COSC 465: Computer Networking
Spring 2014
Project 8: Network performance testing web application
Due: 1 May 2014, 11:59:59pm
Note: You're welcome to work with someone else on this project

Overview

For the last project, we'll build a web-based network performance testing application similar to tools like speedtest.net and FCC Speed Test, but much simpler. The web application we build will be based on the express.js framework, and we will deploy it to the "cloud" using a platform called Heroku.

Specifically, the goals for this project are to create a web application that allows a user to run a simple round-trip time delay measurement test, and to add one other feature of your choice. The detailed steps below will guide you through most of the steps required for adding the RTT measurement capability.

Note: the instructions below assume that we've gone through the anatomy of an express.js app in class (which we will do early in the second-to-last week of classes).

Step 0: set up app locally and on Heroku.

First, fork and clone the git repository at https://github.com/jsommers/cosc465_project8. You *can* do this in an environment other than the Linux VM, but all the instructions below assume that you're using the VM. If you choose to use some other OS, you will need to ensure that you're using the right versions of node.js (>= v0.10.15) and install other tools required for this project. I'd still recommend using the VM, but you don't necessarily have to.

When you clone the repo to your VM, you should NOT clone to a folder that is shared with the host OS (i.e., a folder that is visible between the Linux VM and the main OS on your computer). As far as I can tell, there are some bad interactions with symbolic links that are created when installing various node.js packages.

Before getting started, you'll need to install the "node package manager" in your Linux VM as well as some tools for using Heroku. Just run setup.sh. You'll need to type your password (cosc465, if you haven't changed it) to get things installed:

```
$ ./setup.sh
```

The setup script installs a command line tool called `heroku`, which we'll use below when getting our app deployed to the cloud.

Next, install all the node.js package dependencies required for the project. Just type

```
$ npm install
```

at the shell.

Once that's done, start up the app on your local machine:

```
$ nodejs app.js
```

You can also just type `./rundev.sh` which will do the same thing.

Open a browser and type localhost:3000 in a browser to see the not-so-fancy (yet) site.

Now, we're going to first set up your application to run out in the "cloud" on a platform called Heroku. To do that:

1. Create an account on Heroku. It is free to do so, and you don't need to give any credit card information. Just go to https://www.heroku.com, click on "Sign Up", and follow the prompts.
2. Next, use the newly-installed heroku command line tool to login to Heroku
   ```
   $ heroku login
   ```
   Note that logging into Heroku may cause the heroku command-line tool to create some authentication keys (see https://devcenter.heroku.com/articles/quickstart).
3. Finally, we'll create a new application context on Heroku. Just type "heroku create", and you should see something similar to the following:
   ```
   $ heroku create
   Creating sheltered-stream-5688... done, stack is cedar
   http://sheltered-stream-5688.herokuapp.com/ | git@heroku.com:sheltered-stream-5688.git
   Git remote heroku added
   ```

   Notice that the heroku command line tool told us the URL for our application. That's where your app will live on the web.
5. Now, push your code to Heroku and go see it running live:
   ```
   $ git push -u heroku master
   ```

   You'll see lots of output as your application code is uploaded to Heroku (using git!) and dependencies get installed.

   Lastly, you'll need to set one configuration parameter using the following command:
   ```
   $ heroku config:add BASE_URL=http://YOURAPPNAME.herokuapp.com
   ```

   You'll need to replace YOURAPPNAME with whatever application name you were assigned by Heroku. For the output shown above where the app is created, the configuration command should be:
   ```
   $ heroku config:add BASE_URL=http://sheltered-stream-5688.herokuapp.com
   ```

   If you've forgotten the name of your app, or didn't write it down, you can type "heroku list" to get a list of your apps.

   You should now be able to type "heroku open" to open a web browser at the URL that was assigned to your application. It may take a moment for your application to fully start and be visible within the browser. The app should look pretty much the same as it

looked when you ran it locally.

Step 1: add a non-persistent store of website data.

Our first step in enhancing this sad little application will be to add some simple non-persistent storage to keep track of who visits the site. Eventually, we will keep track of performance measurements for each user that visits the site.

There is already code in the app to assign "session IDs" to each new visitor to the site, using cookies. There's nothing special that you need to do to enable this support --- new IDs will be assigned whenever a browser first visits our site. What we're going to do is create a JavaScript module that contains a hash of session IDs mapped to a hash of user-specific data. There won't be much we store for each user yet, but we'll change that as we add more functionality.

Initially, we will not have any "names" to go along with our users, so we will just make up names. For each new user, we'll assign them a name like "anonymous1", "anonymous2", etc.

