**ENSE 483 Project System Design Document**

October 28, 2019

Daniel Shevtsov (SID: 200351253)

[1. Introduction 1](#_Toc23098002)

[1.1 Purpose of the system 1](#_Toc23098003)

[1.2 Design goals 1](#_Toc23098004)

[1.3 Definitions, acronyms, and abbreviations 1](#_Toc23098005)

[1.4 References 1](#_Toc23098006)

[1.5 Overview 2](#_Toc23098007)

[2. Architectures of similar systems 4](#_Toc23098008)

[3. Proposed software architecture 5](#_Toc23098009)

[3.1 Overview 5](#_Toc23098010)

[3.2 Subsystem decomposition 6](#_Toc23098011)

[3.3 Hardware/software mapping 8](#_Toc23098012)

[3.4 Persistent data management 8](#_Toc23098013)

[3.4.1 Relational Database 8](#_Toc23098014)

[3.4.2 Time Series Database 9](#_Toc23098015)

[3.5 Access control and security 9](#_Toc23098016)

[3.6 Global software control 10](#_Toc23098017)

[3.7 Boundary conditions 10](#_Toc23098018)

[4. Class Interfaces 10](#_Toc23098019)

[4.1 Sensor – Simulated Data Input and Publish Program 11](#_Toc23098020)

[4.2 Cloud – Publish/Subscribe – TSDB Persistence Program 11](#_Toc23098021)

[4.3 Cloud – HTTP REST API Program 11](#_Toc23098022)

# Introduction

## Purpose of the system

This system implements *Meat UC5.1: Pig Farm Management* as specified in the *Internet of Food & Farm 2020 Use Case Architectures and Overview of the Related IoT Systems*. This system automates management of pig farms using sensors that report various parameters of each pig’s health to supply decision makers such as farmers, slaughterhouse workers, and experts with relevant and actionable information.

## Design goals

The design goals for the system are outlined below:

* Implement a basic software-based IoT system consisting of sensors, a gateway, and a cloud. The sensors publish data readings to the gateway via an established protocol. The cloud subscribes to changes in data readings, persists data, and provides an interface for users to interact with the system via CRUD operations.
* Satisfy the requirements of the envisioned product defined in *UC5.1: Pig Farm Management*.
* Incorporate an Eclipse IoT open source project in some form in the software system.

## Definitions, acronyms, and abbreviations

The following definitions and acronyms may be used in portions of this document:

* **CRUD** – Create, Read, Update, and Delete. These describe the typical read/write operations used to interact with data resources in persistent storage.
* **HTTP** – An application layer protocol in the ISO/OSI model. It is a client-server protocol that allows for the exchange of data in computer networks.
* **REST** – Representational State Transfer. A software architecture for web services that maps CRUD operations to the HTTP protocol verbs POST, GET, PUT/PATCH, and DELETE, respectively.

## References

Please refer to the *ENSE 483 Project Requirements Specification* for the set of requirements that have previously been defined for this project and that this design document is based off of.

The system design is inspired and influenced by the following research papers that have been collected in a literature survey prior to the commencement of the project:

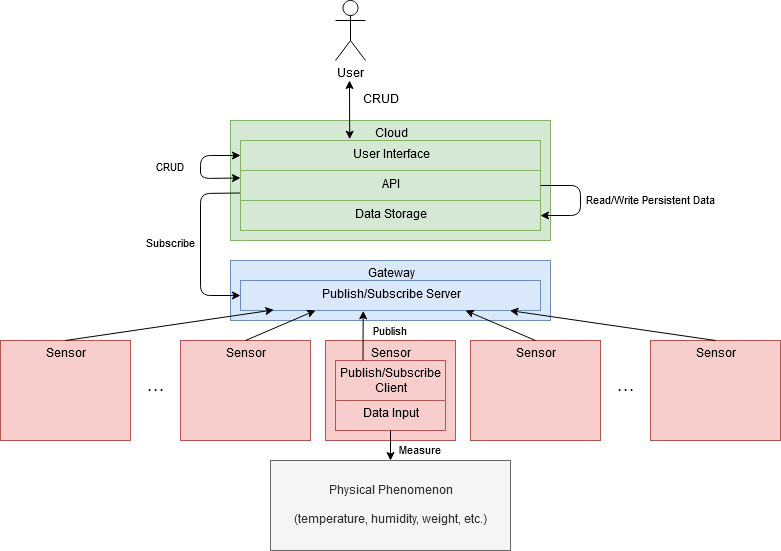
* Krishna, K. L., Silver, O., Malende, W. F., & Anuradha, K. (2017). Internet of Things application for implementation of smart agriculture system. 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 54– 59. Retrieved September 28, 2019, from <https://ieeexplore.ieee.org/document/8058236>.
* Nobrega, L., Tavares, A., Cardoso, A., & Goncalves, P. (2018). Animal monitoring based on IoT technologies. 2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany), 1–5. Retrieved September 28, 2019, from <https://ieeexplore.ieee.org/document/8373045>.
* Pan, L., Xu, M., Xi, L., & Hao, Y. (2016). Research of livestock farming IoT system based on RESTful web services. 2016 5th International Conference on Computer Science and Network Technology (ICCSNT), 113–116. Retrieved September 28, 2019, from <https://ieeexplore.ieee.org/document/8070130>.
* Sanghavi, J., Shah, A., Rane, S., Shah, N., Nayak, S., Kadam, P., & J., D. (2018). Agricultural Productivity Enhancement System & Livestock Management using Internet of Things. 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC), 1–5. Retrieved September 28, 2019, from <https://ieeexplore.ieee.org/document/8479463>.
* Sun, H., Zhu, Q., Ren, J., Barclay, D., & Thomson, W. (2017). Combining Image Analysis and Smart Data Mining for Precision Agriculture in Livestock Farming. 2017 IEEE International Conference on Internet of Things (IThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 1065–1069. Retrieved September 28, 2019, from <https://ieeexplore.ieee.org/document/8276884>.

As well, the following blog post describing the design and implementation of an IoT system for home temperature monitoring has been used as an inspiration:

* Mechling, G. (2019, January 26). IoT - Home sensor data monitoring with MQTT, InfluxDB and Grafana. Retrieved October 25, 2019, from <http://nilhcem.com/iot/home-monitoring-with-mqtt-influxdb-grafana>.

## Overview

The general system design should adhere the general IoT application architecture described below:



*Fig 1. General IoT Application Architecture*

The system consists of the following components, starting from the bottom of the stack:

* Physical phenomena of interest are each measured by a dedicated sensor, typically as an analogue measurement that is converted to a discrete digital value.
* Each sensor includes a publish/subscribe (pub/sub) client that communicates via the established protocol and periodically publishes its measured values to the specified pub/sub server.
* A gateway device at the edge serves as the centralized pub/sub server to which all of the sensors publish their readings periodically. The server allows subscribers to listen to changes in measurements submitted by publishers.
* The cloud includes an API that provides read/write access to persistent data that is persisted in data storage. Data storage consists of a relational database (providing long-term storage of user information and domain models such as pigs and their associated sensors) as well as a time-series database (providing long-term storage of subscribed pub/sub data from the gateway).
* The cloud provides a user interface to enable end-users to perform CRUD operations against the API.

# Architectures of similar systems

Similar IoT systems in the agricultural domain have been researched as part of a literature survey. In the paper *Internet of Things Application for Implementation of Smart Agriculture System*, the authors implemented a wireless robot for monitoring and controlling parameters of crop health in a farm. The robot was implemented as follows:

* Various physical sensors were used for measuring a number of parameters including temperature, humidity, acidity, and UV rays.
* Physical actuators including a DC motor, a speaker, and a water sprinkler were also included in the robot.
* Sensors and actuators would communicate with a Raspberry Pi Model 2B that would serve as the controller or gateway.
* A ZigBee transmitter and Wi-Fi modem were connected to the Raspberry Pi to allow it to communicate with an Android phone and personal computer that provide a user interface for users to control the robot. As such, the Android phone or personal computer serve as the “cloud” in this implementation.

The authors of the system found the accuracy of measurements and the activation of actuators in response to be accurate and satisfactory for the use case of automating crop monitoring. The architecture described by the authors does not fit the requirements of this project as the system is required to be software-based, i.e., without actual hardware components. As well, the authors’ project required the use of actuators in response to abnormal sensor readings, while this project does not incorporate actuators of any kind.

