CAB230 Web Computing
Further Work with Express

In the last prac we worked through a range of exercises targeting node and Express, culminating in the World Cities REST API. In this prac we will start with the World Cities Express app and make it a bit more professional. Last time we used middleware to handle our DB connectivity, but we were a bit sloppy in our handling of the connection. This time we are going to hand it over to the `knex` package. Our main goal is to prepare you for the second assignment, so we will work through `knex` and the associated calls, show you middleware for logging, and then demonstrate how to create Swagger docs for the API. At the end of the worksheet, we will add in a route to update specific city information, to show you how to create and work with a POST-based route. To some extent our work here will seem over the top relative to the (limited) sophistication of the World Cities API, but it will provide you with some good background.

Your starting point should be the completed World Cities API from last time. Please talk to your tutor if you are having difficulties completing that prac sheet. Once that is working, you can proceed as follows.

There will be two more extensions to this system. Again, these may seem over the top, but the idea is to learn how to use the more sophisticated features.

- This week we will release the Server-Side JWT Worksheet, and that will allow you support users and to then authenticate some of the routes – the obvious idea is to update this prac system to require authentication prior to use of the update route.
- Next week we will release a worksheet on securing the system using TLS and serving via https.

Both of these worksheets will be valuable preparation for assignment 2, but you can come back to them as needed.

A quick potential gotcha before we start:
The latest versions of MySQL (v. 8.x.x) have introduced a new authentication scheme which does not appear to be correctly handled by the node `mysql` library. This refers more to the prac from last time, but if you see this error:

```
ER_NOT_SUPPORTED_AUTH_MODE
```

then you should take a look at the quick fix here:

https://stackoverflow.com/questions/50093144/mysql-8-0-client-does-not-support-authentication-protocol-requested-by-server.

This seems to do the trick nicely.

DB Connectivity

In the app from last time, we organised a connection to the database by direct use of the `mysql` module installed via npm. We supplied this to the application using simple custom middleware. Here we are going to use a similar middleware approach, but rely on the knex package to handle the connection and some of the queries. Knex greatly simplifies the creation and management of DB connectivity. There are numerous good resources for this package available on the web, and these include:

- The project home page: https://knexjs.org/
- The project NPM guide: https://www.npmjs.com/package/knex, and
- A setup gist: https://gist.github.com/NigelEarle/80150ff1c50031e59b872baf0e474977.

In what follows we will be closest to the gist, but we are going to simplify things. The knex docs show you how to define the configurations so that we can have a different setup for each of development and production and other phases of the project. In these cases, we would normally also install knex globally in order that we might use it as a command line tool and create a more general knexfile. This is covered nicely in the gist. In our case, we are going to work with a simple knexfile which just maintains a single connection. We do not worry at this point with a database pool, though as you will see from the docs, this too is easy to generate.

We begin by installing the knex package locally in the project directory. Even if we install knex globally, we would still go ahead and install locally using the command below:

```
$ npm install knex --save
```

We now consider the knexfile. You should follow the gist link above and have a good look at the structure of a full knexfile, so that you can see how ours removes things that we don't need. In particular, we don't really care about the stages of the project lifecycle, and we won't cater for migrations and seeds.

The following code shows a basic configuration to specify our World Cities DB connection:

```
module.exports = {
    client: 'mysql',
    connection: {
      host: '127.0.0.1',
      database: 'world',
      user:     'root',
      password: 'XXXXX'
    }
}
```

You should create a file called `knexfile.js` in the project root directory and enter this code, making whatever changes to the JSON are needed for your setup, with the password being the most obvious focus.

We now turn our attention to the app itself. Regardless of the architecture of your app, you need to ensure that the DB connectivity is organised early and high up in the paths so that it is available widely. For this reason, we would normally deal with connectivity at the application or general router level. As with the first version of this project, we will organise the connection in the file app.js.

We first remove the old database connection code in this file, which you will find in the declarations and in the early app calls:

```
const db = require('./database/db');
…
app.use(db);
```

We then include the knex related code – here we will place this above the first route calls near the bottom of the other pre-requisite `app.use()` calls.

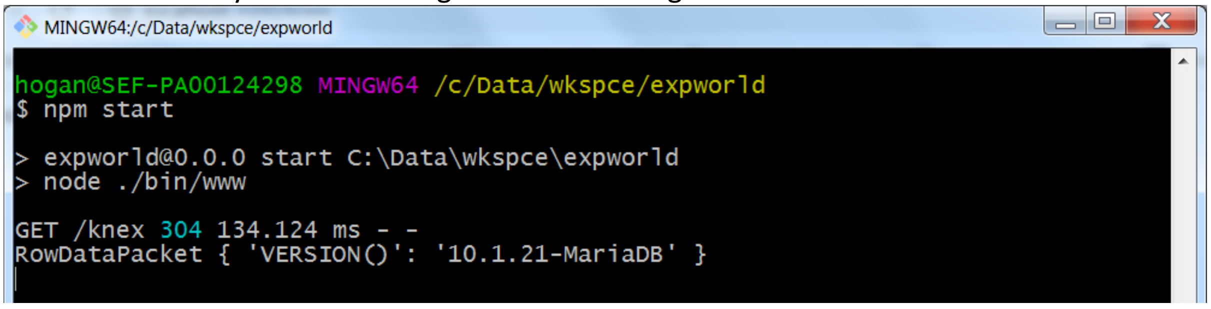
```

const options = require('./knexfile.js');
const knex = require('knex')(options);

app.use((req, res, next) => {
  req.db = knex
  next()
})
```

Here the values are set in the knexfile. The `app.use` call is very similar to the one you saw last time, but instead of working with a connection object it stores a reference to the `knex` object on the `request`.

To test whether your connection is working, introduce a simple route like the one below:

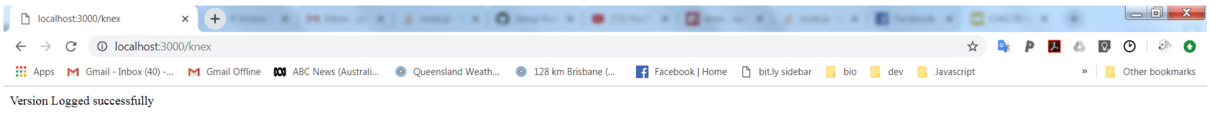
```

app.get('/knex', function(req,res,next) {
    req.db.raw("SELECT VERSION()").then(
        (version) => console.log((version[0][0]))
    ).catch((err) => { console.log( err); throw err })
    res.send("Version Logged successfully");
});

```

Here the console yields something like the following:


MINGW64:/c/Data/wkspce/expworld

```
hogan@SEF-PA00124298 MINGW64 /c/Data/wkspce/expworld
$ npm start

> expworld@0.0.0 start C:\Data\wkspce\expworld
> node ./bin/www

GET /knex 304 134.124 ms - -
RowDataPacket { 'VERSION()': '10.1.21-MariaDB' }
```

And the GET response to the page is a simple text string, as seen on the following page:



```
Version Logged successfully
```

This is simply for test purposes and you can now delete it, but it is also helpful in showing the use of the knex 'raw' mode, in which you simply pass through some standard SQL commands.

