EE579 Group 4 Documentation

This site provides documentation to allow integration with our API and MQTT Broker.

If you would like to learn the web interface and functionality provided by the system you can start by Creating an Account. Then you can get started adding Devices and Rules

If you would like information on how to develop your own IoT device and integrate with our system, start by reading the Register a Device page.

Tip

This PDF has been generated from the documentation site here - docs.ee579-group4.net. We would strongly recommend that the documentation is viewed on the interactive site, rather than this PDF, for the best experience. You may check the date of the last deployment here to verify it has not been updated after the submission deadline

System Overview

Admin Interface

Our admin interface is available here: https://www.ee579-group4.net. This site enables users to login and configure rules, devices, and device groups.

Our system is multi-tenant, with users able to create multiple tenants and invite other users. Rules and devices are then scoped to these tenants, meaning users only have permission to access and edit resources belonging one of their tenants.

API

Our API is written in C# using ASP.NET Core. Full Swagger OpenAPI documentation is available here: https://ee579-dev-api.azurewebsites.net

IoT Hub

We are using Azure IoT Hub as an MQTT Broker. This uses a publish/subscribe model to allow 2 way communication between our server and IoT devices. Information on connecting and receiving/publishing messages can be found here

Test User

An test user with an example tenant, devices, and rules is available to have a play around with. Go to www.ee579-group4.net and login using the following credentials:

- **Email**: `test@test.com`
- **Password**: `Password-0`

GitHub Repos

- ee579-api - C# API repository
- ee579-web - React web interface repository
- ee579-msp430 - Repository containing msp430 and ESP8266 code
- ee579-docs - Repository containing the markdown files used to create these docs

Getting Started

Registering a device

In order for your device to connect to IoT Hub, it must first retreive credentials. This can be done by sending a `POST` request to `https://ee579-dev-api.azurewebsites.net/devices/register`. This will register your device with our system and return credentials to allow connection over MQTT. The device should make this request when first powered on, but the same endpoint can be used even if a device is already registered in the system. This accounts for situations where the device may have lost it's credentials, and allows the device to make the request every time it powers on, if desired.

Request Format

Example Body

```
{
    "deviceId": "<string>"
}
```

Parameters

- **deviceId**: This should be a unique identifier. A MAC adress is recommended, but it can take any form - guid, uuid, etc.

Response Format

Example Body

```
{
    "connectionString": "HostName=IFTTT-Iot-Hub.azure-
devices.net;DeviceId={deviceId};SharedAccessKey=El50EkQ/S/9qp5dd/
V3VpsizIvSv+SA8TVF3QiGc93A=",
    "host": "IFTTT-Iot-Hub.azure-devices.net",
    "port": 8883,
    "topic": "devices/{deviceId}/messages/devicebound/#",
    "password": "SharedAccessSignature sr=IFTTT-Iot-Hub.azure-
devices.net%2Fdevices%2F{deviceId}
&sig=KOYS9LgCJ9eH7TTlMIGvedxIVI3cXmha7uU6yB4Bs6M%3D&se=88018657115"
}
```

Parameters

- **connectionString**: The connection string the broker. {deviceId} should be replace with a unique identifier. A MAC adress is recommended, but it can take any form - guid, uuid, etc.
- **host**: The MQTT broker url.

- **topic**: The topic that the device should subscribe to, to receive cloud-to-device messages.
- **password**: The password that should be used when connecting to the broker, in the form of a SAS token.

Debugging

It may be neccessary to make this request on a computer, rather than IoT device when debugging. The API Swagger Docs can be used to make this request and provides a prefilled request body. Any other method of making HTTP requests can also be used - cURL, Postman, etc.

MQTT Broker

Connecting

To connect to IoTHub there are multiple different protocols that can be used, you can either directly connect to these protocols or prefferably use one of the IoTHub device SDKs.

Connecting using the IotHub SDKs

Connecting to IoTHub using one of Microsoft's SDKs use a connection string to connect. The connection string can be garnered from the response from our API's register request. Using the SDKs is the recommended way of connecting for its ease of use, especially regarding certificate handling as IoTHub forces TLS/SSL.

Some of the available SDKs include:

- C SDK
- Python SDK
- Node.js SDK
- Java SDK
- .NET SDK

Using our Example Code for an ESP8266

We have provided some example code for registering a device with the API and sending/ receiving messages from IoTHub. This example code is for the ESP8266 Wifi Module, but the methods will be similar on alternative devices. The code can be pulled from this repository. Cd into the esp_iothub_client directory for the example code. By following the instructions here, you can setup the Arduino IDE with functionality for flashing our example code to the ESP8266.

Connecting directly to the protocols

As MQTT is usually implemented on memory restricted devices, it is possible that the memory required to implement broker connection/interaction using an SDK is unavailable. For this reason, directly connecting to the protocol might be required. Below is a mosquitto subscribe command showing the parameters required to manually connect to the cloud device topic.

This command can also be used to simulate a device in order to test cloud-to-device messages. Similarly, `mosquitto_pub.txt` contains an example command that can be used to test device-to-cloud messages.

```
mosquitto_sub \
    -h IFTTT-Iot-Hub.azure-devices.net \
    -p 8883 \
    -t "devices/00:0a:95:9d:68:16/messages/devicebound/#" \
    -i 00:0a:95:9d:68:16 \
    -u "IFTTT-Iot-Hub.azure-devices.net/00:0a:95:9d:68:16" \
    -P "SharedAccessSignature sr=IFTTT-Iot-Hub.azure-
devices.net%2Fdevices%2F00%3A0a%3A95%3A9d%3A68%3A16&sig=ulc08e%2FYp%2FMLJdyMLxsV
\
    --cafile C:\Users\Fraser\GitHub\ee579-
api\EE579\EE579.Core\Slices\Devices\Simulate\iotHubCert.pem
    -V mqttv311 \
    -q 1
```

- **-h**: The host name for the broker.
- **-p**: The port number of the broker.
- **-t**: The topic the cloud to device messages will be received on.
- **-i**: The unique identifier for the device. Recommended as the device MAC address.
- **-u**: The MQTT username which here is just set to the hostname/deviceId.
- **-P**: The password for the broker. Can be retrieved from the register API response.
- **--cafile**: Path to the IotHub Certificate. This is available here - `iotHubCert.pem`
- **-V**: The MQTT version. IotHub only supports MQTT version 3.1.1.
- **-q**: This is for the quality of service. IoTHub only supports a quality service of 1.

MQTT Messaging Schema

This page shows schema for messages between device and cloud. All message parameters should be sent in the message body, in JSON format as detailed below. It should be noted that the JSON is case-sensitive so all JSON key and values must be formatted as shown in the 'Message Body' schema.

Device to Cloud Messages

Device-to-cloud messages should be sent when an input has been triggered. These messages will be used to trigger rules if the conditions are met.

Button Pushed

This message is sent to the broker whenever a button is pushed on the microcontroller.

Message Body

```
{
    "InputType": enum, // [Button1, Button2]
    "Value": int // The length the button is held in ms
}
```

Potentiometer Input

This message is sent every time the analogue value from the potentiometer is read. The value is sent with this message.

Message Body:

```
{
    "InputType": "Potentiometer",
    "Value": int // (0 - 1023) - the analogue value of the potentiometer
}
```

Temperature Input

This message is sent every time the analogue value from the temperature sensor is read. The value is sent with this message.

Message Body:

```
{
    "InputType": "Temperature",
    "Value": int // (-50 to 100) - the temperature in degrees celsius
}
```

Cloud to Device Messages

Cloud-to-device messages are used to perform outputs on the device as a result of a rule being triggered. There is also a message used to configure the device's inputs to reduce the number of messages sent.

LED Output

This message is sent to notify the microcontroller to turn an LED on/off and with the specified colour.

Message Body:

```
{
    "OutputType": "LedOutput",
    "Peripheral": enum, // [Led1, Led2, Led3]
    "Value": boolean, // the desired state of the LED.
    "Colour" enum // (Led3 only) [Red/Green/Blue/Purple/Yellow/White]
}
```

Breath LED

This message is sent to notify the microcontroller to breathe an LED at a specified speed and colour.

Message Body:

```
{
    "OutputType": "LedBreathe",
    "Peripheral": enum, // [Led1, Led2, Led3]
    "Period": int, // how long the period of the breathing should be in ms.
    "Colour": enum // (Led3 only) [Red/Green/Blue/Purple/Yellow/White]
}
```

Blink LED

This message is sent to notify the microcontroller to blink an LED at a specified speed and colour.

Message Body:

```
{
    "OutputType": "LedBlink",
    "Peripheral": enum, // [Led1, Led2, Led3]
    "Period": int, // how long the period of the blinking should be in ms.
    "Colour": enum // (Led3 only) [Red/Green/Blue/Purple/Yellow/White]
}
```

Fade LED

This message is sent to notify the microcontroller to fade an LED to a desired state at a specified speed and colour.

Message Body:

```
{
    "OutputType": "LedFade",
    "Peripheral": enum, // [Led1, Led2, Led3]
    "Value": boolean, // the desired state of the LED.
    "Duration": int, // the duration the fade should take in ms.
    "Colour": enum // (Led3 only): [Red/Green/Blue/Purple/Yellow/White]
}
```

Cycle LED 3

This message is sent to notify the microcontroller to cycle the LED colour in the specified direction.

Message Body:

```
{
    "OutputType": "LedCycle",
    "Direction": boolean, // cycle forwards or backwards - [true/false]
respectively
    "Period": int // how long the period of the cycle should be in ms
}
```

Buzzer On

This message is sent to notify the microcontroller to turn the piezo buzzer on for the specified length of time.

Message Body:

```
{
    "OutputType": "BuzzerOn",
    "Duration": int, // the duration the buzzer should be on in ms
}
```

Beep Buzzer

This message is sent to make the microcontroller toggle a buzzer at the desired rate.

Message Body:

```
{
    "OutputType": "BuzzerBeep",
    "OnDuration": int, // the duration the buzzer should be on in ms
    "OffDuration": int // the duration the buzzer should be off in ms
}
```

Configure Device

This message is sent to notify the microcontroller which input devices are involved in rules. This is used to limit the number of messages sent by devices - if an input device is not involved in any rules, messages should not be sent to the broker when its value changes.

This message is sent whenever a device connects to the broker, or if a rule is created, updated, or deleted involving the device.

Message Body:

```
{
    "MessageType": "DeviceConfig",
    "Button1": boolean,
    "Button2": boolean,
    "Potentiometer": boolean,
    "Temperature": boolean
}
```

Web Hooks

What is a Web Hook

Web hooks allow one application to integrate with another over HTTP. This is commonly done by the external application making a post request to an endpoint provided by the application in which the functionality is performed.

Using Web Hooks in our System

In the context of our system, web hooks can be used for both inputs to rules, and outputs from rules.

Inputs

Using a web hook as a rule input allows a 3rd party application to trigger the web hook rule input by making an HTTP request to our API. When a rule is created with a web hook input, a unique URL is generated. In order to trigger the rule, a post request must be made to this URL. Inherently, the logic that determines when the web hook URL should be invoked must be implemented by the 3rd party developer. For example, if a 3rd party wanted to trigger a rule using a moisture sensor, the 3rd party must implement their own low level code to handle the desired analogue thresholds for the sensor and then send the HTTP request to the web hook URL, which will trigger the rule.

An example URL is shown below:

`http://ee579-dev-api.azurewebsites.net/webhooks/trigger/{unique_code}`

This is a callback URL to our API where `{unique_code}` is the code used to identify the specific rule input.

Outputs

Using a web hook as a rule output allows our system to trigger events on a 3rd party application. This works by our API sending an HTTP post request to your specified URL upon the rule's inputs being triggered. The URL must be public facing and have an appropriate CORS policy to allow the receival of our request. That is, POST request should be allowed and the cross origin polciy should allow for requests from our API.

Why use Web Hooks?

Our web hook implementation allows our system to be controlled by, or control, a completely different system, assuming that the different system is capable of sending or receiving HTTP post requests. For a web hook as a rule input it means a rule from any connected device on our system, can be triggered by invoking the web hook URL (that we generate for you) by sending a post request to it. This gives almost limitless potential for input interoperability thereby making our system not subject to the hardware or input implementation limitations on certain microncontrollers such as the MSP430. That is, any complex inputs that we do not support can be still be used where their HIGH event on the 3rd party application, can then invoke the web hook URL, thus triggering a rule where our system need not know of the complex input that might have caused it, only that the web hook has been invoked. Likewise, with using a web hook as a rule output, it can trigger functionality on a different system where the triggered functionality is controled by the rule logic in our own system.

Examples and Use Cases

IFTTT

IFTTT (If This Then That) is a popular software platform that connects apps, devices and services from different developers in order to trigger one or more automations involving those apps, devices and services. IFTTT offers web hooks as both logic inputs and outputs. On our system, by specifying the web hook output URL as your IFTTT web hook input URL, you have potential to control hundreds of different services and devices all from our own system. An example of this could be using a device on our system with a rule setup with a button pressed input event, then with a web hook output from the rule that points to a web hook input on IFTTT, you could automatically order a Dominoes, send an SMS for you, turn your heating up, post a tweet and hundreds more options. Likewise, this system interaction can work in reverse, with devices on our system being influenced by your logic on IFTTT such as sounding a buzzer on oone of our devices and a specific time of day. The possibilities are endless.

Azure Logic Apps

The corporate equivalent to IFTTT, is Logic Apps. A Logic App allows for automation of business workflow tasks and can be controlled with input triggers and produce some output such as sending an email. In a similar fashion to how our system might interact with IFTTT, there is vast potential for controlling your business logic using devices on our system.