In the paper *Research of Livestock Farming IoT System Based on RESTful Web Services*, the authors describe a software system for handling the monitoring and control of livestock farming using a RESTful web service. The authors describe the implementation of an IoT system with a focus on the cloud component. The system describes is as follows:

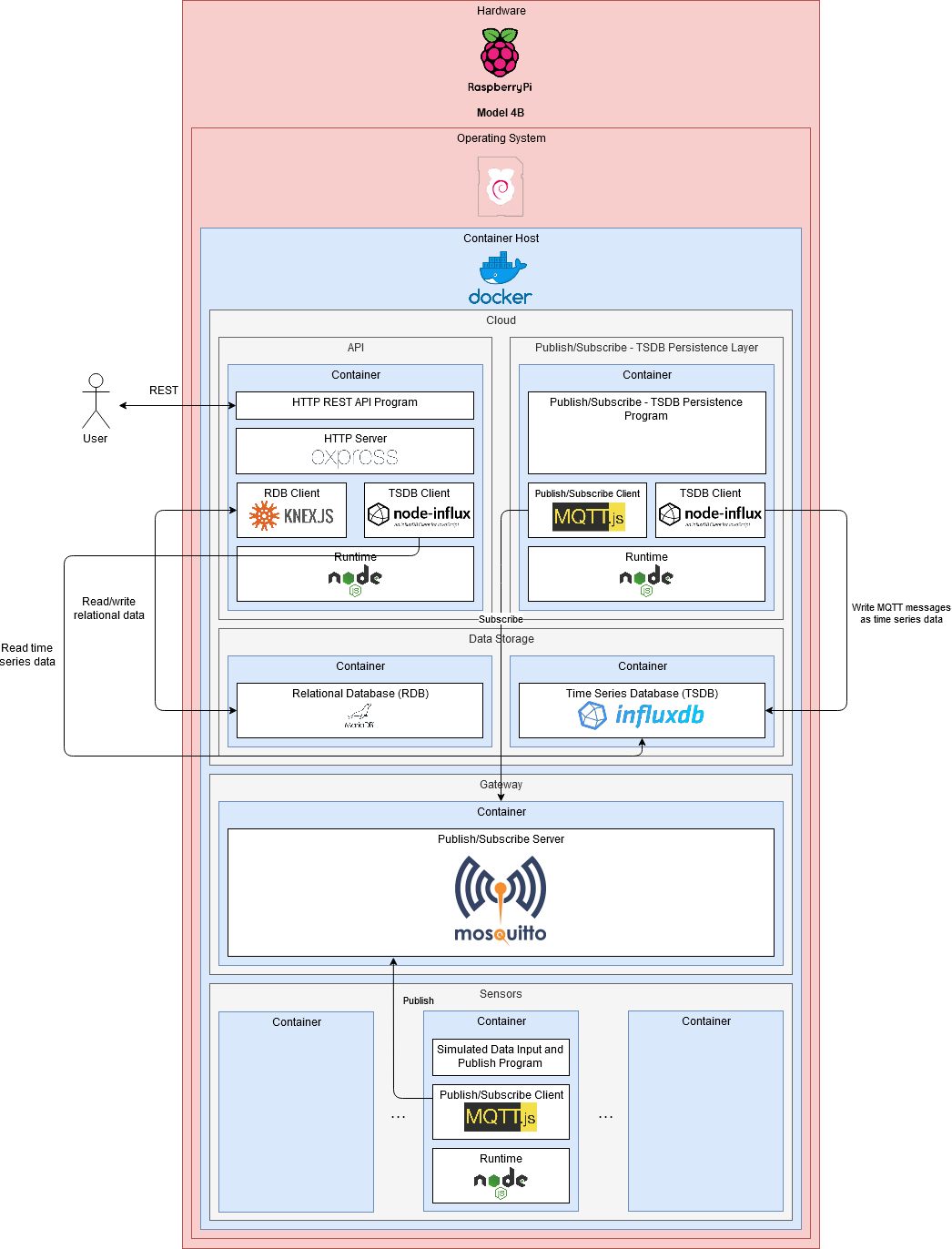
* Sensors would communicate via the Modbus protocol with a control gateway to transmit measurements.
* The control gateway would communicate with a central management server or cloud via TCP/IP using either Ethernet or Wi-Fi.
* The cloud would provide a control layer and a representational layer.
* At the control layer, data management would be implemented including logging, caching, persistence of data in a database, and a file system. As well, device management would be implemented including data processing, hardware processing, and command processing. Finally, resource management would be implemented as a pool of resources, each representing one kind of sensors in the system.
* At the representational layer, the persisted data from the control layer would be exposed as resources with which users can interact over HTTP via CRUD operations.

Although the software system described by the authors disregards design aspects of sensors and the gateway, the design of the cloud component is relevant to this project as well. The RESTful API component in particular has also been implemented in the design of this project.

