CarPool Website:
Improve your travel experience

Section A. Introduction

Project Title: CarPool

Team members: **Xu Zhang (c5zhanic), Ran Yang (g5ran), He Zhang (c5zhanie), Tin Yau Yu (c5yutiny)**

Hosting on: https://carpool309.herokuapp.com/

(super admin: admin@admin.com; admin)

During this project, we build a Carpool website that helps people find others to make sharing of car journeys. The idea of carpooling is to have more people sharing the same vehicle so that we can reduce each person's travelling costs involved in repetitive or long-distance driving. It even has the potential to reduce air pollution and traffic congestion which will benefit all of us. After extensive research, we find that there is no similar website on the market with good design, user friendly interface and easy to use functionalities. Therefore, we decided to build a carpool website for providing this service.
In this website, people will be able to register trips as provider or wanter so that others within this social network can search in the website to find the trip they want to join or provide by providing origin, destination and departure time. We have also introduced a Rating and Commenting system much like what is in Airbnb. Each user can post reviews for others who have previously shared a trip with him or her. The chat system on the website gives users the ability to communicate with each other in real time. Even when they are offline, their messages will be seen once they logged into their account. All of these functionalities set up a tight connection between website users, and sharing economy have been well embodied and exerted.

Section B. Design

<u>Overview</u>

By user experience, our website have 2 phases. On the first stage, the user is required to login or register to start the web app. After that, the user will enter our graceful interface for all the main functionalities. In the interface, the user can find their trip by either direct contact other users from the user list or can search the trip by entering their destination.

By functionality, our system is divided by four parts: User Account, Feedbacks, Message and Trips. Each functionality is spanned in all layers to provide a completed service to users.

<u>Front-end</u>

In the front-end of our application, we have different pages to serve various functions of our system. To each functionality, the page will make appropriate HTTP requests to the server, receive server's echo and display the result to users.

(1) public/getStarted.html ('/')

This is the page which users will first see when accessing our web application. User can login with their email and password, their existing Google ID, or register a new account in this page.

(2) view/userList.html ('/users')

This is the home page of our application after login. Users can access functions of our applications here. This includes to view all or search specific users in our application, to search a new trip, and to view the user's search history.

On the top of each page of the application, there is a navigation bar which allows users to return to this page, see new message notification, visit his own profile and to sign out from the system.

(3) view/profile.html ('/users/{user_id}/?trip={trip_id}')

This is the page to show a user's profile. This includes the email, display name, description, user type (only for administrators) and profile picture of the users. If the profile belongs to the login user or the login user is an administrator, the user can edit the profile. The page also includes the rating statistics and comments from other users. Users can leave their own comment and rating in this page. Leaving a comment or rating on his own profile is prohibited by the system. For administrators, they can view the specific user's behavior information such as the browser used, the type of OS, screen size and location.

If trip id is specified, an extra frame will show the corresponding trip information in the profile page, including the departure date and time, expected price and the google map to show the start point and end point of the trip.

(4) view/chatWindow.html

This is the chat window which will be called when user starts a conversation with another user. User can perform one-to-one real time chat with another user here. He can also leave an offline message if the user is not online at the moment.

(5) view/trips.html (‘/searchTrip/{trip_id}’)

This is the page to show the matched trips and similar trips correspond to the trip parameters. The matched trips are trips to meet user’s requirements, and the similar trips are the trip requests with alike requirements. User then can click the trip to view the trip detail and the corresponding user profile.

(6) view/admin.html (‘/admin’)

This page is the overview of the status of our application, which only available to administrators. This includes the number of current online users, the list of all users, the list of all reviews and all trips that are searched in our application. Administrators can review and delete inappropriate comments and trips here if necessary by clicking the cross button near the comment or trip.

<u>Back-end - HTTP requests and API (‘server.js’)</u>

In the server-side of our application, we have the “server.js” as the entry point and to serve all http requests to the server. Based on the nature of request, functions will be called from different modules to generate and return appropriate echo to clients. The list of HTTP requests handled and API calls available:

User Interface		
Method	**URL**	**Description**
(all)	/	if user is login, redirect to ‘/users’, else redirect to ‘getStarted.html’
GET	/users	return the application homepage after login
GET	/users/{user_id}	return the specific user profile
GET	/admin	return the admin panel page

GET	/searchTrip/{trip_id}	return the matched and similar trips page

User Account API		
Method	**URL**	**Description**
DELETE	/api/users/{user_id}	Delete a speccific user by get the user's id through the request
POST	/api/login	get the login data and check with the database if login succeed go to certain page if not, report error message
GET	/api/logout	Logout the user by resetting the session and redirect to the home page
POST	/api/users	Create new user
PATCH	/api/users/{user_id}	Update the user profile (name, email, icon)
PUT	/api/changePassword	Get the verified password from user and update the password in the database
GET	/api/users	GET command which send all users in the database to the client
GET	/api/users/search/?keyword ={keyword}	GET command which return user who satisfy specifis search conditions
GET	/api/users/current	Get the current user's data from database and send to user

GET	/api/users/{user_id}	GET the user data according to user's ID
GET	/api/users/current/profilePic	Get the profile pic of the current user by knowing keyword current
GET	/api/users/:id/profilePic	GET specific user's icon pic by user's ID
GET	/api/log	Enable to trace the most visited pages log

Feedback API		
Method	**URL**	**Description**
POST	/api/users/:id/feedbacks	create new Feedback api
GET	/api/users/:id/feedbacks	get feedbacks for a user id api
GET	/api/feedbacks	get all feedbacks api
GET	api/feedbacks/:id	get one feedback by it's id api
DELETE	/api/feedbacks/:id	delete one feedback by it's id api

Message API		
Method	**URL**	**Description**

GET	/api/users/:email/chatWindow/	get one chatWindow by it's email api
GET	/api/unreadmessage/:email/	get one's unreadmessage by it's email api
POST	/api/markMsgRead/:sender/:receiver/	mark unread message to read by sender and receiver api
GET	/api/getConversation/:user1/:user2/	get an conversation by user1 and user2 emails api

Trip API		
Method	**URL**	**Description**
POST	/api/updateTrip	Update the trip save to the database
GET	/api/trips	GET all trips in the trip database and return to user
DELETE	/api/trips/{trip_id}	Delete one specific trip by it's id

<u>Back-end - Modules</u>

The modules are to support the functions of pages and API calls. They are in charge of manipulation of mongodb, receive function calls and return data to server.js.

