# Documentation

## Table of Contents

[Project Description 2](#_Toc492395746)

[Project Goals 2](#_Toc492395747)

[App Overview 2](#_Toc492395748)

[App Features 2](#_Toc492395749)

[Use Cases/Mockups 3](#_Toc492395750)

[Summary of Use Cases 3](#_Toc492395751)

[Core Functionality 4](#_Toc492395752)

[Subordinate Functionality 10](#_Toc492395753)

[Prerequisites 16](#_Toc492395754)

[Architecture 16](#_Toc492395755)

[Setup 17](#_Toc492395756)

[Simulated Devices and Azure IoT Hub 17](#_Toc492395757)

[Azure SQL and Azure App Services 20](#_Toc492395759)

[Azure Stream Analytics: Connect IoT Hub and Azure SQL Database 21](#_Toc492395760)

[Azure ML 23](#_Toc492395761)

[Azure Stream Analytics: Integrating Azure ML 27](#_Toc492395762)

[Unity 29](#_Toc492395763)

[Scripts 29](#_Toc492395764)

[Summary of Scripts: 29](#_Toc492395765)

[Game Objects and Prefabs 31](#_Toc492395766)

[Important Game Objects in the Unity Hierarchy 31](#_Toc492395767)

[Important Prefabs 32](#_Toc492395768)

[Known Issues 32](#_Toc492395769)

[Dependencies Used 32](#_Toc492395770)

[Extensibility 32](#_Toc492395771)

[Add Methods of Identifying IoT Devices 32](#_Toc492395772)

[Adding/Changing IoT Data Attributes of Interest 33](#_Toc492395773)

[Adding/Changing Methods for IoT Data Display 34](#_Toc492395774)

[Other Changes to the Checklist 35](#_Toc492395775)

[Change Frequency of Querying for Live Mode 35](#_Toc492395776)

### Project Description

HoloIoT is a project aimed to help factory workers by combining augmented reality and IoT. It allows a user to identify an IoT device, query its telemetry, and visualize the data as holograms. In doing so, a worker can view telemetry live, hands-free, and at the location of the device. This is a drastic improvement over the status quo where workers either use hand-held devices or go to a control room to view telemetry.

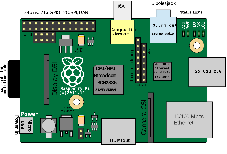
### Project Goals

The goal of this project was to create a HoloLens application with an end-to-end IoT solution that could empower factory workers. We aimed to build a cloud-based backend with an interactive UX and write documentation oriented towards developers that includes setup, code, and extensibility. While this project is self-containing, we expect it to serve as a basis for other IoT applications. Thus, the project is structured so that developers can easily modify and add to it. This extensibility can be in the form of different data sources, other methods to identify IoT devices, 3D models of devices, maintenance records, etc.

### App Overview

[](https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=0ahUKEwjG5tiPko_WAhXLgVQKHdibD4IQjRwIBw&url=http://spancs.com/services/cloud-services/&psig=AFQjCNEF79ZFWfgnOmKzfJ3IFne52JAR6A&ust=1504738519450203)



[](https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwiqpeq5kI_WAhWorFQKHaWQDw4QjRwIBw&url=https://en.wikipedia.org/wiki/Raspberry_Pi&psig=AFQjCNHP0aguNx2nGXYREzIw5r6yQcCSwA&ust=1504738109697106)

IoT devices send telemetry to a data source in the cloud. A user can identify a particular device using QR code scanning or text recognition. The user can then query the database and view the telemetry as a table or chart for the device.

### App Features

1. IoT Object Identification: system pulls frames from camera for up to 30 seconds
   1. QR Code Scanning: ZXing.Net C# library finds a QR code in the frames and decodes it to a device ID
   2. Optical Character Recognition: Windows.Media.Ocr finds text in the frames and uses this as a device ID
2. Anomaly Detection in IoT Data/Telemetry
   1. System generates a list of devices with anomalies in the time range
   2. Once a device is identified, system detects the number of anomalous data points the device measured in the time range
3. Audio-Based System Alerts
   1. System alerts the user of whether the object identification method succeeded or failed
   2. Systems alerts the user of how many devices measured anomalies in the time range
   3. System alerts the user of how many anomalies a specific device measured in the time range
4. Input Gestures
   1. Air-Tap: User can press buttons and check objects on the checklist
   2. Voice Input: Used as an alternative to air-tapping for buttons and the checklist
5. Entering Data through Gestures
   1. Air-Tap and Number Pad: User air-taps a number, a number pad opens, user air-taps numbers on the pad to select them
   2. Air-Tap and Hold: user air-taps or air-taps and holds a number, number increments
   3. Dictation: user specifies the date using a regular expression
6. 3D Visualization of IoT Data
   1. Table: contains the minimum, maximum, average, and standard deviation of an attribute for the device and all devices in the factory
   2. Chart: graphs the telemetry of a device’s attribute; the time is on the x-axis and the measurement on the y-axis
7. Live and Historic Data Query Mode
   1. Live Mode: the system queries the data source every 7 seconds and updates the display
   2. Historic Mode: the system displays the telemetry from within a static time range

### Use Cases/Mockups



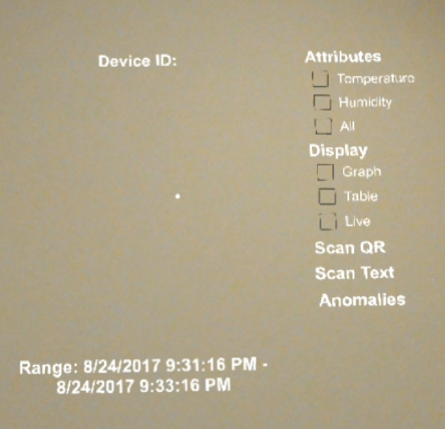
#### Summary of Use Cases

* **Identify IoT Device** – identify the specific device that a user is looking at. Device identification can be done in the following ways (scenarios):
  + Visual Marker – Identify using a distinct marker placed on the device
  + Text Recognition – Identify device by doing optical character recognition of text markings on device
* **Adjust Display and Attribute Preferences** – describes how the user can adjust the way the data is displayed
* **Query Telemetry** – the system queries the data source to get the telemetry for both a specified device and all devices within the time range
* **View Graph** – the system displays the graph of the telemetry
* **Calculate Table Values** – the system calculates the minimum, maximum, average, and standard deviation of the telemetry.
* **View Table** – displays the table with cells for the minimum, maximum, average, and standard deviation for both the device and all devices within a specified time range
* **Change Query Range** – the user navigates to the display where the time range can be changed
* **Change Ranges Display with Air-Tap and Number Pad** – the user specifies a time range by selecting numbers on a number pad
* **Change Ranges Display with Air-Tap and Hold** – the user taps and holds a number, causing the number to increment
* **Change Ranges Display with Dictation** – the user specifies a time range by dictating regular expressions
* **Change Ranges Display with Voice Command** – the user uses “Yesterday” or “Today” voice commands to specify a time range.
* **Set Query Range** – The user sets the new time range
* **Detect Anomalous Devices** – the system creates a list of devices with at least one anomaly in the time range
* **Detect Number of Anomalies** – the system detects the number of anomalous data points for a specific device within the time range
* **Live Mode** – the system queries the data every 7 seconds and updates the display

#### Core Functionality

|  |
| --- |
| Name: **Identify IoT Device**  Description:  Identify the device that the user is looking at.  Preconditions:  User is standing in front of a device attached to an IoT sensor and gazing at it.  Postconditions:  User can view the telemetry data for the device.  Basic Course of Action: QR Code Scan   1. A user gazes at a marker on an object attached to an IoT device. 2. The user air-taps a button labeled “Scan QR” or says, “Scan QR code”. 3. The system updates the display with “Scanning” and uses text to speech to say, “Scanning QR code”. 4. The system analyzes frames from the camera looking for QR codes. 5. The system identifies the QR code and decodes it to a device ID. 6. The system updates the display with the device ID. 7. The system uses text to speech to say, “Scan successful”.   Basic Course Uses:  Can be used in a scenario where the number of devices is small and it’s plausible to place markers on all devices. Should not be used in a scenario where the marker is likely to be covered, fade, or rust.  Alternate Course A: The System Identifies IoT Devices by Text Recognition  A.1. A user is gazing at a text identifier on the device.  A.2. The user air-taps a button labeled, “Scan Text” or says, “Scan Text”.  A.3. The system updates the display with “Scanning” and uses text to speech to say, “Scanning text”.  A.4. The system analyzes frames from the camera looking for text.  A.5. The system identifies text and uses this as the device ID.  Alternate A Uses:  Can be used in a scenario where the identifier is not likely to be covered, fade, or rust.  Exception Case A: Identifier Not Recognized  E.5. The system fails to find an identifier after 30 seconds, and the function times out.  E.6. The system updates the display with “not found” as the device ID.  E.7. The system uses text to speech to say, “Scan failed”. |

|  |
| --- |
| Name: **Adjust Display and Attribute Preferences**  Description:  The user chooses options for the type of display and the attribute(s) to view.  Preconditions:  The app is open.  Postconditions:  When the user looks at a device, the system displays the telemetry according to the preferences.  Basic Course of Action: Air-Tap   1. The user air-taps options from the Attributes and Display sections of the checklist. 2. The system responds by checking or unchecking the option.   Alternate Course A: Voice Command  A.1. The user chooses options from the Attributes and Display sections of the checklist by verbally saying, “[name of preference] mode” (Example: “graph mode”). |



**Figure 1 Main Screen.** The main screen includes a header for the device ID, a range button that allows the user to change the time range, and a preferences panel. The preferences panel contains checklists for the attributes and display in addition to buttons that start device identification and anomaly detection. The main screen tags along with the user’s gaze.

|  |
| --- |
| Name: **Query Telemetry**  Description:  Query the telemetry data for a particular device and time range.  Preconditions:  A device ID has been identified.  Postconditions:  If the query results are not empty, the user can view the telemetry. Otherwise, the system updates the display with “No data to display”.  Basic Course of Action:   1. The system displays a progress ring to the user. 2. The system queries the device’s telemetry within the time range. 3. The system queries the telemetry for all devices within the time range. 4. The system removes the progress ring.   Exception Case A: No Data  A.5. The system updates the display with “No data to display”. |

|  |
| --- |
| Name: **View Graph**  Description:  The user wants to view the telemetry for an attribute as a graph.  Preconditions:  Query Telemetry use case and Adjust Display and Attribute Preferences use case. For the latter, both the Graph Display option and an Attribute option must be checked.  Postconditions:  The user can select different preferences to change what is shown on the graph.  Basic Course of Action:   1. The system plots the data for the measurements of a device’s attribute(s) with a solid line. The time is on the x-axis and the measurement is on the y-axis. 2. The system updates the graph’s header with the attribute name. 3. The system updates the graph’s labels with the minimum and maximum values of the time and measurements.   Alternate Course of Action A: Display All Attributes  A.4. The system creates a legend indicating which line color corresponds to each attribute. |



**Figure 2 Graph View.** Graph view displays telemetry for the attribute(s) checked as a graph and is anchored in space a few feet above where the device is identified. It includes labels for the time range and the minimum and maximum y-value. In all mode, the graph contains a legend that shows the user which line corresponds to each attribute.

|  |
| --- |
| Name: **Calculate Table Values**  Description:  System makes statistical calculations based on the telemetry.  Preconditions:  Query Telemetry use case.  Postconditions:  View Table use case.  Basic Course of Action:   1. The system calculates the minimum, maximum, average, and standard deviation of the device’s telemetry for each attribute 2. The system calculates the minimum, maximum, average, and standard deviation of all devices’ telemetry for each attribute. |

|  |
| --- |
| Name: **View Table**  Description:  The user wants to view the telemetry for an attribute as a table.  Preconditions:  Query Telemetry use case and Adjust Display and Attribute Preferences use case. For the latter, both the Table Display option and an Attribute option must be checked.  Postconditions:  The user can select different preferences to change what is shown on the graph.  Basic Course of Action:   1. The system displays the minimum, maximum, average, and standard deviation of the attribute’s measurements from the specified IoT device within the time range. 2. The system displays the minimum, maximum, average, and standard deviation of the attribute’s measurements from all IoT devices within the time range. 3. The system updates the table’s header with the attribute name.   Alternate Course of Action A: Display All Attributes  A.1. The system does the Basic Course of Action for all attributes.  A.2. The system places the displays for each attribute next to each other. |



**Figure 3 Table View.** Table view contains the minimum, maximum, average, and standard deviation of the measurements for the device and all devices in the factory during the time range. The table is anchored in space a few feet above where the device is identified. All mode displays a table containing this information for all attributes.

