MyTorontoLife

Members:
Adam Benn | g5bennad
Chris Dobson | g4chris
Ekaterina Datsenko | c2datsen
Yuanjun Huang | c1huanha

Hosting Server Site URL:

https://polar-reef-5864.herokuapp.com/

Video:
https://youtu.be/ybJwXjNG_1U

GitHub URL:
https://github.com/ceedob/csc309-a5/

Project Description:
MyTorontoLife is a Community portal for Toronto residents where content is driven by users and group interests. This is a one-place online resource for any vital information related to community life: such as featured events, ventures, announcements, searches and other trivia related to community life. The portal allows users to promote news, activities and events in their neighborhoods, find and connect with people who share their interests and/or are situated in their neighborhood, create new interest groups or organize a poll. The site allow users to receive their daily information feed configured based on their interests, groups and information rated highly by users who share same interests. The portal helps users to learn more about their neighborhoods and be in touch with everyday neighborhood life.

How it works:
Profile: Users create their profile, set up their interests and add themselves to existing groups (neighborhoods).
Post: Users can initiate a post where he/she identifies its type and links it to an existing interest and group. Users can add multiple hashtags to identify a keyword or topic of interest in order to facilitate a search for it in the feed.
Comments and Ratings: Other users can comment on posts and rate them.
Search and Recommendation System :
User default newsfeed is build based on the following recommendation algorithm:

Display top 100 posts highly rated (a rating of 4 or 5) by users who have interests and groups that intersect with this user, created less than a year ago, and have not been seen by the user yet (e.g. not rated by this user). In case of empty result, display feed from the user's groups.

On the main dashboard a user has the option to change newsfeed content by:

- switching between groups / interests/ hashtags.
- by clicking on links for tags /interest/group used in posts
- or adding themselves to the group if they find a group's related posts interesting.

Administration: Users with Admin status can manage users profiles, view/add interests, view/add/delete groups, view/delete any post, promote other users to admin, or revoke admin status (except for Super Administrator).

Software Architecture:

High-level Software Design Overview:

Our project is designed as a single page application using AngularJS which implements client side MVC. We use angular to do all of the routing for our application, which renders different templates in our main view that resides in the index.html page. It responds based on the URL the user is visiting, or location path that is changed based on actions (like navigation). We use nested controllers: MainController is the highest $scope level, and controls the main navigation of the app. All other controllers inherit from it, and are used to control their respective functionality (LoginController, etc). SharedService is used to facilitate communication between controllers: especially allowing the children controllers to talk to their parent to trigger a refresh of the views. The backend contains all of our REST api calls that communicate between the Mongo Database and the Client, and return a JSON object in response.

We used a client-side MVC because it allows a much more dynamic and rich experience for the user and a big boost in performance, because a lot of computing is distributed to the user’s machine, instead of the central server doing all of the heavy duty tasks (rendering, redrawing, routing, monitoring changes).

Backend REST API:
The backend contains many api calls that send JSON objects in response. The backend doesn’t do any routing, it serves as middle layer between the frontend and the database.

Authentication

GET /auth/github: Get github authentication
GET /auth/github/callback: The callback api used by github after authentication
POST /auth/local/signup: Register a new user
POST /auth/local/login: Login the user
GET /auth/logout: Logout
GET /auth/loggedInUser: Check if a user is currently logged in

Dashboard

GET /dashboard: Get posts based on smart recommendation for main feed for the user

Groups

GET /groups/group Get group information
GET /groups List of all groups
POST /groups/addnew Create a group
POST /groups/group/addmember Add the user to the group
PUT /groups/group/removemember Remove the user from the group
DELETE /groups/group/:id Delete the group
GET /groups/group/posts Get all posts posted in this group

Interests

GET /interests List of all interests
POST /addnew Create a new interest
GET /interests/interest/posts Get all posts with this interest tag

Posts

GET /posts Retrieve all posts from the Database
POST /posts/addnew Create a new post
GET /posts/post Get a single post by its ID
POST /posts/post/addcomment Create a comment under this post
POST /posts/post/rate Create a new user rating for this post