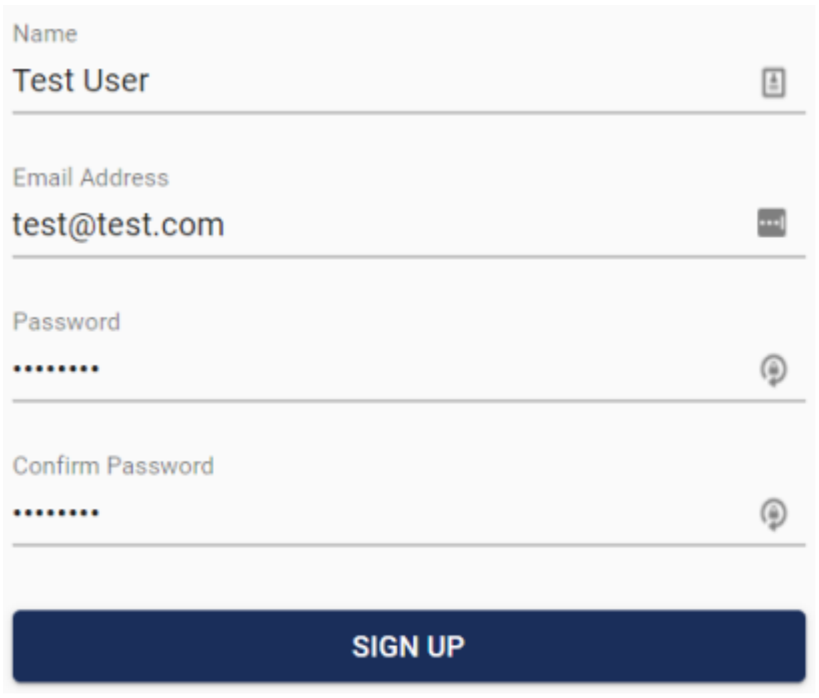
Create an Account

Visit https://www.ee579-group4.net/signup to get started.

There are several methods available for signing in:

Username and Password

To create an account with a username and password, fill out the form on the sign up page.



Name	Test User
Email Address	test@test.com
Password	••••••••
Confirm Password	••••••••

You will then be sent an email to verify that you have access to the email address. You must then click the link in the email before you can sign in.

External Providers

Logging in with an external provider allows you to reduce the number of passwords and accounts you have to manage. This site allows you to sign in with your Google or Microsoft account. Click the respective button to start the external sign in process. You will then be redirected to the external providers site where you can sign in.

After signing in to the external providers account, you will be redirected back to this site and signed in. Verifying your email is not required when using external sign in as signing into the account verifies that you have access to the email.

Multi-Tenancy

What is Multi-Tenancy?

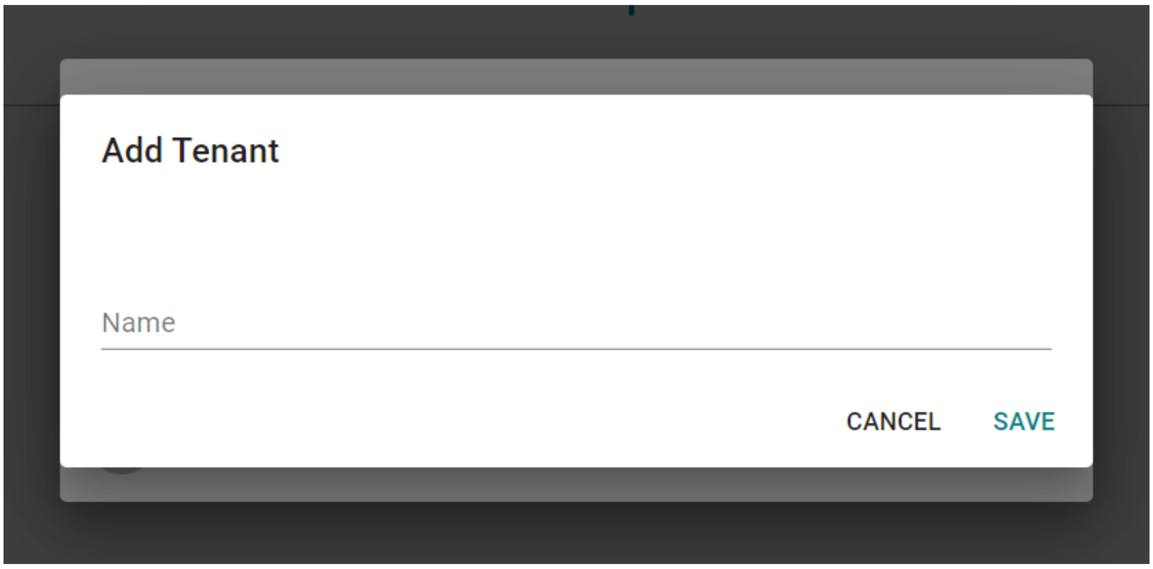
Multi-tenancy is an architecture that allows a single application to be used by multiple customers. This means that data in the application is scoped to the current customer and not visible to other customers of the application. An example of this is the university's use of Microsoft's suite of applications. In this case, the university has their own tenant. This tenant then contains all of the data specific to the university, like users, sharepoint data, email, etc and some services can be accessed by a tenant-specific URL - e.g., strath.sharepoint.com.

Multi-Tenancy in this application

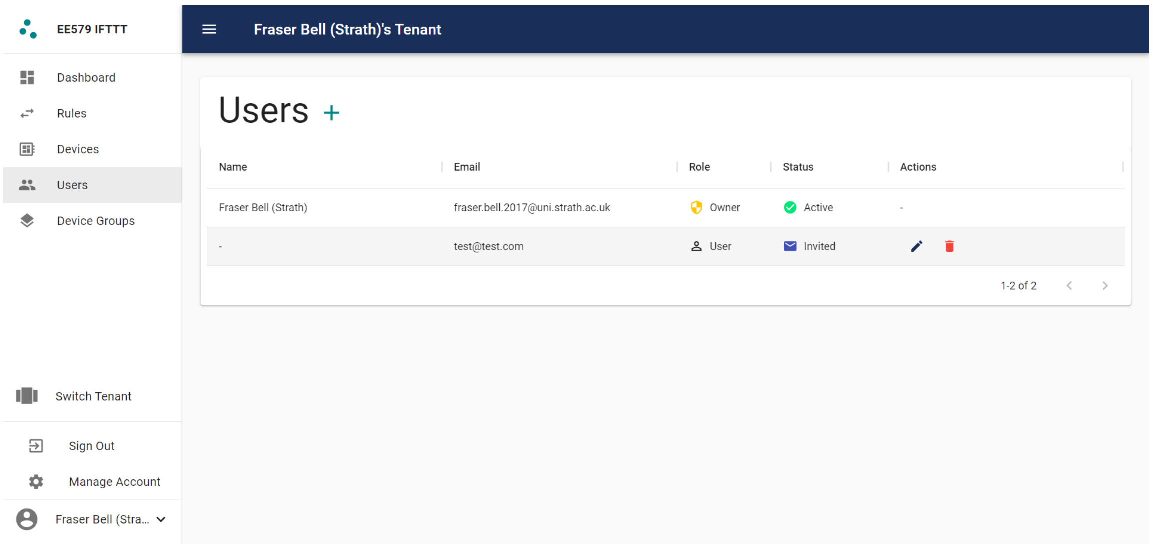
Multi-tenancy in this application involves scoping rules, devices, and device groups to tenants. This allows multiple customers to use the application without other customers seeing their data. Users can then belong to multiple tenants, allowing them to view and manage their data.

Creating a Tenant

When first creating an account a new tenant is created for you, allowing you to get started creating rules and adding devices. You may want to add another tenant in order to keep unrelated data separate - e.g., keeping rules and devices for each building in their own tenant. This can be done by clicking the 'Switch Tenant' button in the left drawer and then selecting 'Create a new tenant'.



Users



Name	Email	Role	Status	Actions
Fraser Bell (Strath)	fraser.bell.2017@uni.strath.ac.uk	Owner	Active	-
-	test@test.com	User	Invited	

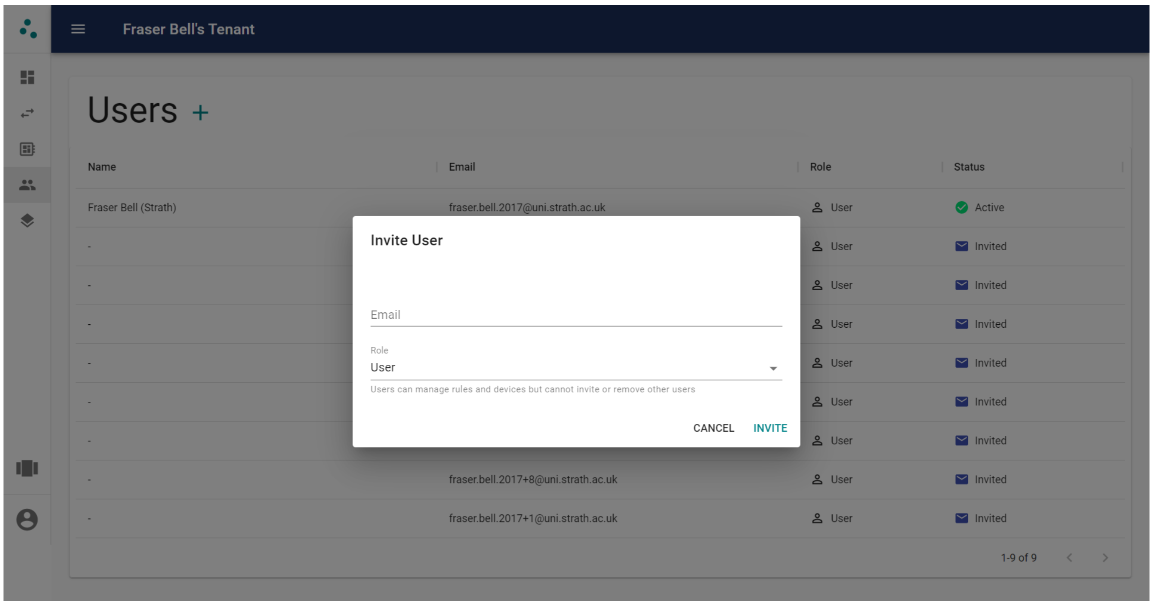
1-2 of 2

Roles

Users can have one of two roles within a tenant:

- **User** - Users with this role can manage devices, rules, and device groups. However, they cannot manage users belonging to the tenant or edit the tenant itself.
- **Owner** - This user has full control over the tenant. They can manage everything a user can, along with inviting and removing users and editing the tenant.

Inviting Users


Fraser Bell's Tenant

Users +

Name	Email	Role	Status
Fraser Bell (Strath)	fraser.bell.2017@uni.strath.ac.uk	User	Active
-		User	Invited
-		User	Invited
-		User	Invited
-		User	Invited
-		User	Invited
-		User	Invited
-	fraser.bell.2017+8@uni.strath.ac.uk	User	Invited
-	fraser.bell.2017+1@uni.strath.ac.uk	User	Invited

1-9 of 9

Invite User

Email

Role: User

Users can manage rules and devices but cannot invite or remove other users

CANCEL INVITE

Users can be invited to tenants by their email address. If the user already has an account, they will have immediate access to the tenant and receive an email telling them that they have been invited to your tenant. If they have not yet created an account they will receive an email informing them they have been invited to your tenant, including a link to sign up.

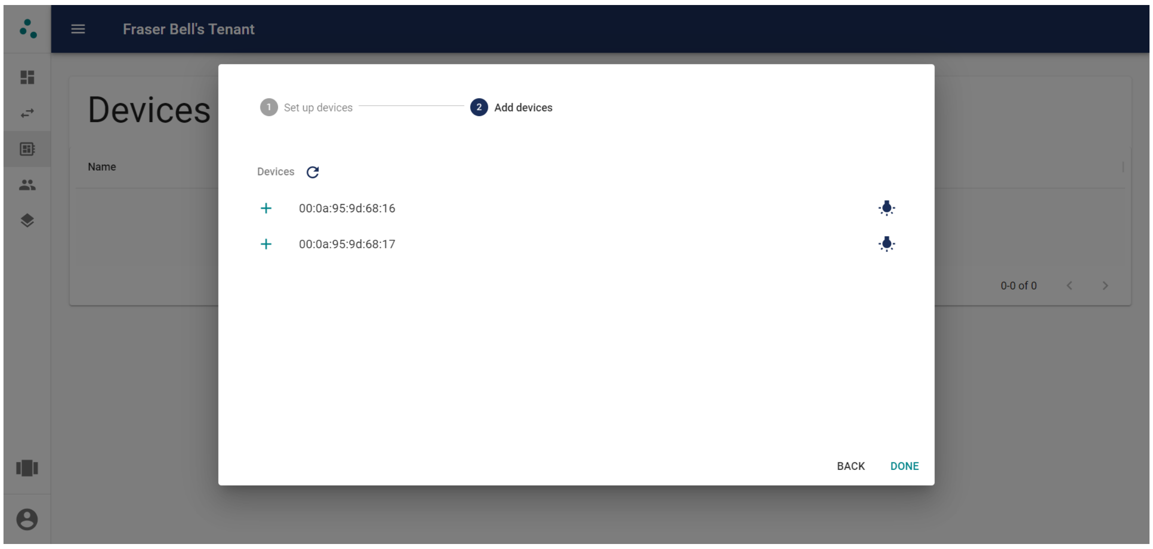
Devices

Devices are the devices that will be used in rules. They send inputs to the API and receive messages to perform outputs.

Adding Devices

Before devices can be used to create rules, they must be claimed by a tenant. The first step is to power the device on. When powered on for the first time, the device will register itself with the system, making it possible to claim it in the web interface.