#### Subordinate Functionality

|  |
| --- |
| Name: **Change Query Range**  Description:  The user wants to change the time range to display.  Precondition:  The app is open.  Postcondition:  The Ranges Display is open, and the user can change the time range.    Basic Course of Action:   1. The user air-taps the Ranges button or says, “Change Range”. 2. The system closes every open display. 3. The system opens the Ranges Display. |

|  |
| --- |
| Name: **Change Ranges Display with Air-Tap and Number Pad**  Description:  The user wants to change the time range to display.  Precondition:  Change Query Range use case.  Postcondition:  Set Query Range use case.    Basic Course of Action:   1. The user air-taps a number. 2. The system opens a number pad. 3. The user air-taps numbers on the number pad. 4. The user air-taps the Enter button. 5. The system closes the number pad. 6. The system updates the number with the user specified number.   Alternate Course of Action A: User Presses Enter Button on Number Pad without Pressing Numbers  A.6. The system updates the number with the previous number.  Alternate Course of Action B: User Presses Exit Button on Number Pad  B.4. The user air-taps the Exit button.  B.5. The system closes the number pad.  B.6. The system updates the number with the previous number. |



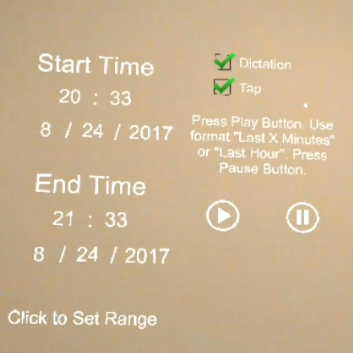
**Figure 4 Number Pad.** In the ranges screen, users can tap on a number and open the number pad. They can tap numbers on the number pad to change the numbers appearing in the ranges screen.

|  |
| --- |
| Name: **Change Ranges Display with Air-Tap and Hold**  Description:  The user wants to change the time range to display.  Precondition:  Change Query Range use case.  Postcondition:  Set Query Range use case.    Basic Course of Action:   1. The user air-taps the Tap button or says, “Tap Mode”. 2. The system displays a check mark next to the Tap button. 3. The user taps or taps and holds a number. 4. The system steadily increments the number and updates the display. |



**Figure 5 Tap Mode.** In tap mode, a user can either tap a number or tap and hold a number. This will cause the number to increment.

|  |
| --- |
| Name: **Change Ranges Display with Dictation**  Description:  The user wants to change the time range to display.  Precondition:  Change Query Range use case.  Postcondition:  Set Query Range use case.    Basic Course of Action   1. The user air-taps the Dictation button or says, “Dictation Mode”. 2. The system displays a check mark next to the Dictation button. 3. The system updates the display with a panel. 4. The system displays instructions for the user to say, “Start Time”, “End Time”, “Start Date”, “End Date”, or “Up To Date” on the panel. 5. The user says one of the above commands. 6. The system updates the panel with a Play and Pause buttons. 7. The system updates the panel with instructions on how to start/stop dictation and an example of the regular expression to use. 8. The user air-taps the Play button. 9. The panel displays the text, “Start recording” and starts dictation mode. 10. The user dictates. 11. The system listens to the dictation and updates the panel with the phrase the user dictates. 12. The user air-taps the pause button. 13. The system ends dictation mode. 14. The system updates the panel with the specified time, an Exit button, and an Enter button. 15. The user air-taps the Enter button. 16. The panel closes. 17. The system updates the corresponding numbers on the Ranges Display.   Alternate Course of Action A: The User Presses the Exit Button  A.15. The user air-taps the Exit button on the panel.  A.16. The panel closes.  A.17. The numbers on the Ranges Display are left unchanged.  Exception Case A: The User Starts Dictation and Doesn’t Dictate  E.A.10. The user does not dictate for 5 seconds.  E.A.11. The system ends dictation mode and updates the display with the text, “Dictation has timed out. Please try again.”.  Exception Case B: The User Dictates and Stops Dictating without Pressing Pause  E.B.11. The user stops dictating for 20 seconds.  E.B.12. The system ends dictation mode and updates the display with the text, “Dictation has timed out. Please try again.”.  Exception Case C: The System Doesn’t Recognize the Phrase  E.C.14. The system updates the display with the text, “Incorrect date or time format.”. |

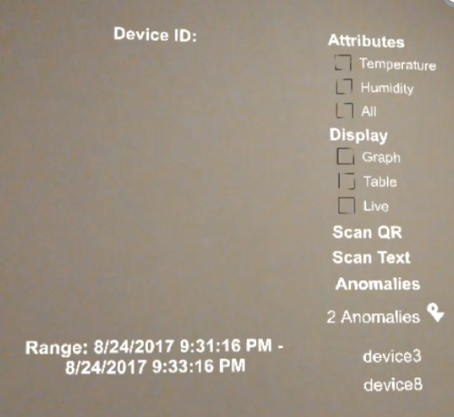


**Figure 6 Dictation Mode.** In dictation mode, a user can dictate a regular expression to change the time range. The above instructions appear to users after they've said, “Up to date”.

|  |
| --- |
| Name: **Change Ranges Display with Voice Command**  Description:  The user wants to change the time range to display.  Precondition:  Change Query Range use case.  Postcondition:  Set Query Range use case.    Basic Course of Action:   1. The user says, “Today” or “Yesterday”. 2. The system updates the Ranges Display’s Start Time and End Time based on the command given. |

|  |
| --- |
| Name: **Set Query Range**  Description:  The user wants to change the time range to display.  Precondition:  The Ranges Display is open.  Postcondition:  The range is updated and the system queries the data source if a device ID is recognized.    Basic Course of Action:   1. The user air-taps the Set Ranges button or says, “Set Ranges”. 2. The system closes the Ranges Display. 3. The system opens the displays that were open before the user entered the Ranges Display.   Alternate Course of Action A: Invalid Time Range  A.4. The user has specified an incorrect time range (either the end time is before the start time or one of the times is after DateTime.Now).  A.5. The system updates the text on the Set Ranges button to say, “Click to Set Range: Invalid Range”.  Alternate Course of Action B: Invalid Date  B.4. The user has specified a date that doesn’t exist (Example: February 31st).  B.5. The system updates the text on the Set Ranges button to say, “Click to Set Range: Invalid Date”. |

|  |
| --- |
| Name: **Detect Anomalous Devices**  Description:  The user wants a list of anomalous devices.  Precondition:  The app is open.  Postcondition:  The user can see a dropdown list of anomalous devices.    Basic Course of Action:   1. The user air-taps the Anomalies button or says, “show anomalies”. 2. The system displays a progress ring. 3. The system queries the telemetry that had anomalies within the time range. 4. The system creates a list of devices that have at least one anomalous data point. 5. The system removes the progress ring from the display. 6. The system uses text to speech to say, “[number of anomalous devices] devices with anomalies” and creates a dropdown list with the device IDs of anomalous devices. |



**Figure 7 Anomalous Devices.** By pressing the anomalies button on the preferences panel or by saying, “show anomalies”, the system will create a dropdown list containing the device IDs of anomalous devices. The system defines an anomalous device as a device that has at least one anomalous data point in the time range.

|  |
| --- |
| Name: **Detect Number of Anomalies**  Description:  The system alerts the user to the number of anomalous data points.  Precondition:  Query Telemetry use case  Postcondition:  The user is aware of the number of anomalies for a device within the time range.    Basic Course of Action:   1. The system counts the anomalous data points in the device’s telemetry. 2. The system uses text to speech to say, “[number of anomalous data points] anomalous data points detected” |

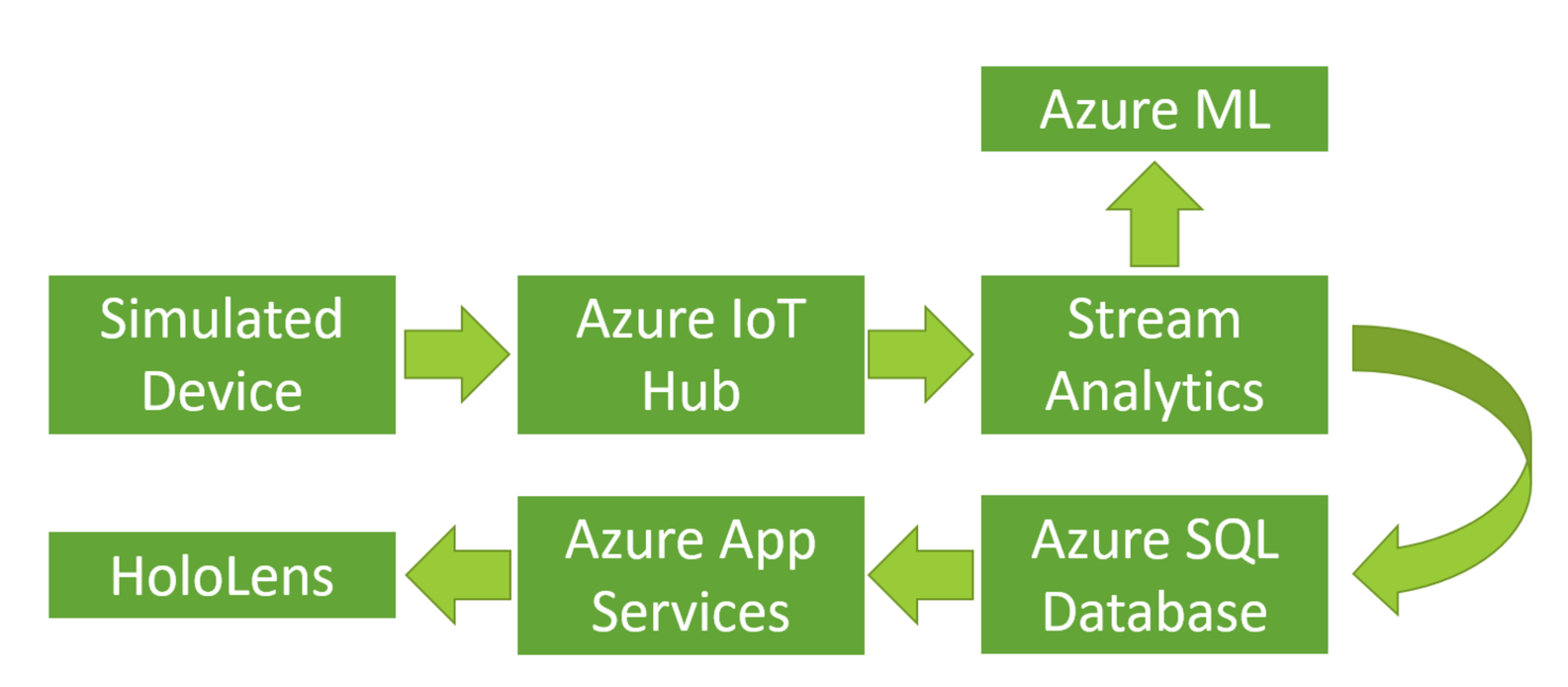
|  |
| --- |
| Name: **Live Mode**  Description:  The user wants to view the most up-to-date telemetry.  Precondition:  A device has been identified.  Postcondition:  The user views the most up-to-date data and can switch back to Historic mode.    Basic Course of Action:   1. The user air-taps the Live button or says, “Live Mode”. 2. The system displays a check next to the Live button. 3. Every 7 seconds, the system calls the Query Telemetry and Detect Number of Anomalies use cases. 4. If a display is open, the system calls the use case for the corresponding display. |

### Prerequisites

Requires Unity v5.3 or greater as [UnityWebRequest](https://docs.unity3d.com/Manual/UnityWebRequest.html) and [JsonUtility](https://docs.unity3d.com/ScriptReference/JsonUtility.html) features are used.

### Architecture

HoloIot has a cloud based backend with the following architecture:

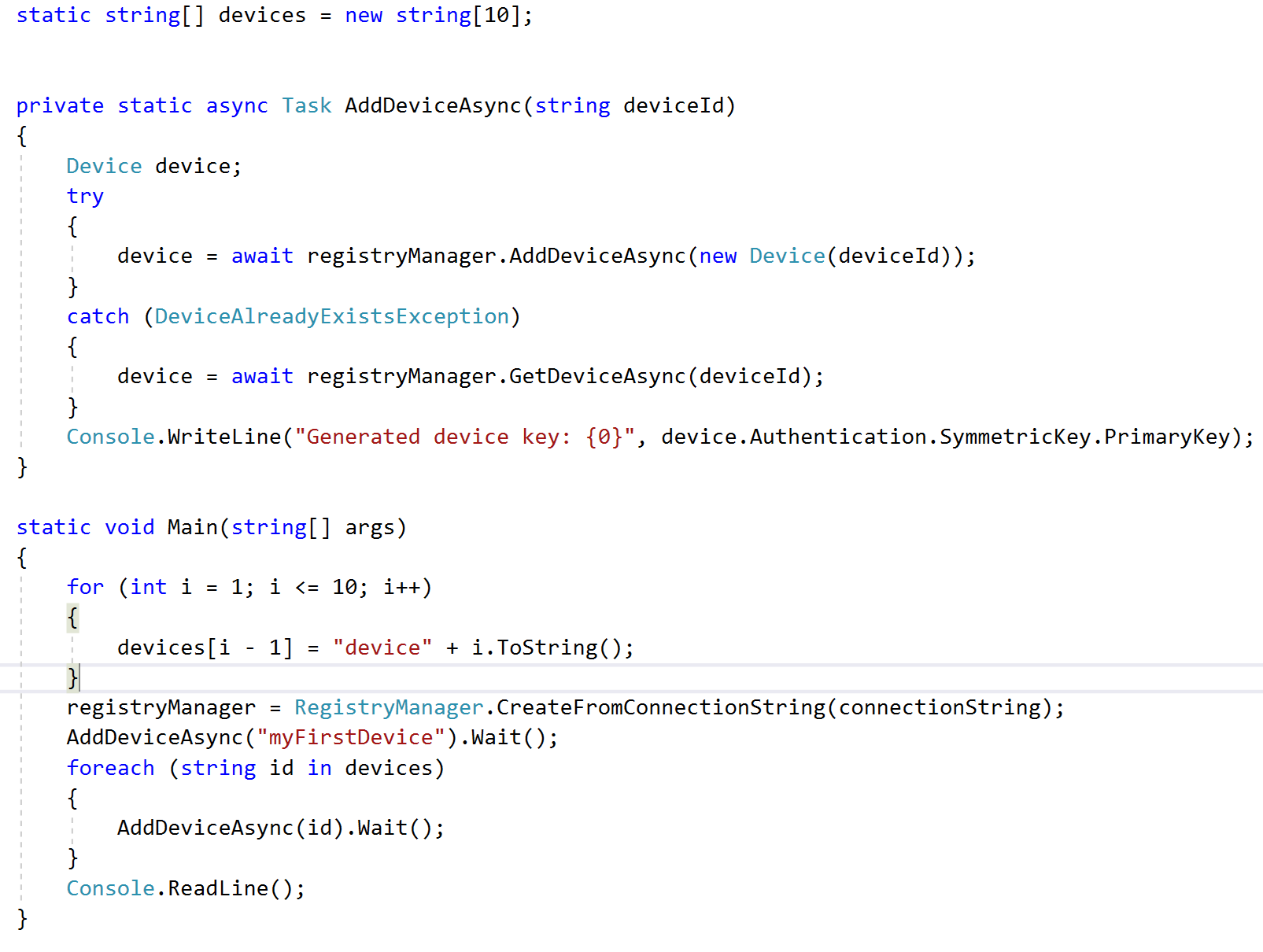


The architecture is based off of the Azure IoT Suite remote monitoring preconfigured solution: <https://docs.microsoft.com/en-us/azure/iot-suite/iot-suite-what-are-preconfigured-solutions>. First, a C# application is used to simulate ten devices that are outputting telemetry every few seconds. An IoT hub receives the telemetry, functioning as a cloud gateway for the IoT solution. A stream analytics job runs a PCA-based anomaly detection model on the data and sends it to an Azure SQL database, where the telemetry and a binary value representing whether an anomaly is present is stored. An Azure App Services mobile app creates an OData data source linked to the SQL database. Finally, the HoloLens sends OData queries to the mobile app, receives the results, and displays the telemetry.

