MyIoT
Telemetry Application
Architecture Document

Background

This document describes the MyIoT application's architecture, the young and promising (and fictitious) MyIoT company's main product.

MyIoT develops a dashboard system that reports the status of the IoT devices its clients are using and managing in almost real-time.

IoT stands for the Internet of Things and represents small, always-connected devices used every day by many of the population, such as home cameras, wifi routers, connected thermostats, and more.

Smart Homes are a great example of IoT consumers. A typical smart home has many IoT devices such as thermostats, lightbulbs, routers, cameras, electric switches, refrigerators, and lots more.

Each one of the devices has its own app and can be managed from a smartphone.

This creates a problem since the end-user has to navigate many apps with a diverse user interface and different information displayed.

This is what MyIoT is trying to solve with its application.

The application collects status information from registered IoT devices and formats the data to a visually pleasing dashboard, allowing the customer to know exactly what is going on with your devices using a unified, easy to read visualization.

Also, the customer can execute some predefined queries to access more information about the devices.

It's important to note that for phase one of the system described in this document, the customer is not adding or updating any data. The status info is received directly from the devices in the field. The data is just presented to the customer.

Another important aspect is that the launch customers will be entered into the system manually by MyIoT's salespeople, following an intensive verification process, to prevent data leaks. Thus, the system does not have a registration process, and it will work with already registered devices.

This document describes the system architecture of the MyIoT's application.

The architecture comprises technology and modeling decisions that will ensure the final product, assuming the architecture is followed, will be fast, reliable, and easy to maintain.

The document outlines the thought process for every aspect of the architecture and clearly explains why specific decisions were made.

The development team must follow the architecture depicted in this document closely. In any case of doubt, please consult the Software Architect.

Requirements

Functional Requirements

1. Receive status updates from IoT devices
2. Store the updates for future use
3. Allow end-users to query the updates and present them with the relevant information

Non-Functional Requirements

The following non-functional requirements were defined after discussions with the customer and are agreed upon by all the team members.

1. Data Volume: 54GB Annually
2. Load: 540 concurrent requests
3. No. of users: 2,000,000
4. Message loss: 1%
5. SLA: Platinum

Executive Summary

This document describes the new MyIoT application's architecture, an innovative breakthrough in the IoT world, allowing end-users to have a unified view of all their IoT devices, thus locating problems quickly and easily.

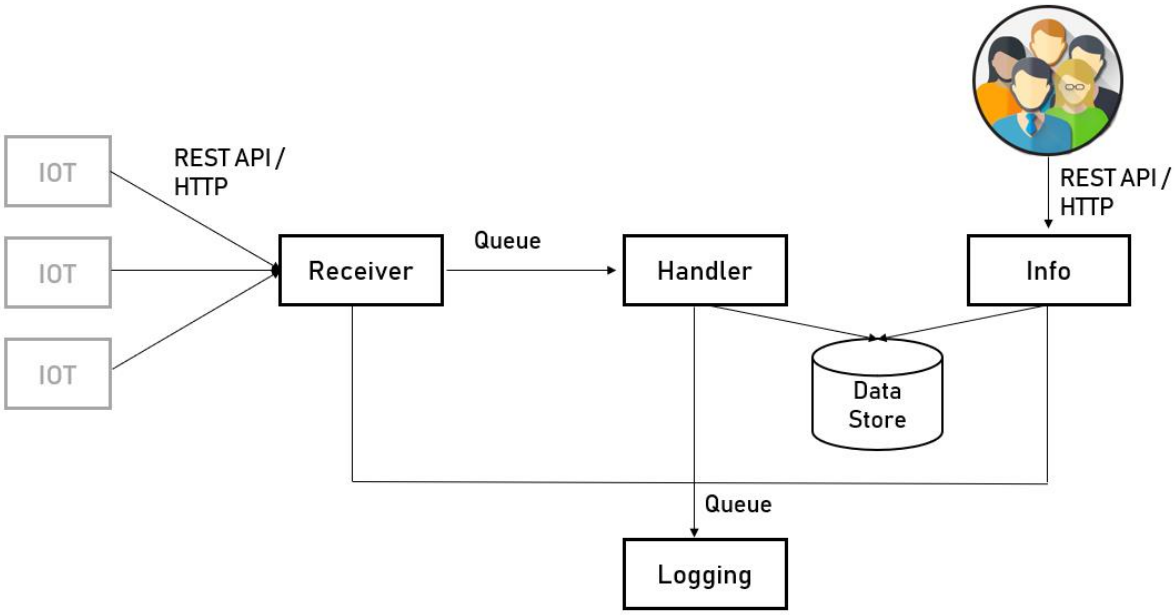
For example, using the MyIoT application, the user can find out whether all the cameras in her smart home are functioning correctly, and if not – what is the reason for that.

