Chapter 3: Amazon Web Services (AWS) IoT Core

After completing this chapter, you will understand the AWS IoT Core services, MQTT, and how to create Things, Policies, and Certificates.

3.1 Introduction

Whether you use AnyCloud, Amazon FreeRTOS, Mbed, or your own solution, at the heart of it your device will connect and communicate using the AWS IoT Core using MQTT. This chapter will discuss some of the concepts that are important to know when connecting your IoT device to AWS.

3.2 MQTT

MQTT is a lightweight messaging protocol that allows a device to **Publish Messages** to a specific **Topic** on a **Message Broker**. The Message Broker will then relay the message to all devices that are **Subscribed** to that **Topic**.

The format of the messages being sent in MQTT is unspecified. The message broker does not know (or care) anything about the format of the data and it is up to the system designer to specify an overall format of the data. All that being said, JavaScript Object Notation (JSON) has become the lingua franca of IoT.

A Topic is simply the name of a message queue e.g. `mydevice/status` or `mydevice/pressure`. The name of a topic can be almost anything you want but by convention is hierarchical and separated with forward slashes.

Publishing is the process by which a client sends a message as a blob of data to a specific topic on the message broker.

A Subscription is the request by a device to have all messages published to a specific topic sent to the client.

A Message Broker is just a server that handles the tasks:

- Establishing connections (MQTT Connect)
- Tearing down connections (MQTT Disconnect)
- Accepting subscriptions to a Topic from clients (MQTT Subscribe)
- Turning off subscriptions (MQTT Unsubscribe)
- Accepting messages from clients and pushing them to the subscribers (MQTT Publish)

MQTT provides three levels of Quality of Service (QOS):

- Level 0: At most once (each message is delivered once or never)
- Level 1: At least once (each message is certain to be delivered, possibly multiple times)
- Level 2: Exactly once (each message is certain to arrive and do so only once)

MQTT operates on TCP Ports 1883 for non-secure and 8883 for secure (TLS).

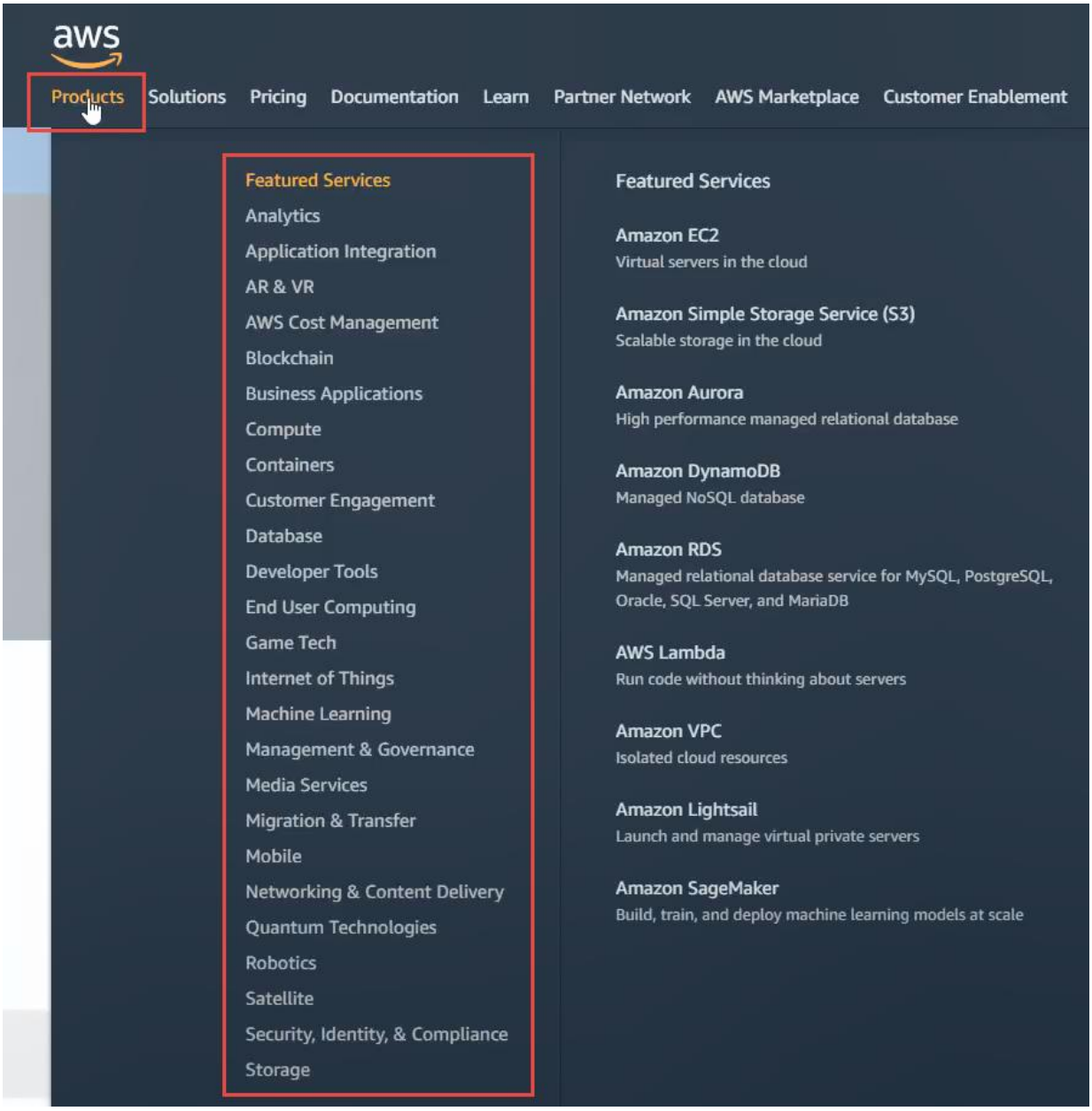
Cloud providers that support MQTT include Amazon AWS and IBM Bluemix.

3.3 Amazon Web Services

From https://aws.amazon.com/what-is-aws/

AWS is a secure cloud services platform, offering compute power, database storage, content delivery and other functionality (which makes more money for Amazon than their retail operations). AWS is built from a vast array of both virtual and actual servers and networks as well as a boatload of webserver software and administrative tools including the following.

You can find the complete list by going to https://aws.amazon.com and clicking on "Products" at the top left. There are a whole bunch of categories along the left, each of which has multiple products associated with it.



A partial list of some tools and services you may find useful:

- AWS IoT Core: A cloud platform that provides Cloud services for IoT devices (the subject of this chapter).
- Amazon Elastic Compute Cloud (Amazon EC2): A virtualized compute capability, basically Linux, Windows etc. servers that you can rent.
- AWS Lambda: A Cloud service that enables you to send event driven tasks to be executed.
- Storage: Large fast file systems called Amazon Simple Storage Service (S3) & Amazon Elastic File System.
- Databases: Large fast databases called Amazon DynamoDB, Amazon Relational Database Service (RDS), Amazon Aurora.
- Amazon Simple Notification Service: A platform to send messages including SMS and Email.
- Amazon Simple Queue Service: A platform to send messages between servers (NOT the same thing as MQTT messages).
- Amazon Kinesis: A platform to stream and analyze "massive" amounts of data. This is the plumbing for AWS IoT.
- Amazon Virtual Private Cloud: A way to provision a logically isolated section of the AWS cloud where you can launch AWS resources in a virtual network that you define.
- Amazon SageMaker: Machine learning for every developer and data scientist.
- Networking: Fast, fault tolerant, load balanced networks with entry points all over the world.
- Developer tools: A unified programming API supporting the AWS platform supporting a bunch of different languages.

