

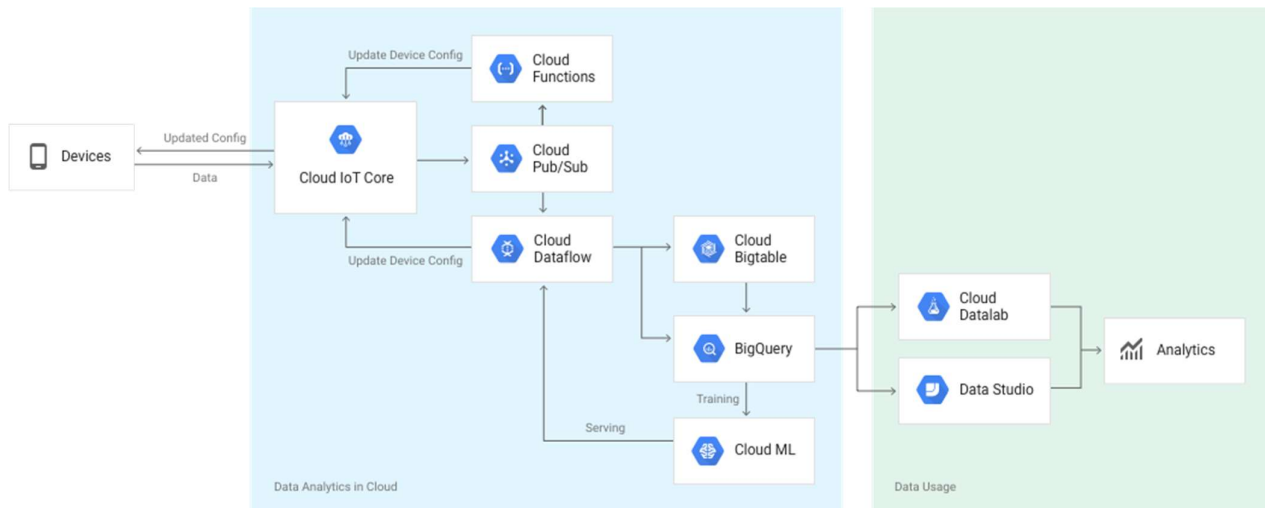
A Tour of Cloud IoT Core

GSP224

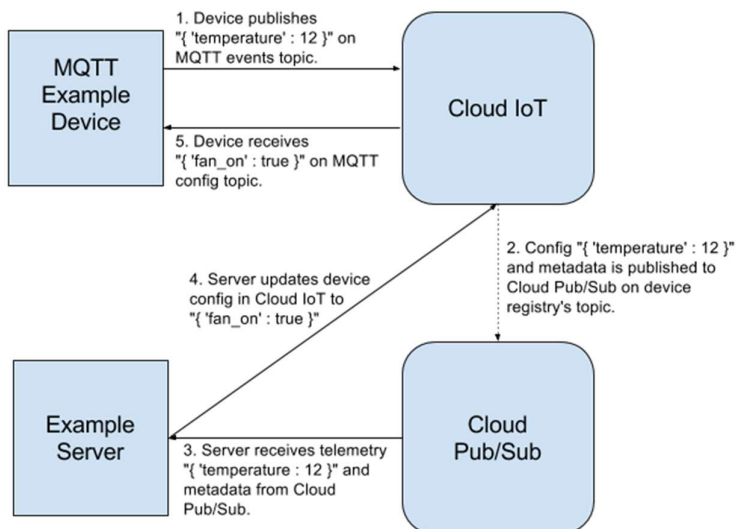


Introductions

Cloud IoT Core is a fully managed service that allows you to easily and securely connect, manage, and ingest data from millions of globally dispersed devices. Cloud IoT Core, in combination with other services on Google Cloud, provides a complete solution for collecting, processing, analyzing, and visualizing IoT data in real time to support improved operational efficiency.



In this lab you'll build a simple but complete IoT system using Cloud IoT Core. The devices in this system publish temperature data to their telemetry feeds, and a server consumes the telemetry data from a Cloud Pub/Sub topic. The server then decides whether to turn on or off the individual devices' fans, via a Cloud IoT Core configuration update. The device will respond to configuration changes from a server based on real-time data.