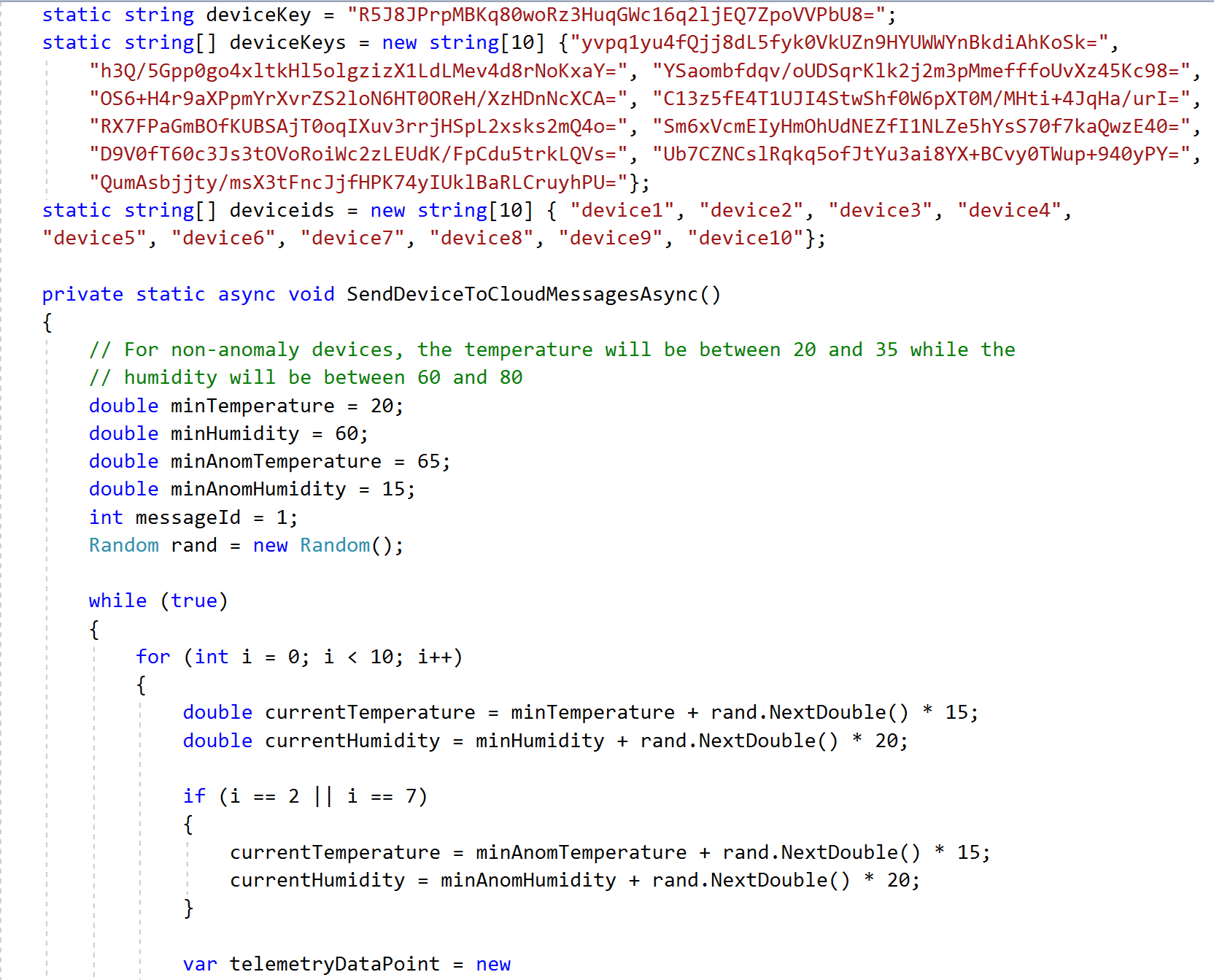
### Setup

#### Simulated Devices and Azure IoT Hub

The solution for the simulated devices is found in the SimulatedDevices folder of the repository. To set up the IoT hub and the simulated devices, follow the tutorial here: <https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-csharp-csharp-getstarted> with a few modifications. The tutorial creates one device that communicates with the IoT hub. For this project, 10 devices were created. To do this, their AddDeviceAsync function and Main method in the CreateDeviceIdentity project were modified to be as follows:



The SendDeviceToCloudMessagesAsync function in SimulatedDevice was also changed:





#### 

The IoT hub can be created exactly as it is in the tutorial.

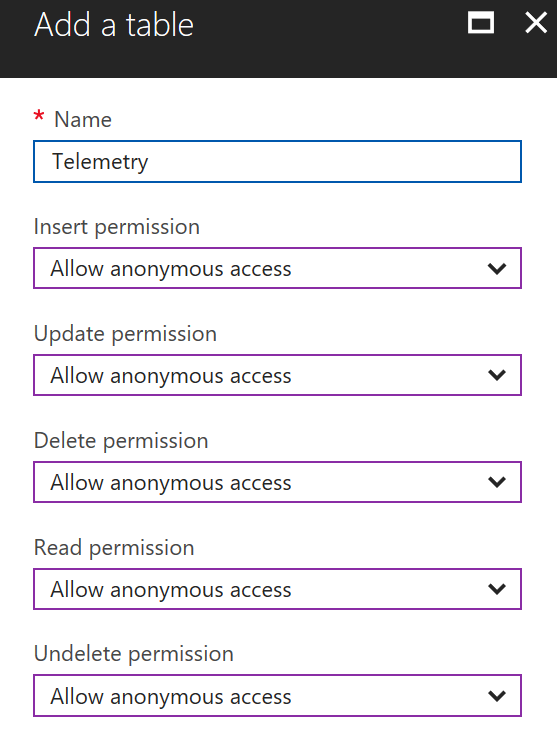
#### Azure SQL and Azure App Services

An Azure SQL server and database can be created following this tutorial: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-get-started-portal>. The source should be a blank database and the cheapest pricing tier can be used.

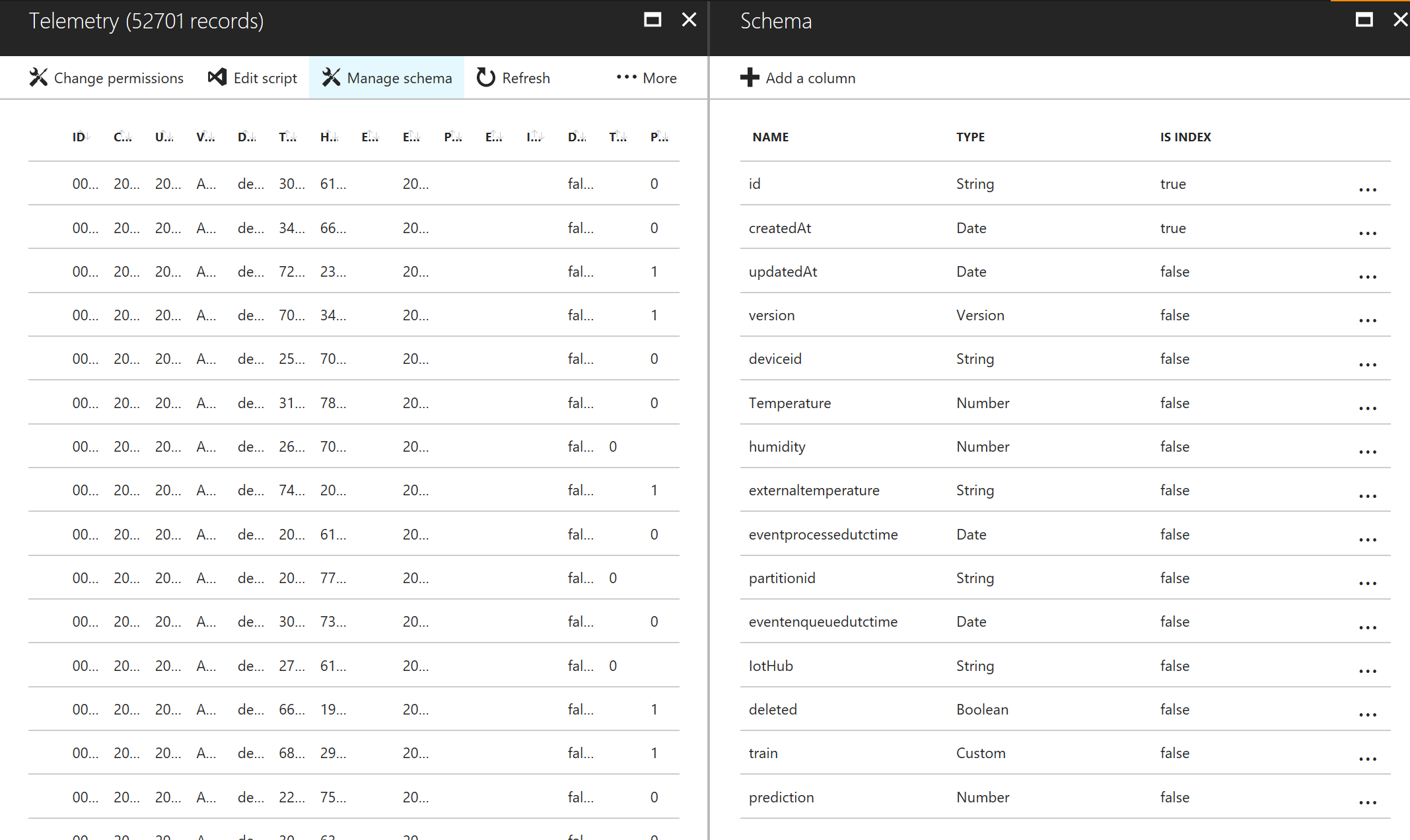
To set up Azure App Services, follow the first part of the video here: <http://www.deadlyfingers.net/azure/azure-app-services-for-unity3d/>.

Once Azure App Services is set up, add a data connection between the app and the SQL database, and add a table to Easy Tables. The Add Data Connection and Adding a New Table sections explain how to do this here: <https://blog.xamarin.com/getting-started-azure-mobile-apps-easy-tables/>.

In the Easy tables section of the mobile app, add a table called, “Telemetry”.



Add columns to the table. When the table is created, it’ll have columns for id, createdAt, updatedAt, version, and deleted. At the minimum, add columns for deviceid, temperature, humidity, eventprocessedutctime, train, and prediction.

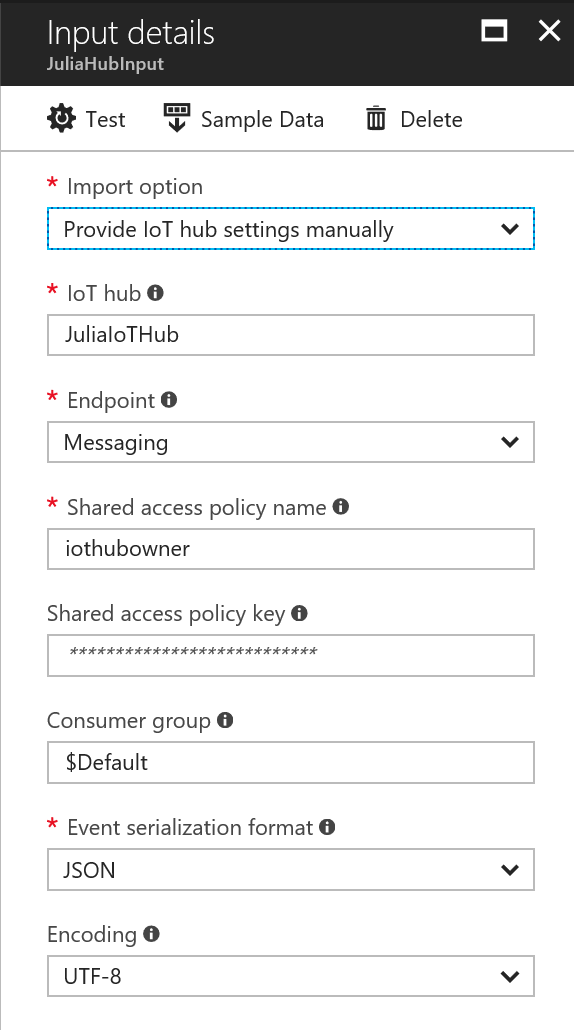


To verify that this part works, you can run a query on the database to insert an entity into the Telemetry table and make sure that the entity appears in the easy table.

