Authors:

Ola Hulleberg (ola.hulleberg@gmail.com) student-id: 498711
Richard Langtinn (r.langtinn@gmail.com) student-id: 512961

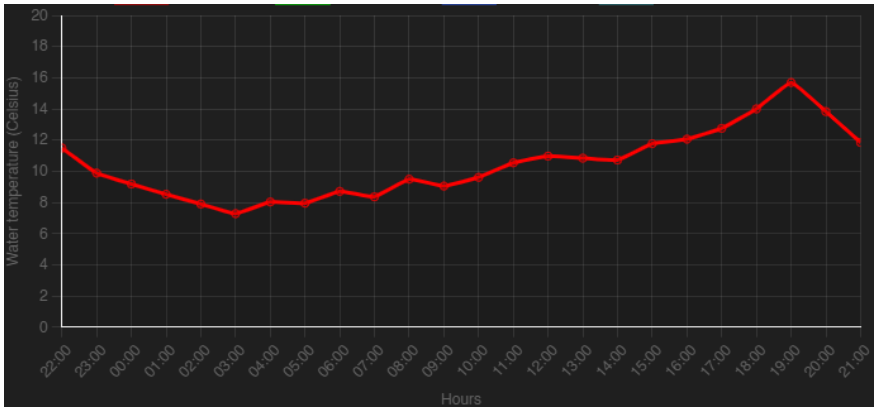
Documentation:

The scenario we thought about was to measure water temperature for an extended period. Our IoT consists of water temperature measurement using an MQTT broker and fetching the data from MongoDB. We're running the MQTT server through websockets which is attached to a HTTP-server that express is attached to.

Our IoT data is generated randomly using a base temperature of 14 celsius, and a fluctuation chart that sets a multiplier per hour (24-hour format).

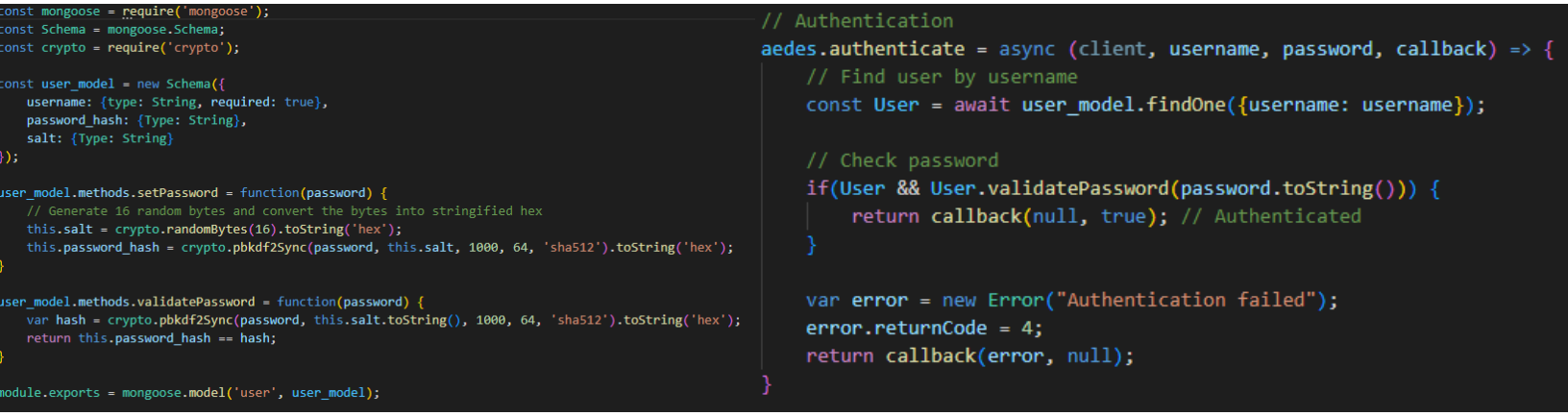
```
if(sensorActive) {
    var BrokerConnected = false;

    client.on('connect', () => {
        BrokerConnected = true;
    });

    const tempBase = 14;
    var fluc = [];
    fluc["Bayside Beach"] = [0.65, 0.6, 0.55, 0.53, 0.52, 0.51, 0.53, 0.54, 0.55, 0.56, 0.56, 0.58, 0.6, 0.62, 0.66, 0.72, 0.78, 0.85, 0.92, 1, 0.93, 0.8, 0.75, 0.7]; // Bayside
    fluc["Paradise Bay"] = [0.63, 0.62, 0.56, 0.53, 0.56, 0.57, 0.59, 0.63, 0.65, 0.68, 0.7, 0.73, 0.76, 0.78, 0.8, 0.84, 0.86, 0.89, 0.97, 1.1, 0.96, 0.84, 0.79, 0.7]; // Parad
    fluc["Sandy Shores"] = [0.63, 0.62, 0.56, 0.53, 0.56, 0.57, 0.59, 0.63, 0.65, 0.68, 0.7, 0.73, 0.76, 0.78, 0.8, 0.84, 0.87, 0.9, 0.93, 0.97, 0.96, 0.89, 0.79, 0.7]; // Sandy
    fluc["Glass Beach"] = [0.73, 0.67, 0.64, 0.6, 0.58, 0.58, 0.58, 0.6, 0.63, 0.66, 0.7, 0.73, 0.76, 0.8, 0.85, 0.9, 0.94, 1, 1.1, 1.13, 1.1, 1, 0.94, 0.8]; // Glass Beach

    cron.schedule("0 * * * *", () => {
        if(BrokerConnected) {
            // IoT sensor data
            //const wt = TemperatureSensor.Celsius;
            const DateNow = new Date();

            const selectedFluc = fluc[beachName];
            const wt = (tempBase * selectedFluc[DateNow.getHours()]+Utils.randomDecimals(-0.5, 0.5, 2)).toFixed(2);

            client.publish('water_quality', JSON.stringify({
                location: beachName,
                timestamp: DateNow,
                water_temperature: wt
            }));
        }
    });
}
```