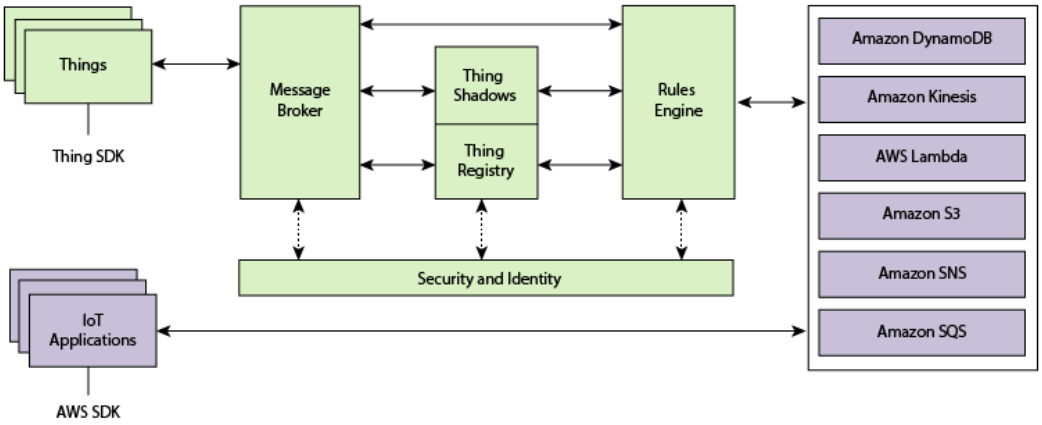
3.4 AWS IoT Introduction

The AWS IoT Cloud service supports MQTT Message Brokers, HTTP access, **plus** a bunch of server-side functionality that provides:

- Message Broker: A virtual MQTT Message Broker.
- A virtual HTTP Server.
- Thing Registry: A web interface to manage the access to your *things*.
- Security and identity: A web interface to manage the certificates and rules about *things*. You can create encryption keys and manage access privileges.
- A "Shadow": An online cache of the most recent state of your *thing***.**
- Rules Engine: An application that runs in the cloud that can subscribe to Topics and take programmatic actions based on messages – for example, you could configure it to subscribe to

an "Alert" topic, and if a *thing* publishes a warning message to the alert topic, it uses Amazon SNS to send a SMS Text Message to your cell phone

- IoT Applications: A solution to build Web pages and cell phone Apps.



3.5 AWS IoT Resources

There are three types of resources in AWS: *Things*, *Certificates*, and *Policies*. The second exercise will take you step by step through the process to create each of them.

3.5.1 Thing

A *thing* is a representation of a device or logical entity. It can be a physical device or sensor (for example, a light bulb or a switch on a wall). It can also be a logical entity like an instance of an application or a physical entity that does not connect to AWS IoT but can be related to other devices that do (for example, a car that has engine sensors or a control panel).

3.5.2 Certificate

AWS IoT provides mutual authentication and encryption at all points of connection so that data is never exchanged between *things* and AWS IoT without a proven identity. AWS IoT supports X.509 certificate-based authentication. Connections to AWS use certificate-based authentication. You should attach policies to a certificate to allow or deny access to AWS IoT resources. A root CA (certification authority) certificate is used by your device to ensure it is communicating with the actual Amazon Web Services site. You can only connect your *thing* to the AWS IoT Cloud via TLS.

3.5.3 Policy

When you create your internet-connected *thing*, you must create and attach an AWS IoT policy that will determine what AWS IoT operations the *thing* may perform. AWS IoT policies are JSON documents and they follow the same conventions as AWS Identity and Access Management policies.

You can specify permissions for specific resources such as topics and shadows. Here is an example of a Policy created for a new *thing* that allows any IoT action for any resource.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
           "Action": [  "iot:*"  ],
           "Resource": ["*"],
           "Effect": "Allow"
        }
    ]
}
```

3.6 Shadows

https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html

A *thing* shadow or device shadow is a JSON document that is used to store and retrieve current state information for a *thing* (device, app, and so on). The shadow service maintains a shadow for *thing* you connect to AWS IoT. You can use shadows to get and set the state of a *thing* over MQTT or HTTP, regardless of whether the *thing* is currently connected to the Internet. Each shadow is uniquely identified by its name.

The JSON Shadow document representing the device has the following properties:

- state:
    - desired: The desired state of the *thing*. Applications can write to this portion of the document to update the state of a *thing* without having to directly connect to a *thing*.
    - reported: The reported state of the *thing*. *Things* write to this portion of the document to report their new state. Applications read this portion of the document to determine the state of a *thing*.
- metadata: Information about the data stored in the state section of the document. This includes timestamps, in Epoch time, for each attribute in the state section, which enables you to determine when they were updated.
- timestamp: Indicates when the message was transmitted by AWS IoT. By using the timestamp in the message and the timestamps for individual attributes in the desired or reported section, a *thing* can determine how old an updated item is, even if it doesn't feature an internal clock.
- version: The document version. Every time the document is updated, this version number is incremented. Used to ensure the version of the document being updated is the most recent.

An example of a shadow document looks like this:

```
{
   "state" : {
      "desired" : {
         "color" : "RED"
         "sequence" : { "RED", "GREEN", "BLUE" }
      },
      "reported" : {
         "color" : "GREEN"
      }
   },
   "metadata" : {
      "desired" : {
         "color" : {
            "timestamp" : 12345
         },
         "sequence" : {
            "timestamp" : 12345
         }
      },
      "reported" : {
         "color" :
            "timestamp" : 12345
         }
      }
   },
   "version" : 10,
   "timestamp" : 123456789
}
```

If you want to update the Shadow, you can publish a JSON document with just the information you want to change to the correct topic. For example, you could do:

```
{
     "state" : {
          "reported" : {
               "color": "BLUE"
          }
     }
}
```

Note that spaces and carriage returns are optional, so the above could be written as:

```
{"state":{"reported":{"color": "BLUE"}}}
```

3.7 Topics

You can interact with AWS using either MQTT or HTTP. While topics are an MQTT concept, you will see later that topic names are important even when using HTTP to interact with shadows. The AWS Message Broker will allow you to create Topics with almost any name, with one exception: Topics named `$aws/…` are reserved by AWS IoT for specific functions.

As the system designer, you are responsible for defining what the topics mean and do in your system. Some best practices include:

1. Don't use a leading forward slash.
2. Don't use spaces.
3. Keep the topic short and concise.
4. Use only ASCII characters.
5. Embed a unique identifier e.g. the name of the *thing.*

For example, a good topic name for a temperature sensing device might be: `myDevice/temperature`.

3.8 Device Shadow MQTT Topics

https://docs.aws.amazon.com/iot/latest/developerguide/device-shadow-mqtt.html

Each *thing* that you have will have a group of topics of the form `$aws/things/<thingName>/shadow/<type>` which allow you to publish and subscribe to topics relating to the shadow. The specific shadow topics that exist are:

MQTT Topic Suffix <type>	Function
`/update`	The JSON message that you publish to this topic will become the new state of the shadow.
`/update/accepted`	AWS will publish a message to this topic in response to a message to `/update` indicating a successful update of the shadow.
`/update/documents`	When a document is updated via a publish to `/update`, the entire updated document is published to this topic.
`/update/rejected`	AWS will publish a message to this topic in response to a message to `/update` indicating a rejected update of the shadow.
`/update/delta`	After a message is sent to `/update`, the AWS will send a JSON message if the desired state and the reported state are not equal. The message contains all attributes that don't match.
`/get`	If a *thing* publishes a message to this topic, AWS will respond with a message to either `/get/accepted` or `/get/rejected` with the current state of the shadow.
`/get/accepted`	
`/get/rejected`	

MQTT Topic Suffix <type>	Function
`/delete`	If a *thing* publishes a message to this topic, AWS will delete the shadow document.
`/delete/accepted`	AWS will publish to this topic when a successful `/delete` occurs.