PostTypes
GET /posttypes Retrieve the list of all post types

Tags
GET /tags Get the 100 most popular and relevant tags (smart recommendation)
GET /tags/tag/posts Get 100 most recent posts order by use count descending with the tag parameter

Users

GET /users/profile Get the user's profile
PUT /users/profile Update the user's profile information
PUT /users/profile/passwordchange Change the user's password (allowable only for the user themself, or admins)
PUT /users/user/assignadmin/:email Assign Admin rights to this user (only admins)
PUT /users/user/revokeadmin/:email Revoke Admin rights for this user (only admins)
POST /users/fileupload Upload an images for this user's profile
GET /users/user/posts Retrieve all this user's posts
GET /users/user/groups Retrieve all the user's groups
DELETE /users/profile/:id Delete the user with this ID
GET /users/hasEditPermission Check if the user has edit permission rights (Admin, or creator of entity)

Backend Middleware

For all of the express routing, we developed some middleware functions to reduce code duplication and reduce the responsibility (and size) of the actual request handlers. This middleware makes it safe to make some assumptions in the actual api call, such as assuming there is a logged-in user, as any requests without a user would only get to the step 4 outlined below.

1) middleware.installHelpers
    Add some custom helper functions for
2) login Routing
    express router for handling non-authenticated urls
3) middleware.setupCORS
    configure CORS for ajax calls to protect the site (for security)

4) middleware.verifyUser

verify that the user is logged in

5) middleware.sendAngularHtml()

check if the request is expecting html or json. if it's html serve the angular index.html file, otherwise continue

6) api Routing,

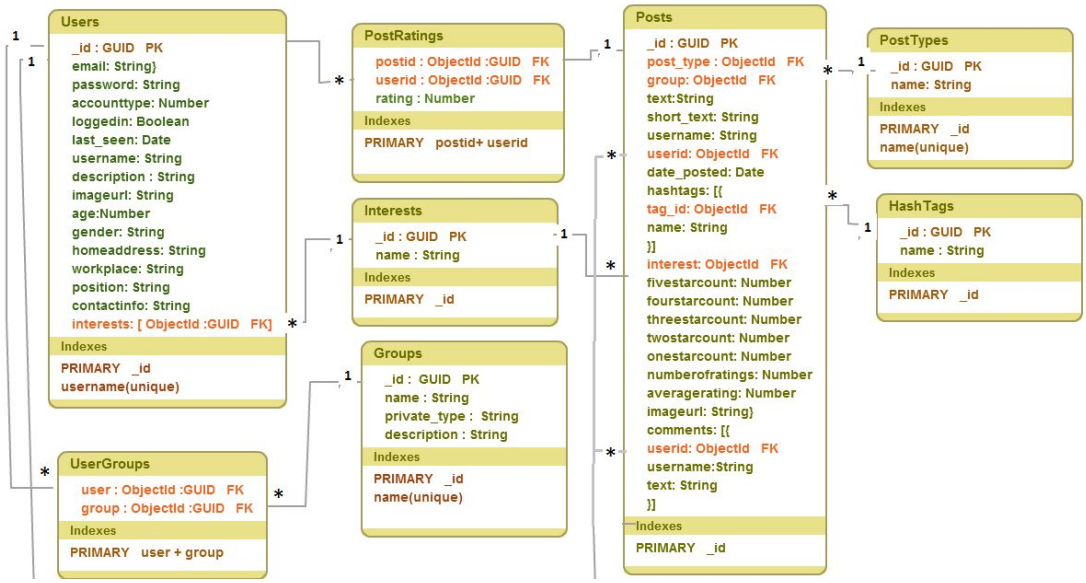
route the api call as described above

The middleware is implemented in middleware/index.js and installed in routes/index.js