What you'll learn

- How to create a device registry and add a device to it
- How to provision a device and transmit telemetry data from it
- How to control a device using a server based on a telemetry stream
- [Optional] How to go from virtual device to hardware

Setup and Requirements

Qwiklabs setup

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This Qwiklabs hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

What you need

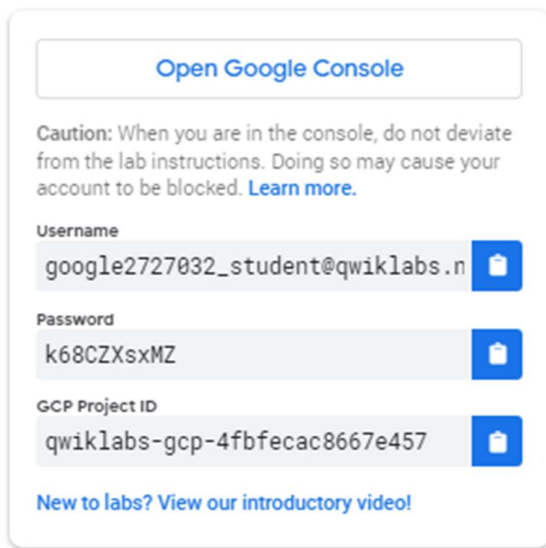
To complete this lab, you need:

- Access to a standard internet browser (Chrome browser recommended).
 - Time to complete the lab.
- Note:** If you already have your own personal Google Cloud account or project, do not use it for this lab.

Note: If you are using a Pixelbook, open an Incognito window to run this lab.


How to start your lab and sign in to the Google Cloud Console


1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.




[Open Google Console](#)

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. [Learn more.](#)

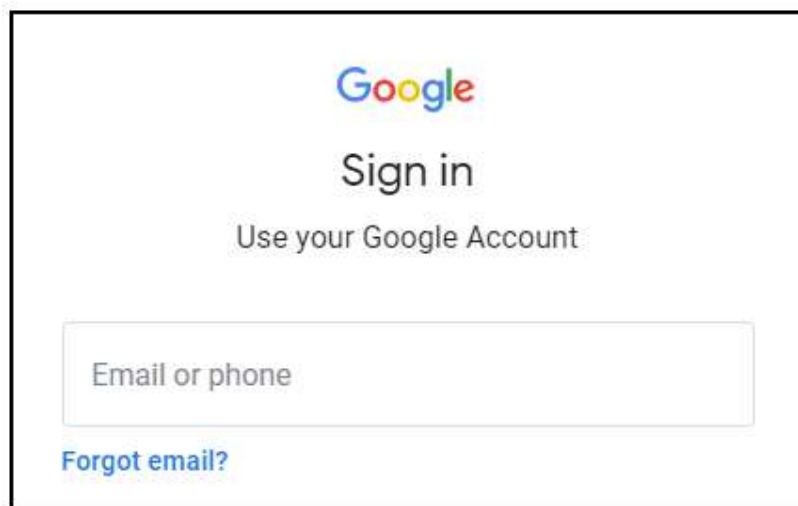
Username
google2727032_student@qwiklabs.n 

Password
k68CZXsxMZ 

GCP Project ID
qwiklabs-gcp-4fbfecac8667e457 

[New to labs? View our introductory video!](#)

2. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.



Google

Sign in

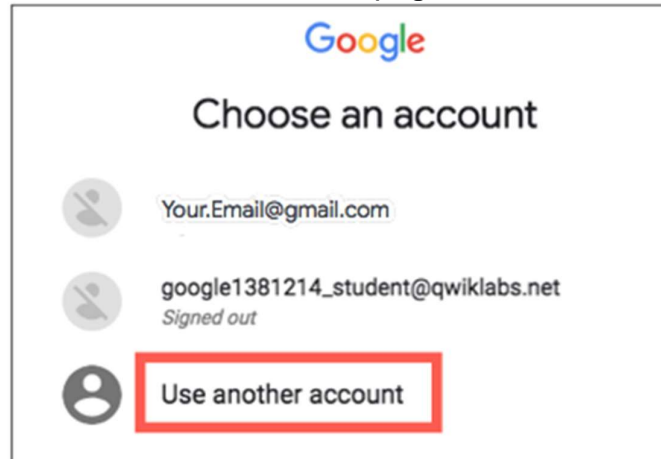
Use your Google Account

Email or phone

[Forgot email?](#)

Tip: Open the tabs in separate windows, side-by-side.

