Distributed Environmental Monitoring System

Riverscout

Conor Farrell

Bachelor of Science (Honours) in the Internet Of Things

Project Description

Riverscout aims to provide near real-time analytics on environmental phenomena such as water levels, water temperature and air quality to provide data for commercial and recreational users.

While the initial focus was on rivers, Riverscout has since become a generic sensor visualisation platform and these considerations have been taken into account with the system design. Riverscout will use cheap low power gauges to record and transmit data using a LPWAN protocol (in this case Sigfox). Then the data will be processed, stored and made available to users through an API and in the future, via a web application.

The current project is intended for system administrators and not end-users as it allows the creation and deletion of devices, device groups and readings.

Riverscout is not limited to rivers, despite the name it can display different types of sensor values such as air quality, crop health and traffic congestion. Once a new type of gauge has been added, the system can be adapted to display it.

Project Aims and Objectives

The overall goal of Riverscout is to allow near real time of various aspects of the environment. Over the course of the project, the aims have changed slightly, beginning with river water quality, temperature and water height level and expanding to account for different types of sensors towards the end.

My goal for this system is to allow low-cost sensors to be installed in far greater locations than what is currently available to provide individuals, businesses and researchers with more granular data about their local environment.

Why this area was chosen

After completing a work placement in the TSSG (Telecommunications Systems and Software Group) working in energy systems, I decided to focus on an environmentally oriented subject area.

The focus on rivers came from a personal interest in whitewater kayaking, where it is helpful to know a river's current water level in advance of planning trips. Knowing the water level determines the 'runnability' of a river which often has an acceptable range. For certain rivers, if the level is outside of this range it can pose a hazard. If the level is too low rocks become more exposed and if the level is too high some 'features' become obscured and difficult to safely navigate.

Existing systems to graph water levels have been created. One such system is RiverSpy (http://riverspy.net) which uses both its own gauges and data from the Office of Public Works (http://waterlevel.ie/). It was from this that the name 'Riverscout' was created. However this system does not cover a wide range of rivers including the Colligan River in Waterford - prompting me to base my project on this area. W.I.T Kayak Club have wanted a level gauge placed on this river for years and previous efforts to contact the OPW have not been successful as they refuse to install gauges purely for recreational use.

In order to qualify as a final year project, the subject area needed to be expanded. My course leader suggested the idea of water pollution monitoring which I decided to

pursue. However, a lack of suitable sensors prevented me from achieving this goal. See ‘Sensor Limitations’

My supervisor suggested that I create a generic platform instead of one designed to serve only one purpose. This required some planning with regards to the system design, see ‘API Design Considerations’
With these new ideas taken into account, Riverscout became less focused on just rivers and hopes to become a central platform for all types of sensors.

Technologies Used

Overview

Riverscout is built with modern technologies such as OpenAPI version 3, nodeJS version 10, MongoDB 4.0 and Vue 2. When selecting technologies I was partially influenced by the work I completed while I was a backend developer during placement but where possible I selected newer versions of these tools which was done for future proofing.

A key part of the system is a Low Power Wide Area Network (LPWAN) called Sigfox (https://www.sigfox.com/en) that allows small amounts of periodic data to be sent with very little power usage and good coverage. A more in-depth explanation of this technology is below.

The system consists of three main parts, the gauge that senses the environment, a backend server and a front end client. Each part is explained below.

Sensor Limitations

Over the course of the initial research into pollution detection, I identified a number of problems that would prevent me from reaching this goal with the main one being a lack of available sensors.
The goal of Riverscout was not to develop new kinds of sensors but to instead utilise existing technology to create a platform.

Water Pollution

To detect water pollution levels I focused on nitrate, nitrite and pH.
The EPA (Environmental Protection Agency) test for water pollution according to the following document:
http://www.epa.ie/pubs/reports/water/waterqua/integratedwq2012/Appendices%20Text%202012.pdf :

- Ammonia
- Bacteria presence testing
- Biochemical Oxygen Demand
- Biological Quality Rating
- Chloride**
- Colour
- Conductivity*
- Dissolved Oxygen*
- Dissolved Inorganic Nitrogen
- Nitrate/Nitrite
- pH*
- Phosphorous
- Temperature*

* can be measured using a solid-state sensor and such sensors are cheaply available
** price of sensor is expensive (€504 at time of writing!).

The following document (https://www.westminster.edu/about/community/sim/documents/SDeterminingtheAmountofNitrateinWater.pdf) shows a sensor to measure nitrate but states that it needs to be "*soaked in the Nitrate High Standard solution for approximately 30 minutes prior to use"* and also needs to be stored in a dry place when not used.
This is not acceptable for Riverscout as these sensors need to last for multiple years in direct contact with water and remain accurate for that duration.

Another existing solution is a device that uses test strips coated with chemicals and a spectrometer to determine the amount of nitrate. Such a solution for monitoring aquariums can be seen here (https://www.seneye.com/). Once again, this is not suitable as the test strips need to be replaced and a spectrometer will draw too much power from the gauge batteries as well as being too fragile and expensive.

There is a sensor to measure 'Oxidation-Reduction Potential' which is a popular measure in water quality used by horticulturists and aquarium owners. However this