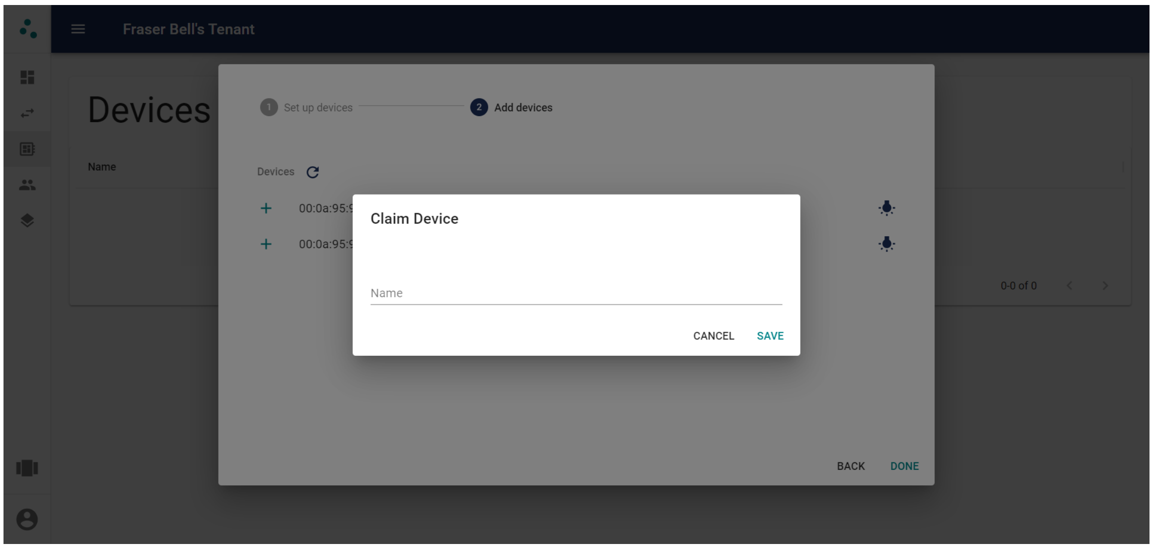
After the device has been powered on, navigatet to the devices page and click the add button above the table. This shows the modal below.



00:0a:95:9d:68:16
00:0a:95:9d:68:17

This shows the devices available for you to claim. You must be on the same network as a device in order to claim it. This prevents other users from claiming devices that do not belong to them. The devices are listed by their MAC address which you may use to identify specific devices when multiple are available. If you do not know the MAC address of your device you can press the lightbulb button. This will flash an LED on the device for 5 seconds, allwoing you to identify which device the list item corresponds to.

To claim a device press the add button next to its MAC address. This will bring up a form allowing you to give the device a user friendly name before claiming it.


Fraser Bell's Tenant

Devices

Name

1 Set up devices — 2 Add devices

Devices

00:0a:95:9

00:0a:95:9

Claim Device

Name

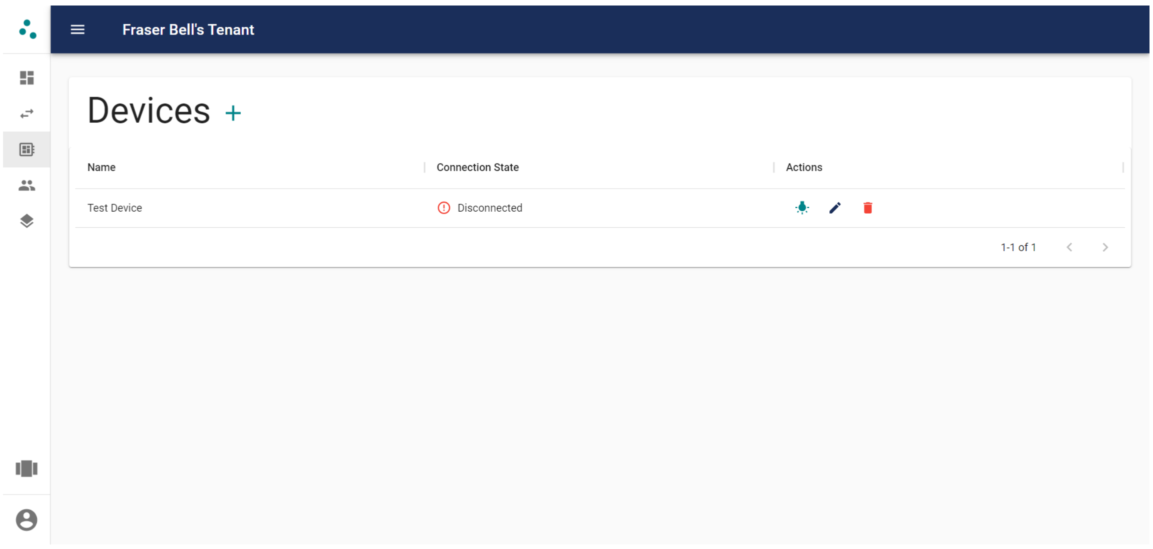
CANCEL SAVE

0-0 of 0

BACK DONE

Managing Existing Devices

After devices have been added, they will be visible on the devices page as shown below.


Fraser Bell's Tenant

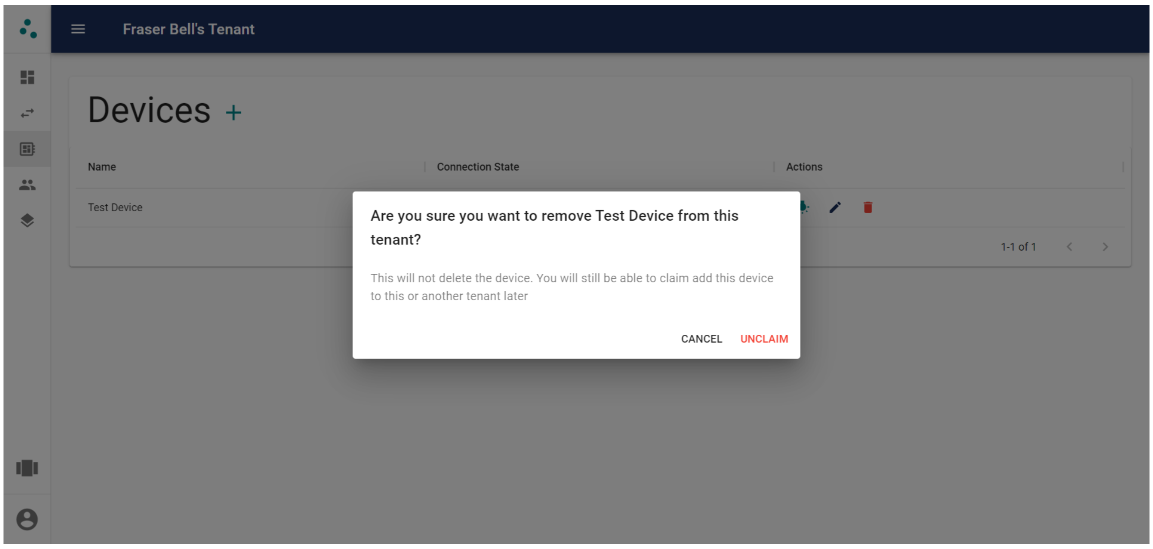
Devices +

Name	Connection State	Actions
Test Device	Disconnected	

1-1 of 1

This page allows you to view all of your devices and their current state. The 'Connection State' column shows whether the device is currently connected to the MQTT broker and able to send and receive inputs and outputs. It also allows you to preform some actions on the device. Here you may identify a device in the same way as when claiming it, by pressing the LED button. You can also change the device's name and unclaim it from the current tenant.

Unclaiming a Device


Fraser Bell's Tenant

Devices +

Name	Connection State	Actions
Test Device		

1-1 of 1

Are you sure you want to remove Test Device from this tenant?

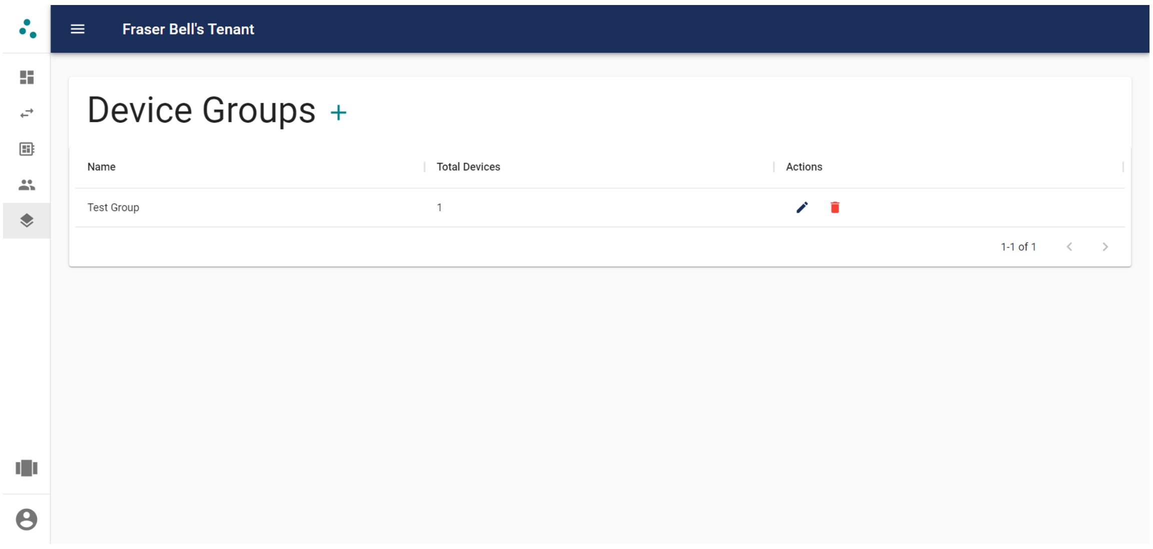
This will not delete the device. You will still be able to claim add this device to this or another tenant later

CANCEL UNCLAIM

You may wish to unclaim a device from a tenant. This removes the device from its current tenant, and removes it from any rules it may have been involved in. This does not delete the device completely, only resets it, allowing it to be claimed by another tenant. If you want to move a device from one tenant to another this would be the way of doing that.

Device Groups

Device groups provide a method of grouping devices with similar functions to improve the process of adding and managing rules.

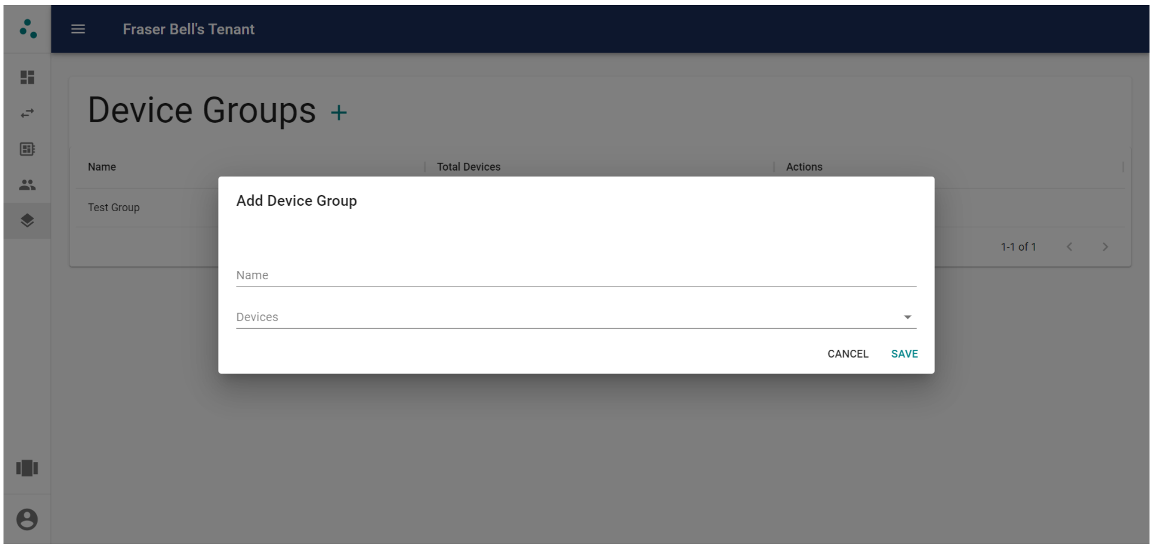

Fraser Bell's Tenant

Device Groups +

Name	Total Devices	Actions
Test Group	1	

1-1 of 1

Device groups can be added by specifying a name, and list of devices that should be long to the group. The name and devices can also be edited later.



Fraser Bell's Tenant

Device Groups +

Name	Total Devices	Actions
Test Group		

1-1 of 1

Add Device Group

Name

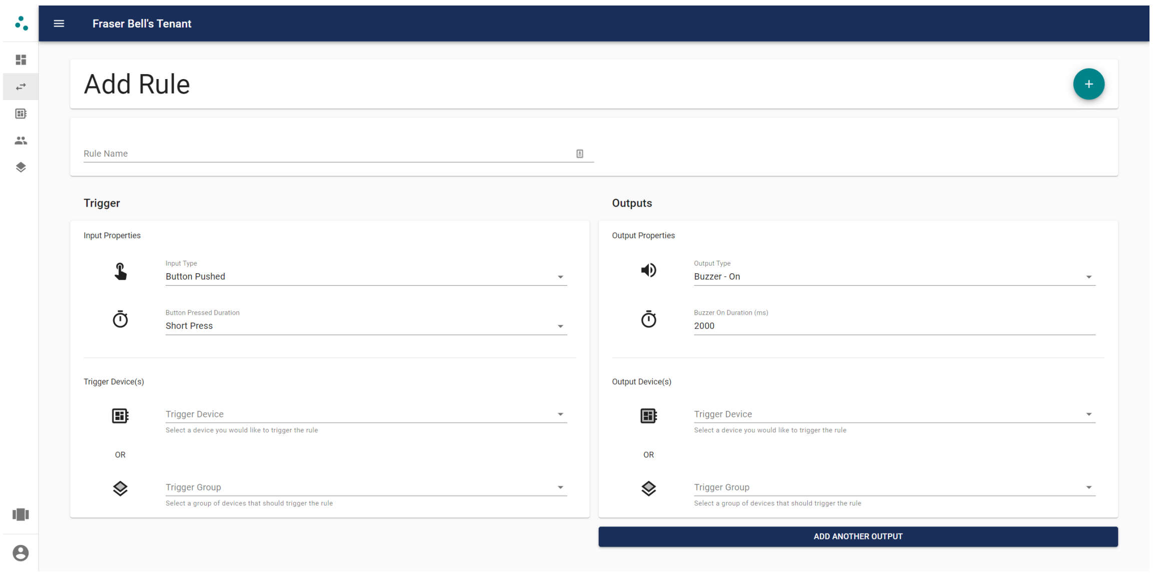
Devices

Rules

Rules are the core functionality of the system, allowing inputs from devices to trigger outputs.

