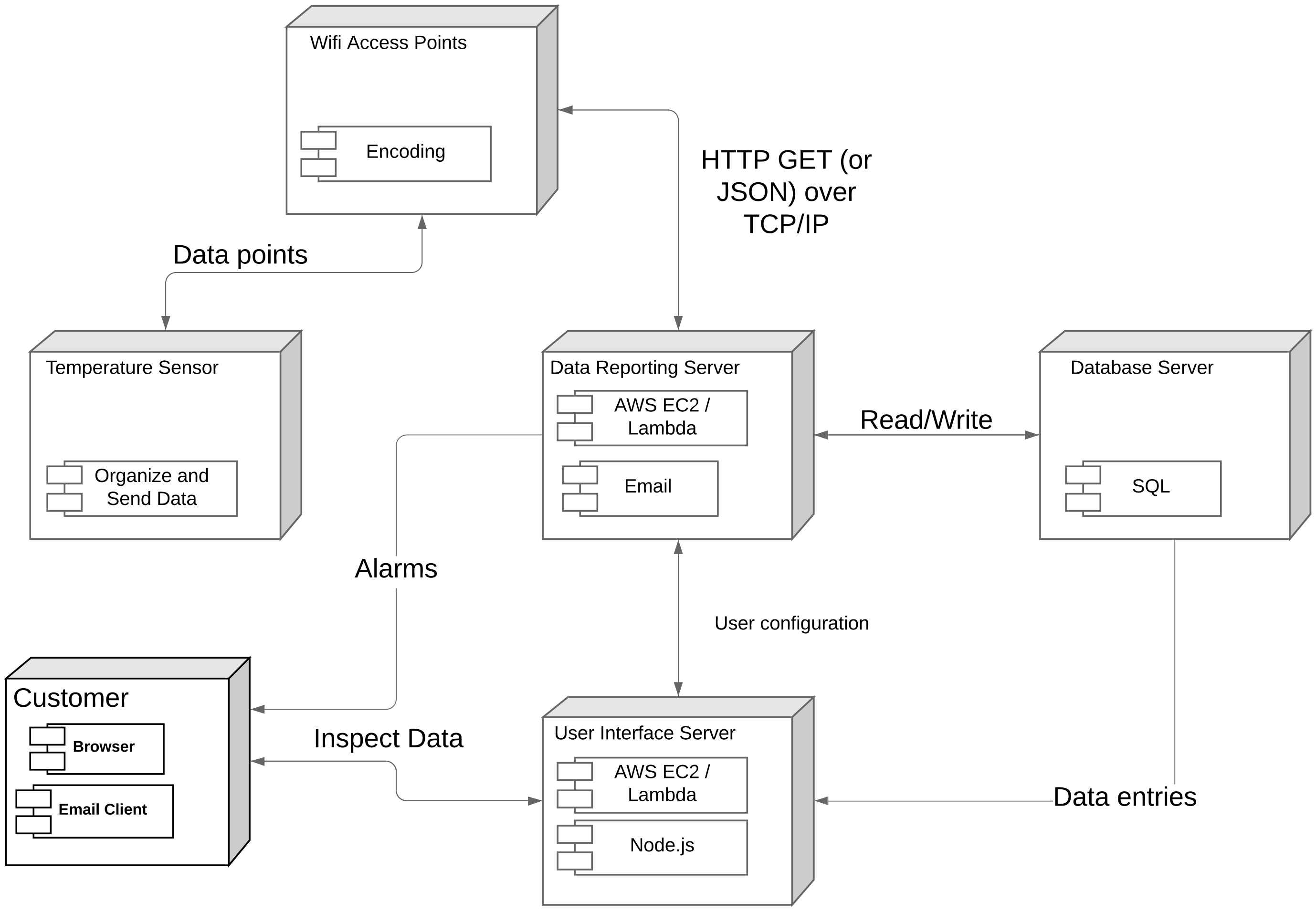
Team 18
Bader Almatouq
Kai Brooks
Andrew Forsman
Jeff Roman
HW 6 - System Design: Modeling