In what follows we will go through to the index routes found in `index.js`. The approach is very similar to what we have seen before, but we no longer need the mysql module (we can delete the require statement) and we will work with the standard knex promise-based syntax found in the tutorials. Here we build on top of the knex object to create a query for columns in the city table as before. The query is simply:

```
SELECT name, district FROM city;
```

In knex form, we have the following:

```
knex.from('city').select('name','district')
```

where the remaining code shown below handles the output and error conditions through the usual chaining of promises with which you are now familiar. These constructed queries are safe and we do not need to use a prepared statement to guard against SQL injection. The `/api/city` route is handled as follows:

```
router.get("/api/city", function(req,res, next) {
  req.db.from('city').select("name", "district")
    .then((rows) => {
       res.json({"Error" : false, "Message" : "Success", "City" : rows})
    })
    .catch((err) => {
      console.log(err);
      res.json({"Error" : true, "Message" : "Error in MySQL query"})
    })
});
```

Note here the use of the `req.db` object property instead of the knex identifier, which is of course out of scope here – the connection is propagated via middleware as in the earlier prac sheet.

Our next step is to deal with the other query route: `/api/city/:CountryCode`. This is again straightforward, but we add an additional clause in the query. The raw SQL query looks something like this:

```
SELECT * FROM city WHERE CountryCode=param;
```

In knex form, we have the following:

```
knex.from('city').select('*').where('CountryCode','=',req.para
ms.CountryCode)
```

As before, you need to replace the knex references with `req.db`. Note how easily the knex query builder handles the standard cases. You will find this very convenient for most of the requirements of the assignment API. The code follows below:

```
router.get("/api/city/:CountryCode",function(req,res,next) {
  req.db.from('city').select('*').where('CountryCode','=',req.params.CountryCode)
    .then((rows) => {
      res.json({"Error" : false, "Message" : "Success", "Cities" : rows})
    })
    .catch((err) => {
      console.log(err);
      res.json({"Error" : true, "Message" : "Error executing MySQL query"})
    })
});
```

Logging

By default, the Express generator includes a reference to the `morgan` logging package, which allows us to write logging middleware quickly and to use it to keep track of everything that is happening in each of the requests that we handle. There are two key aspects to understand:

- Morgan needs tokens to log. Some are built in, and some we need to provide. In particular, we will spend time organising request and response headers.
- Morgan needs a format string to display them, which we provide in the middleware. There are standard format strings, and we can also use a custom format.

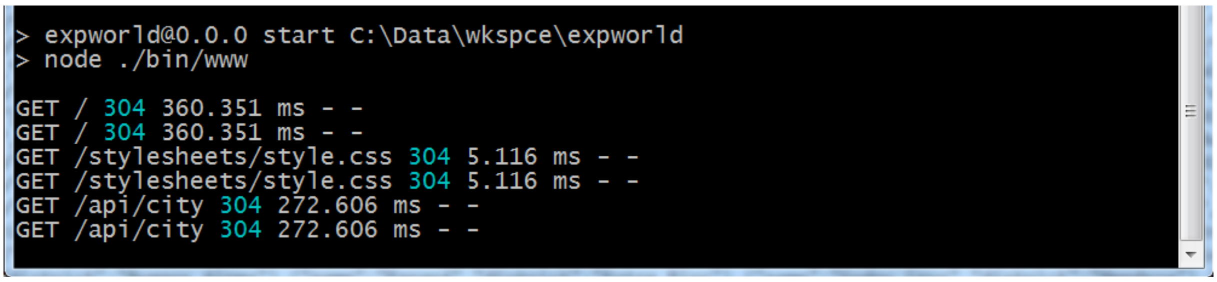
The docs for morgan are really pretty good but a little confusing to the novice. You can find much the same information at the github and npm sites below:

- Github: https://github.com/expressjs/morgan
- NPM: https://www.npmjs.com/package/morgan

There is a gentler blog introduction here https://medium.com/@tobydigz/logging-in-a-node-express-app-with-morgan-and-bunyan-30d9bf2c07a which some of you may find useful.

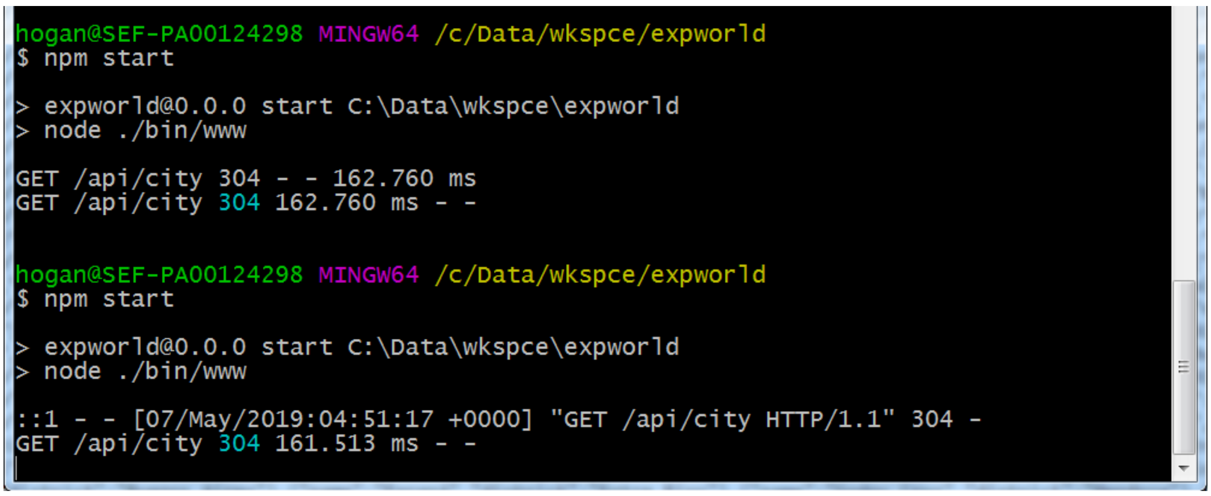
In the example below we are going to set up the logger – the variable is called `logger` by the Express generator – and use the standard `dev` format string:

```
app.use(logger('dev'));
```



```
> expworld@0.0.0 start C:\Data\wkspce\expworld
> node ./bin/www

GET / 304 360.351 ms - -
GET / 304 360.351 ms - -
GET /stylesheets/style.css 304 5.116 ms - -
GET /stylesheets/style.css 304 5.116 ms - -
GET /api/city 304 272.606 ms - -
GET /api/city 304 272.606 ms - -
```

You should explore the use of a number of the alternatives shown in the Github docs for the package: `combined`, `common`, `dev`, `short` and `tiny`. The screenshot below shows the results for `tiny` and `common` for some GET requests:

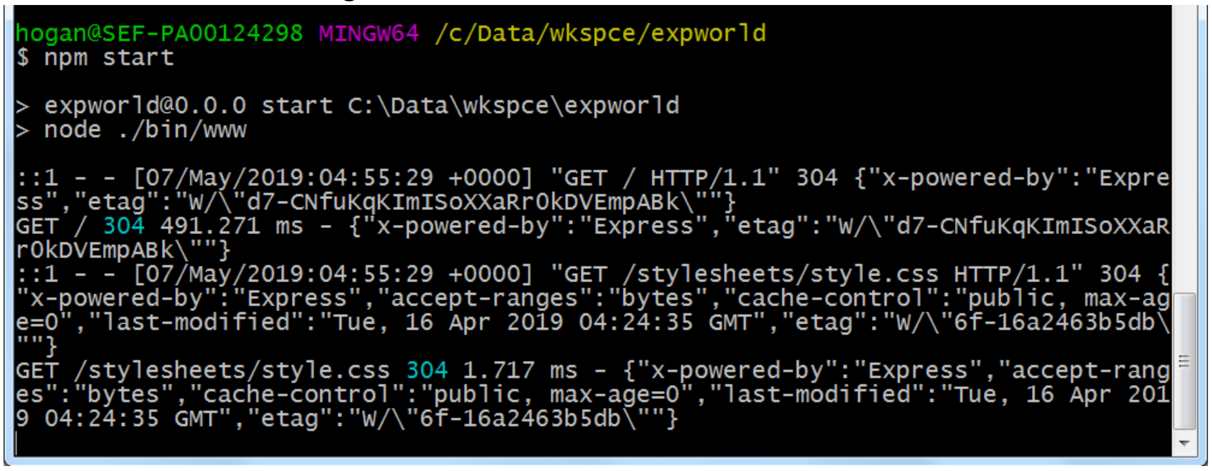


```
hogan@SEF-PA00124298 MINGW64 /c/Data/wkspce/expworld
$ npm start

> expworld@0.0.0 start C:\Data\wkspce\expworld
> node ./bin/www

GET /api/city 304 - - 162.760 ms
GET /api/city 304 162.760 ms - -


hogan@SEF-PA00124298 MINGW64 /c/Data/wkspce/expworld
$ npm start

> expworld@0.0.0 start C:\Data\wkspce\expworld
> node ./bin/www

::1 - - [07/May/2019:04:51:17 +0000] "GET /api/city HTTP/1.1" 304 -
GET /api/city 304 161.513 ms - -
```

Finally, we will take a leaf from other logging code and show how to add the headers to the list of tokens to be processed. The approach is covered in the docs and in the blog linked above, and is very straightforward. The complications in the code below are due to the need to handle each of the headers and create an array, and then to stringify the collection.

```
logger.token('req', (req, res) => JSON.stringify(req.headers))
logger.token('res', (req, res) => {
  const headers = {}
  res.getHeaderNames().map(h => headers[h] = res.getHeader(h))
  return JSON.stringify(headers)
})
```

These tokens are added to the collection tracked by the logger. We can see the output using the `common` format string:



```
hogan@SEF-PA00124298 MINGW64 /c/Data/wkspce/expworld
$ npm start

> expworld@0.0.0 start C:\Data\wkspce\expworld
> node ./bin/www

::1 - - [07/May/2019:04:55:29 +0000] "GET / HTTP/1.1" 304 {"x-powered-by":"Expre
ss","etag":"W/\"d7-CNfuKqKImISoXXaRrOkDVEmpABk\""}
GET / 304 491.271 ms - {"x-powered-by":"Express","etag":"W/\"d7-CNfuKqKImISoXXaR
rOkDVEmpABk\""}
::1 - - [07/May/2019:04:55:29 +0000] "GET /stylesheets/style.css HTTP/1.1" 304 {
"x-powered-by":"Express","accept-ranges":"bytes","cache-control":"public, max-ag
e=0","last-modified":"Tue, 16 Apr 2019 04:24:35 GMT","etag":"W/\"6f-16a2463b5db\
""}
GET /stylesheets/style.css 304 1.717 ms - {"x-powered-by":"Express","accept-rang
es":"bytes","cache-control":"public, max-age=0","last-modified":"Tue, 16 Apr 201
9 04:24:35 GMT","etag":"W/\"6f-16a2463b5db\""}
```

Note the additional header output – this is the homepage GET as you can see from the log.

Swagger Docs

This section is concerned with the API docs using Swagger. As we saw in the lecture, it is relatively straightforward to engage the Swagger machinery. To begin, we will replicate what we had in last week's lecture. We must first require the Swagger packages in the `app.js` file, and of course install them locally using npm. I will not show the latter step.

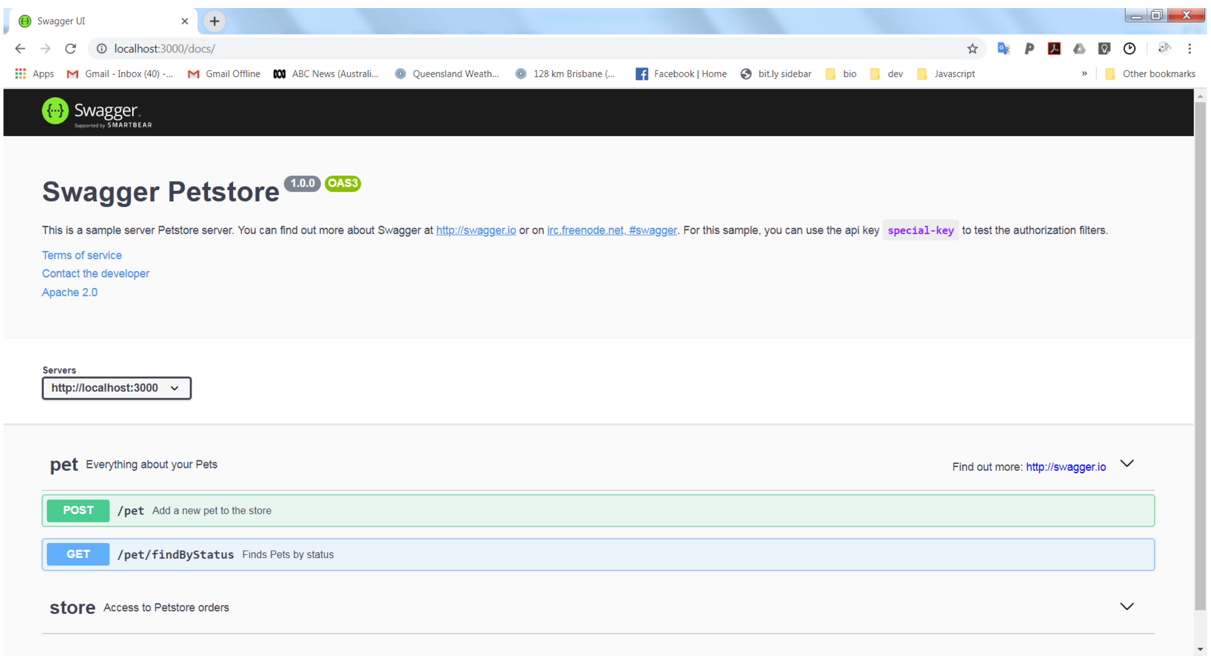
The require statements go near the top above the app declaration:

```
const swaggerUI = require('swagger-ui-express');
const swaggerDocument = require('./docs/swaggerpet.json');
```

We will update the name of the json file later. As you can see, we have created a subdirectory called docs in which we have placed the swagger file. As there is only the one doc related route to be handled, we will deal with this at the application level with the other high level routes. The placement here is not especially important in this case, and the Swagger code can be regarded as boiler plate:

```
app.use('/docs', swaggerUI.serve, swaggerUI.setup(swaggerDocument))
```

If we go ahead and navigate to the docs route, we see:



Swagger UI

localhost:3000/docs/

Swagger Petstore 1.0.0 OAS3

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

Terms of service
Contact the developer
Apache 2.0

Servers
http://localhost:3000

pet Everything about your Pets — Find out more: http://swagger.io

POST `/pet` Add a new pet to the store

GET `/pet/findByStatus` Finds Pets by status

store Access to Petstore orders

We will now hack the JSON file so as to provide an API description that matches the World City API. Note that if you are comfortable with YAML you should feel free to use that instead. You should refresh your memory of the routes by taking a quick look in `app.js` and `routes/index.js`. In `app.js` we see the following:

```
app.use('/', indexRouter);
app.use('/users', usersRouter);
app.use('/docs', swaggerUI.serve, swaggerUI.setup(swaggerDocument))
```

We will ignore the `'/users'` route for now (see the Server-side JWT worksheet) and consider the others. The main routes are handled in the `indexRouter`, and we consider `routes/index.js` next:

```
router.get('/', function(req, res, next)
router.get('/api', function(req, res, next)
router.get("/api/city",function(req,res, next)
router.get("/api/city/:CountryCode",function(req,res,next)
```

All of these routes have at least a basic response – the `'/api'` route is a placeholder, but the others are real. So we turn our attention to the Swagger JSON. Our efforts here will seem too professional for the trivial API deployed, but the point of the exercise is to prepare you for the assignment. We will take a copy of the Pet Shop example and rename it to swagger.json, and then edit it as appropriate. The first section to consider is the information object at the head of the file. Note the initial Open API version and JSON opening brace before we move on to the info section itself:

```
{ "openapi":"3.0.0",
    "info":{
        "title":"World City API",
        "description":"This is a simple Express API based on the World Cities
Database. It supports basic city and province listings, along with filtering
based on Country Code",
        "version":"1.0.0",
        "termsOfService":"http://swagger.io/terms/",
        "contact":{"email":"j.hogan@qut.edu.au"},
        "license":{
            "name":"Apache 2.0",
            "url":"http://www.apache.org/licenses/LICENSE-2.0.html"
        }
    },
```

We next move on to the server array, which remains unchanged, and update the base path to `"/api"`:

```
    "servers":[
        {
            "url": "http://localhost:3000"
        }
    ],
    "basePath":"/api",
```

The first real decision is to choose the tags – essentially the categories in which we will place the routes. Here we will split into Information and Query, where the first one greatly overstates the service. We include some basic descriptions for both of them, though we do not have any external docs to offer, so that block is deleted.

```
  "basePath":"/api",
    "tags":[
        {
          "name":"information",
          "description":"General API description"
        },
        {
          "name":"query",
          "description":"API queries based on city and optional country code."
        }
    ],
```

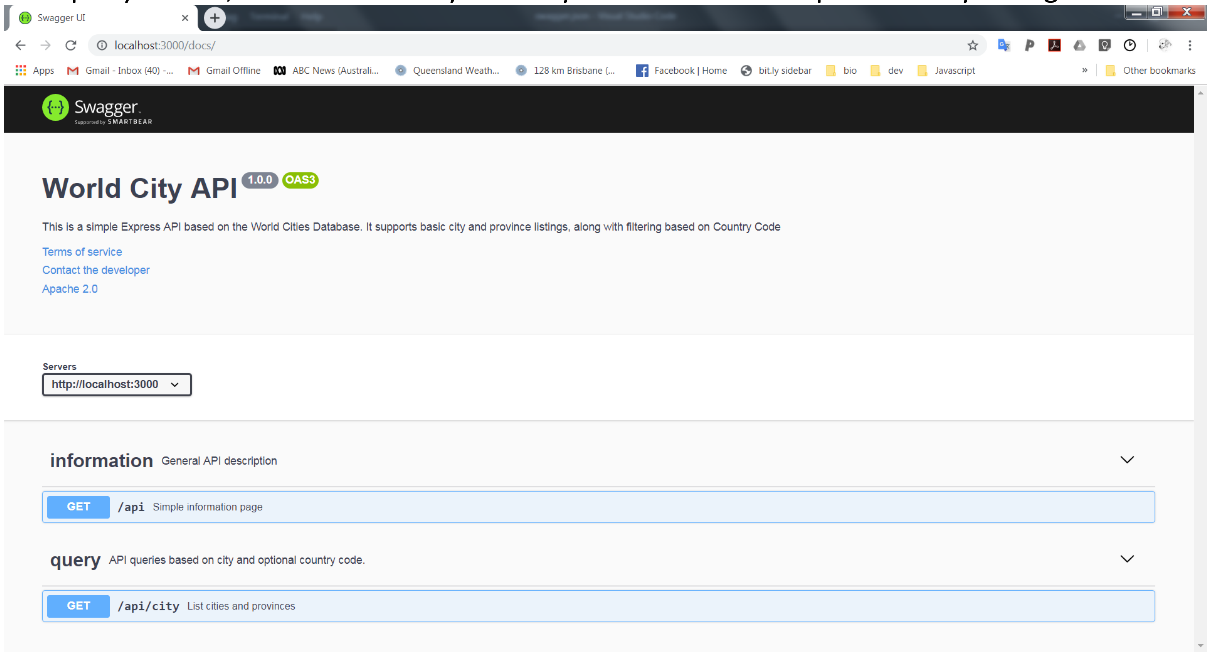
Our first route turns out to be a rather dumb, bare API home page. This is a good place to start as there are no parameters and only a very simple response. From the base path onwards, our file now looks something like this:

```
    "basePath":"/api",
    "tags":[
        {
          "name":"information",
          "description":"General API description"
        },
        {
          "name":"query",
          "description":"API queries based on city and optional country code."
        }
    ],
    "paths":{
        "/":{
            "get":{
                    "tags":["information"],
                    "summary":"Simple information page",
                    "description":"Very basic API home page promising more 
information",
                    "operationId":"apiHome",
                    "produces":["text/html"],
                    "responses":{
                        "200":{
                            "description":"successful operation"
                        }
                    }
            }
        }
    }
}
```

Saving and reloading yields an updated version of the docs page, one that allows the user to try out the API GET. We will now introduce an additional specification to cover the `/city` route of the API. This code is a great deal more complex, as we shall see on the following page. The main complexity here lies in the hierarchy of the datatypes returned – the complex JSON that ends with an array of cities in which each item is itself another JSON object.

```
"/api/city":{
            "get":{
                    "tags":["query"],
                    "summary":"List cities and provinces",
                    "description":"Full listing of cities and their provinces
for the database",
                    "operationId":"listCitiesAndDistricts",
                    "produces":["application/json"],
                    "responses":{
                        "200":{
                            "description":"successful operation",
                            "schema":{
                                "type":"object",
                                "properties":{
                                    "Error":{
                                        "type":"string",
                                        "enum":["true","false"]
                                    },
                                    "Message":{
                                        "type":"string",
                                        "enum":["Success","Failure"]
                                    },
                                    "City":{
                                        "type":"array",
                                        "items":{
                                            "type":"object",
                                            "properties":{
                                                "name":{
                                                    "type":"string"
                                                },
                                                "district":{
                                                    "type":"string"
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
        }
```

After reloading and restarting the server, we see that the docs pages now exhibit information and query routes, and that we may even try out the more complicated city listing route.



