I How it all works

I.1 Goals of the work

A company makes *controllers*[1] for use in factories and plants. These controllers have a touch screen.

- An engineer at the controller can view the state of the machine that it governs.
- With the screen, they can also change the state of controller. Say, they can make it so that the machine is ordered to act a certain way.

Somebody from the company asked for a way to take what is seen and done on the screen and put it on the Web. That way, engineers who are not at the plant could still do part of their job.

- They could get a web page and see what they would see if they were really at the controller.
- They could also make the same kinds of changes that they might make as if they were really at the controller, and have the changes go through to the controller at the plant.

Later on, they asked for a way for engineers to work with controllers that were not yet in a plant.

- An engineer could get a page from the Web on their own machine. On the page, they would see what they might see for a real controller started up for the first time.
- They could make changes to this 'fake' controller just as though it were a real one.
- When they finish the changes, they could save the new state of the controller in a file and have that file remain even after the page had gone.
- Later, a second engineer could take a copy of such a file and load the controller state with the changes the first engineer set.

About the controller

When an engineer is looks at or touches the screen of a controller, the whole of what the controller and engineer do together is called the **human-machine interface** (HMI) [2]. At first, the HMI shows only some of the state of the controller's machine. Through the use of buttons and keys on the touch screen, the engineer can get to other pages to learn more about the machine's state or change the rules that the machine should follow.

One program, the **user interface** (UI), draws most of what the engineer sees on the screen. When it starts, the UI does not know about the pages it must draw on the screen, so it must ask a *PostgreSQL server* that also runs on the controller. This PostgreSQL server is the host to the **UI configuration database**, which keeps records of all the pages that the UI needs to draw. The UI asks the PostgreSQL server for these records.

---

[1] A *controller* is a tool that shows how some machine in a factory or plant behaves and helps set rules for how it should behave. A controller for a boiler, say, might

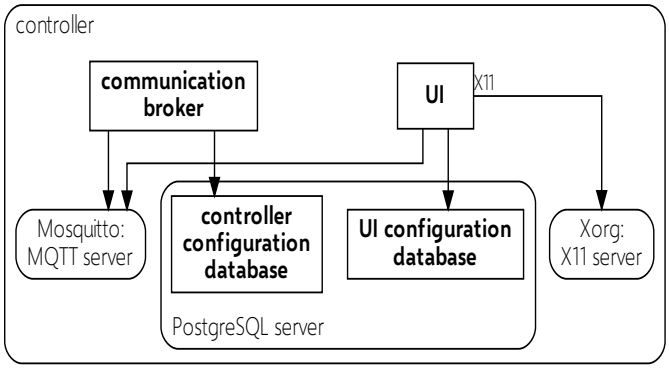
- show an engineer how hot the water is,
- let the engineer set how hot or cold the water is allowed to be, and
- sound an alarm if the water gets too hot or too cold.

[2] The company's team also uses *human-machine interface* (HMI) for what we might call the user interface (UI).

When it gets the records, it figures out orders to draw the pages. Later, when it is time to draw a page, the UI sends messages with these orders to an *X11 server* on the controller. The X11 server works with drivers and other controller parts to draw on the touch screen.

The UI does not get or change the machine state on its own. It works with several other programs on the controller to do its job. These programs talk to one another through *MQTT server* that runs on the controller.[3] They send messages in JSON form on some MQTT topic, and receive such messages on other MQTT topics. One program, the **communication broker**, routes messages from some MQTT topics to others when there is no simpler way for the messages to get where they need to go.

The PostgreSQL server is also host to the **controller configuration database**: it keeps records of, say, what alarms should be raised, which parts of the machine state should be shown and which can be changed, who can make changes, and so on. To look at or make changes to these records, the UI sends a message to the communication broker through the MQTT server. The communication broker receives that message, asks PostgreSQL server to do something based on what is in that message, and pushes messages back to the MQTT server for the UI to receive.



The programs on the controller that matter for our goals The arrows are drawn from clients to servers.

