Explore Azure IoT Edge architecture on Linux

2017-6-7 • 9 min to read • Contributors

In this article

Linux

This article provides a detailed walkthrough of the Hello World sample code to illustrate the fundamental components of the Azure IoT Edge architecture. The sample uses the Azure IoT Edge to build a simple gateway that logs a "hello world" message to a file every five seconds.

This walkthrough covers:

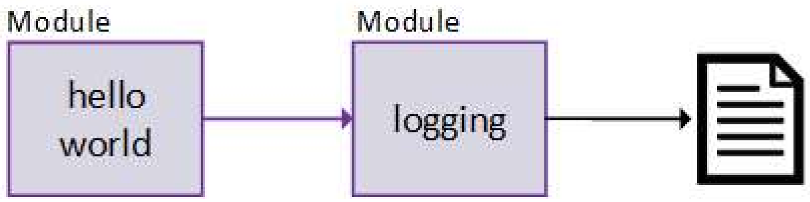
- **Hello World sample architecture**: Describes how Azure IoT Edge architectural concepts apply to the Hello World sample and how the components fit together.
- **How to build the sample**: The steps required to build the sample.
- **How to run the sample**: The steps required to run the sample.
- **Typical output**: An example of the output to expect when you run the sample.
- **Code snippets**: A collection of code snippets to show how the Hello World sample implements key IoT Edge gateway components.

Hello World sample architecture

The Hello World sample illustrates the concepts described in the previous section. The Hello World sample implements a IoT Edge gateway that has a pipeline made up of two IoT Edge modules:

- The *hello world* module creates a message every five seconds and passes it to the logger module.

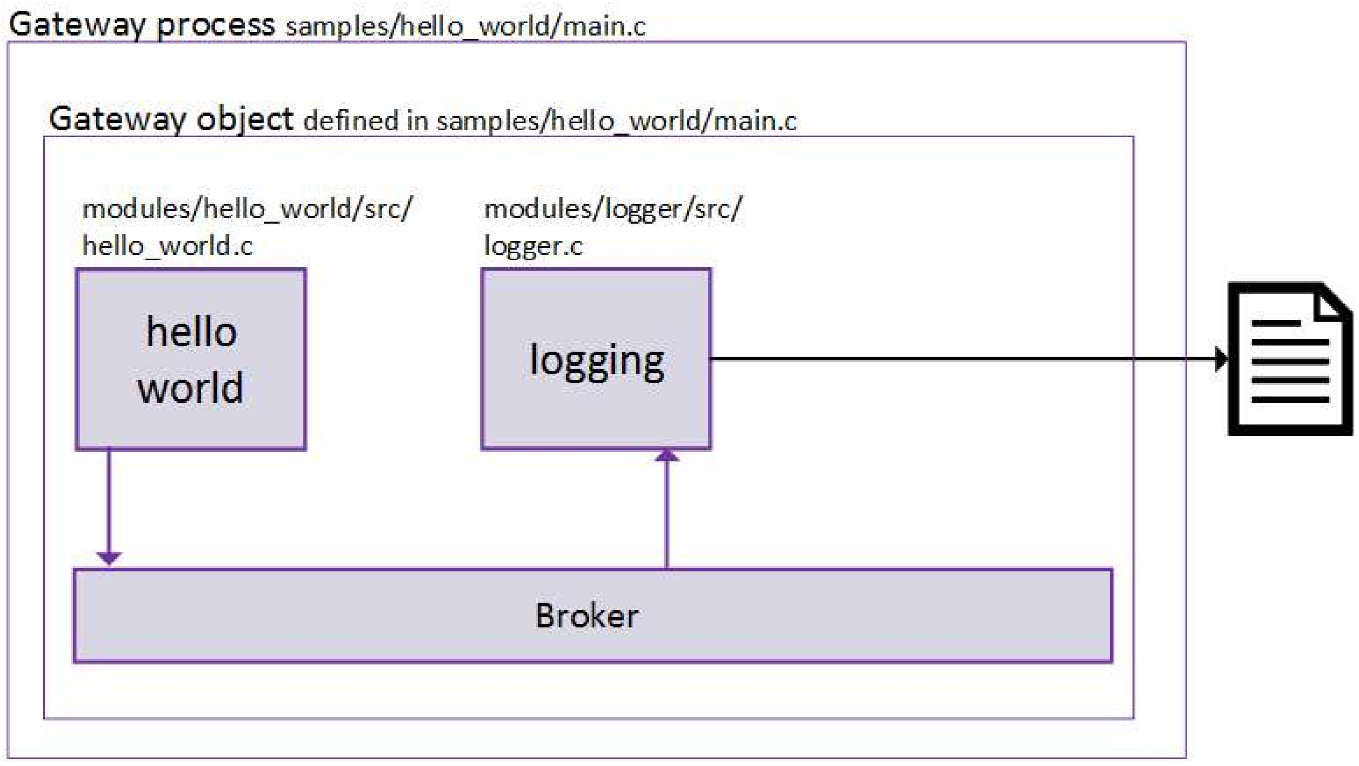
- The *logger* module writes the messages it receives to a file.