Assumptions/Decisions

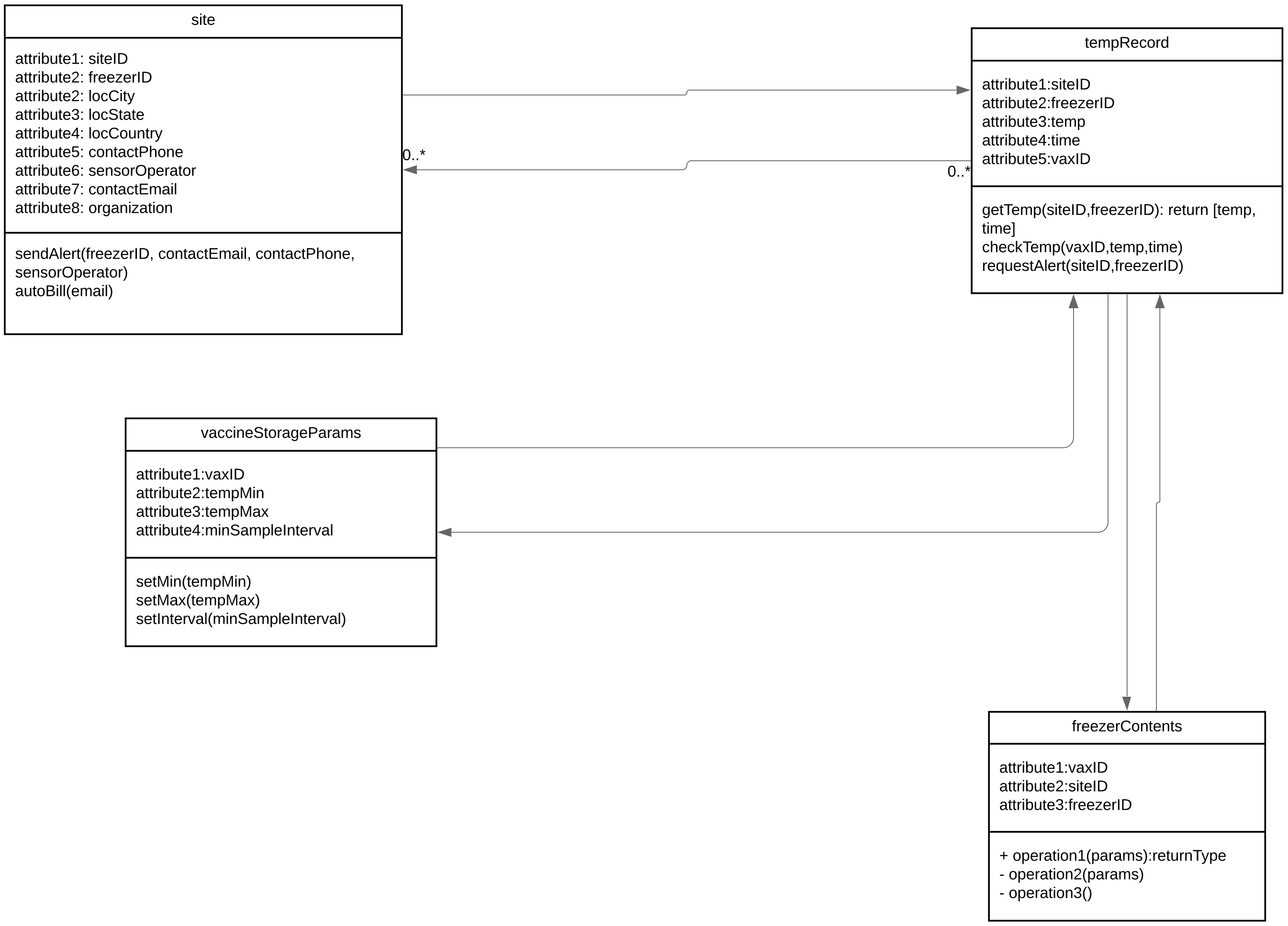
In modeling this system, we assumed and decided certain things in order to model the system. These included:

- Sensors will use Wifi to transmit data to server
- Sensors will take temperature data every 5 minutes and send it to server
- Sensor will stay connected to wifi once
- Server will run on AWS with database using SQL
- Server will assign sensors unique ID’s upon their first connection
- Customer will access temperature current value/history through web
- Customer will be alerted via email regarding temperature outside of acceptable range