Controller keys

There are a few values needed to choose a controller:

- The **controller ID** is a GUID that programs on the controller make up when it is first started or when it is wiped clean.
- The **application ID** is a GUID given to each sort of machine that a controller can govern.
- The **device ID** is the name by which the Azure IoT Hub knows the controller.

The programs on the controller know the controller ID and application ID. They do not know the device ID.

[3] MQTT is a way to send and receive messages through TCP/IP. In MQTT, messages have a string value called the *topic*. When clients connect to a server, they ask to receive messages for some topics. When a client sends a message with a certain topic to the server, all clients who asked for that topic receive a copy of the message. The way that clients send or receive messages through a server described here is called the *publish-subscribe pattern*.

Outline

- Operating system: GNU/Linux
- Windowing system: the display server Xorg for the X Window System (X11).
- Communication broker
  - Process virtual machine (Process VM): the Mono Runtime for the Common Language Infrastructure.
- User interface
  - Process virtual machine (Process VM): the Mono Runtime for the Common Language Infrastructure.
  - Widget toolkit: GTK+ 3, through the library GtkSharp 3[4].
- PostgreSQL server
  - PostgreSQL 10
- MQTT server
  - [Eclipse Mosquitto][def-mosquitto]

What others did

A controller metadata database

Before this work, somebody else used the Azure SQL Database to host the **controller metadata database**, the place to keep records of controller keys (such as the controller ID, application ID, and device ID).

They set up the **metadata database API**, a ReST API application, for other services to get and set records in the controller metadata database.

A backup database

Before this work, somebody set up a way to copy records from the controller configuration database (the **target database**) to some place away from the controller, the **backup database**. The backup database is of same form as the **target database**, but its host is a PostgreSQL server on the internet. The records in the backup database are useful in case the real controller breaks and a new one is needed. In this flow,

1. when a program on the controller posts a message with the changes to a certain MQTT topic,
2. another program, the **agent**, receives the message and sends it over the internet to an Azure IoT Hub[5].

---

[4] At earlier stages of the work, the company's team used the Xamarin.Forms library to make the HMI. This library, in turn, made use of GTK+ 2 through GtkSharp 2, a library that is given out as part of the Mono Project. The team moved away from these libraries because of leaks in `Xamarin.Forms` that nobody could patch up in time.

[5] These messages are through AMQP. *AMQP*, like the MQTT protocol, is a way to send and receive messages through TCP/IP. An AMQP message takes up more space than the MQTT message, but AMQP hosts have more power. They can

- store messages in the order they got them,

3. The Azure IoT Hub routes the message to an Azure Event Hub.

4. When messages are received at the Azure Event Hub, the **configuration endpoint**, a *serverless function*, starts a run.

    1. Using the message it received, the configuration endpoint makes a new message to ask that the backup database be changed.

    2. It sends the message it in an HTTP request to a ReST API application, the **configuration database API**.

        1. Using the message it received, the configuration database API asks the PostgreSQL server to change the backup database.

        2. The configuration database API answers the configuration endpoint with a message about if the change went through.

    3. When the configuration endpoint gets that answer message, it asks the metadata database API for the device ID of the controller.

    4. Finally, it asks the Azure IoT Hub to send that message back to the controller with that device ID.

5. When the answer message gets back to the controller, the agent posts it to a certain MQTT topic.

I.2 Changes that reach the goals

To get an engineer to see the UI from a web page

We made a Docker image[6] from some of the controller's programs. This **remote HMI** borrows

- the UI;
- communication broker;
- MQTT server; and
- PostgreSQL server, which is host to

- use message tags to route messages to other AMQP hosts, and

- use *SASL* (a well known way that two hosts that connect to one another through TCP/IP can make sure that each is who it says it is).

In the cloud platform *Microsoft Azure*, certain services are really just AMQP 1.0 hosts. SASL PLAIN for clients (what in *Microsoft Azure* is called a an *authorization rule*).

- *Azure Event Hub*

- *Azure Service Bus*

- *Azure IoT Hub*

In Microsoft Azure, other services, such as *Azure Event Grid*, can turn messages in other form to AMQP.