If you see the **Choose an account** page, click **Use Another**



Account.

3. In the **Sign in** page, paste the username that you copied from the Connection Details panel. Then copy and paste the password.

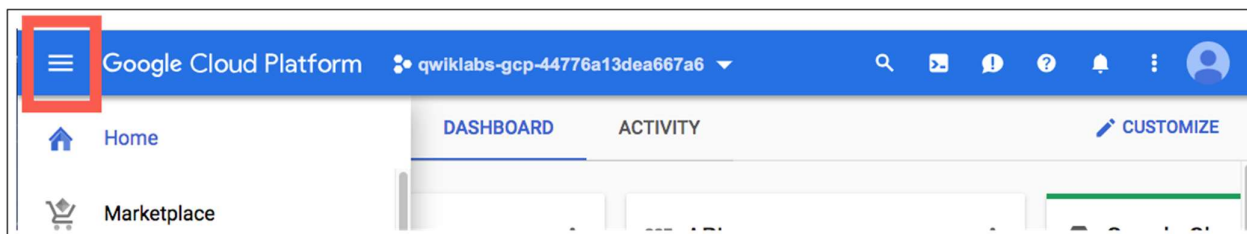
Important: You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

4. Click through the subsequent pages:

- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

After a few moments, the Cloud Console opens in this tab.

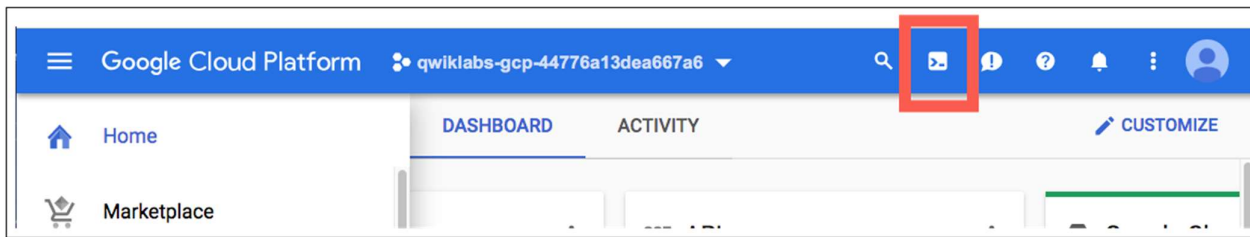
Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.



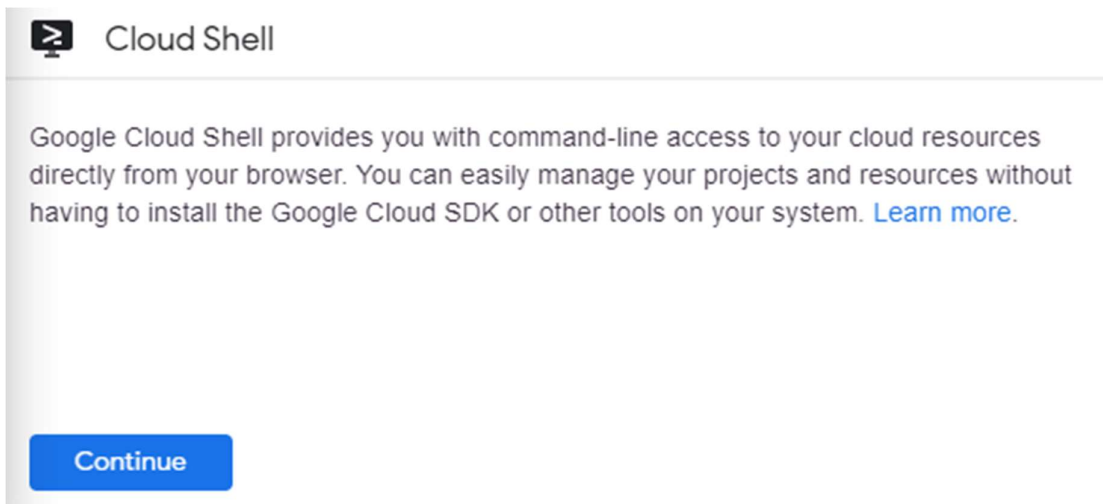
Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