(1) AccountManager.js

This module will handle all operations regarding on users, including:

- verify user login (on-system accounts and Google ID)
- create, update and delete user profile (including rating information of the user)
- fetch all or specific fields from user profile
- search and return users by keyword

(2) FeedbackManager.js

This module will handle all operations regarding the comments and rating , including:

- create and delete reviews
- fetch all reviews of a specific users or all users
- fetch a specific review

(3) MessageManager.js

This module will handle all operations regarding messaging between users, including:

- track number of online users
- fetch messaging history between two users
- fetch unread messages for a user
- mark messages as read

(4) TripManager.js

This module will handle all operations regarding the car trips, including:

- create and delete trips
- fetch a specific trip / all trips
- search similar trip to a given trip
- match trip requests by considering the distance, time and price

<u>Database</u>

We use mongodb to store the data of our application. Four collections are created corresponding to the modules:

(1) Users

The collection is used by AccountManager to store all users information:

- profile information: email, hashed password, display name, description, profile picture

- rating information: number of rating, average rating, number of stars
- user behavior information: user type, browser, os, screen size, mobile device, location

(2) Feedbacks

The collection is used by FeedbackManager to store the reviews given by users:

- sender and receiver id
- comment and rating
- date and time received

(3) Messages

The collection is used by MessageManager to store the messages sent by users:

- sender and receiver id
- content
- date and time sent
- status of message being read

(4) Trips

The collection is used by TripManager to store the trip request posted by users:

- user id
- start point and end point of the trip (text descriptions and coordinates)
- date and price of the trip
- type of the trip (trip provider or trip wanter)
- largest offset allowed in matching with other trip requests

Section C. Security Vulnerability

In this project, we have tackled the security vulnerabilities of Cross-Site Scripting (XSS) and Denial-Of-Service (DOS).

(i)Cross-Site Scripting (XSS)

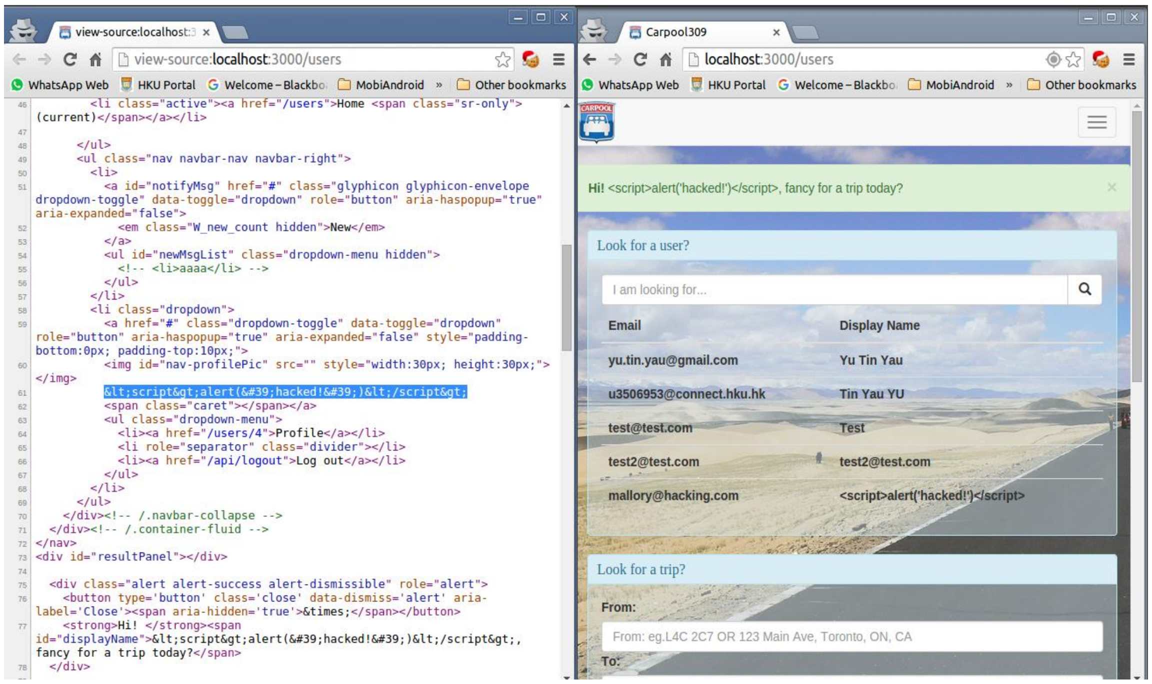
Cross-Site Scripting is a technique for malicious users to inject client-side script into web pages viewed by other users. This vulnerability may be used by attackers to bypass access controls such as the same-origin policy.

In our system, users are allowed to pick their name and profile picture which will be shown to other users; and in user's profile, their descriptions will also be displayed. Also, users can leave arbitrary comments on other users' profile. These are vulnerabilities which attackers may inject JavaScript to perform unauthorized action without noticing the current users.

In order to protect our system from XSS, we apply a filter to user input before saving the content into our database. This is done by using the 'xss-filters' module (https://github.com/yahoo/xss-filters). Using this module in our server-side coding, we can escape "<" character in HTML text and filter out dangerous attributes and keywords (e.g. "javascript", "onload" etc.) after receiving user's input.