Open up a new file named `userdb.js` in a text editor (this file should be saved in the top-level app folder), and add the following contents:

```
var userhash = { };  # session ID -> user data
var next_anonymous = 1;

var add_user = function(id, user) {
    if (userhash[id] === undefined) {
        if (!user) {
            user = "anonymous" + next_anonymous;
            next_anonymous += 1;
        }
        userhash[id] = {
            'id': id,
            'user': user,
            'latency_results': []
        };
    }
    return userhash[id];
};
exports.add_user = add_user;

exports.get_user_name = function(id) {
    if (userhash[id] === undefined) {
        add_user(id, undefined);
    }
```

```
        return userhash[id].user;
    };
```

Read the code. At the top, we've got a hash (userhash) that will contain a mapping of session IDs to objects containing user data, and a variable holding an integer that will be incremented for each "anonymous" user we encounter. There are two functions:

- The add_user function takes a session ID and a user name (which may be the special JavaScript value undefined). If the session ID doesn't exist in the userhash, it creates a user name (e.g., "anonymous1") and adds a new entry to the userhash, where the mapped value is an object containing the user's ID, their name, and a list which will eventually contain some performance results from running latency (round-trip time delay) tests.
- The get_user_name function takes a session ID as a parameter, possibly adds a new user to the userhash, and returns the username mapped to the ID (which may have just been generated as a result of calling add_user).

The code above is written as node.js *module*. There are two lines indicating functions that are *exported* by the module (just add_user and get_user_name). The other variables are *private* to the module and cannot be accessed by users of the module. To read more about modules in node.js, see http://nodejs.org/api/modules.html.

Let's now import this module in the "index" view so that we can display the user's name on each page that they visit. At the top of `views.js` file, add the line:

```
    var userdb = require ('./userdb.js');
```

Change the two lines inside the index function to be the following:

```
    var username = userdb.get_user_name(req.session.id);
    res.render('index', {title: "Speed tests", message: "Hello, world", user:
username});
```

What we're doing now is getting the session ID from the browser request (req.session.id), passing that into the function we just wrote to generate/retrieve a user name for each session, and passing that into the index view to be rendered.

Edit `views/layout.jade` and change the last few lines (starting with the body line) to be:

```
  body
    #header
        p Hello, #{user}

    block content
```

Be careful with those edits --- indentation matters to Jade so it can figure out when HTML tags should be closed. For reference, Jade documentation is found here: http://jade-lang.com/reference/.

When you've made these edits, you can start the app locally to test. You can also push to heroku to test there.

Local test:

```
$ nodejs app.js
```

Heroku test:

```
$ git push -u heroku master
$ heroku open
```

You should see something like the following:

Hello, anonymous1

COSC 465 network speed test (project 8)

I have no results, but I have a message: Hello, world

To test out whether new "anonymous" names get generated, you can open up another browser and go to localhost:3000 (or the heroku URL), or clear cookies from your browser. Note that the user "database" we created will completely go away when the app gets stopped --- that is the intended behavior (we are not going to be adding persistent storage to the app at all).

Step 2: add latency ("ping") test functions.

We're now going to write some more code to add a simple latency (round-trip time) test to our app using socket.io.

We'll create a new page which will serve as a launch point for performance tests. To do that we'll have to create a new route, new controller function, and new view template:

- In app.js, add a line just below the line that reads "`app.get('/', views.index);`" to add a route to a page for launching tests:
  `app.get('/tests', views.tests);`
  Note: you can call the page whatever you want to. If you don't like "tests", name it something else.
- Modify `views.js` to add a controller function for handling access to the route /tests. It should be nearly identical to the index function modulo some name changes. Note that the template you render should be called 'tests', not 'index' (i.e., the first parameter to `res.render` should be 'tests').
- Edit the file `views/tests.jade` and paste in the following content:

```
extends layout

block content
    div#testcontrol
        h3 Run tests
        p
            a(href="/") Home

        a#startping(href="#") Latency test

    div#status
```

The tests template has two divs (each with a different id) to contain some links and for holding some test status. We will soon modify some JavaScript to attach an event handler to the "Latency test" link to start a latency test. Before we do that, modify the index template (views/index.jade) to have a link to the tests page. The template syntax should be something like:

```
a(href='/tests') Launch tests
```

See the Jade reference for help.

If you start the app now, you should be able to easily click back and forth between the index and tests pages (the link to start the latency test won't do anything yet).

To add the JavaScript magic to actually execute the latency test, you need to modify the server-side "ping" handler, add a client-side function, and attach a handler that invokes the client-side function when the "Latency test" link is clicked.