[6] *Docker* is a tool to run programs in their own filesystem and with operating-system virtualization. A *Docker image* is a copy of this filesystem together with orders for how to start the program. A named and tagged Docker image is a *Docker repository*. A file or place on the web that keeps copies of Docker repositories is a *Docker registry*. Finally, a *Docker container* is a Docker image that is run.

- the UI configuration database and
- an empty controller configuration database (the **container database**).

The UI needs a X11 server to draw anything. However, the remote HMI does not use Xorg. Instead, a different X11 server, *Xpra*, runs from a different Docker image, the **X11 bridge**. Xpra is an X11 server that is also an HTTP server. It can send an HTML5 web page to an HTTP client, and with the X11 messages from the UI draw on that web page with HTML5 canvas. This way, an engineer can see the UI on a web page.

There are some values (such as the controller ID and application ID) that the programs from the controller expect to already be set and that they cannot set on their own. To set these values, another program in the remote HMI, the **reverse proxy**, passes along HTTP between the HTTP client and the Xpra, making changes to the state of the remote HMI as needed.

For a real controller

So that an engineer can get and change controller state through the remote HMI (and not just see an empty UI page), the communication broker for the remote HMI behaves differently from the one on the controller:

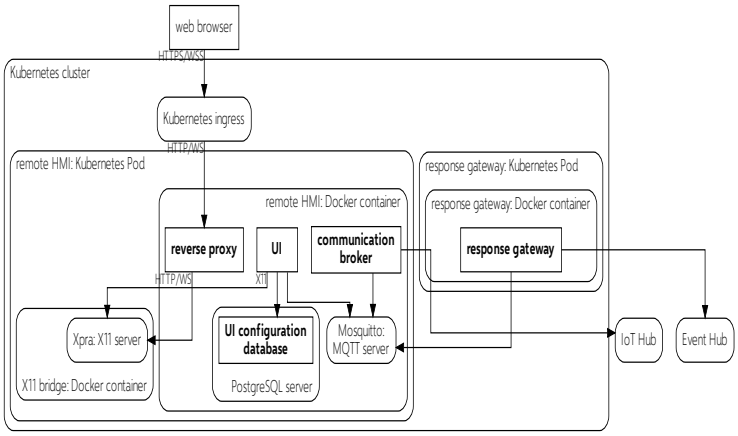
- Before the communication broker and UI of the remote HMI start, the reverse proxy sets the controller ID and application ID of the remote HMI. The reverse proxy then starts the communication broker and UI.
- To *get* the state of a controller, the communication broker for the remote HMI asks the backup database for the state whenever the UI for the remote HMI asks for it.

  When the communication broker for the remote HMI tries to get the state of a real controller, it does not talk to the container database or target database.
- Whenever the remote HMI UI asks to *change* the state of a controller through a JSON message, the communication broker for the remote HMI asks the Azure IoT Hub to send that message to the controller (through AMQP). The agent posts the message to an MQTT topic, and the controller communication broker receives it and does what it would do for a message from the controller UI.

  When the communication broker for the remote HMI tries to change the state of a real controller, it does not talk to the container database, backup database, or target database.

  - To know how to ask the Azure IoT Hub to send a message to the controller, the communication broker for the remote HMI needs to know the device ID. However, when this communication broker starts up, it only has the controller ID. To get the device ID, it asks the metadata database API.
  - Both the remote HMI UI and the controller UI need to get a message back that says if the change went through. To get this message back,

    1. The configuration endpoint sends the message to an Azure Event Grid.
    2. The Azure Event Grid routes it to another Azure Event Hub.
    3. A new Docker container, the **response gateway**, gets messages from the Azure Event Hub and posts them to the right MQTT topic on the remote HMI MQTT server.

