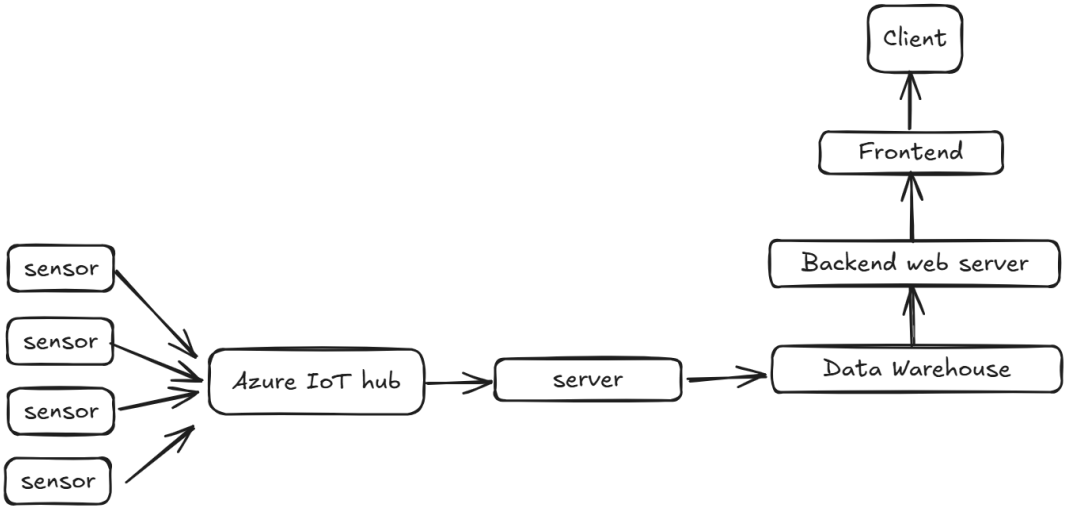
A System for IoT devices

By Lee Shu yun

Simple Diagram



Explanation

Since Azure has been mentioned, I've tailored this response to use Azure services.

Data handling from sensor:

For IoT, sensor data is coming from the sensor side, so using event driven architecture is most suitable for it. There is also an industry standard that outlines the methodology around handling this: the OPC UA PubSub. This will allow the system to be able to ingest data quickly and efficiently from multiple sensors in an event driven way. https://opcconnect.opcfoundation.org/2016/03/opc-ua-is-enhanced-for-publish-subscribe-pubsub/

This can be done in Azure using Azure IoT Hub.

Processing data from Azure IoT Hub:

We can use a server/Azure Function/Databricks to process the sensor data into a standardized format and store into a data warehouse. Which to use requires further discussion, because this depends on how much data is going through per second. If workload is low, a cheap and simple server or Azure Function is fine, if not more dedicated tools to handle data like Databricks is probably required. Needs further consideration and discussion.

Data warehouse:

Data warehouse options can be Azure PostgresSQL Flexible servers. On a later stage when the amount of data grows to unmanageable size even by Postgres, more heavy duty or specialized time series databases can and should be considered. Options will take some time to discuss and finalise. (More explanation below)

Backend web server:

Can be a Spring Boot server, since I'm familiar with it. Its security configurations is robust, trusted and used in government and banking. This server can be free or nearly free to host on Azure.

Frontend Framework:

React, or whichever your team is more familiar with. I know Angular more, but I lean towards React, as React has a lot more people using it so it's easier to find plugins and support from the community if any unusual problems arise.

Maintenance of web services:

In event of backend web services crashing or meeting errors, monitoring and notification alerts is required so we know when things go wrong with the servers. We can use Prometheus & Grafana, or Azure Alerts. Getting Azure support is also essential in my experience.

Thinking Process

The infrastructure has to be able to process and store the time series sensor data, so in order to find out our budget, we should first calculate how many data points our system needs to handle.

5 seconds per data point was mentioned, so I will base the calculations off that.

For every month, the system to be built should be able to process:

12 datapoints per minute x 60min x 24hr x 31days
= 535,680 data points per month per sensor.

Estimating each data point to be 1KB, the system has to handle storing, processing and analysing 6.4GB of data per sensor per year.

With 100 sensors, 640 million data points weighing 640GB is added to our databases per year.