This gives us consistent and realistic water temperatures that fluctuate depending on the hour of day.



Authentication

Authentication is handled through MQTT's authentication method, where you supply your own function to check whether the user has access, and return the appropriate packet. The users are stored in MongoDB (*Atlas Cloud*), with salted passwords. The validatePassword method is a custom MongoDB method that we created.



```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const crypto = require('crypto');

const user_model = new Schema({
    username: {type: String, required: true},
    password_hash: {Type: String},
    salt: {Type: String}
});

user_model.methods.setPassword = function(password) {
    // Generate 16 random bytes and convert the bytes into stringified hex
    this.salt = crypto.randomBytes(16).toString('hex');
    this.password_hash = crypto.pbkdf2Sync(password, this.salt, 1000, 64, 'sha512').toString('hex');
}

user_model.methods.validatePassword = function(password) {
    var hash = crypto.pbkdf2Sync(password, this.salt.toString(), 1000, 64, 'sha512').toString('hex');
    return this.password_hash == hash;
}

module.exports = mongoose.model('user', user_model);
```

```
// Authentication
aedes.authenticate = async (client, username, password, callback) => {
    // Find user by username
    const User = await user_model.findOne({username: username});

    // Check password
    if(User && User.validatePassword(password.toString())) {
        return callback(null, true); // Authenticated
    }

    var error = new Error("Authentication failed");
    error.returnCode = 4;
    return callback(error, null);
}
```

