Assignment Three: Chatroom

In this assignment, you'll use Node, Express, and a selection of handy modules to build an online chatroom.

The Problem

There just aren't enough good chatroom websites out there today: if I want to have a conversation with a bunch of random strangers on the Internet, the barrier to entry is just too high. You've been commissioned to provide a solution.

Requirements

You will write a web server to host your project using Node that runs on port 8080. On the homepage ("`/`") should, at the very least, be a button to create a new chatroom. You might also consider providing some chatroom stats, or a list of recently active chatrooms, but that's entirely up to you.

Creating a new chatroom should generate a new, random, 6-character alphanumeric identifier for the chatroom (e.g., `ABC123`) - referred to in the database schema as the chatroom's name - insert the room into the `rooms` table in the database, and then take the user to the URL `/ABC123`, which is the actual chatroom interface.

Once in the chatroom, the user should be instructed to choose a nickname (if you're lazy, you can use the `prompt(...)` function for that, but try to be creative!), and then they should be presented with an interface to send messages to a shared chatroom. Any previous messages that have been sent in the chatroom should be displayed to the user, with the nicknames of the users that sent them. The list of messages should refresh periodically, ideally at *least* every 5 seconds.

You should consider implementing a RESTful API for the polling aspects of your chatroom. For example, you might want `/ABC123/messages` or `/ABC123/messages.json` (or both) to return a JSON-formatted representation of all messages in the chatroom. It might also make sense that you make a POST request to `/ABC123/messages` to add a new message. Try to come up with a few other URLs you could add to make your chatroom truly featureful.

Getting Started

Create a new directory to hold your project's files, and then run:

```
cs132_install chatroom
```

This will install a single, lowly file into your current directory, called `package.json`, which contains information on all of the modules your project will need. To install your dependencies into the `node_modules` folder local to your project, run:

```
npm install
```

If anything goes wrong, it'll very clearly yell at you with an error message. Otherwise, if you see a dependency tree printed out at the end, you can assume it worked!

```
any-db@0.3.0 node_modules/any-db
├── any-db-pool@0.0.0
└── generic-pool@2.0.3

express@3.0.6 node_modules/express
├── methods@0.0.1
├── fresh@0.1.0
├── range-parser@0.0.4
...
```

Now that all your dependencies are installed, create a new file called `server.js`, require Express from the file, setup an app, and tell it to listen on port 8080. When all is said and done, it should look something like this:

```
var express = require('express');
var app = express();

app.use(express.bodyParser()); // definitely use this feature

// your app's code here

app.listen(8080);
```

In Express, you can capture parameters from URLs like this:

```
app.get('/:roomName', function(request, response){
    var name = request.params.roomName;  // 'ABC123'
    // ...
});
```

You can handle POSTs from forms in much the same way from the `request.body` object, as long as you're using the `bodyParser` middleware:

```
app.post('/:roomName/messages', function(request, response){
    var name = request.params.roomName;   // 'ABC123'
    var nickname = request.body.nickname; // 'Justin'
    var message = request.body.message;   // 'Baby baby baby ohhhhh'
    // ...
});
```

If you want to play around a bit more with Express, the getting started guide might be useful. You can start after the first few steps, we already took care of setting up your dependencies for you.

Connecting to the Database

For this project, you'll be using any-db with PostgreSQL, both of which have already been installed on your EC2 image. To establish a connection, require any-db, use the `anyDB.createConnection(...)` method:

```
var anyDB = require('any-db');
var conn = anyDB.createConnection('postgres://local:local@localhost/chatroom');
```

You can now talk to Postgres using the standard any-db API. Using a database abstraction layer like any-db is a "Good Thing" when you don't need any features that a specific database provides, because it gives you the flexibility to migrate to a different database at some point in the future (any-db currently also supports MySQL and sqlite3).

To execute a simple query - for instance, to get a list of rooms on the server - you can use the `conn.query(...)` method:

```
var q = conn.query('SELECT * FROM rooms');
q.on('row', function(row){
    // this code is executed for each row
});
q.on('end', function(){
    // this code is executed after all rows have been returned
});
```

You may want to return a list of rooms that have messages within the past 5 minutes ("active" rooms). One example of a query to do that would be:

```
SELECT DISTINCT rooms.* FROM rooms, messages
    WHERE messages.room = rooms.id AND messages.time >= now() - INTERVAL '5 minutes';
```

The schema for the `chatroom` database is as follows:

```
CREATE TABLE rooms (
    id SERIAL PRIMARY KEY,
    name CHAR(6) UNIQUE NOT NULL
);

CREATE TABLE messages (
    id BIGSERIAL PRIMARY KEY,
    room INTEGER NOT NULL REFERENCES rooms(id),
    nickname VARCHAR(255) NOT NULL,
    body TEXT NOT NULL,
    time TIMESTAMP WITH TIME ZONE NOT NULL
);
```

You are free to make modifications to it as you see fit, but this schema should be sufficient for the requirements of the project. Note well that room names *must* be unique, and you will get a database error if you try to reuse one. Be sure to handle that error, and generate a new identifier under those circumstances.