In order to test the effect of the 'xss-filters', we created a malicious user who tried to inject some JavaScript into the profile page. The following cases are tested:

(1) Injecting JavaScript using <script> tag in HTML text
(2) Injecting JavaScript using 'onload' attribute
(3) Injecting JavaScript using 'javascript:' in 'src' attribute



```
      <li class="active"><a href="/users">Home <span class="sr-only">
(current)</span></a></li>

    </ul>
    <ul class="nav navbar-nav navbar-right">
      <li>
        <a id="notifyMsg" href="#" class="glyphicon glyphicon-envelope
dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true"
aria-expanded="false">
          <em class="W_new_count hidden">New</em>
        </a>
        <ul id="newMsgList" class="dropdown-menu hidden">
          <!-- <li>aaaa</li> -->
        </ul>
      </li>
      <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown"
role="button" aria-haspopup="true" aria-expanded="false" style="padding-
bottom:0px; padding-top:10px;">
        <img id="nav-profilePic" src="" style="width:30px; height:30px;">
</img>
        &lt;script&gt;alert(&#39;hacked!&#39;)&lt;/script&gt;
        <span class="caret"></span></a>
        <ul class="dropdown-menu">
          <li><a href="/users/4">Profile</a></li>
          <li role="separator" class="divider"></li>
          <li><a href="/api/logout">Log out</a></li>
        </ul>
      </li>
    </ul>
  </div><!-- /.navbar-collapse -->
 </div><!-- /.container-fluid -->
</nav>
<div id="resultPanel"></div>

  <div class="alert alert-success alert-dismissible" role="alert">
    <button type='button' class='close' data-dismiss='alert' aria-
label='Close'><span aria-hidden='true'>&times;</span></button>
    <strong>Hi! </strong><span
id="displayName">&lt;script&gt;alert(&#39;hacked!&#39;)&lt;/script&gt;,
fancy for a trip today?</span>
  </div>
```

Hi! <script>alert('hacked!')</script>, fancy for a trip today?

Look for a user?

Email	Display Name
yu.tin.yau@gmail.com	Yu Tin Yau
u3506953@connect.hku.hk	Tin Yau YU
test@test.com	Test
test2@test.com	test2@test.com
mallory@hacking.com	<script>alert('hacked!')</script>

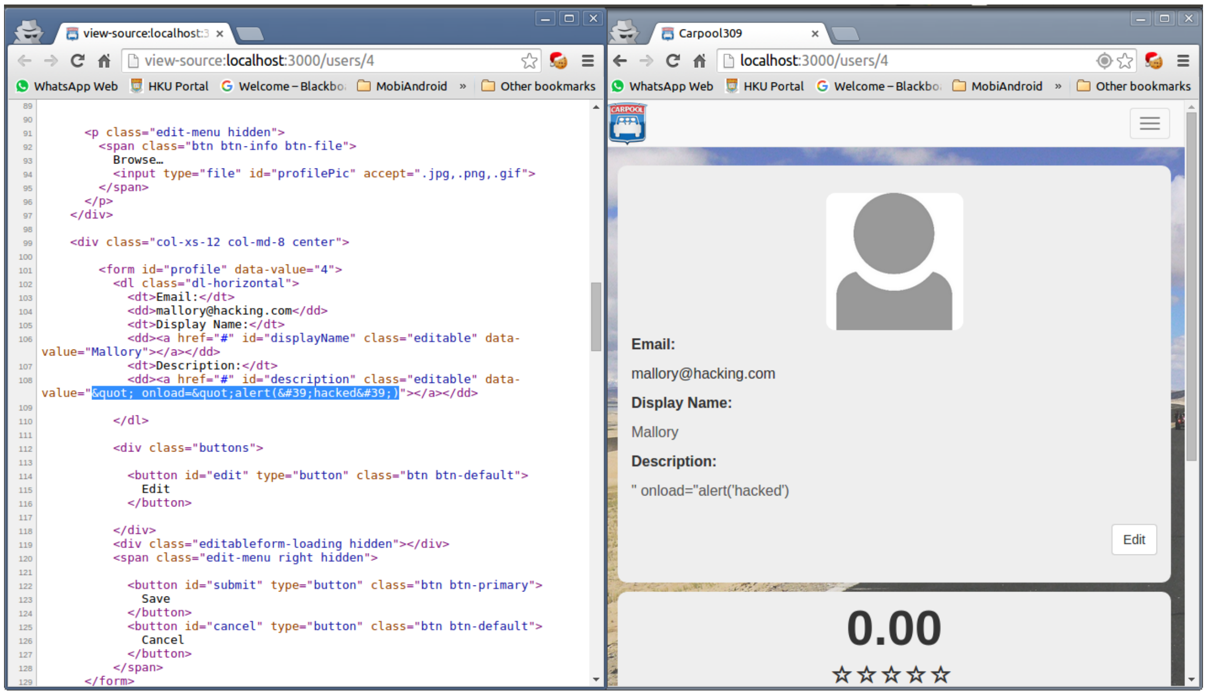
Look for a trip?

From:

To:

view-source:localhost:3000/users
localhost:3000/users
Carpool309
I am looking for...
From: eg.L4C 2C7 OR 123 Main Ave, Toronto, ON, CA

In the first case, the user injects JavaScript using <script> tag in his display name. However, the character '<', '>' and quotation marks are escaped. This prevents the execution of the injected JavaScript.



```


      <p class="edit-menu hidden">
        <span class="btn btn-info btn-file">
          Browse…
          <input type="file" id="profilePic" accept=".jpg,.png,.gif">
        </span>
      </p>
    </div>

    <div class="col-xs-12 col-md-8 center">

        <form id="profile" data-value="4">
          <dl class="dl-horizontal">
            <dt>Email:</dt>
            <dd>mallory@hacking.com</dd>
            <dt>Display Name:</dt>
            <dd><a href="#" id="displayName" class="editable" data-
value="Mallory"></a></dd>
            <dt>Description:</dt>
            <dd><a href="#" id="description" class="editable" data-
value="&quot; onload=&quot;alert(&#39;hacked&#39;)"></a></dd>

          </dl>

          <div class="buttons">

            <button id="edit" type="button" class="btn btn-default">
              Edit
            </button>

          </div>
          <div class="editableform-loading hidden"></div>
          <span class="edit-menu right hidden">

            <button id="submit" type="button" class="btn btn-primary">
              Save
            </button>
            <button id="cancel" type="button" class="btn btn-default">
              Cancel
            </button>
          </span>
      </form>
```