When designing the architecture, a strong emphasis was put on two major features:

- The application should be reliable
- The application should be swift

The architecture is based on the most up-to-date best practices and methodologies to achieve these features, ensuring high-availability and performance.

Here is a high-level overview of the architecture:



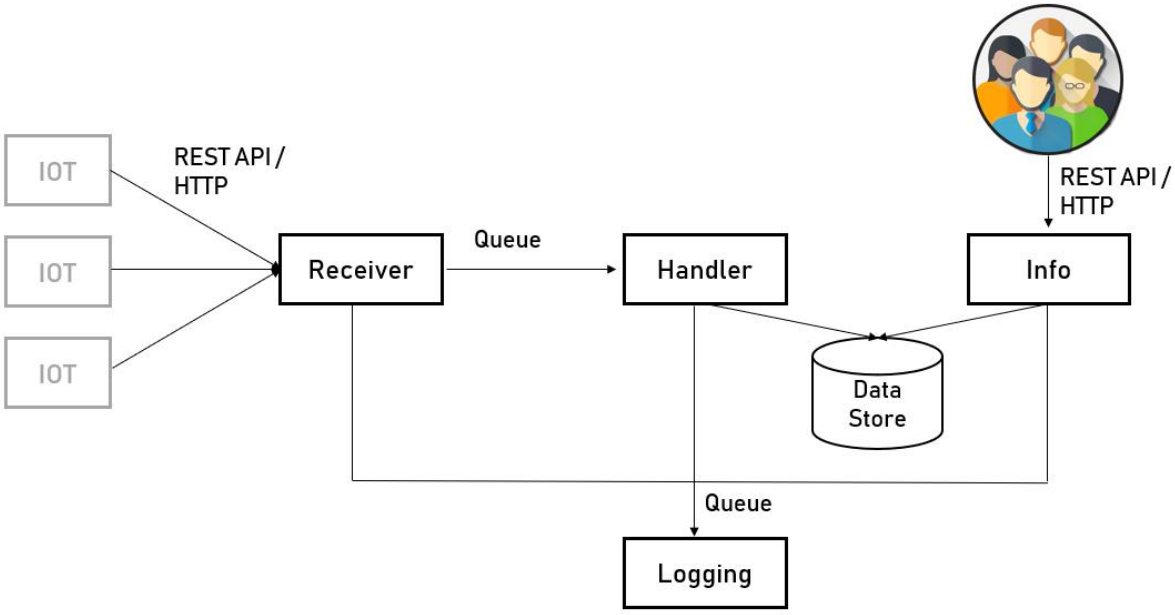
As shown in the diagram, the application comprises four separate, independent, loosely-coupled services. Each has its own tasks, and each communicates with the other services using standard protocols.

All the services are built as stateless services, meaning – no data is lost of the service is suddenly shutting down. The only places for data in the application are the Queue, and the Data Store, both of them serialize the data to the disk, thus protecting it from shutdown cases.

In conjunction with a modern development platform (.NET Core), this architecture will help create a modern, robust, easy to maintain, and reliable system that can serve the company successfully for years to come and helps it achieve its financial goals.

Overall Architecture

Here is the architecture diagram for the MyIoT application:



Services

The architecture comprised of the following services:

Receiver – Receives the status updates from the various devices and adds them to a Queue for further handling. The receiver puts a strong emphasis on performance, and its main task is to make sure the update was received and stored. It does not take any action on the update, which is the Handler service's role.

