**AWS IoT to Device Communication  
A close-up of a computer

AI-generated content may be incorrect.**

1. **First we need to register our device( witch can be sensor, actuator, mobile phone) as a Thing which is virtual representation of our device.**
2. **Once we do that, it generates IoT certificate, which is used by device.**
3. **Then we use our** [**solution**](#_Solution_design) **to communicate with AWS IoT Core service**

# Register the Device

1. First is to register the device as a **Thing**in the AWS IoT core. Log in to AWS Account and choose a region in the upper right corner.
2. First step in registering the device as a Thing is creating an IoT policy which authorizes the device to perform actions within AWS IoT core. Go to Security → Policies, in AWS console. Device has a certificate which is associated with the policy we create. Depending on a policy, certain actions can be made by the device.   
     
   We are creating non restriciton policy. We allow all types of **action** by all types of resources. But sometimes we will have to restrict our device to only certain opearations and topics they can subscribe to.

A screenshot of a computer

AI-generated content may be incorrect.  
  
A screenshot of a computer

AI-generated content may be incorrect.

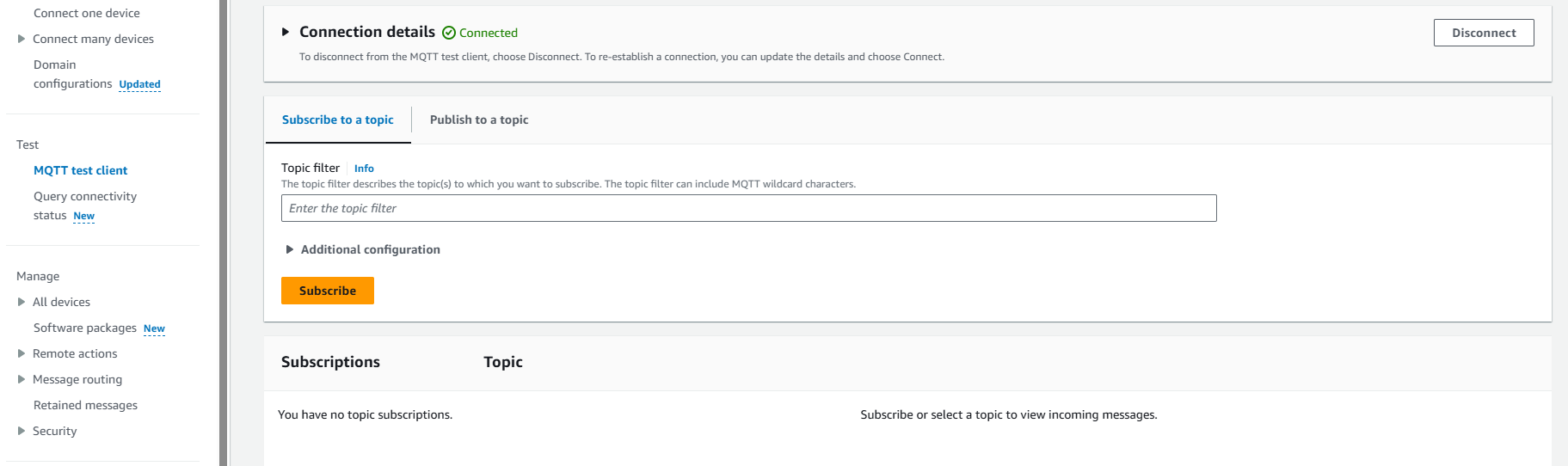
1. After creating the policy, we need to create a device as a Thing and attach the policy to it. Go to Manage → Things → Create things → Create single thing – > enter thing name and check no shadow → Auto-generate a new certificate( It will create certificate files which the device uses for the authentication purpose. AWS IoT supports **X.509 client certificates**.) → Create thing
2. Checking device data endpoint. Devices connect to AWS IoT and other services through AWS IoT Core. Through AWS IoT Core, devices send and receive messages using device endpoints that are specific to your account. Each account has several device endpoints that are unique to the account and support specific IoT functions. The AWS IoT device data endpoints support a publish/subscribe protocol that is designed for the communication needs of IoT devices; however, other clients, such as apps and services, can also use this interface if their application requires the specialized features that these endpoints provide. The AWS IoT device service endpoints support device-centric access to security and management services. Go to Connect → Domain configurations. Copy your Domain name from the domain configurations table.