Email:

mallory@hacking.com

Display Name:

Mallory

Description:

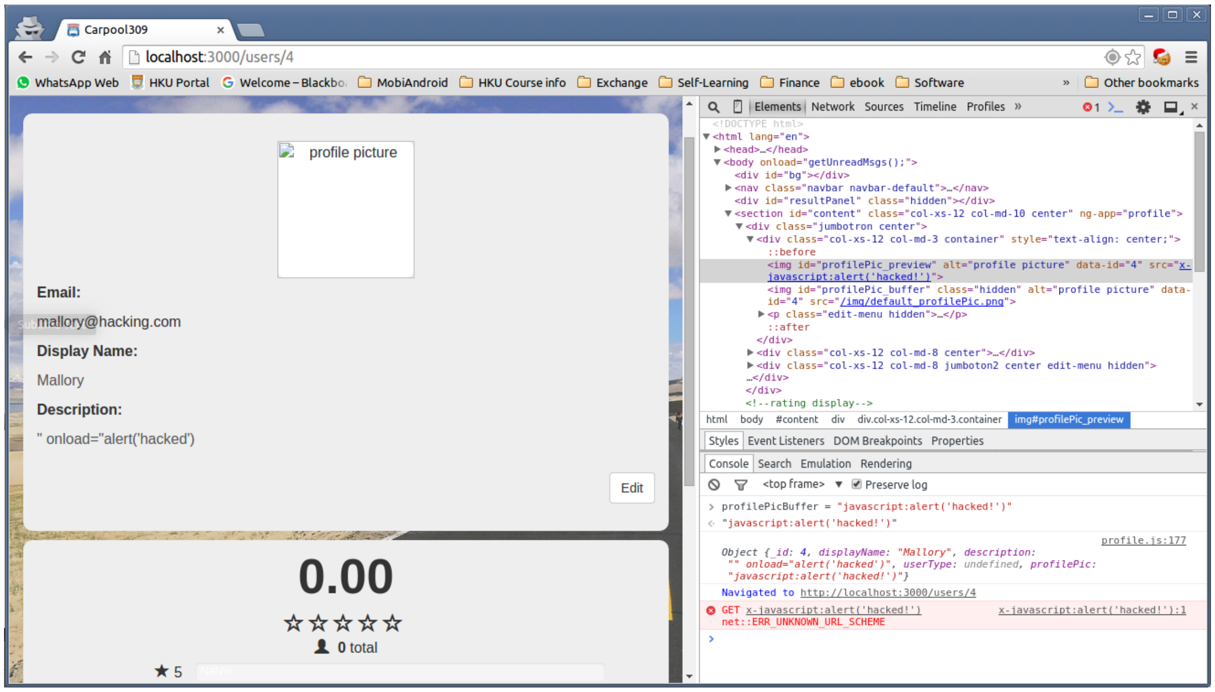
" onload="alert('hacked')

Edit

0.00

☆☆☆☆☆

In the second case, the user tries to inject JavaScript using 'onload' attribute in his description. The user intended to close the value attribute with a double quote, then insert an ‘onload’ attribute will contain the JavaScript. Yet, the quotation marks are being escaped again, thus the JavaScript is not working.



Email:

mallory@hacking.com

Display Name:

Mallory

Description:

" onload="alert('hacked')

0.00

☆☆☆☆☆

0 total

★ 5

```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
  ▼<body onload="getUnreadMsgs();">
     <div id="bg"></div>
    ▶<nav class="navbar navbar-default">…</nav>
     <div id="resultPanel" class="hidden"></div>
    ▼<section id="content" class="col-xs-12 col-md-10 center" ng-app="profile">
      ▼<div class="jumbotron center">
        ▼<div class="col-xs-12 col-md-3 container" style="text-align: center;">
           ::before
           <img id="profilePic_preview" alt="profile picture" data-id="4" src="x-
           javascript:alert('hacked!')">
           <img id="profilePic_buffer" class="hidden" alt="profile picture" data-
           id="4" src="/img/default_profilePic.png">
          ▶<p class="edit-menu hidden">…</p>
           ::after
         </div>
        ▶<div class="col-xs-12 col-md-8 center">…</div>
        ▶<div class="col-xs-12 col-md-8 jumboton2 center edit-menu hidden">
        …</div>
       </div>
       <!--rating display-->
```

```
html  body  #content  div  div.col-xs-12.col-md-3.container  img#profilePic_preview
```

```
> profilePicBuffer = "javascript:alert('hacked!')"
< "javascript:alert('hacked!')"
                                                              profile.js:177
  Object {_id: 4, displayName: "Mallory", description:
   "" onload="alert('hacked')", userType: undefined, profilePic:
   "javascript:alert('hacked!')"}
  Navigated to http://localhost:3000/users/4
  GET x-javascript:alert('hacked!')              x-javascript:alert('hacked!'):1
  net::ERR_UNKNOWN_URL_SCHEME
>
```

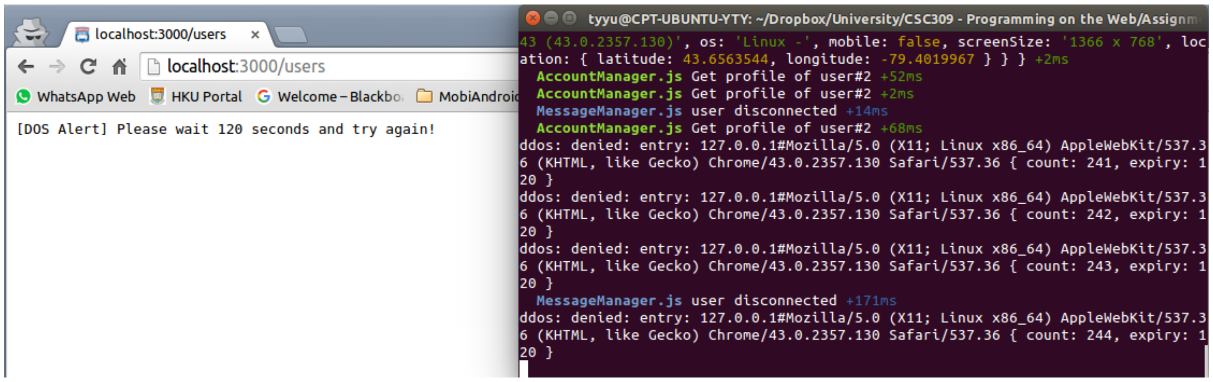
In the third case, the user tries to change its profile picture URL to the JavaScript prefixed with ‘javascript:’, intended to make browser execute the injected JavaScript when the profile picture is loaded. However, the ‘xss-filters’ sanitize the word ‘javascript:’ and replaced it with ‘x-javascript’. The injected JavaScript is not working again.

(ii) Denial-Of-Service (DOS)

The objective of a denial-of-service (DoS) is to make the network resource unavailable to its intended users by extensive access of that. Although there are no absolute method to prevent the DOS attack, we can minimize its effect by restricting the access of suspicious users from high-cost operations such as searching and writing to database.

In order to recognize potential DOS attacks and restrict these accesses, we implemented the ‘ddos’ module (https://github.com/rook2pawn/node-ddos) which will stop users to access our system if their HTTP requests are too intense in a short period of time.

The module will maintain an internal table which records every request marks from each host. For each HTTP access, the ‘count’ for the host will increase by 1. If the count exceeds the ‘burst amount’, then ‘expiry’ will grow in exponential. When the ‘count’ exceeds the ‘limit’, requests of the corresponding host will be denied, otherwise, the requests are permitted. For each time the internal table is accessed, the ‘expiry’ goes down by the time elapsed. And when expiry reaches 0, the count will be reset. The denied user will thus have to wait for the time ‘expiry’ before he can access the service again.



```
[DOS Alert] Please wait 120 seconds and try again!
```

```
43 (43.0.2357.130)', os: 'Linux -', mobile: false, screenSize: '1366 x 768', loc
ation: { latitude: 43.6563544, longitude: -79.4019967 } } } +2ms
  AccountManager.js Get profile of user#2 +52ms
  AccountManager.js Get profile of user#2 +2ms
  MessageManager.js user disconnected +14ms
  AccountManager.js Get profile of user#2 +68ms
ddos: denied: entry: 127.0.0.1#Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.3
6 (KHTML, like Gecko) Chrome/43.0.2357.130 Safari/537.36 { count: 241, expiry: 1
20 }
ddos: denied: entry: 127.0.0.1#Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.3
6 (KHTML, like Gecko) Chrome/43.0.2357.130 Safari/537.36 { count: 242, expiry: 1
20 }
ddos: denied: entry: 127.0.0.1#Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.3
6 (KHTML, like Gecko) Chrome/43.0.2357.130 Safari/537.36 { count: 243, expiry: 1
20 }
  MessageManager.js user disconnected +171ms
ddos: denied: entry: 127.0.0.1#Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.3
6 (KHTML, like Gecko) Chrome/43.0.2357.130 Safari/537.36 { count: 244, expiry: 1
20 }
```

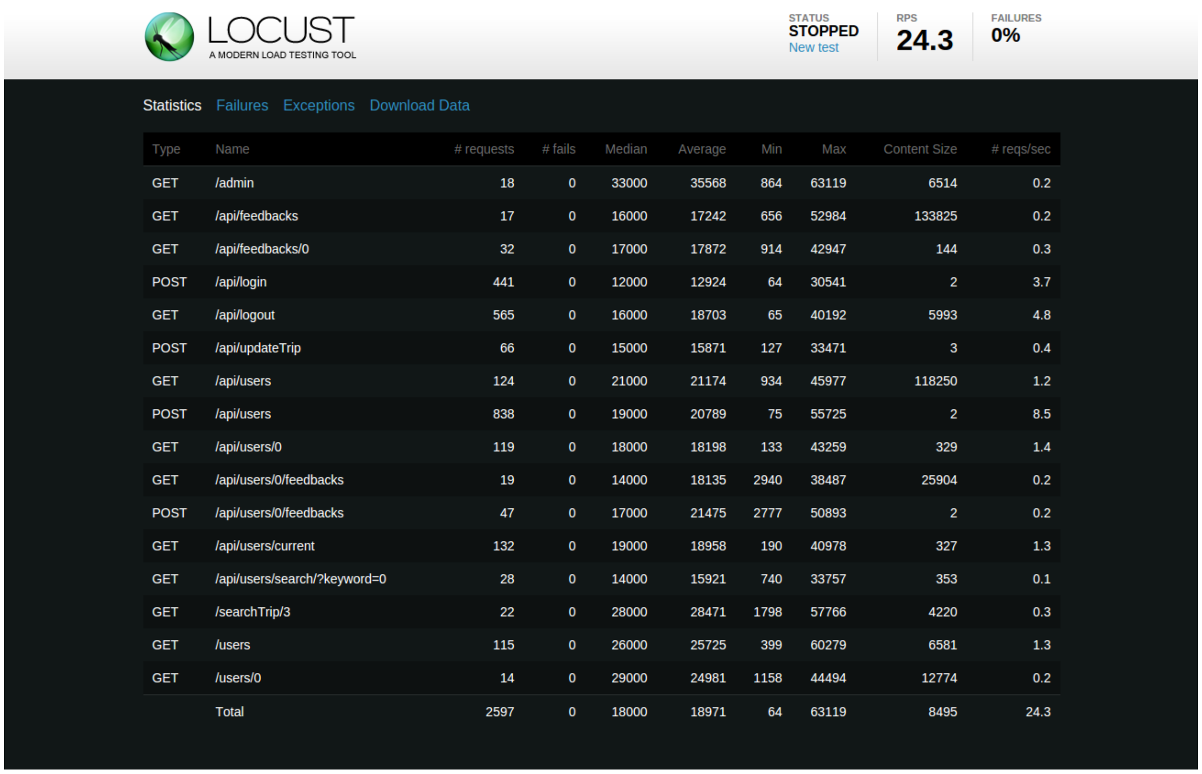
To test the function of ‘ddos’ module, we simulate the DOS attack by continuous refreshing of a page of our service. When the number of HTTP exceeds 240 (count>240), the server will deny the access of the host. This can thus prevent malicious users to occupy high-cost resources of our application.