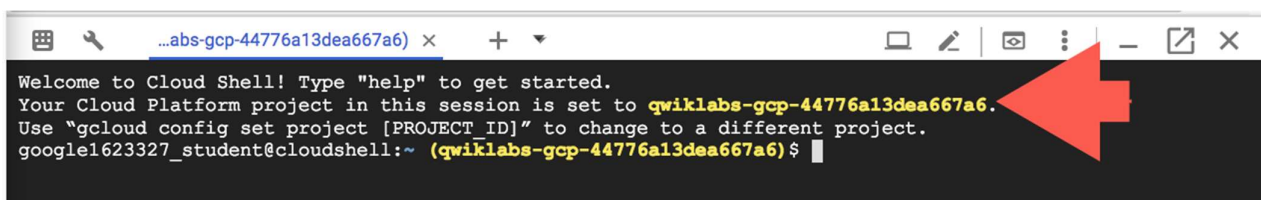
In the Cloud Console, in the top right toolbar, click the **Activate Cloud Shell** button.



Click **Continue**.



It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your *PROJECT_ID*. For example:



`gcloud` is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

You can list the active account name with this command:

```
gcloud auth list
```

(Output)

```
Credentialed accounts:
- <myaccount>@<mydomain>.com (active)
```

(Example output)

```
Credentialed accounts:  
- google1623327 student@gwiklabs.net
```

You can list the project ID with this command:

```
gcloud config list project
```

(Output)

```
[core]  
project = <project_ID>
```

(Example output)

```
[core]  
project = qwiklabs-gcp-44776a13dea667a6
```

For full documentation of `gcloud` see the [gcloud command-line tool overview](#).

This lab focuses on Google Cloud IoT Core. Tangential concepts and code blocks are glossed over and are provided for you to simply copy and paste.

Use Cloud IoT Core with Virtual Devices

You will connect a virtual device and run the server that listens to the telemetry messages from the connected device, then use the server to control that virtual device.

Start in the Cloud Console

In Google Cloud Shell, clone the repo containing the end-to-end sample:

```
git clone https://github.com/GoogleCloudPlatform/python-docs-samples
```

Enter the following command to `cd` into the sample folder:

```
cd python-docs-samples/iot/api-client/end_to_end_example
```

Next, initialize virtual environment and install the sample dependencies:

```
virtualenv env --python=python3 && source env/bin/activate
```

Install version 1.16 of Google API Core:

```
pip install google-api-core==1.16.0
```

Now install the requirements.txt file:

```
pip install -r requirements.txt
```

At this point, check that you have installed the Python dependencies correctly by running all the Python scripts without passing any parameters:

```
python clouddiot_pubsub_example_mqtt_device.py
python clouddiot_pubsub_example_server.py
```