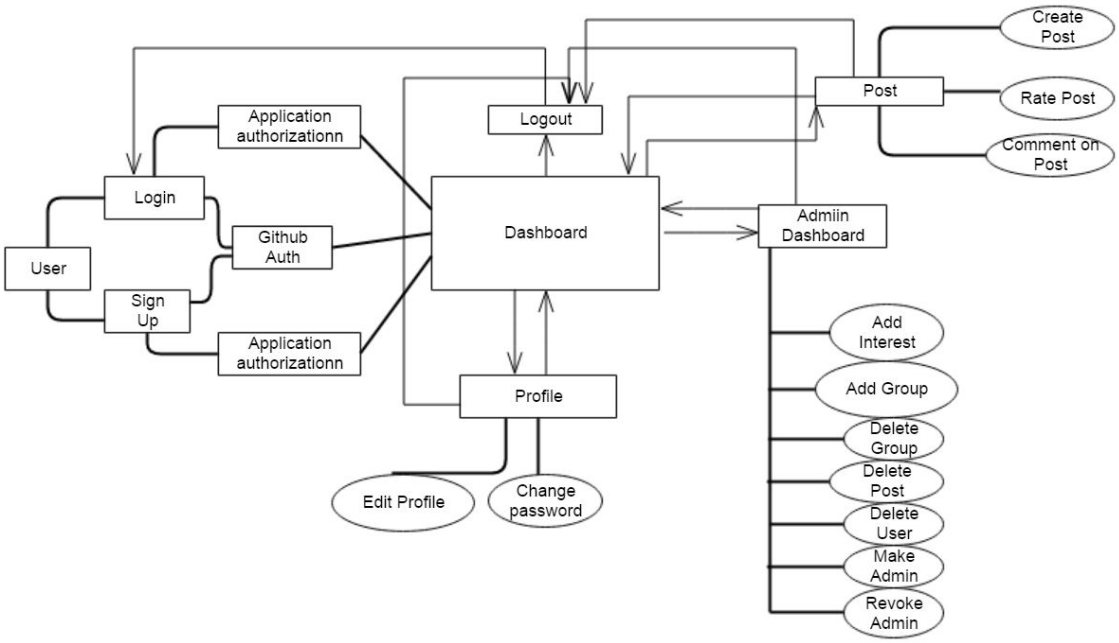
Database:

We are using MongoDB with the Mongoose ODM in our project. The database is used to save all information about the users, posts, and activity in our app. It is accessed only by the backend api. In order to improve the performance of the Database, we denormalized the database. We also implemented cascading deletes using the pre and post hooks, and used the Q module to work around asynchronous callbacks when we need to do a batch upsert of child entities before saving the parent entity.

The relational schema is provided below:



Functional Diagram:



Module Design:

Login Module

This module provides the user with the ability to sign up or login into the application. Ability to register via Github is implemented.

Dashboard Module

This module is used to display posts based on the smart recommendation system described above in the high-level project description. It allows the user to filter the posts based on group, hashtags, and interests that are listed on the left and right panel. The users can also navigate to posts, or to create new posts, or add themselves to the selected group. The search bar allow users to search by hashtag.

Profile Module

The profile displays the user's information. The user's own profile is accessible from the left nav bar. Other users' profiles can be accessed using the username link on the posts. It can be used to personalize user preferences that will be used in the smart recommendation system on the main feed in the dashboard. The user can also update their photo, and change their password.

Post Module

Create and view posts, rate and comment on a post. Categorize posts using Group, Interest, Hashtags.

Administration Module

Only the user with Admin rights can access the administration module. Using administrative dashboard, the Admin can manage the users, interests, groups, and posts.

UI Elements

Main Page:

- index.html

Templates:

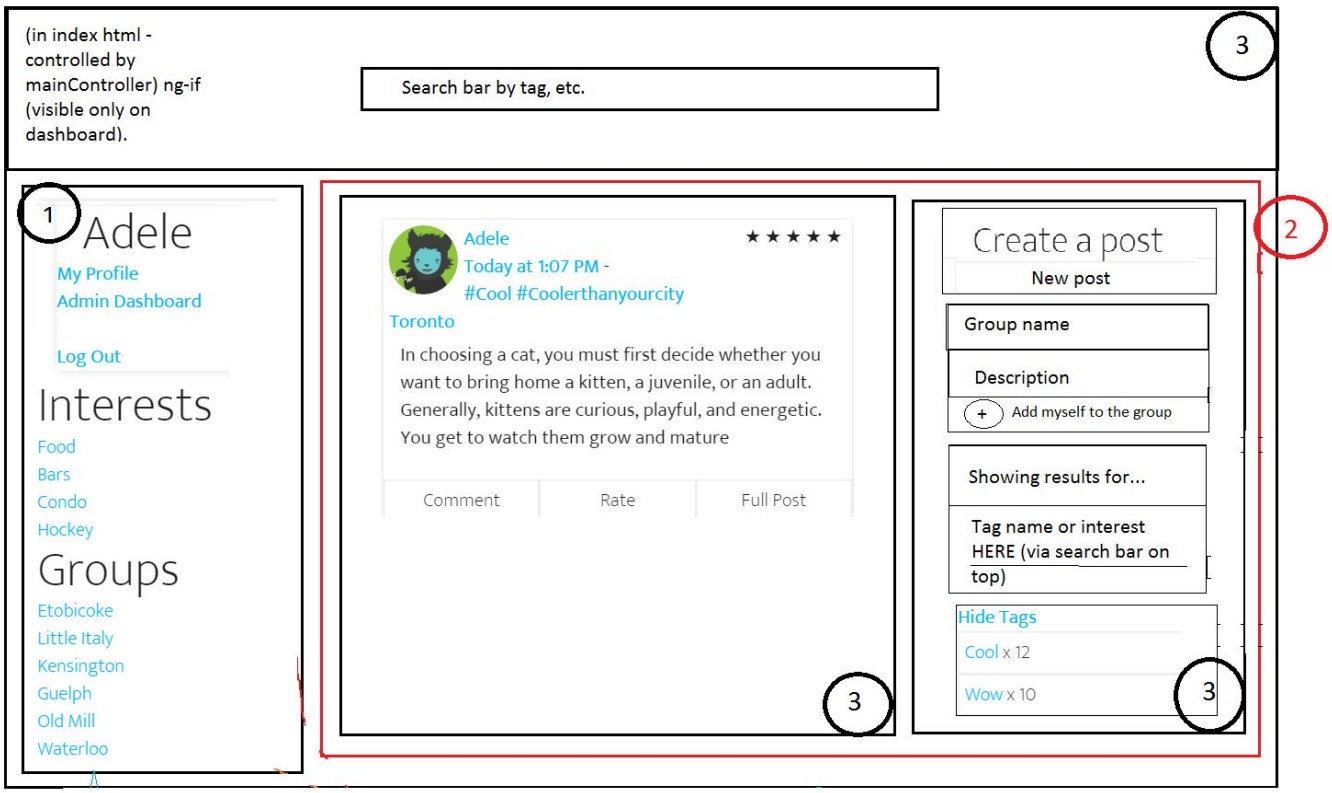
- feed.html
- login.html
- newpost.html
- postpermalink.html
- profile.html
- dashboard.html

Partial Views:

- groupList.html

- interestList.html
- postList.html
- userList.html
- loginbar.html
- post.html
- userbox_groups.html
- userbox_interests.html
- userbox_profile.html

Angular UI Architecture



All UI elements are located on the main index.html page and their visibility and inclusion in the DOM is controlled by angular directives (ng-show, ng-if).

1. Is marshalled by *mainController*. The top part of the left panel presents the navigation bar that is accessible from all templates in the view (except for the main login template).
2. **ng-view**: Is a container where all templates will be loaded.