As described in the previous section, the Hello World module does not pass messages directly to the logger module every five seconds. Instead, it publishes a message to the broker every five seconds.

The logger module receives the message from the broker and acts upon it, writing the contents of the message to a file.

The logger module only consumes messages from the broker, it never publishes new messages to the broker.


Gateway process samples/hello_world/main.c
Gateway object defined in samples/hello_world/main.c
modules/hello_world/src/hello_world.c
modules/logger/src/logger.c
hello world
logging
Broker

The figure above shows the architecture of the Hello World sample and the relative paths to the source files that implement different portions of the sample in the repository. Explore the code on your own, or use the code snippets below as a guide.

Install the prerequisites

The steps in this tutorial assume you are running Ubuntu Linux.

Open a shell and run the following commands to install the prerequisite packages:

bash Copy

```bash
sudo apt-get update
sudo apt-get install curl build-essential libcurl4-openssl-dev git cmake libssl-dev uuid-d
```

In the shell, run the following command to clone the Azure IoT Edge GitHub repository to your local machine:

bash Copy

```bash
git clone https://github.com/Azure/iot-edge.git
```

How to build the sample

You can now build the IoT Edge runtime and samples on your local machine:

1. Open a shell.
2. Navigate to the root folder in your local copy of the **iot-edge** repository.
3. Run the build script as follows:

   sh Copy

   ```sh
   tools/build.sh --disable-native-remote-modules
   ```

This script uses the **cmake** utility to create a folder called **build** in the root folder of your local copy of the **iot-edge** repository and generate a makefile. The script then builds the solution, skipping unit tests and end to end tests. If you want to build and run the unit tests, add the `--run-unittests` parameter. If you want to build and run the end to end tests, add the `--run-e2e-tests`.

Note

Every time you run the **build.sh** script, it deletes and then recreates the **build** folder in the root folder of your local copy of the **iot-edge** repository.

How to run the sample

The **build.sh** script generates its output in the **build** folder in your local copy of the **iot-edge** repository. This output includes the two IoT Edge modules used in this sample.

The build script places **liblogger.so** in the **build/modules/logger/** folder and **libhello_world.so** in the **build/modules/hello_world/** folder. Use these paths for the **module path** values as shown in the following example JSON settings file.

The hello_world_sample process takes the path to a JSON configuration file a command-line argument. The following example JSON file is provided in the SDK repository at **samples/hello_world/src/hello_world_lin.json**. This configuration file works as is unless you modify the build script to place the IoT Edge modules or sample executables in non-default locations.

Note

The module paths are relative to the current working directory from where the hello_world_sample executable is launched, not the directory where the executable is located. The sample JSON configuration file defaults to writing 'log.txt' in your current working directory.