# Proposed software architecture

## Overview

The proposed software architecture, based off of the general IoT application architecture shown in *Fig 1* is described below:



*Fig 2. Proposed software architecture overview*

At a high level, the system consists entirely of software components hosted on a single hardware device (Raspberry Pi 4). The system is composed of three sub-systems, namely sensors, gateway, and cloud.

## Subsystem decomposition

A description of each component is given below. Components that must be implemented as part of this project are marked with **\***.

* **Hardware – Raspberry Pi Model 4B** (<https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>)
  + All software components of the system are hosted on a Raspberry Pi single-board computer, namely version 4, model B. The main specifications of this device are listed below:
    - CPU: Quad-core ARM v8 (64-bit) @ 1.5GHz Broadcom BCM2711
    - RAM: 4GB LPDDR4-3200 SDRAM
    - HDD: 240GB Kingston A400 SSD
* **Operating System – Raspbian Buster** (<https://www.raspberrypi.org/downloads/raspbian/>)
  + The operating system on the Raspberry Pi is the recommended Raspbian Buster Linux distribution, version September 2019.
* **Container Host – Docker** (<https://www.docker.com/>)
  + The entirety of the system (sensors, gateway, cloud) are deployed to a single hardware device due to the lack of physical sensors and access to a cloud service. Docker is an Operating System-level virtualization technology that allows software to be packaged into containers that run on the host machine’s operating system. Containers can be provisioned their own networks and storage volumes that allow the components in each layer to exist in isolation of the other layers. As such, the separation of sensors, the gateway, and the cloud can be simulated by isolating Docker containers that run the software for each component.
* **Sensors**
  + Each sensor is deployed as a Docker container. Sensors publish measurements of physical phenomena via the MQTT protocol (<http://mqtt.org/>) to a gateway. A sensor consists of the following software components:
    - **Runtime – Node.js** (<https://nodejs.org/>)
      * Node.js is a JavaScript runtime that allows JavaScript to be used as a general-purpose programming language, similar to Python, Java, Ruby, etc. Each sensor includes the Node.js runtime to allow for the execution of user programs. The Node.js runtime is required for simulating physical sensor measurements and interfacing with an MQTT client, however, it would not be required for a physical sensor that has the capability to communicate over MQTT.
    - **Publish/Subscribe Client – MQTT.js** (<https://github.com/mqttjs/MQTT.js>)
      * MQTT.js is an MQTT client library for JavaScript that allows publishing and subscribing to messages. It is used by sensors to publish measurements to a specified MQTT server (in this case, the gateway).
    - **\* Simulated Data Input and Publish Program**
      * A custom JavaScript program that simulates sensor measurements and publishes them as messages at the specified topic to the specified channel.
* **Gateway**
  + The gateway is deployed as a Docker container. A gateway includes an MQTT server to which publishes (sensors) publish their measurements at regular intervals. The gateway consists of the following software components:
    - **Publish/Subscribe Server – Eclipse Mosquitto** (<https://mosquitto.org/>)
      * Eclipse Mosquitto is an Eclipse IoT open source software project that implements an MQTT message broker or server compliant with MQTT protocol versions 3.1, 3.1.1, and 5.0.
* **Cloud**
  + The cloud is deployed as a set of Docker containers. The cloud consists of the following software components:
    - **Data Storage**
      * There are two types of data that the cloud should persist in databases:
        + **Relational Database (RDB) – MariaDB** (<https://mariadb.org/>)

MariaDB is deployed as a Docker container. MariaDB is an open-source community-developed fork of the MySQL relational database. It is used to store persistent relational data such as users, user-entered data, pigs, sensors associated with pigs, and other system settings.

* + - * + **Time Series Database (TSDB) – InfluxDB** (<https://docs.influxdata.com/influxdb/>)

InfluxDB is deployed as a Docker container. InfluxDB is a time series database that is optimized for the storage and querying of time series data (pairs of timestamps and values). It is used to persist sensor measurements so that historical data may be queried.

* + - **Publish/Subscribe – TSDB Persistence Layer**
      * The Publish/Subscribe – TSDB Persistence Layer is deployed as a Docker container. This layer persists messages published to the MQTT server to the InfluxDB time series database. It consists of the following software components:
        + **Runtime – Node.js** (<https://nodejs.org/>)
        + **Publish/Subscribe Client – MQTT.js** (<https://github.com/mqttjs/MQTT.js>)
        + **TSDB Client – node-influx** (<https://github.com/node-influx/node-influx>)

Node-influx is an InfluxDB client library for JavaScript that allows querying the time series database. It is used to persist MQTT messages whenever they arrive at the MQTT server.

