Car Child Seat Alert System

Christopher Atkinson   
College of Engineering and Computer Science  
University of Michigan - DearbornDearborn, USA  
ckatkins@umd.umich.edu

Prithvi Thimmanayakanahalli Bachireddy  
College of Engineering and Computer Science  
University of Michigan - DearbornDearborn, USA  
prithvit@umich.edu

*Abstract*—Every year, an average of 37 children die in the United States after being accidently left in hot vehicles [1]. Each death is a tragedy that could have been averted with a simple and affordable engineering solution. This paper proposes a simple, affordable solution to this problem using an embedded system to monitor driver and child seats and sending an alert to the user if a child has been left alone in a vehicle. The embedded system encompasses two HX711 load cell amplifier interfaced with Arduino, one to monitor Child seat and another to Driver seat and a backend software that runs in the PC. When Child is left alone in the car for a configurable amount of time the Arduino acts as a Client and sends an alert to backend Server. The Server in turn sends the alert to AWS IoT. The AWS IoT sends an alert to Parent and Guardian Via Text and Email.

Keywords—automobile, vehicle, child, driver, hot car, software engineering, embedded system, HX711, amplifier, Arduino Nano 33 IoT, Server, alert, Email, monitor, interfaced, Client, AWS IoT, Text.

Contents

[I. Introduction 1](#_Toc26124789)

[II. Related Work and State of the Industry 2](#_Toc26124790)

[III. Software Engineering 2](#_Toc26124791)

[A. Requirements…………… 2](#_Toc26124792)

B. Software Development Methodology...…..2

[C. Design Methodology 2](#_Toc26124793)

[1) Requirements Phase 2](#_Toc26124794)

[2) Design, Implementation, and Subsystem Testing 3](#_Toc26124795)

[3) Integration and System Testing 3](#_Toc26124796)

[C. Design 4](#_Toc26124797)

[1) Firmware Design 4](#_Toc26124798)

[2) Software Design 4](#_Toc26124799)

[IV. Hardware Selection 4](#_Toc26124800)

[A. Microprocessor 4](#_Toc26124801)

[B. Sensors 4](#_Toc26124802)

[C. Other Hardware 4](#_Toc26124803)

[V. Implementation Details 5](#_Toc26124804)

[A. Hardware Implementation 5](#_Toc26124805)

[B. Firmware Implementation 5](#_Toc26124806)

[C. Software Implementation 6](#_Toc26124807)

[VI. Functionality and Testing 8](#_Toc26124808)

[A. Unit Testing 8](#_Toc26124809)

[1) Firmware Testing 8](#_Toc26124810)

[2) Backend Testing 8](#_Toc26124811)

[B. Functional Requirements Testing 8](#_Toc26124812)

[C. Performance Requirements Testing 8](#_Toc26124813)

[VII. User Manual and Maintenance 8](#_Toc26124814)

[A. User Manual 8](#_Toc26124815)

[1) Hardware 8](#_Toc26124816)

[2) Firmware 9](#_Toc26124817)

[3) Software 9](#_Toc26124818)

[VIII. Bill of Materials 9](#_Toc26124819)

[IX. Future Work 9](#_Toc26124820)

[A. Temperature sensor 9](#_Toc26124821)

[B. Adjustable Alert Intervals 9](#_Toc26124822)

[C. Additional Sensors 9](#_Toc26124823)

[X. Problems and Difficulties 9](#_Toc26124824)

[XI. Conclusion 10](#_Toc26124825)

[XII. Appendix I: Firmware Code 11](#_Toc26124826)

[XIII. Appendix II: Backend Software Code 15](#_Toc26124827)

[XIV. Amazon AWS IoT Full User Manual 17](#_Toc26124828)

# Introduction

Modern vehicles are engineering wonders, containing more than 50 pounds of copper wire [2], increasingly sophisticated microprocessors, and safety systems that were unimaginable just ten years ago. The vehicle’s advanced sensors and controls allow vehicles to alert drivers to the presence of other vehicles, perform emergency braking for the driver, and even park and drive themselves. However, car manufacturers have been very slow to include systems that monitor the vehicle for children left alone. These systems are just beginning to be sold in the 2019 and 2020 model year vehicles [3] in many GM vehicles with several other manufacturers following suit. But what about parents that own older vehicles? Options exist, but many of them are video based, adding a device that could become a projectile in an accident. This paper describes a different alert system prototype that can be constructed using a simple microprocessor, simple sensors, and an internet connection that can monitor the vehicle for both a driver and a child that can be retrofitted to any vehicle.

# Related Work and State of the Art

Currently, solutions to this problem do exist. However, these solutions are often not ideal [4]. Multiple companies make what are essentially modified baby monitors that use a video camera and connect to your phone, then send an alert if the driver walks too far from the car. Several apps also exist to alert the user [5]. Unfortunately, most of these solutions require the user to consciously enable something when they get into the car. This greatly diminishes the usefulness of the system, as people can forget to turn them on, connect their phone, etc. If people never forgot things, there would be no need for a child monitoring system in the first place. CYBEX makes a child’s car seat that can also alert a parent’s phone if the child is left alone, but it is extremely expensive. This seems like the best solution currently available, albeit an expensive one.

# Software Engineering

## Requirements

1. System Requirements
   * [SYSFUNCT001] The system shall consist of hardware and software components that have the capability of monitoring a vehicle for the presence of a driver and child as well as the capability of alerting a user via text or email.
   * [SYSFUNCT003] The system shall have a user-configurable delay for signaling an emergency event.
   * [SYSPERF001] The system shall alert a user within 30 seconds (1-sigma) of an emergency event.
   * [SYSPERF002] The system shall alert a user for the second time with 3 minutes delay if no action is taken for the first alert.
2. Embedded System Requirements
   * [ESFUNCT001] The hardware shall consist of an embedded system with one or more microprocessors, one or more sensors, and the hardware to support a wireless communication method (WI-FI or Bluetooth preferred).
   * [ESPERF001] The hardware shall transmit an alert to the backend system within 15 seconds (1-sigma) of an emergency event.
3. Backend Requirements
   * [BEFUNCT001] The software shall receive notifications wirelessly (using WI-FI or Bluetooth) from the hardware and transmit a notification to the user via text message or email.
   * [BEFUNCT002] The software shall be written in a high-level programming language such as C, Java, Python, or a similar language.
   * [BEPERF001] The software shall transmit an alert to the user within 15 seconds (1-sigma) of an alert received from the hardware.

## Software Development Methodology

The software development methodology used here is Agile Methodology because there is no defined System Specifications at the beginning. Rapid development due to time constraint. Regular adaptation to changing circumstances. Even late changes in requirements are welcomed.

## Design Methodology

The project was divided into three parts, each comprising several weeks. These parts are requirements phase, design and implementation phase, and system integration and testing. The requirements phase was focused on identifying the system and subsystem requirements. This process involves identifying the needs of the system as well as the amount of available time and creating requirements for the system that are both reasonable to accomplish and together create a functional system. The design and implementation phase involved the system-level design, including block diagrams and flow charts, as well as selecting the hardware that would be used for the project. The final phase involved combining the two parts of the systems and testing them for functionality and requirement validation. Short weekly meetings were used to collaborate between team members to make major design decisions as well as to re

### Requirements Phase

The initial phase of the design revolved around defining the system that would be developed, including requirements definition (see Requirements) for the functionality and performance of the system. The system was divided into two main parts: the embedded system containing the sensors and microprocessor, and the backend software that runs on a PC and the AWS Internet of Things (IoT), which would be used to send a text message to users. The system was laid out using a block diagram to define the parts of the hardware and the major software modules of the system (see Figure 1).



Figure 1: System Block Diagram

Once the entire system was defined, the next step was to decide upon the various parts. The hardware selection process (detailed in Section IV) analyzed various components and their pros and cons before selecting. Python was decided upon for the programming language for the backend software and Amazon Web Services (AWS) portion of the design.

### Design, Implementation, and Subsystem Testing

The second phase began with designing the Entity Relationship Diagram (ERD) for the system. This defines the relationship between the various parts of the system. Figure 2 shows the ERD for the system. The embedded system portion consists of a single microprocessor that monitors a minimum of two or more sensors. These sensors are wired for monitoring the child or the driver. There can only be one sensor for the driver, but there could be multiple sensors for monitoring multiple children. The child and driver may not be present at all times, so this part alone has a modality of zero. The backend of the system, consisting of the PC software and AWS software, is required for the system to operate (modality of one) and functions on a single instance. The embedded system must communicate with the backend to transmit alerts to the user via text. The driver must also interface with the backend to configure the system with one or more phone numbers to notify during an alert.

Child

Driver

Embedded System

Backend

Sensors

Monitor

Monitor

Communicates with

Alerts

Figure 2: ERD for the System

Once the ERD was completed, the logical flow of the code was designed using data flow diagrams (Figure 3 and Figure 4) and flow charts (see Section C). The actual design of the code had to take into consideration the programming language in use. This was not important for the Python code, but it was very important for the Arduino code design because of the way code works on these devices. The setup() function is only run at startup, so any one-time configurations needed to be performed here, while any continuous functions needed to be performed in the loop() function that runs continuously. Using the flow charts in Section C, the code was written for the separate parts of the system. The hardware and parts of the code were then tested separately (see Section VI) for functionality using the test plans detailed Section VI.A.



Figure 3: Level 0 Data Flow Diagram

Sensors

Sensor Status

Embedded System

Send an Alert

Alert

AWS IoT

Send an Alert

Figure 4: Level 1 Data Flow Diagram

### Integration and System Testing

The final phase of the project was focused on integrating the various parts of the system and testing them together (system testing). Due to the highly separated functions of the system, there was only a single point of connection between the two pieces consisting of a TCP connection between the firmware and server used for sending alerts. Once the parts were integrated, the system was tested using the requirements specified in phase one.

## Design

### Firmware Design

The firmware monitors the sensors for the presence of a child and, in the case of a child being left alone, sends an alert to the backend software that will then relay the alert to the user. Thus, the firmware must first monitor for the presence of a child. If no child is present, it is not required to operate at all and must periodically monitor if that status has changed. If a child is present, the system must monitor both the child and driver. The design also requires a timer before sending an alert to prevent false alarms if the driver is filling the car with fuel, loading or unloading the car, or getting out of the car to remove the child, which can take several minutes. The flow chart in Figure 5 shows the flow of the design.

No

Yes

Reset timer

Check Child

Check Driver

Yes

No

Send Alert

Timer

Yes

No

1. Check for child
2. Check for adult
3. If child present and driver is not, set a timer
4. Keep checking for driver or if child was removed
5. If child still alone at end of timer, send alarm

Figure 5: Firmware Flow Chart

### Software Design

The software runs on a PC between the embedded system and the AWS IoT. As a result, most of what the software does is monitor incoming messages and relay them as necessary. The software must establish connections to the various components then begin the monitoring process. The software design was based on the flow chart in Figure 6. The software checks for a connection, receives a message from the embedded system, and then examines the message, then either sends an alert, or sleeps for a time before repeating the process.

No

Yes

Wait & Retry

Send Alert

Heartbeat

Alert?

No

Yes

Figure 6: Backend Software Flow Chart

# Hardware Selection

## Microprocessor

During hardware selection several types of microprocessors were examined, including PIC processors, a Raspberry Pi, and Arduino. Ultimately an Arduino Nano 33 IoT was selected for several reasons. First, it is significantly easier to work with than the PIC, especially because a Wi-Fi connection was a requirement. The Arduino Nano IoT contains a Wi-Fi and Bluetooth module on the board as well as included libraries, making this process much easier. The Arduino was also affordable, small, and had more than enough processing power to complete the required tasks.