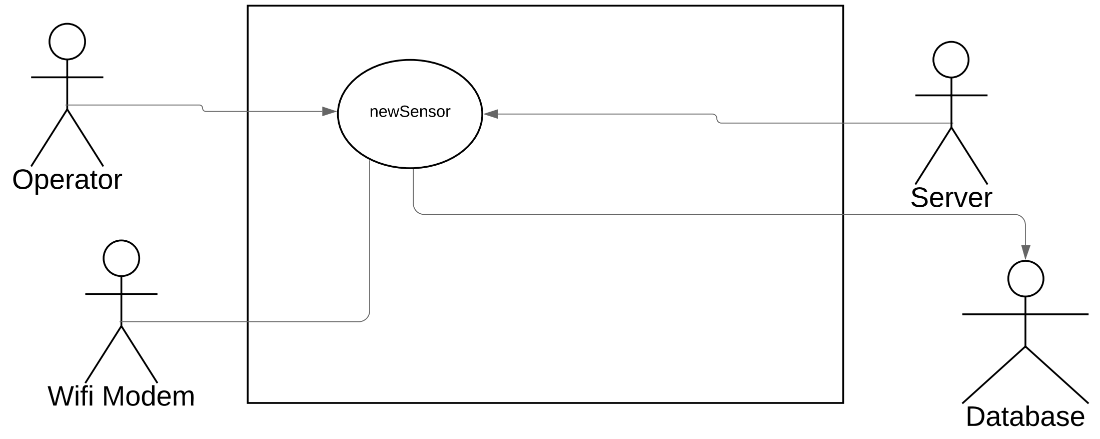
Vaccine Temperature Monitoring Physical View