Creating Rules

Navigate to the rules page and press the add button to create a new rule. The rule form is shown below:



Fraser Bell's Tenant

Add Rule

Rule Name

Trigger

Input Properties

Input Type: Button Pushed

Button Pressed Duration: Short Press

Trigger Device(s)

Trigger Device
Select a device you would like to trigger the rule

OR

Trigger Group
Select a group of devices that should trigger the rule

Outputs

Output Properties

Output Type: Buzzer - On

Buzzer On Duration (ms): 2000

Output Device(s)

Trigger Device
Select a device you would like to trigger the rule

OR

Trigger Group
Select a group of devices that should trigger the rule

The form is composed of 3 sections: Name, Trigger, and Outputs.

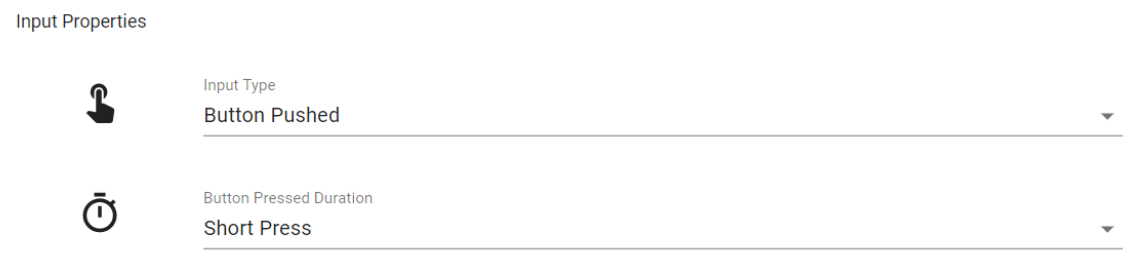
Trigger

The Rule trigger is the device input that will trigger the rule and cause the outputs to be performed. This is composed of the input type and parameters desired, and the device or device group the input will come from. If a device group is chosen, an input matching the desired type and parameters from any of the groups devices will trigger the rule.

There are several input types that can be chosen:

Input Types

Button Pushed


Input Properties

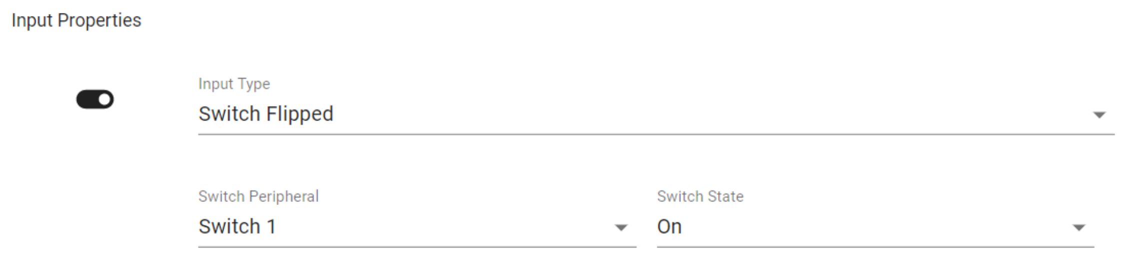
Input Type
Button Pushed

Button Pressed Duration
Short Press

This input is sent when the button is pushed on a device. The duration desired to trigger the rule can be set to one of three values:

- **Short Press** - 0s < 2s
- **Medium Press** - 2s < 10s
- **Long Press** - > 10s

Switch Flipped


Input Properties

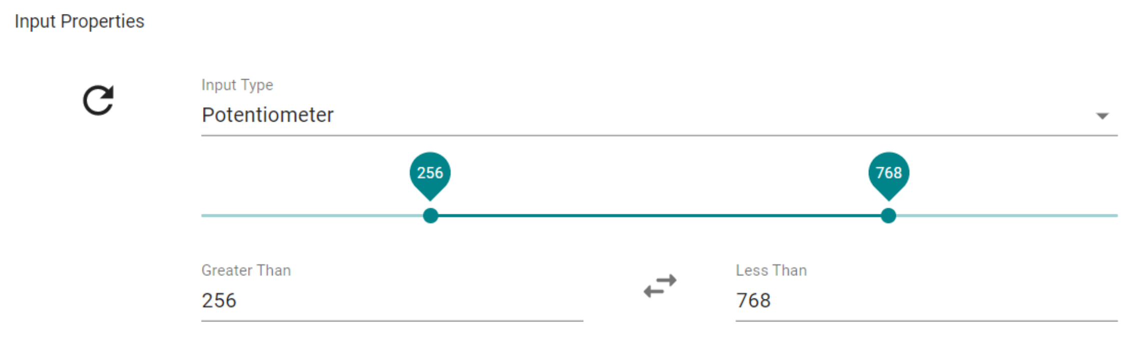
Input Type
Switch Flipped

Switch Peripheral
Switch 1

Switch State
On

This input allows rules to be triggered when one of the 2 switches on the device are flipped. The peripheral (switch 1 or 2) can be chosen and the switch state. The switch state is used to specify whether the switch should be triggered when the switch is flipped to an on or off state.

Potentiometer

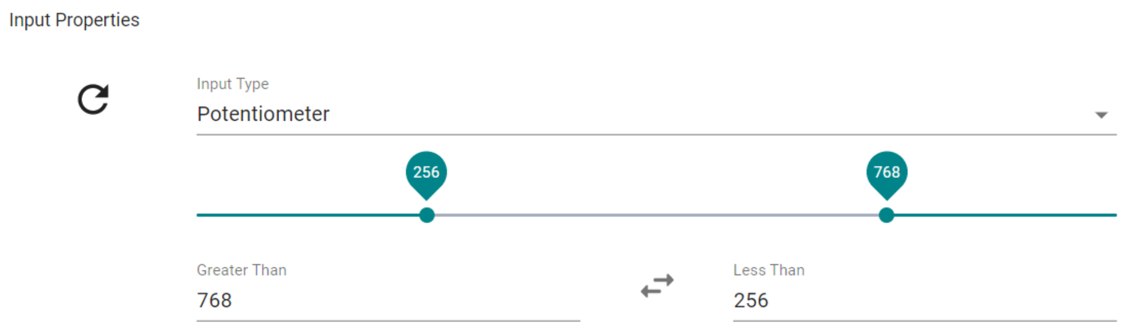

Input Properties

Input Type
Potentiometer

Greater Than
256

Less Than
768

This input allows you to specify a range of potentiometer values that will trigger the rule. The default configuration is a range between two values but this can also be inverted to allow for specifying a range of values less than the lower value *or* greater than the higher value. This can be done by pressing the button between the values, or manually setting the greater than value to be higher than the less than.


Input Properties

Input Type: Potentiometer

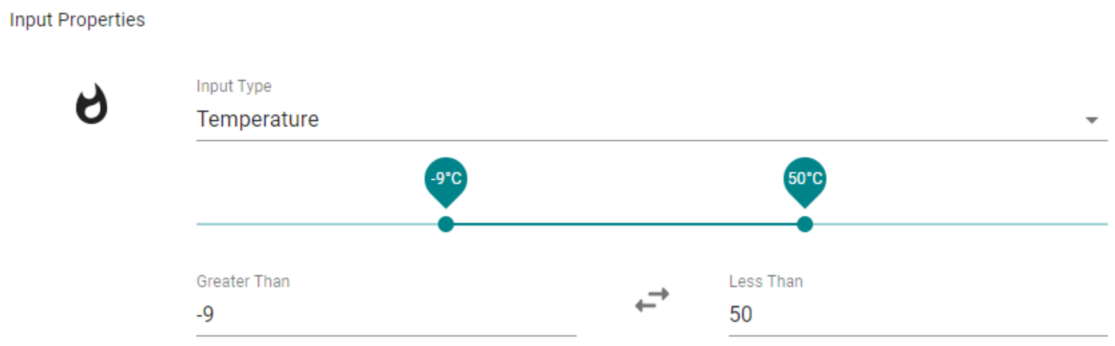
256 — 768

Greater Than: 768

Less Than: 256

When using a potentiometer value as an input it should be noted that the rule will not be triggered every time the potentiometer value changes if it is within the trigger range. Instead, the system keeps track of the last potentiometer value sent by the device and only triggers the rule if the last value was outwith the range and the new value is within it.

Temperature


Input Properties

Input Type: Temperature

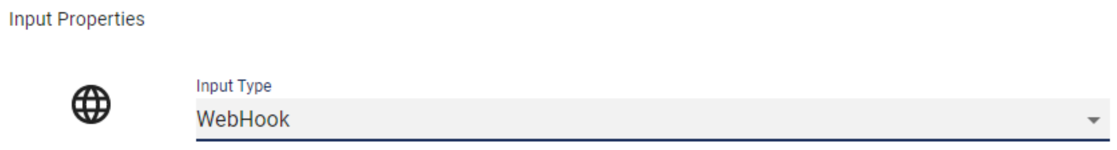
-9°C — 50°C

Greater Than: -9

Less Than: 50

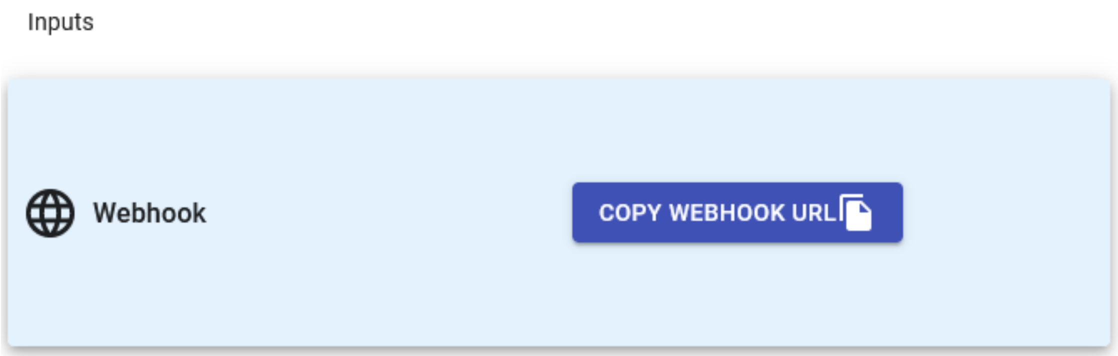
This input allows you to use the temperature sensor on your device as a trigger for the rule. It functions in a similar way to the potentiometer trigger, with a range picker. Again this range can be inverted, and the rule is only triggered when the last value did not satisfy the conditions.

Web Hooks


Input Properties

Input Type: WebHook

This is an input that can be triggered via the invocation of a generated URL, commonly known as a web hook. It works similar to a physical input such as a button press, but instead of being triggered upon the button being pressed, the trigger happens upon invocation of the URL. After adding a rule with a web hook as an input, from the rules page on the website, find the added rule from which the invoke URL can be garnered. Below is an example of what this looks like. For more information for developers and using web hooks for interoperability please see here.


Inputs

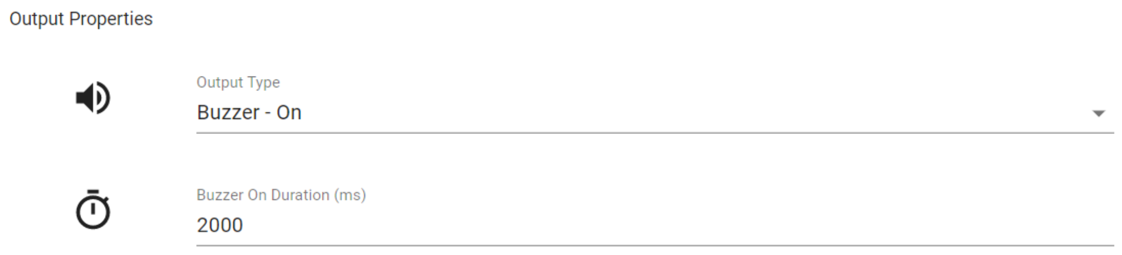
Webhook

Outputs

These are the outputs that will be performed when a rule is triggered. Multiple outputs can be performed for each rule.

OUTPUT TYPES

Buzzer - On

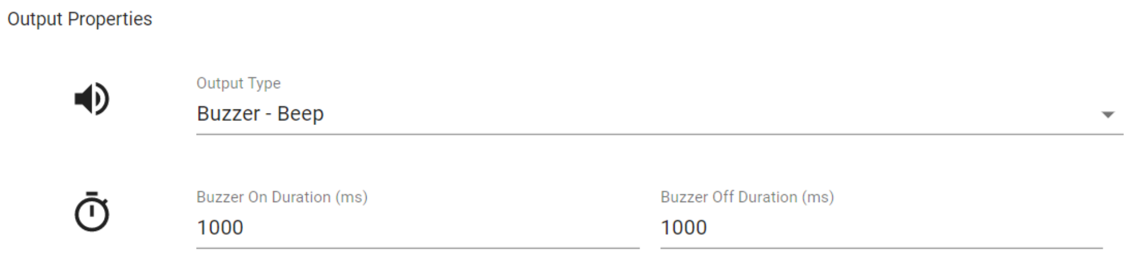

Output Properties

Output Type: Buzzer - On

Buzzer On Duration (ms): 2000

This output is used to sound a continuous buzzer tone for the specified duration.