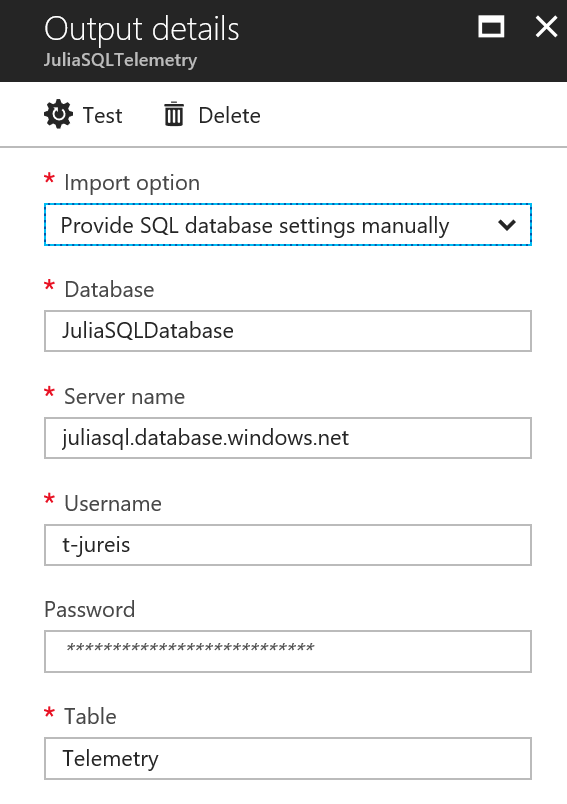
#### Azure Stream Analytics: Connect IoT Hub and Azure SQL Database

The next step is to create a basic stream analytics job that transfers telemetry from the IoT hub to the database. After the machine learning model is added, this job will change slightly, but for now, simply transfer telemetry to the database to generate data to train the machine learning model on. To do this follow this walkthrough with a few modifications: <http://gunnarpeipman.com/2016/02/beer-iot-using-stream-analytics-to-save-data-from-iot-hub-to-sql-database/>.

To the stream analytics job, add the IoT hub as the input:

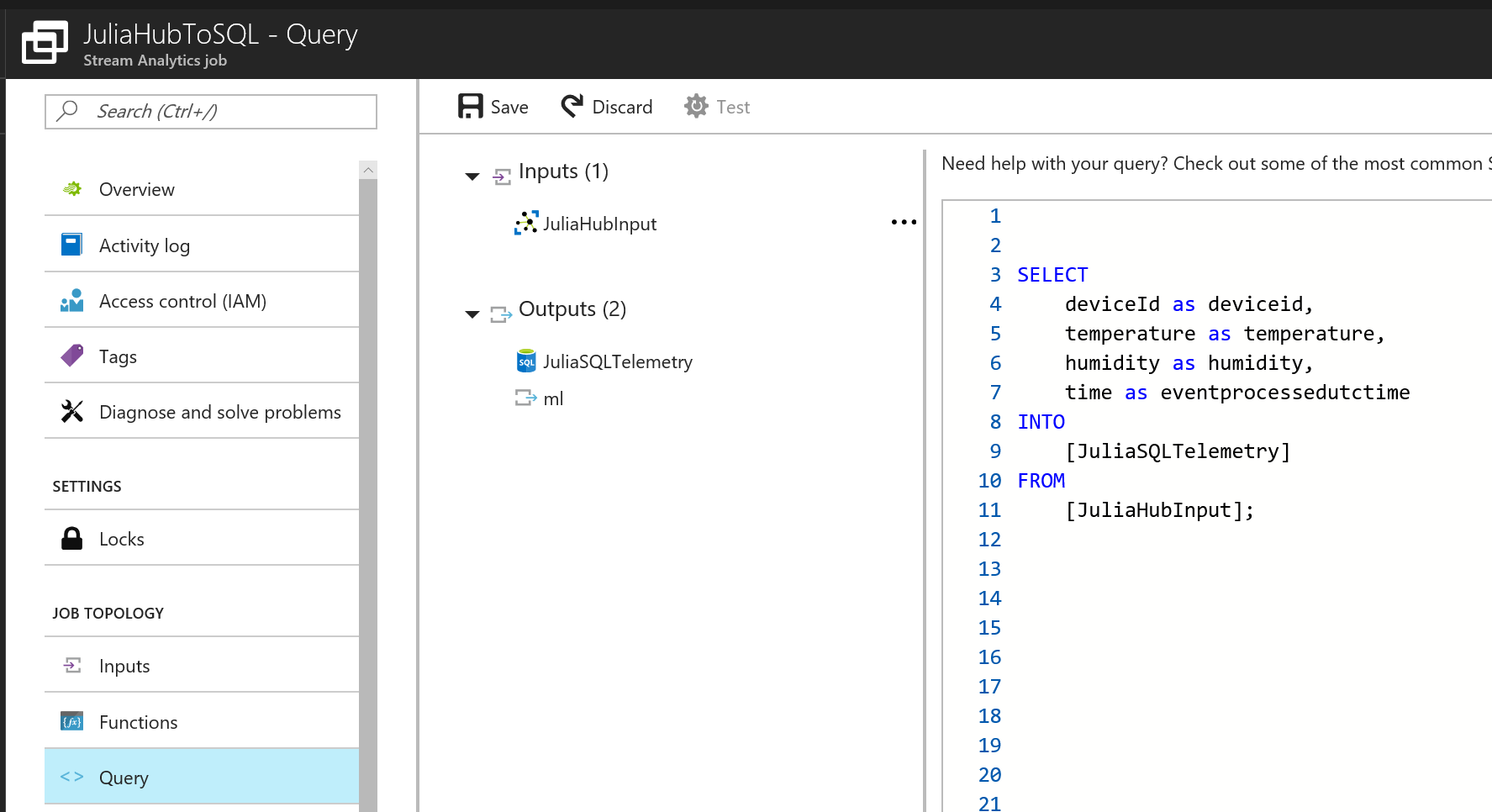


And the SQL database as the output:





Use the following query:

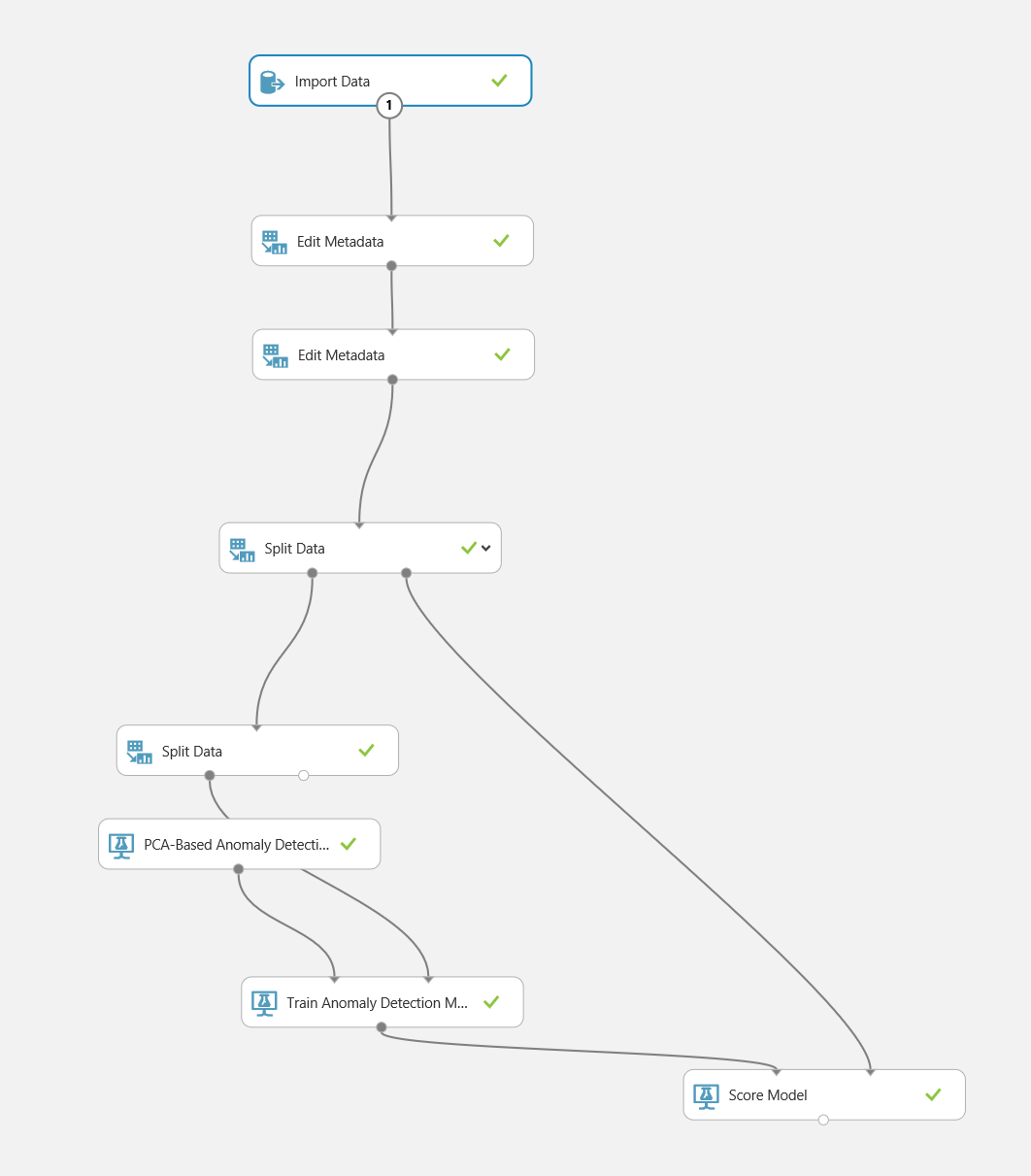


Then, run the stream analytics job and the simulated devices app. Run both until there are 10,000 rows in the table to use as training data for the machine learning model. Finally, use the *train* column in the table to label the data as normal or anomalous. The simulated devices are programmed so that device3 and device8 have anomalous data while the other devices have normal telemetry. Thus, query the SQL database, updating the *train* column so that if the deviceid is device3 or device8, the column value is 1; otherwise, set it to 0. Alternatively, the simulated devices could be modified to give *train* a value.

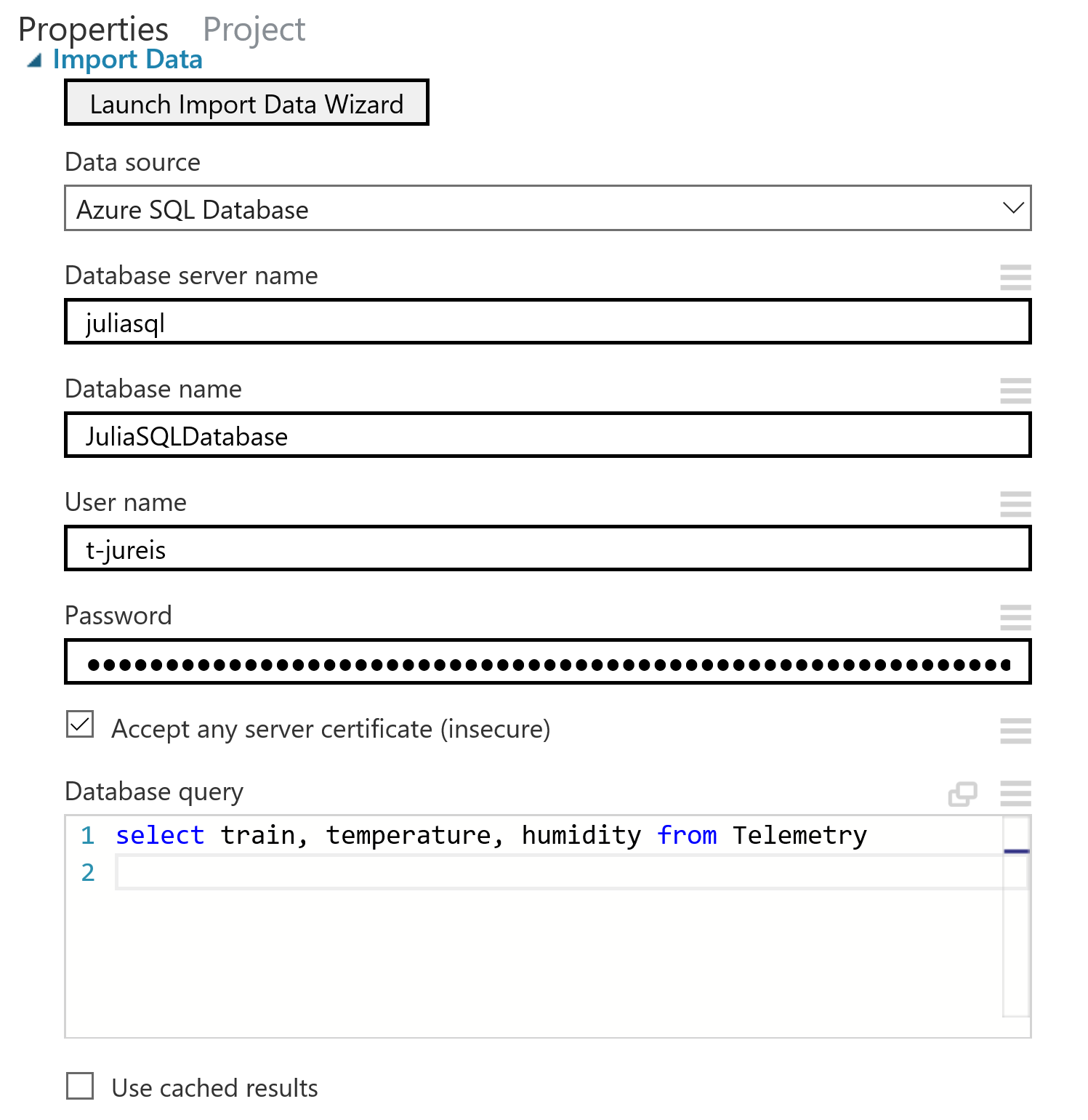
#### Azure ML

An Azure Machine Learning model was created to detect anomalies using PCA-based anomaly detection. The model trains on data that’s labeled normal and uses distance metrics to determine whether a new input is anomalous. Below explains how to replicate this model. Alternatively, Azure also provides an API for anomaly detection in time series data that could probably be used: <https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-apps-anomaly-detection-api>.

