Dronegrid – Innovaton Project

Microsoft DX Germany helped DroneGrid to redesign their IoT solution and switch the back- and frontend to Azure PaaS services to give DroneGrids customers a better experience by using their software.

Key technologies

- Visual Studio 2017 RC
- Windows 10 Enterprise
- Azure SDK for .NET
- GitHub
- Azure IoT Hub
- Azure Functions
- Azure Web App
- Shared Access Signature

Core Team

- Benjamin Tokgöz, Technical Evangelist, Microsoft
- Ville Rantala, SDE, Partner Catalyst Team Microsoft
- Maxime Bossens, CTO , DroneGrid

Customer profile

DroneGrid is a Belgian company which was founded in August 2016. DroneGrid targets specific end-to-end enterprise applications and is active in the energy, construction/mining and infrastructure sectors.

DroneGrid in their own words:

"DroneGrid helps enterprises increase operational efficiency by giving them access to accurate, repeatable aerial information. Our enterprise solution allows our customers to focus on what's important i.e. working with business-critical data to take better informed decisions. Collecting aerial drone data has never been easier for our customers. Accessible to anyone with a web browser and internet connection, managing a drone operation is as easy as few clicks."

Problem statement

DroneGrid is working on a solution that realizes automated drone operation management for customers all over the world. They have automated operations for mining & quarring, energy and construction. For example, nearly 60% of the solar panel operator and O&M companies in Belgium are customers of them. A customer can manage each of their operations through a ReactJS web-interface and select areas which they want to scan from automated drones through browser interactions and marking lines in the web-browser. After their solution was built on VMs with frameworks like RabbitMQ and self-managed database, DroneGrid wanted to change their solution in a modern, scalable PaaS plattform on Azure to give their own business and customers a higher impact and user experience. Please note that at this point every DroneGrid VM still runs on Azure.

The most interesting use case for the docking is on a mining site, where customers want to collect data every single day or even multiple times a day. Currently the customers perform weekly flights through the DroneGrid platform, but with a human drone pilot coming to their site. A mine can be covered using one or maximum two drones. Since the drone can recharge itself, covering large areas can be done by performing multiple flights.

In the future, the total amount of drones managed by the system could easily go up to a few hundred.

Proposed solution

To support DroneGrid in analyzing, redesigning and building their solution in Azure, Microsoft decided to do a few steps to reach that goal:

- Technical Assessment Calls - where we analyzed the current state of their solution
- IoT and App Service workshop in Munich - to define an optimized and scalable architecture in Azure and explain some code samples and specifics by building the architecture.

Before we could start with a solution proposal we had to understand DroneGrids current architecture. The current state of the DroneGrid customer scenario looked like this:

![DroneGrid state by beginning]({{ site.baseurl }}/images/2017-05-24-DroneGrid/DroneGridCurrentState.png)

A typical DroneGrid customer has at least one object – for better understanding a solar site – which they want to manage and analyze. In the solar site example they possibly want to analyze which solar panels have defects. To realize and start an operation like this, they can easily manage his objects through the DroneGrid-web-interface which is implemented in ReactJS. He can define an object, initiate an operation for sending automated drones which will take images of the object, take it back to a docking station and send these images to an analyzing processing partner. About 50 to 60% of the cost of a drone operation is the drone pilot. If a customers does more than 5-6 flights a month, it becomes cheaper to install a docking station. Docking stations are leased to the customer on a monthly basis. The drones are running on DJI autopilots. DroneGrid does not build the drone software or hardware, but adds a module which connects into the API/SDK of the flight controller. In the current prototype, a raspberry pi 3 is added as a companion computer, running Linux. The processing partner analyzes the received images and sends them back to the DroneGrid-database. After all, DroneGrid is able to display the current states of the object and its possibly broken panels and their position.

In this early state DroneGrid's solution worked pretty well, but seperating the individual software-components in PaaS, especially the IoT component and the web services could reduce costs, increase the scalability and make it easier to manage new features and extensions would be worthwhile in the future.

During the first technical assessment calls, DX Germany proposed a solution as a starting point like the architecture below:

![DroneGrid first design approach]({{ site.baseurl }}/images/2017-05-24-DroneGrid/DroneGridTargetState.png)

The main difference in this first approach is to seperate each component of DroneGrid's solution in its own PaaS service on Azure. Especially the IoT Hub as a component for communication with the drone devices is a critical part of the software, because every request from and to the drones is managed by this service. Using an Azure Function, metadata from IoT Hub is directly stored into

the PostGreSQL database through an EventHub trigger. DroneGrid is with a solution like this able to focus more on their own business logic and the also important part of the UI. Other important resources are the blob storage for shared images and the Web App as a service for all control-elements from the user and the administrator perspective. Regarding the storage, DroneGrid has the possibility to share data easily with their customers through a dynamically generated SAS-Token.

Furthermore a future perspective could be to handle the image processing by DroneGrid instead of employing a third party analyzer. In this case we decided that the state is too early to integrate some Intelligence services for image analytics.