Parameterized Queries

As you've probably learned by now, it's very dangerous to put user-provided information directly into a SQL query, as that leaves you wide open to SQL injection attacks. For that reason, whenever you want to use user input in a query, you should use *bound parameters* or (ideally) prepared statements. any-db does not support the latter, so we'll use the former.

Say, for instance, you want to get all messages attached to the room named "ABC123". Use a dollar sign, and a number, in place of the value of the field, and then pass an array of parameters to the `conn.query(...)` method:

```
var sql  = 'SELECT nickname, body, time FROM messages LEFT JOIN rooms ON messages.room = rooms.id'
;
    sql += 'WHERE rooms.name = $1 ORDER BY time ASC';

var q = conn.query(sql, ['ABC123']);
q.on('row', function(row){
    // ...
});
```

In this example, the variable `$1` corresponds to the first (and only) element in the array, which is `'ABC123'`.

Generating Room Identifiers

There are a number of strategies you could use to generate room identifiers. One straightforward one is this:

```
function generateRoomIdentifier() {
    // make a list of legal characters
    // we're intentionally excluding 0, O, I, and 1 for readability
    var chars = 'ABCDEFGHJKLMNPQRSTUVWXYZ23456789';

    var result = '';
    for (var i = 0; i < 6; i++)
        result += chars.charAt(Math.floor(Math.random() * chars.length));

    return result;
}
```

Refreshing the Messages List

There are a couple of strategies you could feasibly use to keep the messages list updating. One strategy would be to create a separate URL, like `/ABC123/messages`, that contains only the HTML for the messages in your chatroom (without any of the chrome, like the new message input box), and show that in an iframe. You can then refresh that iframe periodically without affecting the rest of your page's content. One easy way to do that is with the Refresh header:

```
app.get('/:roomName/messages', function(request, response){
    response.setHeader('Refresh', '5');  // refresh every 5 seconds
    // go on about your business
});
```

Perhaps a better way, though, would be to use Ajax. Setup an Ajax request to fire on a timer every 5 seconds or so to a URL that returns a JSON dictionary of messages. Express makes it easy to send JSON in a response, using `response.json(...)` instead of `response.send(...)`:

```
app.get('/:roomName/messages.json', function(request, response){
    // fetch all of the messages for this room
    var messages = [{nickname: 'Justin', body: 'Baby, baby, baby ohhhh'}, ...];

    // encode the messages object as JSON and send it back
    response.json(messages);
});
```

`response.json(messages)` is equivalent to `response.send(JSON.stringify(messages))`.

— Rendering HTML —

When it comes time to send HTML back to the browser, you have a few options. You can compose the HTML by hand in your JavaScript, like so:

```
var html  = '<!DOCTYPE html>\n';
    html += '<html>\n';
    html += '<head>\n';
    html += '    <title>Room: ' + roomName + '</title>\n';
    // ...
```

... but why ever would you want to use *that* monstrosity? We recommend that you use a templating engine to process your HTML. Mustache is the templating engine recommended by the TAs, and we've included a Node implementation of Mustache, mu, in your package's dependencies.

If you'd like to use mu, start out by creating a directory called "templates" (or whatever you like, really) in your project directory, and then import `mu2` into your project:

```
var mu = require('mu2');
mu.root = __dirname + '/templates';
```

Then, inside your templates directory, create (e.g.) a `room.html` file. Write out your template using Mustache (see the Mustache website and the Mustache manual):

```
<!DOCTYPE html>
<html>
<head>
    <title>Room: {{roomName}}</title>
...
```

... and then use mu to compile it, and serve it as a response:

```
app.get('/:roomName', function(request, response){
    // do any work you need to do, then
    response.status(200).type('html');

    var t = mu.compileAndRender('room.html', {roomName: request.params.roomName});
    t.pipe(response);   // pipe the data back to the browser
});
```

If you use Ajax to refresh the messages list, and only send JSON back and forth between the client and server, your HTML rendering requirements should be fairly lightweight. For instance, generally speaking, it is likely that the only information you'll need to get to your room page is the name of your room. If you find yourself with a fairly small amount of variables that you need to expose, you might consider using the `meta` tag trick.