World City API 1.0.0 OAS3

This is a simple Express API based on the World Cities Database. It supports basic city and province listings, along with filtering based on Country Code

Terms of service
Contact the developer
Apache 2.0

Servers: http://localhost:3000

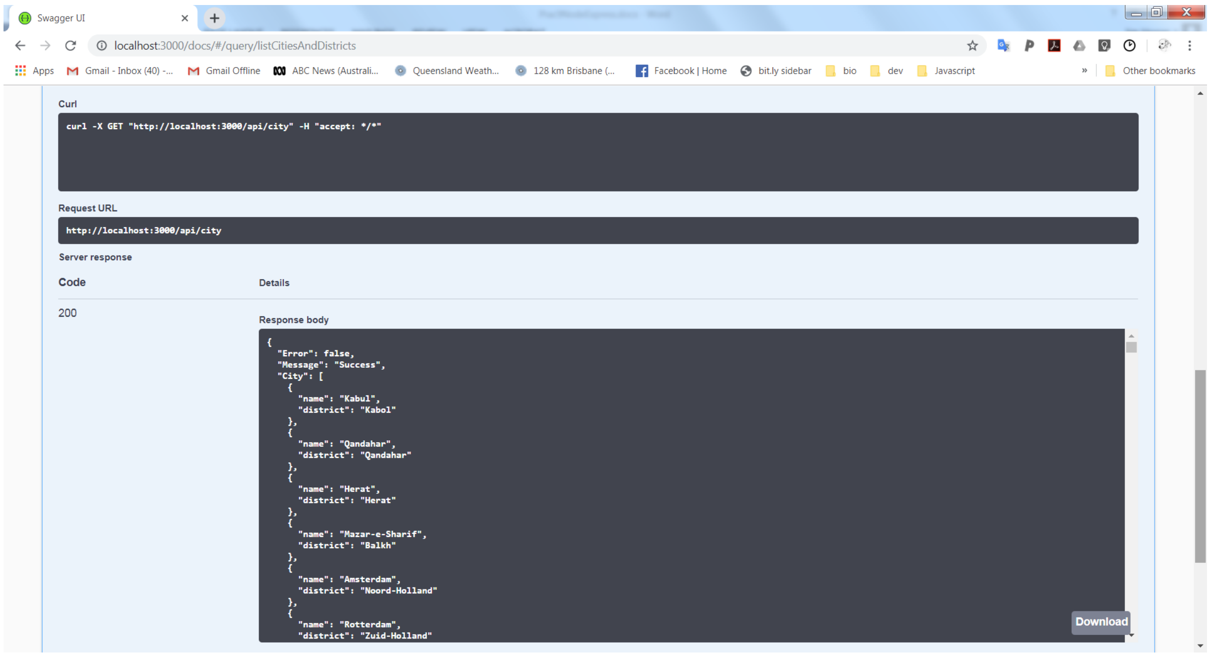
information General API description

GET /api Simple information page

query API queries based on city and optional country code.

GET /api/city List cities and provinces

When we click on the `/api/city` route, and follow the buttons to try and the execute the query, we see results something like the following, which match the schema supplied above:



Curl

```
curl -X GET "http://localhost:3000/api/city" -H "accept: */*"
```

Request URL

```
http://localhost:3000/api/city
```

Server response

Code	Details
200	Response body

```
{
  "Error": false,
  "Message": "Success",
  "City": [
    {
      "name": "Kabul",
      "district": "Kabol"
    },
    {
      "name": "Qandahar",
      "district": "Qandahar"
    },
    {
      "name": "Herat",
      "district": "Herat"
    },
    {
      "name": "Mazar-e-Sharif",
      "district": "Balkh"
    },
    {
      "name": "Amsterdam",
      "district": "Noord-Holland"
    },
    {
      "name": "Rotterdam",
      "district": "Zuid-Holland"
```

Our final step – which we leave to you as an exercise – is to follow the steps we have outlined above to produce some docs for the `/api/city/:CountryCode` route. This one has a lot in common with the `/api/city/` route but it also requires that we consider the parameters supplied to the method. You should work from our example above and also from the Swagger Pet Shop example. You can verify your solution by looking at the results obtained.

A POST Example

In most of the work we have done so far we have handled only GET requests. These are simpler, we are requesting information that is already available at the site, and we can verify that they work by typing the query into the browser address field. However, sometimes we want to update information on the server. Generally we will do this in a carefully controlled way, so we are going to be cautious and assume that the update requires a form of some kind, and that the data we receive is coming in URLEncoded form as the result of a POST request. We will not use parameters as in the `/api/city/CountryCode` endpoint. Our data will consist of `<key,value>` pairs parsed from the body, in requests that will look something like this one:

http://localhost/api/update?City=Shanghai&CountryCode=CHN&Pop=24183300

In the example which follows, we are going to allow the user to specify a city and its country, and to then update the population listed in the database. To help us, we are going to use:

https://en.wikipedia.org/wiki/List_of_cities_proper_by_population

This Wikipedia article has a list of the largest cities in the world, and is obviously a good start if we want some interesting numbers to use. More technically, the POST operation under Express requires that we take the body of the request and parse it in order to get the fields we need to update the database. The requests will be similar to those commonly used for registration routes:

```
  fetch(`${API_URL}/api/register`, {
    method: "POST",
    body: 'email=x.xxxx%40xxx.xxx.xx&password=xxxxxx',
    headers: {
      "Content-type": "application/x-www-form-urlencoded"
    }
  })
```

Here we see that the body is consistent with the examples we have seen in form processing, with the string urlencoded. Later on, we will see a JSON-based example which updates the value for Tokyo. As we can use the `body-parser` middleware, parsing will be a lot less painful than it sounds. In the more recent versions of express, the body-parser is included in the application and we do not need to install it separately (see https://github.com/expressjs/express/releases/tag/4.16.0). The relevant code in `app.js` is seen below:

```
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
```

Before proceeding further, we will update the application to allow Cross-Origin Resource Sharing (see the MDN article: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS). The opening paragraph of that piece describes the situation well:

Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin. A web application executes a ***cross-origin HTTP request*** *when it requests a resource that has a different origin (domain, protocol, and port) than its own origin.*

We will resolve potential issues by using a fairly permissive approach to CORS – we might be more careful in a production environment. You will need to install the `cors` package in the application root directory as usual. The use of this middleware is documented at the following link: https://expressjs.com/en/resources/middleware/cors.html. We insert the require statement at the bottom of the group at the top of the page:

```
const helmet = require('helmet');
const cors = require('cors');
```

and then we include the simplest usage statement above the application routing:

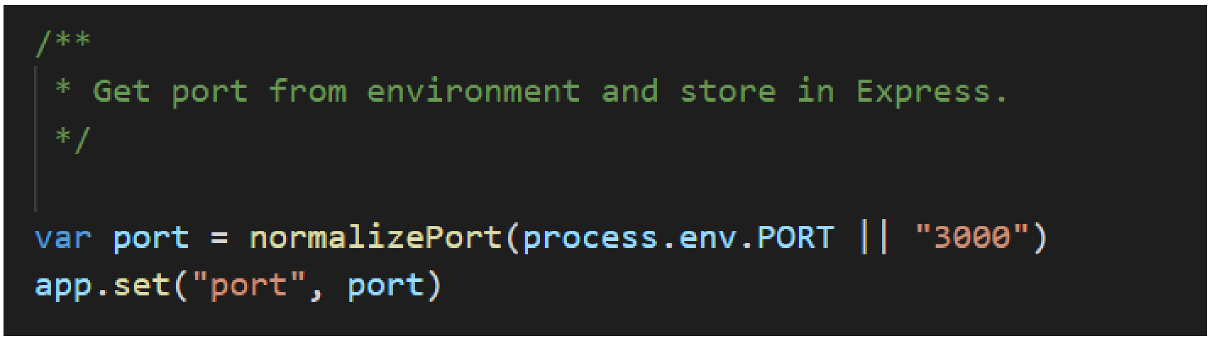
```
app.use(logger('common'));
app.use(helmet());
app.use(cors());
```