Buzzer - Beep


Output Properties

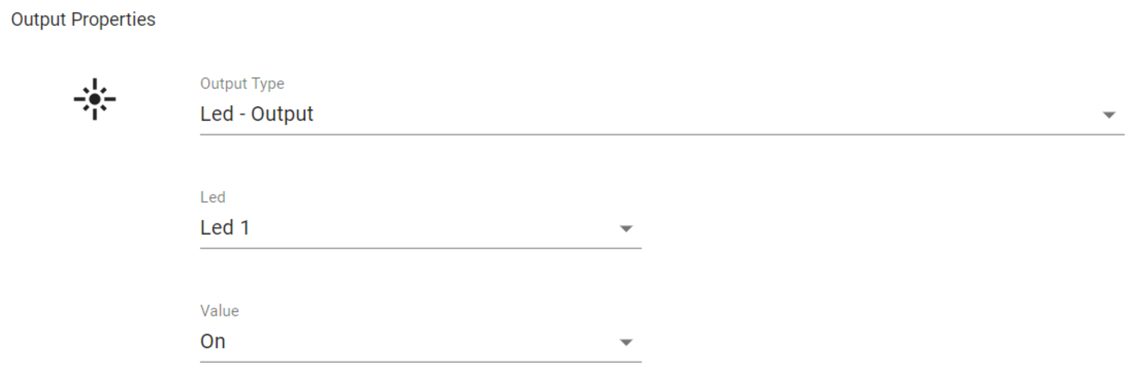
Output Type: Buzzer - Beep

Buzzer On Duration (ms): 1000

Buzzer Off Duration (ms): 1000

This output beeps the buzzer with the duty cycle specified. The minimum values for the on and off duration are 200ms.

LED - Output



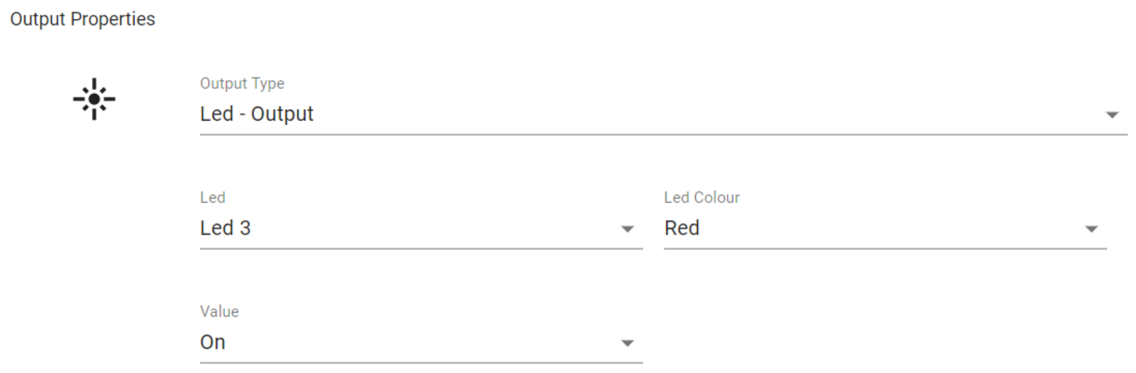
Output Properties

Output Type: Led - Output

Led: Led 1

Value: On

This output can be used to turn an LED on or off. There are 3 LEDs that can be selected, corresponding to the LEDs on the MSP430G2553. As such, LEDs 1 and 2 are single colour, but a colour - *Red, Green, Blue, Purple, Yellow, White* - can be chosen when LED 3 is selected.



Output Properties

Output Type: Led - Output

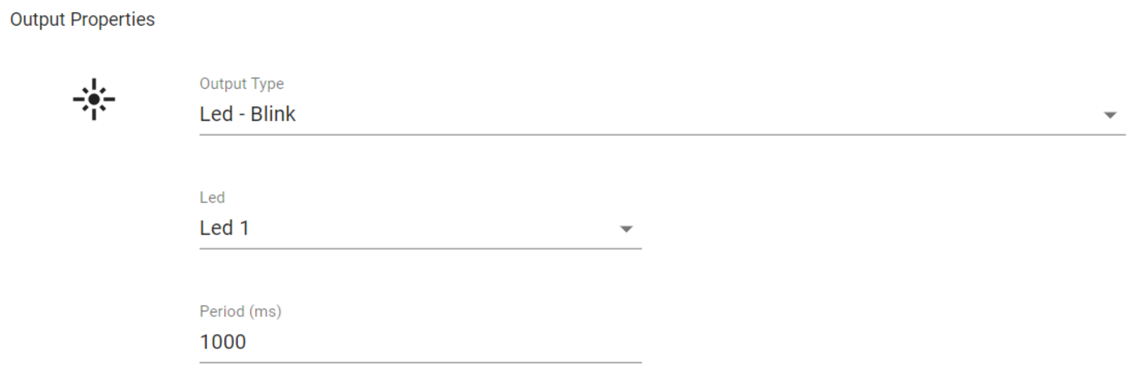
Led: Led 3

Led Colour: Red

Value: On

This choice of colour also applies to the other output types involving LED 3.

LED - Blink



Output Properties

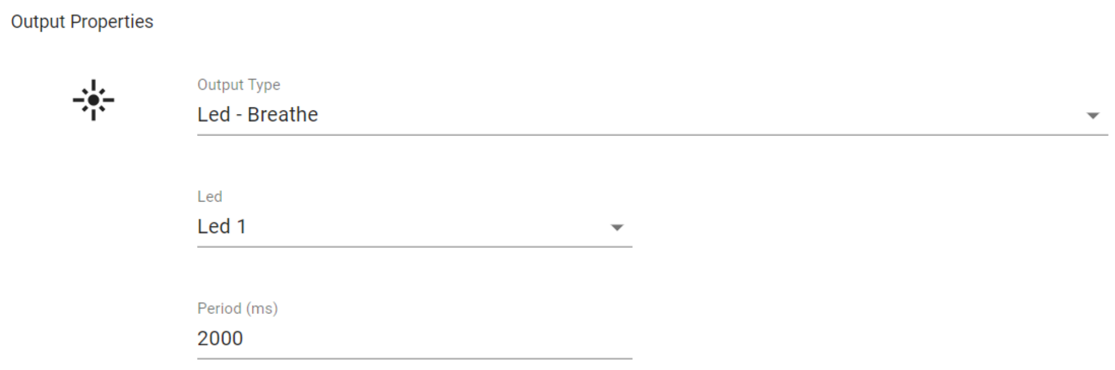
Output Type: Led - Blink

Led: Led 1

Period (ms): 1000

This output blinks the LED with the specified period.

LED - Breathe


Output Properties

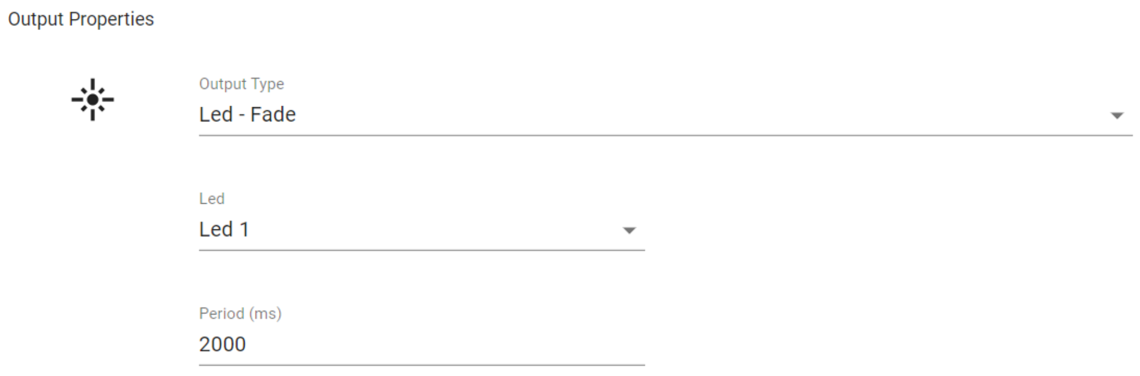
Output Type
Led - Breathe

Led
Led 1

Period (ms)
2000

This output makes the LED 'breathe' by smoothly changing the brightness from zero to full brightness and back again, and then repeats. The period of this breathing can be configured.

LED - Fade


Output Properties

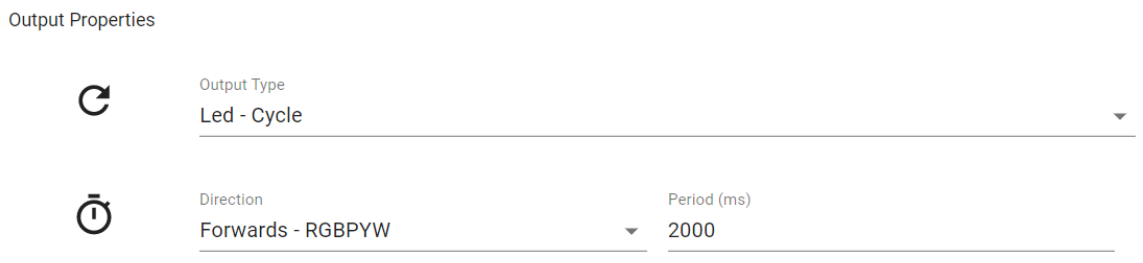
Output Type
Led - Fade

Led
Led 1

Period (ms)
2000

This output fades the LED off over a given period of time.

LED - Cycle


Output Properties

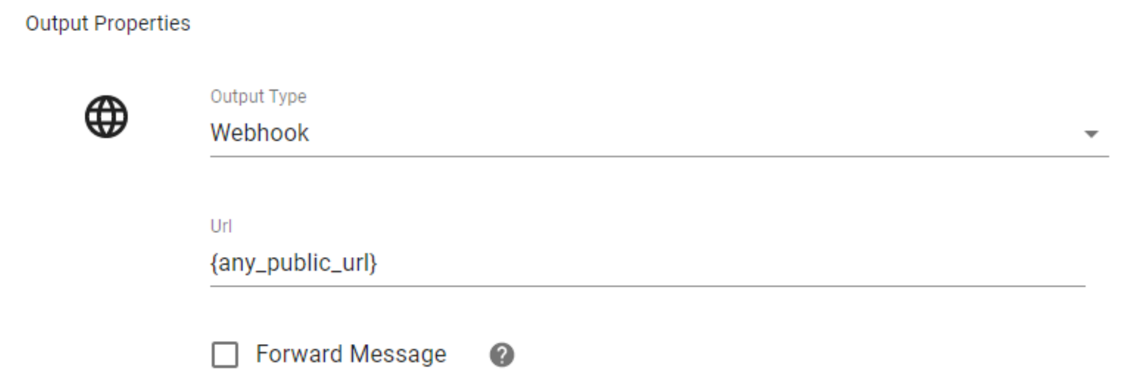
Output Type
Led - Cycle

Direction
Forwards - RGBPYW

Period (ms)
2000

This output applies only to LED 3 as it cycles through all possible colours. The direction of the cycle can be chosen along with the period - the total amount of time it will take to cycle through all colours

Web Hooks

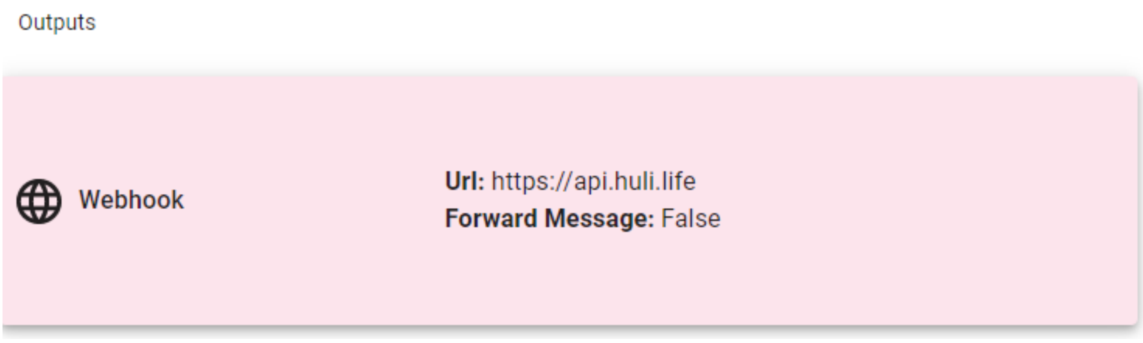

Output Properties

Output Type
Webhook

Url
{any_public_url}

☐ Forward Message

This ouput sends an HTTP post request to the specified URL. The `Forward Message` option allows for the message body of the input on this rule to be forwarded to the specified URL. For example, if the input was also a web hook that posted some payload in the body, this will be retained and posted to the specified URL. If it was a button press as the input, the MQTT JSON message also gets forwarded. For more information for developers and using web hooks for interoperability please see here.


Outputs

Webhook

Url: https://api.huli.life
Forward Message: False

API

Framework and Language

The API was built using the framework ASP.NET Core, built on top of Microsoft’s increasingly trending language, C#. ASP.NET Core is an web API specific framework that extends .NET Core. We are using .NET 5.