JSON   Copy

```json
{
    "modules" :
    [
        {
            "name" : "logger",
            "loader": {
            "name": "native",
            "entrypoint": {
                "module.path": "./modules/logger/liblogger.so"
            }
            },
            "args" : {"filename":"log.txt"}
        },
        {
            "name" : "hello_world",
            "loader": {
            "name": "native",
            "entrypoint": {
                "module.path": "./modules/hello_world/libhello_world.so"
            }
            },
            "args" : null
        }
    ],
    "links":
    [
        {
            "source": "hello_world",
            "sink": "logger"
        }
    ]
}
```

1. Navigate to the **build** folder in the root of your local copy of the **iot-edge** repository.
2. Run the following command:

sh Copy

```sh
./samples/hello_world/hello_world_sample ../samples/hello_world/src/hello_world_l
```

Typical output

The following example shows the output written to the log file by the Hello World sample. The output is formatted for legibility:

JSON Copy

```json
[{
    "time": "Mon Apr 11 13:48:07 2016",
    "content": "Log started"
}, {
    "time": "Mon Apr 11 13:48:48 2016",
    "properties": {
        "helloWorld": "from Azure IoT Gateway SDK simple sample!"
    },
    "content": "aGVsbG8gd29ybGQ="
}, {
    "time": "Mon Apr 11 13:48:55 2016",
    "properties": {
        "helloWorld": "from Azure IoT Gateway SDK simple sample!"
    },
    "content": "aGVsbG8gd29ybGQ="
}, {
    "time": "Mon Apr 11 13:49:01 2016",
    "properties": {
        "helloWorld": "from Azure IoT Gateway SDK simple sample!"
    },
        "content": "aGVsbG8gd29ybGQ="
    }, {
        "time": "Mon Apr 11 13:49:04 2016",
        "content": "Log stopped"
    }]
```

Code snippets

This section discusses some key sections of the code in the hello_world sample.

IoT Edge gateway creation

You must implement a *gateway process*. This program creates the internal infrastructure (the broker), loads the IoT Edge modules, and configures the gateway process. IoT Edge provides the **Gateway_Create_From_JSON** function to enable you to bootstrap a gateway from a JSON file. To use the **Gateway_Create_From_JSON** function, pass it the path to a JSON file that specifies the IoT Edge modules to load.

You can find the code for the gateway process in the *Hello World* sample in the main.c file. For legibility, the following snippet shows an abbreviated version of the gateway process code. This example program creates a gateway and then waits for the user to press the **ENTER** key before it tears down the gateway.

```c
int main(int argc, char** argv)
{
    GATEWAY_HANDLE gateway;
    if ((gateway = Gateway_Create_From_JSON(argv[1])) == NULL)
    {
        printf("failed to create the gateway from JSON\n");
    }
    else
    {
        printf("gateway successfully created from JSON\n");
        printf("gateway shall run until ENTER is pressed\n");
        (void)getchar();
        Gateway_LL_Destroy(gateway);
    }
    return 0;
}
```

The JSON settings file contains a list of IoT Edge modules to load and the links between the modules. Each IoT Edge module must specify a:

- **name**: a unique name for the module.
- **loader**: a loader that knows how to load the desired module. Loaders are an extension point for loading different types of modules. IoT Edge provides loaders for use with modules written in native C, Node.js, Java, and .NET. The Hello World sample only uses the native C loader because all the modules in this sample are dynamic libraries written in C. For more information about how to use IoT Edge modules written in different languages, see the Node.js, Java, or .NET samples.
  - **name**: the name of the loader used to load the module.
  - **entrypoint**: the path to the library containing the module. On Linux this library is a .so file, on Windows this library is a .dll file. The entry point is specific to the type of loader being used. The Node.js loader entry point is a .js file. The Java loader entry point is a classpath and a class name. The .NET loader entry point is an assembly name and a class name.
- **args**: any configuration information the module needs.

The following code shows the JSON used to declare all the IoT Edge modules for the Hello World sample on Linux. Whether a module requires any arguments depends on the design of the module. In this example, the logger module takes an argument that is the path to the output file and the hello_world module has no arguments.

JSON Copy

```json
"modules" :
[
    {
        "name" : "logger",
        "loader": {
          "name": "native",
          "entrypoint": {
            "module.path": "./modules/logger/liblogger.so"
        }
        },
        "args" : {"filename":"log.txt"}
    },
    {
        "name" : "hello_world",
        "loader": {
          "name": "native",
          "entrypoint": {
            "module.path": "./modules/hello_world/libhello_world.so"
        }
        },
        "args" : null
    }
]
```

The JSON file also contains the links between the modules that are passed to the broker. A link has two properties:

- **source**: a module name from the `modules` section, or `\*`.
- **sink**: a module name from the `modules` section.

