Dr Tomasz Kurowski
t.j.kurowski@cranfield.ac.uk

Database interface practical

On Monday you have learned the basics of JavaScript both on the front-end (browser, client-side applications) and the back-end (Node, server-side applications). Today you will use those skills to implement a simple REST API which will allow users to fetch data from the microbial growth database you designed on Tuesday.

If you manage to implement and fully populate your SQLite microbial database, you should try using it in this practical; otherwise you can find a pre-generated `microdb.sqlite` database file on Canvas among the example solutions for Tuesday. Also available are three implementations (one in JavaScript, one in Java, and another in Python) of an application, which can be used to populate such a database.

In case your schema or approach for populating the database is different, feel free to continue using your version – it's better if you catch-up on your own today instead of copying the code on Canvas!

Once you have a populated SQLite database file, open it using the `sqlite3` command line tool. Investigate the number of rows, and test some basic queries, such as:

- Selecting all data points for a specific experiment.
- Selecting all experiments for a specific organism.
- Counting the number of experiments, data points, and authors.

In today's practical you will create an interface which will allow users to make these kinds of queries over the HTTP protocol (e.g., through a browser, an application like curl, or purely programmatically).

Project setup

Create a new npm project using `npm init` and create a `server.js` file to serve as an entry point. Like yesterday, we will use the express to create a server application, and make it listen at a port of our choice, but this time we will not be serving any static Web pages, but only responding to data requests:

```
const app = express();
const port = 3000;
app.listen(port, function () {
   console.log(`Application deployed on port ${port}`);
});
```

We could simply add the code to handle requests (such as GET and POST) for different API routes / end-points inside `server.js` as we did previously, but generally it is a good practice to separate the deployment of our server from the handling of our routes. Create a `router.js` file which will contain our modular Router, and add the following code there:

```
const express = require('express');

const microbe_router = express.Router();
microbe_router.use(function (req, res, next) {
    console.log('Received request');
    next();
});
microbe _router.get('/greeting', function (req, res) {
    res.send('Hello world!');
});

module.exports = microbe_router;
```

We create a router and store it in the `microbe_ router` variable, which is then exported at the end of the script. We have added one route at `/greeting`,

which respond to GET requests by sending a 'Hello world!' string. We have also added a "middleware" (with no specific route) which will log all received requests. This will trigger every time a request is received, prior to it being handled by more specific routes, allowing us to do router-specific processing or logging before the request is handed over (via the `next()` function) to the actual route handler.

With our basic router ready, we can import it in our server.js

script: `const router = require('./router');`

And attach it to a path for our API:

```
app.use('/api', router);
```

Run the server and access our new route at http://localhost:3000/api/greeting. Note that while a browser is often a good tool for investigating the outputs of an API, you can just as easily make this request in the command line by using a tool called curl (or cURL):

```
curl http://localhost:3000/api/greeting
```

It is easy to include positional URL parameters in any route (their identifiers are preceded with a : colon), which are then available in the `req.params` object. Add the following endpoint to our router:

```
microbe_router.get('/greeting/:name', function (req, res) {
    res.send(`Hello ${req.params.name}!`);
});
```

Try it out! You can also make parameters optional by appending a question mark (?):

```
microbe_router.get('/greeting/:name/:lastname?',
                   function (req, res) {
    if (req.params.lastname) {
        res.send(`Hello Dr ${req.params.lastname}!`);
    } else {
          res.send(`Hello ${req.params.name}!`);
    }
});
```

Database interface

Now that we know how to control routes with parameters, we can start linking up our SQLite database with the router. Install the sqlite3 package (you may wish to install version 4.2.0 if you experience issues with versions over 5.0.0):

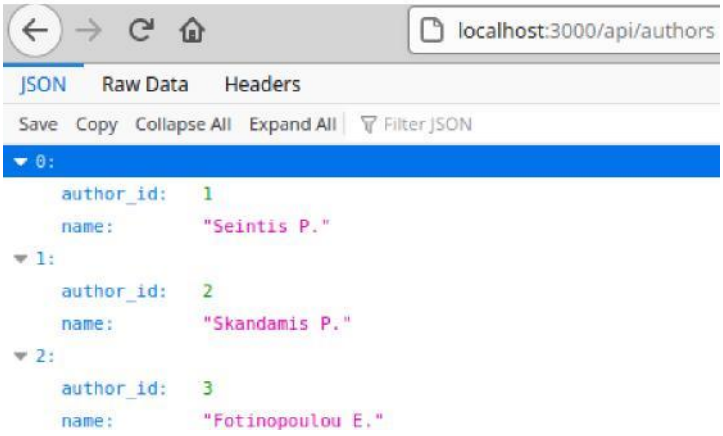
```
npm install sqlite3 --save
```

Import the package and open a connection to your database in the router:

```
const sqlite3 = require('sqlite3').verbose();
const db = new sqlite3.Database('./microdb.sqlite');
```