Section D. Performance Testing

For performance testing, we have chosen locust.io as the tool to perform massive HTTP GET and POST requests. The locustfile.py is the python file which simulate variety of HTTP requests to the server. The basic logic here is for each user we register them and log out in order to perform log in. After these, users will perform different functionalities by probabilities and densities. Meanwhile, locust.io will help to record the data. The whole optimization is done by three versions. Version 1 is the one without any optimization. Version 2 is with front end optimization. Version 3 is the final version with optimization in back end.

- **Version 1, Original performance result**

Below is the screen shot of locut.io result:



Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
GET	/admin	18	0	33000	35568	864	63119	6514	0.2
GET	/api/feedbacks	17	0	16000	17242	656	52984	133825	0.2
GET	/api/feedbacks/0	32	0	17000	17872	914	42947	144	0.3
POST	/api/login	441	0	12000	12924	64	30541	2	3.7
GET	/api/logout	565	0	16000	18703	65	40192	5993	4.8
POST	/api/updateTrip	66	0	15000	15871	127	33471	3	0.4
GET	/api/users	124	0	21000	21174	934	45977	118250	1.2
POST	/api/users	838	0	19000	20789	75	55725	2	8.5
GET	/api/users/0	119	0	18000	18198	133	43259	329	1.4
GET	/api/users/0/feedbacks	19	0	14000	18135	2940	38487	25904	0.2
POST	/api/users/0/feedbacks	47	0	17000	21475	2777	50893	2	0.2
GET	/api/users/current	132	0	19000	18958	190	40978	327	1.3
GET	/api/users/search/?keyword=0	28	0	14000	15921	740	33757	353	0.1
GET	/searchTrip/3	22	0	28000	28471	1798	57766	4220	0.3
GET	/users	115	0	26000	25725	399	60279	6581	1.3
GET	/users/0	14	0	29000	24981	1158	44494	12774	0.2
	Total	2597	0	18000	18971	64	63119	8495	24.3

As you can see, there are 0% failures with a 24.3 request per seconds. This is performed by having around 1600+ users at the same time according to the locust. The highest average response time is for the GET /admin since it is retrieving all data from all our databases.

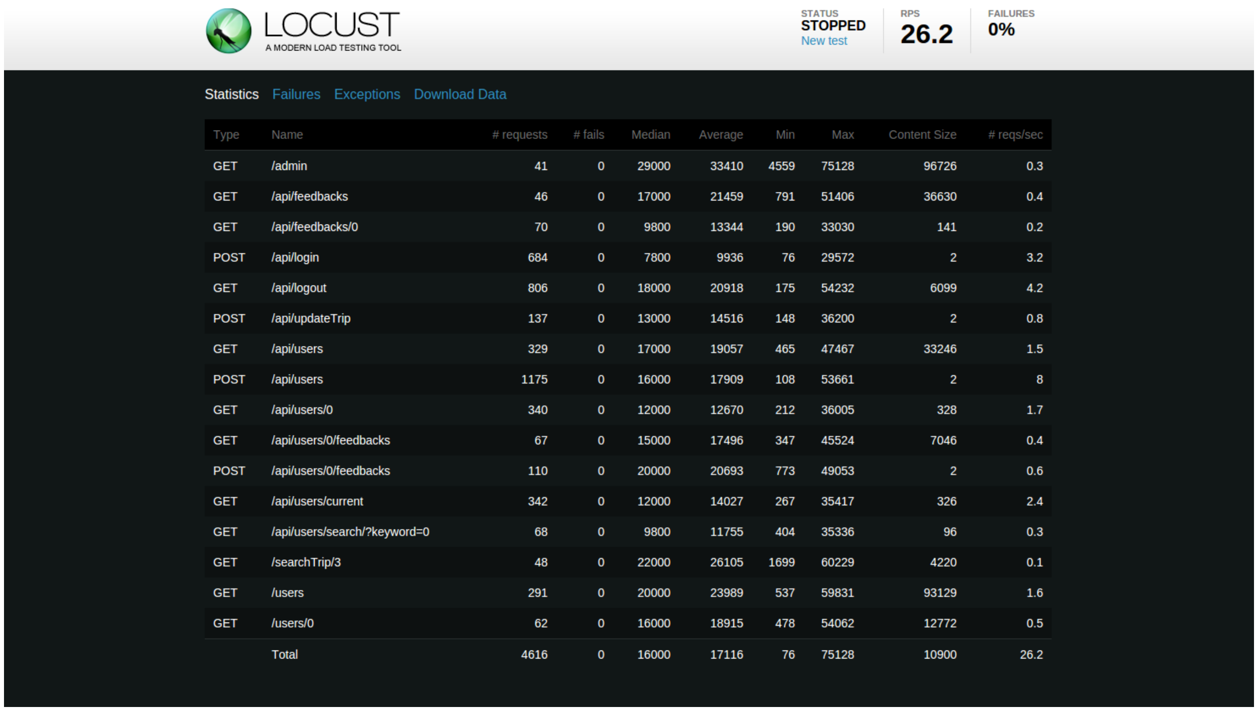
- **Version 2, +Front end optimization result**

For front end optimization, we have used all two techniques learned in tutorial, gzip and cache control:

For ***gzip***, we have implement and require the compression package to compress each html, javascript, css files processed with server and client. In such a way, the time to transmit these files between server and client will be reduced due to the shrink of file size.