If the dependencies installed successfully, the programs will print their respective usage messages. For example:

```
(env) gcpstaging20065_student@qwiklabs-gcp-4f438a9a5e0b68c9:~/python-docs-
samples/iot/api-client/end_to_end_example$ python
clouddiot_pubsub_example_mqtt_device.py
usage: clouddiot_pubsub_example_mqtt_device.py [-h] --project id PROJECT_ID
--registry id REGISTRY_ID
--device_id DEVICE_ID
--private key file
PRIVATE KEY FILE --algorithm
{RS256,ES256}
[--cloud region CLOUD_REGION]
[--ca_certs CA_CERTS]
[--num messages NUM_MESSAGES]
[--mqtt_bridge_hostname
MQTT_BRIDGE_HOSTNAME]
--mqtt_bridge_port MQTT_BRIDGE_PORT]
[--message_type {event,state}]
clouddiot_pubsub_example_mqtt_device.py: error: the following arguments are required: --
project id, --registry id, --device_id, --private key file, --algorithm
(env) gcpstaging20065_student@qwiklabs-gcp-4f438a9a5e0b68c9:~/python-docs-
samples/iot/api-client/end_to_end_example$ python clouddiot_pubsub_example_server.py
usage: clouddiot_pubsub_example_server.py [-h] --project_id PROJECT_ID
--pubsub subscription
PUBSUB_SUBSCRIPTION
[--service_account_json SERVICE_ACCOUNT_JSON]
clouddiot_pubsub_example_server.py: error: the following arguments are required: --
project id, --pubsub subscription
```

If you see an error similar to `ImportError: No module named ...``, go back and make sure you installed Virtual Environment correctly, or see the [Python Development Environment Setup Guide](#) for detailed information on using Python with Google Cloud.

Now that you have the program libraries installed, it's time to set up your Google Cloud IoT Core project.

Set up the virtual device demo

You will need to know the **project id** value when you run the client and server programs. For the Cloud Shell, the environment variable `${DEVSHHELL_PROJECT_ID:-Cloud Shell}` is populated with your current **project id** after you have set it to your Cloud IoT project.

```
gcloud config set project <your-project-id>
```

Create a Pub/Sub topic using the following gcloud command:

```
gcloud pubsub topics create tour-pub --project="${DEVSHHELL_PROJECT_ID:-Cloud Shell}"
gcloud pubsub subscriptions create tour-sub --topic=tour-pub
```

Create your Cloud IoT Device Registry using the following gcloud command:

```
gcloud iot registries create tour-registry \
  --region=us-central1 --event-notification-config=topic=tour-pub
```

Next, you will need to generate RSA public and private keys that will be used for authenticating your virtual device when it connects.

Important: Your **private** key should never be transmitted or stored anywhere other than on a device and should only be used by the device to generate an authorization credential.

```
openssl req -x509 -newkey rsa:2048 -days 3650 -keyout rsa_private.pem \
  -nodes -out rsa_public.pem -subj "/CN=unused"
```

With your keys in hand, you're ready to register a device. Register your device using the **public** key.

```
gcloud iot devices create test-dev --region=us-central1 \
  --registry=tour-registry \
  --public-key path=rsa_public.pem,type=rs256
```

Congratulations, you have set up Cloud Pub/Sub, created your device registry, and added a device to the registry!

Start the virtual server

Now that you have created all of the Cloud resources, you need to connect your device and communicate with it via Pub/Sub - it's time to simulate a device and server.

Edit `cloudiot_pubsub_example_server.py` and replace the code used to generate service account credentials from a provided JSON file to instead use the built-in credentials for Compute Engine, which is what is running under the hood of the Cloud Shell.

Replace the following code:

```
def __init__(self, service_account_json):
    credentials = ServiceAccountCredentials.from_json_keyfile_name(
        service_account_json, API_SCOPES)
```

With this:

```
def __init__(self, service_account_json):
    from google.auth import compute_engine
    credentials = compute_engine.Credentials()
```

Now that you have changed the server to use the Compute Engine credentials, start the server:

```
python cloudiot_pubsub_example_server.py \
    --project_id="${DEVSHHELL_PROJECT_ID:-Cloud Shell}" \
    --pubsub_subscription=tour-sub
```

When the server starts, you will see the message "Listening for messages on projects/your-project-id/subscriptions/tour-sub", which indicates the server is running.

Start the virtual device and observe

Add a new tab to your Cloud Shell by clicking the + icon on the Cloud Shell ribbon.