You will now be able to use the `db` object to interact with the database, similar to what you did to populate your tables. It includes functions which can be used for querying data: `all`, `each` and `get`. When using the all function, you receive the results of your query in the form of an array containing all the rows, as in the route below (try it!):

```
microbe_router.get('/authors', function (req, res) {
    const query = 'SELECT * FROM authors;';
    db.all(query, [], function (err, rows) {
        if (err) {
            throw err;
        }
        res.json(rows);
    });
});
```



Note that we are sending the result as JSON objects rather than plain text. With the get function, we receive only the first result, which is useful when what we are querying for is a single value, such as the number of authors:

```
microbe_router.get('/authors/count', function (req, res) {
    const query = 'SELECT COUNT(*) AS author_count FROM authors;';
    db.get(query, [], function (err, row) {
        if (err) {
            throw err;
        }
        res.json(row);
    });
});
```

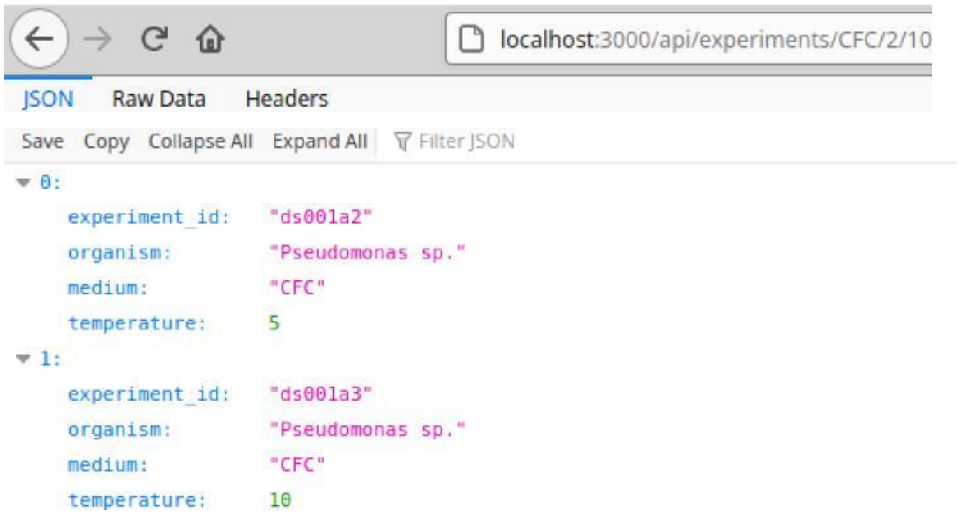
The each function executes a callback function separately for each row, which is useful in many situations (for example in complex, slow queries we may want to start processing outputs as they arrive, instead of waiting for all of them to be ready), but we will not be using it today.

```
db.each(query, paramteres, (err, row) => {
    // process row
});
```

You may have noticed the empty array ([]) passed as a second argument to the database object functions. It is used to provide query parameters, which

take the place of placeholders (question marks) in the SQL query string, as in the following route:

```
microbe_router.get('/experiments/:medium/:mintemp/:maxtemp',
                    function (req, res) {
    const query = 'SELECT * FROM experiments WHERE medium = ? ' +
                  'AND temperature >= ? AND temperature <= ?;';
    const parameters = [
        req.params.medium,
        req.params.mintemp,
        req.params.maxtemp
    ];
    db.all(query, parameters, function (err, rows) {
        if (err) {
            throw err;
        }
        res.json(rows);
    });
});
```



Your task for today is to create analogous API routes for **all** our tables.

Basic visualisation in R

You can use the httr package to make HTTP requests in R, and the jsonlite package to parse JSON responses. Here is an example:

```
experiment <- "ds001a3"
request <- GET(paste("http://localhost:3000/api/datapoints/",
               experiment, sep=''))
response <- content(request, as = "text", encoding = "UTF-8")
df <- fromJSON(response)
```

Write an R script in which you contact your REST API, retrieve bacterial growth data for an experiment, and plot it in the form of a scatter plot – we will get to more analytics tomorrow.

To be continued tomorrow!