## Sensors

The purpose of the sensors is to provide the Arduino with data about the presence of a driver and/or child in the vehicle. Strain gauges were selected to perform this task. A strain gauge is a piece of material (usually metal) that has a flexible backing attached [6]. When this material is bent even slightly, it alters the resistance of the device, producing a minor, but measurable, difference in the voltage across the device.

## Other Hardware

Due to the extremely small voltage change generated by applying force to a strain gauge, the Arduino requires some assistance. This comes in the form of a microchip called an HX711. This chip combines one or more strain gauges into a Wheatstone bridge [7] that amplifies the signal. In addition, the HX711 has a 24-bit ADC [8] for digitizing the signal with greater sensitivity than the Arduino’s 12-bit ADC.

# Implementation Details

The Vehicle Child Alert System consists of several distinct parts that have been divided into multiple parts. This includes the hardware design, consisting of the strain gauges, HX711 microprocessors, wires, and Arduino, the Firmware design, which consists of the code that runs on the Arduino to monitor the gauges and relay alerts to the backend, and the backend, which alerts the user in case of an emergency.

## Hardware Implementation

The hardware implementation connects the various pieces of the hardware physically. Figure 7 shows the wiring diagram for the system. For two 3-wire strain gauges to work properly, one white and one black wire from each gauge must be connected and run to the HX711. The remaining white and black wires are similarly connected. These wires are then attached to E+ or E- connection on the HX711 board. The red wires from the strain gauges are connected to the A+ and A- connections on the HX711 board, completing the circuit. The HX711 combines these connections to form the Wheatstone bridge internally. The HX711 is then connected to the Arduino with 4 wires. Power and Ground are connected to pins 2 and 14, respectively. The SCK (clock) and DT (Data Out) pins from the HX711 are then connected to two digital pins on the Arduino board. The Arduino board, in turn, is powered from a USB connection.

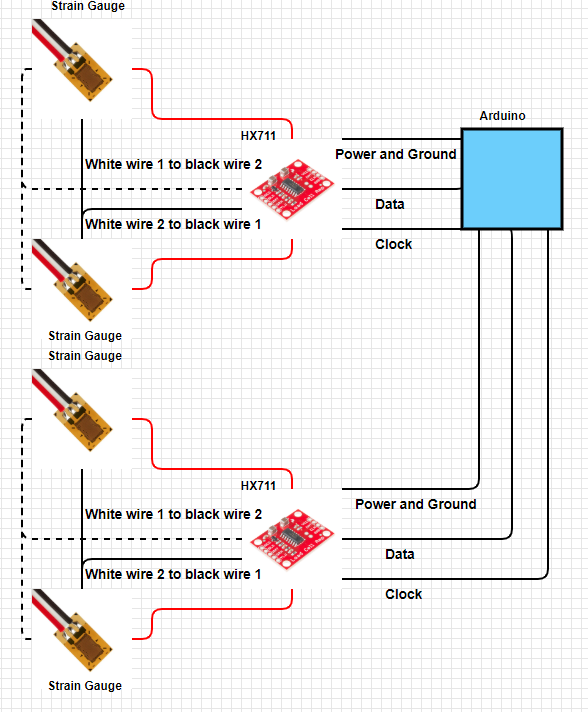


Figure 7: Hardware Wiring Diagram [drawn using 9]

## Firmware Implementation

#### Major Functions

The code is organized very simply, as most Arduino code is. There is a setup() function that performs all of the one-time operations. This includes opening the serial interface between the PC and the Arduino (when developing and debugging), setting up the HX711’s pin configurations, and connecting to the internet (in this case, Wi-Fi).

The other major function is the loop() function. This is the typical continuous loop found in most embedded systems that contains most of the functional code. This section calls several other functions that monitor the sensors, check the Wi-Fi status, and send an alert if necessary.

#### Minor Functions

The connectWifi() function performs the initial connection to the Wi-Fi network, or can be called if a connection is lost at any time. This uses the configured networks SSID and password to connect to a WPA2 encrypted Wi-Fi network and blink the LED when it connects.

The dispWifiStatus() function displays the status of the Wi-Fi connection to the serial monitor, including the network SSID and the Arduino’s assigned IP address.

The checkWifi() function checks that the Arduino is still connected to the network and tries to reestablish its connection if it has been lost using the connectWifi() function.

The checkDriver() and checkChild() functions check the values from the sensors to determine if the driver or child is currently in the vehicle, returning a simple Boolean answer.

The checkSensors(int delayTime) function is the most important function of the code. This is the section where the logic that monitors the driver and child is located. See Figure 5 for the flow chart. This code consists of a while() loop that breaks when the logic determines that an alert should or should not be sent. It first checks for the presence of a child. If there is no child, it breaks the loop and returns to the loop() without sending an alert. If there is a child, it then checks for a driver. If a driver is present, then it breaks the while loop and returns to loop() without sending an alert. Otherwise, if there is a child and no driver, it delays for the specified delayTime before performing the checks again. If the driver has returned to their seat of the child has been removed from theirs, it breaks the while loop and returns to loop() with no alert being sent. Otherwise it returns a true for alert and an alert will be sent to the driver when it returns to loop().

Setup()

Serial Library

HX711 Library

Begin()

println()

Set\_scale()

tare()

ConnectWifi()

Wi-Fi Library

begin()

Figure 8: Arduino setup() Function Call Graph

Loop()

CheckSensors()

SendAlert()

delay()

Figure 9: Arduino loop() function Call Graph

## Software Implementation

Device Gateways

Amazon SNS

Message Broker

Rule Engine

Security and Identity

Subscribers

Figure 10: AWS IoT Module Detailed Design

The software for this project (Device Gateway in Figure 10) was written in Python, so the body of code is the \_\_main\_\_ function. This code connects the firmware to the AWS IoT using a protocol called MQ Telemetry Transport (MQTT) to perform this connection. The “MQ” represents an IBM product line called “MQSeries”[11]. The code then opens a Secure Sockets Layer (SSL) connection to the AWS IoT using the ssl\_alpn() function. ALPN stands for Application Layer Protocol Negotiation. This requires a public and private key configuration to encrypt the connection between the local PC and the AWS IoT. Once this connection has been made, the programs starts a monitoring loop for the MQTT connection to receive messages from the system. Once this has been started, a while loop continually monitors the received messages. These messages consist of a single number that relays whether there is an alert situation (2) or not (anything other number). If an alert message is received, the software sends a message to AWS IoT to send an alert message.

\_\_main\_\_()

SSL Library

MQTT Library

Socket Library

Client()

Tls\_set\_context()

Connect()

Loop\_start()

Publish()

Ssl\_alpn()

Create\_default\_context()

Set\_alpn\_protocols()

Load\_verify\_locations()

Load\_cert\_chain()

Socket()

Accept()

Figure 11: Software Backend Function Call Graph

The AWS IoT message broker makes it possible for clients to communicate with AWS IoT and for AWS IoT to communicate with clients. Clients send data by publishing a message on a topic. Clients receive messages by subscribing to a topic. When the message broker receives a message, it forwards the message to all clients subscribed to the topic.

Devices connected to AWS IoT are represented by AWS IoT thing in the AWS IoT registry.

An AWS IoT policy is a JSON document that contains one or more policy statements. Each statement contains:

* Effect, which specifies whether the action is allowed or denied.
* Action, which specifies the action the policy is allowing or denying.
* Resource, which specifies the resource or resources on which the action is allowed or denied.

Each device must be connected and registered with AWS IoT. This process is detailed in Figure 12 and the following steps:

1. To register the thing, create a single thing. To create a thing, we should add a certificate for thing.
2. One-click certification is selected and activated in which the AWS creates a certificate for thing, private key and a root CA for AWS IoT. Download and save the certificates for further process.
3. Create a Policy, attach the policy and thing to the certificate.
4. Thing acts as Message Broker.

Device Gateway

Certificates

Use the Certificates to connect to Thing

Policy

Attach to Certificate

Thing

Interconnected to Certificate

Attach to Certificate

Connect to Thing

Message Broker

Acts as

Figure 12: AWS IoT Device Registration

Rules give devices the ability to interact with AWS services. Rules are analyzed and actions are performed based on the MQTT topic stream. Rules Augment or filter data received from a device and Send a push notification to all users using Amazon SNS. A rule, like the one used for this project, (detailed in Figure 13) was created by:

1. Create a rule by selecting the SQL version and modifying the rule query statement according to actions required.
2. Send a message as an SNS push notification action is added to rule by giving the ARN of the created SNS topic.
3. Permission is given to access and publish by creating and updating a role.

Message Broker

Send data

Role

Gives permission to publish

SNS Topic

Attach topic ARN to Rule

Amazon SNS

Topic is Created

Send Filtered data when triggered

Rule

Figure 13: AWS IoT Rule Engine

To send a text message, a service called Amazon SNS was used. To work properly for the alert message, a topic must be generated. This is done by performing the following steps that are also detailed in Figure 14.

1. Create an SNS topic, An ARN is created for SNS topic.
2. To receive SMS messages on cell phone, subscribe to the Amazon SNS topic.
3. Subscription is done by enabling the SMS protocol and entering the endpoint which is the cell number.

Amazon SNS

SNS Topic

Create a Topic

Subscription

Add Subscribers

Subscribers

Add Endpoints

Send Data

Figure 14: Amazon SNS Usage

# Functionality and Testing

## Unit Testing

A test plan was created that would test the various parts of the subsystems before integration. Unit testing was then performed on each part of the subsystems before they were combined.

### Firmware Testing

The first test performed was to ensure that the Arduino could connect to the Wi-Fi. The tested network used WPA2 network encryption protocol. This was successfully completed using the wifinina library. The one caveat is that there was no support for WPA2 Enterprise, which uses a network ID, user ID, and password. It appears that this system was deprecated at some point.

The next tests were wiring the strain gauges and testing for a voltage differential using a voltmeter. This required wiring the strain gauges together, wiring the HX71 connections, and powering the HX711 from the Arduino. A voltmeter was connected to the A+ and A- pins on the HX711 and the voltage changed as pressure was applied to the gauges. The tests were deemed successful.

The HX711 communication with the Arduino was the next step of testing performed. This involved using the HX711 library on the Arduino. Readings were taken by feeding a clock on the SCK pin and receiving data on the DT pin. The results were displayed on the serial monitor. The test was successful.

The final test of the Arduino subsystem was to make sure that the alert logic and timer worked. This was done by putting weight on the driver and child sensors and then removing the driver for more than 30 seconds. When this time had passed, the serial monitor reported that an alert message was being sent. This test was successful. The testing document has been embedded below.



### Backend Testing

Most of the backend testing involved checking that the various parts of the AWS IoT were working correctly. The first test created was to make sure that the account was created and that the Thing was registered correctly. This was done by logging in and checking that the certificate was identical to the expected value, which it was. The next test was making sure that the created policy was visible in the AWS console, which it was. This test was also successful.

The policy and Thing then had to be connected to each other and verified. This was done in the AWS console and the policy and Thing matched the certificates as expected. The next step was to publish a message to the created topic, which was verified in the AWS console.

