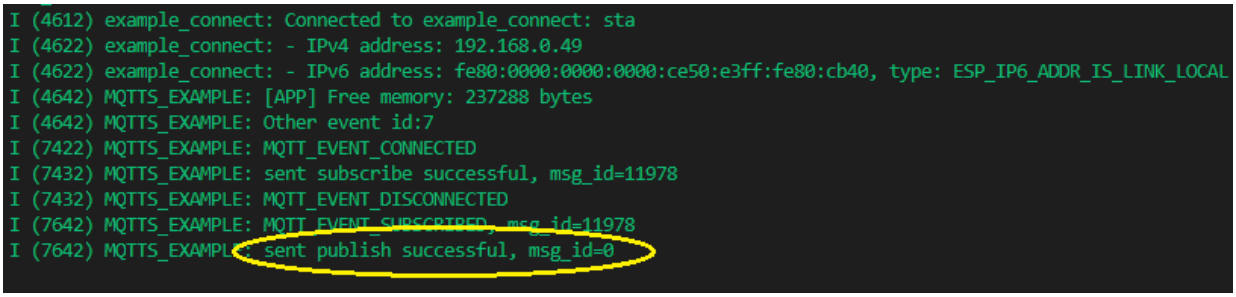
Los mensajes aparecerán en la parte inferior:



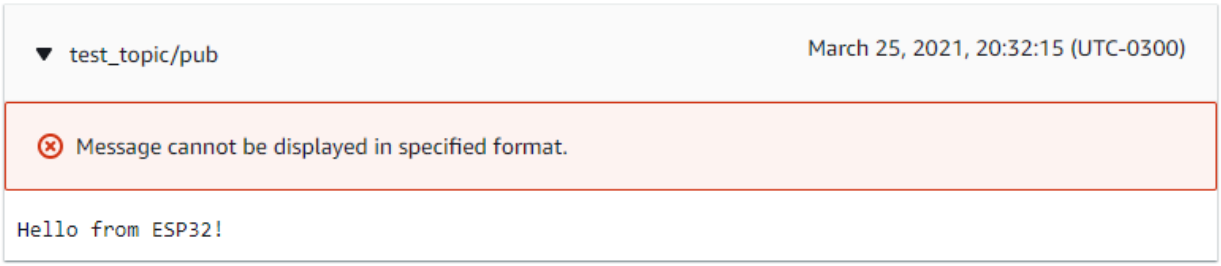
Podemos ahora iniciar el flasheo. Vamos a VS Code y en la parte inferior izquierda, buscamos este ícono:

Esto compilará el código, lo bajará al microcontrolador y luego abrirá el monitor serial automáticamente.

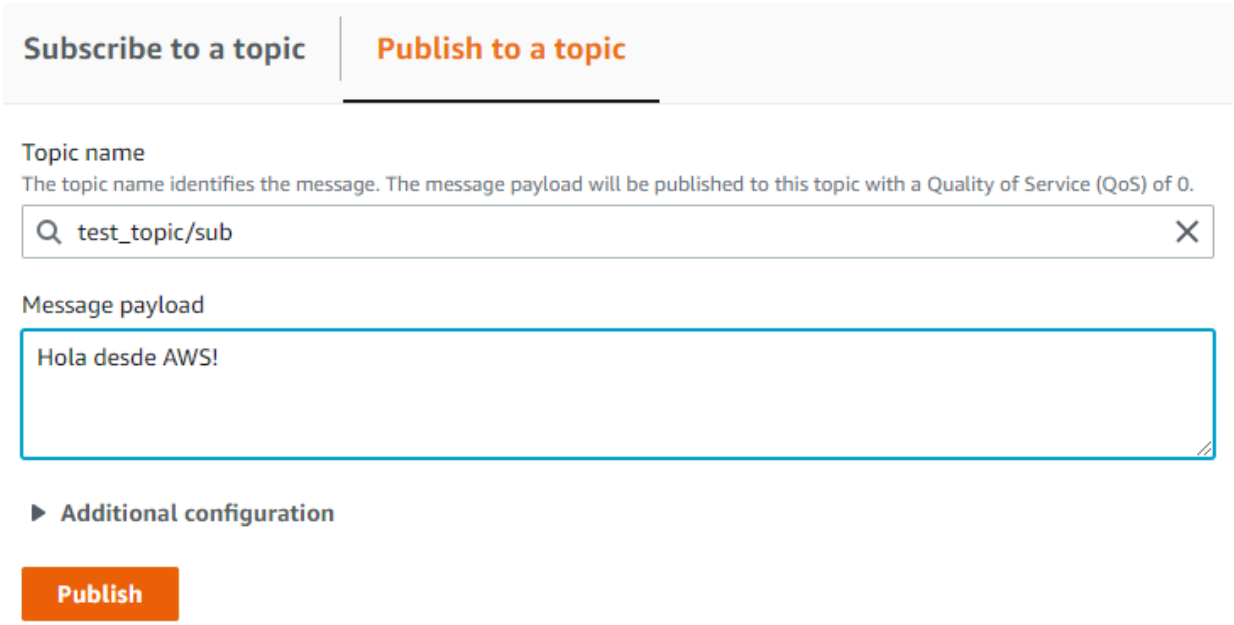
Pasados unos segundos, veremos esto:



Nos notifica que el mensaje fue publicado con éxito. Si vamos al monitor de AWS, podemos ver el mensaje que nos envió el ESP32:



Ahora vamos a enviar un mensaje desde AWS. Vamos a *Publish to a topic* y escribimos un mensaje para el topic **`test_topic/sub`** y lo publicamos:



Lo podemos ver inmediatamente en la consola del VS Code:

```
I (7642) MQTTS_EXAMPLE: sent publish successful, msg_id=0
I (329672) MQTTS_EXAMPLE: MQTT_EVENT_DATA
TOPIC=test_topic/sub
DATA=Hola desde AWS!
```

Y con esto terminamos de probar el código de ejemplo. Podemos ahora modificarlo a gusto, para adaptarlo a nuestras necesidades e incorporarlo a la aplicación que deseemos implementar.

Fuentes

YouTube - Quick User Guide for the ESP-IDF VS Code Extension

Espressif - ESP-IDF Programming Guide

Espressif – API Reference – ESP-MQTT

YouTube – Connecting ESP32 to AWS IoT Core

YouTube - ESP-IDF WITH AWS IOT SUBSCRIBE AND PUBLISH EXAMPLE

Blog - ESP32 and AWS IoT Tutorial

Github - ESP-MQTT SSL Sample application

Espressif – API Guides – Embedding Binary Data

Blog - Connect ESP32 to AWS IoT (with Arduino code)

AWS IoT Core Developer Guide

Github – ESP-AWS-IOT