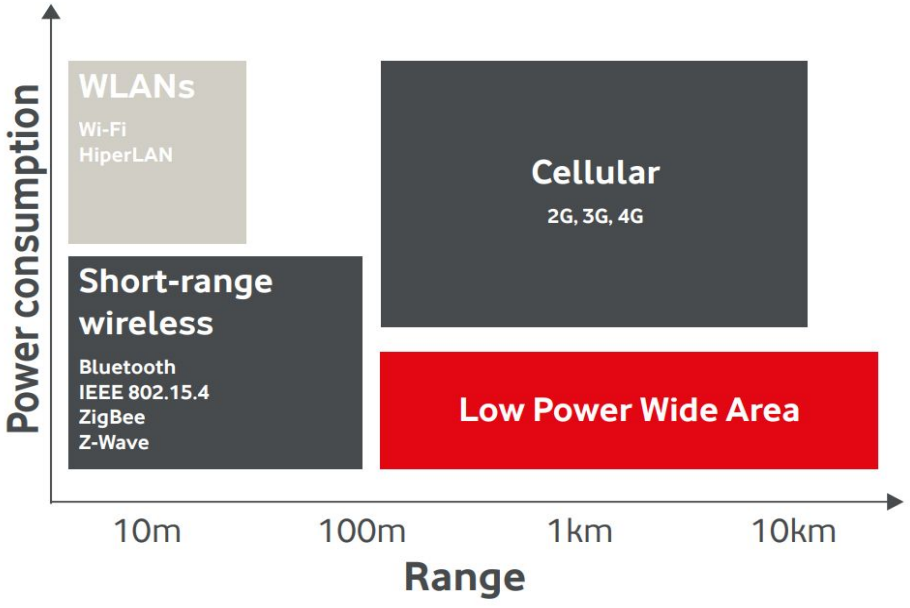
device is also costly, see: https://www.dfrobot.com/product-1071.html for an example sensor. Also that sensor is not fully waterproof and cannot be submerged in water for too long as it becomes inaccurate and needs to be cleaned with acid sharing this problem with the nitrate sensor above (see:https://www.dfrobot.com/wiki/index.php/Analog_ORP_Meter(SKU:SEN0165) under 'Precautions')

Overall, measuring water pollution using continuously immersed solid state sensors is currently impossible but in the future if such a sensor becomes available, Riverscout can adapt.

LPWAN / Sigfox explained

For applications with a tight power budget or only a need for a small message size, using normal protocols like 3G and 4G is not recommended. Special protocols have been developed to fill this need such as: LoRa, Sigfox, LTE-M and Narrowband IoT. All of these differ in implementation but the end goal is the same.

These systems are not designed for constant high bandwidth applications but instead are more suited to sensor networks where a small amount of data gets transferred infrequently in set intervals.



Source: Vodafone Business whitepaper: "Narrowband-IoT: pushing the boundaries of IoT" available at https://www.vodafone.com/business/iot/managed-iot-connectivity/nb-iot

From the above image, we can see that both LPWAN and cellular have been designed to offer long range. For applications where low power consumption is key, we are also restricted to a small message payload. In certain situations, a tradeoff

between power usage and message size is acceptable, therefore making LPWAN the only choice. Riverscout fits into this category as the gauges will only transmit once per hour and send less than 10 bytes of data each time.

Sigfox competes with multiple other LPWAN protocols such as NB-IOT, LTE-M and LoRa. NB-IOT and LTE-M use technology derived from 4G to offer an increased message size, allowed transmissions and security. Since they operate within a licensed radio spectrum (as opposed to using the unlicensed ISM band), they are also not subject to the same 1% duty cycle limitations as Sigfox and LoRa[1, 2]. This means that a device using NB-IoT can transmit more frequently and with more data.

There are other constraints with Sigfox. A message payload can be up to 12 bytes (96 bits) and has a 26 byte overhead [3]. The devices also need to be optimised for low power consumption in order to last for a long time on battery.

[1]ComReg: 'Permitted Short Range Devices in Ireland', page 35 https://www.comreg.ie/media/dlm_uploads/2015/12/ComReg0271R9.pdf
[2] Vodafone: "Narrowband-IoT: pushing the boundaries of IoT" whitepaper, page 6
[3] Introduction to the Sigfox protocol, Sigfox: https://www.youtube.com/watch?v=_aTzrOxIroY

Device Level - The Gauge

Riverscout uses a Pycom SiPy (https://pycom.io/product/sipy/) with a US-100 ultrasonic sensor (https://www.adafruit.com/product/4019) and a DS18B20 temperature sensor. The SiPy is an internet enabled microcontroller that connects to the Sigfox network and can be programmed using a specialized variant of Python called MicroPython.

The SiPy runs on an Espressif ESP32 System On Chip and offers the following hardware (adapted from official spec sheet):

- 32 bit CPU
- Multi thread support through MicroPython
- Separate low power co-processor to handle deep sleep
- Multiple interfaces (I2C, SPI, UART, DAC, ADC)
- Built in support for Wifi and Bluetooth and Sigfox.

For a full list of specifications, refer to official datasheet:
https://pycom.io/wp-content/uploads/2018/08/sipy-specsheet.pdf

Note about demo code

The demo code differs slightly from the main code in that it uses a potentiometer in place of the ultrasonic sensor because of a bug with the serial port of the SiPy. This

occurs in multiple revisions of the device firmware and causes the SiPy to hang while sending a Sigfox message with the serial port connected.
It may be related to this issue:
https://forum.pycom.io/topic/2607/socket-send-stops-working-sigfox

I had another issue with using the OneWire bus that had the same effect (which was resolved with an older firmware revision). For the demonstration I decided that temperature would be more important and easier to showcase so I prioritised that. I reported the issue to Pycom via their GitHub here:
https://github.com/pycom/pycom-micropython-sigfox/issues/295

US-100 ultrasonic sensor

The US-100 is an ultrasonic distance sensor that measures distance by emitting 40kHz pulses and measuring the time it takes to detect them reflecting back [1]. It has a range of between 20 to 450 cm, however it loses accuracy after 250cm. It runs on between 3 and 5 V DC and is easy to interface with the SiPy. The sensor is an improved version of the 'HC-SR04', a popular choice in hobby projects [2].
I chose the US-100 over the SR04 because of its serial mode, built in thermal compensation [3] and 3.3v operation[4]. By connecting the jumper pins on the rear of the sensor, it sends the measurements over a standard UART interface @ 9600 baud which I thought could be handled more easily than the trigger / echo pins. Also performing the distance calculations independently allows the main microcontroller to perform other tasks while the US-100 is calculating the distance.