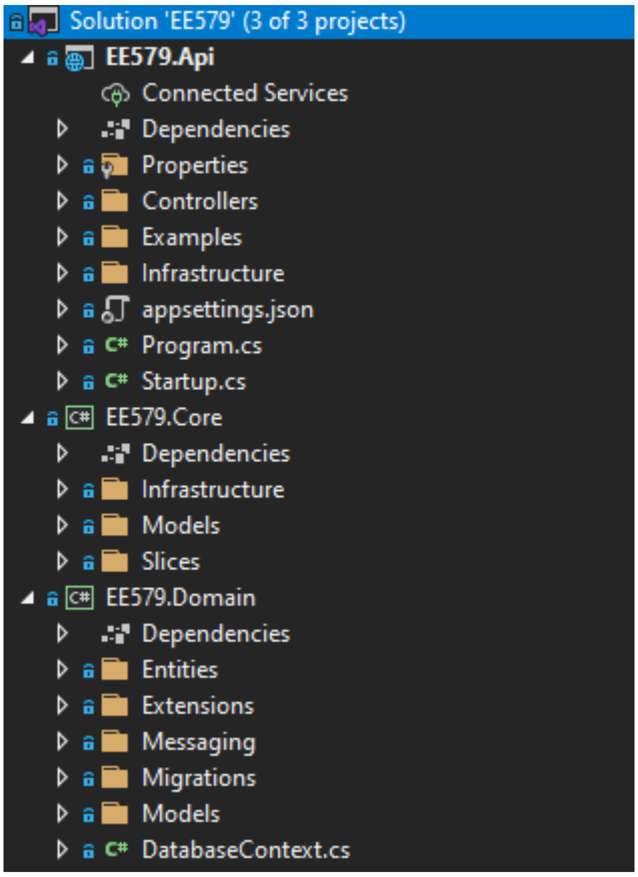
As we have followed industry level design and architectural patterns for ASP.NET Core web APIs, anyone with any ASP.NET Core experience should be able to pick up our API without knowing detailed knowledge of the component level logic. To learn more about ASP.NET Core, see here.

Running the API

1. First install Visual Studio from here
2. Pull our API with `git clone https://github.com/fraserb99/ee579-api.git`
3. Navigate into the `EE579` directory
4. Open `EE579.sln` with Visual Studio. Visual Studio should automatically install all the necessary dependencies
5. Press the green play button on the top toolbar to run the API. This starts the API on localhost:5001

Understanding the API's Architecture and Directory Structure

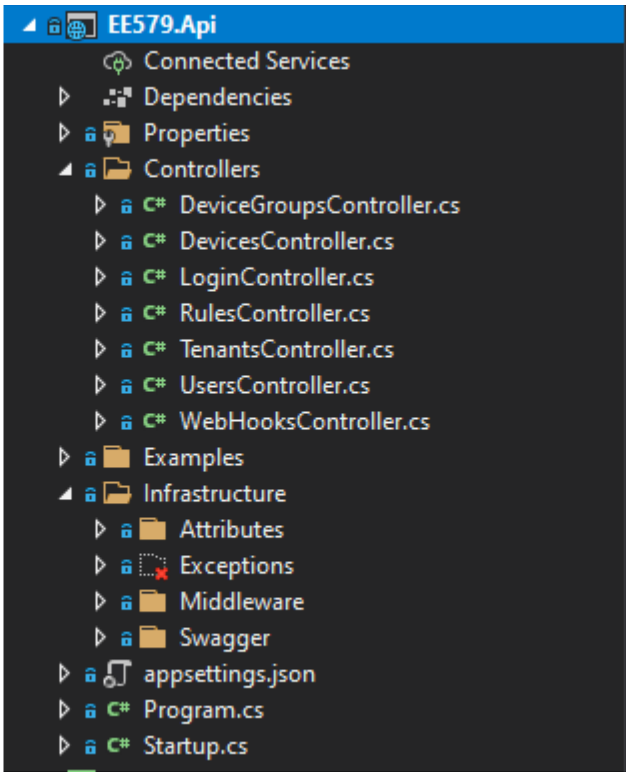
The API is organised into three modules: Api, Core and Domain. The root directory structure for each module is shown below:



```
Solution 'EE579' (3 of 3 projects)
◢ EE579.Api
      Connected Services
   ▷  Dependencies
   ▷  Properties
   ▷  Controllers
   ▷  Examples
   ▷  Infrastructure
   ▷  appsettings.json
   ▷  Program.cs
   ▷  Startup.cs
◢ EE579.Core
   ▷  Dependencies
   ▷  Infrastructure
   ▷  Models
   ▷  Slices
◢ EE579.Domain
   ▷  Dependencies
   ▷  Entities
   ▷  Extensions
   ▷  Messaging
   ▷  Migrations
   ▷  Models
   ▷  DatabaseContext.cs
```

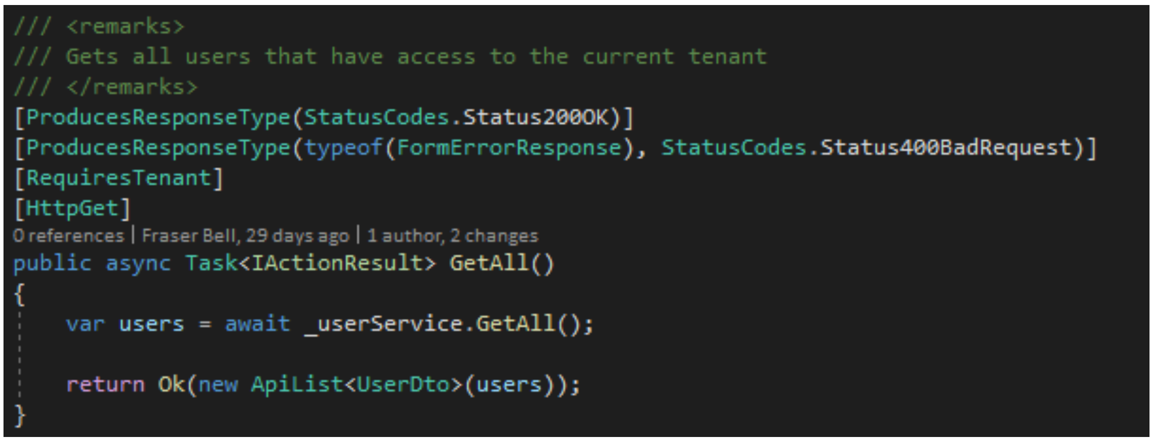
- The `API` module contains functionality related to basic input/output in the API as well as launch configurations and application settings.
- The `Core` module contains all the API logic for handling requests.
- The `Domain` module contains all the mapped database entity objects. An object representation of the database is also stored at the root level in `DatabaseContext.cs.`

API Module



```
◢ EE579.Api
    Connected Services
  ▷ Dependencies
  ▷ Properties
  ◢ Controllers
    ▷ DeviceGroupsController.cs
    ▷ DevicesController.cs
    ▷ LoginController.cs
    ▷ RulesController.cs
    ▷ TenantsController.cs
    ▷ UsersController.cs
    ▷ WebHooksController.cs
  ▷ Examples
  ◢ Infrastructure
    ▷ Attributes
    ▷ Exceptions
    ▷ Middleware
    ▷ Swagger
  ▷ appsettings.json
  ▷ Program.cs
  ▷ Startup.cs
```

Above, is the API Module with some of its main directories expanded. Within the `Controllers` directory, lies all the controllers for the API. A controller is a class that contains methods for the entry points for all our different endpoints. Controllers are split into their category. For example, all endpoints to do with users, such as getting a list of all users in the tennant, lie within the `UsersController` class. This get all endpoint within the UsersController can be seen below:



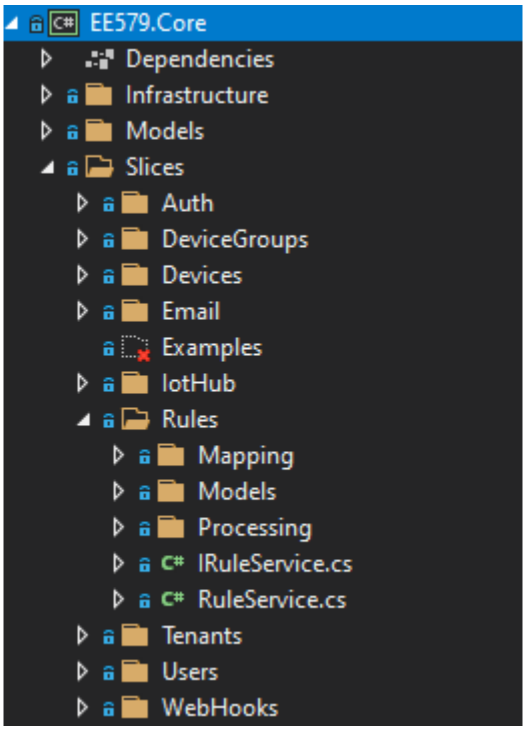
```
/// <remarks>
/// Gets all users that have access to the current tenant
/// </remarks>
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(typeof(FormErrorResponse), StatusCodes.Status400BadRequest)]
[RequiresTenant]
[HttpGet]
0 references | Fraser Bell, 29 days ago | 1 author, 2 changes
public async Task<IActionResult> GetAll()
{
    var users = await _userService.GetAll();

    return Ok(new ApiList<UserDto>(users));
}
```

We can see that there are various "attributes" above the method definition. Attributes can be used for various attribution related configuration to do with a particular method. In this case, there is the `[HttpGet]` attribute that defines this method as a GET request. You can find out more about attributes here. Within the methody body, a call to the method `GetAll()` is invoked upon the user service. This returns a list of user objects. More about services and dependency injection will be covered later. On the last line of this controller method, the list of users is returned to the front end. ASP.NET Core automatically serialises the list of user objects to JSON behind the scenes.

Moving back up one directory from the `Controllers` directory, is the `Infrastructure` directory. Within this directory contains definitions for custom attributes, exceptions, middleware that handles status codes, and as well as the swagger documentation configurations. In the route directory of the API module, lies two important classes: `Program` and `Startup`. `Program` launches the API and boostraps the `Startup` class. Within `Startup` contains many API configurations, service registering and authorization handling. Things such as CORS policies and bearer token authentication go in here.

Core Module

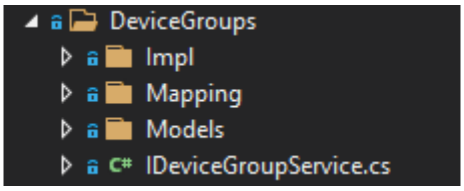


```
EE579.Core
    Dependencies
    Infrastructure
    Models
    Slices
        Auth
        DeviceGroups
        Devices
        Email
        Examples
        IotHub
        Rules
            Mapping
            Models
            Processing
            IRuleService.cs
            RuleService.cs
        Tenants
        Users
        WebHooks
```

Above, is the `Core` Module with some of its main directories expanded. At the highest directory level, is `Infrastructure`, `Models` and `Slices`. Similar to the `Infrastructure` directory in the `API` module, this directory contains classes that provide functionality to the system as a whole. Such as general utlity method overrides and extensions etc.

`Models`, is a directory for data objects. The classes typically should only contain properties and not give any functionality around them. A `Models` directory, can be seen in multiple different places and levels within the API. This is for organisational purposes, for example, all models associated with users will be within a `Models` directory within the `Slices/Users` directory.

Within the `Slices` directory, contains the bulk of the API logic. In the context of the API, a slice, is a grouped section (or slice) of the API, where the grouping is organised by entity relation, such as users, rules, devices etc. That is, all logic handling with regards to that entity, is contained in its own slice.


```
DeviceGroups
    Impl
    Mapping
    Models
    IDeviceGroupService.cs
```

We can see above the `DeviceGroups` slice directory expanded. This shows a typical architecture for a slice. Within contains:

- A `Mapping` directory containing mapping profiles that define behaviour for mapping one object to another (with regards to this slice).
- Another `Models` directory containg all data objects associated with this slice.
- An `Impl` directory (short for implementation) that contains the service class for this slice that implements the respective service interface.
- The respective service interface with regards to this slice.

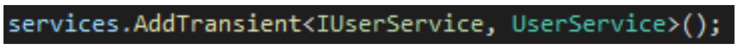
Services

So what actually *is* a service? A service is a logic component used by the API. It is good practice to create a service for every slice in the API. In simple terms, the service will provide functionality and methods for interacting and manipulating the respective entity in that slice. So for example, the user service will have methods for getting a user from the database, creating a new user in the database, updating a user and so forth. The key thing that differentiates a service from a conventional class, is that services get registered in the aforementioned Startup class and can thereby be **dependency injected** into other classes such as controllers or even injected into other services.

DEPENDENCY INJECTING SERVICES

Dependency injection is when an object receives additional objects it depends on. For example, the `UsersController` needs access to the `UsersService` so that when a request to get a user is received, it can call this method on the user service which will return the user. For this to be possible, two steps are required:

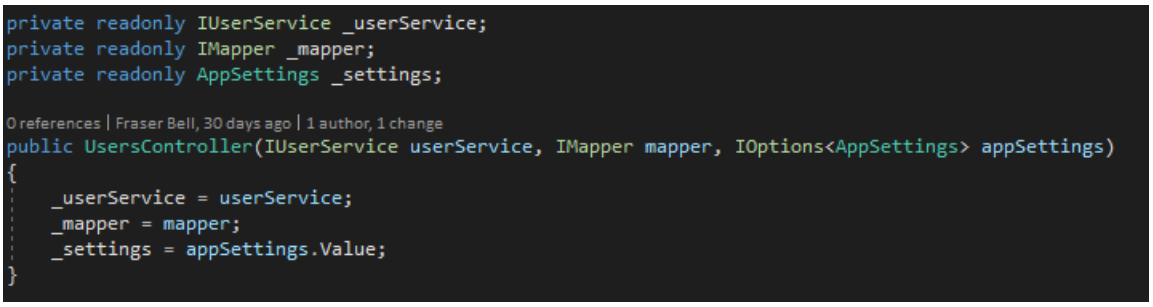
1. **Register the class as a service in Startup.cs** - This is done like so within the `ConfigureServices` method in `Startup.cs`:



```
services.AddTransient<IUserService, UserService>();
```

This tells the API that the interface `IUserService` is parent to the `UserService` implementation. AddTransient means that upon every request to the API, ASP.NET Core will inject an instance of `UserService` with type `IUserService` into all objects that depend on it. AddTransient is so called named due to the transient lifecycle of the injected object. That is, after the request has finished execution, .NET will perform clean up and garbage collection to free up memory. It is injected with type `IUserService` rather than the `UserService` type itself because programming to the interface gives greater code decoupling and abstraction providing a well architectured and maintainable design.