The meta Tag Trick

In your template, define `meta` tags that contain the values of your variables:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="roomName" content="{{roomName}}">
    ...
```

Then, when you need to access the variable from (client-side) JavaScript, access it via the DOM:

```
var meta = document.querySelector('meta[name=roomName]');
var roomName = meta.content;
```

If you do this a lot, you might consider writing a simple wrapper:

```
function meta(name) {
    var tag = document.querySelector('meta[name=' + name + ']');
    if (tag != null)
        return tag.content;
    return '';
}

var roomName = meta('roomName');
```

Using Forms

You're going to have to provide some way for your users to post new messages, and the HTML mechanism for that is forms. There are plenty of tutorials online, but as a simple example, a basic message form (in a Mustache template) might look something like this:

```
<form method="POST" action="/{{roomName}}/messages" id="messageForm">
    <input type="text" name="message" id="messageField">
    <input type="hidden" name="nickname" id="nicknameField" value="">
    <input type="submit" value="Send">
</form>
```

The basic form element is the `input` element. The "text" type renders a text field the user can type into; the "submit" type renders a button that submits the form; and "hidden" fields do not display on the page, but allow you to send additional content back with the form data. We've included a nickname field here, which you should populate with information after your user picks his or her nickname:

```
document.getElementById('nicknameField').value = 'Justin';
```

When you click submit, the browser will send a `POST` request to `/ABC123/messages`, and then take the user there. Since you probably don't want the user to stay at `/ABC123/messages`, you'll probably want to add some server-side code to redirect them back to `/ABC123`:

```
app.post('/:roomName/messages', function(request, response){
    // post everything to the database, then...
    response.redirect('/' + request.params.roomName);
});
```

That will all work just fine and dandy, but it isn't perfect! You can eliminate the refresh using Ajax.

Submitting a Form with Ajax

> ***Note:*** *the following only works on recent versions of modern browsers (Chrome 7+, Firefox 4+, Safari 5+, IE 10+). The TAs will only test on modern browsers, so it'll work for us, but make sure you're using a browser that supports this.*

When the page finishes loading, attach a new submit handler to the message form:

```
window.addEventListener('load', function(){
    var messageForm = document.getElementById('messageForm');
    messageForm.addEventListener('submit', sendMessage, false);
}, false);
```

Inside the handler, create a new `FormData` object (documented on the Mozilla Developer Network), and send that as the body of your `XMLHttpRequest`. Make sure to also call `event.preventDefault()` to prevent the form from *actually* submitting.

```
function sendMessage(e) {
    // prevent the page from redirecting
    e.preventDefault();

    // create a FormData object from our form
    var fd = new FormData(document.getElementById('messageForm'));

    // send it to the server
    var req = new XMLHttpRequest();
    req.open('POST', '/' + meta('roomName') + '/messages', true);
    req.send(fd);
}
```

You can also build arbitrary `FormData` objects yourself:

```
// arbitrary form data
var fd1 = new FormData();
fd1.append('nickname', 'Justin');
fd1.append('message', 'Baby baby baby ohhhhhh');

// equivalent to `new FormData(formEl)`
var fd2 = new FormData();
fd2.append('nickname', document.getElementById('nicknameField').value);
fd2.append('message', document.getElementById('messageField').value);
```

If you want to show progress to the user as their message is posting, you can use the standard mechanism for attaching an event listener to your `XMLHttpRequest`.

Using Additional Modules

You may find that you want to use other Node modules to complete this project. That's fine by us! We only ask that you **make sure** to add any new dependencies you introduce to your `package.json` file. The dependency versioning format is well-documented at:

```
npm help json
```

When in doubt, just run:

```
npm info themoduleIadded version
```

... and use the version you get back as the version in your `dependencies` object.

Other Niceties

Any value you can add to your user experience is always good for brownie points. Think about what happens when someone sends a new message. Does it appear (for that user) in the message list immediately, or does it just seem to "disappear" until the next message list refresh? Can you think of a way to make things more responsive? If you can't speed things up, you can get a lot of mileage out of some kind of loading or progress indicator.

— Handing In —

Before handing in your project, make sure you've checked to make sure you've listed any additional dependencies in your `package.json` file, and you've filled in your `README.md` file (you can use any format you like for your README, but Markdown is highly recommended).

To hand in your project, from your project directory, run:

```
cs132_handin chatroom
```

That's it!