[1] How an Arduino Ultrasonic Sensor Works, LittleBots.
https://www.youtube.com/watch?v=1jGvzNrtF24
[2] Comparison of ultrasonic sensors US-100 and HC-SR04, Rafa. G
https://arduibots.wordpress.com/2013/10/14/comparison-of-ultrasonic-sensors-us-100-and-hc-sr04/
[3] US-100 ultrasonic sensor in serial mode, Rafa. G
https://arduibots.wordpress.com/2014/10/12/us-100-ultrasonic-sensor-in-serial-mode/
[4] US-100 Ultrasonic Distance Sensor , Adafruit Industries https://www.adafruit.com/product/4019

To use the sensor, send `0x55` for a distance measurement (or `0x50` for a temperature measurement).
The distance response will come back in two bytes. To get the distance in mm:

$$D = MSB * 256 + LSB$$

where MSB = Most Significant Bit, LSB = Least Significant Bit [3]

To get the temperature from the sensor (not used in this project) subtract 45 from the byte value that comes back [3].

DS18B20 temperature sensor

The DS18B20 is a low-cost temperature sensor that uses the OneWire protocol. I chose this sensor because of its popularity and low cost. It operates between -55 ℃ and +125 ℃ on a supply of 3 to 5 volts [1].

To operate this sensor, I used the libraries provided by Pycom available here: https://github.com/pycom/pycom-libraries/tree/master/examples/DS18X20. This meant I did not have to write a OneWire driver from scratch and instead rely on an existing solution. The code to get a value from the sensor is as follows:

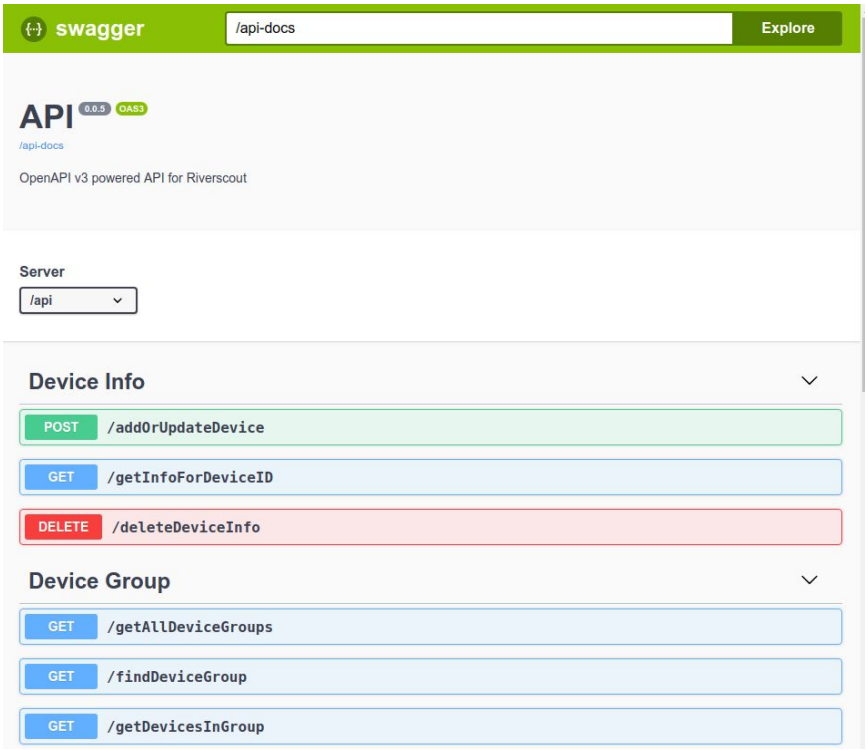
```
ow = OneWire(Pin('P8')) // init the Onewire bus on pin 8
temp = DS18X20(ow) // init the sensor class

# round the result value to two decimal places
waterTemp = round(temp.read_temp_async(), 2) // read the value
print(float(waterTemp)) // print to console
```

Server Middleware

The Riverscout application server is written in NodeJS. It uses MongoDB for data storage with Mongoose as an ORM. The external API uses oas-tools (https://github.com/isa-group/oas-tools) which is based on Swagger and complies with the OpenAPI spec version 3. This allows the API to be defined with a simple YAML file.

Oas-tools (Swagger)

This is a module used to define APIs conforming to the Swagger (OpenAPI v2) or OpenAPI v3 standards. It uses a YAML or JSON file to define the routes, controllers, data formats and responses of the API. This allows one to automatically create documentation for the API. When the server is running, a built in docs page can be viewed.



The built in docs page and API tester

MongoDB + Mongoose

Riverscout used the latest available LTS version of MongoDB (at time of writing) which is version 4. The database access is handled with a library called Mongoose (https://mongoosejs.com/) which provides schema validation, an API that is easier to use than vanilla Mongo and SQL like joins. Also Mongoose allows to create relations between models by referencing one model in another.

The case for forcing schema validation on MongoDB (and also foreign keys) is to allow a certain degree of conformity but also to allow easy schema changes at a future date. Using an SQL database would mean that I would have to ensure the schemas were correct and if anything needed to be changed, the whole data model would need to be revised. WIth noSQL, the changes can be performed much faster and do not require a redesign each time.

Previous experience using these technologies during work placement also factored strongly into the decision as I would not need to learn a new database from scratch

and had already learned complex queries. The justification was that I could achieve more complex goals.

Front end client

Section provided for reference only as this part was omitted due to time constraints.