2. **Receive the injection within an object** - Receiving a dependency happens within the constructor of the class. This is shown below:

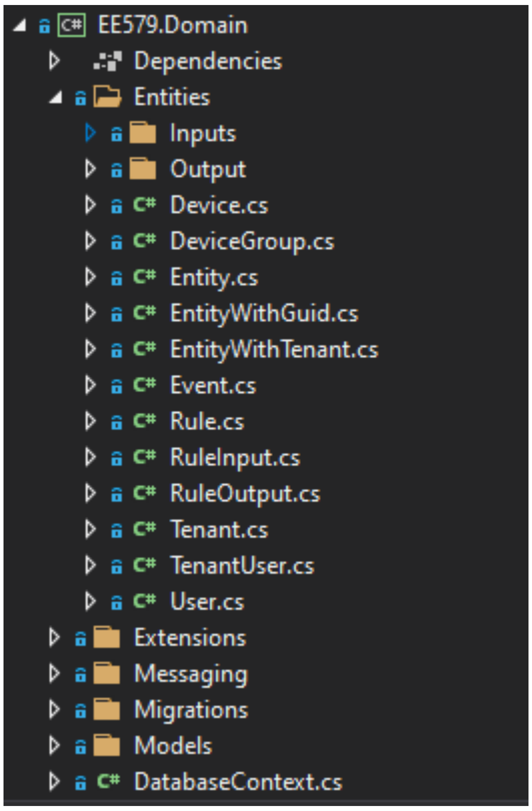


```
private readonly IUserService _userService;
private readonly IMapper _mapper;
private readonly AppSettings _settings;

0 references | Fraser Bell, 30 days ago | 1 author, 1 change
public UsersController(IUserService userService, IMapper mapper, IOptions<AppSettings> appSettings)
{
    _userService = userService;
    _mapper = mapper;
    _settings = appSettings.Value;
}
```

You can see above, that the constructor for the `UsersController` takes an `IUserService` as one of its parameters. What gets passed in for this type, is then set as an instance variable. This is the user service dependency. But then where does the `IUserService` get passed in from? Because of the AddTransient method call made in `Startup.cs` in step 1, ASP.NET Core knows to look for all contructors that take the type `IUserService` as a parameter and will pass in an instance of UserService whenever a request to the API is made. This is seamless to the programmer, one of the benefits of such a design pattern. The dependency can then be interacted with using the instance variable, in this case the `_userService` reference.

Domain Module


```
EE579.Domain
  Dependencies
  Entities
    Inputs
    Output
    Device.cs
    DeviceGroup.cs
    Entity.cs
    EntityWithGuid.cs
    EntityWithTenant.cs
    Event.cs
    Rule.cs
    RuleInput.cs
    RuleOutput.cs
    Tenant.cs
    TenantUser.cs
    User.cs
  Extensions
  Messaging
  Migrations
  Models
  DatabaseContext.cs
```

Above is the directory structure of the domain module with the `Entities` directory expanded. The `Entities` directory contains object representations of database entities. These entity classes are mapped from the database and vice versa. This is achieved by the use of Entity Framework Core or EF Core for short.

EF Core

Entity Framework Core is an object relational mapper. This means that it can map database tables to lists of objects where an object is an entry in the table. For example, a users table with 3 columns: id, username and password would get mapped to a list of `User` objects. Where a `User` object would have propeties of `id: int`, `username: string` and `password: string`. By interacting with these objects within the API in a conventional object orientated fashion, EF Core will update the database seamlessly behind the scenes. For example, to update the password of a user with `id` 1, you would search through the list of `User` objects until you find a `User` with its `id` property value equal to 1 and then updating the password on this selected object. EF Core will then update the database to reflect this change. For creating a `User` it is as simple as adding a new `User` object to the list of users.

Migrations

But then how does the database schema stay in sync with the object representation in the API? Keeping the two domains in sync is inherently simple using the Code First design approach with EF Core Migrations. Code First means that the database schema is reflective of the API domain entities rather than the API domain entities updating to match the database schema. For example, to add a new column/propert to the User domain object (and thus the database) the following steps are required:

1. Open the `User` domain object in Entities/User.
2. Add a new field to the class. E.g `email: string`.
3. In the terminal in Visual Studio, run the command `Add-Migration AddedAnEmailColumn`. ..* This will generate some simple code that will add an `email` column to the database. It is worth quickly reviewing this to make sure it is what you want to do.
4. Run `Update-Database` this will execute the migration code and add the column to the database.

When the API starts up, it will automatically check to make sure it has applied all the latest migrations. This ensures that the database always matches the object representation in the code.

React Front End

Language and Framework

The web interface (or front end) for this system is written in React. React is a javascript framework that enables developers to easily build responsive front ends with its component-based architecture. It is created and maintained by Facebook. Detailed React documentation can be found here.

Single Page Applications

Single Page Applications (SPAs) are a type of web application, along with server-rendered (or multi-page) applications. Traditional websites use server side rendering to load a new page every time the user navigates to a different URL - this page will consist of HTML and any other related files (css, javascript, etc.). In contrast, SPAs use client side languages to dynamically change the web page content without a request to the server. Instead, the client side scripts (most commonly in the form of javascript) are loaded on the first page load and the required data is then loaded from an external API. React supports both server-side rendering and SPAs. THe SPA approach is used in this project as it allows for a more responsive site as the user does not have to wait for a request to be made on every navigation. It is important to understand how SPAs work in order to understand how they can be hosted on the cloud

Running the Site

Prerequisites

- NodeJS > v10

First, clone the github repository:

```
git clone https://github.com/fraserb99/ee579-web.git
```

Change directory to the cloned repo:

```
cd ee579-web
```

Then install the required npm packages:

```
npm i
```

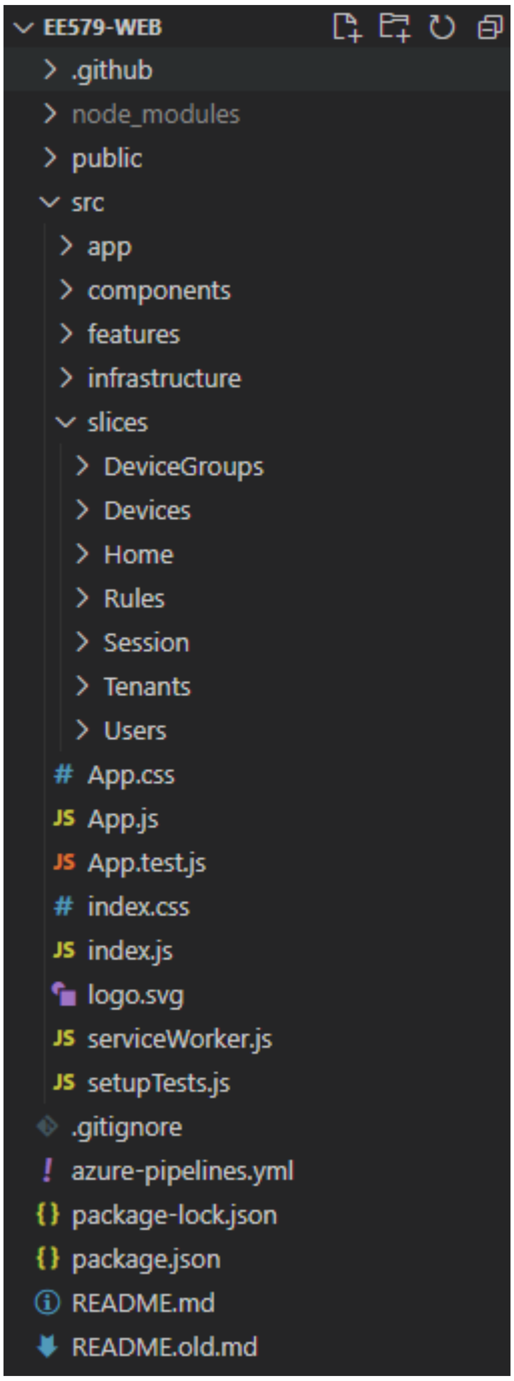
Then run the application:

```
npm start
```

Your browser should now open at `http://localhost:3000`, displaying the site. Run the API alongside this in order for all functionality to work

Application Structure

The folder structure of the application is shown below:



```
EE579-WEB
  > .github
  > node_modules
  > public
  v src
    > app
    > components
    > features
    > infrastructure
    v slices
      > DeviceGroups
      > Devices
      > Home
      > Rules
      > Session
      > Tenants
      > Users
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    serviceWorker.js
    setupTests.js
  .gitignore
  azure-pipelines.yml
  package-lock.json
  package.json
  README.md
  README.old.md
```

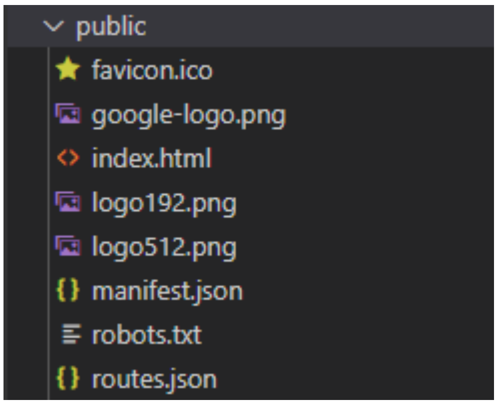
package.json

The `package.json` file is automatically generated and stores information about the installed npm packages and their versions. Wen a new package is installed via `npm i`, this file is updated. React-specific script references are also stored in the scripts section

```
    "scripts": {
        "start": "react-scripts start",
        "build": "react-scripts build",
        "test": "react-scripts test",
        "eject": "react-scripts eject"
    },
```

This allows the project to be built and run with `npm start`

`/public`



```
public
    favicon.ico
    google-logo.png
    index.html
    logo192.png
    logo512.png
    manifest.json
    robots.txt
    routes.json
```

This folder contains the `index.html` used by the React DOM to render the React app, along with various meta files such as the `favicon.ico`. `routes.json` is an important file used to configure the Azure Static App Settings.

`/src`

This folder contains the majority of the code for the site. The most important files and subdirectories and their functions are as follows:

- **App.js** - This is the entry point to the React application
- **/slices** - This folder contains most of the code for the site, split into sections with similar functionality

- **/infrastructure** - This folder contains code used for general configuration of libraries and integration with other services (e.g. API integration, middleware, React-Redux config, routing config)
- **/components** - This folder contains components shared across multiple slices of the application

Patterns and Libraries Used

This section introduces some of the common patterns and libraries used in the application.

Components

React uses a component-based architecture. This allows the UI to be split into reusable pieces and each piece can be tested individually. This project primarily uses functional components, consisting of an input and output value:

- **Props** - this is the components input. This allows parent components to pass different values to the component to change its behaviour.
- **Return value** - the data components return is what is used to display content. This takes the form of JSX which allows UI to be written in the style of HTML while enabling inline javascript to be used.

A simple example component definition is shown below:

```
function HelloWorld(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

This component can then be used:

```
<HelloWorld name='World' >
```

More information about React components can be found here.

React-Router

This application uses React-Router to provide routing. This is done by wrapping the app in the `<ConnectedRouter>` component and then using `Route` components as shown below.

```
<Route path='/signin' component={SignInPage} />
```

The offic

React-Redux

React-Redux is a state management library for React, and an alternative to React's Context API. It provides an API that can be used to store and access data from a central location. This allows data to be shared efficiently across components without passing it as props.

This application most commonly uses it to store data retrieved from the REST API. The Redux store is used to store and organise the response data returned from API requests. Middleware can also be used in conjunction with Redux to perform some actions on every request. For example, the `injectAuthMiddleware` middleware adds the current user's JWT to the `Authorization` header for every request. This means it does not have to be manually added.