The needs of the sensors can quickly reach the level of big data in the long run. Storing, analyzing and querying that much data in an efficient and timely manner is probably the biggest problem we will need to solve in designing and futureproofing this system, so the data warehouse we choose is very important.

Backend API Data Retrieval Endpoint:

Task: Define a GetSensorData endpoint that retrieves sensor readings specific to a user's company.

Answer:

Example Endpoint URI: https://data.aprisium.com/api/v1/sensor

Example request:
GET /api/v1/sensor HTTP/1.1
Authorization: Bearer PsTBOjba1y4g0zqMGpV3S1
Host: api.auth-azure-server.com
….
{ "companyId" : "C1234567" }

Example HTTP response


```
{
    "companyId": “C1234567”,
    "name": “Client Name Pte Ltd",
    "sensorId": “DS18BXYZ123",
    "value": [
     {
     "temp" : 14.5,
     "temp_units" : “C”,
     "chlorine": 0.9,
     "chlorine_units": "mg/L",
     "iron": 0.3,
     "iron_units": "mg/L",
     "timestamp":"2024-11-19 23:59:05T+0800"
     },
     {
     "temp" : 14.5,
     "temp_units" : “C”,
     "chlorine": 0.9,
     "chlorine_units": "mg/L",
     "iron": 0.3,
     "iron_units": "mg/L",
     "timestamp":"2024-11-19 23:59:10T+0800"
    }
  ]
}
```


Authentication and Authorization

Each user gets their own unique login, and their passwords and details should be salted and encrypted on our databases, amongst other security measures.

Communication and logins between user and system can be handled by the webserver by using OAuth2.

Frontend

Data can probably be displayed in more intuitive forms like a thermometer for temperature for quick reading by anyone, or line charts to show trends in the data. Of course for industrial uses graphical interfaces may not be necessary and get in the way, but it's good UX to anticipate that not every client looking at the data is an engineer. I'll be able to draw up graphics mockups if necessary.

Database

Since the sensor data is following a schema (temperature, iron levels, chlorine levels), I prefer a relational database like PostgresSQL.

Pros of using Postgres:

- open source relational database that is ACID compliant for data with consistent schema
- widely used in the industry due to its flexibility and myriad of features and datatypes, including support for timestamps, large floats, JSON data, multi-version concurrency control, granular user access controls.
- has a planner/optimiser that allows us to optimise queries for analysis. Queries can be incredibly fast.
- future proofing is simpler. There is a rich extension ecosystem around Postgres due to its popularity, so if any new features required of our system in the future, it's likely we'll find an extension that already exists that we can use.
- The query language SQL is simple to learn and powerful, capable of being used in data analysis. Even non-programmers in the company can query the database easily if required.
- Large amounts of data can be stored in a single table in Postgres. The maximum limit is around 32TB by default.

Cons of using Postgres:

- PostgresSQL has scalability issues on VERY massive scale (which usually most companies will never hit. This problem can be easily solved by using cloud services and sharding.
- Querying or manipulating massive amounts of stored data can be slow on PostgresSQL. This can also be solved by sharding.

As for MongoDB, pros of using MongoDB:

- Document based. Any kind of data can be stored
- noSQL, very quick storage of unstructured data. It does support explicit schema as well.
- Query language supports aggregation of data and other useful functionalities great for data analysis.
- Just speed.

Cons of using MongoDB:

- Known reliability issues. Among developers, MongoDB is notorious for causing unsolvable issues. Other devs report intermittent loss of responsiveness, unexplainable spikes in RAM usage, slower query times compared to Postgres.

- Verbose and difficult query language. MongoDB uses its own query language that is not very human readable, which slows down anyone using it. The learning curve is also steeper, and non-programmer colleagues might not be able to access data without developer help. In my previous experience, without an easy query language developers may have to become very expensive support staff instead of building and maintaining the infrastructure.
- Not ACID compliant, missing or corrupted data is possible and have been reported.

Further DB Considerations

If we want to push for performance with lower regard for price and development time, we can consider more heavy duty or specialized data analytics time series database options like: Apache Druid, Azure Cosmos DB, Clickhouse, Apache Doris, Apache Spark, InfluxDB.

DB Table

sensorId – likely to be alphanumerical. Use strings

timestamp – timestamps with time zone. Can be down to fractions of a millisecond.

value – numerical float. Might need another column that specifies the units used

companyId – likely same as sensorId, alphanumerical string.