The final phase of testing this subsystem was testing the AWS SNS system that allows the system to send alerts to users. The first step in doing so was to create an SNS topic and verify that it was visible in the console, which it was. The next step was to create an SNS subscription to the end users’ mobile devices. This was done and verified in the console. The final test of this subsystem was to send a message to the users’ mobile devices, which was received successfully. All tests were successful. The testing document is embedded below.



## Functional Requirements Testing

* [SYSFUNCT001] - This requirement is met by the successful performance of the system. It uses a hardware and firmware system to monitor for the presence of a driver and a child and alerts the user via text and email.
* [SYSFUNCT002] - The user can configure a delay into the program by changing the checkTime variable in the code using milliseconds.
* [ESFUNCT001] - Using a series of strain gauges and the Arduino Nano IoT that can connect to a Wi-Fi network, this requirement was met successfully.
* [BEFUNCT001] - This requirement was met by running Python code on a local PC connected to a Wi-Fi network, then using Amazon Aws IoT to transmit the alert messages to the user.
* [BEFUNCT002] - This requirement was met by using Python as the programming language for the backend.

## Performance Requirements Testing

* [SYSPERF001], [ESPERF001], [BEPERF001] – These three requirements are all related, as the subsystem level requirements flow down from the system-level requirement. During performance testing, the alert was received at the endpoint in less than 4 seconds, vastly exceeding the requirements.

# User Manual and Maintenance

## User Manual

### Hardware

The hardware for this system consists of the strain gauges, HX711 boards, an Arduino board, and wires and connectors. To begin setting up the correct configuration, the strain gauges must be connected properly. Each seat is monitored by a pair of strain gauges in a half-Wheatstone bridge. This is performed by the HX711 chip, but it requires the user to correctly wire the strain gauges together. In this configuration, only four (4) of the six (6) connectors are used. To simplify the explanation, the gauges will be referred to as strain gauge 1 and strain gauge 2. The white wire from strain gauge 1 should be wired to the black wire of strain gauge 2. These wires can then be connected to A+ or A-. However, the red wire from strain gauge 1 (with the white wire) MUST be connected to the same polarity (E+/E-) as its white wire. Strain gauge 2 is wired similarly and connected to the remaining strain gauge, keeping the red wire with the same polarity as the black wire. The HX711 board must then be wired to power and ground. The ground connector must go to either pin 14 or 19 while power must be routed to pin 2 on the Arduino board (3.3 Volt output). The connectors for DT and SCK are then wired to pins 2 and 3, respectively, for the child sensor and pins 4 and 5, respectively, for the driver’s sensor. The power for the Arduino can come from a USB connector, DC power supply, or a battery, but it should be noted that the Arduino Nano 33IoT runs on 3.3 volts NOT the standard Arduino voltage of 5 volts, so be wary if using anything other than a USB connector.

### Firmware

The firmware component consists of the Arduino board’s code that monitors the sensors and transmits an alert to the backend software. To configure the board, the Arduino IDE [10] must be installed on a PC. The WifiNINA and HX711 libraries must be installed from inside the IDE by going to Tools -> Manage Libraries, then entering “wifinina” and installing the library, and then doing the same for the HX711 library. The driver for the Arduino must be installed in a similar manner by going to Tools -> Board -> Board Manager and searching for “nano”, then selecting “Arduino SAMDBoards” and installing it making sure that Arduino Nano is listed in the description. Once this has been done, the board should be visible by the computer and have all the correct libraries installed.

Once the libraries are installed and the Arduino has been connected to the PC, the code can be loaded to the board via USB. The file “alertproject.ino” contains code for the Arduino board. To configure the Arduino to connect to a Wi-Fi network, the SSID and password for a WPA2 network must be input into the networkSsid and networkPass variables. The next configuration change is the delay time for detecting if a driver has left the child in the car alone. The variable is in milliseconds and the default is 30,000 ms (30 seconds). In practice this would likely be much longer to allow a driver to pump gas, remove the child from the seat, unload the car, etc.

The final configuration that the user must make is to configure the strain gauges. The variances in manufacturing require that this be done for each strain gauge pair to make sure the numbers are correct. Using the program at the bottom of [12], the strain gauges can be calibrated by starting the system with no weight and then adding a known weight and using the serial monitor and the +/- keys to increase or decrease the calibration factor. When the printed value matches the known weight, the calibration factor that is printed should be used to calibrate that strain gauge assembly at startup.

### Software

The software and Amazon IoT setup and operation is a detailed process and has been documented in XII. The short version will be documented here. The first step is to sign into the AWS IoT console (or make an account, if necessary) and then register the device and create a policy. Each device must have a certificate generated for it along with a key pair. These must then be downloaded to the local PC. Once this has been done, the device must have a policy set for it.

Once the device has been setup, the system to transmit a text message must be created using Amazon SNS. Once logged into the SNS console, create a topic and subscribe to the topic. The protocol for communication must be chosen (in this case, SMS) and a number input for receiving the message. The next step is to create a rule to send the push notification.

Python was the chosen programming language for this project, so Anaconda 3.7 must be installed to function properly. The included code (Section XIII) can then be run in the Spyder IDE.

# Bill of Materials

|  |  |
| --- | --- |
| Item | Quantity |
| Arduino Nano 33 IoT | 1 |
| 50 kg strain gauge | 4 |
| HX711 ADC | 2 |

# Future Work

## Temperature sensor

A temperature sensor connected to the Arduino would be a possible source for future work. This number could then be reported along with the alert message.

## Adjustable Alert Intervals

An adjustable alert interval is another future improvement that could be made. Ideally this would use the temperature sensor from the previous section to adjust the alert interval based upon the temperature inside the vehicle. This would provide an earlier alert when the temperature reaches more dangerous levels.

## Additional Sensors

Additional (preferably plug-and-play) sensors would improve the system, as they would allow for the monitoring of multiple children or even multiple adults. For instance, if an adult is in the passenger seat but the driver is not present (if the vehicle were getting refueled, for example), then there would be no need to send an alert. This would likely require code in the firmware to measure the amount of weight on the sensors and some threshold weight level that determines if the person is a child or an adult.

# Problems and Difficulties

During the implementation of the child alert monitoring system, many difficulties were encountered, especially in regards to hardware integration. One of the most common causes of problems was due to poor documentation of hardware components, including the strain gauges and Arduino. While the solutions to these problems were generally extremely simple, the process of finding hardware problems was very time-consuming.

The initial difficulty with the hardware was with deprecated software. The Arduino Wi-Fi library had changed fairly recently, adding a wrapper layer called wifiNina around the previous wifi library. This change was not particularly well-documented, requiring fairly extensive research and debugging to find the problem, which was fixed by adding a single library and include file, then slightly altering a function call.

The strain gauges used for this project had little to no documentation available, while the HX711 chip had a small datasheet that was somewhat lacking in detail. The strain gauge had to be connected in a very specific manner to function properly. Initially, the connection between the white and black wires and the HX711 was reversed, causing incorrect readings from the sensors. This was corrected by close examination of the wiring diagram for the Wheatstone bridge and the HX711 datasheet and moving the wires to different pins as necessary.

# Conclusion

The purpose of this project was to build a child car seat monitoring system that uses sensors to monitor for the presence of a driver and a child and to alert the driver if they have left a child in the vehicle. A list of requirements was created to make this possible and to ensure it was done in a timely manner. The Agile software methodology was used to divide the project into several phases and subsystems so that work could be performed in parallel. Once the individual subsystems were completed and tested, they were combined into a system that successfully met the initial requirements.

##### References

1. Willingham, A. (2018, July 20). *Hot Car Deaths Child Charts Graphs*. Retrieved 11 28, 2019, from CNN: https://www.cnn.com/2018/07/03/health/hot-car-deaths-child-charts-graphs-trnd/index.html
2. Sprovieri, J. (2014, July 1). *Wire Harness Recycling*. Retrieved 11 28, 2019, from Assembly Mag: https://www.assemblymag.com/articles/92263-wire-harness-recycling
3. Miller, S. S. (2019, 8 9). *High-tech alarms go off when kids are left in hot cars*. Retrieved 11 28, 2019, from Mashable: <https://mashable.com/article/car-seat-alarms-prevent-hot-car-death>
4. Baldwin, D. (2019, 8 21). Life-Saving Car Seat Alarms That Remind Parents There’s a Baby in the Back Seat. Retrieved 28 11, 2019, from Fatherly: https://www.fatherly.com/gear/best-car-seat-alarms/
5. Marcoux, H. (2019, 7 18). 5 safety apps that remind parents there's a baby in the back seat. Retrieved 11 28, 2019, from Motherly: https://www.mother.ly/news/best-car-seat-sensor-app?rebelltitem=5#rebelltitem5
6. Wikipedia. (2019, 11 07). *Strain Gauge*. Retrieved 11 28, 2019, from Wikipedia: <https://en.wikipedia.org/wiki/Strain_gauge>
7. Wikipedia. (2019, November 27). *Wheatstone Bridge*. Retrieved 11 28, 2019, from Wikipedia: https://en.wikipedia.org/wiki/Wheatstone\_bridge
8. Avia Semiconductor. (2017, 1 1). *HX711*. Retrieved 11 28, 2019, from alldatasheet: https://www.alldatasheet.com/datasheet-pdf/pdf/1132222/AVIA/HX711.html
9. Digikey. (2014, 5 28). *Schemeit*. Retrieved 11 28, 2019, from Digikey: https://www.digikey.com/schemeit/project
10. *Software*. (2015, 5 15). Retrieved 11 28, 2019, from Arduino: https://www.arduino.cc/en/Main/Software
11. Wikipedia. (2019, 11 5). *MQTT*. Retrieved 11 28, 2019, from Wikipedia: https://en.wikipedia.org/wiki/MQTT

[12] Fahad, E. (2019, 4 27). *HX711 Load cell or Strain Gauge and Arduino*. Retrieved 11 28, 2019, from Electronic CLinic: <https://www.electroniclinic.com/hx711-load-cell-or-strain-gauge-and-arduino>

[13] AWS IoT Developer Guide Reterived 11 28, 2019, from AWS Console: <https://docs.amazonaws.cn/en_us/iot/latest/developerguide/iot-dg.pdf>

[14] Annaconda Python Software Download, Reterived 11 28, 2019 from Anaconda Distributions: <https://www.anaconda.com/distribution/>

[15] Anaconda Python Documentation for reference, Reterived 11 28, 2019 from Anaconda Distribution <https://docs.anaconda.com/anaconda/>

[16] AWS IoT Reterived 11, 28, 2019 from AWS IoT Console: <https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html>

[17] TCP/IP Socket Programming for server connction, Reterived 11,28,2109:

<https://pymotw.com/2/socket/tcp.html>

[18] Setting Calibration Factor for load cells, Reterived 11,28,2019: <https://gist.github.com/matt448/14d118e2fc5b6217da11>

[19] Connecting the HX711 to a three wire load Cells , Reterived 11,28,2019<https://electronics.stackexchange.com/questions/199487/connect-hx711-to-a-three-wire-load-cell>

[20]

# Appendix I: Firmware Code

#include "HX711.h"

#include "Queue.h"

// HX711 circuit wiring//Libraries

#include <SPI.h>

#include <WiFiNINA.h>

#include <HX711.h>

#define DT 2

#define SCK 3

#define DT1 4

#define SCK1 5

float childsenscalibration\_factor = -7050;

float driversenscalibration\_factor = 7050;

HX711 childseatsens;

HX711 driverseatsens;

char networkSsid[] = "My Home";

char networkPass[] = "trishok4";

