Project 1 - Part 2
Due: 23:59 May 6
Submission: Please submit a readme file, your curl test commands, and a zip of your project directory (source for the website + service) through the online submission portal.
This project is worth 15%

Overview

You are required to extend Part 1 of this project by creating a REST service that provides your website with content. Using Node.js and Express, two industry standards, you will design and implement a web service with a RESTful interface. You will need to modify your Part 1 solution to use ajax calls to communicate with the service. The service should be supported by a postgresql database.

This project will take place in four parts:

1. Create and run a Node.js service locally to serve your website content.
2. Connect your website with your server via appropriate ajax calls.
3. Construct a Postgresql database to maintain the state of the service.
4. Host the service on the cloud service, Heroku.

Marking

The project will be marked out of 100. The following describes the distribution of marks.

Core (65%)

- (20%) Define an appropriate REST interface for your server
- (20%) Implement a Node.js service with the REST interface
- (20%) Implement Ajax calls to send and receive data from your website to the service
- (5%) Define a set of test cases which demonstrate your REST interface works.

Completion (25%)

- (10%) Create a local postgres database to support your service
- (10%) Use the database to support your REST service.
- (5%) Implement appropriate error catching

Challenge (10%)

- (10%) Deploy the service to Heroku

Background

Node.js is an open source runtime environment for creating web applications. It is optimised for scalability and throughput. Node.js is used extensively by industry leaders, including Netflix and Linkedin. You use Node.js to run a web application.

More information and documentation can be found at:
https://nodejs.org/en/

Express.js is a Node.js framework for building web applications. It is the defacto standard for building Node.js applications and provides utility methods for accelerating their creation. You use Express.js to build a web application.

More information and documentation can be found at:
http://expressjs.com/

Postgresql is an object-oriented relational database management system. It is primarily used as a database server to store data securely and enables retrieval of data through SQL. It is designed to handle workloads ranging from a single user to enterprise level web applications. Using SQL you can create, insert, delete, and query data in a postgresql database.

Heroku is Platform as a Service (PaaS) cloud provider. It allows you to host web applications of various languages. You can use it as a git repository to upload your content. It also provides the capability to host a postgres database.

Part 1: Create a Node.js service

Design and implement a simple REST service on your local machine. The lab machines have Node.js installed on them. If you are running this from home, you will need to install Node.js before running your service.

1.1 Building the service:

Create a working directory for the Node.js service and use npm init to perform the initial setup. It will ask you for configuration settings, either set these or press enter to use the default options.

> npm init

Now install Express.js in the directory (this may already be done by the lab machines):

> npm install express --save

Include a body parser so you can pull text/JSON out from requests:

> npm install body-parser --save

1.2 Create an index.js file for your service:

Enter the following:

```
var express = require('express');
var app = express();
var port = process.env.PORT || 8080;

var bodyParser = require('body-parser')
app.use( bodyParser.json() );       // to support JSON-encoded bodies
app.use(bodyParser.urlencoded({     // to support URL-encoded bodies
  extended: true
}));

//Accessible at localhost:8080/
app.get('/', function (req, res) {
  res.send('Hello World!');
});

//Accessible at localhost:8080/get/tasks/
app.get('/get/tasks/', function (req, res) {
  res.send('This is a task.');
  // Extend this later to return tasks from the database.
});

app.listen(port, function () {
  console.log('Example app listening on port 8080!');
});
```

1.3 Run the service with node:

> node index.js

That should bring up your service with the console log saying it is listening on port 8080.

1.4 Testing and debugging your service:

There are a few ways you can test to see if your service is working as intended. Web based may be easiest to start with, but as you start posting data to your service it becomes more

complicated to accurately and reliably test. Curl is a command line tool that you can use to perform HTTP requests and pass data to your service.

Web based:
http://localhost:8080/

Curl:
> curl <lab machine name or ip address>:8080

Curl can also be used to perform other HTTP operations, such as POST and PUT. E.g.,

> curl -H "Content-Type: application/json" -X POST -d '{"username":"xyz","password":"xyz"}' http://localhost:3000/api/login

Curl can't use localhost on the lab machines due to the ECS proxy. Instead, put in the name of the computer you are running on (or its ip address). You can still use localhost in the browser to refer to your local machine.

To debug errors make use of the console.log() function. You should also use your browser's development tools. In chrome this is accessed with Ctrl+Shift+J.

1.5 Define an appropriate REST interface:

This should conform to a RESTful architecture, making use of appropriate resource identifiers and suitable operations. You will need to include functionality to GET todo items, POST items, DELETE items, and PUT items. You may need to check when to use these appropriately.

Verb + URI	**Parameters**	**Description**	**Response status and body (when successful)**	**Response status and body (when failed)**
GET /hello	**name** - name of person to whom to say hello (required)	Simple hello world operation, echoes back name along with a greeting.	**hello_string** - "hello "+name passed to the operation 200 OK	error_string - "missing name" 401 Bad Request

When answering this remember a few things:

- Choose the right HTTP verb. For example, GET for idempotent operations versus POST for non-idempotent operations.

- Remember that you can send parameters with POST HTTP requests and results are returned in the HTTP responses.
- Remember to return URIs where appropriate to allow the application to drill down as necessary.
- When returning HTTP status codes try to use the most appropriate code (see http://en.wikipedia.org/wiki/List_of_HTTP_status_codes -- note do not use “I am a teapot”).
- When returning errors you can include information in the response that could be used to help the client correct the error and resubmit.

1.6 Define test cases:

Define a set of curl-based test cases to demonstrate that your service is working correctly. There should be at least one curl command for each of your function of your REST interface. These must be submitted and will be used for marking.

1.7 Create Ajax calls to utilise your REST service.

Implement appropriate Ajax calls to send and receive data between your website and service.