`/delete/reject`	AWS will publish to this topic when a rejected `/delete` occurs.

The update topic is useful when you want to update the state of a *thing* on the cloud. For example, if you have a *thing* called `myThing` and want to update a value called `temperature` to 25 degrees in the state of the *thing*, you would publish (for MQTT) or POST (for HTTP) using the following topic and message:

topic: `$aws/things/myThing/shadow/update`

message: `{"state":{"reported":{"temperature":25}}}`

Once the message is received, the MQTT message broker will publish to the `/accepted` and `/documents` topics with the appropriate information.

If you are using the MQTT test server to subscribe to topics, you can use the `#` character as a wildcard at the end of a topic to subscribe to multiple topics. For example, you can use `$aws/things/theThing/shadow/#` to subscribe to all shadow topics for the *thing* called `theThing`.

You can also use the plus sign (`+`) as a wildcard in the middle of a topic to subscribe to multiple topics. For example, you can use `$aws/things/+/shadow/update` to subscribe to update topics for all shadows.

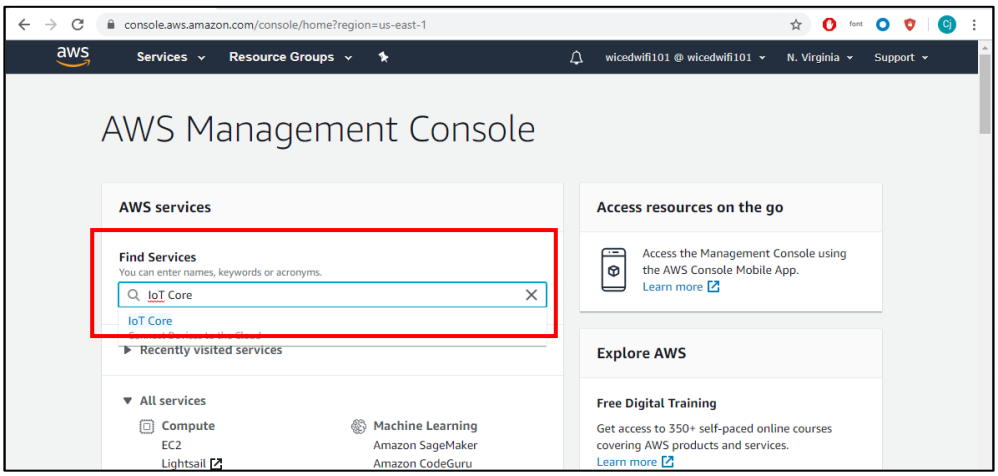
3.9 Creating Things

Now let's step through the process of manually creating a new *thing* in AWS. Note that these steps may be slightly different when you try it because Amazon updates their web interface often, but it should be close enough to understand.

At the end of the *thing* creation process, AWS will ask you which policy to attach to your *thing*. You can have a generic policy that applies to multiple *things*, or you can have a specific policy that applies to each *thing*. This is a security architecture decision that the applications developer is responsible for. In this case we will create a generic policy first and then we will create the *thing* after that.

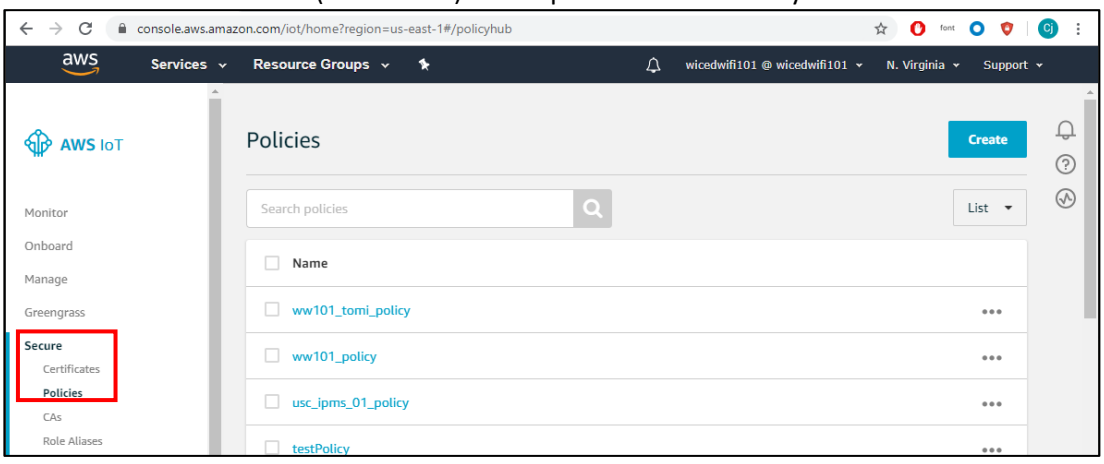
3.9.1 To create a policy:

1. Go to https://aws.amazon.com/console/ and login using the credentials provided.
2. If needed, click the "AWS" icon in the top left.
3. Under **Find Services**, search for and select **IoT Core**.

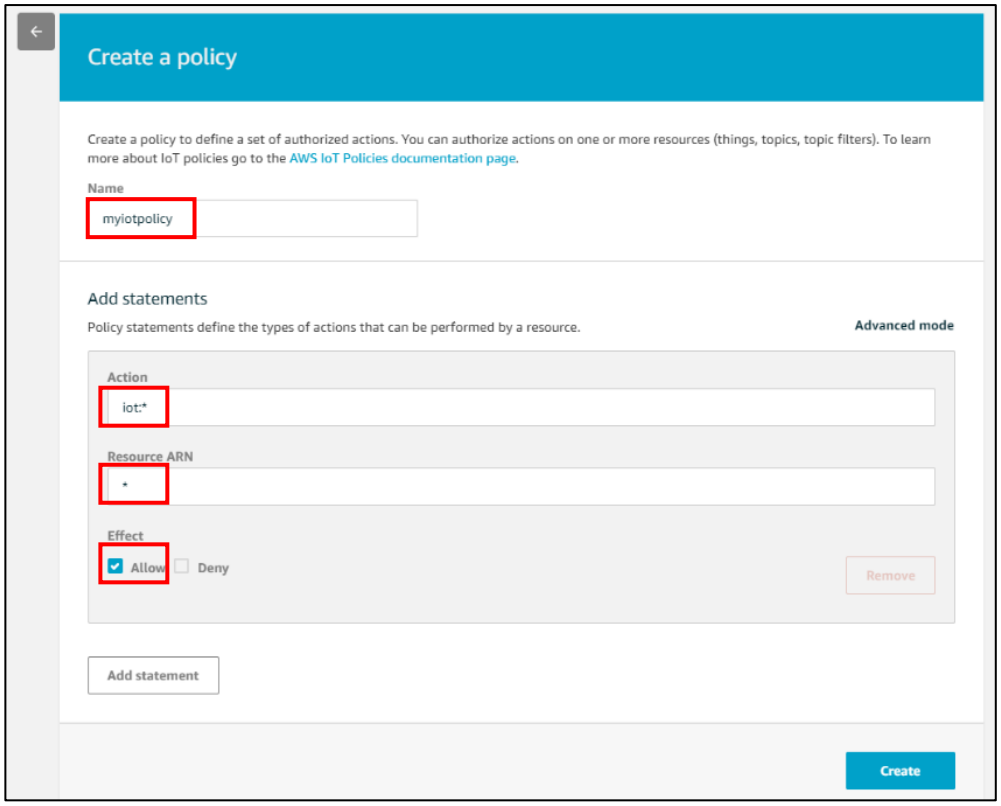


4. On the left, click on **Secure > Policies**.

   You will be able to look at (and search) all the policies attached to your account.



5. To create a new policy, click **Create**. On the "Create a Policy" page:



a. Give the policy a **Name** (in this case `myiotpolicy`). Your name should reflect your security architectural objectives.

b. Then assign the specific **Action**. In this case I am using `iot:*` which means this policy will allow all IoT actions (i.e connect, publish).

c. Specify which **Resource** that this policy applies to. To simplify this I use "*" meaning all resources. However, there are a complicated set of rules that let you constrain the resource based on certificates, connection types etc.

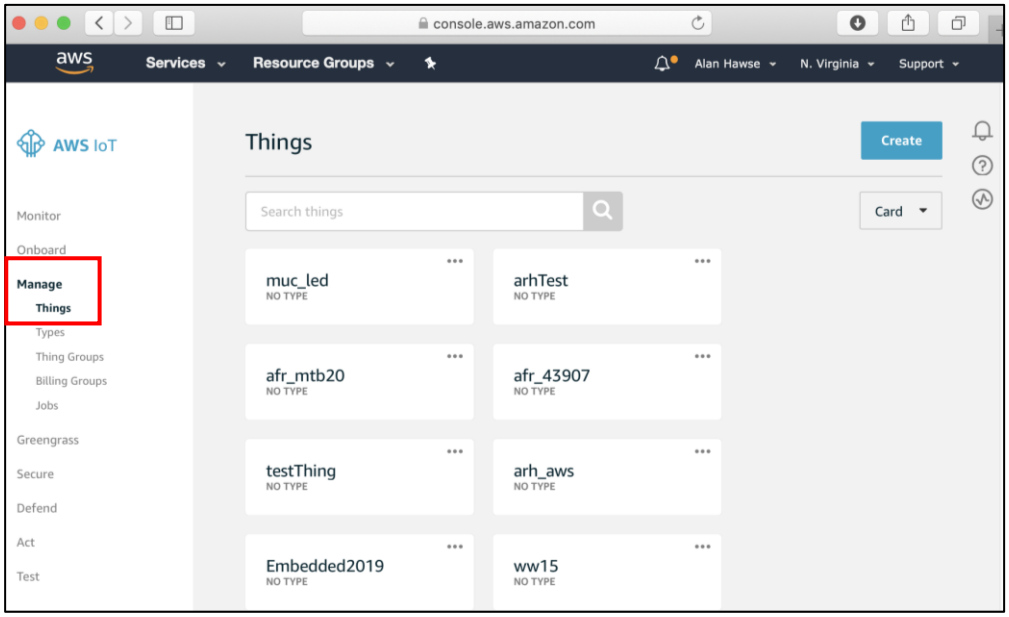
   You can read about those rules here:
   https://docs.aws.amazon.com/iot/latest/developerguide/iot-policies.html

d. Check the box under Effect for "Allow" so that your specified actions are allowed on the specified resources. Note that you can also create policies that deny actions on specified resources to limit things.

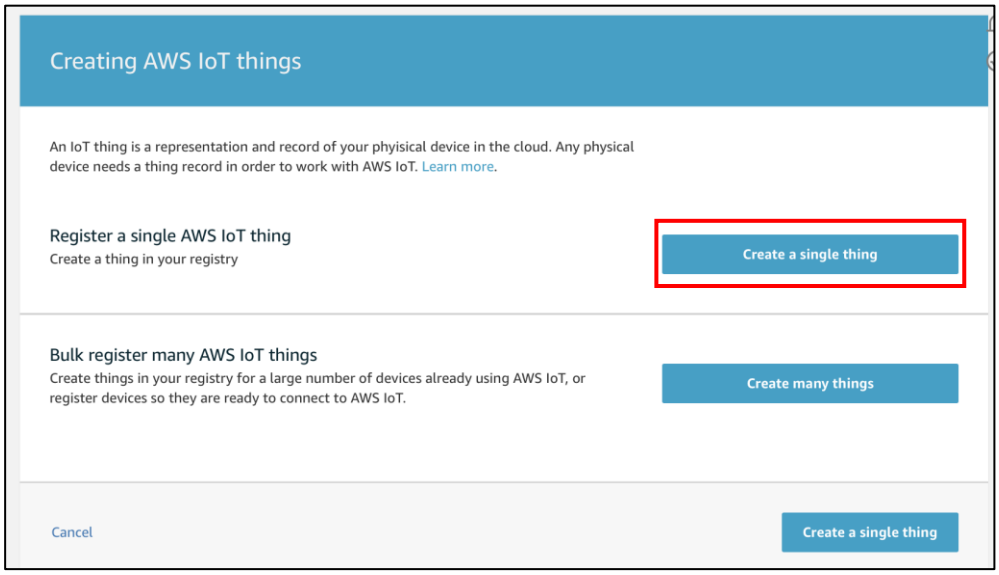
e. Click **Create** at the bottom to make the policy.

3.9.2 To create a thing:

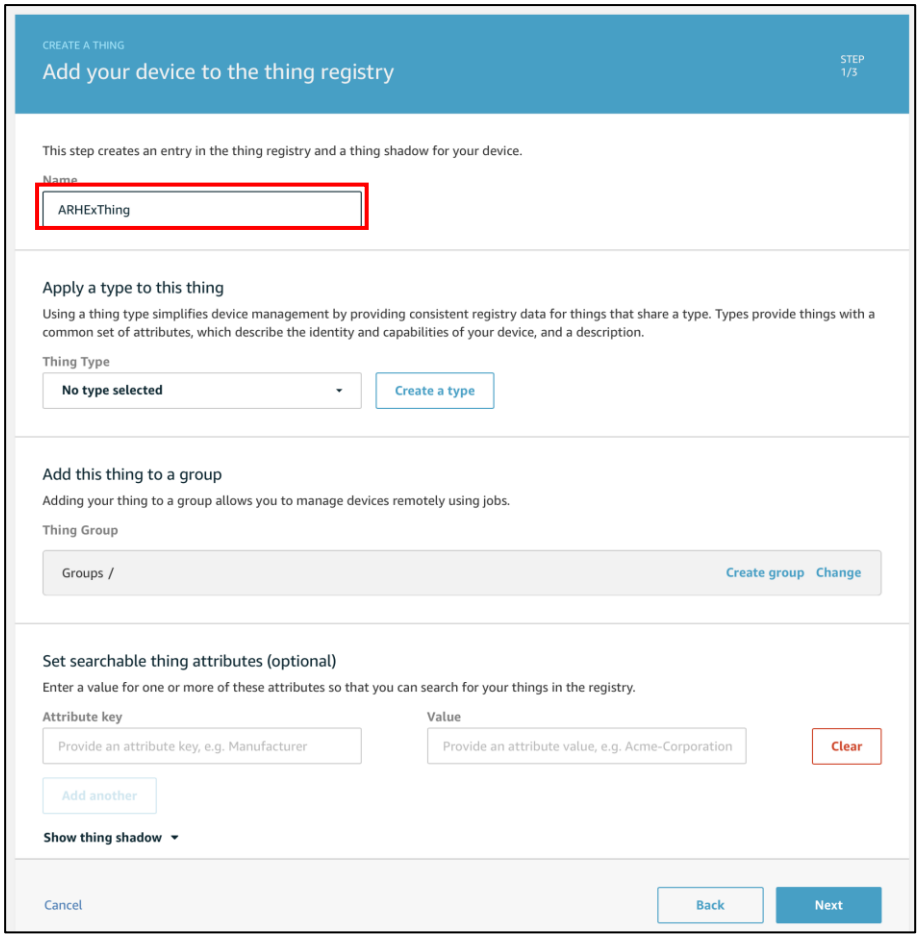
1. Go back to **IOT Core** (use the back arrow at the top left) and select **Manage > Things**.



2. Once there, click **Create** to make a new *thing*.
3. On the "Creating AWS IoT things" page, choose **Create a single thing**.

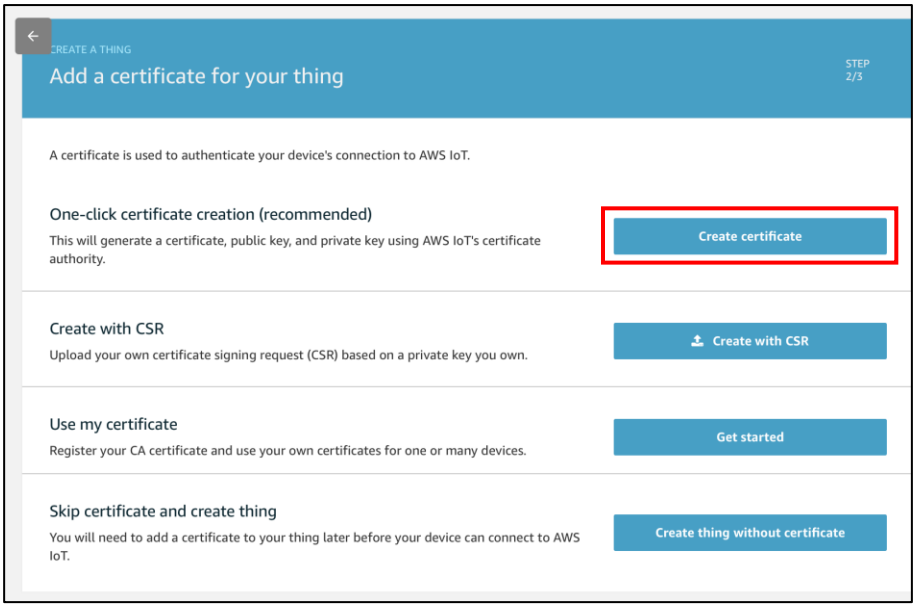


4. Give it a unique name and click **Next**.



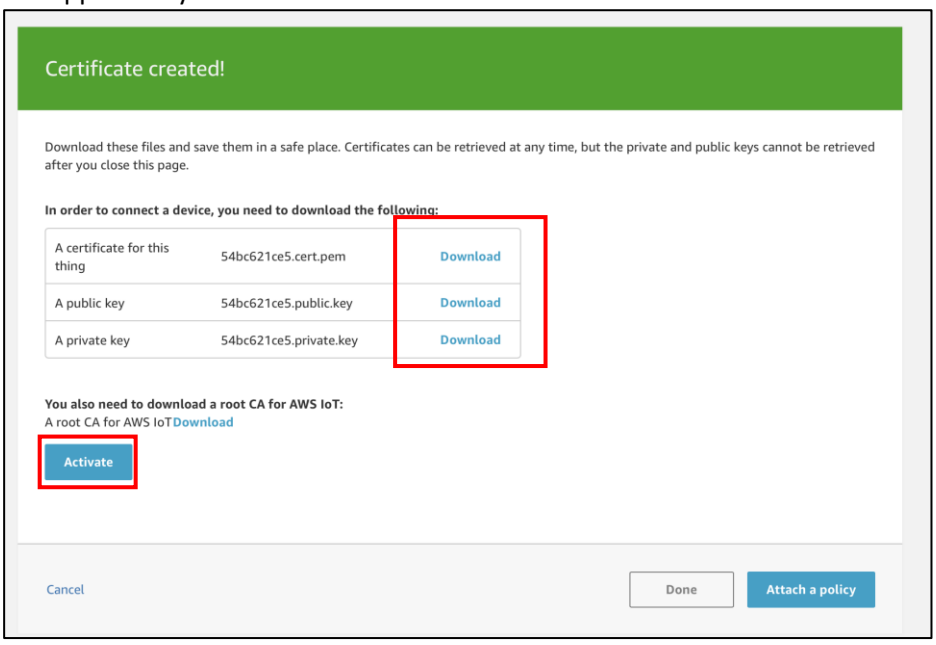
The security associated with a *thing* is controlled using X.509 certificates. Each *thing* should have a certificate attached to it. The easiest (and probably best) thing to do is let AWS create and manage certificates for you.

5. Click **Create certificate**.



6. This will create an AWS signed certificate, a public key and a private key.

   Now, you MUST download all three files from this screen as it will be the last time you have the opportunity. **Do this now!**



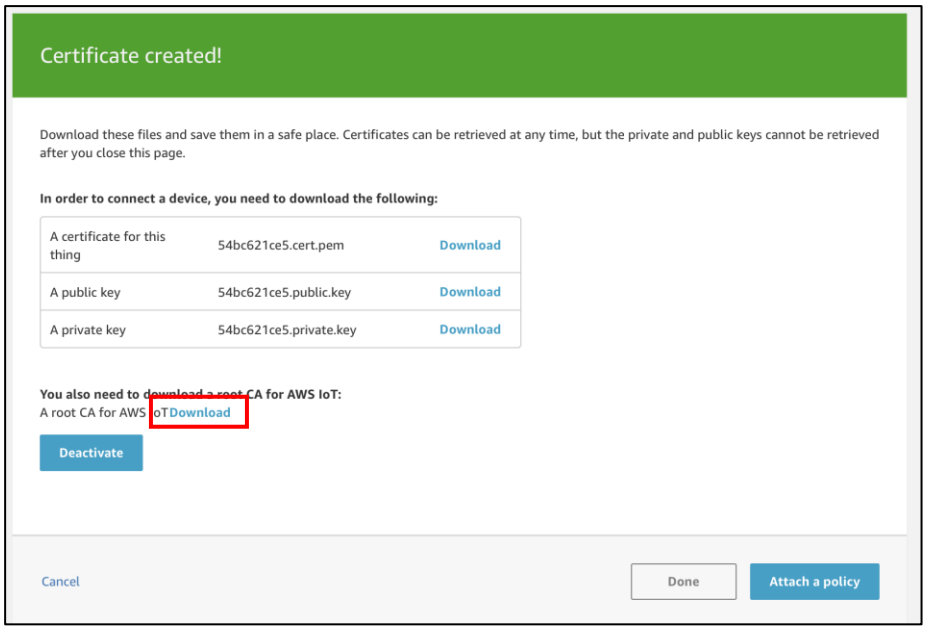
Note: You need all three files for every new certificate.

7. Click **Activate** to enable the certificate.

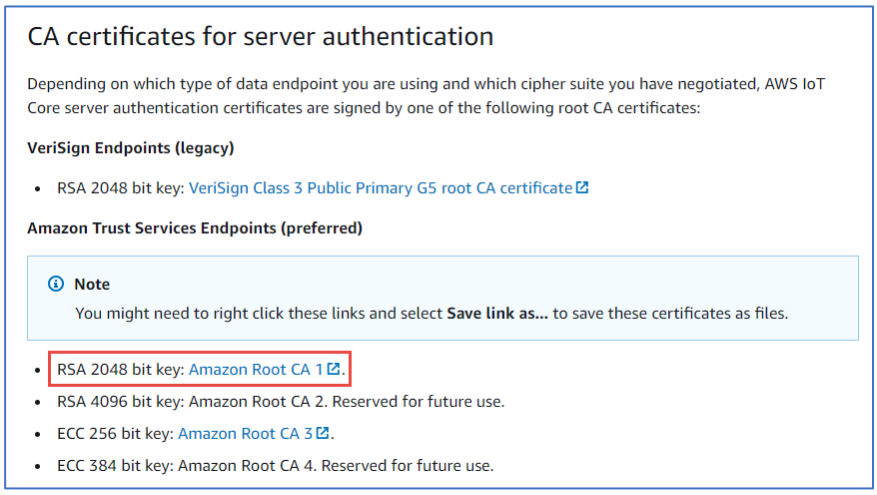
8. When making a TLS connection, it is best for your device to verify the certificate of the other side.

   In this case, your device should verify that it is really Amazon on the other side - not some nefarious site pretending to be Amazon. To do this, you need to include the AWS root certificate in your firmware.