//Time allowance for driver to return to vehicle or remove child before an alert is sent

static int checkTime = 30000;

int status = WL\_IDLE\_STATUS;

IPAddress server(192,168,1,3); // Google

//Sensor check variables

struct sysStatus{

bool driver = false;

bool child = false;

};

// Initialize the client library

WiFiClient client;

void setup() {

// initialize LED

pinMode(LED\_BUILTIN, OUTPUT);

Serial.begin(9600);

while (!Serial)

Serial.println("Starting Program.");

//Connect to the server using WiFi

connectWifi();

childseatsens.begin(DT, SCK);

driverseatsens.begin(DT1, SCK1);

childseatsens.set\_scale();

childseatsens.tare(); //Reset the scale to 0

childseatsens.set\_scale(childsenscalibration\_factor);

driverseatsens.set\_scale();

driverseatsens.tare(); //Reset the scale to 0

driverseatsens.set\_scale(driversenscalibration\_factor);

}

void connectWifi(){

//Connect to the network and server

while (status != WL\_CONNECTED) {

Serial.print("Attempting connection to network: ");

Serial.println(networkSsid);

status = WiFi.begin(networkSsid, networkPass);

// Delay to establish connection

delay(10000);

IPAddress ardAddr = WiFi.localIP();

Serial.println(ardAddr);

//if connected, blink the LED

digitalWrite(LED\_BUILTIN, HIGH);

delay(1000);

digitalWrite(LED\_BUILTIN, LOW);

delay(1000);

}

}

void dispWifiStatus() {

// display network information

Serial.print("Network: ");

Serial.println(WiFi.SSID());

// display Arduino's IP

IPAddress ipAddr = WiFi.localIP();

Serial.print("IP Address: ");

Serial.println(ipAddr);

}

void checkWifi(){

if(!client.connected()){

Serial.println("Connection Lost. Trying to Reestablish Connection.");

connectWifi();

}

}

//Check for the presence of the driver if driver is not there for more than 1 min then send status as false

//Inputs: None

//Outputs:

// bool driver:

// true: Driver is present in vehicle

// false: Driver is not in vehicle

bool checkDriver(){

//int driverseatavg=0;

int driverseatval=255;

int count=0; /\* driver not present count 1ms\*60=60 sec \*/

if(driverseatsens.is\_ready()){

Serial.print("Driver Scale Reading: ");

driverseatval=(int)driverseatsens.get\_units()\*0.453592;

//driverseatsensqueue.dequeue();

if(driverseatval == 0)

{

count += 1;

}

else

{

count=0; /\*reset timer \*/

}

Serial.print(driverseatsens.get\_units()\*0.453592, 3);

Serial.print(" kg"); //Change this to kg and re-adjust the calibration factor if you follow SI units like a sane person

Serial.println( );

if(count == 60)

{

return false;

}

}

else{

Serial.println("Driver Scale Not reading .");

}

return true;

}

//Check for the presence of the child

//Inputs: None

//Outputs:

// bool child:

// true: Child is present in vehicle

// false: Child is not in vehicle

bool checkChild(){

// int childseatavg=0;

int childseatval=255;

int count=0; /\* driver not present count 1ms\*60=60 sec \*/

if(childseatsens.is\_ready()){

Serial.print("Child Scale Reading: ");

childseatval=(int)childseatsens.get\_units()\*0.453592;

//childseatsensqueue.dequeue();

//childseatsensqueue.enqueue(childseatval);

Serial.print(childseatsens.get\_units()\*0.453592, 3);

Serial.print("kg"); //Change this to kg and re-adjust the calibration factor if you follow SI units like a sane person

Serial.println( );

if(childseatval == 0)

{

count += 1;

}

else

{

count=0; /\*reset timer \*/

}

if(count == 60)

{

return false;

}

}

else{

Serial.println("Child scale Not reading .");

}

return true;

}

// Function to check the sensor.

//Inputs:

// delayTime - controls how long to check for driver presence before sending an alert

//Outputs:

// bool alert - returns the status of the alert check.

// 1 = no alert (driver present or child is not)

// 2= alert because child is alone and driver has not reentered vehicle over

int checkSensors(int delayTime){

int alert = 1;

//Create the structure to keep track of sensor status and initialize to false

sysStatus sensStatus;

sensStatus.driver = false;

sensStatus.child = false;

bool check = false;

//Loop to check the status of the system

while(check == false){

//if there's no child, immediately stop checking since there's no reason to proceed

if(checkChild() == false){

check == true;

alert == 1;

}

//otherwise check for the driver

else if(checkDriver() == true){

check == true;

alert == 1;

}

//otherwise, child is alone in the vehicle. Use delayTime to delay for a set amount of

//time to give driver time to get back in vehicle or remove child

else{

delay(delayTime);

if(checkChild() == false){

check == true;

alert == 1;

}

//otherwise check for the driver

else if(checkDriver() == true){

check == true;

alert == 1;

}

else{

check == false;

alert == 2;

}

}

}

return alert;

}

//Send an alert that the child is alone

//Inputs: None

//Outputs: None

void sendAlert(void){

Serial.println("Sending alert.");

if (!client.connect(server, 80)) {

Serial.println("Connection to host failed");

return;

}

Serial.println("Connected to server successful!");

client.print("Alert");

client.stop();

}

void loop() {

//Check to make sure the connection still exists

//checkWifi();

// Make a HTTP request:

int alertval=0;

alertval=checkSensors(checkTime);

//sendAlert(alertval);

if(alertval == 2)

{

sendAlert();

}

delay(1000);

}

# Appendix II: Backend Software Code

# -\*- coding: utf-8 -\*-

"""

Created on Mon Nov 11 13:39:12 2019

@author: Trishok

"""

from \_\_future\_\_ import print\_function

import sys

import ssl

import time

import os

import logging, traceback

import paho.mqtt.client as mqtt

import socket

fileDir = os.path.dirname(os.path.realpath('\_\_file\_\_'))

IoT\_protocol\_name = "x-amzn-mqtt-ca"

aws\_iot\_endpoint = "a2gw161u0ey957-ats.iot.us-west-2.amazonaws.com" # <random>.iot.<region>.amazonaws.com

url = "https://{}".format(aws\_iot\_endpoint)

ca = os.path.join(fileDir, 'VeriSign-Class-3-Public-Primary-Certification-Authority-G5.pem')

cert = os.path.join(fileDir, 'a407f17540-certificate.pem')

private = os.path.join(fileDir, 'a407f17540-private.pem')

logger = logging.getLogger()

logger.setLevel(logging.DEBUG)

handler = logging.StreamHandler(sys.stdout)

log\_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(log\_format)

logger.addHandler(handler)

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.bind(('192.168.1.3', 80))

s.listen(0)

def ssl\_alpn():

try:

#debug print openssl version

logger.info("open ssl version:{}".format(ssl.OPENSSL\_VERSION))

ssl\_context = ssl.create\_default\_context()

ssl\_context.set\_alpn\_protocols([IoT\_protocol\_name])

ssl\_context.load\_verify\_locations(cafile=ca)

ssl\_context.load\_cert\_chain(certfile=cert, keyfile=private)

return ssl\_context

except Exception as e:

print("exception ssl\_alpn()")

raise e

if \_\_name\_\_ == '\_\_main\_\_':

topic = "ECE574\_Project"

try:

mqttc = mqtt.Client()

ssl\_context= ssl\_alpn()

mqttc.tls\_set\_context(context=ssl\_context)

logger.info("start connect")

mqttc.connect(aws\_iot\_endpoint, port=443)

logger.info("connect success")

mqttc.loop\_start()

message = "Hey you left you Child in the Car please Hurry"

msgflag = False

while True:

client, addr = s.accept()

print("Got a connection from %s" % str(addr))

if client:

content = client.recv(32)

alertval=int(content)

if alertval == 2:

print("Child alert:")

mqttc.publish(topic, message)

time.sleep(1)

except Exception as e:

logger.error("exception main()")

logger.error("e obj:{}".format(vars(e)))

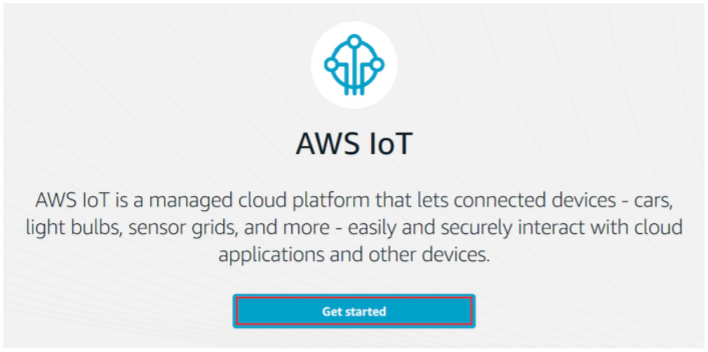
logger.error("message:{}".format(e.message))

traceback.print\_exc(file=sys.stdout)

# Amazon AWS IoT Full User Manual

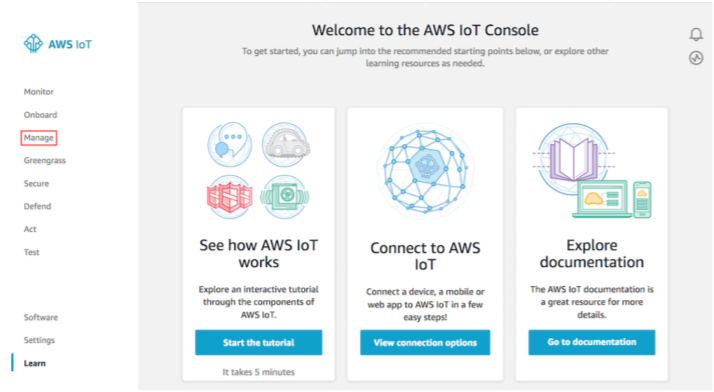
1. Sign into the AWS IoT Console

* Open the [AWS home page](https://aws.amazon.com/) and choose Sign into the Console.
* Follow the online instructions. Part of the sign-up procedure involves providing a valid Email address and password.
* Sign into the AWS Management Console and open the AWS IoT console.
* On the Welcome page, choose Get started.