Each link defines a message route and direction. Messages from the **source** module are delivered to the **sink** module. You can set the **source** module to `\*`, which indicates that the **sink** module receives messages from any module.

The following code shows the JSON used to configure links between the modules used in the hello_world sample on Linux. Every message produced by the `hello_world` module is consumed by the `logger` module.

JSON Copy

```
"links":
[
    {
        "source": "hello_world",
        "sink": "logger"
    }
]
```

Hello_world module message publishing

You can find the code used by the hello_world module to publish messages in the 'hello_world.c' file. The following snippet shows an amended version of the code with comments added and some error handling code removed for legibility:

C Copy

```c
int helloWorldThread(void *param)
{
    // create data structures used in function.
    HELLOWORLD_HANDLE_DATA* handleData = param;
    MESSAGE_CONFIG msgConfig;
    MAP_HANDLE propertiesMap = Map_Create(NULL);

    // add a property named "helloWorld" with a value of "from Azure IoT
    // Gateway SDK simple sample!" to a set of message properties that
    // will be appended to the message before publishing it.
    Map_AddOrUpdate(propertiesMap, "helloWorld", "from Azure IoT Gateway SDK simple sample

    // set the content for the message
    msgConfig.size = strlen(HELLOWORLD_MESSAGE);
    msgConfig.source = HELLOWORLD_MESSAGE;

    // set the properties for the message
    msgConfig.sourceProperties = propertiesMap;

    // create a message based on the msgConfig structure
    MESSAGE_HANDLE helloWorldMessage = Message_Create(&msgConfig);

    while (1)
    {
        if (handleData->stopThread)
        {
            (void)Unlock(handleData->lockHandle);
            break; /*gets out of the thread*/
        }
        else
        {
            // publish the message to the broker
            (void)Broker_Publish(handleData->brokerHandle, helloWorldMessage);
            (void)Unlock(handleData->lockHandle);
        }

        (void)ThreadAPI_Sleep(5000); /*every 5 seconds*/
    }

    Message_Destroy(helloWorldMessage);

    return 0;
}
```

Hello_world module message processing

The hello_world module never processes messages that other IoT Edge modules publish to the broker. Therefore, the implementation of the message callback in the hello_world module is a no-op function.

```
C                                                                    Copy
```

```
static void HelloWorld_Receive(MODULE_HANDLE moduleHandle, MESSAGE_HANDLE messageHandle)
{
    /* No action, HelloWorld is not interested in any messages. */
}
```

Logger module message publishing and processing

The logger module receives messages from the broker and writes them to a file. It never publishes any messages. Therefore, the code of the logger module never calls the **Broker_Publish** function.

The **Logger_Receive** function in the logger.c file is the callback the broker invokes to deliver messages to the logger module. The following snippet shows an amended version with comments added and some error handling code removed for legibility:

```c
static void Logger_Receive(MODULE_HANDLE moduleHandle, MESSAGE_HANDLE messageHandle)
{

    time_t temp = time(NULL);
    struct tm* t = localtime(&temp);
    char timetemp[80] = { 0 };

    // Get the message properties from the message
    CONSTMAP_HANDLE originalProperties = Message_GetProperties(messageHandle);
    MAP_HANDLE propertiesAsMap = ConstMap_CloneWriteable(originalProperties);

    // Convert the collection of properties into a JSON string
    STRING_HANDLE jsonProperties = Map_ToJSON(propertiesAsMap);

    //  base64 encode the message content
    const CONSTBUFFER * content = Message_GetContent(messageHandle);
    STRING_HANDLE contentAsJSON = Base64_Encode_Bytes(content->buffer, content->size);

    // Start the construction of the final string to be logged by adding
    // the timestamp
    STRING_HANDLE jsonToBeAppended = STRING_construct(",{\"time\":\"");
    STRING_concat(jsonToBeAppended, timetemp);

    // Add the message properties
    STRING_concat(jsonToBeAppended, "\",\"properties\":");
    STRING_concat_with_STRING(jsonToBeAppended, jsonProperties);

    // Add the content
    STRING_concat(jsonToBeAppended, ",\"content\":\"");
```