9. Click the **Download** link to open the "CA certificates for server authentication" section of the AWS documentation.



10. From the new tab, right-click on the **RSA 2048 bit key** link and save the link to a file as appropriate to your web browser (e.g. *Save link as…* or *Download Linked File As…*)

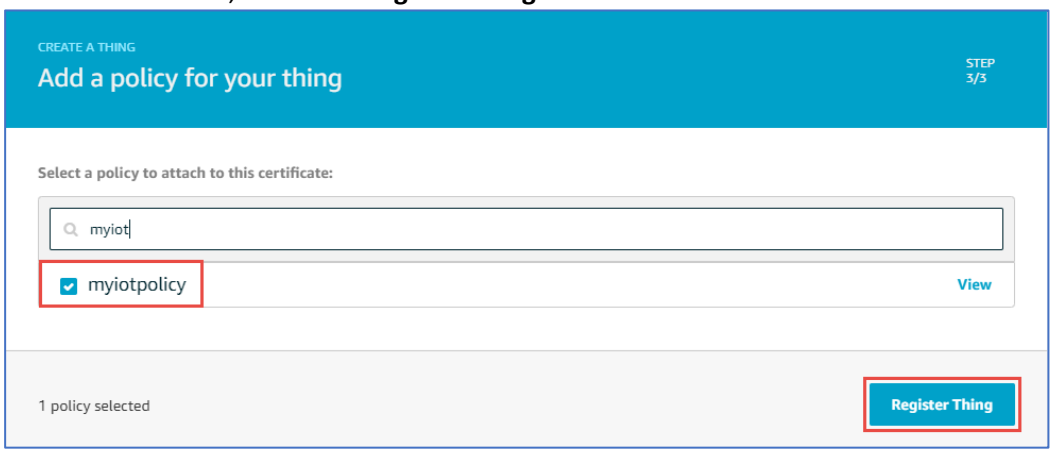


Note: You only need to download this file once since it will be the same for any project that you create.

The certificate and keys for your *thing* allow AWS to validate the identity of your *thing*. The root CA will allow your firmware to validate that the connection to AWS is legitimate (i.e. you didn't connect to an AWS imposter).
The next step in the process is to attach a policy to the certificate (the policy we created way back at the start).

11. Click **Attach Policy**.
12. On the "Add a policy for your thing" page, search for your policy (if there is a long list).

    Click the check box, then click **Register Thing**.



3.10 Transform Keys Into "C"

Once you have your certificates and keys you need to turn them into a format that is usable by your firmware. If you are using Amazon FreeRTOS, the easiest way to change your certificate into a C-File is to use the utility provided by Amazon FreeRTOS. See section 3.10.1.

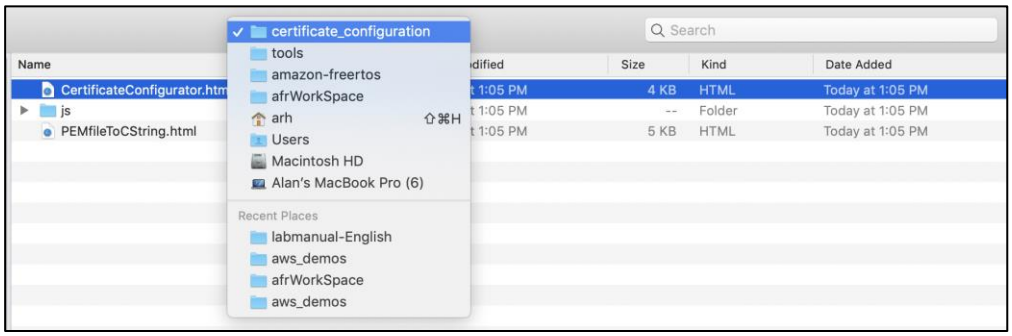
If you are using AWS without Amazon FreeRTOS or Mbed, you will need to do one of the following:

- convert the certificate and keys to C strings on your own;
- use the PEMfileToCString.html utility (if you have the Amazon FreeRTOS repository cloned);
- use a Python script that we will provide.

The latter two methods are described in sections 3.10.2 and 3.10.3. For Mbed, the file *aws_config.h* contains examples of what the strings look like. For AnyCloud, examples will be provided in the file containing the keys - the specific file varies by code example.

3.10.1 Amazon FreeRTOS Utility

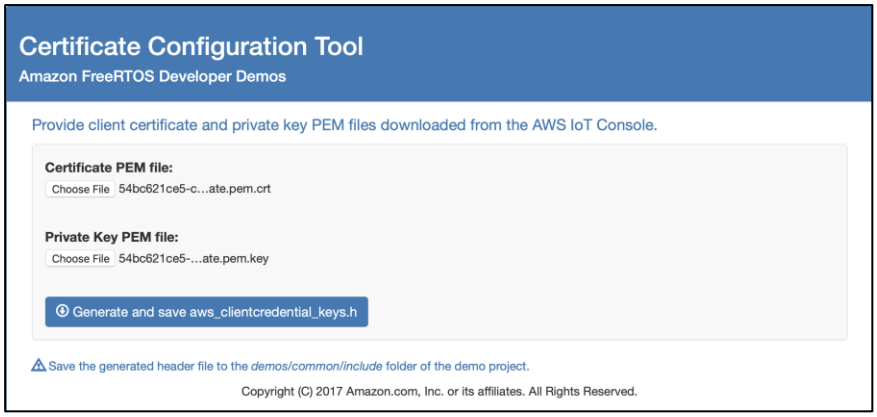
1. First, go to the *amazon-freertos/tools/certificate_configuration* directory and open the *CertificateConfigurator.html* in a web browser.



Note: If you have not downloaded the amazon-freertos repository yet, you can download it using:

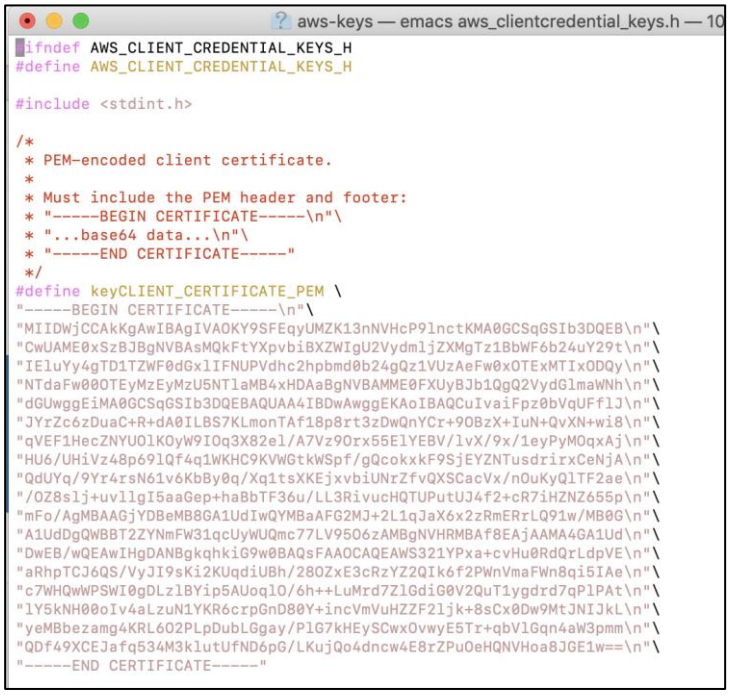
```
git clone https://github.com/cypresssemiconductorco/amazon-freertos.git
```