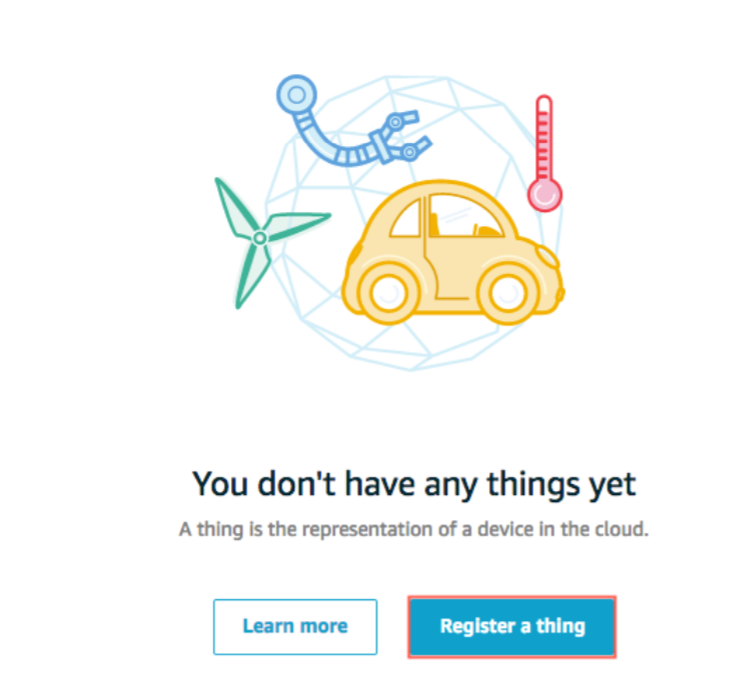


1. Register a Device(Thing) and Policy

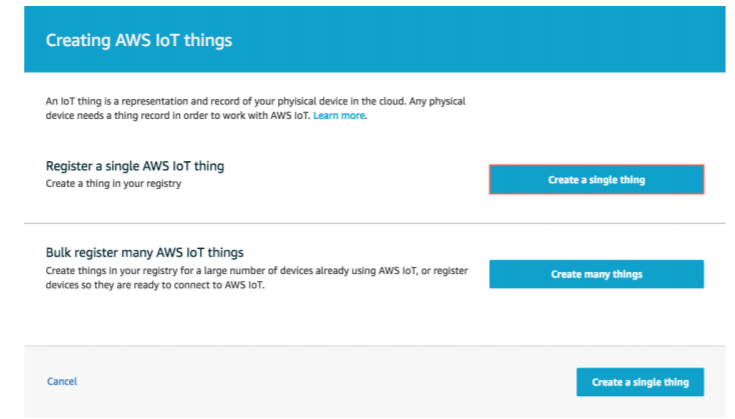
* On the Welcome to the AWS IoT Console page, in the navigation pane, choose Manage.



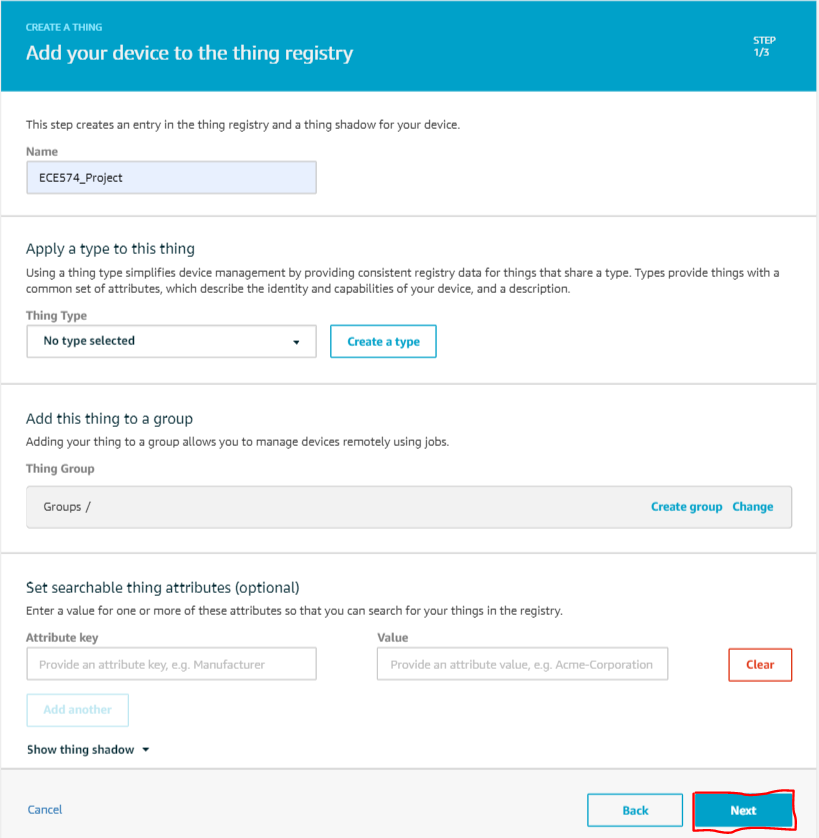
* On the You don't have any things yet page, choose Register a thing.



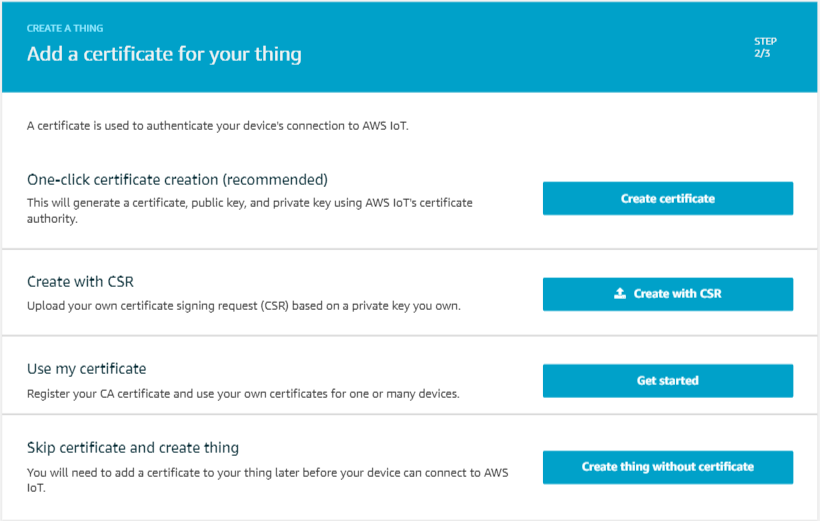
* On the Creating AWS IoT things page, choose Create a single thing.



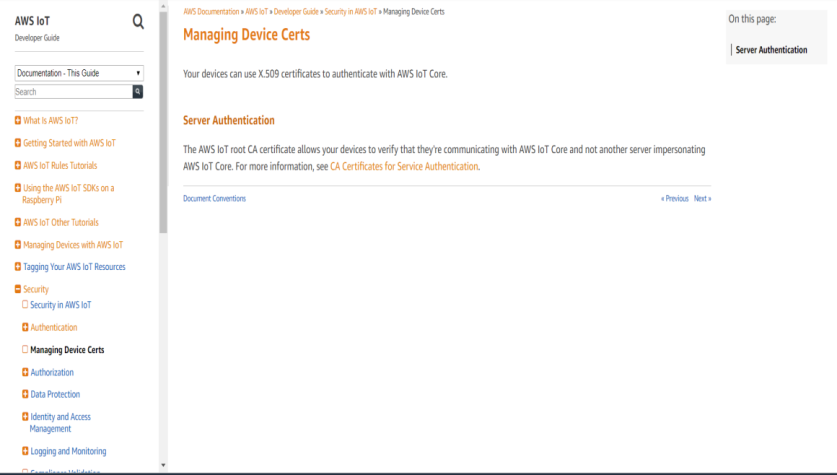
* On the Create a thing page, in the Name field, enter a name for your thing, such as ECE574\_Project. Choose Next.



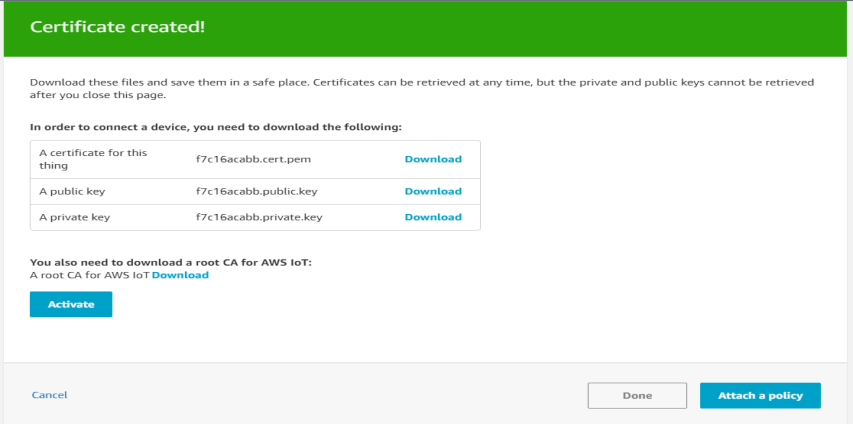
* On the Add a certificate for your thing page, choose Create certificate(one-click certificate creation). This generates an X.509 certificate and key pair.



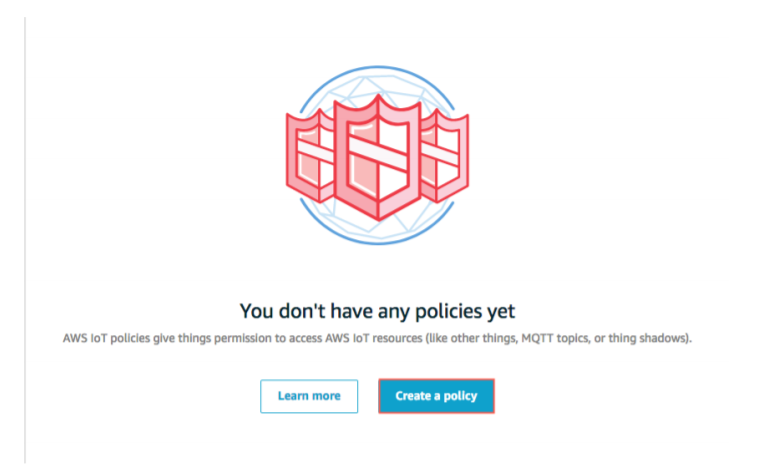
* On the Certificate created! page,
  + Choose Download A certificate for your Thing
  + Choose Download Public and Private key.
  + Choose Download A root certificate authority (CA) for AWS:
    - A new webpage is displayed.



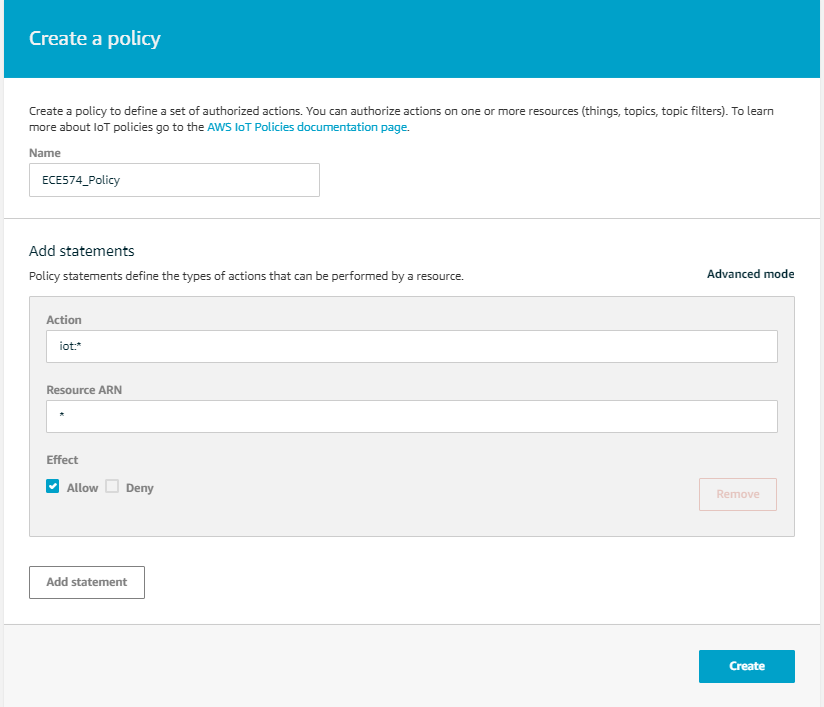
* + - Choose CA certificates for Server Authentication
    - Choose RSA 2048 bit key: VeriSign Class 3 Public Primary G5 root CA Certificate. This will display a page with the text of the root CA certificate. Copy this text and paste it into a file named VeriSign Class 3 Public Primary G5 root CA Certificate.pem.
* Choose Activate to activate the X.509 certificate, and then choose Attach a policy.