In the new Cloud Shell tab, run the following to navigate to the device sample folder and initialize your virtual environment:

```
cd python-docs-samples/iot/api-client/end_to_end_example
source env/bin/activate
```

Retrieve the latest root certificate from Google:

```
wget pki.goog/roots.pem
```

Now, connect the virtual device using the private key, registry ID, device ID, and so on:

```
python cloudiot_pubsub_example_mqtt_device.py \
    --registry_id tour-registry \
    --device_id test-dev \
    --project_id "${DEVSHHELL_PROJECT_ID:-Cloud Shell}" \
    --private_key_file rsa_private.pem \
    --mqtt_bridge_port 443 \
    --algorithm RS256 \
    --ca_certs roots.pem \
    --cloud_region us-central1
```

When you connect the device, it will show and report its temperature, which increases when the fan is turned off. If the fan is enabled, the virtual device's temperature will decrease. Because the device is controlled from the server, which is analyzing the stream of incoming sensor data and making this decision for it, the device does not need to be aware of the conditions for enabling or disabling its fan.

Note: If you are seeing an error indicating "Out of Memory" this usually means that one of the connection parameters is incorrect. Make sure that the `${DEVSHHELL_PROJECT_ID:-Cloud Shell}` environment variable is correctly set (Step 4) and check that you are using the correct device ID and registry ID when invoking the sample app.

The following section shows the output of the server that is subscribed to the telemetry events from the device:

```
The device (test-dev) has a temperature of: 11
('Setting fan state for device', u'test-dev', 'to on.')
The device (test-dev) has a temperature of: 10
The device (test-dev) has a temperature of: 9
The device (test-dev) has a temperature of: 8
The device (test-dev) has a temperature of: 7
The device (test-dev) has a temperature of: 6
The device (test-dev) has a temperature of: 5
The device (test-dev) has a temperature of: 4
The device (test-dev) has a temperature of: 3
The device (test-dev) has a temperature of: 2
The device (test-dev) has a temperature of: 1
The device (test-dev) has a temperature of: 0
The device (test-dev) has a temperature of: -1
('Setting fan state for device', u'test-dev', 'to off.')
The device (test-dev) has a temperature of: 0
The device (test-dev) has a temperature of: 1
The device (test-dev) has a temperature of: 2
```

The following section shows the output of the device that is transmitting its telemetry events to the server:

```
('Publishing payload', '{"temperature": 11}')
Published message acked.
Received message '{"fan_on": true}' on topic '/devices/test-dev/config' with Qos 1
Fan turned on.
('Publishing payload', '{"temperature": 10}')
Published message acked.
('Publishing payload', '{"temperature": 9}')
Published message acked.
('Publishing payload', '{"temperature": 8}')
Published message acked.
('Publishing payload', '{"temperature": 7}')
Published message acked.
('Publishing payload', '{"temperature": 6}')
Published message acked.
('Publishing payload', '{"temperature": 5}')
Published message acked.
('Publishing payload', '{"temperature": 4}')
Published message acked.
('Publishing payload', '{"temperature": 3}')
Published message acked.
('Publishing payload', '{"temperature": 2}')
Published message acked.
('Publishing payload', '{"temperature": 1}')
Published message acked.
('Publishing payload', '{"temperature": 0}')
Published message acked.
('Publishing payload', '{"temperature": -1}')
Published message acked.
Received message '{"fan on": false}' on topic '/devices/test-dev/config' with Qos 1
Fan turned off.
('Publishing payload', '{"temperature": 0}')
Published message acked.
('Publishing payload', '{"temperature": 1}')
Published message acked.
('Publishing payload', '{"temperature": 2}')
```

You have now set up a virtual device and are successfully transmitting telemetry data and receiving configuration changes! At this point you have a basic understanding of Google Cloud IoT Core.

Troubleshooting

[The troubleshooting section of the end-to-end sample](#) has helpful pointers if you get stuck.