To create the PCA model, follow this tutorial with adjustments to create the experiment if this is your first time using Azure ML: <https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-walkthrough-develop-predictive-solution>.

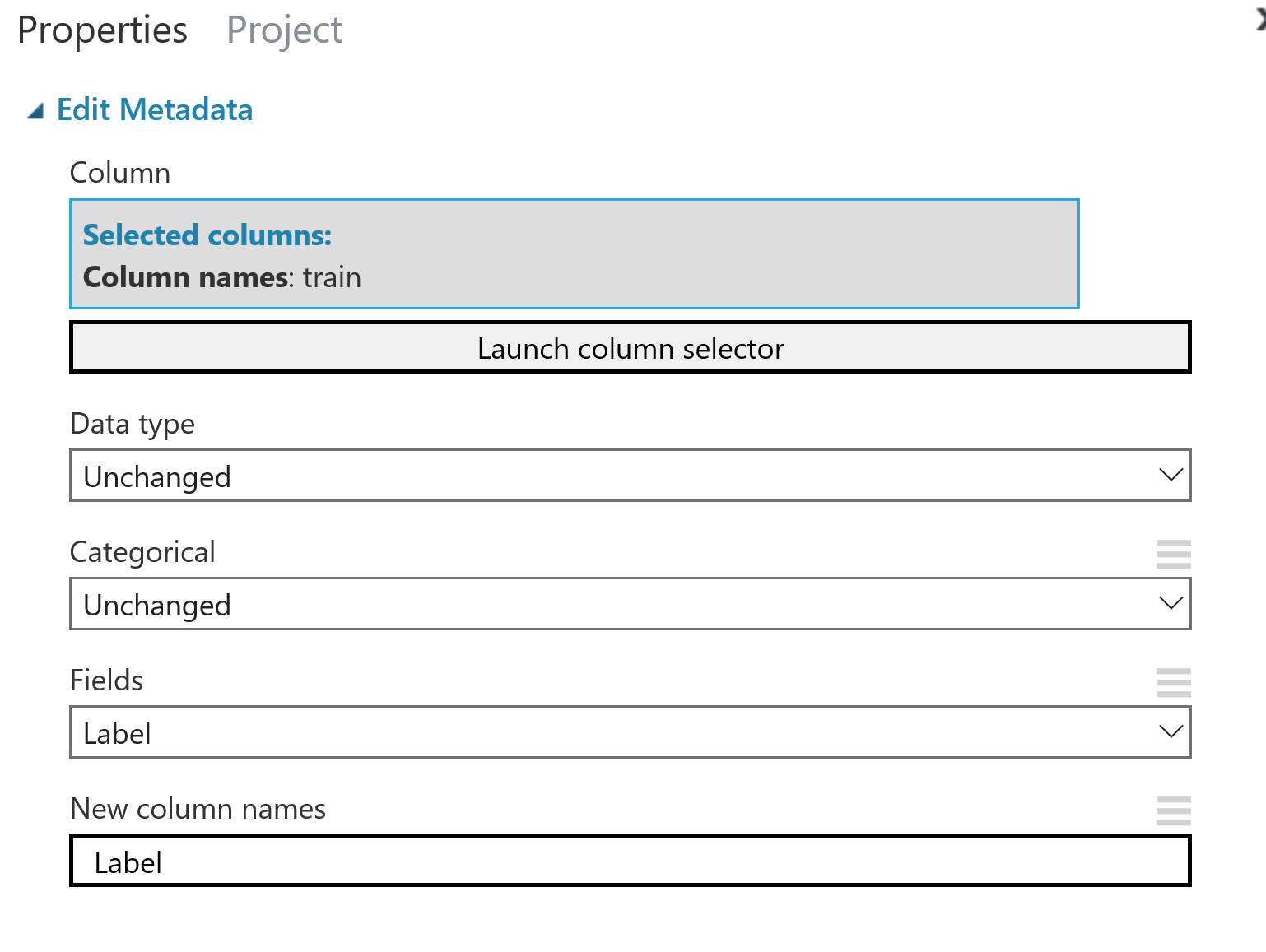
Here’s an overview of the completed PCA-based training model: 

First, import the data from the database:

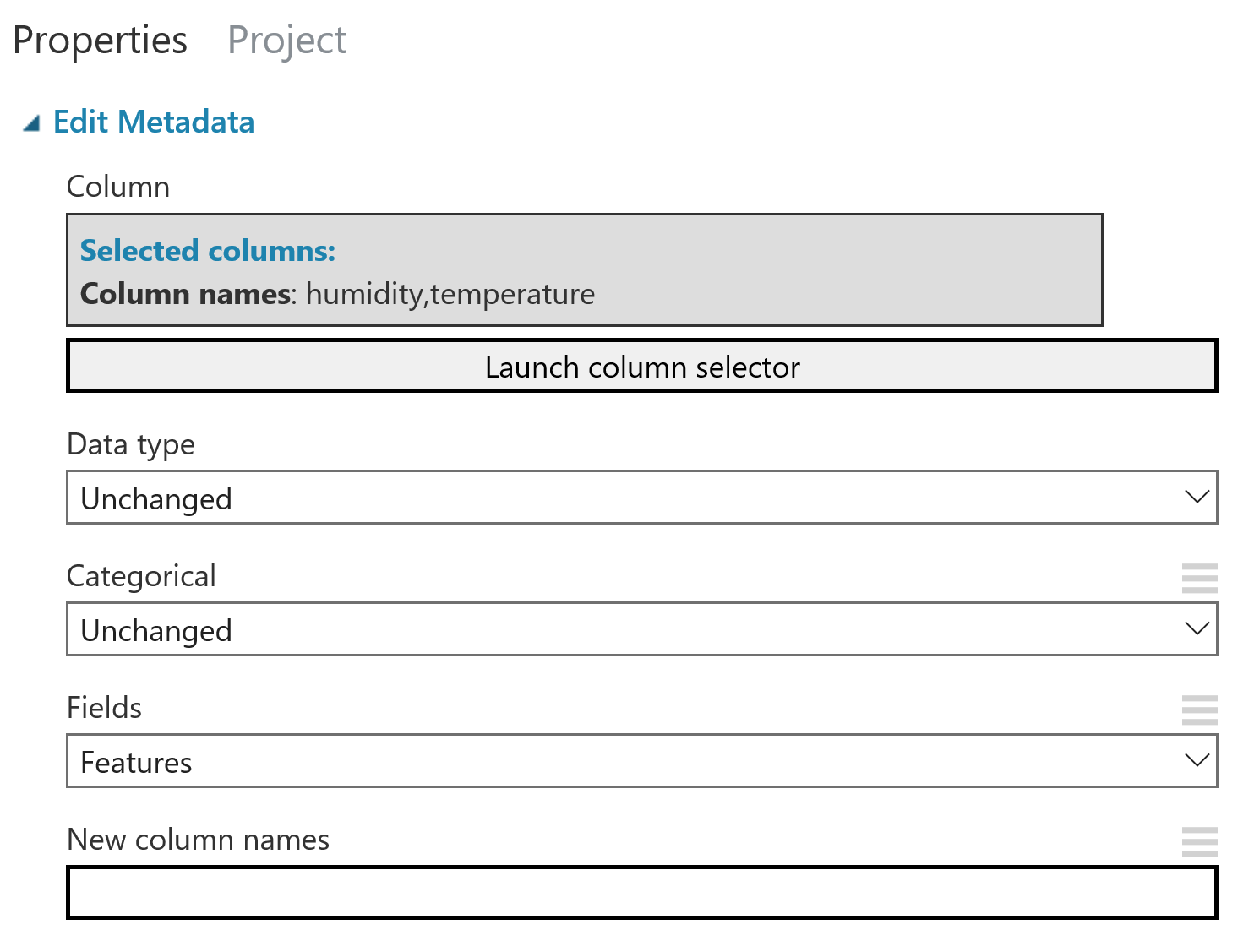




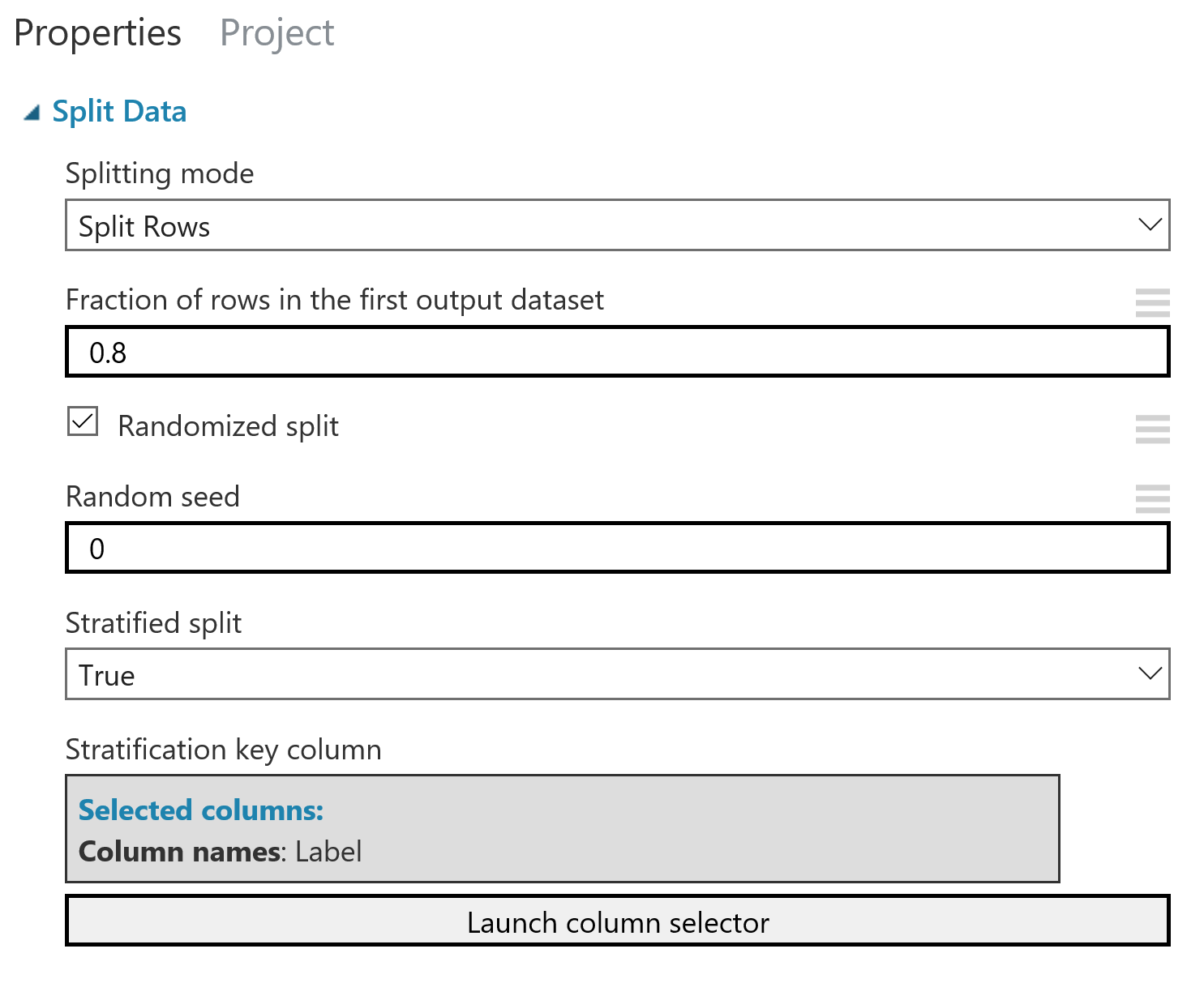
Edit the metadata so that the system recognizes the *train* column as the telemetry’s labels and rename the column to *Label*:



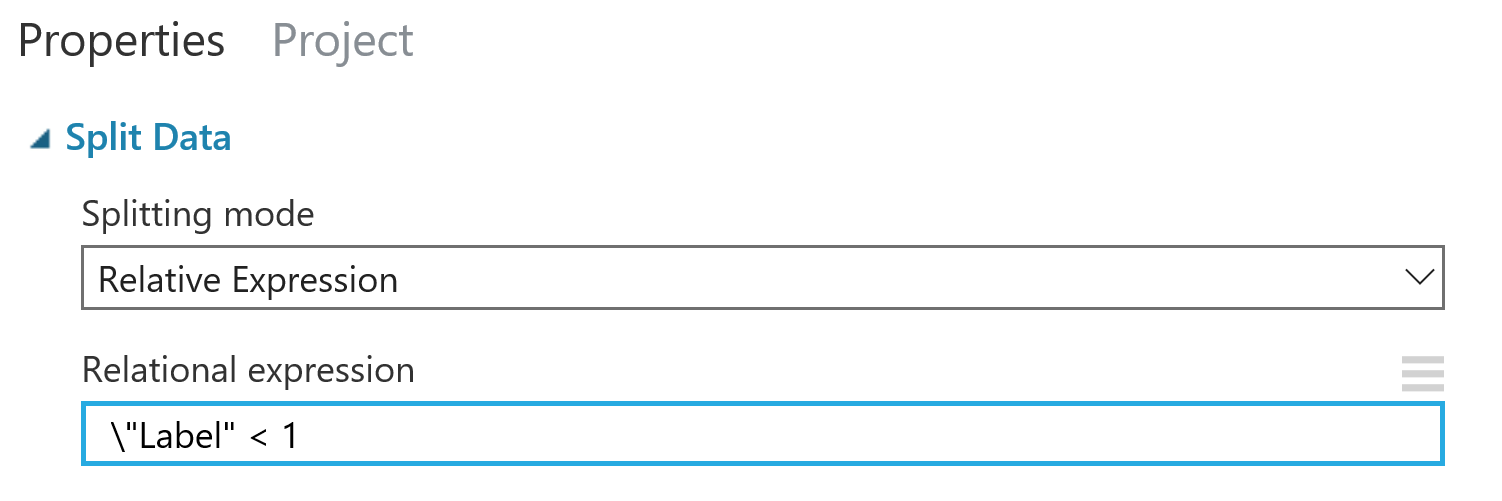
Edit the data again to mark *temperature* and *humidity* as the feature columns:



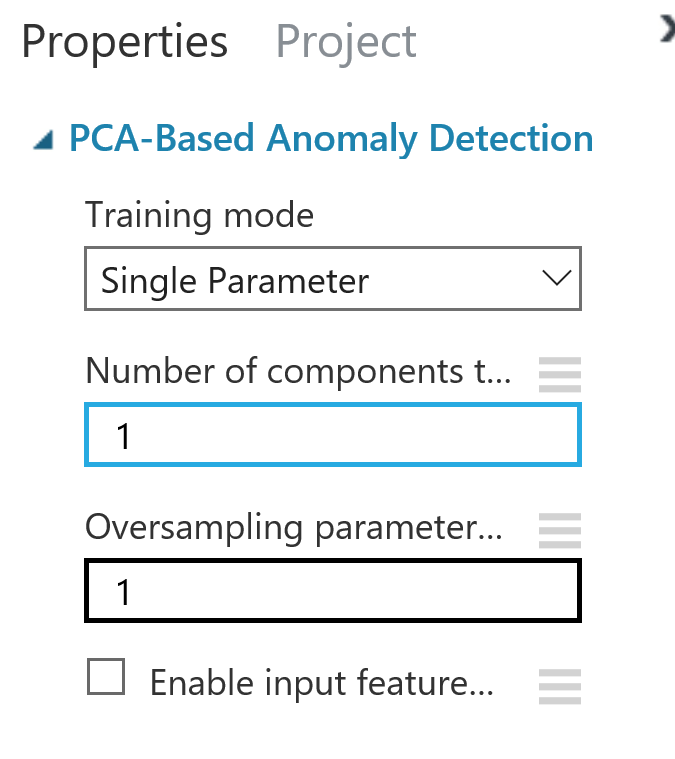
Split the data into 80% training and 20% testing:



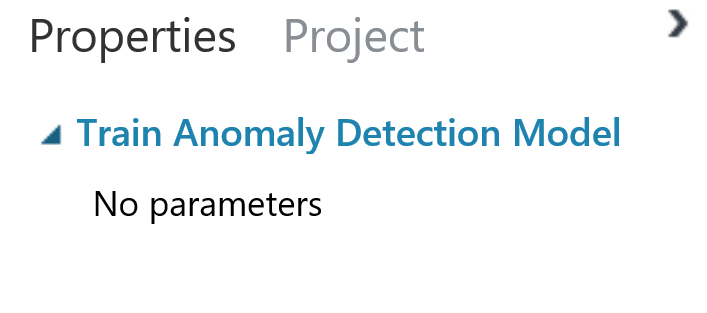
Then, from the training set, split the data again based on whether it is anomalous or normal:



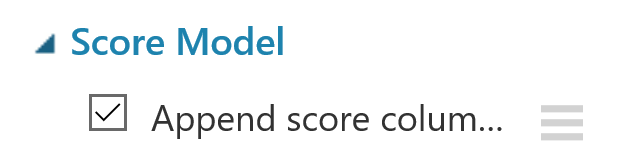
Create a PCA-based anomaly detection model:



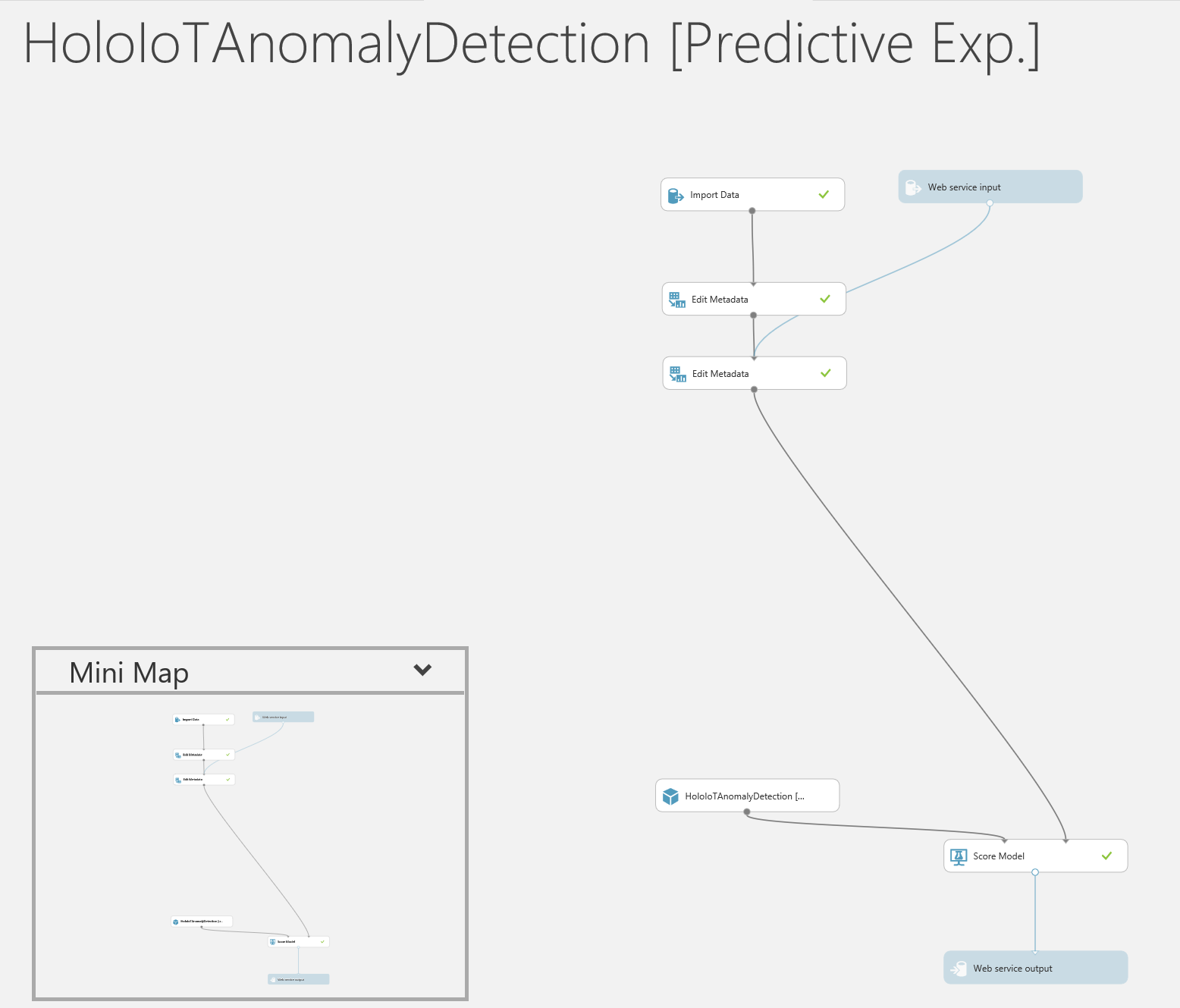
Train the anomaly detection model on the telemetry from the training set that’s labeled normal:



Finally, I score the model:



Run the model and look at the output of the scored model. If the model had a high accuracy of determining whether data was anomalous or not, create a predictive experiment from the training experiment:



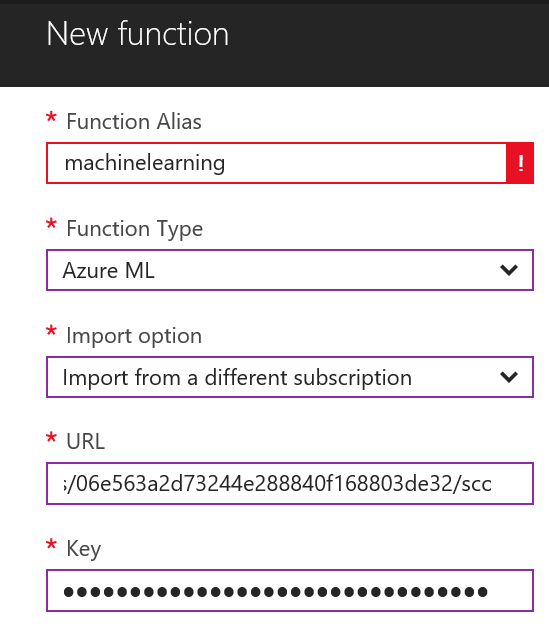
Deploy the web service (following the walkthrough) as a classic web service.

#### Azure Stream Analytics: Integrating Azure ML

To integrate the machine learning model with stream analytics, first download *Excel 2010 or earlier workbook* from the Default Endpoint section of the web service dashboard:

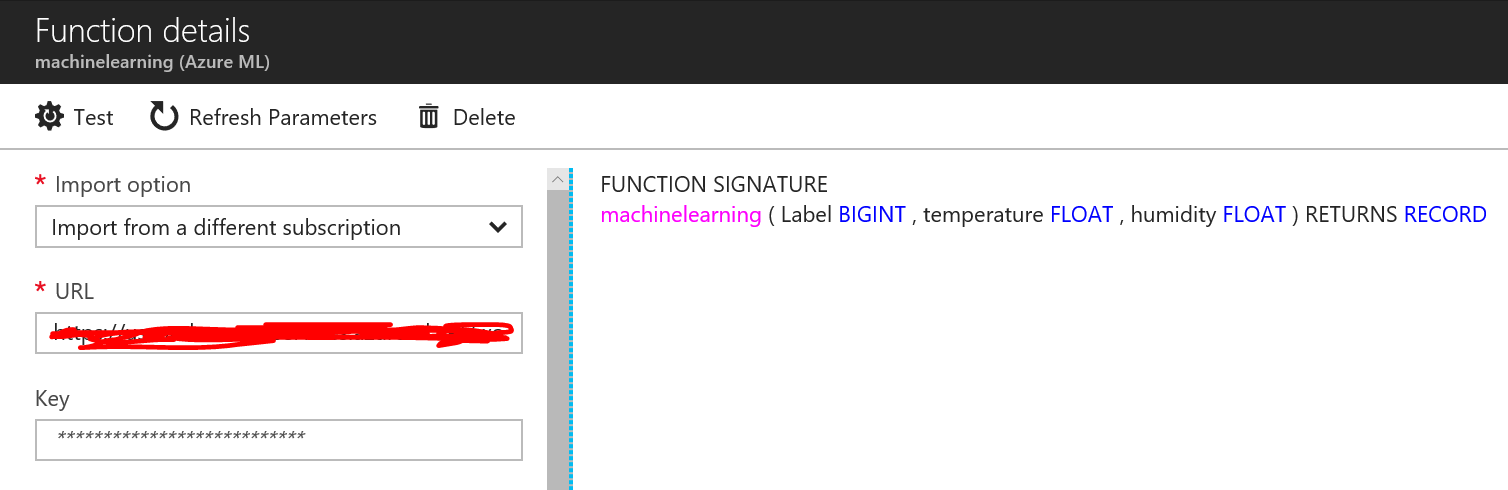


From this file, take note of the web service URL and access key. Then, in the Functions section of the stream analytics job, add a ML function using the URL and access key:

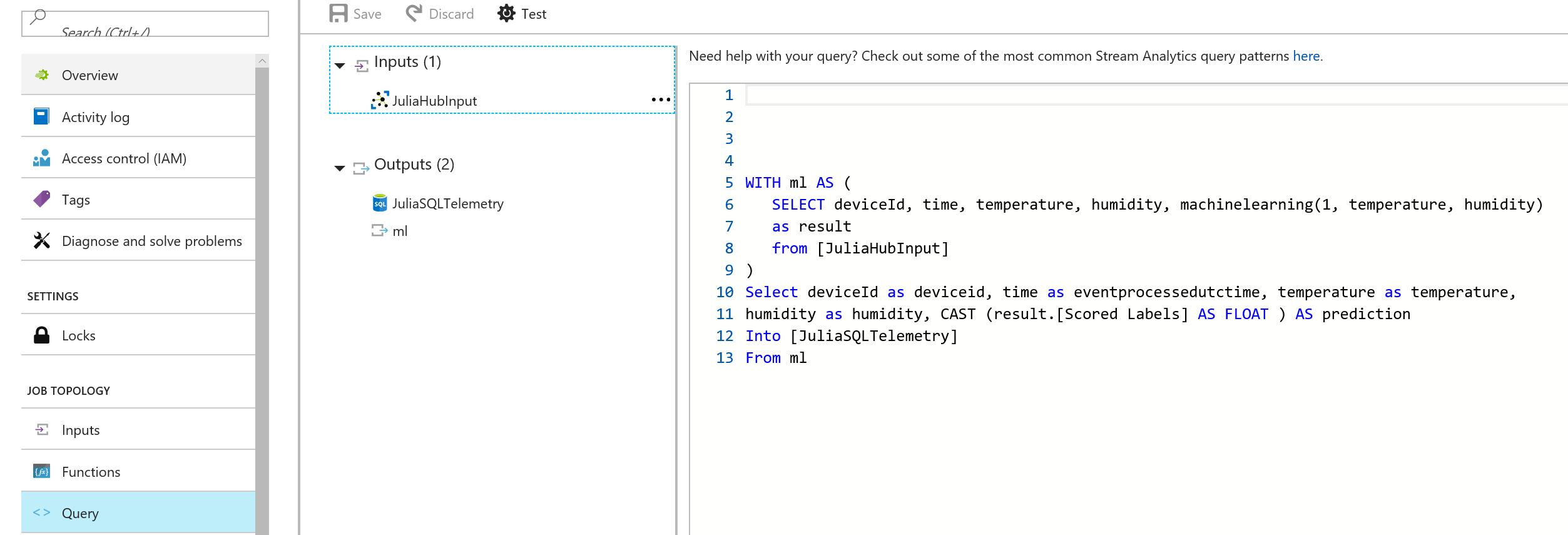




The function details should look like this after its created:



Lastly, edit the stream analytics query as follows:

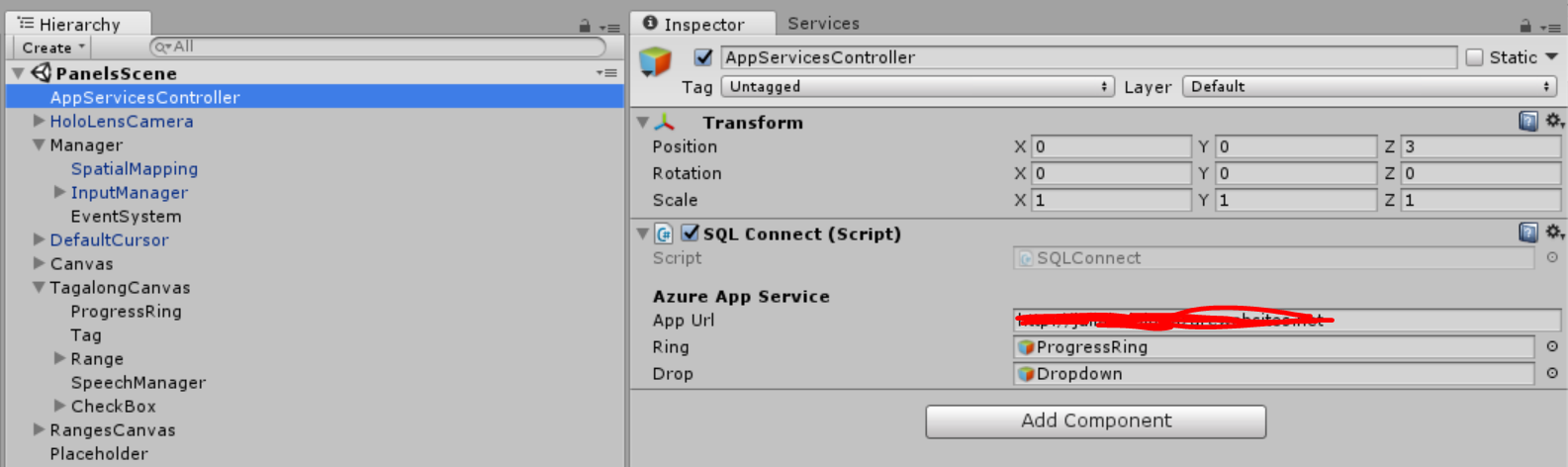


Note that the machinelearning function takes a label as its first argument. Since the model only evaluates the feature columns and not the label column, it’s fine to use any number as a placeholder. The *prediction* column is populated with 0 or 1, depending on whether the model classified the data as an anomaly.

Now, running the streaming analytics job and the simulated devices should generate simulated telemetry, evaluate the data for whether an anomaly is present, and store this information in the Telemetry table.

#### Unity

The last setup step allows the HoloLens app to connect to the Azure App Services mobile app. To do this, in Unity, open PanelsScene.unity. In the Unity Hierarchy, click on the AppServicesController gameobject. Place the Azure App Services URL in the input field:



### Scripts

All scripts that I wrote are found in the Assets/Scripts folder.

#### Summary of Scripts:

* **Anomaly**: contains a single field for the device ID. When the system queries for anomalous devices, each query result becomes an instance of this class. Not attached to any game object.
* **CameraWork**: handles the data that the chart displays. Attached to Graph in Canvas/Graph in the Unity Hierarchy.
* **HighlightButton**: changes the material on a button so that the button appears highlighted when the cursor is over it. Attached to every game object with a button component.
* **InstantiateTables**: Used to instantiate or destroy a table. Also changes the flag value in PopulateTable. Attached to the Table game object in Canvas/Table in the Unity Hierarchy.
* **MicrophoneManager**: handles dictation from the user. Not attached to any game objects.
* **MoveToView**: moves the canvas containing the charts and tables so that these displays appear a few feet above where a device is identified. Attached to the Canvas game object.
* **OpenDropdown**: after anomaly detection is ran, this fills the dropdown list with the device IDs of anomalous devices. It is attached to the Dropdown game object that is in TagalongCanvas/CheckBox/Anomalies/Dropdown in the Unity Hierarchy.
* **Placeholder**: handles the QR code scanning and text recognition. Attached to the Placeholder game object.
* **PointerHold**: increments the number on the ranges screen during tap mode when a user uses the tap and hold gesture. Attached to all game objects used to display numbers on screen in range mode.
* **PopulateLabel**: used to populate the graph header and the minimum and maximum labels on the x and y axes. Attached to the Labels game object in Canvas/Graph/Labels in the Unity Hierarchy.
* **PopulateLegend**: used to populate the legend during all mode.
* **PopulateTable**: populates the table values when the values are available. Attached to the NewTable prefab.
* **ProgressRing**: activates the progress ring when the app is querying the database, controls the progress ring’s behavior, and deactivates it when the results from the query are in. Attached to ProgressRing game object in TagalongCanvas/ProgressRing in the Unity Hierarchy.
* **SendQuery**: handles the buttons clicked on the checkbox by updating the displays and calling the function that runs the queries. Attached to the CheckBox game object in TagalongCanvas/CheckBox
* **SQLConnect**: queries the database, creates arrays of telemetry to populate the graph, and calculates the statistics for the tables. Attached to the AppServicesController game object
* **Telemetry**: Contains fields for different properties from the telemetry. When the system queries for telemetry, each query result becomes an instance of this class. Not attached to any game object.
* **TextToSpeechControl**: handles the text to speech. Attached to the AudioManager game object in HoloLensCamera/AudioManager in the Unity Hierarchy.
* **TimeVals**: handles everything on the ranges screen including inputs from the number pad, tap mode, and dictation and checks that the specified time range is valid. Attached to the TimeRange game object in RangesCanvas/TimeRange in the Unity Hierarchy.
* **UpdateRange**: updates the range display on the main screen to show the correct time range. Attached to the Range game object in TagalongCanvas/Range in the Unity Hierarchy.
* **UpdateTag**: updates the header on the main screen to display the device ID. Attached to the Tag game object in TagalongCanvas/Tag in the Unity Hierarchy.

### Game Objects and Prefabs

#### Important Game Objects in the Unity Hierarchy

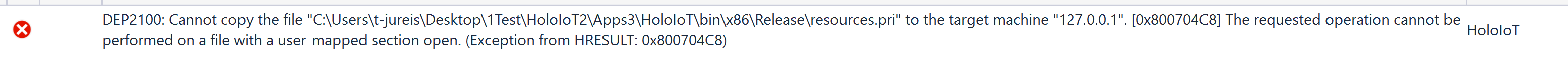
* **AppServicesController**: used to hold the SQLConnect script, which allows the database to be queried.
* **HoloLensCamera**: represents the camera for the HoloLens. Parent to GazeControls, which handles gaze, and AudioManager, which handles text to speech.
* **Manager**: holds managers for spatial mapping and user input.
* **DefaultCursor**: the cursor to use during the app. This cursor has different states for when the user’s finger is up to begin air-tap vs. when the users finger is holding the air-tap.
* **Canvas**: houses the displays. When an object is identified, the canvas moves to a point in space a few feet above where it’s identified and stays here until a new object is identified.
  + **Graph**: contains the line chart, chart labels, legend, and backdrop.
  + **Table**: houses the tables when they are instantiated.
  + **Text**: appears when there are no query results to alert the user.
* **Tagalong Canvas**: the main screen that is shown to the user when the app is open. Simple tagalong is used so that this always follows the user’s gaze.
  + **Progress Ring**: the progress ring that appears next to the header tag when the telemetry is being queried.
  + **Tag**: the header that contains the device ID.
  + **Range**: displays the time range and is a button used to navigate to the ranges screen.
  + **SpeechManager**: handles the voice commands that can be used while in the main screen.
    - “scan QR code” and “scan text” can be used to start object identification.
    - “temperature mode”, “humidity mode”, “all mode”, “graph mode”, “table mode”, and “live mode” are used to check options on the checklist.
    - “change range” closes the main screen and opens the ranges screen.
    - “show anomalies” runs anomaly detection to find anomalous devices.
  + **CheckBox**: a panel that allows the user to choose preferences in displaying the data, run anomaly detection, and identify devices
* **RangesCanvas**: the ranges screen that is shown when the user is changing the time range. Simple tagalong is used to that this always follows the user’s gaze.
  + Voice commands allowed in this mode:
    - “dictation mode” and “tap mode” check options on the checklist and change the mode by which a user inputs times.
    - “yesterday” and “today” change the time range to these respective dates.
    - “set range” sets the time range as the new range, closes the ranges screen, and opens the main screen.
  + **TimeRange**: contains the dates and times for the start and end times and the set range button.
  + **KeyBoard**: contains the keys for the number pad.
  + **Panel**: contains the panel with buttons and instructions for dictation mode.
* **Placeholder**: houses the Placeholder script used for object identification

#### Important Prefabs

* Prefabs I made are found in the Assets/Prefabs folder.
* **NewTable**: a prefab for the table used for single attributes
* **NewTable2**: a prefab for the tables used in all mode
* **TableMeasures**: a prefab that is used inside of NewTable and NewTable2 to hold the statistics.
* **CheckItem**: prefab for each item on the checklist.
* **LegendItem**: the prefab used to populate the legend for the graph during all mode.

### Known Issues

When deploying the app from Visual Studio, this error message occurs:





To fix it, rebuild the app in Visual Studio and redeploy it.

### Dependencies Used

* [AppServices](https://github.com/Unity3dAzure/AppServices)
* [QrCodes](https://github.com/mtaulty/QrCodes)
* [IPAddressCameraScanning](https://github.com/mtaulty/IPAddressCameraScanning)
* [HoloToolkit](https://github.com/Microsoft/MixedRealityToolkit-Unity)
* [MeshChart](https://www.assetstore.unity3d.com/en/#!/content/18196)

### Extensibility

Developers should be able to add or modify features of this application to meet the needs of their scenarios. Some of this extensibility includes methods of identifying IoT devices, altering data attributes, adding data displays, and changing the frequency of live mode querying. For situations where QR code scanning or text recognition aren’t plausible, it’s possible to integrate other forms of object identification into the application. Rather than using simulated devices, real IoT devices that output telemetry with different data attributes can be included. The app can be extended to include other data displays, such as past maintenance reports for a device, 3D models of devices, and instructions for maintenance. In a scenario where devices are measuring attributes less frequently than in the simulated devices, the app will not need to re-query the database as often. Steps to add these into the application are below.

#### Add Methods of Identifying IoT Devices

Different methods to identify device IDs can be added to the app. The following are examples of other methods that can be used:

* + Location – Identify device based on user location & orientation
* Can be used in a scenario where the devices are spread out. Would be difficult to implement in a scenario where devices are very close together or stacked on top of each other.
  + Spatial Anchor – Identify device based on Spatial anchor
* Can be used in a scenario where the number of devices is small and it’s plausible to place anchors on all devices.
  + Image Recognition – Identify device based on image
    - Can be used in a scenario where devices are different or have physical properties that are different from other devices.