Technical delivery

We started the project with a few technical assessment calls with technical experts from Microsoft and DroneGrid to analyze the current state of the solution and plan a new PaaS architecture. Afterwards, Microsoft and DroneGrid came together to a workshop in the Microsoft Germany headquarters in Munich to code and work on unresolved challenges.

The result of the workshop was a better technical knowledge for DroneGrid especially in the IoT Services and shared storage. Furthermore Microsoft and DroneGrid collaborating creating code samples for each approach to help DroneGrid in their development.

In the following sections it will be described on which topics we worked together in the workshop, the learnings and corresponding code snippets.

![DroneGrid workshop design]({{ site.baseurl }}/images/2017-05-24-DroneGrid/MucArchitecture.png)

The image above shows our first architecture approach we worked together on at the workshop in Munich. As you see it's quite similar to the first approach during the technical assessment calls. Caused by the possibility to upload files directly from the IoT Hub we don't need the Stream Analytics service in this scenario anymore. There are two types of data transmission.

- During flight: only telemetry data is sent (over a long-range radio connection), which is status, gps location, altitude, velocity, battery level. The frequency is once every second.
- After flight, during charging: mission data is transferred over a fast 2.4Ghz connection. This data includes pictures taken by the camera. A 30 min mission could easily gather up to 2GB of images.

The first coding challenge was to handle file uploads from devices to the IoT Hub with Python. At this point I would like to give a special thanks to Ville Rantala, who demonstrated a solution to help DroneGrid implement an IoT Hub Connection from their Drone Stations.

```
...

PROTOCOL = IoTHubTransportProvider.MQTT

CONNECTION_STRING = "[Device Connection String]"

client = None


// print commands show how to get informations from the message
def receive_message_callback(message, counter):

         global RECEIVE_CALLBACKS

         message_buffer = message.get_bytearray()
```

```
          size = len(message_buffer)

    print("Received Message [%d]:" % counter)

    print("    Data: <<<%s>>> & Size=%d" % (message_buffer[:size].decode('utf-8'), size))

    map_properties = message.properties()

    key_value_pair = map_properties.get_internals()

    print("    Properties: %s" % key_value_pair)

    counter += 1

    RECEIVE_CALLBACKS += 1

    print("    Total calls received: %d" % RECEIVE_CALLBACKS)

    return IoTHubMessageDispositionResult.ACCEPTED


def send_confirmation_callback(message, result, user_context):

    global SEND_CALLBACKS

    print("Confirmation[%d] received for message with result = %s" % (user_context,
result))

    map_properties = message.properties()

    print("    message_id: %s" % message.message_id)

    print("    correlation_id: %s" % message.correlation_id)

    key_value_pair = map_properties.get_internals()

    print("    Properties: %s" % key_value_pair)

    SEND_CALLBACKS += 1

    print("    Total calls confirmed: %d" % SEND_CALLBACKS)


def device_twin_callback(update_state, payload, user_context):

    global TWIN_CALLBACKS

    print("\nTwin callback called with:\nupdateStatus = %s\npayload = %s\ncontext = %s" %
(update_state, payload, user_context))

    TWIN_CALLBACKS += 1

    print("Total calls confirmed: %d\n" % TWIN_CALLBACKS)



def send_reported_state_callback(status_code, user_context):

    global SEND_REPORTED_STATE_CALLBACKS

    print("Confirmation for reported state received with:\nstatus_code = [%d]\ncontext =
%s" % (status_code, user_context))

    SEND_REPORTED_STATE_CALLBACKS += 1

    print("    Total calls confirmed: %d" % SEND_REPORTED_STATE_CALLBACKS)
```

```
def device_method_callback(method_name, payload, user_context):

    global METHOD_CALLBACKS

    print("\nMethod callback called with:\nmethodName = %s\npayload = %s\ncontext = %s" %
(method_name, payload, user_context))

    METHOD_CALLBACKS += 1

    print("Total calls confirmed: %d\n" % METHOD_CALLBACKS)

    device_method_return_value = DeviceMethodReturnValue()

    device_method_return_value.response = "{ \"Response\": \"This is the response from the
device\" }"

    device_method_return_value.status = 200

    return device_method_return_value



def blob_upload_conf_callback(result, user_context):

    global BLOB_CALLBACKS

    print("Blob upload confirmation[%s] received for message with result = %s" %
(user_context, result))

    BLOB_CALLBACKS += 1

    print("    Total calls confirmed: %d" % BLOB_CALLBACKS)

    message_text = "{\"filename\": \"%s\",\"status\": \"complete\"}" % user_context

    message = IoTHubMessage(message_text)

    client.send_event_async(message, send_confirmation_callback, 1)

//client initialization
def iothub_client_init():

    global client

    # prepare iothub client

    client = IoTHubClient(CONNECTION_STRING, PROTOCOL)

    if client.protocol == IoTHubTransportProvider.HTTP:

        client.set_option("timeout", TIMEOUT)

        client.set_option("MinimumPollingTime", MINIMUM_POLLING_TIME)

    # set the time until a message times out

    client.set_option("messageTimeout", MESSAGE_TIMEOUT)

    # to enable MQTT logging set to 1

    if client.protocol == IoTHubTransportProvider.MQTT:

        client.set_option("logtrace", 0)
```

```
    client.set_message_callback(

        receive_message_callback, RECEIVE_CONTEXT)

    if client.protocol == IoTHubTransportProvider.MQTT or client.protocol ==
IoTHubTransportProvider.MQTT_WS:

        client.set_device_twin_callback(

            device_twin_callback, TWIN_CONTEXT)

        client.set_device_method_callback(

            device_method_callback, METHOD_CONTEXT)

    return client

...
```