2. Select your Certificate and Private Key files. Then click **Generate and save...**



This will make a C-header file named *aws_clientcredential_keys.h* and download it to your PC. Move the file to the amazon-*freertos/demos/include* directory if your browser does not put it there by default. This is just a normal C-header file with #defines for the `keyCLIENT_CERTIFICATE_PEM`, `keyCLIENT_PRIVATE_KEY_PEM`. And this file is named exactly the right thing for the Amazon FreeRTOS demo.

The file will look like this:



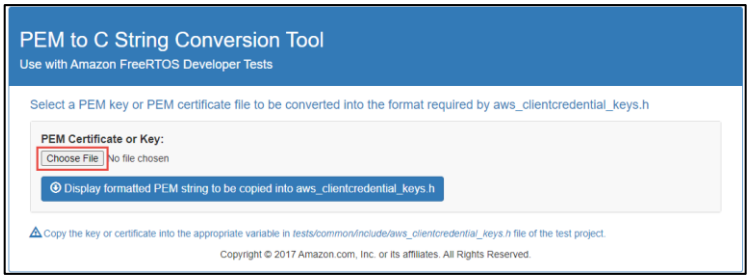
3.10.2 PEMfileToCString.html Tool

You can also use the Amazon FreeRTOS Tool called *PEMfileToCString.html* to convert any (single) PEM file into a C-String. This is useful for applications that use AWS without Amazon FreeRTOS or for Mbed applications. However, you must have the Amazon FreeRTOS repo to use this tool.

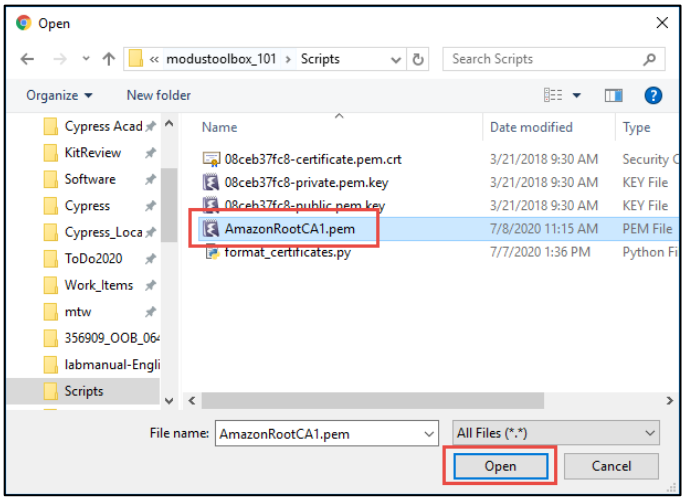
1. Go to the *amazon-freertos/tools/certificate_configuration* directory and open the PEMfileToCString.html file in a web browser.

   Note: If you have not downloaded the amazon-freertos repository yet, you can download it using:
   `git clone https://github.com/cypresssemiconductorco/amazon-freertos.git`

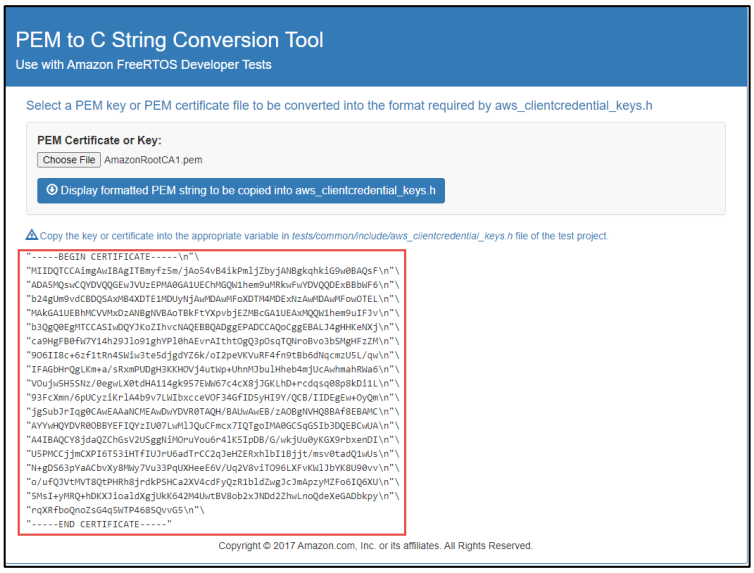
2. For example, to convert the Root Certificate for Amazon, click **Choose File**.



3. Select the *AmazonRootCA1.pem* file and click **Choose** or **Open**.



4. Then click **Display formatted PEM string…** to show the formatted string.



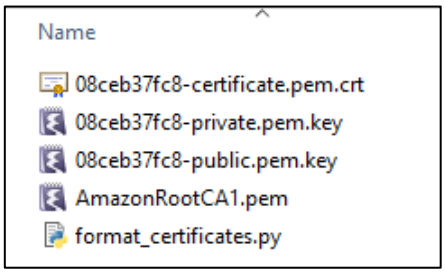
5. This string can then be copy/pasted into your keys file (or any other C-file).

   For instance, you could add the string to the file *aws_clientcredenital_keys.h* for Amazon FreeRTOS or to *aws_config.h* for Mbed. Make sure you copy each key string into the correct #define or array in your application.

3.10.3 Python Script format_certificates.py

The final option to convert the certificates to strings is to use the Python script *format_certificates.py*. This script is available in the class material under the *Scripts* directory. To use it:

1. Put the script and your certificates in a directory as shown below.



   The private key file name must end with *private.pem.key*, the certificate file name must end with *certificate.pem.crt* and the Amazon Root CA must end with *CA1.pem*.

2. Open an terminal and enter:

```
python ./format_certificates.py
```

   This prints the formatted strings to the terminal. You can copy/paste these to your application code. Make sure you copy each key string into the correct #define or array in your application.

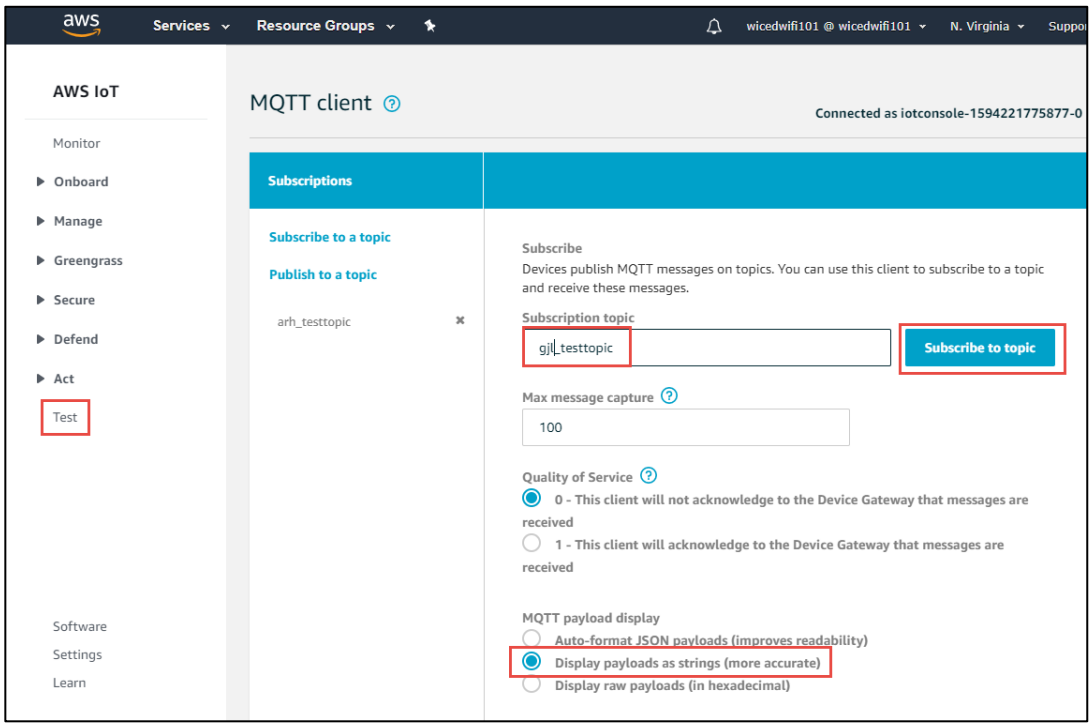
   Note that the script doesn't do anything with the public key because it is not used by your firmware. Rather, it is needed by whatever will connect with your kit - in this case Amazon.

3.11 Test Client

The AWS website has an MQTT Test Client that you can use to test publishing and subscribing to topics. Think of it as a terminal window into your message broker, or as a generic IoT *thing* that can publish and subscribe.

3.11.1 Subscribing to a Topic from the Test Client

1. Select "Test" from the panel on the left of the screen.
2. Enter a topic that you want to subscribe to such as **<your_initials>_testtopic** in the "Subscription topic" box.
3. Select "Display payloads as strings", and click on **Subscribe to topic**.
4. Put your initials or some other unique string in the topic if you are using the class AWS account. If not, you may see messages from someone else publishing to the same topic.



3.11.2 Publishing to a Topic from the Test Client

Now that I am subscribed to a topic, I can publish messages to that topic from another instance of the MQTT test client.

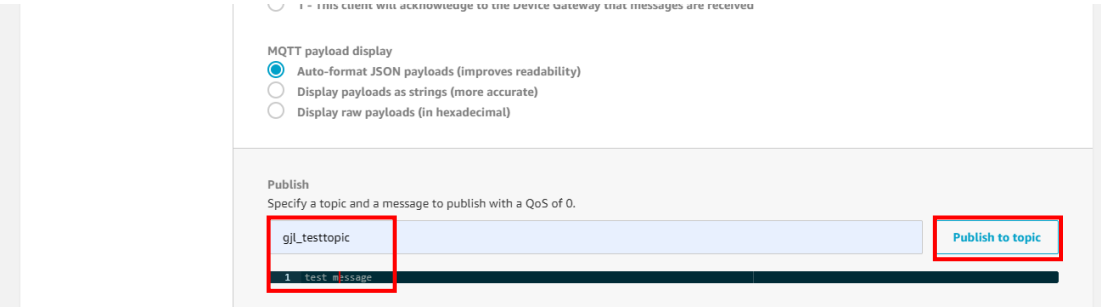
1. First, open another web browser tab, login to the AWS account, and go to the Test page.

   Note: If possible, team up with another student if you can so that one person subscribes and the other publishes to the same topic.

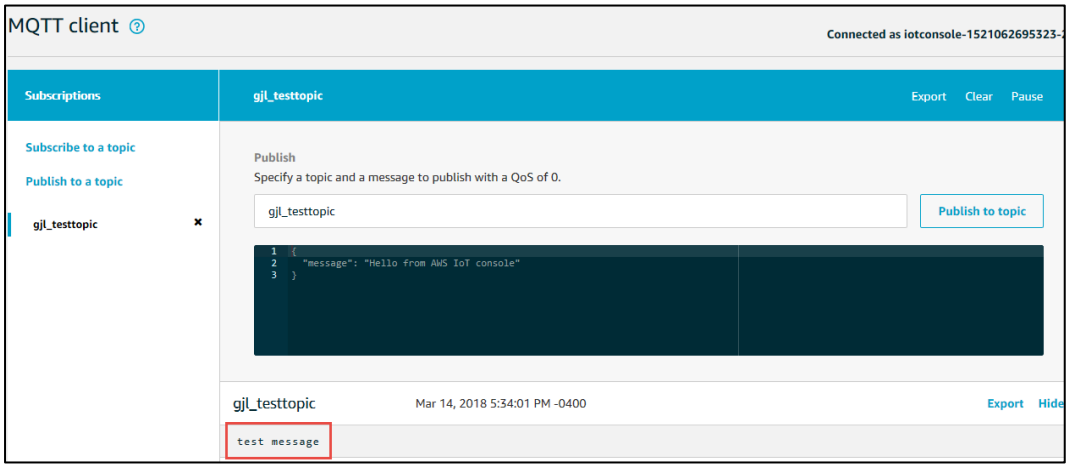
2. Scroll down to the "Publish" section of the page and fill in the name of the topic that you subscribed to earlier. The name must be exactly the same (including case).

3. Then type in your message and press "Publish to topic".

   You can see in the box below I sent "test message".



4. Now, go back to the tab with the subscription and see that the published message was sent to the subscriber.



3.12 GreenGrass

AWS IoT GreenGrass is available at https://aws.amazon.com/greengrass/

From the Webpage, AWS IoT GreenGrass is described as follows:

> "AWS IoT Greengrass seamlessly extends AWS to edge devices so they can act locally on the data they generate, while still using the cloud for management, analytics, and durable storage. With AWS IoT Greengrass, connected devices can run AWS Lambda functions, Docker containers, or both, execute predictions based on machine learning models, keep device data in sync, and communicate with other devices securely – even when not connected to the Internet."

The following image (also from the AWS website) shows how devices can be used in the AWS IoT GreenGrass system: