University of Toronto

Department of Computer Science

CSC309 Programming on the Web

Web Application on Sharing Economy

Project Name	*Umbrella*
Group Information	Nisal Perera: c4perera Sayf Boujan: c5boujan Priyank Purohit: c2purohi Kelvin Luo: c3luozhi
Instructor	**Professor:** Mashiyat Ahmed Shah **TA:** Mohammad Hossein Danesh danesh@cs.toronto.edu; t2danesh
Due Date	December 6, 2015

Description

Umbrella is a first of its kind web app that allows research labs at universities to effectively manage and organize lab equipment sharing amongst the lab members. There are two kinds of users in this system, lab admins and lab members. Lab admin created the lab, added equipment to the lab, and approved current lab members. Regular users can view and join labs, rate and comment on labs and equipment, message other users, and book lab equipment in the labs that they are member of.

Users can join the system and search for labs directly, or get recommendations for the labs that may be relevant to them based on their field of study and specialization. The shared resources in our system are the labs and the equipment in those labs. Users can book equipment based on available time slots decided by the lab admin. Users can view which time slots are available and they can book them. Our app allows lab members to effectively plan their lab experiments and easily communicate with other lab members and lab administrative staff.

The web application is deployed on Heroku (http://umbrella-resource-management.herokuapp.com/), and is built using NodeJS, EJS, HTML5, CSS3, Bootstrap, JavaScript, jQuery, and AJAX.

How it Works/Features

Profiling: When users register for our application, they create a personal profile. Here they provide their department of study and the area of specialization. This information is later used for recommending labs to the user. Users can update their personal information, and can see the profile of other users.

User Authentication and Authorization: System authenticates users based on correct email and password combination. We require every user to have a unique email address. There is also an option for users to sign up using their Google account.

Social Network: Members of the same labs form a social network. Users can interact with one another by sending private messages. Users can review and rate the labs that they are in, and other users can make decision based on those reviews and interactions.

Rating and Commenting: Lab equipment is added to the lab by the lab admin. Users are then able to comment and rate the equipment. Using this commenting and rating features, other users can decide whether to use that resource or not.

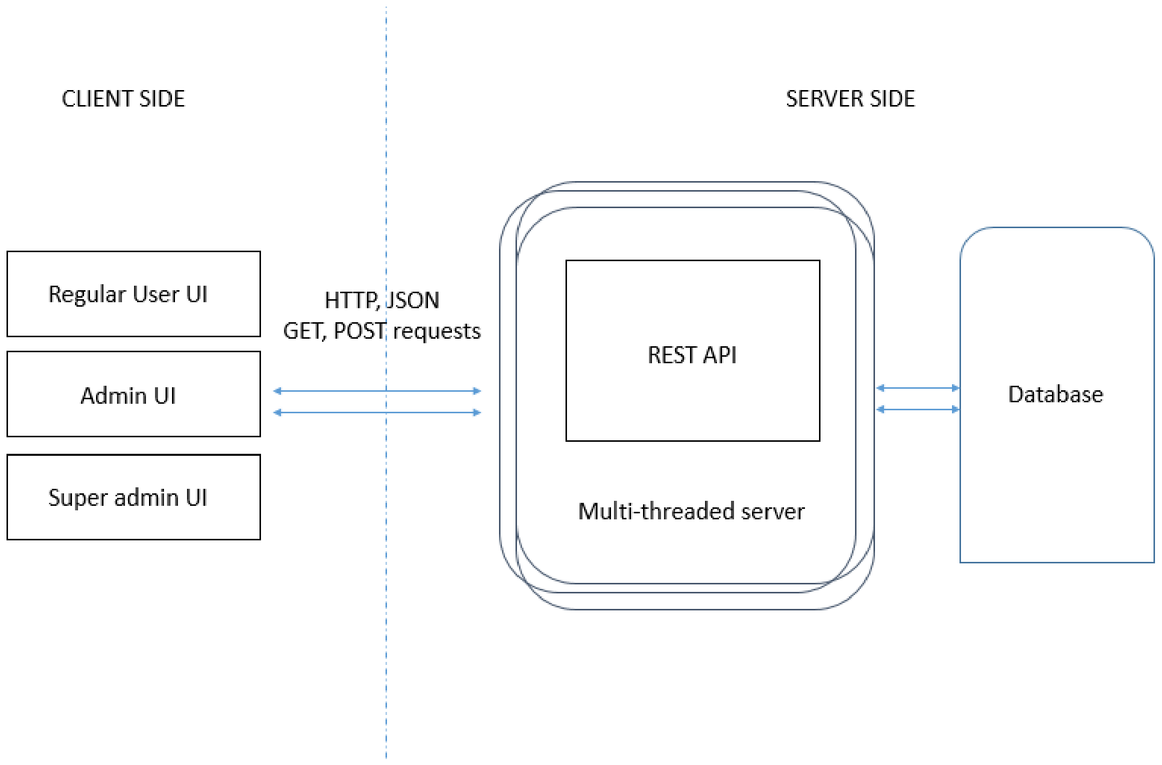
Search and Recommendation System: Our system provides the basic functionalities to search for labs. Using this feature, users are able to search and discover what they want to use. There is also a smart recommendation system that ranks the matching labs according to user's personal data such as department of study and area of specialization.

Admin: Admin users have basic admin functionalities, including adding new labs to the system, adding new equipment to the labs. Admin can approve new members to the lab and remove existing members from the lab.

Super Admin: Super Admin can edit all member profiles and view and edit all lab pages. The super admin also has to approve a new lab before it becomes available in the system for users.

Detailed Design of the Application

A high-level view of your software



Section Descriptions

Client side interacts with the server by making GET/POST requests to the REST APIs. The REST APIs handle the request and respond with relevant information (JSON), or respond with rendered EJS views.

The main REST API calls are outlined below. MongoDB is used to maintain persistent information about the web app users, and the labs created by the users. Server interacts with the mongoDB database using mongoose which simplifies CRUD interactions between server and database.

The mongoDB stores two schemas: users and labs.

User Schema**:** The user schema holds information about all the users who sign up for Umbrella. There are three distinct user types: super admins, regular users, and admins.

- Super Admin**:** Super admin privileges are assigned to the first user to sign up for Umbrella. This user can perform any action in the app without restrictions. Umbrella is deployed with one user already signed up to allow us to maintain the system.
- Regular User**:** All subsequent users to sign up for Umbrella after the super admin are regular users. They can only edit their own user information, and book equipment in labs of which they are a member of.
- Admin**:** When a regular user creates a lab and gets the lab approved by the super admin, the user becomes the lab's admin and has full control over it. The admin can approve new lab users, or remove existing users.

Other user information such as username, email, password, join date, display picture, messages, and more is also stored in this user schema.

Lab Schema**:** A regular user can create new lab by providing the required information about the laboratory. This lab will not show up in the system until it is approved by the super admin to prevent fake laboratories from being created in the system (expensive to maintain database).

Detailed fields of both schemas can be seen in the `/app/models/` folder.

List of pages and UI elements of the Project

This section includes the major pages and UI elements of Umbrella.

- Main/Home Page(/)
  User is given three options on this page
  - Local Login (/login)
    - User is asked for email and password that was chosen upon signup
    - Log In button to initiate the login request

        - Links to other pages (homepage or signup page)
    - Local Signup (/signup)
        - User is asked to fill out all required fields for signup
        - Sign up button to initiate the signup request
        - Links to other pages (homepage or signup page)
    - Sign In with Google
        - Our app is registered with Google to sign in through Gmail account
        - We require basic permission from user to use their name and email from Google
        - Links to other pages (homepage or signup page)

- Welcome Page (/profile)
  Once user logs into the system, they get directed to this page.
    - Search Bar: User can type in a lab and if it exists, they will be directed to the main page of the lab
    - Tabs: options to create lab, list labs, list users, and view messages
    - Option to logout, or edit the profile
    - User info (including the profile picture)
    - List of Labs the user is a member of and a list of recommended labs that the user might be interested in

- Create New Laboratory page (/createLab)
  User can create a new laboratory
    - Fill in all the required fields
    - Submit the request (Add Lab)
    - Redirect to Homepage

- List of all Labs in the system page (/listLabs)
  User can view all the labs in the system from this page
    - User is shown a brief info about each lab
    - User can Submit a request to the lab admin to join the lab
    - If he is already a member of the lab, user can click to view the lab

- A Particular Lab's Page (/listThisLab?specificLab=<lab name>)
  User is now a member of this lab and can perform these actions:
    - Add new equipment to the lab (Admin only)
    - Book an equipment (if there is any)

    - Post a comment about the lab
    - Submit a rating about the lab
    - View all members of the lab
    - Approve or remove lab members (Admin only)

- “Add New Equipment to a lab” page(/addEquipment) (Lab admins only)
  Lab admin can add equipments to a lab
    - Admin is required to enter all fields to provide info about the equipment
    - Submit the action to redirect to a different page
    - Admin specify the number of booking time slots here

- “Book equipment” page (/getBookingTable/<index of equipment>)
  Users can book an equipment to use in the lab or sign out at the chosen time
    - User is given a table where he/she gets to choose which day and which slot in the day to book an equipment

- “List all Users” page (/listUsers)
  A user can view profiles of all other users in the system
    - Edit profile: user can choose new picture, update personal information or change password
    - Choose a particular user and send a message to that user
    - Super Admin can remove users from the system (super admin only)

- View messages page (/listMessages)
  Users can view messages if they have any in their inbox
    - User can view the message and either delete it or reply to the person

Security

We improved the security of our application using following methods.

1. Secure third party authentication using Passport-Oauth module
   OAuth provides client applications a 'secure delegated access' to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without sharing their credentials. Designed specifically to work with Hypertext Transfer Protocol (HTTP), OAuth essentially allows access tokens to be issued to

third-party clients by an authorization server, with the approval of the resource owner. The client then uses the access token to access the protected resources hosted by the resource server.

2. Data encryption in Database using bcrypt.js
   Sensitive user data such as user passwords are encrypted at the server when stored in the mongoDB database. Because of this even if someone gets access to the database the data will be useless without the decryption key.

   We can test this by using our app to register a new user and then use a database management client such as robomongo to view the content of database. You will observe that password field is encrypted.

3. Helmet.js is used to prevent following security vulnerabilities of the site
   Helmet helps you secure your Express apps by setting various HTTP headers. This can be test by printing out the http request object and check if those fields are set properly. See Appendix B below for complete comparison. Below is a list of some of the attacks mitigated by Helmet.

   XSS Filter

   This occurs when an attacker injects executable code to an HTTP response. When an application is vulnerable to this type of attack it will send back invalidated input to the client (mostly written in JavaScript). It enables the attacker to steal cookies, perform clipboard theft and modify the page itself.

   The XSS Filter was created to help protect against XSS attack. The **`X-XSS-Protection`** HTTP header is a basic protection against XSS. It was originally by Microsoft but Chrome has since adopted it as well.

   Frame options

   Trying to prevent your page being put in a <frame> or <iframe> without your consent. This can result in click jacking attacks, among other things.

   Helmet to mitigate this by using the **`X-Frame`** HTTP header restricts who can put your site in a frame. It has three modes: DENY, SAMEORIGIN, and ALLOW-FROM. If your app does not need to be framed, you can use the default DENY. If your site can be in frames from the same origin, you can set it to SAMEORIGIN. If you want to allow it from a specific URL, you can allow that with ALLOW-FROM and a URL.

HTTP Strict Transport Security

Trying to force users to view the site on HTTP instead of HTTPS because HTTP is insecure.

Helmet to mitigate this by using a middleware that adds the **`Strict-Transport-Security`** header to the response. This tells browsers to only use HTTPS for the next period of time.

Hide X-Powered-By

Trying to prevent hackers who can exploit known vulnerabilities in Express/Node if they see that your site is powered by Express (or whichever framework you use). **`X-Powered-By`**: Express is sent in every HTTP request coming from Express, by default.

Helmet to mitigate this by using the hidePoweredBy middleware to remove the X-Powered-By header if it is set (which it will be by default in Express).

Automated Testing

For automated testing we used Mocha, a JavaScript test framework running on Node.js and the browser to do asynchronous testing. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

All the test cases are included in our git repo under `/test` folder. To test the application run:

```
mocha “test/test-server.js”
```

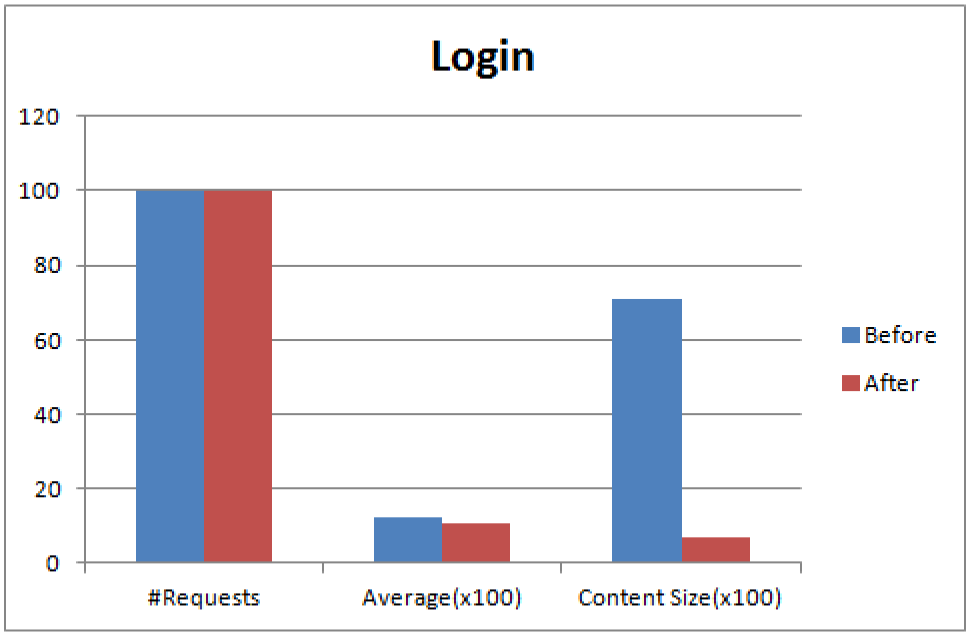
Furthermore, we used locust for load testing, which also reports any failures in the web app. The failure rate for 100 users attacking the main routes is 0%.

Performance

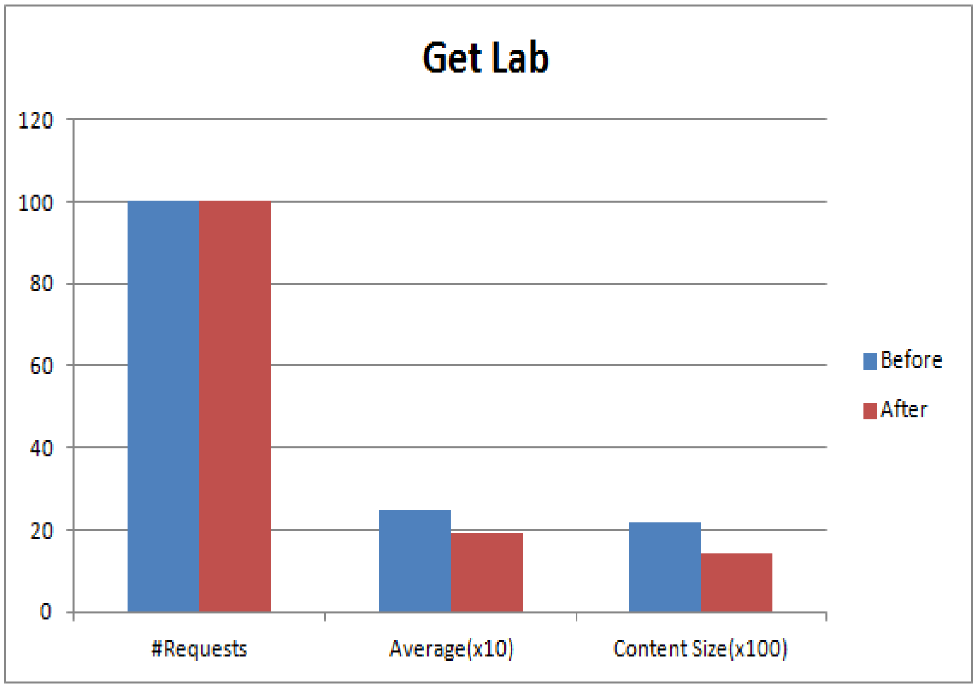
Optimization 1: Gzip for express.js

Tested routes for before using Gzip vs. after using Gzip.

- Login:



- Get Lab:



Both results show a reduction in the amount of time it takes to serve information back to client, and in the size of response.

Optimization 2: Reduce number of database queries in the code

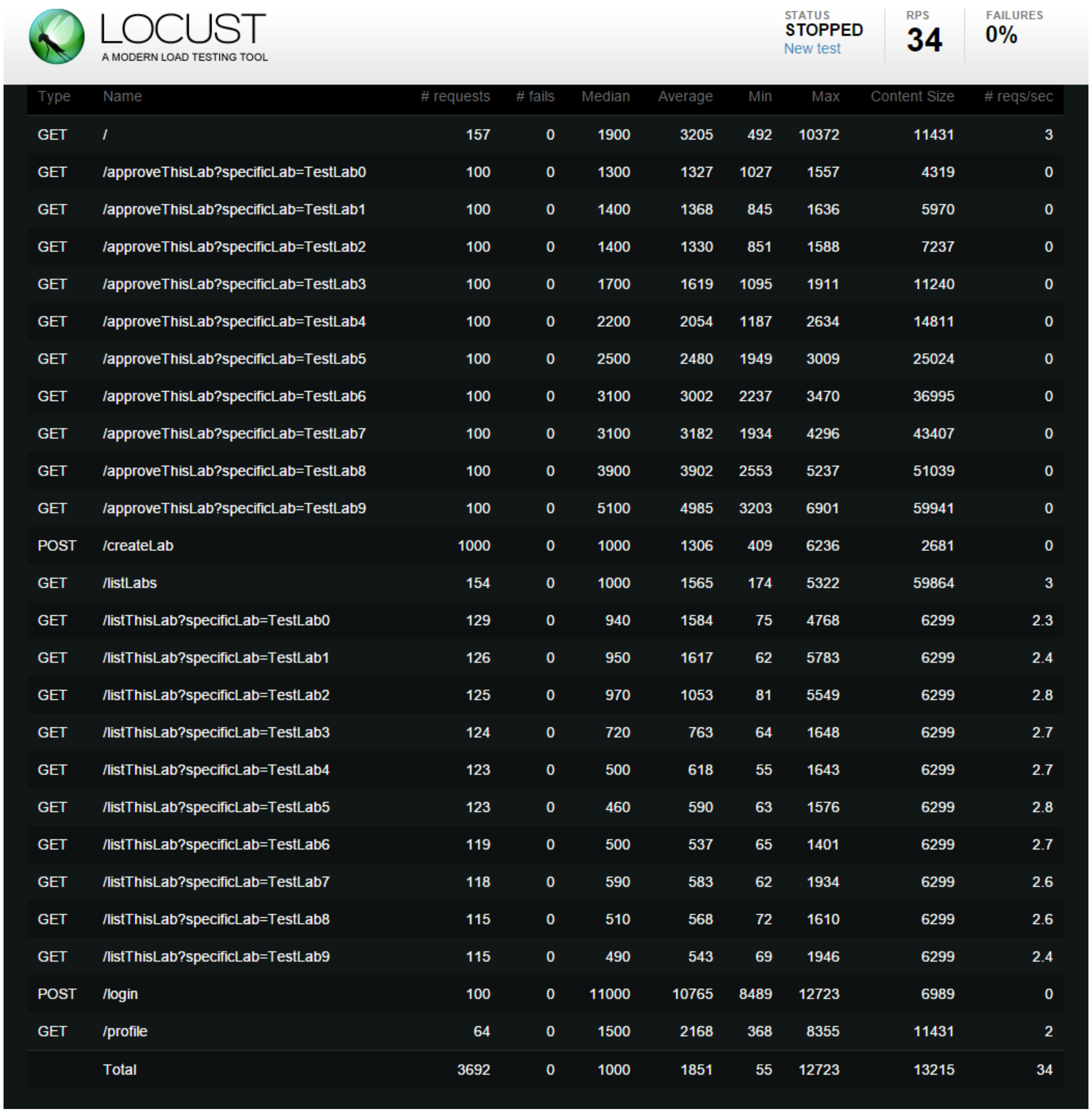
a. We retrieve all system users as a single JSON object rather than making one request per user.
b. We compress the images prior to sending it to the client.