The code snippet above represents a sample to connect from a device to an Iot Hub and be ability to upload blobs.

The second code snippet represents a way to store device metadata via the IoTHub to a PostgreSQL Database, using an Azure Javascript Function.

```
var Pool = require('pg').Pool;

var config = require('./config.js');

var pool = new Pool(config);

module.exports = function (context, event) {

  context.log(context.bindingData);

  context.log(event);

  pool.connect()

    .then((client) => {

      client.query('INSERT INTO events(event) VALUES($1)', [JSON.stringify(event)])

        .then((res) => {

          context.log('Successfully inserted');

          client.release();

          context.done();

        })

        .catch((err) => {

          client.release();

          context.log(err);

          context.done();

        });

    })

    .catch((err) => {

      context.log(err);
```

```
      context.done();

    });

};
...
```

Furthermore we discussed a possibility to upload and download files from a blob storage with a SAS Token. This is important for the Image Processing Partner and also for a user who wants to upload files manually.

The code snippet below represents a way to upload and download files with a SAS-Token from a blob storage.

```
public LinkedList<CloudBlockBlob> getBlobList(CloudBlobContainer container)
{
       LinkedList<CloudBlockBlob> tempBlobList = new LinkedList<CloudBlockBlob>();

       foreach (IListBlobItem item in container.ListBlobs(null, false))

       {

           if (item.GetType() == typeof(CloudBlockBlob))

           {

                CloudBlockBlob blob = (CloudBlockBlob)item;

                tempBlobList.AddLast(blob);

           }

        }

        return tempBlobList;

 }

public void downloadBlockBlob(CloudBlobContainer container, string blockblobname)
{

             CloudBlockBlob blockBlob = container.GetBlockBlobReference(blockblobname);

        using (var fileStream =
System.IO.File.OpenWrite(System.AppDomain.CurrentDomain.BaseDirectory + "\\" +
blockblobname))

        {

            blockBlob.DownloadToStream(fileStream);

        }

        Console.WriteLine(blockBlob.Name + " downloaded");

}


public void uploadBlockBlobManipulated(CloudBlobContainer container, string blockblobnname)
{

        //manipulateFile

        string newPrefix = "new-";
```

```
        CloudBlockBlob blockBlob = container.GetBlockBlobReference(newPrefix +
blockblobnname);

        using (var fileStream =
System.IO.File.OpenRead(System.AppDomain.CurrentDomain.BaseDirectory + "\\"
+blockblobnname))

        {

            blockBlob.UploadFromStream(fileStream);

        }

        Console.WriteLine(blockBlob.Name + " uploaded");


}
...
```

Furthermore we discussed implementing a resumable upload to blob storages. After the workshop, DroneGrid decided to use “FineUploader” as a solution that is available on the market today. However, the code for a resumable Upload is located under the following URL: https://github.com/benjamintokgoez/AzureBlobUploadJS .

The last subject of the discussion at the workshop was a solution for an easy and fast deployment from an existing Web App on GitHub to Azure Web Apps. We chose to use the Azure Web Portal to to deliver the Web App from GitHub. For this reason there will be no sample snippet at this point.

In the following 2 months DroneGrid decided to develop their redesigned solution by themselfs. During this development process DroneGrid faced a few challenges and focused mainly on the implementation of the Web App Service and the UI.

![DroneGrid current state after workshop]({{ site.baseurl }}/images/2017-05-24-DroneGrid/newCurrentArchitecture.png)

The architecture above shows the new and current architecture of DroneGrid. They are now using PaaS Services on Azure for each of their components. The next steps would be to integrate an IoT Hub for the communication with their drones and drone docking stations during the next months. All communication between the drone, dock and the cloud is secured. Security (from both hardware and software perspective) is not yet addressed specifically in the current prototype, development on this is planned in the coming months.

Conclusion

In DroneGrid's project, Microsoft helped successfully to implement a PaaS solution on Azure. In additon we helped DroneGrid to get a better understanding of Azure IoT approach. After the workshop, DroneGrid was able to implement their code. The only issue was a time constraint which is the reason for the currently missing Iot Hub service.

After all, it was a great experience to work with DroneGrid on their solution and we are looking forward to further projects together.

Finally, a quote from DroneGrid after our workshop:

"It was great to work with Microsoft on a new architecture on Azure. We learned a lot and hope to work together in the future again"