* + - * + **\* Publish/Subscribe – TSDB Persistence Program**

A custom JavaScript program that handles the logic of subscribing to the specified MQTT server and persisting messages in the specified InfluxDB TSDB.

* + - **API**
      * The API is deployed as a Docker container. The API provides an interface for users to interact with the system using CRUD operations. It is a REST API that users interact with over the HTTP protocol. It consists of the following software components:
        + **Runtime – Node.js** (<https://nodejs.org/>)
        + **TSDB Client – node-influx** (<https://github.com/node-influx/node-influx>)

Node-influx is used to query InfluxDB to read time series data.

* + - * + **RDB Client – knex.js** (<http://knexjs.org/>)

Knex.js is a JavaScript Relational Database client and Structured Query Language (SQL) query builder. It is used to create and manage the Relational Database schema and perform CRUD operations on persistent data.

* + - * + **HTTP Server – Express** (<https://expressjs.com/>)

Express is a JavaScript HTTP server and framework for web applications. It is used to implement the REST API.

* + - * + **\* HTTP REST API Program**

A custom JavaScript program that implements the various API endpoints and the logic to connect to and query databases.

## Hardware/software mapping

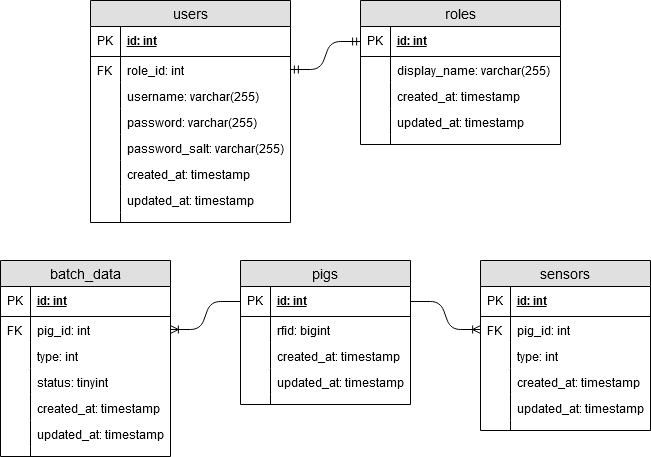
As described in section **3.2**, the majority of the components are implemented as software hosted on a single hardware device. This is done due to the lack of physical devices and access to a real cloud service. In a real production system, there would be separate hardware for sensors, gateways, and the cloud.

## Persistent data management

As described in section **3.2**, there are two types of persistent storage that the system must manage, namely a Relational Database (RDB) and a Time Series Database (TSDB). The sections below describe the design for each.

### Relational Database

Below is the schema for the relational database:



*Fig 3. Relational Database Entity Diagram / Schema*

A description of each table is given below:

* **users** – Users that log into and interact with the system.
  + **id** – Unique, incrementing identifier.
  + **role\_id** – Foreign key to **roles** table defining the role of this user.
  + **username** – Unique username entered by the user to log into the system.
  + **password** – Hashed and salted user password.
  + **password\_salt** – Salt generated when user is created and used for hashing their password.
  + **created\_at** and **updated\_at** – Timestamps that are updated when a user is created and updated.
* **roles** – Roles that a user can have.
  + **id** – Unique, incrementing identifier.
  + **display\_name** – A friendly name for the role to display to users.
  + **created\_at** and **updated\_at** – Timestamps that are updated when a role is created and updated.
* **pigs** – Pigs in the farm that are being tracked by the system.
  + **id** – Unique, incrementing identifier.
  + **rfid** – Unique RFID tag identifier.
  + **created\_at** and **updated\_at** – Timestamps that are updated when a user is created and updated.
* **sensors** – Sensors that are associated with pigs. Note that sensor measurements are not stored in the RDB and are instead queried from the TSDB.
  + **id** – Unique, incrementing identifier.
  + **pig\_id** – Foreign key to **pigs** table defining the pig that the sensor is associated with.
  + **type** – Enumeration of sensor type (temperature, humidity, weight, etc.).
  + **created\_at** and **updated\_at** – Timestamps that are updated when a user is created and updated.
* **batch\_data** – User entered supplementary pig health parameters in addition to sensor measurements.
  + **id** – Unique, incrementing identifier.
  + **pig\_id** – Foreign key to **pigs** table defining the pig that the sensor is associated with.
  + **type** – Enumeration of batch data type (boar taint, barn characteristics, etc.).
  + **status** – Enumeration of batch data status (unknown, good, bad, etc.).
  + **created\_at** and **updated\_at** – Timestamps that are updated when a user is created and updated.