3. Html template is marshalled by *feedController,* and is loaded in the ng-view.

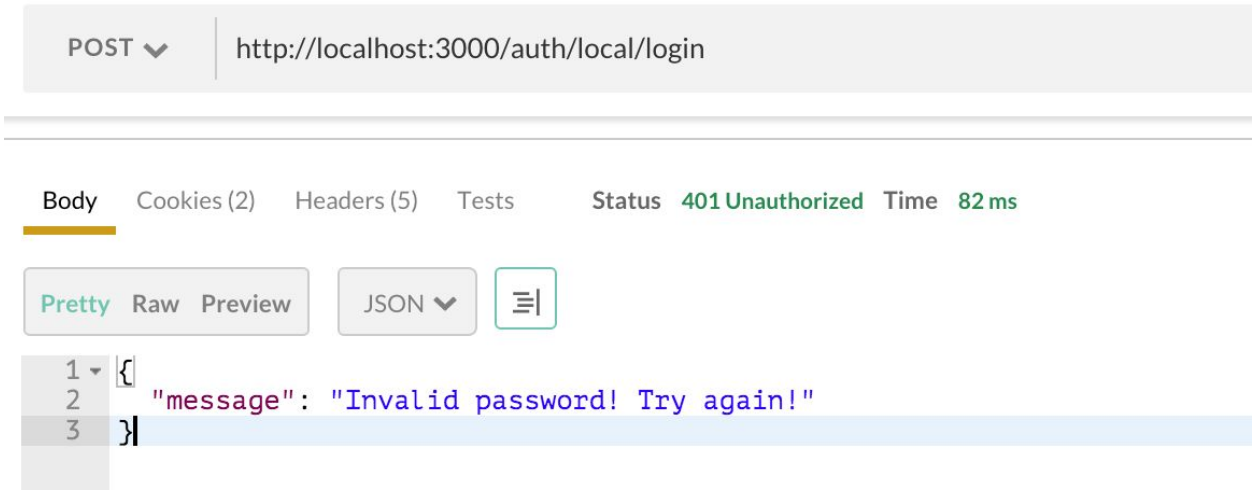
The implementation of Profile, Post, and Admin pages are implemented as templates that marshalled by their own controllers and are loaded in the ng-view.

Security:
We used passport to authenticate users, of whom can be either local or via github. In the authentication process, we store user passwords hashed using bcrypt, and use that same library to unhash and verify passwords as required. The reason we chose to do this is in the unfortunate event that a malicious user gains access to the database, they'll be unable to read user passwords.

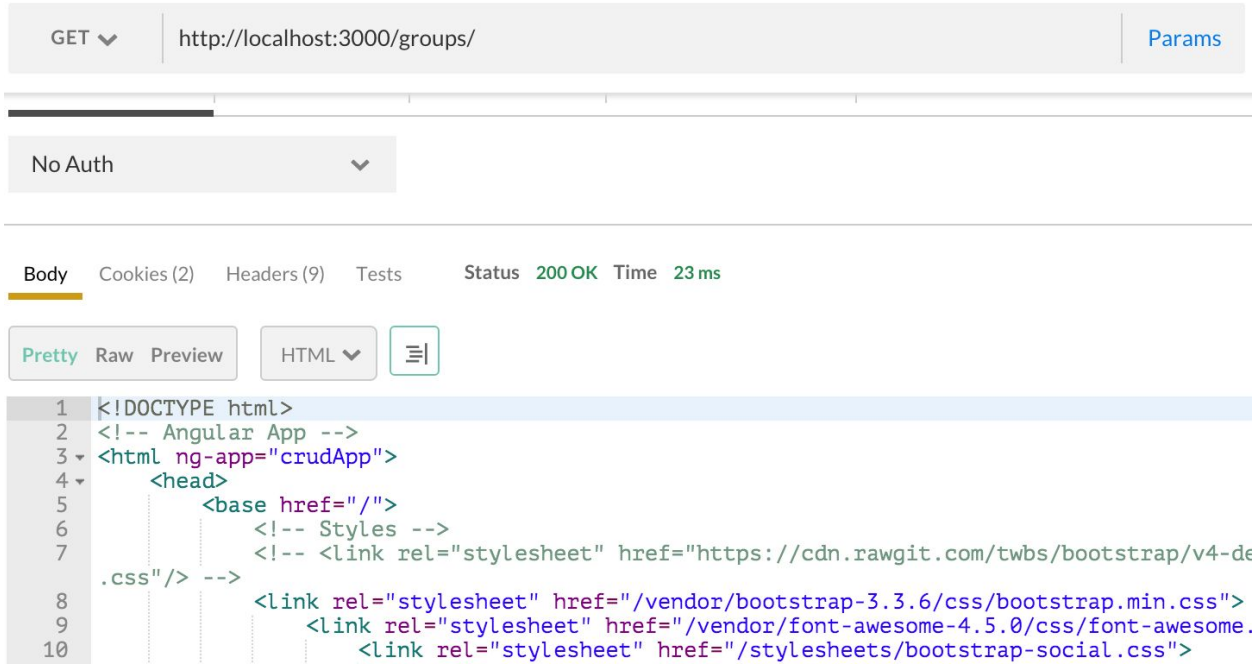
Furthermore, to ensure fully that no-one is able to access the backend api if they are unauthorized, we use express-session to keep track of logged in users, and at each sensitive api call we installed a custom middleware the checks the session's validity (logged in as a valid user in the db). This ensures a user must be logged in to use our api, and that they must be a real user.