Below are two example Ajax calls. The first does a simple GET request and the second does a slightly more complicated POST request (sending data).

Please note: The following are example calls and may not work directly out of the box. You may need to find other examples online for how to use and implement the calls appropriately. There are also shorthand requests that use $.get, $.post etc. rather than $.ajax which you may want to look into.

Example 1:
Here is a simple GET request. This does a GET request to localhost:8080/get_tasks. The response is then sent to the ‘redraw’ function if successful, or the ERROR_LOG (which prints the message to the console) if unsuccessful.

```
// Define an error log that will print error messages to the console.
var ERROR_LOG = console.error.bind(console);

// A GET request. If successful, this passesdata to the ‘redraw()’ function
function get_tasks() {
  console.log("Get task.")
    $.ajax({
        url: "http://localhost:8080/get/tasks/"
    }).then(redraw, ERROR_LOG);
}
```

```
// Redraw the two lists
function redraw(data) {
    console.log('redrawing', data);
    });
}
```

Example 2:
Here is an example of posting JSON data with Ajax. JSON.stringify() is used to convert text to a JSON format, which can then be passed via the HTTP request. This has an embedded function (success_func) which is executed if the request is successful.

```
// Define a log to capture the failure messages. This goes in the .then() part of the request.

    function complete_task(task) {
        $.ajax({
            method: 'POST',
            url: 'http://localhost:8080/complete/task/',
            data: JSON.stringify({
                task: task.find('.task').html()
            }),
            contentType: "application/json",
            dataType: "json"
        }).then(
             function success_func(data) {
                  // Function that handles successes
                  console.log('posted data.', data);
            },
            ERROR_LOG);
    }
```

NOTE:
You may experience a problem with cross-site scripting, where you will not be allowed to invoke functions on a service hosted in a different location. I got around this by adding the following to my REST service:

```
// Add headers
app.use(function (req, res, next) {
    // Website you wish to allow to connect
    res.setHeader('Access-Control-Allow-Origin', '*')

    // // Request methods you wish to allow
```

```
    res.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH,
DELETE');

    // Request headers you wish to allow ,
    res.setHeader('Access-Control-Allow-Headers', 'Content-Type,
Access-Control-Allow-Headers');

    // Pass to next layer of middleware
    next();
});
```

OR

Some students have found success using the ‘cors’ library. You should research this if you intend to use it. You will need something along the lines of:

```
>npm install cors --save
```

And

```
var cors = require('cors');
app.use(cors());
```

Part 2. Support the service with a PSQL database

RESTful services should be stateless, meaning it should not have user data stored in variables etc. Instead, it should rely on a database to maintain the state of the service. You are required to build a small postgresql database and have the service read and write content from it.

There will be a lecture on PSQL databases and how to use them in Week 7.

Instructions on how to set up a local psql database on the lab machines can be found on the projects page, called postgres_instructions.pdf.

However, a quick summary is:

1. On a lab machine you need to type ‘need postgres’. The databases are stored on a machine called Depot (not your local host) this is important to know when connecting your service to the database.
2. You can then create a database with ‘createdb <username>_nodejs
3. Then you can connect to it with the ‘psql <dbname>’ e.g. ‘psql ryan_nodejs’.
4. You should change your database password (to something other than your ecs account password) with: ‘alter user <username> with password 'XyZZy';’
5. To exit the postgres interpreter type ‘\q’.
6. To list tables in your database use \dt or \d <table name>.

2.1 Create a simple database:

Your database does not need to be very complex. Connect to your database and create a single table with an id and a task. This can be extended later if necessary.

> create table todo (id serial primary key, item varchar(255));

Insert a value into the table:

> insert into todo (item) values ('first todo item');

Query the table to check it is now in there:

> select * from todo;

You will later need to modify your table to store whether a job has been completed or not. This is done with an SQL update command.

2.2 Connect to your database in your service:

NOTE: You will need to use: “npm install pg-native”

```
var pg = require('pg').native;
var connectionString = "postgres://myuser:mypassword@depot:5432/mydatabase";

var client = new pg.Client(connectionString);
client.connect();
```

Then make a function to query the data and return the result:

```
app.get('/test_database', function(request, response) {
    // SQL Query > Select Data
    var query = client.query("SELECT * FROM todo");
    var results = []
    // Stream results back one row at a time
    query.on('row', function(row) {
        results.push(row);
    });

    // After all data is returned, close connection and return results
    query.on('end', function() {
        response.json(results);
    });
});
```

TESTING:
If this doesn't seem to work, first make sure you are using require('pg').native. Then try connecting to the database with a function that prints error messages.
E.g.:

```
pg.connect(connectionString, function(err, client, done) {
    if(err){
        console.error('Could not connect to the database');
        console.error(err);
        return;
    }
    console.log('Connected to database');
    client.query("SELECT * FROM todo;", function(error, result){
    done();
    if(error){
    }
    console.log(result);
    });
});
```

2.3 Use your database in the service:

Implement appropriate database calls in each of your REST services.

2.4 Implement appropriate error checking

Ensure both your website and service are robust by performing error checking and handling throughout them. You should consider invalid database transactions, invalid ajax requests, invalid ajax responses, etc.

Part 3. Host your TODO service in the cloud

3.1 Host your service on Heroku:

There will be a lecture on Heroku and how to use it in Week 7.

Sign up for Heroku and use the free tier to host your service. You will need to read the Heroku documentation to work out how to do this.

https://www.heroku.com/

Things for submission

You should submit the following through the online portal:

1. A readme file explaining:
    a. How to use your system.
    b. What the REST interface is.
    c. What error handling has been conducted.
2. A zip of your entire project directory, including the source for the web page and REST service.
3. The test cases you defined in Section 1.6.