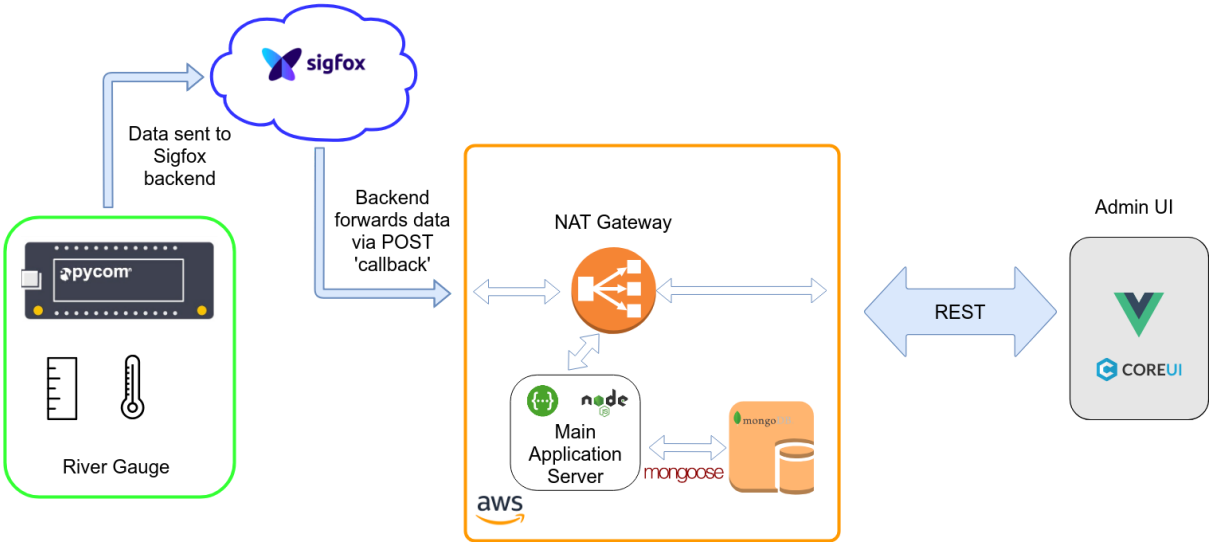
The Riverscout front end is built using the CoreUI framework (https://coreui.io/vue/) which is written in Vue.JS (https://vuejs.org/). The existing framework was used because designing and building a coherent, responsive UI is a serious undertaking. I decided to focus on the backend server platform and chose to use a UI template over building one from scratch. This also meant that the learning curve was reduced.

System Architecture

Overview

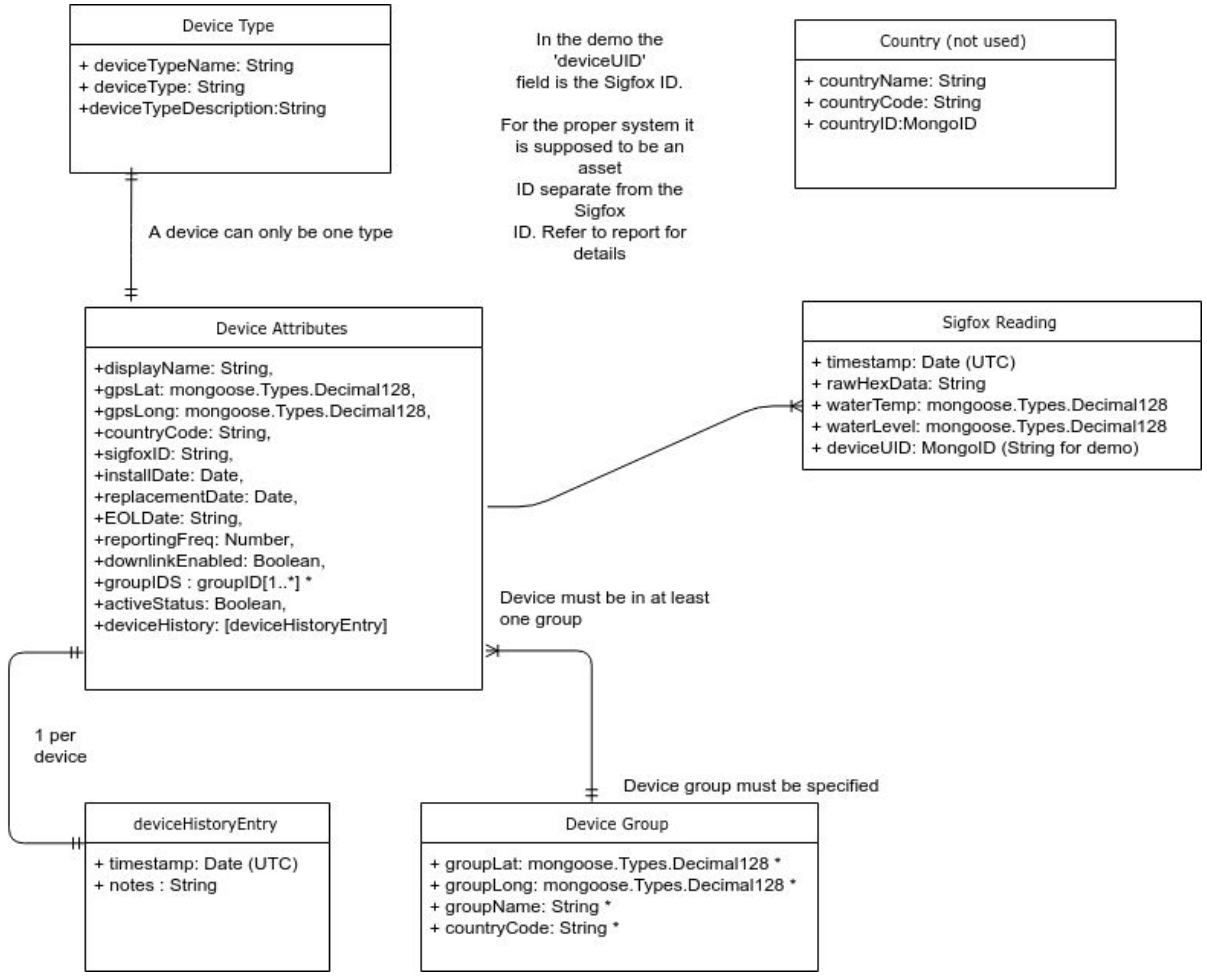
The Riverscout platform ranges from embedded devices (the gauges), to end-user web technologies. Overall the architecture can be split (as above) into three main categories:

- Embedded devices (gauges)
- Server middleware
- Web client - not implemented.