Vaccine Temperature Monitoring Class Diagram

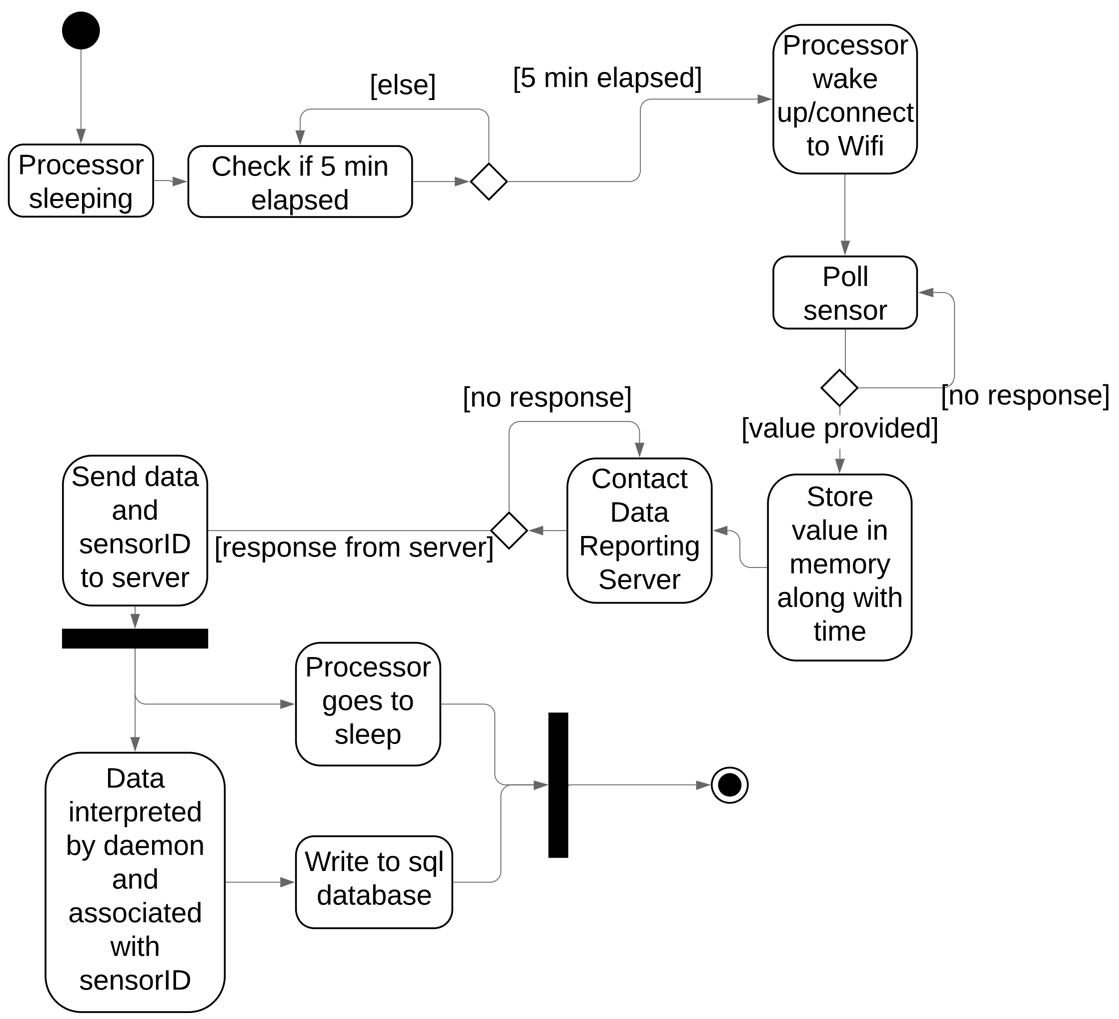


Vaccine Monitoring System Use Case View: Installing a new sensor



Use Case	Install a new sensor.
Actors	siteOperator, server, database, wifi modem.
Description	siteOperator installs new sensor in a freezer with a vaxID. Sensor establishes connectivity with Wifi modem, then connects to server, which searches for it in the database. Not finding it, server sends back an assignation of xsiteID based on IP address and freezerID based on number of freezers at that site already registered. Server then stores this information in database along with vaxID and adds it to its temp check routines.
Stimulus	siteOperator plugs sensor in.
Response	Enable connectivity, register new sensor, incorporate it along with metadata into that site's server workflow.

Vaccine Temperature Monitoring Activity View


Processor sleeping
Check if 5 min elapsed
[else]
[5 min elapsed]
Processor wake up/connect to Wifi
Poll sensor
[no response]
[value provided]
Store value in memory along with time
Contact Data Reporting Server
[no response]
[response from server]
Send data and sensorID to server
Processor goes to sleep
Data interpreted by daemon and associated with sensorID
Write to sql database

Assumptions:

The various servers and databases may or may not be the same, physical server. It's possible to have a single server running three different applications, or three separate servers running one application.

If we apply the 'three servers' strategy we would provision three separate dedicated servers, which gives us more control over hardware specifications and hosting, though is significantly more expensive. However, if we provision servers from AWS or similar, we will often get shared 'server space' with others regardless.

If we use one server, it will run two separate applications, and a database. If the hardware supports it, this is the most economical option, with little impact to performance.

We use AWS EC2 or AWS Lambda. EC2 is an 'always-on' server, which is useful for when we would expect continuous use. Lambda is a serverless, 'on-demand' application that turns on in response to a query, runs the query, and then kills the server. Lambda has the advantage of only charging us for use, however, steady, continual use favors EC2. Alternately, we can use both EC2 and Lambda in conjunction, having most 'normal' operations running on EC2, and firing up Lambda servers on-demand as needed, such as for a response to heavy traffic.

All our servers software stacks run Node.js, as it offers the most flexibility in design and future use. Node.js can call functions to send emails automatically through EmailAlert, interpret database information for the user, and allows for rapid prototyping.

We use a JSON payload as it's simple to interpret, reasonably robust against data corruption, and offers flexible database entries. When we develop future products with increased functionality, JSON structuring allows us to simply add or remove fields from the payload as needed, without needing to transition the database.

Specifications for HTTP Get

The HTTP GET command requests a representation of a specified resource, such as a webpage, image, or other network resource over the HTTP protocol. Typically this is a 'request-only' command, and only retrieves data. It has no body, elicits a response from the server, is safe, is idempotent, cacheable, and allowed in HTML forms.

Notably, with the lack of a body, HTTP GET cannot send data to a server, as could be done with HTTP POST or HTTP PUT.

However, by intentionally malforming the GET request through requesting an abstract URL, the GET request can be used to 'send' content through encoding it in the URL as a request.

We set up the server with a specific, hidden endpoint that interprets these malformed URL's and inserts their contents into the database, as an entry.

For example, the entire server endpoint could be:
http://server.url/endpoint/

Note: Consider the endpoint to be secret. The endpoint can also be masked by using a load-balancing service, which never reveals the 'true' endpoint to the device, which strengthens security.

If the payload is data1_data2_data3, that gets appended to the end of the base endpoint url, resulting in:
http://server.url/endpoint/data1_data2_data3

The server interprets the appended URL as information to write to the database.