We used Locust load testing tool to test how many concurrent clients our web app can handle and also how to improve the performance of the main/busy routes.

After installing locust, we wrote a script to connect to the web app and allow a swarm of locusts to attack our server at different routes.

The locust python script file can be found on the main directory of our app (`locustfile.py`).

Locust load testing results:



Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
GET	/	157	0	1900	3205	492	10372	11431	3
GET	/approveThisLab?specificLab=TestLab0	100	0	1300	1327	1027	1557	4319	0
GET	/approveThisLab?specificLab=TestLab1	100	0	1400	1368	845	1636	5970	0
GET	/approveThisLab?specificLab=TestLab2	100	0	1400	1330	851	1588	7237	0
GET	/approveThisLab?specificLab=TestLab3	100	0	1700	1619	1095	1911	11240	0
GET	/approveThisLab?specificLab=TestLab4	100	0	2200	2054	1187	2634	14811	0
GET	/approveThisLab?specificLab=TestLab5	100	0	2500	2480	1949	3009	25024	0
GET	/approveThisLab?specificLab=TestLab6	100	0	3100	3002	2237	3470	36995	0
GET	/approveThisLab?specificLab=TestLab7	100	0	3100	3182	1934	4296	43407	0
GET	/approveThisLab?specificLab=TestLab8	100	0	3900	3902	2553	5237	51039	0
GET	/approveThisLab?specificLab=TestLab9	100	0	5100	4985	3203	6901	59941	0
POST	/createLab	1000	0	1000	1306	409	6236	2681	0
GET	/listLabs	154	0	1000	1565	174	5322	59864	3
GET	/listThisLab?specificLab=TestLab0	129	0	940	1584	75	4768	6299	2.3
GET	/listThisLab?specificLab=TestLab1	126	0	950	1617	62	5783	6299	2.4
GET	/listThisLab?specificLab=TestLab2	125	0	970	1053	81	5549	6299	2.8
GET	/listThisLab?specificLab=TestLab3	124	0	720	763	64	1648	6299	2.7
GET	/listThisLab?specificLab=TestLab4	123	0	500	618	55	1643	6299	2.7
GET	/listThisLab?specificLab=TestLab5	123	0	460	590	63	1576	6299	2.8
GET	/listThisLab?specificLab=TestLab6	119	0	500	537	65	1401	6299	2.7
GET	/listThisLab?specificLab=TestLab7	118	0	590	583	62	1934	6299	2.6
GET	/listThisLab?specificLab=TestLab8	115	0	510	568	72	1610	6299	2.6
GET	/listThisLab?specificLab=TestLab9	115	0	490	543	69	1946	6299	2.4
POST	/login	100	0	11000	10765	8489	12723	6989	0
GET	/profile	64	0	1500	2168	368	8355	11431	2
	Total	3692	0	1000	1851	55	12723	13215	34