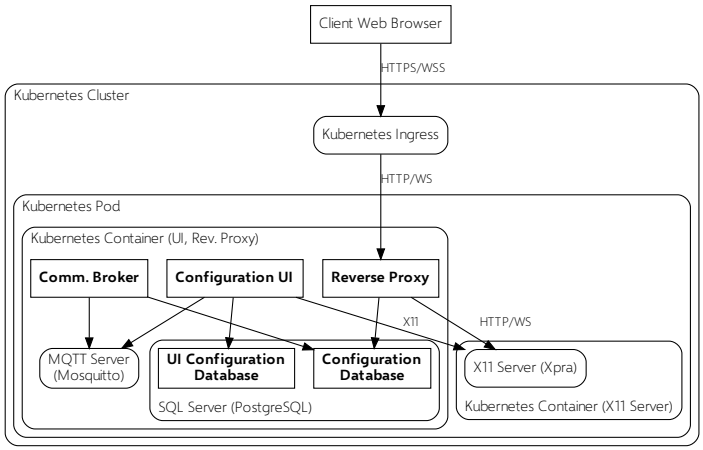


The parts of the remote HMI in use for a real controller. The arrows are drawn from clients to servers.

For a new controller

- Before the communication broker and UI of the remote HMI start, the reverse proxy empties the container database and adds new records as there would be for a real new controller, controller ID, and application ID.

- To get or change the state, the communication broker for the remote HMI asks the container database. It does not to talk to the backup database, say.



The parts of the remote HMI in use for a new controller. The arrows are drawn from clients to servers.

Other stuff needed for the big stuff to work

Making the remote HMI work for many clients

So that more than one HTTP client can use the remote HMI, a Kubernetes cluster is host to several remote HMI Docker containers.

HTTP clients reach a remote HMI Docker container through another reverse proxy called the **Kubernetes ingress**. The Kubernetes ingress has orders to not send the HTTP client to a different remote HMI once it has reached one and gotten a good message back (this way of setting routes, **sticky sessions**, works through the use of an HTTP cookie) .

Making sure only allowed people can see controller state or make changes

Before anything else happens in the remote HMI, the reverse proxy in the remote HMI looks for an HTTP cookie. If the HTTP client does not have the right one, the remote HMI sends them off to the Microsoft Identity Platform.

If they do have the right one, the reverse proxy uses the HTTP cookie to ask Microsoft Graph for the name and group. It sets these values so that the UI can use them when it needs to decide if it should show a given page.

Making sure a remote HMI Docker container is locked to one client

So that no other HTTP client can reach a Docker container already in use, the reverse proxy gives any new HTTP client an HTTP cookie and goes into a locked state: only with HTTP request with that HTTP cookie can get a good HTTP response back. There is a time out, but only after no HTTP client has come to that remote HMI Docker container in a while.

Making a choice of controller

The HTTP client passes the choice of controller (real or new) through the HTTP query string.

- If the HTTP query string has **`target-state=unprovisioned`**, the reverse proxy sets up the remote HMI for a new controller.
- If the HTTP query string has **`target-state=provisioned`**, the reverse proxy sets up the remote HMI for a real controller.
  - The HTTP query string must have parts
    - **`controller-id=<guid>`** and
    - **`application-id=<guid>`**.

    The reverse proxy sets the controller ID and application ID to the values so passed.

The reverse proxy tells the communication broker about these values through a JSON file.

I.3 Parts of the work

The reverse proxy

The reverse proxy is an HTTP server in the Docker container. The process VM of the reverse proxy is .NET Core 3.1, and the HTTP server libraries are those of ASP.NET Core 3.1.

The reverse proxy has four jobs. Each job is the charge of a single library: for each HTTP request that comes in, the libraries act one after another [7].

1. If someone has already started to use the Docker container to which the reverse proxy belongs, the **session middleware** library makes sure that nobody else can use that Docker container at the same time.

[7] That is, they follow the *pipeline pattern*.

2. The **auth. middleware** library
    1. It makes sure that anyone who tries to use that Docker container has checked in with the Microsoft Identity Platform (MIP)[8] first so that it can learn who they are. It adds their name to the UI settings.
    2. Once it knows that, it asks Microsoft Graph[9] if they belong to one several groups and adds the answer to the UI settings. The UI uses the that setting to decides what buttons and pages about the controller to show.
3. The **provisioning middleware** library sets up the Docker container to act like the controller.
    1. It takes the choice of controller that the HTTP client passed and writes it in a JSON file for the communication broker to read.
    2. It starts the communication broker and UI.
4. If earlier steps finished without trouble, the **proxy middleware** library passes the HTTP request from the HTTP client through to Xpra, and passes the HTTP response from Xpra back to the HTTP client.