A simplified view of the project. Source, Author

UML Diagram



Embedded level (gauge)

The SiPy is connected to the two sensors via the Pycom Expansion Board v2.0[1]. This provides a USB to serial interface, a battery charging circuit, easy access to pins and an SD card slot [2].

[1, 2] Expansion Board 2.0, Pycom Ltd, https://docs.pycom.io/datasheets/boards/expansion2.html

Schematic

The circuit is wired as follows:

Sensor	Pin	SiPy pin (exp.board pin)
DS18B20	Data	8 (G15)
US-100	TX	4 (G24)
US-100	RX	3 (G11)

Code overview

The SiPy runs code written in Micropython which is an embedded derivative of CPython (standard Python). Micropython aims to provide most of the features of Python 3, with some differences that are documented here: https://docs.micropython.org/en/latest/genrst/index.html

The SiPy runs Pycom's own Micropython distribution. When powered on, the interpreter will look for a file called 'boot.py' If this does not exist, the board will not be able to connect to the computer via serial. A boot.py file is provided as standard. The boot.py file sets up the serial terminal and specifies the main code file (usually called main.py). It runs on device startup similar to a main.c file in C / Arduino. Its purpose is to perform board initialization and point to the main file.

The other important file is 'main.py'. This is where most of the processing gets done.

In the Riverscout project, there are a number of other files:

- onewire.py: Provided by Pycom, this library provides a Onewire bus driver to interface with the temperature sensor as well as functions to get data from it.
- us100.py : This is a driver for the ultrasonic sensor.

main.py

This is the main file that runs on the microcontroller after the boot.py. It performs the following tasks:

- Import all other files and libraries for the project
- Set up the interfaces
- Runs an infinite loop to poll the sensors, send the data to Sigfox and wait for a set amount of time before polling again

us100.py

This provides helper functions that abstract the process of polling the sensor for data. These functions are:

- distance() = returns the distance in millimeters

- temperature() = returns the temperature in °C

onewire.py

Provides a driver for the Onewire bus and the read_temp_async function to interface with the temperature sensor. This is provided by Pycom and contains no code written by myself.

Refer to the comments in the source code for a detailed explanation.

Compression Scheme

In order to reach the constrained 12 byte (96 bit) message limit of Sigfox and remain reasonably accurate, the system needs to be able to compress the data so that as little information is lost as possible. Also for power saving reasons, we would like the message size to be as short as possible. The compression scheme is as follows:

```
Bytes 0 to 4 : 32-bit float value, little endian
Byte 5: 8-bit unsigned integer
```
Extra space is not padded - some devices will pad out the remaining bytes to get a full 96 bit message

The initial plan was to use just two bytes of data (use two integers) where the temperature value was divided by 10 on the receiving end to get a float value. This would have worked but it's limited to a certain range. As the max value for an 8-bit int is 255, this meant that the max temperature could be 25.5 °C. This poses a problem for values above that. Temperatures of up to 28 degrees were reported during last year's heatwave [1]. Therefore Riverscout needs to be able to account for sharp increases in temperature above this value.

After performing some research [2] and in the name of future-proofing I decided that using a 32 bit float value was the best option. It uses 4 times the amount of data but gives me the max range of the sensor down to 0.1 °C. This scheme (including a single byte for water level) still only uses 40 out of 96 bits of data so there is room for more types of sensors.

For the level a value of 1 represents 1 centimetre. The millimetre value from the sensor would be divided by 10 to get a centimetre value and then rounded to an integer. This results in some data loss, but the millimeter levels of precision are not needed for this application and are noisy due to the inaccuracy of the sensor.

Code - device side

The Micropython code for compressing the data is below:

```
struct.pack('f',float(waterTemp)) + bytes([waterLevel])
```

This converts the float value (with 2 decimal places) into a 32 bit IEEE 754 value (which is the binary representation). The float is now represented with 4 hexadecimal characters (8 bits) which is combined with the temperature value and sent to Sigfox. Specifying 'f' as the first argument in struct.pack tells Python to store the value as a C float [3]. The values are stored as little endian (least significant bit first) as the ESP32 is a little-endian architecture (as reported by sys.byteorder [4])

Code - server side

The hex string is passed from Sigfox into the 'parseSigfoxData' decompression function. It takes the raw string, slices the first 8 bits into one variable called 'waterTemp' and slices the rest of the buffer into another variable called 'waterLevel' To parse the waterTemp variable, use the readFloatLE() [5] NodeJS function specifying 0 as the offset. We need to specify little endian as the encoding because the values were originally encoded that way.

To parse the 'waterLevel' value, we can use the readUInt8() [6] function.

The full code is available to view in 'controllers/sigfoxReadingController.js'. See 'Backend Middleware' for an overview of the project structure.

[1] Irish freshwater temperatures 'lethal' in 2018 - fisheries body, Kevin O Sullivan, Irish Times: https://www.irishtimes.com/news/environment/irish-freshwater-temperatures-lethal-in-2018-fisheries-body-1.3744585

[2] Build an End-to-End Sigfox GPS Tracker Using Wia and Pycom", Austin Spivey/Wia.io, https://dzone.com/articles/build-an-end-to-end-sigfox-gps-tracker-using-wia-a

[3, 4] Interpret strings as packed binary data, Python Foundation: https://docs.python.org/3.5/library/struct.html

[5, 6] Buffer, NodeJS Foundation: https://nodejs.org/api/buffer.html

Backend Middleware

The server code is written in Javascript with the API definition written in YAML. The code is divided into several folders. The project structure is as follows:

- README.md : brief description of the repository
- api-spec/ : contains the YAML file to define the API

- app.js : main starting point for a Node app, this loads the YAML file via js-yaml and oas-tools, sets up Express to handle connections and controls the connection to the database via Mongoose.
- controllers/ : contains individual ‘controller’ files that hook into the definitions you provide in your YAML file.
- dao/ : contains files that use Mongoose to perform database operations.
- node_modules/ : appears after running ‘npm-install’, this will contain any extra dependencies required by Riverscout.
- package-lock.json : created after running the above command, npm uses this to determine what versions of what packages are installed.
- package.json : specifies what to install with npm-install.
- schemas/ : contains the Mongoose schema files explained below.
- test/ : contains Mocha test files that unit test the API.