A screenshot of a computer

AI-generated content may be incorrect.

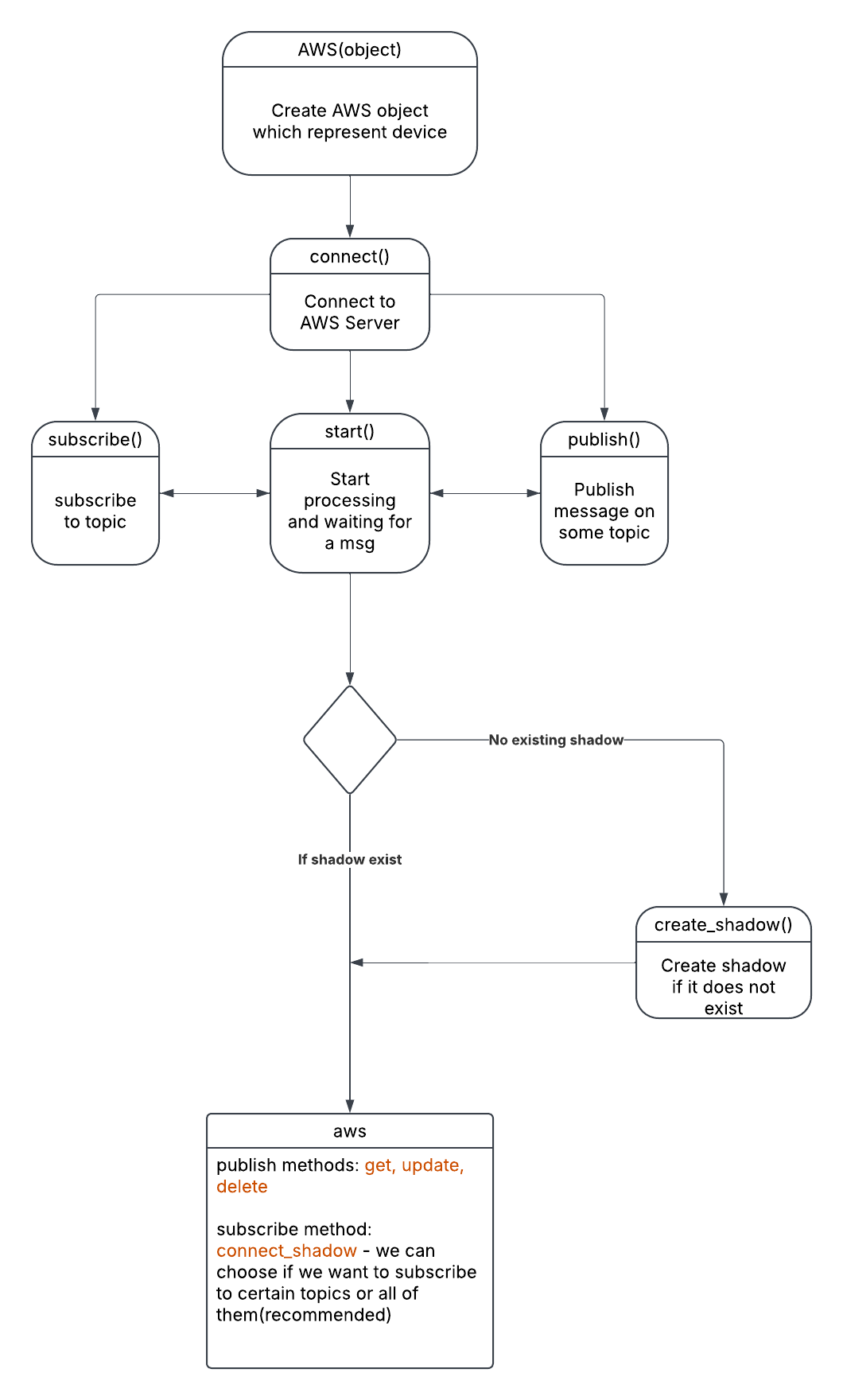
**MQTT Client**

Mqtt test client is used for monitoring the MQTT messages being passed in your AWS account. Devices publish MQTT messages that are identified by topics to communicate their state to AWS IoT. AWS IoT also publishes MQTT messages to inform devices and apps of changes and events. You can subscribe to MQTT message topics and publish MQTT messages to topics by using the MQTT test client. In our case we use mqtt test client to see if our device publishes and subscribes succsessfully to certain topic.



As you can see, MQTT test client offers nice user interface for testing purposes.

# Solution design



# Usage

The first step to successfully using the AWS library (\*\*After creating a **THING**\*\*) is to create an **AWS object**, which will initialize the **MQTT client**, an object linked to the **Shadow Manager**, and the **Client ID**.

The next function we will define is the **connect method**, a function within the AWS class that establishes a connection to the **AWS IoT Core server**. The previous chapter explained how to create an **endpoint** and other necessary configurations.

Regarding the connection itself, **under the hood**, the **MQTT protocol** is used, which operates over **TCP protocol**, while **IP protocol** is used at the network level to connect to the server.

Finally, we have the **disconnect method**, which is responsible for disconnecting the device from the server. When we create an AWS object, we are essentially creating **our device** within the program.

After that, we have the basic **MQTT-related methods**, which are **Subscribe** and **Publish**.

* The **Subscribe method** is used to **subscribe** to a specific topic in order to receive messages from it.
* The **Publish method** allows **sending data** to a topic so that subscribed clients can receive it.

For example, let's say we have a **sensor connected to AWS Cloud** and an **application** that wants to read values from that sensor.

* In this case, the mobile application connects to **AWS Cloud** and subscribes to a specific **MQTT topic**.
* At the same time, the sensor uses the **Publish function** to send its data to that topic.
* Since the application is subscribed to the topic, it will receive the data that the sensor publishes.

Another important feature that has been implemented is the **ability for the user to define a custom callback method**, which will be triggered when a message arrives on a specific topic.