Handler – Validates and parses the update. The handler pulls the updates from the Queue (where the Receiver placed them), validates their content (for example – making sure the current status is within the right range), and convert them to a unified format, so it doesn't matter from which device the status was received – it will always look the same. After handling the message – the handler will store it in the message store.

Info – Exposes API for querying the data. This service accepts query requests from end-users and returns the required data about the status updates.

Logging – Aggregates all the logs generated by the various services to enable a unified view of all the system's events.

Scaling

This architecture allows efficiently scaling the services as needed. Each service is laser-focused on a specific, single task. Each can be scaled independently, either automatically (by service managers such as Kubernetes) or manually.

Also, the service's inner code is fully stateless, allowing scaling to be performed on a live system without changing any code lines or shutting down the system.

Messaging

The various services communicate with each other using multiple messaging methods. Each method was selected based on the specific requirements of the services. Here are the different messaging methods used in the system:

- The **Receiver** exposes REST API / HTTP. The reason for that is quite simple – the devices are programmed to call the application using REST API, so the receiver service has to comply with that.
- The **Handler** service does not have a classic API, and it grabs messages from a Queue. The reason for that is there is no requirement for synchronous handling of the messages, and the handler does not report back to the receiver when the handling is done. Besides, the Queue adds a layer of reliability that does not exist in a classic API.
- The **Info** service exposes REST API. Since REST API is the de-facto standard for most of the API consumers, and since this service will be used by an unknown number and types of consumers, it's best to go for the most widely-used messaging method - REST API. The REST API is also best suited for a request/response model, which is how this service will be used.
- Finally, the **Logging** service does not expose a classic API. It polls a queue that stores the log records. The reason is that performance is not very important, but the Queue provides reliability and order.

Services Drill Down

Logging

Role

The logging service is the aggregator of all the logs generated by the other system's benefits.

All the other services write their logs to a queue, and the logging service polls them and stores them in a central data store.

This way, the developers can get a unified view of all the system events and are not required to go through different log files in various formats to get a coherent picture of a specific flow or error.

Besides, since the logs are stored in a queryable data store, other programs can access it and query it in any way imaginable, from simple, REST-based queries to massive dashboards with loads of information. The data is there, and everyone can access it.

Technology Stack

Because the logging service is an always-active service and does not wait for requests to return response, as in traditional web apps, it will be a service.

As such, there are not many requirements for its implementation technology.

The technology to be able to access the Queue API and store data in a data store. This is nothing special, and any development platform can do that. Also, there are no special requirements for performance. Of course, we want it to be fast, but there is no specific requirement that limits us here.

In addition, one of the factors we should consider is the recent experience of the development team. Learning a new technology stack can take time, and the probability of a poor-quality code is higher with new technology.

The development team is familiar with the Microsoft stack, meaning - .NET platform and SQL Server.

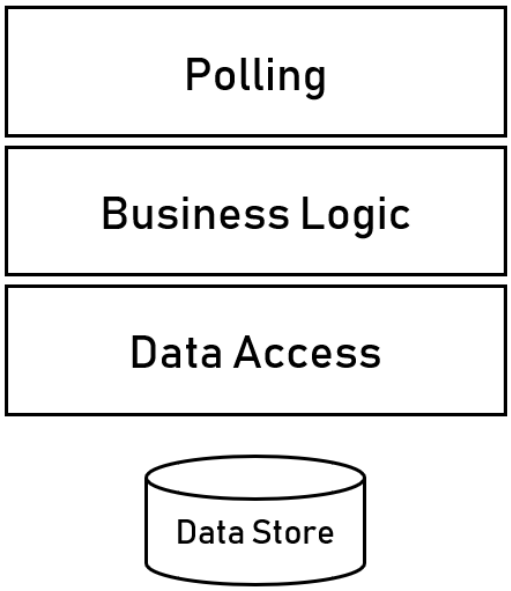
.NET is a general-purpose platform that can be used in Services and web apps and proved useful and fast in previous years.

The only caveat here is that .NET is a little bit outdated, and the new project should use .NET Core, which is the fresh, cross-platform, modular successor of .NET.