The package.json file contains a number of commands that can be invoked with `npm <command>`. The command list is as follows

- prestart: runs “npm install"before the app.js file is executed
- start: runs the app.js file with Nodemon enabled [1]
- test: runs Mocha and provides a test coverage report with nyc.js [2]

Mongoose schemas.

A Mongoose schema sets up rules for what kind of data can be stored in the database. In this file you specify a schema which is then compiled into a model for use elsewhere. A schema may look like the following (schemas.countrySchema.js):

```
const CountrySchema = new Schema({
      countryName: String,
      code: String,
      countryID: mongoose.Schema.Types.ObjectId
});
```

In this schema a ‘country’ is being defined as having a name, a code and a unique ID. The data types are also defined for each key. A number of different data types exist for defining objects, refer to the official documentation [1] for more info.
When a schema is defined, in order to be able to use it elsewhere it has to be ‘compiled’ into a Model. This model can then be exported from the file by adding it to a module.exports statement like the following:

```
module.exports = {
      countrySchema : CountryModel
    };
```

More than one model can be exported at once. To use the model in a different file (such as in a DAO file), add a require statement that includes the relative path of the model file. Database operations can then be performed on the model. When saving a record, Mongoose will ignore any fields that are not present in the schema. This provides a filter against inconsistent data.

The following is an example of a DAO file performing DB operations on the above schema. ‘CountrySchema’ is what we have imported the schema file as, ‘countrySchema’ is the model name in the schema file:

```
var CountrySchema = require('../schemas/countrySchema') // require schema

countryDAO.getAllCountries = function(){
   return CountrySchema.countrySchema.find({})
   .select('_id countryName code')
   .then(function(z){
       return z
   })
   .catch(err => {
       return(err.message)
   })
}
```

[1] Schemas, MongooseJS : https://mongoosejs.com/docs/guide.html

countrySchema

This defines a ‘country’. Originally, this would store a list of countries and country codes that the front end would query for upon adding a device or group. This was implemented to prevent incorrect values for a country code being entered. The values ‘ie’, ‘IE’ and ‘Ie’ all represent Ireland but querying for the string ‘ie’ will only return records containing that string and not records containing the other two.

Originally I had planned to use an enum type within Mongoose to hold the country

codes. This is possible but when testing I found it did not work. Using this separate API route, schema and input could be considered a shortcoming of Riverscout. See ‘Reflection’ for a more detailed explanation and a possible alternative.

Field Name	Type	Description
countryName	String	Name of a country
code	String	2 character country code same as used in web domains
countryID	Mongoose ID	Unique ID

The countryID is not required as the _id of each country is already a unique identifier.

deviceAttributeSchema

This schema contains all the attributes of a ‘device’ which is also referred to as a ‘gauge’. These include its name, location, install date and replacement dates. Different gauges will add extra fields to this schema, the schema below is for a river gauge with temperature and water level sensors. The location is stored with 4 decimal places - this gives a location accuracy of up to 11 meters. [1]

Field Name	Type	Description
displayName	String	name to display on front end
gpsLat	mongoose.Types.Decimal128	Latitude stored to 4 decimal places [2]
gpsLong	mongoose.Types.Decimal128	Longitude stored to 4 decimal places [3]
countryCode	String	Two characer code, see above
sigfoxID	String	Sigfox Device ID
installDate	Date (UTC)	Date of gauge install
replacementDate	Date (UTC)	Date gauge was replaced

EOLDate	Date (UTC)	Date gauge is to be replaced by
reportingFreq	Number	how often (in minutes) the device is expected to send up data
downlinkEnabled	Boolean	Specifies if device is capable of receiving data from the cloud.
groupIDS	Array of Mongo IDS, foreign keys from ‘groupModel’	array of references to 'groupModel' in 'deviceGroups.js'
activeStatus	Boolean	Is the device currently active?
deviceHistory	Array - see below	array containing all the history associated with a device. Empty on creation, not currently specified through add device API

[1, 2, 3] Measuring accuracy of latitude and longitude?, StackExchange: https://gis.stackexchange.com/questions/8650/measuring-accuracy-of-latitude-and-longitude/8674#8674

The ‘deviceHistory’ array contains two elements, a ‘timestamp’ field with a UTC date and a ‘notes’ field which is a String.
There are no max values specified for the Number or String data types.

deviceGroupSchema

This schema contains information on a device group. When adding a new device, it is mandatory to specify a group. Refer to the above footnote for an explanation behind using 4 decimal places.

Field Name	Type	Description
groupLat	mongoose.Types.Decimal128	Latitude stored to 4 decimal places
groupLong	mongoose.Types.Decimal128	Longitude stored to 4 decimal places
groupName	String	Name of group

countryCode	String	2 character code, see countrySchema

deviceTypeSchema - not implemented fully

Originally when planning the system out, I decided to have a separate device type. This was to differentiate certain devices by an enumerated string. For example one device could be a ‘river’ gauge while another could be a ‘crop’ gauge. This was so that different types of gauges could be grouped and sorted by type. Eventually due to problems using the enum in Mongoose, I abandoned the idea and each new device does not currently contain a type. The schema is listed below:

Field Name	Type	Description
deviceTypeName	String	Name of the type
deviceTypeDescription	String	Short description of what this type is
deviceType	String Enum [‘river’, ‘crop’, etc...]	Enum for different types

sigfoxDataSchema

This schema contains the measurements as coming from the Sigfox backend portal. The data is processed in the ‘sigfoxReadingController’ before being stored here. A device can have multiple readings but the device needs to exist before a reading can be created.