### Time Series Database

The time series database is used exclusively for persisting and reading sensor measurements. Following is the schema for sensor measurements:

* **measurement: string** – The type of measurement (temperature, humidity, weight, etc.).
* **tags**: **Object<string, string>** – Metadata associated with the measurement.
  + **pig\_id: number** – Unique ID of the pig associated with the sensor that performed the measurement.
* **fields: Object<string, \*>** – Value of measurement at the current timestamp (timestamp is populated automatically).

## Access control and security

Following is the access matrix for the CRUD operations of the system by role:

|  |  |  |  |
| --- | --- | --- | --- |
| **Action / Role** | **Expert** | **Worker of**  **Slaughterhouse** | **Farmer** |
| **Login & Logout** | ✓ | ✓ | ✓ |
| **View users** | ✓ |  |  |
| **Create/update/delete users** | ✓ |  |  |
| **View pigs** | ✓ | ✓ | ✓ |
| **Create/update/delete pigs** | ✓ |  |  |
| **View sensors associated with pig** | ✓ | ✓ | ✓ |
| **Create/update/delete sensors** | ✓ |  |  |
| **View batch data** | ✓ | ✓ | ✓ |
| **Create/delete batch data** | ✓ |  |  |
| **Update batch data** | ✓ | ✓ | ✓ |
| **View roles** | ✓ |  |  |
| **Create/update/delete roles** | ✓ |  |  |

*Table 1. Access Matrix*

Following are additional security and access control design considerations:

* The API authentication mechanism to be used is access tokens in the JWT format (<https://jwt.io/>).
* User passwords should be salted and hashed.
* End-to-end encryption may be provided by restricting API communication to HTTPS (over SSL/TLS).

## Global software control

Users control the software over an HTTP REST API. Users request resources exposed at defined API endpoints (e.g., pigs, sensors, etc.) by creating and sending an HTTP request with the appropriate verb (GET to read, POST to create, PUT to update, and DELETE to delete) and headers (including their access token for authentication). The server processes the request and returns an HTTP response to the user with either the requested resource/s or a confirmation that the resource was modified successfully. The server should set the HTTP response code appropriately based on whether the request was completed successfully or not.

## Boundary conditions

The system contains various components that may be selectively turned on or shut down. The following list describes the effects of turning off each component:

* If an individual sensor is turned off, it stops publishing measurements to the MQTT server and those measurements are no longer persisted in the RTDB. As a result, users would see a gap in measurements during the time frame that the sensor is turned off.
* If the gateway is turned off, measurements for all sensors are no longer persisted in the RTDB. As a result, users would see a gap in measurements for all sensors during the time frame that the gateway is turned off.
* If the cloud is turned off, users become unable to interact with the system via the API. As well, there would be gaps in persisted RTDB measurements.

Each component is expected to have the ability to be individually restarted without requiring the entire system to be restarted. A restarted component should return to normal operation immediately without additional configuration.

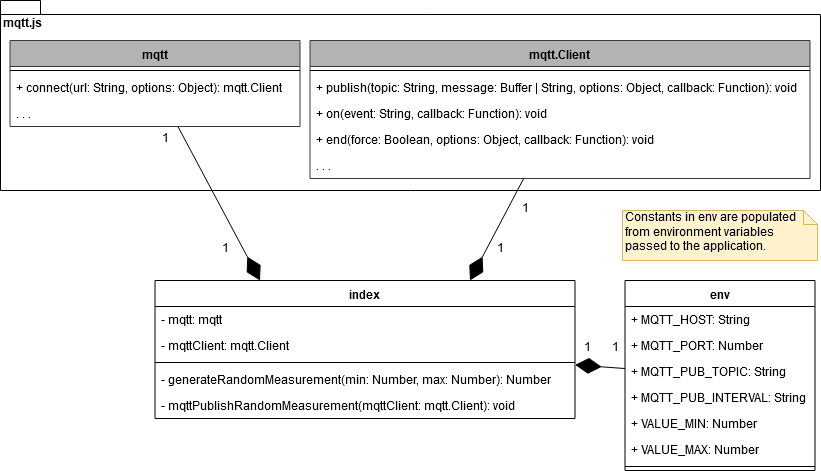
# Class Interfaces

As described in section **3.2**, there are three software components to be implemented by this system. These are:

1. Sensor – Simulated Data Input and Publish Program
2. Cloud – Publish/Subscribe – TSDB Persistence Program
3. Cloud – HTTP REST API Program

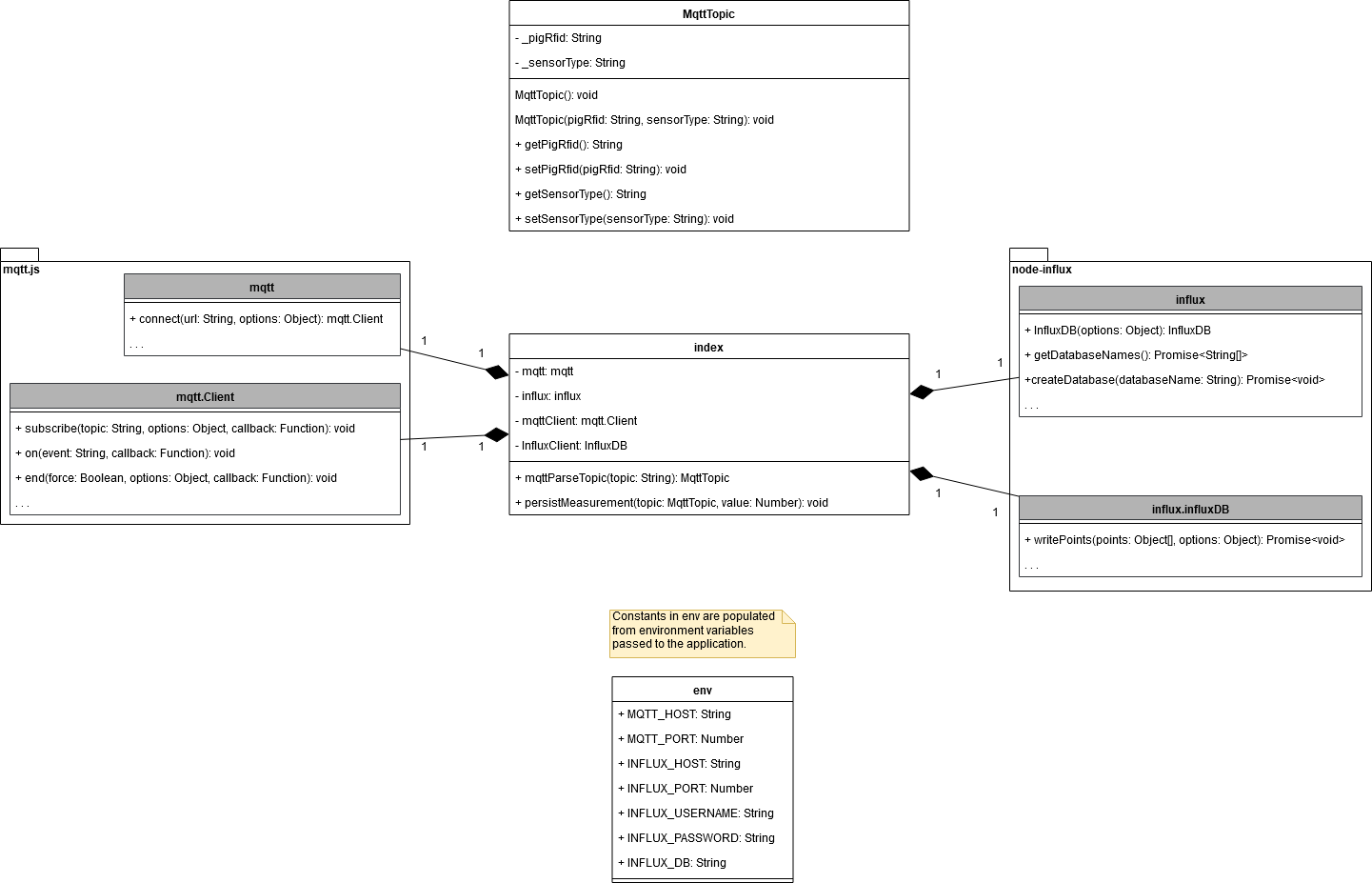
The following sections describe the class interface of each of the components above, respectively, as UML class diagrams.