For ***cache control***, we have choose to cache all files in the /public directory which are all front end html, javascript and css files. We store them into the cache once the server get them. Since these file are mostly visited, caching them reduce lots of time for server to get them from client side.

Below is the screen shoot for the result:



Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
GET	/admin	41	0	29000	33410	4559	75128	96726	0.3
GET	/api/feedbacks	46	0	17000	21459	791	51406	36630	0.4
GET	/api/feedbacks/0	70	0	9800	13344	190	33030	141	0.2
POST	/api/login	684	0	7800	9936	76	29572	2	3.2
GET	/api/logout	806	0	18000	20918	175	54232	6099	4.2
POST	/api/updateTrip	137	0	13000	14516	148	36200	2	0.8
GET	/api/users	329	0	17000	19057	465	47467	33246	1.5
POST	/api/users	1175	0	16000	17909	108	53661	2	8
GET	/api/users/0	340	0	12000	12670	212	36005	328	1.7
GET	/api/users/0/feedbacks	67	0	15000	17496	347	45524	7046	0.4
POST	/api/users/0/feedbacks	110	0	20000	20693	773	49053	2	0.6
GET	/api/users/current	342	0	12000	14027	267	35417	326	2.4
GET	/api/users/search/?keyword=0	68	0	9800	11755	404	35336	96	0.3
GET	/searchTrip/3	48	0	22000	26105	1699	60229	4220	0.1
GET	/users	291	0	20000	23989	537	59831	93129	1.6
GET	/users/0	62	0	16000	18915	478	54062	12772	0.5
	Total	4616	0	16000	17116	76	75128	10900	26.2

We can see there are some obvious time saving in average response time in most of HTTP requests. And the request per second also increase by 2 which is almost 10% improvements based on the version 1 result. This result is also simulated with around 1600+ users according to the data on locust.

- **Version 3, +Back end optimization result**

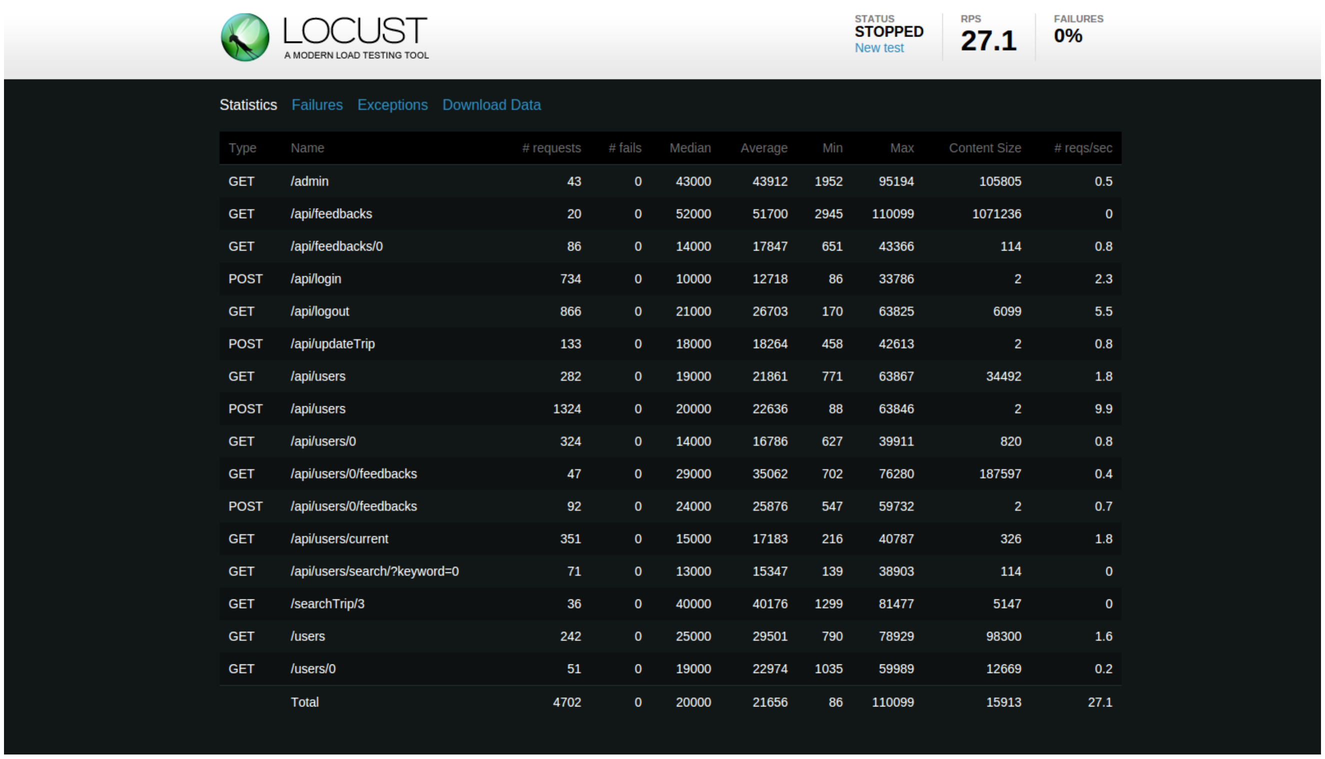
For back end optimization, we have tried to use three techniques for optimizations.

Increase parallelism, cluster for node.js, by trying to apply cluster to the server definitively will give a boost in performance since there will be multiple cores processing requests. However, we have discovered that our web page cannot simply apply the cluster package. Since, cluster is for most basic fundamental parallelism in html and js which can share same database and session. In our case, since we have used ***socket*** for real time chat, it’s not possible to use simple cluster framework to let multi threads to share and know which socket they should use. After massive research and trying, we consul the professor on this, in which we finally decide to keep the real time chat functionality to trade off for this performance improvements.