Processing layer and graph of water temperature

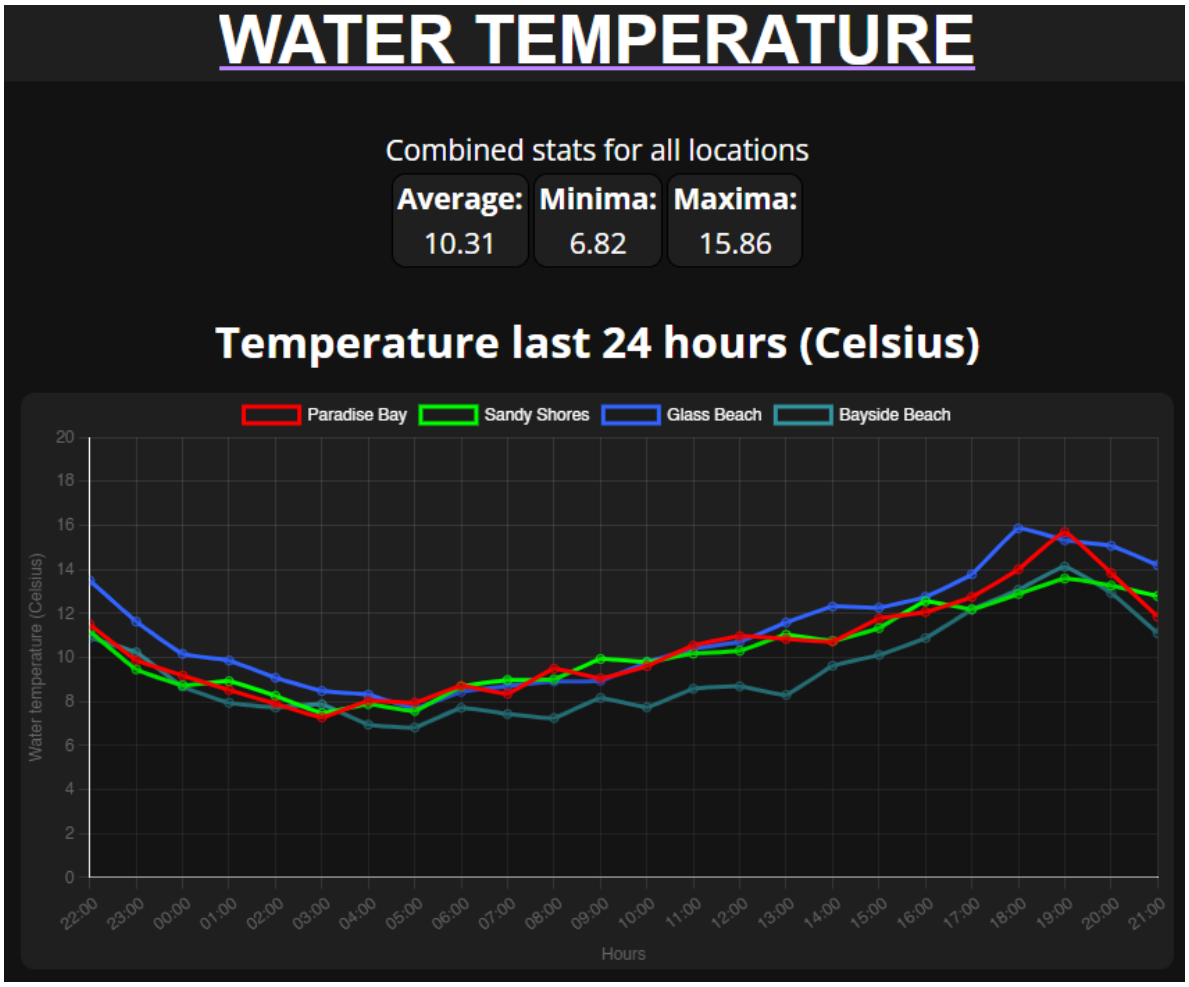
Our processing layer consists of endpoints running on Express, we currently have these analytical functions directly implemented in our API: Average, Min and Max (Available from the endpoints `'/API/v1/water_quality/{function}?hours={amountOfHoursBack}'` Example: `'/API/v1/water_quality/average?hours=24'`(*Grabs the average temperature from the last 24 hours*)

To find the min/max faster, we simply sort the water_quality readings from top/bottom depending on which function we're calling, and limit the result to 1. The pseudo code looks something like this:

```
const wq_list = await water_quality_model.find().sort({timestamp:
1}).limit(1);
```

Lastly but not least, we have our client-sided processing layers (AKA graphs). This line chart visualization is made with Chart.js, and fetched one minute past every hour with a cron-like scheduler. This is done to give the server one minute of time to process the data, and fill it into the DB before the users fetch it.

Portal served with Express.static() function



IoT - MQTT publishers

All the data you see on our portal is fed every hour via cron-like scheduled jobs (*node-cron)* by 4 separate publishers (All running in our Heroku app). They send data over the MQTT protocol with JSON payload. Each publisher has its own name denoting the location the water temperature measurement is taken.

IoT - MQTT broker

To be able to run an MQTT broker and a webserver on Heroku, you first need to create a HTTP server that you attach your Express app to. You then attach a websocket which points to the broker handler, to the HTTP server. The HTTP server is now the only server listening on a port, but it delegates the different protocols like ws:// and http:// to the correct destinations. In an ideal world, we would generate a certificate and keychain to use the HTTPS server that Node provides so our auth communication is encrypted.