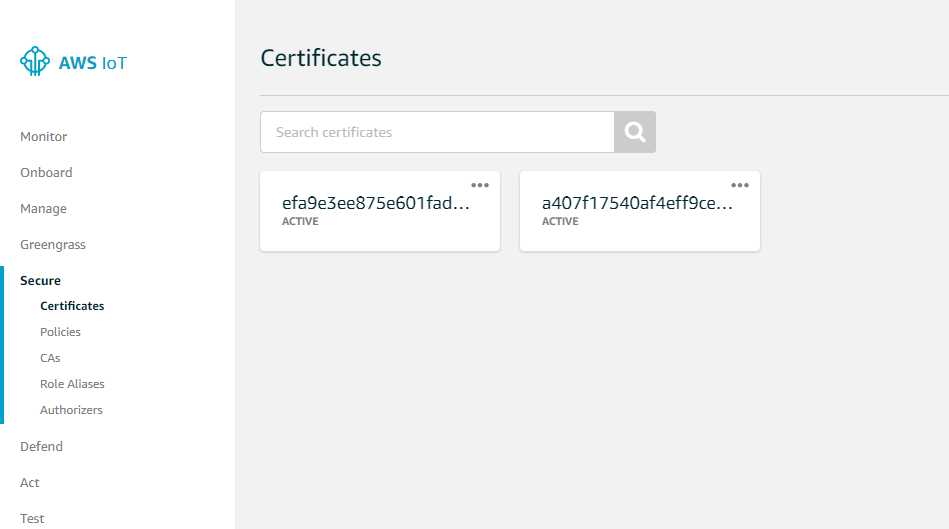
* On the Add a policy for your thing page, choose Register Thing.
* On the AWS IoT console, in the navigation pane, choose Secure, and then choose Policies. Choose Create.



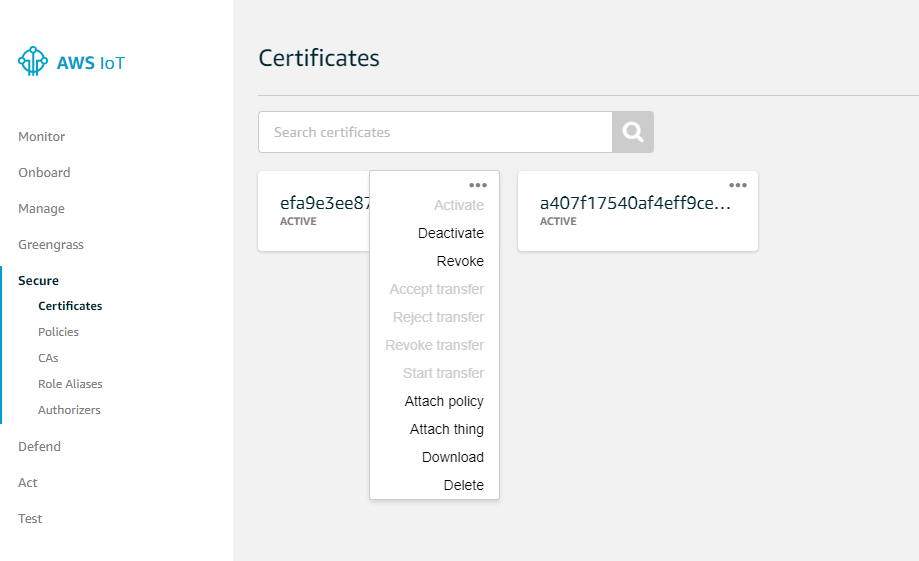
* On the Create a policy page:
  + Enter a Name for the policy, such as ECE574\_Policy.
  + For Action, enter iot:\*. For Resource ARN, enter \*.
  + Under Effect, choose Allow, and then choose Create.



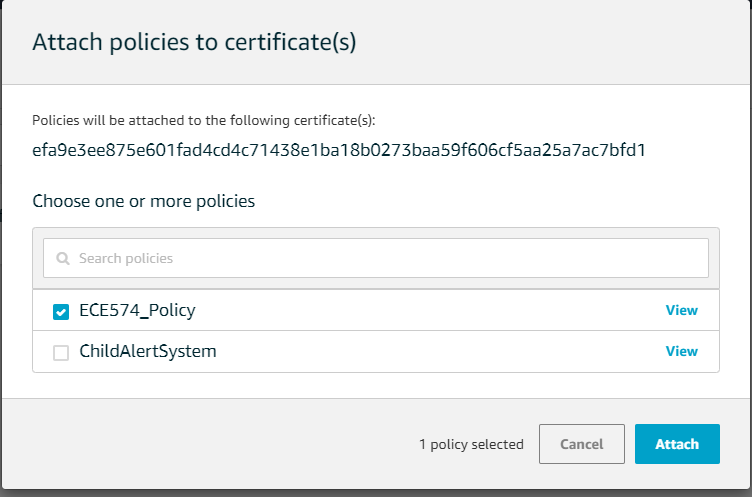
1. Attach an AWS IoT Thing and Policy to a Device Certificate.
   1. In the left navigation pane, choose Secure, and then choose Certificates.



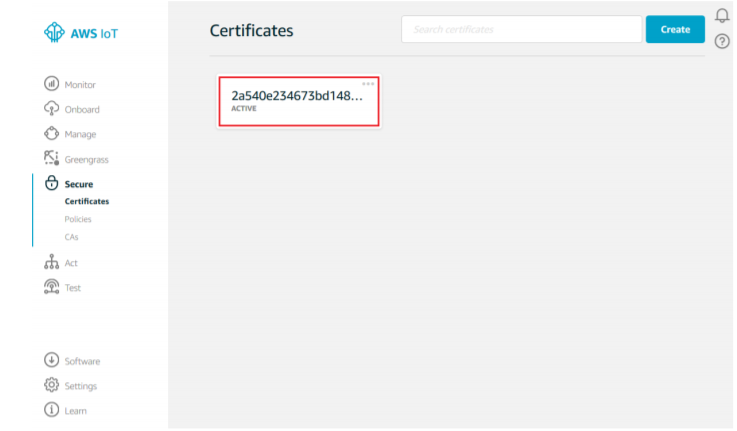
* 1. In the box for the certificate you created, choose ... to open a drop-down menu, and then choose Attach policy.



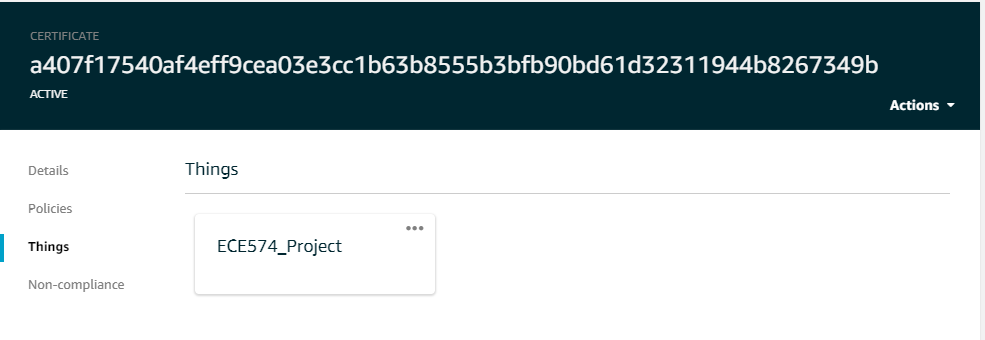
* 1. In Attach policies to certificate(s), select the check box next to the policy you created in the previous step, and then choose Attach.



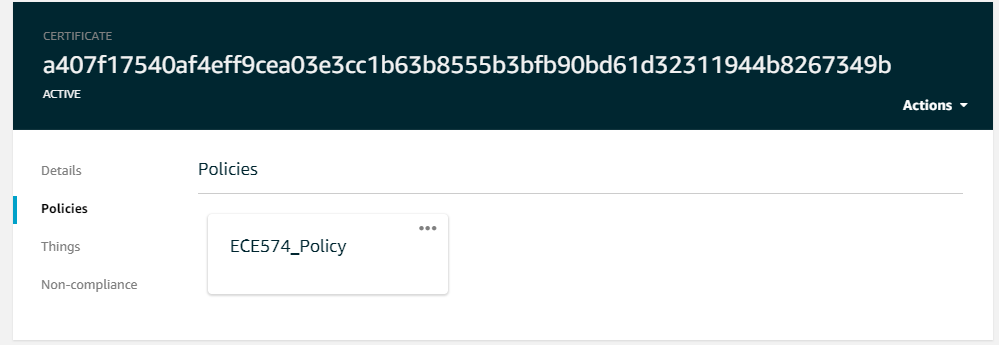
* 1. Repeat the above two steps and attach the Thing instead of Policy.
  2. To verify the thing is attached, select the box for the certificate.



* 1. On the Details page for the certificate, in the left navigation pane, choose Things.



* 1. To verify the policy is attached, on the Details page for the certificate, in the left navigation pane, choose Policies.

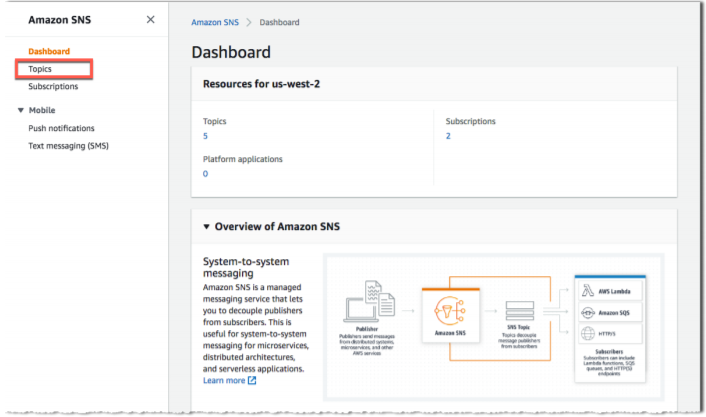


* 1. Save the End point of the Thing you created for communication purpose.
     + Click on the Thing box you created in the previous step.
     + In the left pane, click on Interact.
     + Save the HTTPS API Endpoint for further Communication.