Suppose we have a datapoint, 'data1' that can be encoded in plain text, numerically, or some other form realizable through text, including encrypted information, up to a maximum allowed GET size of 2048 characters. We configure the server to read these URLs coming from a specific endpoint, and take them as information to write to its database.

For example, if we wanted to encode a measurement, the payload could look like YYYYMMDD-HHMM-TT, where this is of the format ISO8601 data, followed by a time and temperature measurement. For a measurement of 42 degrees, taken January 2nd, 2001, at 130PM we could encode 20010102-1330-42 as the datapoint we're interested in. The server would write 'January 2nd, 2001', '130PM', and '42 degrees' in the table entry governing this request. We configure the server with a relational database to correctly catalog these events and group them together.

Further datapoints, such as data2, could be other encoded information, such as the serial number of the sensor or other diagnostic information. Any number of data points can be encoded this way (up to the GET limit of 2048 characters), and the server understands them through some arbitrary delineator, such as an underscore. Furthermore, some type of encoded 'header' can be prefixed or appended to any requests, as a deterrent to bad actors trying to manipulate such data.

Because HTTP GET confirms a response from the server, this response can be doubled as a confirmation of successful write (assuming some database diagnostic application exists to verify data integrity).

Generally, we would not use HTTP PUT requests, as some networking hardware strips PUT requests from outgoing communication. This may work in a controlled environment, through is uncertain in complex networking situations.

As a better solution, we could also use HTTP POST as a standard way of delivering payload data, with the POST sending a JSON payload (or similar), containing the relevant information. This has the benefit of using the correctly-intended POST command to write to the database, instead of malforming HTTP GET and interpreting this server-side. It also allows the payload it self to be encrypted, while keeping the HTTP header clear.

Note: we consider below variables to be for illustrative purposes only, and we describe actual variables in use in the Class Diagram.

For example, the above data, as a JSON payload, could be:

```
[{
        "entity": "measurement",
        "id": “(some arbitrary encoded ID)”,
        "fields": {
                “date.year": “2001",
                “date.month": “01",
                “date.day": “02”,
                “time.hour": “13”,
                “time.minute”: “30”,
                “time.zone”: “PST”,
                “sensor.temperature”: “42”
                }
}]
```

It would be encoded in an HTTP POST request, targeted at:
http://server.url/endpoint/

The advantage of the above is that it delineates data correctly from the beginning, which ensures accurate database recording. It allows greater than 2048 characters, and offers flexibility in adding or removing data fields. For instance, if a software error occurs with the JSON payload, it will corrupt a specific data point, versus fundamentally altering the HTTP GET request.
For example, suppose a data corruption occurs and the hour spills out as 0A2F03_CD??1B

In the strict HTTP GET, the URL would be encoded as:

http://server.url/endpoint/20010102-0A2F03_CD??1B30_46

This has two problems. One, if the ? In the URL is not stripped by the network (which would be what happens if it strips PUT requests), it would write 20010102 to the ‘date’ field, 0A2F03 to the ‘time’ field, ‘CD??1B30’ to the ‘temperature’ field, and 46 to whatever database field follows.

If the URL was stripped by the network, it would remove all data past the ??, which would write 20010102 to the ‘date ’field, 0A2F03 to the ‘time ’field, ‘CD ’to the ‘temperature ’field, and have no information past that.

If, for instance, an authentication datapoint was supposed to be sent, this data would never be written, even the data which would be deemed correct.

Now, if the same corruption occurred but with a JSON payload in a POST request:

```
[{
        "entity": "measurement",
        "id": “(some arbitrary encoded ID)”,
        "fields": {
                “date.year": “2001",
                “date.month": “01",
                “date.day": “02”,
                “time.hour": “0A2F03_CD??1B”,
                “time.minute”: “30”,
                “time.zone”: “PST”,
                “sensor.temperature”: “42”
                }
}]
```

Obviously, this payload still contains an error, though the error is captured and does not ‘leak’ into other fields or otherwise compromise the integrity of the rest of the entry. In an isolated incident, this data may still be sufficient to continue normal operation.

Regardless, the individual data fields must be known and decided ahead of time, though their ordering must be specifically coded for the GET method, versus the POST method which fields could be add or removed as needed.

In conclusion, strictly using HTTP GET is possible, and an example URL shown, though HTTP POST using JSON is a more durable system, while still offering the same network mobility.