We now turn our attention to the file `index.js` in the `routes` directory. We will add another route to the list, this time at `/api/update`. The route handling has the same form as the others, only this time we will be working with a POST rather than a GET. The `body-parser` takes the query parameters and parses them into properties sitting on the request object. In this case, we are going to use three query parameters:

Key	Example Value	Request Object Property
City	Shanghai	`req.body.City`
CountryCode	CHN	`req.body.CountryCode`
Pop	24183300	`req.body.Pop`

We are going to make our updates from a very simple React app which we will supply below. But by default, both of our apps are going to operate on port 3000. So we will need to change one of them. Later, when we serve https we will be listening on port 443, so we will leave React alone and make the changes to the Express server.

If you look in the Express app filesystem you will find a directory called bin, and inside it, you will see a single file called *www*, and we are interested in this section of code:

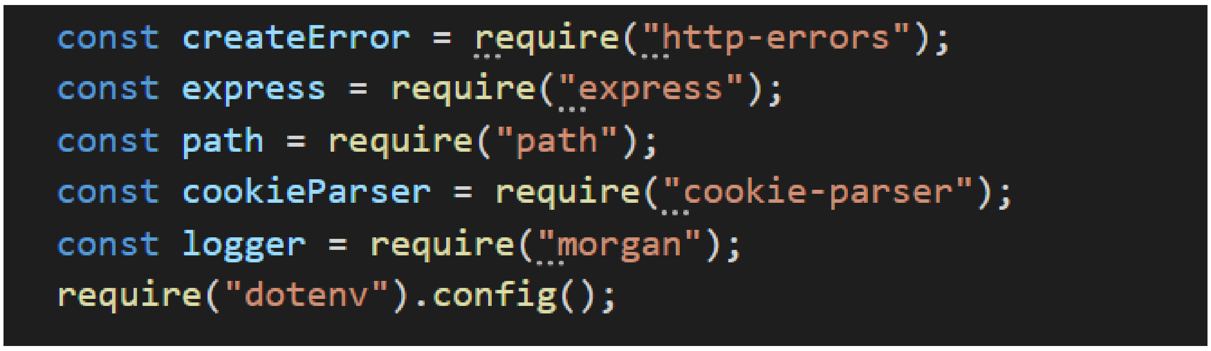


```
/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || "3000")
app.set("port", port)
```

Here the environment variable `process.env.PORT` may be set from a file called .env in the top directory of the application. We need to do two things. First, we create the file `.env` - making sure to do so in the top directory of your application. It should contain a single line as follows:

```
PORT=3001
```

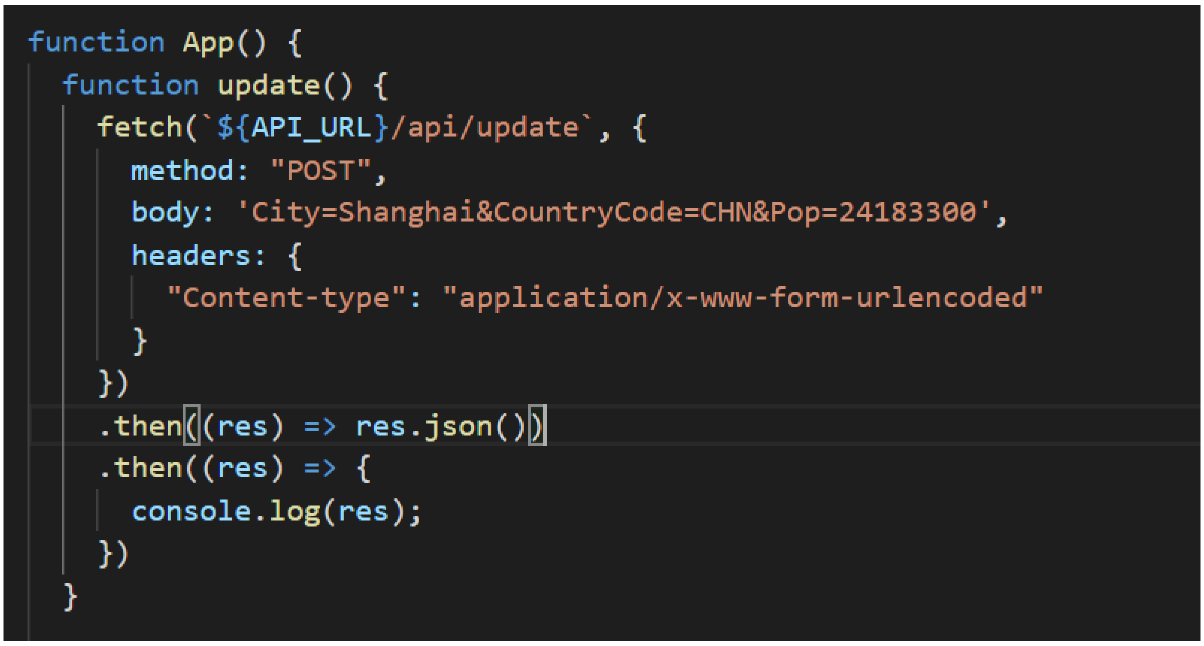
This setting is made available to the app by including the `dotenv` package in the file app.js as shown below:



```
const createError = require("http-errors");
const express = require("express");
const path = require("path");
const cookieParser = require("cookie-parser");
const logger = require("morgan");
require("dotenv").config();
```

I have converted the `var`s to `const`s. The final `require` statement makes the environment available to the app, and the result is that the server will listen not on the default setting of 3000, but on our new choice of 3001.

In the starter-files you will find an initial version of the file App.js (here App1.js) which will allow you to test the update route. You will need to set up a react app - use `create-react-app` as usual - and then replace the default App.js with the file provided, keeping the name App.js. The main function will look something like the one below:



```
function App() {
  function update() {
    fetch(`${API_URL}/api/update`, {
      method: "POST",
      body: 'City=Shanghai&CountryCode=CHN&Pop=24183300',
      headers: {
        "Content-type": "application/x-www-form-urlencoded"
      }
    })
    .then((res) => res.json())
    .then((res) => {
      console.log(res);
    })
  }
  return (
    <div className="App">
      <h1>The DB Upload Example</h1>

      <button onClick={update}>Update</button>
    </div>
  );
}
```

Once we click the button, we will simply send off the POST and log the response - we are not especially interested in displaying the results this time.