The files for each library are listed in turn under

1. `$(ROOT)/src/Proxies/Middleware/Session`
2. `$(ROOT)/src/Proxies/Middleware/Auth`
3. `$(ROOT)/src/Proxies/Middleware/Provisioning`
4. `$(ROOT)/src/Proxies/Middleware/ProxyKitProxy`,

the files for reverse proxy proper under `$(ROOT)/src/Proxies/ProvisioningReverseProxy`.

The parts of the reverse proxy are tied together through use of the *dependency injection pattern*:

- A *service* is any part charged with a single sort of task.
- One part of the program, the *provider*, knows about all of the services needed for the program to work.
- If a service needs other services to work, the provider makes sure that all those other services were added to the heap first before the `new` service.

The session middleware

The session middleware library is made up of two services:

- The **client session key service**
    - checks if the HTTP request from the HTTP client has an HTTP cookie with a certain shared secret (the **session key**),
    - gets the session key if it is there, and
    - sets the key in messages going back to the HTTP client.
- The **lock service** keeps track of the session key for a certain amount of time, then forgets it. It
    - 'locks' the reverse proxy to a given session key value for that amount of time,

---

[8] The *Microsoft Identity Platform* is a Web service that other services can use to make sure that someone is who they say they are. It serves a form of *OpenID Connect*, which is the most often used way for Web services agree to talk to one another for this job. It checks who the person is against records in *Azure Active Directory*, a Web service that businesses use to keep lists of people and some simple facts about them (such as their name, their title, or the groups they belong to).

[9] *Microsoft Graph* is a Web service that other services can use to get and set files and records that Microsoft 365 looks after. In turn, *Microsoft 365* is a set of office tools and Web services that businesses use to make, keep track of, and share records (taken to mean written pieces, slides, tables of account, plans, drawings, mail, and lists of people and addresses). Azure Active Directory is sold as part of Microsoft 365.

  - checks if the reverse proxy is in a locked state,
  - checks if a session key is equal to the one it keeps track of, and
  - sets the time left before the lock ends back to the value it first had.

When the new HTTP request comes in, the session middleware asks the lock service to check if the reverse proxy is locked.

- If so, it asks the client session key service if the message has a session key.
  - If so, it
    1. asks the client session key service for the value of the key and
    2. asks the lock service if that value is equal to the stored value.
       - If so, it passes the HTTP request to the next part of the reverse proxy;
       - otherwise, it sends an HTTP response 503 back to the HTTP client.
  - Otherwise, it sends an HTTP response 503 back to the HTTP client.
- If not, it
  1. makes up a new session key,
  2. asks the lock service to lock with that session key,
  3. asks the client session key service to set the session key, then
  4. passes the HTTP request to the next part of the reverse proxy.

Values to set

The auth. middleware

The job of the auth. middleware is to

- make sure the person behind the HTTP request is who they say they are and, once that is done,
- get the name and group of that person to the UI.

To do the first part, it wraps the **`Microsoft.Identity.Web`**[10] library with extra set-up needed to make it work for the reverse proxy.

Two services together do the second part:

- The **remote system user repository** gets the name and group of the person by making an HTTP request to Microsoft Graph.
- The **system user service** sets the name and group of the person in the set-up for the UI. What it really does is set an environment variable for each:
  - `$UILOADER_AUTHENTICATEUSER__NAME` holds the name and
  - `$UILOADER_AUTHENTICATEDUSER__ROLE` holds the group.

Values to set

Each **`HostEnvironment`**—**`Local`**, **`Docker`**, **`Kubernetes`**—has its own needs for these values.

---

[10] The *Microsoft Authentication Library* (MSAL) is a library that service programs can use to talk to the MIP without the need to set up their own HTTP client. The library **`Microsoft.Identity.Web`** packages MSAL in a way that makes it easy for ASP.NET Core programs to use.