To test the security, we used postman to try and access api calls without being logged in, we also tried calling api when the logged in user does not have the proper permissions. These tests assured us that our security measures were working as intended.

Here is an example of the kind of response when trying to login without the proper credentials (particularly with an invalid password):



Furthermore, when trying to access resources without being logged in, the requireLogin middleware runs and redirects the user to the login screen instead of sending an error:



This way we ensure a seamless user experience (no one likes 401 error screens). Because in this particular request we expect a list of JSON objects detailing all the public groups, but instead we are sent the login page.

Further examples of security tests can be found in the **Testing** section of this report.

Performance:
Example 1

One example of notable increased performance, particularly in regards to space efficiency, was when we changed user profile image uploads from each uploaded image occupying its own file to just overwriting the user's profile image when they upload a new one.

Example 2

As another example, recall that we used bcrypt to hash our passwords. Note that the way bcrypt works is it hashes the password multiple number of rounds as specified by the developer, each round taking exponentially longer than the one before. As a result of this, we had to choose the number of rounds that would be the optimal tradeoff for security (since each round makes the password more secure) and performance.

To determine this, we used locust to test ~200 users logging in to an account stored with passwords varying in the number of rounds used to hash the password. Note that when logging in, the password inputted by the user is hashed the same number of rounds as the one in the database and then string compared to see if they match. Because of this, logging in takes as long as hashing a new password.

Group A:
~200 users logging in at a rate of 20 per second with 4 rounds of bcrypt encryption.

Name	# reqs	# fails	Avg	Min	Max	Median	req/s
GET /	346	0(0.00%)	3	3	12	3	23.60
POST /auth/local/login	236	0(0.00%)	12	10	22	12	18.30
Total	582	0(0.00%)					41.90

As you can see, 236 users at a rate of 18/second logged in for an average of 12ms to verify that their password was correct.

Group B:
~200 users logging in at a rate of 20 per second with 8 rounds of bcrypt encryption.

Name	# reqs	# fails	Avg	Min	Max	Median	req/s
GET /	249	0(0.00%)	1162	72	2372	1300	15.40
POST /auth/local/login	214	0(0.00%)	932	80	1799	960	16.40
Total	463	0(0.00%)					31.80

As you can see, 214 users logged in at a rate of 16.4/second, for an average response time of 932ms, or 0.9 seconds.

Group C:
~50 users logging in at a rate of 2 per second with 12 rounds of bcrypt encryption.

Note that here, the decryption took so long that we couldn't run the tests at the same speed as the others without the requests failing and node crashing.

Name	# reqs	# fails	Avg	Min	Max	Median	req/s
GET /	30	0(0.00%)	13148	867	24729	13000	0.40
POST /auth/local/login	44	0(0.00%)	11910	916	22153	11000	1.50
Total	74	0(0.00%)					1.90

As you can see, 44 users logged in at a rate of 1.5/second, for an average response time of 11920ms, or 11.9 seconds.

As is extremely evident, 12 rounds proved far too inefficient to be a viable course of action for our site. It came down to 4 or 8 rounds, and we decided that for the far greater amount of security offered by 8 rounds compared to 4, it was worth the increased performance cost. Thus this was another of the optimizations we made.