After some discussions, it was decided that .NET core is still the preferred option, and the project can still be ready on time so that the service will be based on .NET Core, and the database will be SQL Server.

Architecture

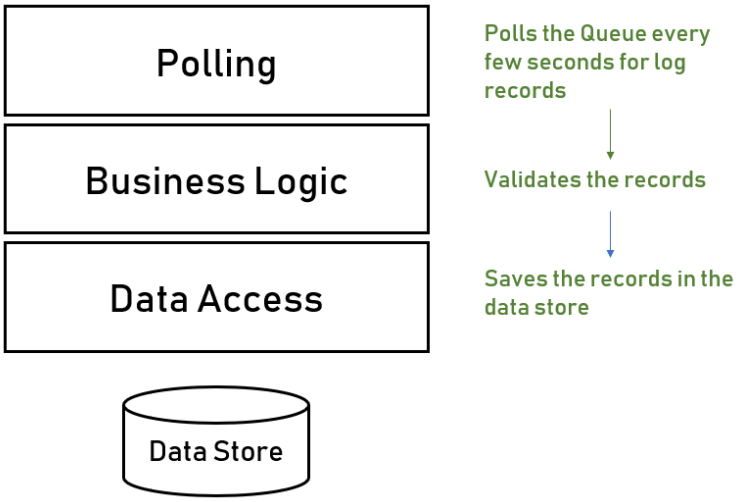
Here is the architecture of the logging service:



As you can see, this is a classic layered architecture, with a significant change - the top layer, which is usually a service layer, and a polling layer in this service.

The reason for that is since the logging service is not a Web API or Web app, it does not expose any interface, and therefore does not need a service interface layer. Instead, its polling layer manages the work with the Queue, from which the logs are pulled.

Here is the architecture with a description of every layer:



Note that every layer has a well-defined role. It's crucial to keep it this way and make sure no layer interferes with other layers, which will make the service much more difficult to maintain.

Implementation Instructions

- Use dependency injection between the various layers. Implement the dependency injection using `Microsoft.Extensions.DependencyInjection` package.
- Use the Entity Framework package for accessing the database. Do not use direct SQL statements.

Receiver

Role

The Receiver service is the service that the various IOT devices access to send their status updates. This is their interface with the system and the only benefit they're aware of.

Since there is more than one type of device, the Receiver can receive messages of various types and handle them all.

To make the Receiver as lightweight and fast as possible, its designed functionality is very focused and very minimal. The Receiver's only task is to receive the message and push it into a queue. Other services (in this case, the Handler) are responsible for taking care of the messages afterward.

Technology Stack

The Receiver is a Web API service since the devices will communicate via REST API, and this is a given.

So when selecting the technology stack, we need to make sure it supports the Web API scenario.

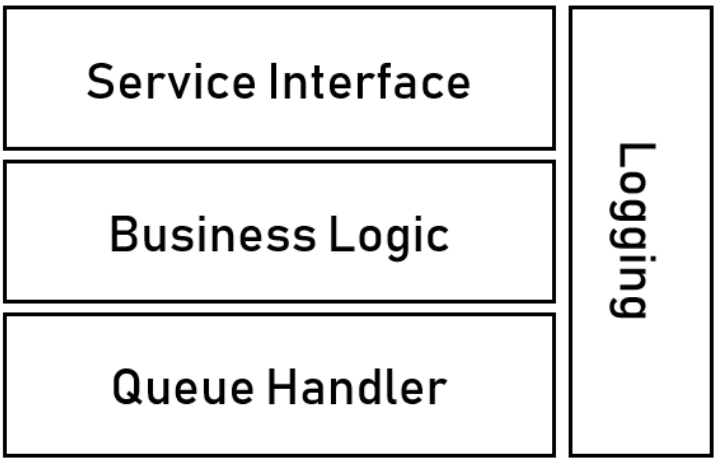
We already decided the first service will be based in .NET Core, and we will need an excellent reason to divert from this decision. Using multiple technologies in a single application can create many headaches if done for the wrong reasons, so let's see if there is any reason NOT to continue with .NET core on this service.

.NET core was designed from the ground up to support Web API applications using its ASP.NET Core libraries and has excellent support for it. Besides, its Web API performance is excellent.