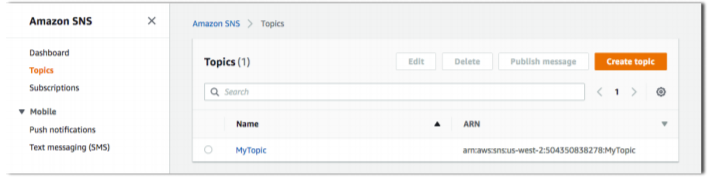
AWS SNS

1. Create an SNS Topic

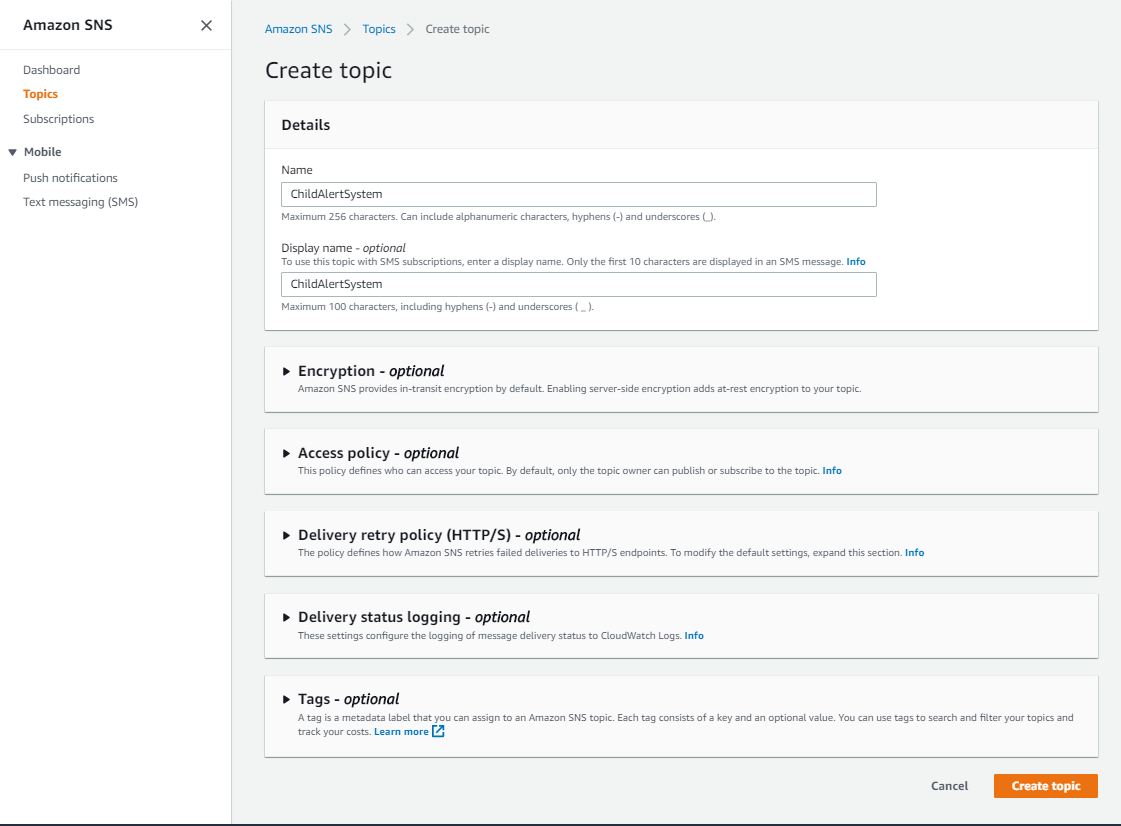
* Use the Amazon SNS console to create an Amazon SNS topic. (Note Amazon SNS is not available in all AWS Regions.)
  + Open the Amazon SNS console.
  + On the left pane, choose Topics.



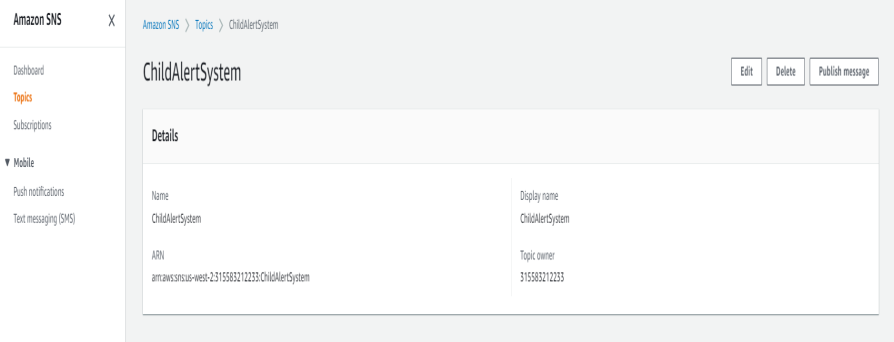
* Choose Create topic.



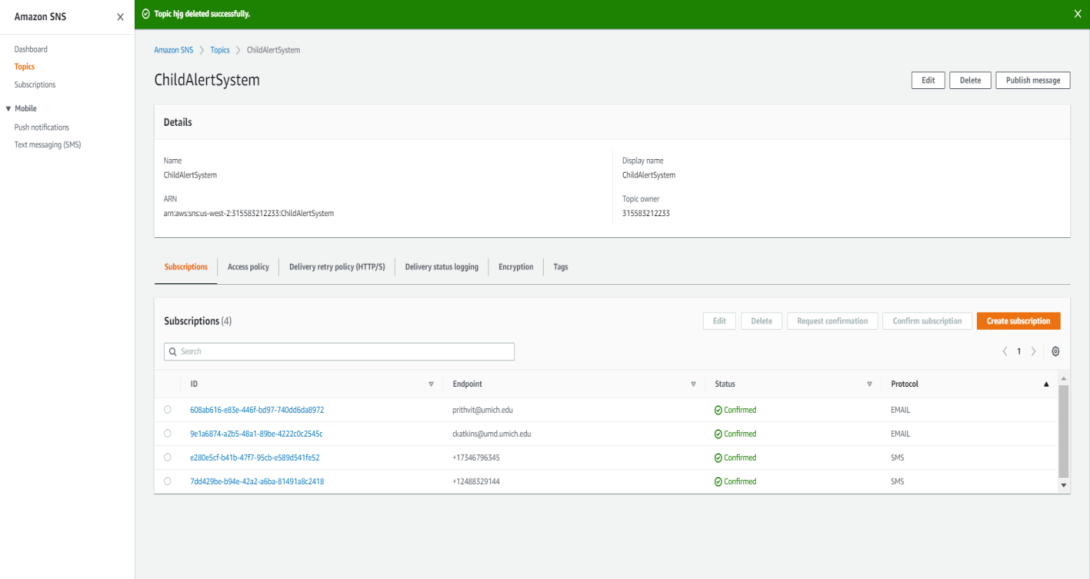
* Enter a topic name and a display name, and then choose Create topic.



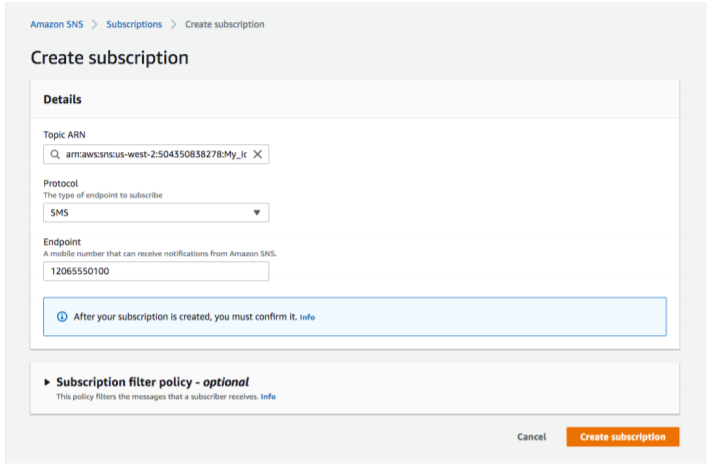
* Make a note of the ARN for the topic you just created.



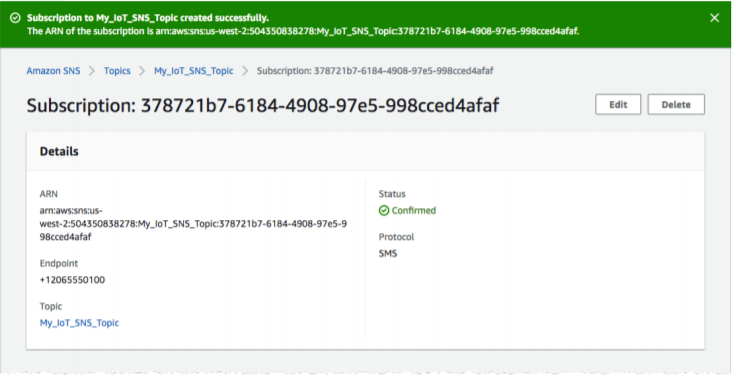
1. Subscribe to an Amazon SNS Topic
   * In the Amazon SNS console, select the check box next to the topic you just created. Under the Topic, choose Subscribe to topic.



* + On Create subscription, from the Protocol drop-down list, choose SMS



* + The Amazon SNS console displays the following message, but you might not receive a confirmation message.

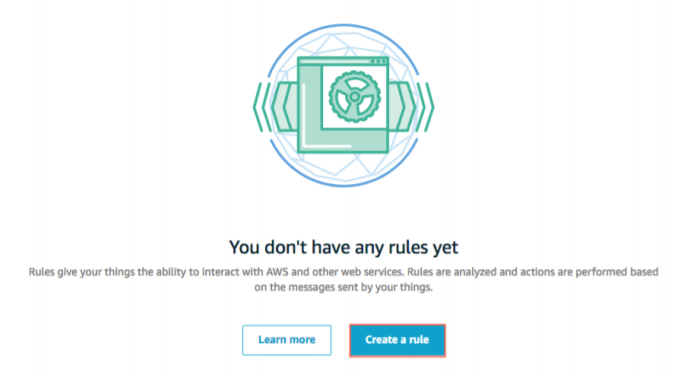


Rule

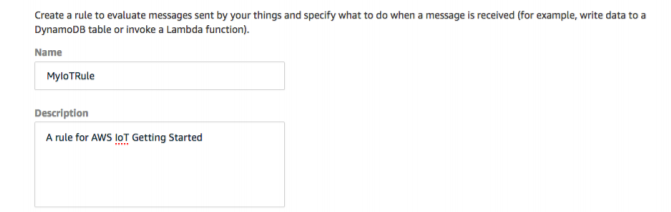
1. Create a Rule
   * In the AWS IoT console, in the left navigation pane, choose Act.



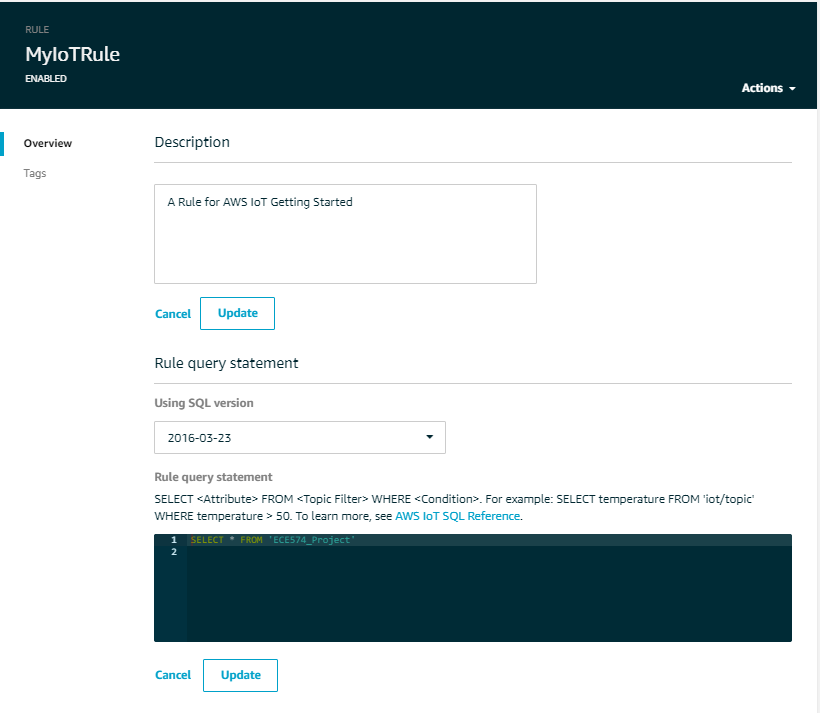
* + On the Act page, choose Create a rule.