Caching, we have use caching framework in node.js to store one or two expensive data from the database for further usage.

Reduce Internal Calls, last but not least, we have modified our codes and design to reduce the number of internal http request caused by each function. Also, trying to minimize the need to access the database.

Here are the results for our final performance test:



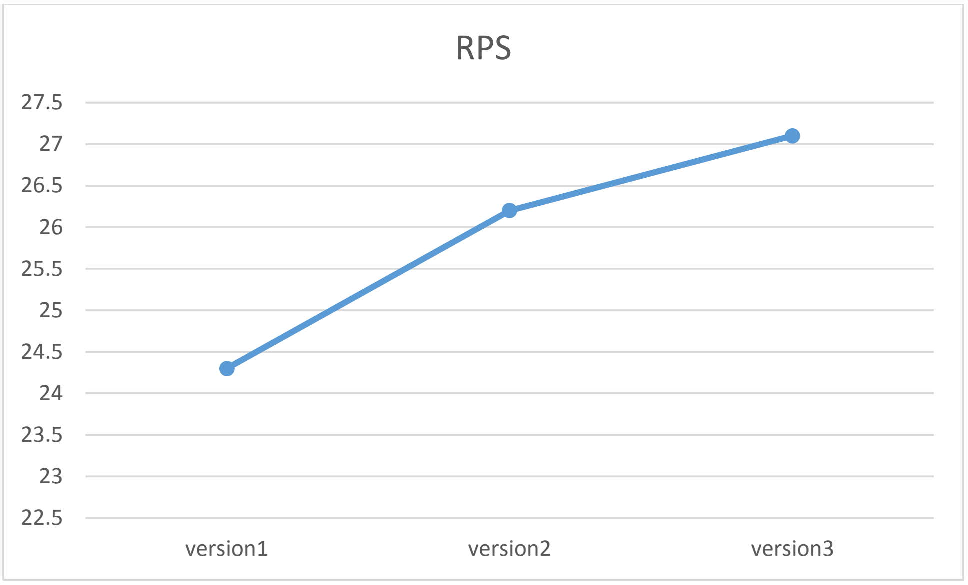
Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
GET	/admin	43	0	43000	43912	1952	95194	105805	0.5
GET	/api/feedbacks	20	0	52000	51700	2945	110099	1071236	0
GET	/api/feedbacks/0	86	0	14000	17847	651	43366	114	0.8
POST	/api/login	734	0	10000	12718	86	33786	2	2.3
GET	/api/logout	866	0	21000	26703	170	63825	6099	5.5
POST	/api/updateTrip	133	0	18000	18264	458	42613	2	0.8
GET	/api/users	282	0	19000	21861	771	63867	34492	1.8
POST	/api/users	1324	0	20000	22636	88	63846	2	9.9
GET	/api/users/0	324	0	14000	16786	627	39911	820	0.8
GET	/api/users/0/feedbacks	47	0	29000	35062	702	76280	187597	0.4
POST	/api/users/0/feedbacks	92	0	24000	25876	547	59732	2	0.7
GET	/api/users/current	351	0	15000	17183	216	40787	326	1.8
GET	/api/users/search/?keyword=0	71	0	13000	15347	139	38903	114	0
GET	/searchTrip/3	36	0	40000	40176	1299	81477	5147	0
GET	/users	242	0	25000	29501	790	78929	98300	1.6
GET	/users/0	51	0	19000	22974	1035	59989	12669	0.2
	Total	4702	0	20000	21656	86	110099	15913	27.1

As you can see, we have gained another 1 request per second increase. And this result is simulated under nearly 3000 users reported by locust. The response time increase for /admin since at the end, each /admin will pull over thousands of data from all database we have design which will cost a lot.

- **Conclusion**

Note, all detail data produced by locust has been recorded in /performance/a5PerformanceTestResult.txt in the zip file we have submitted.

Here by summarizing the request per second of all three versions:



We can see we indeed succeed in improve the massive loading performance for our website. Given more time, we may still increase more performance by using more advanced design and technologies.

Section E. YouTube Link

https://www.youtube.com/watch?v=Hpv_LEbpSdk

Section F. Architecture and Functionality Testing

System Architecture

We have chosen to use MVC architecture to construct and scaffold our website. The view part is consisting of HTML and CSS files. They collectively display our website in a fashionable way with the display HTML DOM is separate from the styles in CSS files. The model part is our database using mongoDB as storing mechanism. It provides data to our visual part of the website. We design our table schema using object-relational mapping technique. We start with Entity/Relationship schema and then transfer into logical schema so that it reduces the use of the storing space and make the access to database more efficiently. For example, we make sure each table has a unique primary id instead of the weak primary id. Finally, the control part is the different controllers such as the Account Manager Controller, Feedbacks Controller, etc. We use these controllers to control the dataflow of the server and create or delete the database. Moreover, we carefully designed our website API in the server so that it connects our MVC parts together tightly. With the help of the architecture we can easily connect an integral website that can interact with user in the front end DOM part. We use JavaScript files to handle these interactions and the injection of data into the DOM. So it can also be viewed as a three-tier architecture since we separate the database layer, data access logic layer from our front end presentation and interaction layer. In this point of view, our data access logic layer will check the conditions before we access to the database. For example, it will check whether the user is admin before it proceeds some deletion of database information. It will check the encrypted user login information to ensure that the user is in the database before proceed to access other user's private information inside the database layer.

Functionality Testing

We use the mocha testing framework to test our functionality. Before we start our implementation of our website, we created requirements for each of our functionality. This helps latter on that we can start with some basic unit tests for our very modular functions. In these unit tests we make sure that some of the corner cases and the correctness of the functions are covered. For example, we input a zero or none value into a function to see if it will catch the exception. Latter on we go to some more complicate test which will see if these small unit functions will work coherently together when they interact with each other. Last but not least we test the server but running it on the local host and then mock some user behavior with http module in node.js to access the API we designed to see if it will handle the request correctly.