So the decision for the Receiver service is to base it, too, on the .NET Core platform.

Architecture

Here is the architecture of the receiver service:



This is, similar to the logging service, a layered architecture, with some tweaks.

Here is a short description of every layer in the diagram:

Service Interface – Exposes REST API for the devices. This layer receives the messages and immediately transfers them to the business logic layer.

Business Logic – Receives the messages from the Service Interface layer, makes sure the messages are valid and passes them to the Queue Handler layer.

Queue Handler – This layer replaces the classic Data Access layer since there is no data store in this service. The Queue handler receives the messages from the Business Logic layer and adds them to the Queue. The Handler service is waiting on the other end of the queue to handle the messages.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to

log as fast and straightforward as possible. The log records are added to a queue, which is polled by the Logging Service.

Implementation Instructions

- The Service Interface layer should contain as little code as possible. No logic should occur, and its only task is to receive the messages and pass them to the Business Logic layer.
- Every step in the service must be logged. Use the logging component generously. Since there is no UI for this service, logging is the only way of figuring out what's going on.

Handler

Role

The Handler service is responsible for validating and parsing the messages and storing them in the data store.

The Handler listens to the Queue containing the messages, grabs them, and then handles them. (The messages were placed in the Queue by the receiver service).

Technology Stack

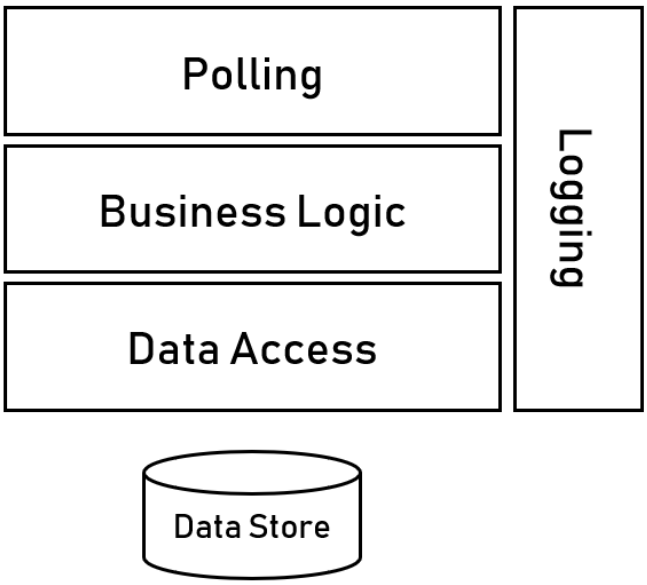
Like the logging service, the Handler service is a service, not a Web API or Web app, since it's always active and does not wait for requests to return a response but instead polls the queue for new messages, basically initiating the action instead of waiting for one.

By now, we already have two services for which we determined the technology stack – the logging service and the receiver. In both, we went for .NET Core.

Since it looks like there is no real reason to select other technology for the handler service, since there are no special requirements, and it will be developed by the same teams who developed the other services, we can be quite comfortable in using .NET core here too.

Architecture

Here is the architecture of the Handler service:



As you can see, this architecture is very similar to the Logging service's one.

Here is a short description of every layer in the diagram:

Polling – Manages the work with the messages' queue. This layer polls the queue for new messages, and when such messages are received, it immediately passes them to the business logic layer.

Business Logic – Receives the messages from the Polling layer, validates and parses the messages, and passes the resulting entities to the Data Access layer.

It's important to note here that since there are many kinds of messages requiring many validating and parsing types, it might be a good idea to implement a plug-in mechanism for each type of message. This way, when a new type of message should be supported, there will be no need to update the whole application and perhaps even to shut it down for the update to take place. Instead, a new plug-in will be placed in the plug-ins folder, and that's it. This mechanism can

easily be implemented using various plug-ins libraries, such as MEF, and it's recommended to use one of them.

Data Access – This layer stores the parsed messages in the data store. It's quite a superficial layer and exposes the minimal API required for these tasks.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible. The log records are added to a queue, which is polled by the Logging Service.