During the data ingest process, I had planned to do the following:

1. Take in the reading from the Sigfox backend,
2. Take the Sigfox ID from the incoming request.
3. Search the ‘deviceAttribute’ collection for a device containing that Sigfox ID.
4. Get that devices’ _id field.
5. Place this ‘_id’ into the ‘deviceUID’ field.
6. Store the measurement in the ‘sigfox_device_measurements’ collection.

The reason behind this process is that in the event of a sensor being replaced, it’s Sigfox ID will change. The existing measurements for that location now contain an ID

that is no longer valid. The existing measurements could be found and updated but I thought the above process was more future-proof.

Currently the mapping is not done, the deviceUID field just contains the Sigfox ID for now.

Field Name	Type	Description
deviceUID	String	_id field from the deviceAttribute schema
rawHexData	String	Raw value from S.F backend, stored in case mapping process was done incorrectly
waterTemp	mongoose.Types.Decimal128	Temperature in ℃
timestamp	Date (UTC)	UTC timestamp from S.F backend
waterLevel	mongoose.Types.Decimal128	Water height in cm

Mocha unit tests

Riverscout contains unit tests for use with Mocha (https://mochajs.org/), a Javascript unit test framework. The test/ folder contains the following folders:

- country : tests CRUD operations on the country specific routes
- deviceAttributes : tests CRUD ops on devices
- deviceGroups : add/update/delete device groups
- deviceTypes : files currently empty
- helpers : contains utility code (see below)
- sigfoxReadings : currently empty

The helpers directory contains three helper files and a folder with JSON files for inserts into the database.
The helper files purge and populate the respective collections. They provide the following functions:

- populate() : inserts that helper file's JSON object into the respective collection
- purge() : clears that helper file's associated collection.

Mocha tests use ‘hooks’ these are files that run before or after a test suite. Riverscout uses the ‘hooks.js’ file to define setup and teardown hooks with the following functions:

- before() : runs once before tests begin
- after() : runs after all tests have finished.
- assets : contains SCSS files, use the _custom.scss file to override default CSS values
- containers : contains the DefaultContainer which forms the core of a S.P.A. Other components will be ‘switched’ into this one.
- main.js : entry point for the application, bootstraps all other
- router : contains ‘index.js’ which defines application routes, error pages and login/signup pages
- shared : utilities shared between multiple components
- views : contains sub directories for each of the main pages and components.

API Specification

API Design Considerations

Efficiency when searching for data

The API has been designed to be as flexible as possible. The first area is group searching. When querying for devices, a group query needs to be made first. This will return a list of groups and their IDs. Querying a group based on it’s ID will return a list of devices in that group.

The reasoning behind this is that it reduces the workload on both the client and the server, making the application more responsive. If we were to simply query for all devices matching a country code and type, we could be inundated with thousands of results. The groups would appear on the map as clusters and when a user zooms in on a cluster, the application would get the current devices in the visible groups. This complicates the process of getting data but reduces the bandwidth and resource usage on the server side.

Date searching for queries

The second area of flexibility is Riverscout allowing date ranges in it’s device data queries. When searching for a device, we need to specify ‘timestampGt’ and ‘timestampLt’ which are dates greater than / less than respectively. This allows for powerful queries as a user may only want data for one day, or 1 week. The system will return hourly data within the dates specified.

Using the Riverscout API

The API is accessed with the following steps:

1. Get a list of device groups for your country code
2. Get the ID of the group of interest
3. Using this group ID, query for all devices in that group
4. Once all the devices have been returned, get the ID of the desired device and get readings for that device ID between the specified dates.

API Routes

The following is a brief description of each API route, broken into the various categories.

Device Info

These routes handle CRUD operations for a gauge/device

POST: /addOrUpdateDevice

Adds a new device (Sigfox/NBIoT) or if the device already exists, updates it's details

Value	Type	Description
`displayName`	`string`	Name to display in front end
`gpsLong`	`number`	GPS longitude of the gauge install location
`gpsLat`	`number`	GPS latitude of the gauge install location
`countryCode`	`string`	Uses the same codes as web domains. Eg: IE => Ireland
`sigfoxID`	`string`	Sigfox ID as generated by the Sigfox backend
`installDate`	`string`	Install date of device. Please use UTC format. Eg: 2018-09-09T14:00:00Z
`replacementDate`	`string`	UTC formatted date of the expected date the device is to be replaced

`EOLDate`	`string`	UTC format date of the time a device was actuallly replaced
`reportingFreq`	`string`	How often (in minutes) the device is expected to send up data
`groupIDS`	`Array [string]`	An array of the group IDs this device is in
`activeStatus`	`boolean`	Device active status
`downlinkEnabled`	`boolean`	Whether the device allows data to be sent to it or not.

DELETE: /deleteDeviceInfo

Deletes the selected device

Value	Type	Description
`deviceID`	`String`	Database ID of device to delete data for.

GET: /getInfoForDeviceID

Returns data for the specified device like it's location and group

Value	Type	Description
`deviceID`	`String`	Database ID of device to return data for.

Device Group

GET: /getAllDeviceGroups

Returns all the device groups for the specified country code

Value	Type	Description
`countryCode`	`String`	Return all groups matching this country.

GET: /findDeviceGroup

Returns a device group matching the specified name

Value	Type	Description

`groupName`	`String`	The name of the group to return

GET: /getDevicesInGroup

Returns the IDs of the devices in a group

Value	Type	Description
`groupID`	`String`	Return devices in a group with this ID.

GET: /getDevicesInCountry

Note: this route is not required and was added to facilitate testing. Use the above method when querying the API

Returns the IDs of all devices matching a specified country code

Value	Type	Description
`countryCode`	`String`	Return devices with this countryCode.