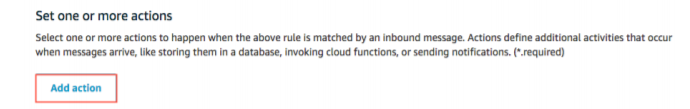
* + On the Create a rule page, in the Name field, enter a name for your rule. In the Description field, enter a description for the rule.



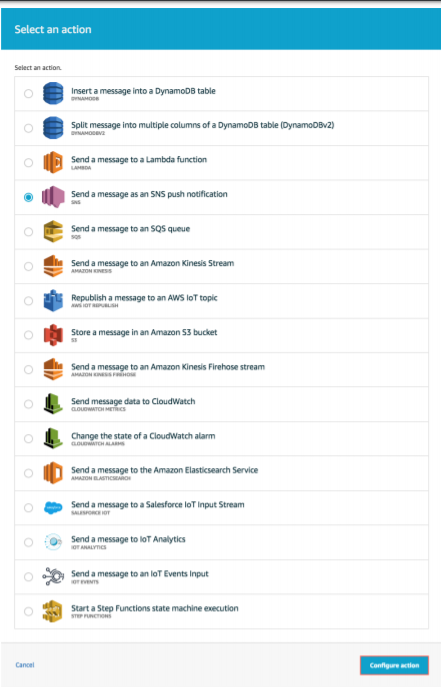
* + Scroll down to Rule query statement. Choose the latest version from the Using SQL version dropdown list. In the Rule query statement field, enter SELECT \* FROM 'my/topic'.



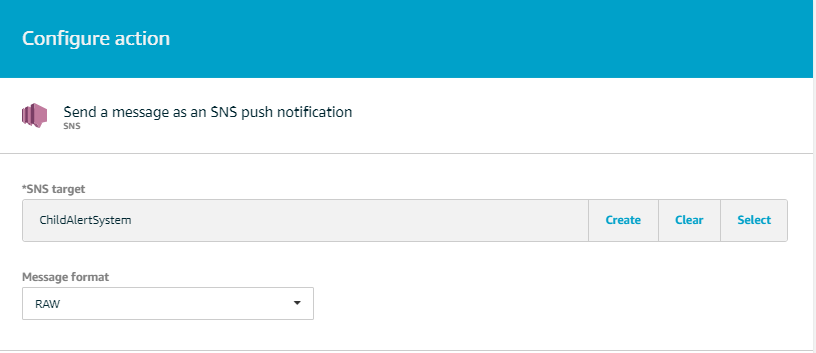
* + In Set one or more actions, choose Add action.



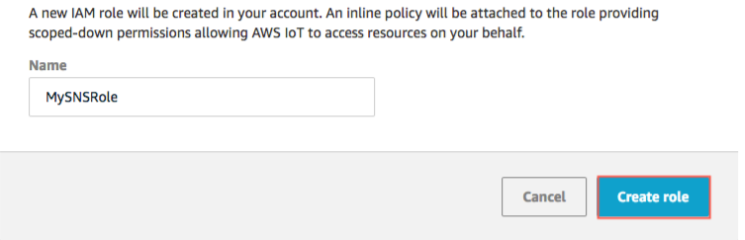
* + On the Select an action page, choose Send a message as an SNS push notification, and then choose Configure action.



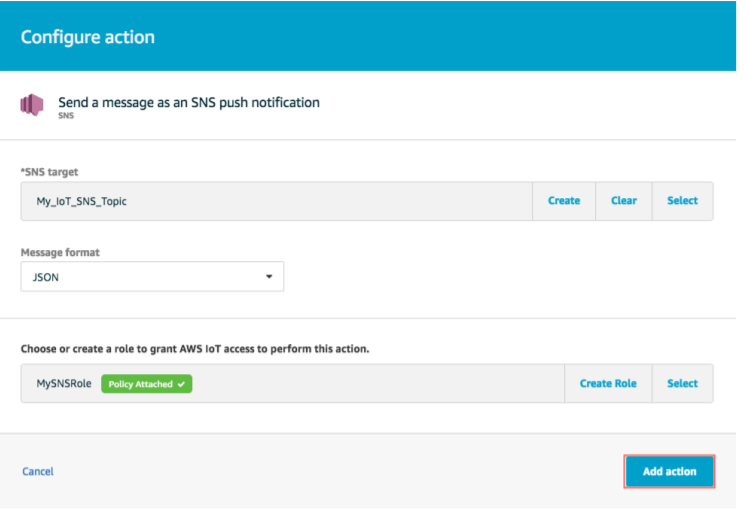
* + On the Configure action page, under SNS target, choose Select to expand the SNS topic. Then choose Select next to the Amazon SNS topic you created earlier. Under Message format, choose RAW.



* + Choose Create a new role. In IAM role name, enter a name for your new role, and then choose Create a new role.



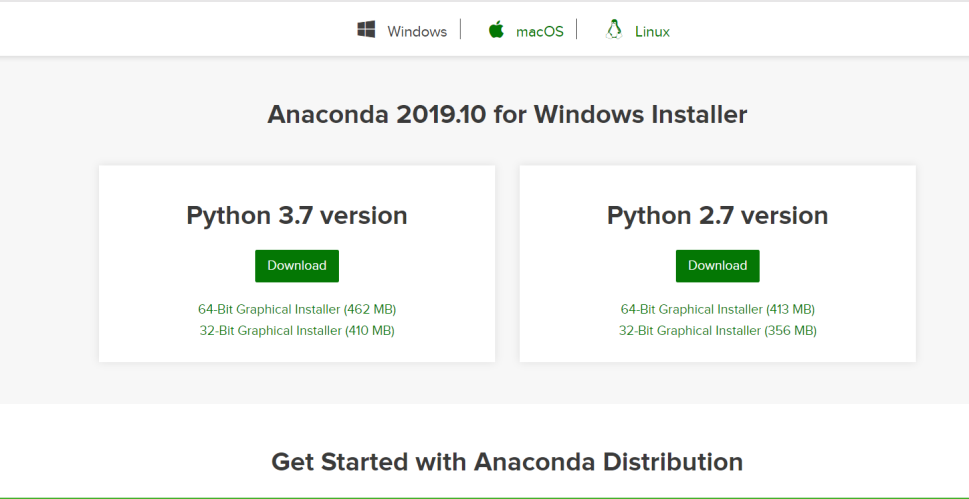
* + Under IAM role name, choose Update role to apply the permissions to the newly created role. Choose the role, and then choose Add action.



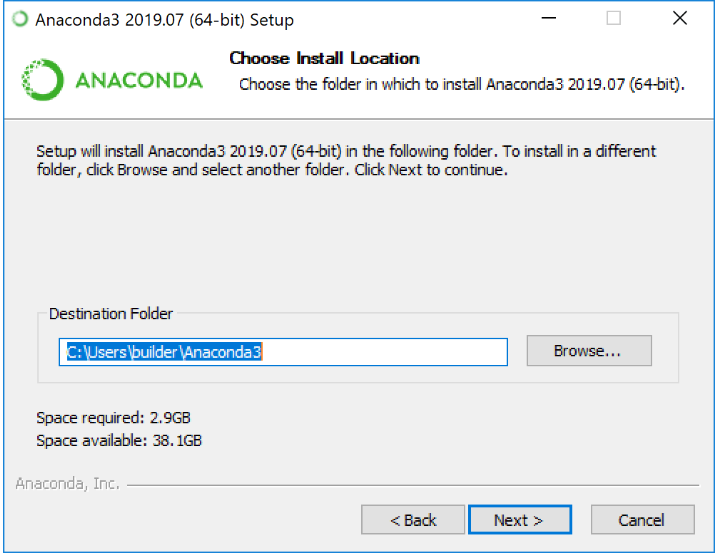
* + On the Create a Rule page, choose Create rule.

**Central Module**

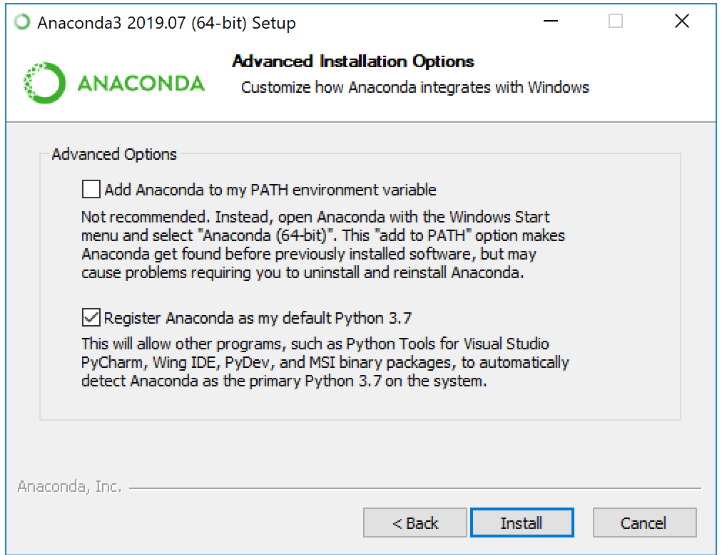
1. Installation of Anaconda Python
   * Open [Anaconda Distributions](https://www.anaconda.com/distribution/) Website.
   * Choose Windows Python 3.7 version 64-bit Graphical Installer and Download the Installer.



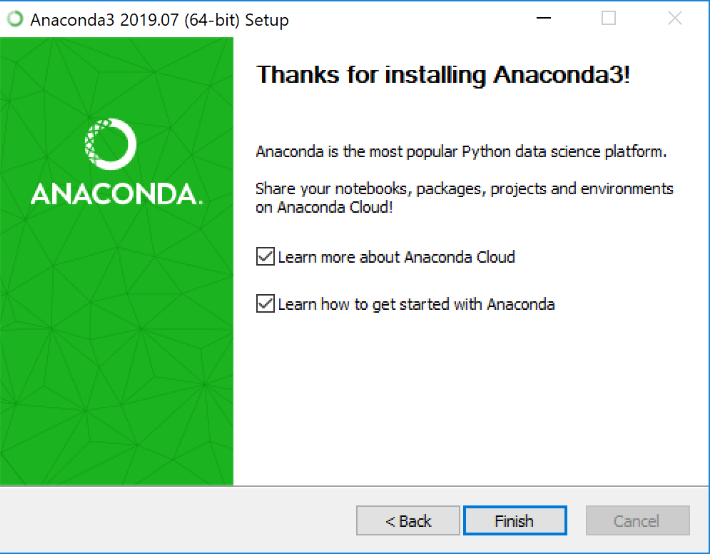
* + Open the Location where you Downloaded, right click on Installer and click on Run as Administration.
  + Click on Next.
  + Read the licensing terms and click “I Agree”.
  + Select an install for “Just Me”.
  + Select a destination folder to install Anaconda and click the Next button.



* + Choose Register Anaconda as my Default Python 3.7. Click on Install.



* + Click on Next.
  + After a successful installation, Click on Finish.



* + In Computer, go to Search, type Spyder and open the Spyder.
  + In Spyder IDE, Click on File and open a New File.
  + Save the File with .py extension.
  + Write the Python Algorithm (Code).
  + Click on Green Run Button.
  + If the Details given are Correct, it makes a connection with AWS IoT Thing ECE574\_Project. If not Check the Details and retry the connection.