This callback function is passed to the **MQTT client**, which will call it when a message is received.

For example, let's say a **mobile application** sends a request to change the state of a lightbulb from **off to on**.

* When **AWS IoT Core** server forwards this message to the device,
* the **callback method** we previously set up will be automatically triggered,
* and it will execute the necessary operations to change the lightbulb’s state.

The method we created for setting callback functions is called **set\_callback**. It allows us to add **multiple callback functions** to a list, and when a message arrives, the appropriate function will be triggered based on the topic it was sent to.

# Shadows

**What is AWS IoT Device Shadow?**

AWS IoT Device Shadow is a service that enables **persistent state management** for IoT devices, allowing cloud applications to interact with devices even when they are offline. This is achieved by maintaining a **virtual representation** (shadow) of the device's state, stored as a JSON document in AWS IoT Core. Each device can have multiple named shadows, as well as a default (unnamed) shadow.

The shadow document consists of:

* **desired state**: Set by cloud applications to request changes in the device state.
* **reported state**: Updated by the device to reflect its actual status.
* **delta state**: Automatically generated by AWS, representing the difference between desired and reported.

Using MQTT, devices can interact with their shadows by **publishing** and **subscribing** to specific shadow topics.

**How AWS IoT Shadows Work with MQTT**

Our existing Python implementation interacts with AWS IoT Core via MQTT, which is the primary mechanism for shadow operations. AWS reserves the following **MQTT topics** for shadow communication:

**Shadow Update**

Creates a shadow if it doesn't exist, or updates the contents of an existing shadow with the state information provided in the message body. AWS IoT records a timestamp with each update to indicate when the state was last updated. When the shadow's state changes, AWS IoT sends /delta messages to all MQTT subscribers with the difference between the desired and the reported states. Devices or apps that receive a /delta message can perform actions based on the difference. For example, a device can update its state to the desired state, or an app can update its UI to reflect the device's state change.