A way to implement image recognition or spatial anchors could be by using Vuforia’s object recognition: <http://southworks.com/blog/2017/05/26/how-to-create-an-image-recognition-app-with-vuforia/>. The SDK can be used to identify VuMarks, flat images, and 3D objects: <https://library.vuforia.com/getting-started>. Currently in the HoloIot app, when a QR code is decoded or text is recognized, the app updates SQLConnect.ID to be the device ID. Once this is updated, the device ID appears in the header of the main screen and the system uses the device ID in its queries. Thus, when adding more ways of identifying devices, a developer must include a button or voice command in the app to allow the user to start the identification, implement the method of identification, and then simply update SQLConnect.ID with the recognized device ID. If the button is added in the Unity Hierarchy with TagalongCanvas/CheckBox/Image as its parent, modify SendQuery.cs so that the numIdentification variable equals the number of object identification buttons with Image as its parent. Also, make sure that in the Unity Hierarchy, the new button is after the Historic game object. Finally, voice input for the object identification can be added by adding to the list of existing commands on SpeechInputSource.cs and SpeechInputHandler.cs attached to TagalongCanvas/SpeechManager in the Unity Hierarchy.

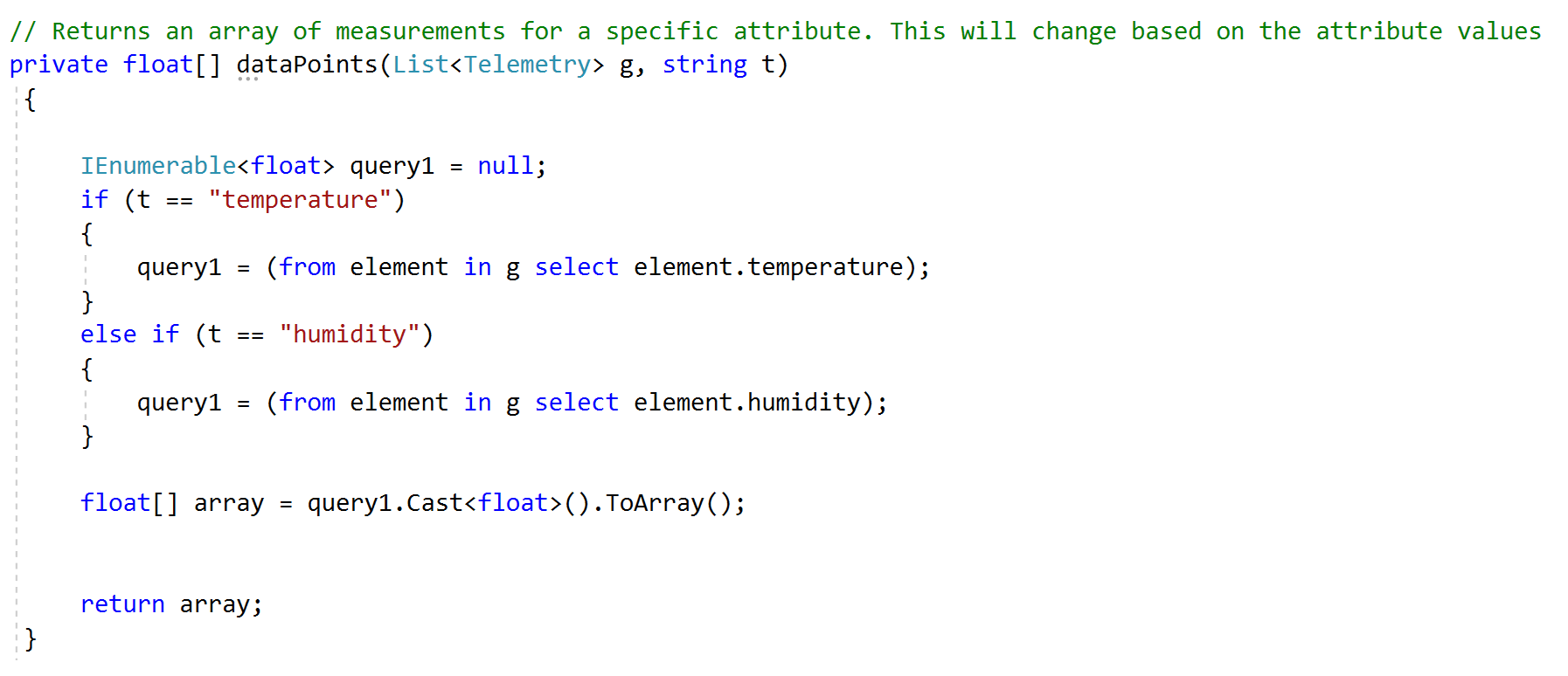
#### Adding/Changing IoT Data Attributes of Interest

When adding or changing the attributes, the developer only has to do a few steps. First, in the Unity Hierarchy under TagalongCanvas/CheckBox/Image, a CheckItem prefab should be added anywhere after the Attributes game object but before the All game object. The text of this game object should be changed to match the name of the attribute. Next, in Telemetry.cs, add a class field that is a public float. This field will hold the measurement for each entity in the database. Make sure the variable name matches the column name in the database. In SQLConnect.cs, change the initialization of attributes from:



to: public static string[] attributes = new string[2] { "temperature", "humidity", "[name of new attribute]"};

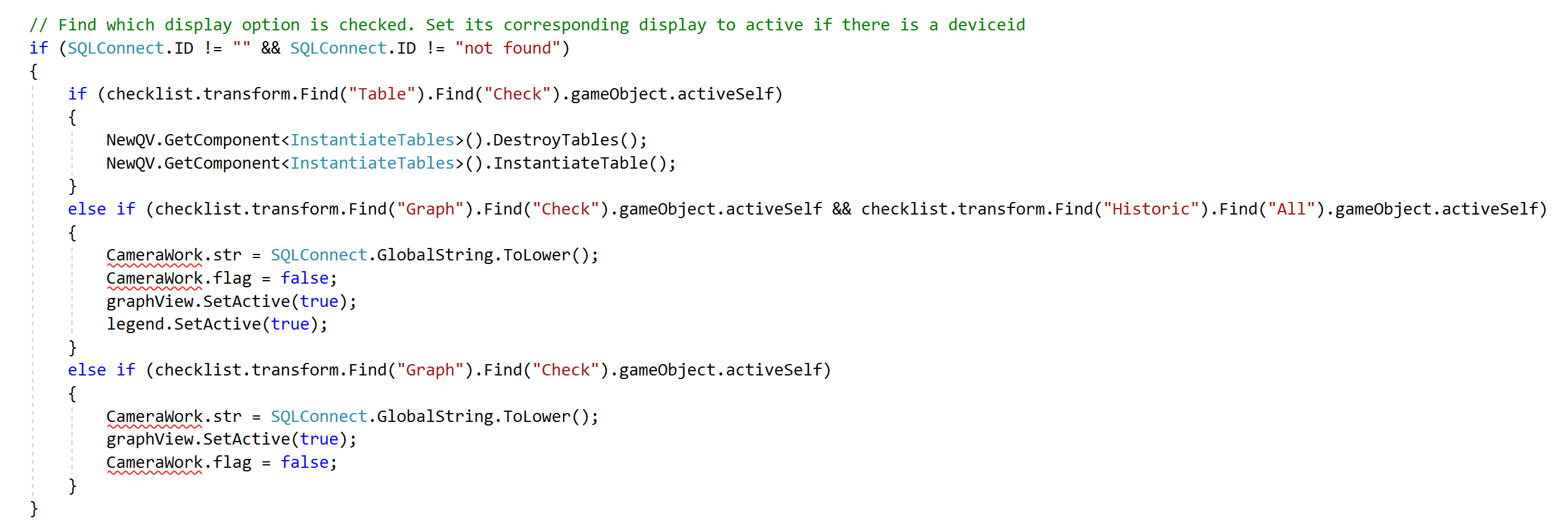
Make sure the order of the elements in attributes matches the order of the elements in the checklist in the Unity Hierarchy. Change the dataPoints function:



Add a condition for the new attribute’s name. Finally, voice input for the attribute buttons can be added by adding to the list of existing commands on SpeechInputSource.cs and SpeechInputHandler.cs attached to TagalongCanvas/SpeechManager in the Unity Hierarchy.

#### Adding/Changing Methods for IoT Data Display

To add a new type of display, add a button corresponding to the display on the checklist. Place a copy of the CheckItem prefab in the Unity Hierarchy with TagalongCanvas/CheckBox/Image as its parent. Make sure the game object is after the Display game object but before the Historic game object in the hierarchy. Change the name of the game object to the desired name and change its text to the display name. In SendQuery.cs, change numDisplays so that it’s equal to the number of display buttons that have TagalongCanvas/CheckBox/Image as their parents. In SendQuery’s Update function, the script checks which display is active and updates the UI appropriately:



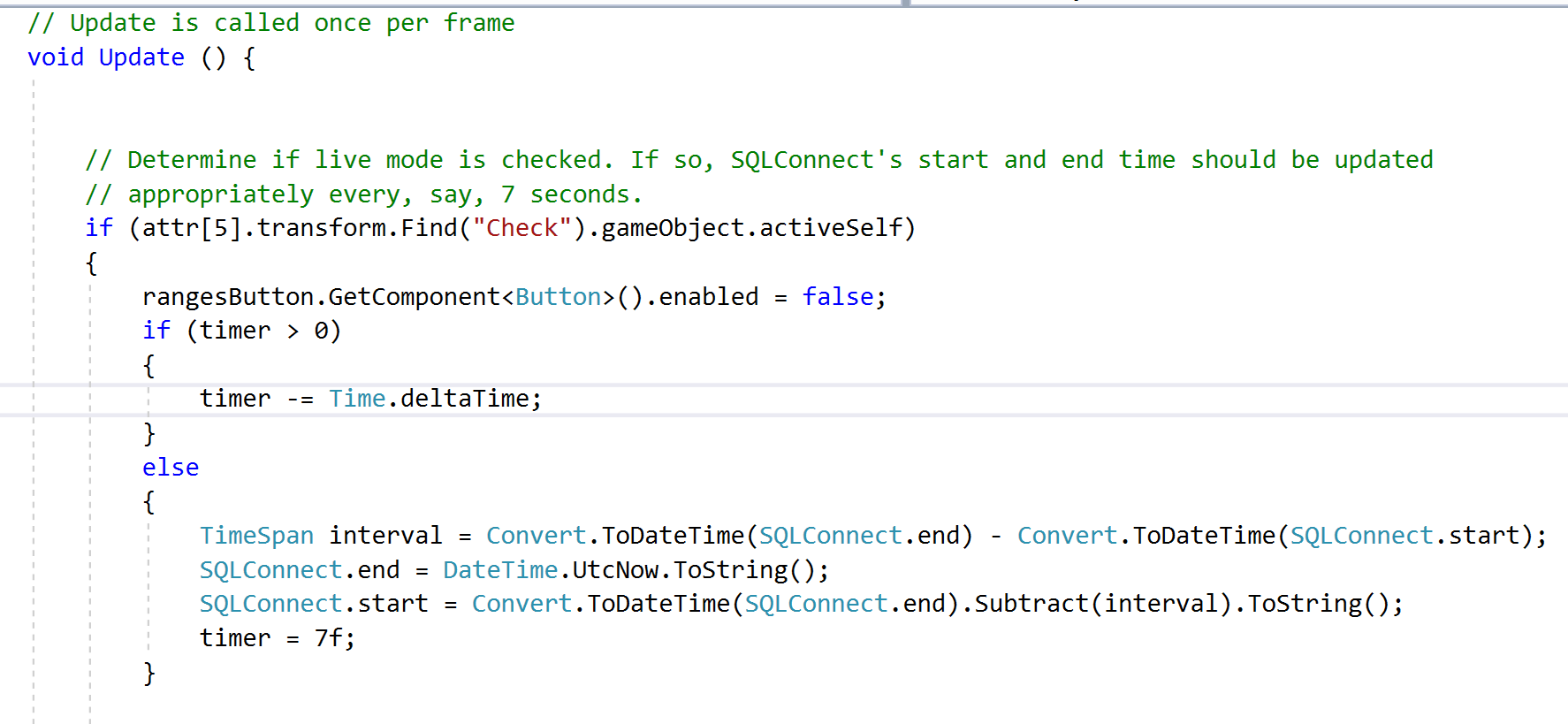
If a new display button is added, it should be handled in the above block. Finally, voice input for the display buttons can be added by adding to the list of existing commands on SpeechInputSource.cs and SpeechInputHandler.cs attached to TagalongCanvas/SpeechManager in the Unity Hierarchy.

#### Other Changes to the Checklist

With the way the code is written, it is assumed that the checklist is split into groups in the Unity Hierarchy with the attributes first, then the displays, the Historic game object, and finally the object identification buttons. When adding to the checklist, be sure to change values in SendQuery.cs.

#### Change Frequency of Querying for Live Mode

In the app, live mode is handled in SendQuery.cs. Since the simulated devices send telemetry to the database every 5 seconds, live mode re-queries the database every 7 seconds, as shown below:



timer is a float, and the time delta each frame is subtracted from timer until timer reaches 0, in which case SQLConnect’s start and end are updated, causing the system to re-query. In the Start method of SendQuery.cs, timer is initialized to be 2f; however, after the start of the app, timer is always reset to be 7f after reaching 0. To change the interval of re-querying, timer should be reset to another float, with the float representing the time in seconds between queries.