Device Monitoring API

Problem Specification

Create a simple Device Monitoring API that would allow a remote device (e.g. a single-board computer) to send status data to it, such that it could be processed centrally to identify any mis-functioning devices. The data will need to be stored in a table in SQL Azure.

The example should have the following characteristics:

- Devices are identifiable by serial number
- Each message delivered to the API will include a timestamp, a battery level and an uptime
- The API will use ASP.NET Core, and expect to receive and process JSON data
- The API will not talk directly to the database, but instead relay messages via a queue
- Messages will be received from the queue and stored in SQL Azure
- The API should support OpenAPI/Swagger
- Dependency Injection should be via Autofac
- Logging should be via Nlog
- Free reign to choose the queuing technology. It can be external (e.g. an Azure PaaS service) or internal (e.g. using one of the specialized .NET queuing classes)
- Free reign to choose an ORM or use raw sql directly
- All code in C#

It doesn't need to be a working sample. I care far more about interpretation or requirement, code style and quality, and system design.

Further Questions about the Specification

1. What should the format of the data stored in SQL Azure be, JSON text format or traditional SQL-Server tabular/relational format?

    Answer: Preferably the latter, i.e. tabular/relational format.

2. Can the device ID be a traditional auto-generated, auto-increment integer ID or the ID should actually be the serial number of the remote device?

Answer: Preferably the latter, i.e. the actual serial number of the device.

Problem Interpretation

My understanding of the problem is as follows:

We need to build a web API that can cope with various types of HTTP requests, typical such requests are the following:

- **HTTP GET**

- **HTTP POST**
- **HTTP PUT**
- **HTTP DELETE**

These requests will be initiated by various distributed devices, such as single-board computers, etc. Specifically these devices need to send messages containing specific information to the API using HTTP POST and HTTP PUT.

The API should be able to receive and process JSON data, so the messages need to be in that particular format.

A typical message will contain the following information:

- The **serial number** of the remote device that initiated the message.
- A **timestamp** that indicates when the message was generated.
- The **battery level** of the device.
- The **uptime** of the device.

A typical message presented in JSON format should look like this:

```
{
        "serialNumber":"SBC-00008891",
        "timestamp":"2019-03-03T03:34:17",
        "batteryLevel":68,
        "uptime":"13:27:43"
}
```

In the above example the battery level value is translated as a percentage, i.e. the battery has 68% of charge left in it, the rest of the data fields are self-explanatory.

All the messages received by the API from the remote devices need to be processed and then stored to a database.

A main requirement here is that the API shouldn't store the messages directly to a database. Instead t**he API should be able to forward these messages to a message queue**. This queue can be local, on a remote server, or on the cloud. Ideally the messages sent to the queue must be in a simple text format that is appropriate for data exchange, such as JSON. So, there is an implied requirement here that **the API should be able to serialise data objects to JSON text format** in order for this information to be easily attached as text to the messages.

Another obvious requirement is that **we need a queue in order to be able to store temporarily all these messages**.

Finally we need a mechanism, in the form of one or more applications of some sort, that can **receive these messages from the queue and store them to a cloud database**.

In order to store the messages we must be able to translate the data that they carry correctly, e.g. to recognise which part of the message is the device ID and which part is the rest of the information. This is because 1. the device serial number identifies a device uniquely and 2. the same device might send several messages over a period of time; receiving and storing a new message should not affect the persistence of all the previous ones for this particular device, i.e. our system should be able to store and recognise easily all the messages sent by the same device.

Solution Approach

Based on the above I have approached the problem as follows:

A. API

I have created an API with the following characteristics:

1. The web API than can receive and process the HTTP requests mentioned above, i.e. post, put, get and delete.
2. The API architecture is based on the ASP.NET MVC approach and consists of a main controller, called DeviceController.
3. The DeviceController can receive and process JSON data.
4. The data are automatically modelled as appropriate entity objects, i.e. Device objects, and stored in an in-memory SQL database using the DeviceContext class.
5. In order to be able to generate the messages that will be sent to the queue the API can serialise Device objects to JSON text format using the JsonSerialiser class.
6. The API sends the data to the message queue by using the QueueWritter class.

When a new POST request reaches the API the request is dealt with by the DeviceController that adds the device data to the in-memory database, calls a method from another service to process the data (this is left purposely blank as there are no specific processing requirements in the problem specification), serialises the device object and finally adds the serialised object to the messaging queue.

A PUT request is very similar to the above but instead modifies the existing device object in the in-memory database instead of generating a new one.

A GET request either returns all the existing device objects or the one requested by serial number (which is the unique ID for a device object).

Finally a DELETE request just deletes the specified object from the in-memory database.

Considerations:

1. Please notice that after an object has been serialised and sent to the queue there is no need to remain in the in-memory database. So I could have chosen to delete it at this point. I decided against that for testing and illustration purposes and because I consider this to be rather an exercise than a fully functional production system.
2. From the information above it is obvious that the controller uses other services in order to store data, serialise data generate and send queue messages, etc. In order to de-couple my main controller from these services I have used the built-in ASP.NET MVC Dependency Injection container. All the dependencies required by the constructor of the main controller are resolved by the DI container. Please have a look in the ConfigureServices() method of the Startup class for more details.

B. Queue

For the queueing technology I have used an Azure Service Bus queue.

C. Queue Client

I have created a simple console application that reads messages from the Service Bus queue, deserialises the messages to appropriate entity objects, and then stores the object data in a local SQL-Server database utilising Entity Framework Core and the Repository Pattern.

The application is clever enough to understand if a Device has been recorded in the database before or not, i.e. if the device’s unique serial number has been recorded before.

The database consists of two tables as follows:

1. The Devices table
2. The DeviceMessage table

The Device table keeps track of all the devices that have sent messages in the past. The DeviceMessages table keeps track of all the messages that have been received by all the devices. The SerialNumber of a device a the unique ID of the device in

the Devices table, and thus the primary key. The same key acts as a foreign key in the DeviceMessages table and in order to keep track which device sent which messages.

This is a simple application written in a hurry and for that reason certain characteristics are missing, such as: Exception handling, logging, etc. that I hope to add soon when I have the time to revisit the application.

For the IoC I have used a simple Service Locator pattern that registers and relolves the various services needed by the main program.

System Design

For an illustration of the architectural design of this system please see separate document:

Documents\SystemDesignOverview.pdf

Testing

Because of time constraints I didn't manage to produce a simple application that would generate HTTP requests to the API. In order to simulate this process and test the API I have used an application called:

Postman

Here is the link to the application download page:

https://www.getpostman.com/downloads/

Additionally in order to be able to inspect and test the functionality of the message queue I have used another application called:

Service Bus Explorer

ScreenShots

For a visualisation of the various components of the application please see the enclosed screenshots. The numbering of the screenshot files indicate the order in which they should be viewed, please view them in the correct order. All screenshots are in the folder **Documents**.