## Sensor – Simulated Data Input and Publish Program



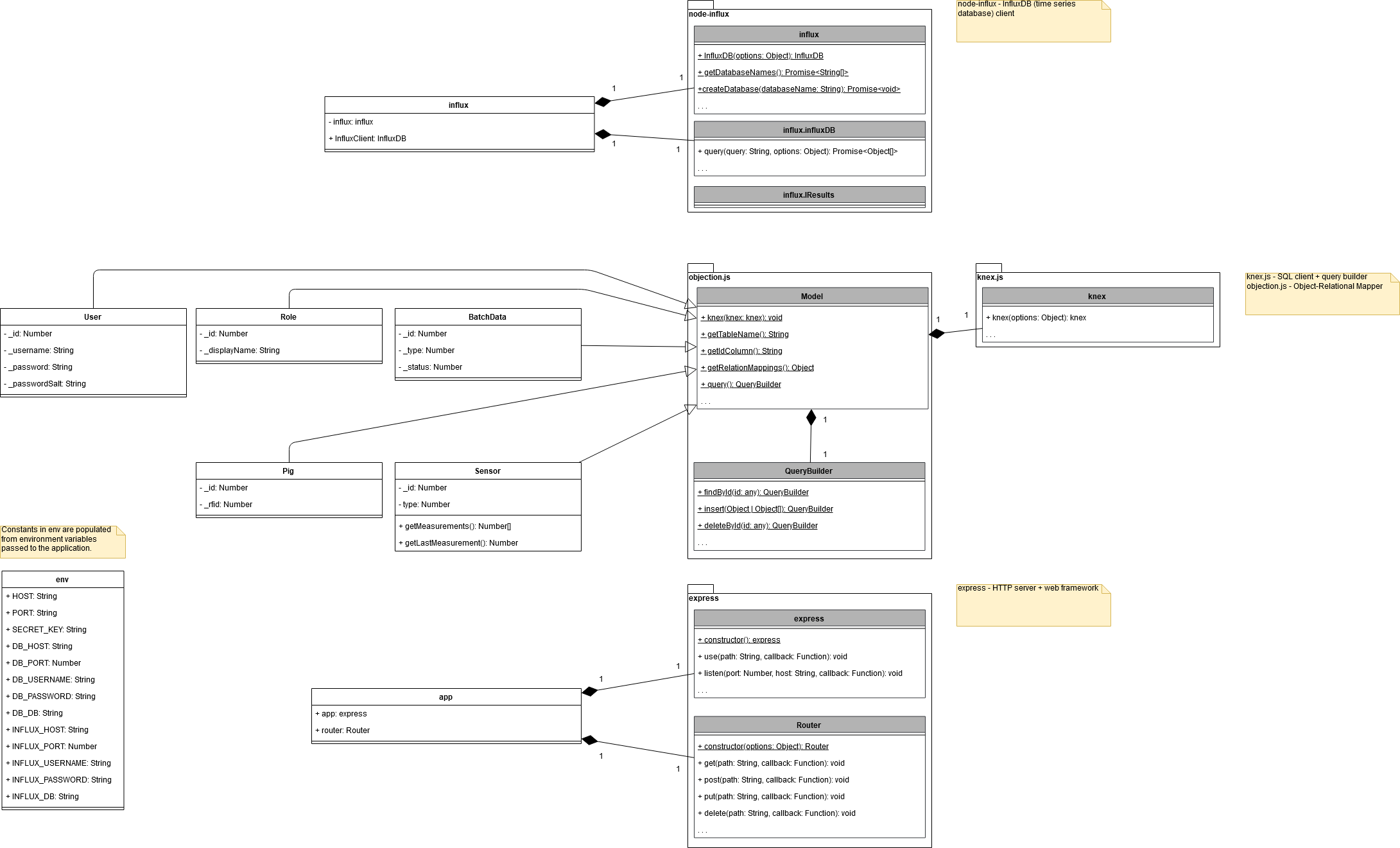
*Fig 4. Sensor – Sensor - Simulated Data Input and Publish Program - UML Class Diagram*

## Cloud – Publish/Subscribe – TSDB Persistence Program



*Fig 5. Sensor – Cloud – Publish/Subscribe – TSDB Persistence Program - UML Class Diagram*

## Cloud – HTTP REST API Program



*Fig 6. Sensor – Cloud - HTTP REST API Program - UML Class Diagram*