(Optional) Going from the virtual to the real

Completion time: 45 minutes

Now that you have a full simulation of your device and server working, it's time to move from the virtual device to a physical device. You will do this by:

- If you haven't installed Android Studio yet, [follow the developer documentation here](#).
- If you haven't installed Android Studio on your device yet, follow the board installation instructions, for example [these instructions cover imx7 setup](#).
- Opening Android Studio and installing the demo
- Connecting the demo device to the network and Google Cloud
- Sending telemetry data to the Cloud-connected server
- Replacing the device-simulated server control with Google Cloud IoT Core controls

Install the demo app to your device

Clone the following project:

```
git clone https://github.com/googlecodeclabs/cloud-iot-core-tour
```

In Android Studio, open the Fan Control demo project. When you run the project for the first time, it will probably start blinking green with the screen indicating -999 and 0. This is to indicate that the Wifi is currently disconnected.



If necessary, [connect the device to the wireless network](#) by running the following adb command:

```
am startservice -n com.google.wifisetup/.WifiSetupService -a WifiSetupService.Connect -e ssid <SSID> -e passphrase <PASS>
```

After your device has functioning network access, the project simulates the server control. While the fan is "off", the simulated temperature will increase at a rate of .1 degrees per tick it reaches 40 degrees at which point the fan will automatically be enabled in the app. When the fan is "on", the simulated temperature decreases at a rate of .1 degrees per tick until it reaches 10 degrees.

The following image shows the fan controller "cooling" -- note that the color is icy blue and the animation moves right to left to signal the temperature is going down.



The following image shows the fan controller "heating" -- the color is red hot and the animation moves left to right to indicate the temperature is going up.



After you have the sample starting on your Android Things device, you're ready to connect the sample to Google Cloud IoT.

Replace device control simulation with Cloud IoT Core

Update the device code so that the boolean, *mlsSimulated*, is set to **false**. This will disable the device and if you run the code at this point, the device temperature will increase indefinitely.

Next, add the following line to your **APP** build.gradle script to add support for the [Cloud IoT Core Client for Android Things](#).

```
implementation 'com.google.android.things:cloud-iot-core:1.0.0'
```

After you have added support for the Cloud IoT Core client, convert the private key to PKCS8 from PEM form. The following command will perform this conversion:

```
openssl pkcs8 -topk8 -inform PEM -outform DER -in rsa_private.pem \ -nocrypt >
rsa_private_pkcs8
```

Now copy the PKCS8 key to the *src/main/res/raw* folder of the Android Things project.

```
cat rsa_private_pkcs8 > src/main/res/raw/privatekey
```

Import the classes for the Cloud IoT Core connector and add a member to the application class for the client.

```
import com.google.android.things.iotcore.ConnectionParams;
import com.google.android.things.iotcore.IotCoreClient;
import com.google.android.things.iotcore.TelemetryEvent;

import org.json.JSONException;
import org.json.JSONObject;

import java.io.ByteArrayOutputStream;

import java.io.InputStream;
import java.io.UnsupportedEncodingException;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;

//...

public class FanControlActivity extends Activity {

    //...

    private IotCoreClient client;
```

Add a helper function for converting from an **InputStream** to byte array.

```
private static byte[] inputStreamToBytes(InputStream is) throws IOException{
    ByteArrayOutputStream buffer = new ByteArrayOutputStream();
    int nRead;
    byte[] data = new byte[16384];

    while ((nRead = is.read(data, 0, data.length)) != -1) {
        buffer.write(data, 0, nRead);
    }
```

```

    }

    buffer.flush();
    return buffer.toByteArray();
}

```

Add a new callback for receiving configuration change messages.

```

private void onConfigurationReceived(byte[] bytes) {
    if (bytes.length == 0) {
        Log.d(TAG, "Ignoring empty device config event");
        return;
    }

    try {
        JSONObject message = new JSONObject(new String(bytes, "UTF-8"));
        m_fanOn = message.getBoolean("fan_on");
        Log.d(TAG, String.format("Config: %s", new String(bytes, "UTF-8")));
    } catch (JSONException je) {
        Log.d(TAG, "Could not decode JSON body for config", je);
    } catch (UnsupportedEncodingException iee) {
        Log.e(TAG, "Could not decode configuration message", iee);
    }
}