AZUREAD.CLIENTID
The value of `.appId` in the `Microsoft.DirectoryServices.Application` for the reverse proxy.

AZUREAD.CLIENTSECRET
One of the `base64` strings from `.passwordCredentials | map(.value)` in the reverse proxy `Microsoft.DirectoryServices.Application`.

AZUREAD.DOMAIN
The host name that stands for the Azure Active Directory tenant.

Example: `contoso.microsoftonline.com`

AZUREAD.INSTANCE
The string `https://login.microsoftonline.com/`.

AZUREAD.TENANTID
The GUID that stands for the Azure Active Directory tenant.

AZUREAD.CALLBACKPATH
Where the reverse proxy gets back information about the person who made the HTTP request. Should always be `/signin-oidc`.

Also:

```
{
  "AzureADOpenID": {
    /* What AAD needs to allow the reverse proxy to do in Microsoft Graph to look up the
group. */
    "Scope": [
      "User.Read",
      "GroupMember.Read.All"
    ]
  }
  "Graph": {
    /* Groups that set what the person who uses the remote HMI can do. */
    "Groups": [
      {
        "GroupId": "64d3af98-d1ec-4442-a189-e09224e886a9",
        "Index": 2,
        "Name": "Administrator"
      },
      {
        "GroupId": "64d3af98-d1ec-4442-a189-e09224e886a9",
        "Index": 1,
        "Name": "Vendor"
      },
      {
        "GroupId": "6a80aba7-348a-4515-b5be-60bec7d76b58",
        "Index": 0,
        "Name": "Operator"
      }
    ]
  }
}
```

The provisioning middleware

The job of the provisioning middleware is to

1. set up the communication broker and UI for the choice of controller to set up—that is, the **configuration target**—and
2. start the communication broker and UI.

Each part is the charge a single service:

- The **configuration target service** gets and sets the configuration target. There are two parts to that:
  - for a new controller, it connects to the container database, gets rid of all old records, and sets the controller ID and application ID to a new values, and
  - it puts the configuration target in a JSON file for the communication broker to read.
- The **app manager service** starts and stops the communication broker and UI.

Values to set

```
{
  "AppManager": {
    /* The value of the DISPLAY environment variable. *.
    "AppDisplay": ":14",
    /* The path to the Mono Runtime */.
    "AppInterpreter": "/usr/bin/mono",
    /* The path to the communication broker `*.exe`. */
    "CommunicationBrokerFileName": "/usr/share/CommunicationBroker.exe",
    /* The path to the UI `*.exe` */
    "UILoaderGtkFileName": "/usr/share/UI.exe"
  },

  "ConfigurationTarget": {
    /* Components of the database connection string to set the controller ID and
application ID */
    "DatabaseConnectionProperties": {
      "Database": "controllerconfigurationdatabase",
      "Host": "localhost",
      "KeepAlive": 1,
      "Password": "strongpassword",
      "Username": "restricteduser"
    },
    /* The path to the configuration target JSON file */
    "Path": "/etc/configuration-target.json"
  },
}
```

The proxy middleware

The job of the proxy middleware is to pass on the HTTP request to Xpra.

It wraps around `ProxyKit`, a library written for this job. However, the need to keep the reverse proxy locked when in use meant that the reverse proxy instead calls a changed `ProxyKit`. The files for this new `ProxyKit` library are listed at `$(ROOT)/src/Proxies/ProxyKit`.

The configuration endpoint

The configuration endpoint is a serverless function whose job is to read the messages that come from controllers out in the field and take further steps. At first, the company hired another group for this work; the

configuration endpoint was part of what they made. Still, the goals of the work meant further changes to the configuration endpoint. Some of the simple changes:

	old	new
Process VM	.NET Framework 4.7.1	.NET Core 3.1
Host library	WebJobs 2.3.0	Functions 3.0.3
Auth. library	ADAL 4.5.1	MSAL 4.8.2

The configuration endpoint files are listed under **`$(ROOT)/src/BrokerApi/BrokerInstance`**.