Latency:

Local on the server:

```
CBOR-to-jObj:   394ms | Data-size: 225.24KB
 XML-to-jObj:    74ms | Data-size: 421.01KB
 EXI-to-jObj:    FAIL | Data-size: 108.91KB
JSON-to-jObj:     3ms | Data-size: 305.46KB
```

Remote from Ola's PC:

```
CBOR-to-jObj:   635ms | Data-size: 225.24KB
 XML-to-jObj:   192ms | Data-size: 421.01KB
 EXI-to-jObj:    FAIL | Data-size: 108.91KB
JSON-to-jObj:    49ms | Data-size: 305.46KB
```

Code used to measure latency from MQTT Publisher to MQTT broker, as well as data format performance:

```
// Data format tests
const CBORTestTopic = 'data_format_test_cbor';
const XMLTestTopic = 'data_format_test_xml';
const EXITestTopic = 'data_format_test_exi';
const JSONTestTopic = 'data_format_test_json';

aedes.on('publish', async (packet, client) => {
    var payload = {
        dataSize: Buffer.byteLength(packet.payload)
    };

    if(packet.topic === CBORTestTopic) {
        const jObj = await cbor.decodeAll(packet.payload);

        if(!jObj)
            payload.error = "Failed parsing CBOR";

        aedes.publish({
            topic: CBORTestTopic+"_response",
            payload: JSON.stringify(payload)
        });
    } else if(packet.topic === XMLTestTopic) {
        const jObj = await xml2js.parseStringPromise(packet.payload);

        if(!jObj)
            payload.error = "Failed parsing XML";

        aedes.publish({
            topic: XMLTestTopic+"_response",
            payload: JSON.stringify(payload)
        });
    } else if(packet.topic === EXITestTopic) {
        const jObj = await xml2js.parseStringPromise(packet.payload, {strict: false});

        if(!jObj)
            payload.error = "Failed parsing EXI";

        aedes.publish({
            topic: EXITestTopic+"_response",
            payload: JSON.stringify(payload)
        });
    } else if(packet.topic === JSONTestTopic) {
        const jObj = await JSON.parse(packet.payload);

        if(!jObj)
            payload.error = "Failed parsing JSON";

        aedes.publish({
            topic: JSONTestTopic+"_response",
            payload: JSON.stringify(payload)
        });
    }
});
```

```
{
    // SenML benchmarking (Each step is delayed to prevent MQTT from filling buffer before sending resu

    // CBOR - 0 sec delay
    setTimeout(() => {
        CBORSendTime = Date.now();
        const data = fs.readFileSync('./data/water_qualities.cbor');

        client.subscribe(CBORTestTopic+"_response");
        client.publish(CBORTestTopic, data);
    }, 0);

    // XML - 1 sec delay
    setTimeout(() => {
        XMLSendTime = Date.now();
        const data = fs.readFileSync('./data/water_qualities.xml');

        client.subscribe(XMLTestTopic+"_response");
        client.publish(XMLTestTopic, data);
    }, 1000);

    // EXI - 2 sec delay
    setTimeout(async () => {
        EXISendTime = Date.now();
        const data = fs.readFileSync('./data/water_qualities.xml.exi');

        client.subscribe(EXITestTopic+"_response");
        client.publish(EXITestTopic, data);
    }, 2000);

    // JSON - 3 sec delay
    setTimeout(() => {
        JSONSendTime = Date.now();
        const data = fs.readFileSync('./data/water_qualities.json');

        client.subscribe(JSONTestTopic+"_response");
        client.publish(JSONTestTopic, data);
    }, 3000);
}
```

Conclusion latency:

jQuery seems to be the absolute champion of speed when it comes to converting a string (*file*) into a javascript object. EXI is unfortunately a poorly supported standard, and therefore has no real support in Node.

There is currently only one available library for encoding/decoding EXI:

https://github.com/EXIficient/exificient.js

This library is however not complete, doesn't parse our file, and contains `console.log();` in every single function, leading to the library being slowed down by the console output itself. One method we've been investigating is EXI4JSON, where you compress and bit-pack JSON with the EXI spec. The library mentioned above (*Exificient*) exports EXI4JSON functions, and the way this would be used is `const jObj = EXI4JSON.parse(packet.payload);` However, as mentioned earlier, this library is chock full of console spam, leading to an unrealistic measurement anyways, as the console messages slow down the process by hundred-folds if not thousand-folds.

Our guess as to why JSON performs so well with Javascript is that it's specifically designed for the language. JSON is after all just a stringified version of the Javascript Object that we're trying to convert it into.

Github:

`https://github.com/rlangtinn95/cloudobligdos.git`

Heroku:

https://cloudobligdos.herokuapp.com/