* **Publish to:**
  + $aws/things/{client\_id}/shadow/update
  + $aws/things/{client\_id}/shadow/name/{shadow\_name}/update *(for named shadows)*
* **Subscribe to (for response handling):**
  + $aws/things/{client\_id}/shadow/update/accepted
  + $aws/things/{client\_id}/shadow/update/rejected
  + $aws/things/{client\_id}/shadow/update/delta

Example JSON payload for updating a shadow (device reporting its state):

A computer code with text

AI-generated content may be incorrect.

Example JSON payload for requesting a device state change (cloud application):

A computer code with blue text

AI-generated content may be incorrect.

**Shadow Retrieval**

GET - Retrieves a current shadow document that contains the complete state of the shadow, including metadata.

* **Publish to:**
  + $aws/things/{client\_id}/shadow/get
  + $aws/things/{client\_id}/shadow/name/{shadow\_name}/get
* **Subscribe to:**
  + $aws/things/{client\_id}/shadow/get/accepted
  + $aws/things/{client\_id}/shadow/get/rejected

**Shadow Deletion**

Deletes the device shadow and its content.

You can't restore a deleted device shadow document, but you can create a new device shadow with the name of a deleted device shadow document. If you create a device shadow document that has the same name as one that was deleted within the past 48 hours, the version number of the new device shadow document will follow that of the deleted one. If a device shadow document has been deleted for more than 48 hours, the version number of a new device shadow document with the same name will be 0.

* **Publish to:**
  + $aws/things/{client\_id}/shadow/delete
  + $aws/things/{client\_id}/shadow/name/{shadow\_name}/delete
* **Subscribe to:**
  + $aws/things/{client\_id}/shadow/delete/accepted
  + $aws/things/{client\_id}/shadow/delete/rejected

It is important that we first subscribe to topics before using get update and delete shadow, so we can see the response.

# Shadow management

When it comes to **shadow creation**, we have two options:

1. **Creating a shadow via the AWS Console**
2. **Creating a shadow directly from the program** using the **create\_shadow method**

If we are working with an **unnamed shadow**, the **create\_shadow** method will first check if it has already been created in the console.

* If it does not exist, the method will send a **publish request** to the **update topic** and create a new **unnamed shadow**.
* If no **state** is specified, it will send a **default state** that we have predefined.

Regarding **named shadows**:

* If a **named shadow** already exists and we call **create\_shadow** with its name, instead of creating a new shadow, the method will simply **update the existing shadow** and write the new values.

After creating a **shadow**, we need to **subscribe it to specific topics** to track changes and system responses.

If we want to **update a shadow** (i.e., publish to the **update topic**), we must be **subscribed to the following topics**:

* **update/accepted** – to confirm that the update was successful
* **update/rejected** – to know if the update failed
* **update/delta** – to track changes in the shadow state
* **update/documents** (optional) – to see the history of shadow changes

If we want to **get a shadow,**we must be **subscribed to the following topics**:

* **get/accepted** – to confirm that the get was successful
* **get/rejected** – to know if the get failed

If we want to **delete a shadow,**we must be **subscribed to the following topics**:

* **delete/accepted** – to confirm that the delete was successful
* **delete/rejected** – to know if the delete failed

**The connect\_shadow Function**

A very important function is **connect\_shadow**.

* If **we do not specify specific topics**, the function will automatically **subscribe to all relevant topics** defined in the previous document.
* We also have the option to **explicitly specify** which topics we want to subscribe to.

**Core Methods for Shadow Management**

There are three main methods for working with shadows:

1. **update\_shadow** – used to **update the shadow**
2. **get\_shadow** – retrieves the **current shadow state**
3. **delete\_shadow** – deletes the **shadow**

All these methods **under the hood** function as **publish methods**, meaning they send MQTT messages to the appropriate **shadow topics**, as explained in the previous document.