We will work through the route code on the server side in stages. Once more we are in the router `index.js`. The conditional at the top is there to check that each of the parameters are set, that they have been parsed successfully from the request. The message strings have been shortened for space reasons.

```
router.post('/api/update', (req, res) => {
  if (!req.body.City || !req.body.CountryCode || !req.body.Pop) {
    res.status(400).json({ message: `Error updating population` });
    console.log(`Error on request body:`, JSON.stringify(req.body));

  } else {
```

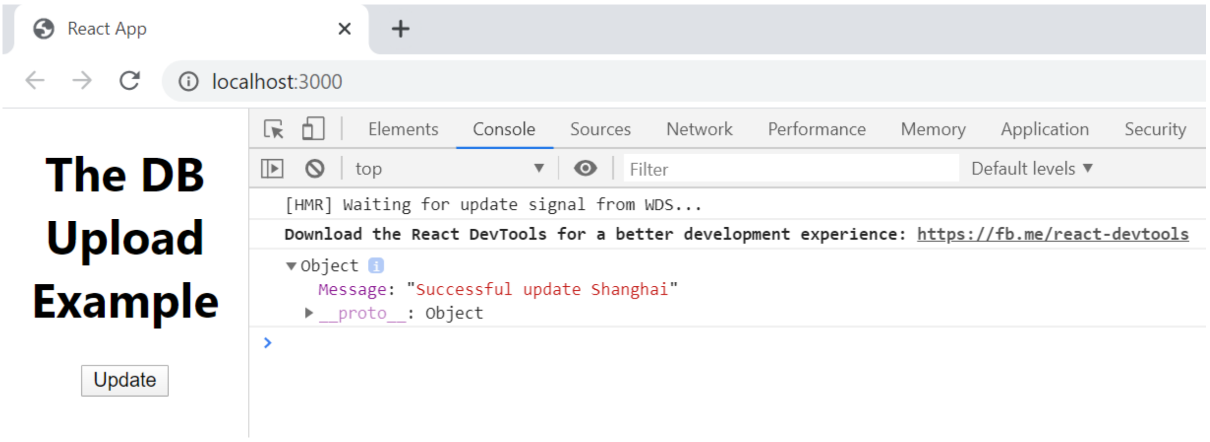
Just below the else clause, we define two objects, one we call `filter` – which contains the city name and country code in form that can be used in the `where` clause - and the other, `pop` specifies the population value for the table update:

```
  } else {
    const filter = {
      "Name":req.body.City,
      "CountryCode":req.body.CountryCode
    };

    const pop = {
      "Population":req.body.Pop
    }
```

The most important line is then the `knex` query which follows below. Here we take the `city` table and use the `filter` object to limit consideration to the single row containing `Shanghai` and `CHN`. This check is important as there are of course duplicate city names in the table. Here we ensure we update the right one. The `pop` object specifies the update value, as noted above. The `knex` cheat sheet at https://devhints.io/knex is very good.

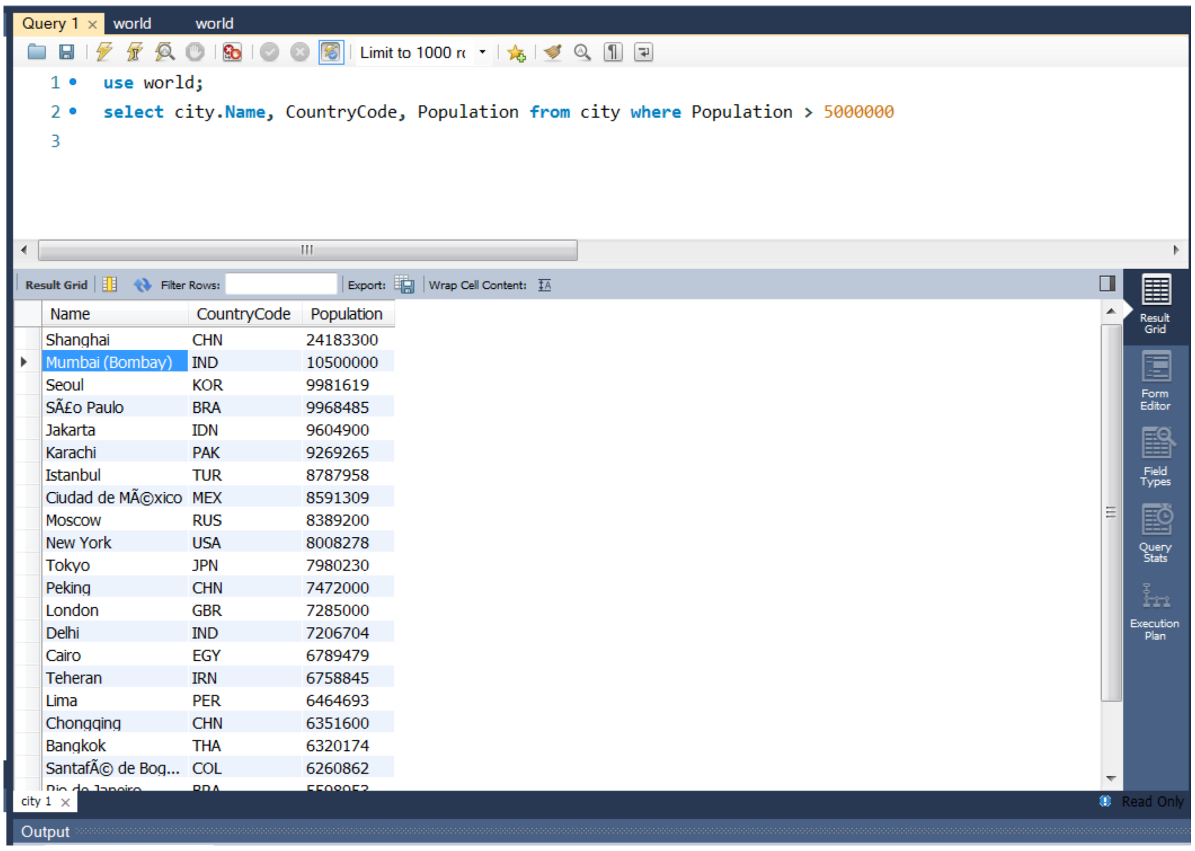
```
    req.db('city').where(filter).update(pop)
      .then(_ => {
        res.status(201).json({ message: `Successful update ${req.body.City}`});
        console.log(`successful population update:`, JSON.stringify(filter));
    }).catch(error => {
        res.status(500).json({ message: 'Database error - not updated' });
    })
  }
});
```

In the end, we start up the app and click on the button. We can check the console log and we will see that the response is appropriate:



```
[HMR] Waiting for update signal from WDS...
Download the React DevTools for a better development experience: https://fb.me/react-devtools
▼Object
    Message: "Successful update Shanghai"
  ▶ __proto__: Object
```

We can verify the change through the database workbench or the mysql command line:



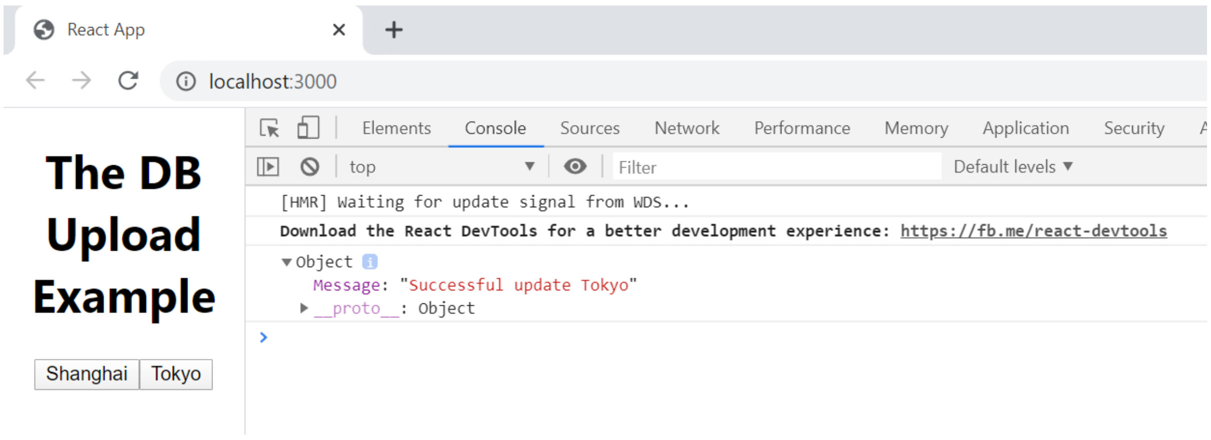
```
use world;
select city.Name, CountryCode, Population from city where Population > 5000000
```

Name	CountryCode	Population
Shanghai	CHN	24183300
Mumbai (Bombay)	IND	10500000
Seoul	KOR	9981619
SÃ£o Paulo	BRA	9968485
Jakarta	IDN	9604900
Karachi	PAK	9269265
Istanbul	TUR	8787958
Ciudad de MÃ©xico	MEX	8591309
Moscow	RUS	8389200
New York	USA	8008278
Tokyo	JPN	7980230
Peking	CHN	7472000
London	GBR	7285000
Delhi	IND	7206704
Cairo	EGY	6789479
Teheran	IRN	6758845
Lima	PER	6464693
Chongqing	CHN	6351600
Bangkok	THA	6320174
SantafÃ© de Bog...	COL	6260862
Rio de Janeiro	BRA	5598953

So far so good, but most POST methods feature a body containing JSON objects in stringified form. To show how to manage this request, we can add another button to the React app. You should now replace the first version of App.js with App2.js from the starter files, once more keeping the name App.js. The function to perform the Tokyo update is as follows:

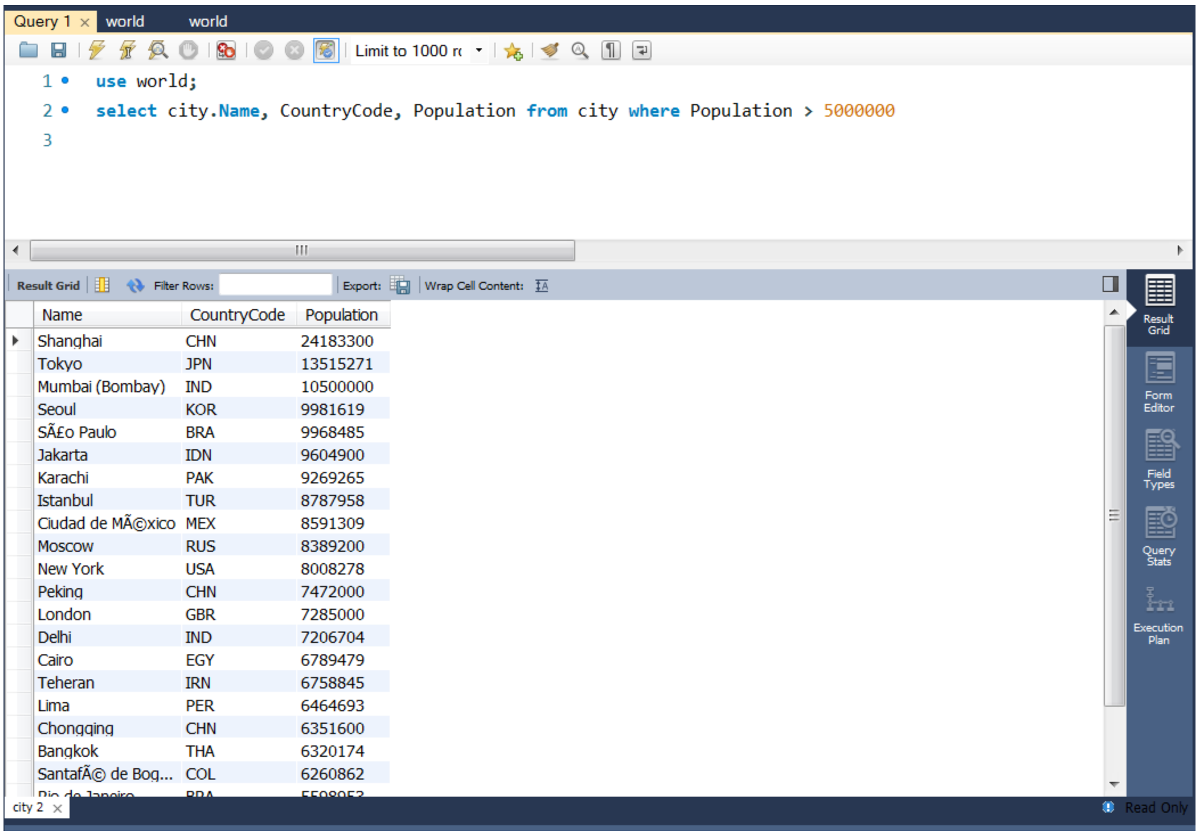
```
function updateTokyo() {
  fetch(`${API_URL}/api/update`, {
    method: "POST",
    body: JSON.stringify({
          "City":"Tokyo",
          "CountryCode":"JPN",
          "Pop":"13515271"
    }),
    headers: {
      "Content-type": "application/json"
        }
  })
  .then((res) => res.json())
  .then((res) => {
    console.log(res);
  })
}
```

Note the different format of the body and the correspondingly different content type. The React app will update without the need to restart, and we will then click on the Tokyo button. As before, the console shows that the update has been successful:



```
[HMR] Waiting for update signal from WDS...
Download the React DevTools for a better development experience: https://fb.me/react-devtools
▼Object
   Message: "Successful update Tokyo"
  ▶__proto__: Object
```

And we can again verify the changes in the database itself:



```sql
use world;
select city.Name, CountryCode, Population from city where Population > 5000000
```

Name	CountryCode	Population
Shanghai	CHN	24183300
Tokyo	JPN	13515271
Mumbai (Bombay)	IND	10500000
Seoul	KOR	9981619
SÃ£o Paulo	BRA	9968485
Jakarta	IDN	9604900
Karachi	PAK	9269265
Istanbul	TUR	8787958
Ciudad de MÃ©xico	MEX	8591309
Moscow	RUS	8389200
New York	USA	8008278
Peking	CHN	7472000
London	GBR	7285000
Delhi	IND	7206704
Cairo	EGY	6789479
Teheran	IRN	6758845
Lima	PER	6464693
Chongqing	CHN	6351600
Bangkok	THA	6320174
SantafÃ© de Bog...	COL	6260862

You may also hit these and other endpoints using the Postman tool. See: [https://www.getpostman.com/downloads/.](https://www.getpostman.com/downloads/)

Your remaining task is to return to the Swagger file, and update the JSON to reflect the new route that you have added. As noted earlier, there will be a new worksheet extending this system to support users and authenticated routes, and then finally another to convert the system to support TLS and to serve https.