```
export const injectAuthMiddleware = store => next => action => {
    const callApi = action[RSAA];
    if (isRSAA(action)) {
        const state = store.getState();
        callApi.headers = Object.assign({}, callApi.headers, {
            Authorization: `Bearer ${state.session.get('token')}`,
        });
    }

    return next(action);
};
```

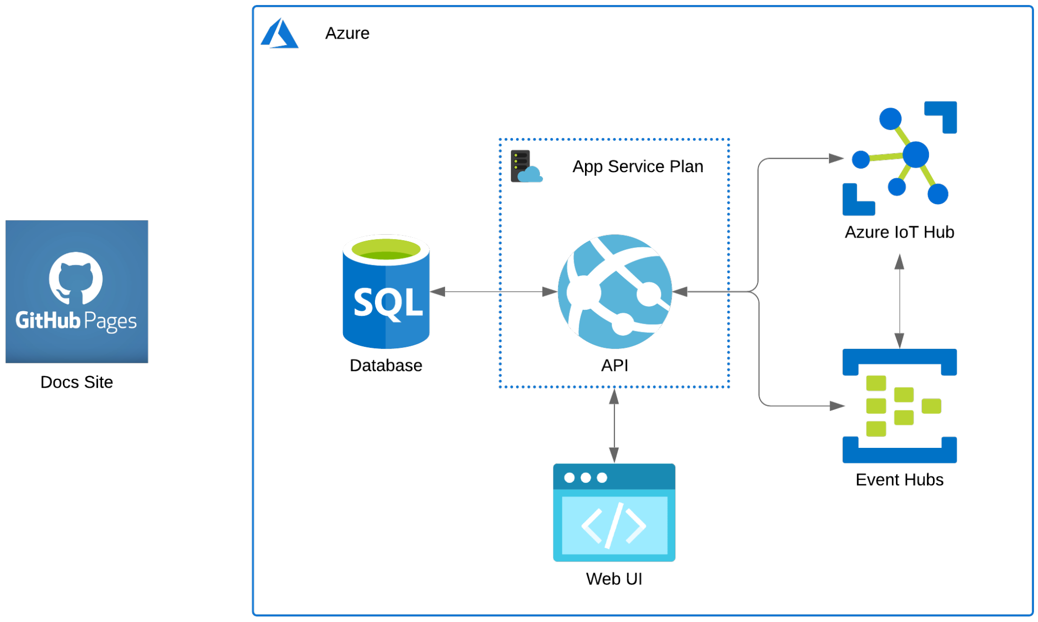
Cloud Architecture

Cloud Provider

The majority of the cloud components of this system are currently hosted on the Microsoft Azure platform. The cloud provider is not tightly coupled to the application for the most part, so you are free to use the provider of your choice. The only exception to this is for IoT Hub and consequently Event Hubs, for which more information can be found here. It should also be noted that this docs site is hosted using GitHub Pages - more detail about how this works can be found in the Docs Site Section.

The Azure Portal is available at portal.azure.com and can be used to create and manage cloud resources.

Architecture Overview

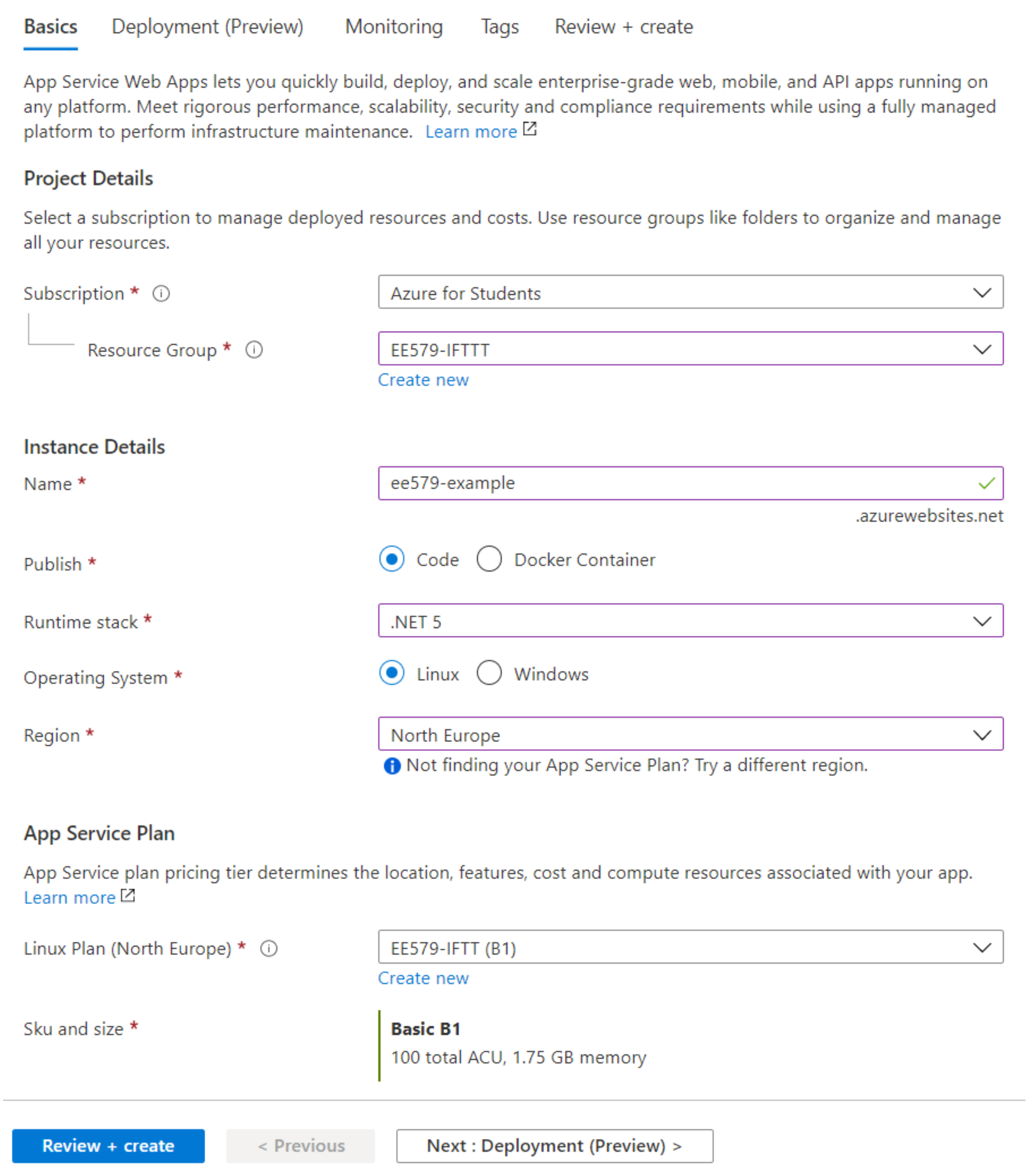


This diagram shows how the cloud services integrate with each other. This section will detail how each of these services is hosted to allow you to replicate the hosting setup on the provider of your choice.

API

The API is hosted as an Azure App Service. This is a Platform-as-a-Service (PaaS) offering from Azure that enables applications to be run without having to configure the underlying server as everything like the web server config is handled behind-the-scenes by Azure. Before an App Service can be created, an App Service Plan must be created. App Service Plans can contain multiple App Services, with the scale being configured at the App Service Plan level.

The following screenshot shows an example configuration for an App Service Plan.



App Service Web Apps lets you quickly build, deploy, and scale enterprise-grade web, mobile, and API apps running on any platform. Meet rigorous performance, scalability, security and compliance requirements while using a fully managed platform to perform infrastructure maintenance. Learn more

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * : Azure for Students

Resource Group * : EE579-IFTTT
Create new

Instance Details

Name * : ee579-example .azurewebsites.net

Publish * : [✓ Code] Docker Container

Runtime stack * : .NET 5

Operating System * : [✓ Linux] Windows

Region * : North Europe
Not finding your App Service Plan? Try a different region.

App Service Plan

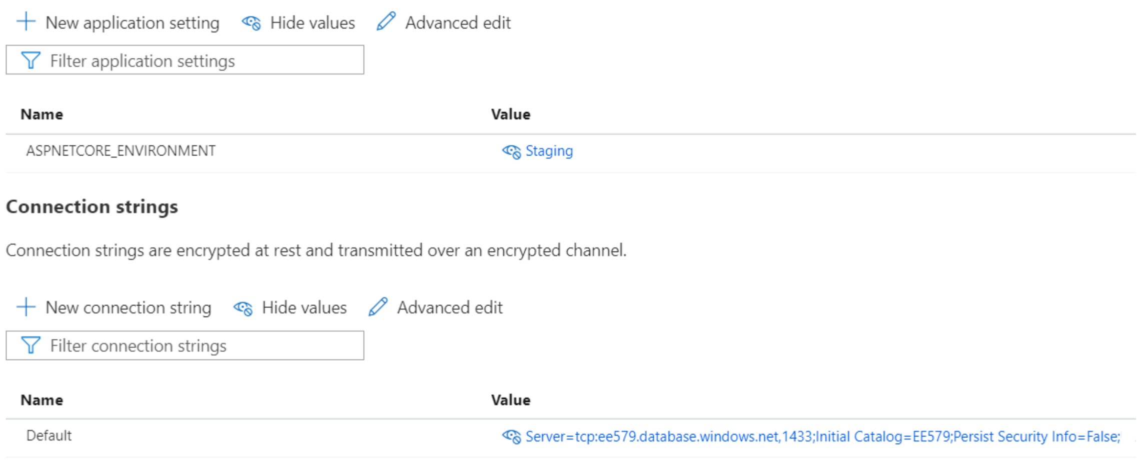
App Service plan pricing tier determines the location, features, cost and compute resources associated with your app.
Learn more

Linux Plan (North Europe) * : EE579-IFTT (B1)
Create new

Sku and size * : **Basic B1**
100 total ACU, 1.75 GB memory

The only options that must be set as shown in the screenshot are the `Publish: Code` and `Runtime Stack: .NET 5` options. All of the rest are up to you. Our API has cross-platform support and can run on Windows or Linux. We cannot provide any direct instructions as we have not done it, but the API can be hosted on other providers like AWS, or alternatively can be run on a local server running IIS, Kestrel, or another web server.

There are several environment variables that must be set for the API to function correctly. If you are hosting the app on an App Service, it is as simple as setting 2 values in the configuration section of the App Service:


+ New application setting
Hide values
Advanced edit
Filter application settings

Name	Value
ASPNETCORE_ENVIRONMENT	Staging

Connection strings

Connection strings are encrypted at rest and transmitted over an encrypted channel.

+ New connection string
Hide values
Advanced edit
Filter connection strings

Name	Value
Default	Server=tcp:ee579.database.windows.net,1433;Initial Catalog=EE579;Persist Security Info=False;

If you are hosting on another service then the environment variables should be set directly with the following names:

- `ASPNETCORE_ENVIRONMENT` - This should be set to `Staging`
- `ConnectionStrings__Default` - This should be the connection string for your database

You may also need to edit some values in `appsettings.Staging.json`. The values from this file get loaded as configuration options.

```
    "AppSettings": {
        "AdminUrl": "https://www.ee579-group4.net",
        "ApiUrl": "http://ee579-dev-
api.azurewebsites.net",
        "SmtpSettings": {
            "EmailEnabled": true,
            "Host": "smtp.gmail.com",
            "Port": 587,
            "Email": "ee579.ifttt@gmail.com",
            "Password": "**********"
        }
    }
```

For example, if you are hosting the API on another URL, you should change the `"ApiUrl"` value to reflect this. You can also change the Smtp server used for sending email here.

Database

We are using Azure SQL to host our database, a cloud database service using Microsoft's SQLServer. You are in no means tied to this SQL variation, however. One of the benefits of using EF Core is that it abstracts the database itself from the code. This means that you are free to use any of the supported database providers.

To use a different database provider, you should first set the connection string as described in the previous section. You then must also change the following line in `Startup.cs` where the database connection is configured.

```
private void ConfigureEfCore(IServiceCollection services)
{
    services.AddDbContext<DatabaseContext>(opts =>
        opts.UseLazyLoadingProxies()
            .UseSqlServer(Configuration.GetConnectionString("Default"
}
```

Change this to use the corresponding function, e.g., `UseSqlite` if you are using a Sqlite database.

Web Interface

The web interface is hosted as an Azure Static Web App. Azure Static Web Apps allow static sites to be hosted without using a web server. This is especially useful when the site is an SPA - as described here. SPAs don't require any processing to be performed on the server to serve the response, the server only needs to return the file containing the client-side code. Static Web Apps allow these files to be served through the use of a CDN, providing users with extremely fast page load times.

The site can also be hosted on a traditional server as well. Running `npm build` will generate a production-ready, minified build containing an `index.html` with references to the minified javascript files. This build can then be deployed to the wwwroot folder of any web server and served.

There is an additional piece of configuration required when configuring the web server running the site. Because the React app is an SPA, the web server only sees a single `index.html` file so assumes that the only root is the default `/`. This means that if someone navigates to `/sign-up` for example, the web server will not find a `sign-up.html` file, so will return a 404. To fix this we must route all requests to `/`. If hosting using a Static Web App, this can be done using a `routes.json` file in the `/public` folder.

routes.json

```
{
    "routes": [
        {
            "route": "/*",
            "serve": "/index.html",
            "statusCode": 200
        }
    ]
}
```

The method for doing this may vary between web servers. to take IIS as an example, a `web.config` file can be used to configure the routing.

web.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <remove name="pushState" />
                <rule name="pushState" stopProcessing="true">
                    <match url=".*" />
                        <conditions logicalGrouping="MatchAll">
                        <add input="{REQUEST_FILENAME}" matchType="IsFile"
negate="true" />
                        <add input="{REQUEST_FILENAME}"
matchType="IsDirectory" negate="true" />
                        </conditions>
                        <action type="Rewrite" url="/" />
                    </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

IoT Hub and Event Hubs

Note

The IoT Hub setup is somewhat involved and requires more setup than the other components. If you are looking to replicate our cloud architecture we recommend that you continue to use our IoT Hub to simplify setup on both devices and the server.

IoT Hub implements a publish-subscribe pattern, used in this system to facilitate communication between the IoT devices and server.

IoT Hub is not a traditional MQTT broker, so to read messages sent from devices the server cannot just subscribe to their topics. Instead, IoT Hub exposes these messages through an Event Hub built-in-endpoint. The API then reads the messages from this Event Hub endpoint and processes them asynchronously.

A Blob Storage Account is also required to read messages from the built-in-endpoint. This is used to store checkpoint files which allow the API to keep its place in the message queue, like a bookmark. This prevents messages from being read twice. The storage account was omitted from the architecture diagram for brevity.

IoT Hub also allows you to expose device connection state events by routing them through an Event Hub. This means that when a device connects to the IoT Hub, a connection state message is routed to the corresponding Event Hub. Our application then reads the messages from this endpoint and updates the corresponding device's connection state in the database. This connection state is then displayed in the UI as shown here.

If you do create your own IoT Hub, Event Hub, and storage account, simply change the connection strings in `IotHubWorkerService.cs` and `IotMessagingService.cs` to point to your resources.

Docs Site

The docs site is the site this documentation is currently available from - docs.ee579-group4.net. This site is created using MkDocs, a static site generator that converts Markdown files into a static site composed of html, css, and javascript.

Unlike the other components, this site is not hosted on Azure. Instead, GitHub pages is used. MkDocs provides easy integration with GitHub pages through their CLI. The following commands show how you can contribute to the documentation and deploy it.

1.
   ```
   pip install mkdocs && pip install mkdocs-material
   ```

2.
   ```
   git clone https://github.com/fraserb99/ee579-docs.git
   ```

3.
   ```
   cd ee579-docs
   ```

4.
   ```
   mkdocs serve
   ```
5. Make any changes you would like at this point

6.
   ```
   mkdocs gh-deploy
   ```