The new configuration endpoint keeps the flow of the old one, but adds a few new parts:

- When the communication broker in the remote HMI asks for changes to the target database,
    1. the configuration endpoint sends messages about what happened to the target database on to the Azure Event Grid,
    2. the Azure Event Grid puts these messages in an Azure Event Hub queue, and finally,
    3. the response gateway gets these messages back to the communication broker in the remote HMI.
- It sends messages about what happened to the backup database on to the Azure Event Grid as well.

The file **`img/configuration-endpoint-activity.pdf`** has a picture of the full new flow. (That picture is too big to fit here.)

The host

The configuration endpoint does not run on its own: another program must host it. As this host gets the configuration endpoint ready to run, it gives key-value pairs to the configuration endpoint.

Values to set

`AuthenticationClientId`
The value of **`.appId`** in the **`Microsoft.DirectoryServices.Application`** for the configuration endpoint.

Example: **`6df0567f-8219-43ce-b346-54931e0de093`**

`AuthenticationClientSecret`
One of the **`base64`** strings from **`.passwordCredentials | map(.value)`** in the configuration endpoint **`Microsoft.DirectoryServices.Application`**.

`AuthenticationInstance`
The string **`https://login.microsoftonline.com/{0}`**.

`AuthenticationTenant`
The host name that stands for the Azure Active Directory tenant.

Example: **`contoso.microsoftonline.com`**

`BrokerApiScope`
The URL that stands for being allowed to ask the configuration database API about controllers. The URL of the configuration database API is the value for **`BrokerApiUri`**.

Often, it looks like **`$identifierUri/.default`**, where **`$identifierUri`** is one of the string values of **`.identifierUris`** in the **`Microsoft.DirectoryServices.Application`** for the metadata database API.

Example: **`https://contoso.onmicrosoft.com/2c33ee88-3c02-4109-9be2-32eb4c3f1531/.default`**

`BrokerApiUri`
The URL of the configuration database API.

Example: `https://as-prod-configuration-database-api-x2-contoso.azurewebsites.net/api/Config/ExecuteConfigData`

`EventGridTopicHostName`
The full host name for the Event Grid topic.

Example: `egt-rhmi-prod-x2-contoso.centralus-1.eventgrid.azure.net`

`EventGridTopicKey`
The `base64` form of the Event Grid shared secret.

`EventHubConnectionString`
The connection string for the Event Hub Namespace from which the configuration endpoint gets messages.

`EventHubName`
The name of the Event Hub on which the configuration endpoint receives messages from the controllers.

Example: `eh-targetrequests`

`IotHubConnectionApiScope`
The URL that stands for being allowed to ask the metadata database API about controllers. The URL of the metadata database API is the value for `IotHubConnectionApiUri`.

The usual form is similar to that for the value of the `BrokerApiScope` key.

`IotHubConnectionApiUri`
The URL of the metadata database API as the API Management Service serves it.

Example: `https://apim-prod-config-x2-contoso.azure-api.net/api/TableOperations`

`OcpApimSubscriptionKey`
The secret, in `hexdump` form, for the API Management Service subscription.

`IotHubHost`
The full host name of the Azure IoT Hub with which controllers talk.

Example: `iot-stag-x2-contoso.azure-devices.net`

`IotHubPolicyName`
The name of the authorization rule to use for the Azure IoT Hub.

Example: `configuration-endpoint`

`IotHubKey`
The shared secret in `base64` form that belongs to the Azure IoT Hub authorization rule and that is used in SASL PLAIN authentication.

The communication broker

So as to reach the goals of the work, the Docker container is host to a changed communication broker with extra services. The files for this communication broker are listed at `$(ROOT)/src/CommunicationBroker`. The files for the services are listed at `$(ROOT)/src/BrokerApi/BrokerApi/Services`.

- The **device ID service** calls the target metadata API to get a device ID from the controller ID specified in the configuration target file.

- The **Azure IoT Hub service** gives messages to the Azure IoT Hub to send to the real controller.
- The **cloud configuration snapshot service** does export and backup for configuration files in the cloud.
- The **system user service** holds a fake list of users as a supplicant for the UI requests.