- Edit public/javascripts/appclient.js and add the following function after the top line, but before the return statement that returns a hash of function names:

  ```
  var start_ping = function() {
      var socket = io.connect();
      socket.emit('ping', {timestamp: Date.now()});
      socket.on('pong', function(data) {
          var rtt = Date.now() - data.timestamp;
          console.log("Ping RTT (milliseconds): " + rtt);
      });
  };
  ```

  This is basically the same code that we saw (or will see) in class.
- Also in public/javascripts/appclient.js, edit the init function (just after the console.log statement) to include the lines:

  ```
  jQuery("#startping").click(start_ping);
  ```

  This will install a click handler on the "Latency test" link on our tests page so that the `start_ping` function gets invoked.

- Lastly, add a line to the server-side ping function in `perftests.js`:
  `req.io.emit('pong', req.data);`

  Note that `app.js` already has a line to call a function in `perftests.js` when the socket.io "ping" message is received.

  This line will send back a "pong" message to the client, whenever the server gets a ping. The server will send back all the same data that the client sent to us (e.g., the data that includes the timestamp that was originally included in the ping).

If you start up the app now and click on the "Latency test" link, you should see a log message at the server. If you open up the JavaScript console on your browser, you should also see a log message at the client indicating the measured round-trip time in milliseconds.

You'll need to modify the client and/or server side JavaScript code to do the following:

- For each latency test (each time someone clicks the "Latency test" link), you should send 5 ping/pongs back and forth, and compute the average RTT. Suggestion: one easy way to keep track of how many ping/pongs have been sent is to add additional information to the "data" hash sent back and forth, e.g., a sequence number.
- Send the test results from the client/browser to the server so that they can be stored in the userdb module that we created earlier. It is probably easiest to send the test results via a socket.io call. You could simply add a special server route to accept test results, or modify the ping/pong protocol so that the server records the right result.
- Show the latency test result to the user (browser) at the end of a test. The results should not only be stored up at the server, but they should also be shown to the client. If you want to get fancy, you could even update the screen after each ping/pong sample is received at the client, to give the user some feedback about how the test is proceeding.

If you want to verify that your ping measurements are "good", you can start up Mininet and test the app. There is a start_mininet.py script in the repo. Once you start mininet, open an xterm on h1, and another xterm on h2:

```
$ sudo python start_mininet.py
mininet> xterm h1
mininet> xterm h2
```

Note that h1 has IP address 10.0.0.1 and h2 has IP address 10.0.0.2. The RTT between these hosts should be 50 milliseconds.

On h1, start up your app server:

```
h1# nodejs app.js
```

On h2, start up a browser and make a request to the app server:

```
h2# google-chrome http://10.0.0.1:3000/
```

Step 3: make things look less bad.

First, add a link to the venerable bootstrap CSS library to make things look nicer. Add the following line to views/layout.jade, just *before* the line to include the CSS file css/app.css:

```
link(href="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css", rel="stylesheet")
```

This should *immediately* make your pages look better. You're welcome to tinker with the visual characteristics of the app further by adding specific bootstrap classes to page elements, or by modifying the file `public/css/app.css` (which is currently empty). The bootstrap documentation is quite good.

Step 4: add one other feature.

To finish off this project, you'll need to add one more feature of your choice. Here are a small number of possibilities for features to add:

- Add another page to show ping results for different users, with some capability to sort results or display them in some nice way (maybe even with a graph using HTML5 canvas, or some external library that makes nice graphs).
- Add a throughput test capability, like upload or download throughput testing. This isn't much harder than what you did for the latency (ping) test. If you want to do this, you'll probably need to create a large chunk of data to upload or download. An easy way to do that is to create a JavaScript "typed array". See http://www.html5rocks.com/en/tutorials/webgl/typed_arrays/ for examples of how to do that. You could make a 100KB (or 1MB, if you're feeling lucky) array and download or upload it a few times, while measuring how long it takes to send the data.
- Add authentication by piggybacking off of Google's services. I'd strongly recommend using "passport" to do this. See http://passportjs.org/guide/google/ for some examples and details. If you add this feature, you should also modify the userdb code (and possibly other code) to show the authenticated user's name and/or email address instead of "anonymousX" on various pages.
- If you want to do something else, talk to me first --- the scope needs to be reasonable (not too small, but also not too big).

Step 5: be done with it.

When you're done, push your app to Heroku and submit your application code and Heroku URL on Moodle. To submit the application code, you can either submit a github repo URL, or zip up the project in a (big) zip file. Either way is ok.