Optimization 3: Multithreading the node.js server

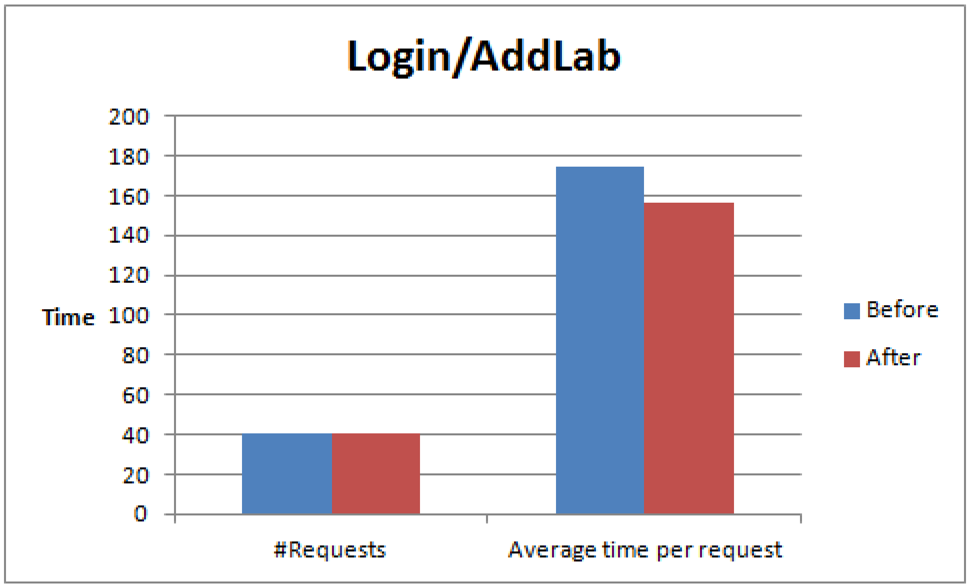
A single instance of Node.js runs in a single thread. To take advantage of multi-core systems we need to launch a cluster of Node.js processes to handle the load. The pm2 module allows you to easily create child processes that all share server ports.

PM2 is a production process manager for Node.js applications with a built-in load balancer. It allows you to keep applications alive forever, to reload them without downtime and to facilitate common system admin tasks.

To run the app using PM2 execute the following three commands:

`npm install pm2 -g`	(to install the pm2 module)
`pm2 start server.js -i -1`	(to start the maximum processes)
`pm2 stop  server.js`	(to shut down the pm2 server cluster)

Here is the Locust data results before multithreading and after multithreading.



Video Demo

https://youtu.be/yeZkzJtGO7k

Hosting in a server

Note : This is hosted on a free server and it is pretty slow

http://umbrella-resource-management.herokuapp.com/

Appendix A - Helment.js setting various HTTP headers

Added following headers

```
_headers:

      { 'x-content-type-options': 'nosniff',

        'x-frame-options': 'SAMEORIGIN',

        'x-download-options': 'noopen',

        'x-xss-protection': '1; mode=block' },

     _headerNames:

      { 'x-content-type-options': 'X-Content-Type-Options',

        'x-frame-options': 'X-Frame-Options',

        'x-download-options': 'X-Download-Options',

        'x-xss-protection': 'X-XSS-Protection' },
```

Removed following header fields

```
_headers: { 'x-powered-by': 'Express' },
```