Implementation Instructions

- Use a plug-in mechanism for loading the various messages' parser and validator. Do not embed this code directly in the Business Layer code. Use an existing library for the plug-in mechanism and do not implement a new one. MEF is a popular choice for this.
- Every step in the service must be logged. Use the logging component generously. Since there is no UI for this service, logging is the only way of figuring out what's going on.

Info

Role

The Info Service allows end-users to query the data stored in the data store and display it in various forms.

The service is responsible only for data retrieval. It does not display the data. That's the responsibility of the client, whatever client it is.

The service exposes a standard REST API, enabling a wide range of clients to access it and query the data, using easy to use semantics and JSON-based data structures.

Technology Stack

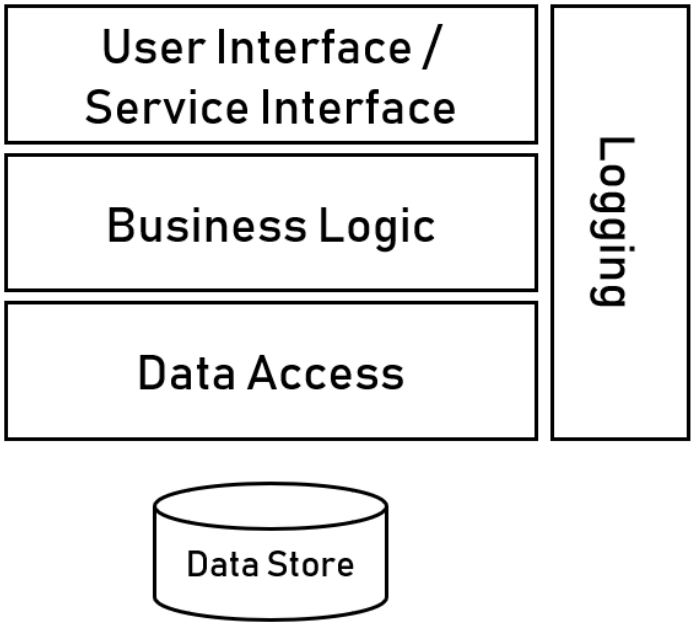
The Info is a Web API service since the various clients will communicate via REST API, which is the de-facto standard for Web API.

As with previous services, especially the Receiver Service, it looks like using .NET Core here, specifically – ASP.NET Core, the Web API library of .NET Core, is the right choice. It's lightweight, easy to implement, has excellent performance, and the developers are familiar with it.

So the Info Service will be based on .NET Core too.

Architecture

Here is the architecture of the receiver service:



This is, similar to the receiver service, a classic layered architecture.

Here is a short description of every layer in the diagram:

Service Interface – Exposes REST API for the clients. This layer receives the query requests and passes them to the Business Logic layer.

Business Logic – Receives the query requests from the Service Interface layer, validates them, and passes them on to the Data Access layer.

Data Access – Receives the query requests from the Business Logic layer, translates them to a query that can be understood by the database (i.e., SQL statements), and returns the results.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible. The log records are added to a queue, which is polled by the Logging Service.

It's crucial to craft an API that is easy to use and well understood by the clients to make the Info service usable.

For this reason, the API was defined as part of the architecture process, including its role, path, and response code.

The following table displays the API actions available by the Info service:

Functionality	Path	Return Codes
Get all the updates for a specific house's devices for a given time range	GET /api/house/*houseId*/devices/updates?from=*from*&to=*to*	200 OK 404 Not Found
Get the updates for a specific device for a given time range	GET /api/device/*deviceId*/updates?from=*from*&to=*to*	200 OK 404 Not Found
Get the current status of all the devices in a specific house	GET /api/house/*houseId*/devices/status/current	200 OK 404 Not Found
Get the current status of a specific device	GET /api/device/*deviceId*/status/current	200 OK 404 Not Found

Implementation Instructions

- The Service Interface layer should contain as little code as possible. No logic should occur, and its only task is to receive the messages and pass them to the Business Logic layer.
- Make sure always to return the correct response code.
- Every step in the service must be logged. Use the logging component generously. Since there is no UI for this service, logging is the only way of figuring out what's going on.