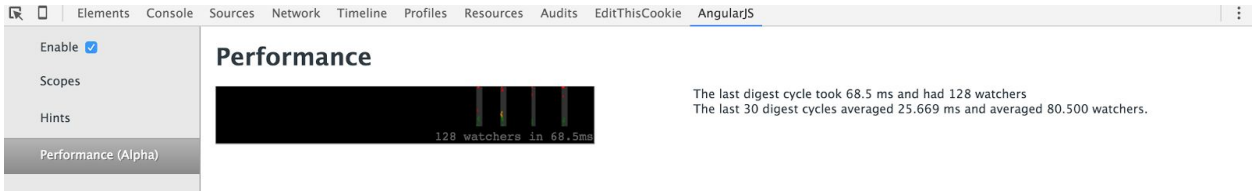
Example 3

As for an allround performance test, consider the following results from locust:

Name	# reqs	# fails	Avg	Min	Max	Median	req/s
GET /	4454	0(0.00%)	334	3	10129	8	47.40
GET /admin	6372	0(0.00%)	469	3	13483	9	71.50
POST /auth/local/login	114	0(0.00%)	7184	6169	8519	7100	0.00
GET /profile	2131	0(0.00%)	504	4	13399	10	23.90
PUT /users/profile	2203	0(0.00%)	388	4	11541	10	23.90
Total	15274	0(0.00%)					166.70

Considering there was a huge spike in the response time at the beginning because we were logging in 1000 users at 20/second, we will use the median as our measure of performance (since the spike skewed the average greatly).

These results tell us that varying requests to different parts of the site take a really small amount of time: 8 - 10ms. The reason this is the case is that all the routing on the server is either api calls, or sending the required files to angular to let it do all the routing. It is for that reason that we must measure angular's performance measured via AngularJS Batarang:



As we can see, angularjs spends a lot of time running. Since we having angular doing all the routing and templating, we essentially move all the server performance issues

associated with those tasks onto the client, allowing us to effectively serve the highest amount of users without any performance hits.

Testing:

In `test.js` in the root folder we have a mocha test that tests multiple aspects of the website. Here's an example of the output after running

`$ node_modules/mocha/bin/mocha test`

from the project's root directory:

```
Adams-MacBook:csc309-a5 adambenn$ node_modules/mocha/bin/mocha test


  Full Test
[Function]
    /
connection successful
Sun Dec 06 2015 21:44:31 GMT-0500 (EST) 1449456271256
Hello we are sitting on test function in app.js!
POST /auth/local/signup 401 266.166 ms - -
GET / 200 4.959 ms - 4191
      ✓ Expecting a 200 OK (237ms)
    Login Tests
      Successful Logins
POST /auth/local/login 200 65.175 ms - -
        ✓ Logging in as Adam should give 200 (68ms)
POST /auth/local/login 200 59.729 ms - -
        ✓ Logging in as Adele should give 200 (62ms)
      Failed Logins
        ✓ Logging in as fake account should give 401
POST /auth/local/login 401 2.359 ms - -
POST /auth/local/login 401 64.663 ms - -
        ✓ Logging in with wrong password should give message 'Invalid password! Try again!' (68ms)
POST /auth/local/login 401 1.493 ms - -
        ✓ Logging in with non-existant email should give message 'No user found with those credentials!'
    Registration Tests
      Successful Registrations
Saving
POST /auth/local/signup 200 72.868 ms - -
        ✓ Registering a new user properly should give 200 (76ms)
      Failed Registrations
POST /auth/local/signup 401 1.017 ms - -
        ✓ Registering without a username should fail 401 and give message 'Username missing!'
POST /auth/local/signup 401 0.810 ms - -
        ✓ Registering without an email should fail 401 and give message 'Missing credentials'
POST /auth/local/signup 401 2.161 ms - -
        ✓ Registering with an existing email should fail 401 and give message 'Email already in use!'
POST /auth/local/signup 401 2.719 ms - -
        ✓ Registering with an existing username should fail 401 and give message 'Username already in us
e!'
    API calls
      Test that API calls redirect to Login
GET /groups/ 200 3.194 ms - 4191
        ✓ Response should not be JSON since not logged in


  12 passing (1s)
```