```

Add a new **Runnable** class to the FanControlActivity that will be used for posting data to PubSub via Cloud IoT Core.

```

private Runnable mTempReportRunnable = new Runnable() {
    @Override
    public void run() {
        Log.d(TAG, "Publishing telemetry event");

        String payload = String.format("{\"temperature\": %d}", (int)m_currTemp);
        TelemetryEvent event = new TelemetryEvent(payload.getBytes(),
            null, TelemetryEvent.QOS_AT_LEAST_ONCE);
        client.publishTelemetry(event);

        mHandler.postDelayed(mTempReportRunnable, 2000); // Delay 2 secs, repost
temp
    }
};

```

Configure the Cloud IoT Core connector in the app's OnCreate method.

```

// Configure the Cloud IoT Connector --
int pkId = getResources().getIdentifier("privatekey", "raw", getPackageName());
try {
    if (pkId != 0) {
        InputStream privateKey = getApplicationContext()
            .getResources().openRawResource(pkId);
        byte[] keyBytes = inputStreamToBytes(privateKey);

        PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec(keyBytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        KeyPair keys = new KeyPair(null, kf.generatePrivate(spec));

        // Configure Cloud IoT Core project information
        ConnectionParams connectionParams = new ConnectionParams.Builder()
            .setProjectId("<your Google Cloud project ID>")
            .setRegistry("<your Cloud IoT Core registry ID>", "<your
registry's cloud region>")
            .setDeviceId("<the device's ID in the registry>")
            .build();
    }
}

```



```

        // Initialize the IoT Core client
        client = new IotCoreClient.Builder()
            .setConnectionParams(connectionParams)
            .setKeyPair(keys)
            .setOnConfigurationListener(this::onConfigurationReceived)
            .build();

        // Connect to Cloud IoT Core
        client.connect();

        mHandler.post(mTempReportRunnable);
    }
} catch (InvalidKeySpecException ikse) {
    Log.e(TAG, "INVALID Key spec", ikse);
} catch (NoSuchAlgorithmException nsae) {
    Log.e(TAG, "Algorithm not supported", nsae);
} catch (IOException ioe) {
    Log.e(TAG, "Could not load key from file", ioe);
}
}

```

Add the following method to the `onDestroy` method to disconnect the client when the app is terminated.

```

@Override
protected void onDestroy() {
    //...

    // clean up Cloud publisher.
    if (client != null && client.isConnected()) {
        client.disconnect();
    }
}

```

Now, when you run the sample application, the device will be controlled through the backend logic responding to the Pub Sub messages!

For further exploration, you could try adjusting the thresholds to dynamically control the temperature limit or using TensorFlow to predict when the value will exceed a threshold.

Congratulations!

Finish Your Quest



This self-paced lab is part of the Qwiklabs [IoT in the Google Cloud](#) Quest. A Quest is a series of related labs that form a learning path. Completing this Quest earns you the badge above, to recognize your achievement. You can make your badge (or badges) public and link to them in your online resume or social media account. [Enroll in this Quest](#) and get immediate completion credit if you've taken this lab. [See other available Qwiklabs Quests](#).

Take your Next Lab

Continue your quest with [Streaming IoT Core Data to Dataprep](#), or check out these suggestions:

- [Building an IoT Analytics Pipeline on Google Cloud](#)
- [Using Cloud Logging with IoT Core Devices](#)

Next steps / Learn More

- Check out the [Building for Enterprise](#) and [IoT and Cloud for Industrial Applications](#) from Google I/O 2017.
- [Listen](#) to the Google Cloud Podcast episode on Cloud IoT Core.
- [Learn how to improve IoT security](#) by securing the authentication between your device and Google Cloud IoT.

Google Cloud Training & Certification

...helps you make the most of Google Cloud technologies. [Our classes](#) include technical skills and best practices to help you get up to speed quickly and continue your learning journey. We offer fundamental to advanced level training, with on-demand, live, and virtual options to suit your busy schedule. [Certifications](#) help you validate and prove your skill and expertise in Google Cloud technologies.

Manual Last Updated May 26, 2020

Lab Last Tested May 26, 2020

Copyright 2021 Google LLC All rights reserved. Google and the Google logo are trademarks of Google LLC. All other company and product names may be trademarks of the respective companies with which they are associated.