POST: /addOrUpdateDeviceGroup

Adds a group for devices

Value	Type	Description
`groupLat`	`Number`	GPS Latitude of the new group.
`groupLong`	`Number`	GPS Longitude of the new group
`groupName`	`String`	Name of new group. MUST be unique otherwise will trigger an update instead of an add.
`countryCode`	`String`	Two character code representing countries

DELETE: /deleteDeviceGroup

Deletes the selected device group

Value	Type	Description

deviceGroupID	String	Database ID of the group to remove

Device Types

GET: /getAllDeviceTypes

Has no parameters, returns all the possible types of devices

DELETE: /deleteDeviceType

Deletes the selected device type

Value	Type	Description
deviceTypeID	String	Database ID of the type to remove

POST: /addOrUpdateDeviceType

Adds a new / updates an exising device type

Value	Type	Description
deviceTypeName	String. Min:1, Max: 30	The name to use for a new device type
deviceType	String Default: 'river'	The type of this new device
deviceTypeDescription	String. Min:1, Max: 250	Describe the device characteristics here

Sigfox Device Readings

POST: /addSigfoxDeviceReading

Adds a reading for a Sigfox device

Value	Type	Description
sigfoxID	String	The Sigfox ID for a device

`rawHexString`	`String`	The raw hex data coming from the Sigfox backend
`timestamp`	`String`	UNIX timestamp coming from the Sigfox backend

DELETE: /deleteOneSigfoxDeviceReading

Deletes single device reading.

Value	Type	Description
`readingID`	`String`	Database ID of the reading to remove

DELETE: /deleteAllSigfoxDeviceReadings

Deletes all readings for a Sigfox device matching the specified ID

Value	Type	Description
`deviceID`	`String`	Database ID of the device to remove readings for

GET: /getReadingsForDeviceID

Returns readings for the specified device between the timestamps specified

Value	Type	Description
`deviceID`	`String`	The ID of the device to get data for
`timestampGt`	`String`	Fetch results greater than this timestamp Timestamp is to be specified in UTC format
`timestampLt`	`String`	Fetch results less than this timestamp Timestamp is to be specified in UTC format

Country Codes & Info

POST: /addOrUpdateCountry

Adds a new 'country' with a name and a country code. If the specified country exists, it will be updated instead.

Value	Type	Description
`countryName`	`String`	Country Name (MUST be unique)
`countryCode`	`String`	Code for the new country

GET: /getAllCountries

Returns all countries.

DELETE: /deleteCountry

Deletes the selected country

Value	Type	Description
`countryID`	`String`	Database ID of the country to remove

Setting up the dev / staging environment

Required Software

The Riverscout environment was developed and tested on Ubuntu 18.04 LTS. It also needs the following installed to function:

- Node JS (select the .deb version of the latest LTS release). Setup instructions available here: https://github.com/nodesource/distributions#deb
- MongoDB (use version 4 or above for Ubuntu Bionic 18.04). Instructions available on MongoDB website https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

Database configuration

By default MongoDB comes with access control disabled and outside traffic blocked. When we open it up to external traffic, access control **must** be enabled otherwise the database will be wide open for anyone to access! See https://www.theregister.co.uk/2017/01/09/mongodb/ for an idea.

The following instructions were based on Ian London's article "How to connect to your remote MongoDB server", https://ianlondon.github.io/blog/mongodb-auth/

To setup a new user and configure access control:

- Start the mongo service if it is not already running (`sudo service mongod start`)
- Check the status of the mongod.service (`systemctl status mongod.service`)
- If the service is running, log into Mongo by typing mongo at the shell. This brings you into the MongoDB Shell.
- Switch to the 'riverscout' database by typing `use riverscout`
- Create a new user called 'riverscout' with the password 'riverscout' by typing

```
db.createUser({
    user: 'riverscout',
    pwd: 'riverscout',
    roles: [{ role: 'readWrite', db:'riverscout'}]
})
```

- Now the new user has been added, type `db.auth("riverscout","riverscout")`
- This should return 1 which means your new user can successfully authenticate on the 'riverscout' db
- Exit the Mongo shell by pressing Ctrl+D
- Edit the file `/etc/mongod.conf` with your editor of choice
- Bind the server to all interfaces:
  ```
  # network interfaces
  net:
    port: 27017
    bindIp: 0.0.0.0
  ```
  NOTE this needs to be explicitly declared as such in Mongo DB versions 4 +

- Enable access control by adding the following to the #security line
  `security:` **uncomment this**
    `authorization: 'enabled'` **add this**

Save this file and exit. Restart MongoDB by typing `sudo service mongod restart` Check the status. If MongoDB fails to start, it is probably an indentation issue with the config file. Refer to the log file and the output from systemctl.

Vagrant based development environment

The Riverscout dev environment uses Vagrant to provision and manage virtual machines. To set up the development environment perform the following steps

- Install Vagrant (+ whatever dependencies it needs)
- Install Virtualbox and its corresponding expansion pack

VMWare has not been tested and would require some modification to the Vagrantfile

- Clone the dev-environment erpo and cd into it
- If necessary, change the amount of RAM the VM receives by editing the `vb.memory` line in the Vagrantfile to a value that fits your system. Min recommended value is 1024MB
- Start the Riverscout VM by typing `vagrant up`
- This will download and set up the new VM. When this is done, login with `vagrant ssh`

Now the following steps can be performed as the dev environment is ready

Installing Riverscout

To install Riverscout follow these instructions

- Clone the Riverscout repo
- Cd into this repo and run `npm-install`
- Start the server by typing `npm-start`

The Swagger docs page should be accessible at http://10.10.1.10:8080/docs.

Reflection

This has been a challenging project from start to finish with a lot of setbacks. Overall I spent far too much time on considering different approaches to the various parts of the project. I also focused too much on making sure the system was future-proof and scalable.

What could have been added

- Use GraphQL to power the API
- Use seperate PUT routes in existing API instead of relying on findOneAndUpdate provided by Mongoose
- Use a SQL database with a ‘hybrid’ schema - a mix of SQL and JSON
- Design a prototype river gauge

A ‘hybrid’ schema would consist of a static set of fields that never change even with different device types. These include a name, deviceID, type and location. The other part is the ‘fluid’ set of fields, these contain attributes unique to that device.

I originally considered this but decided against it because of previous experience with Mongo/Mongoose.