

CSCC43 TERM PROJECT REPORT

Zachariah Glover

998876408

1 Project Overview

A botnet is a network of computers infected with malicious software designed to allow an attacker to remotely control the group of computers. Their use is entirely focused on orchestrating large scale digital crimes (most commonly sending spam and distributed denial of service attacks). One emerging functionality of botnets is the ability for its controller to rent out portions of the network for others to use. That is the concept that this application was built around. I've named the application it 'Hivemind.' It is a botnet management system. The application allows a botnet owner to store, analyze and manipulate data related to users, bots and rentals.

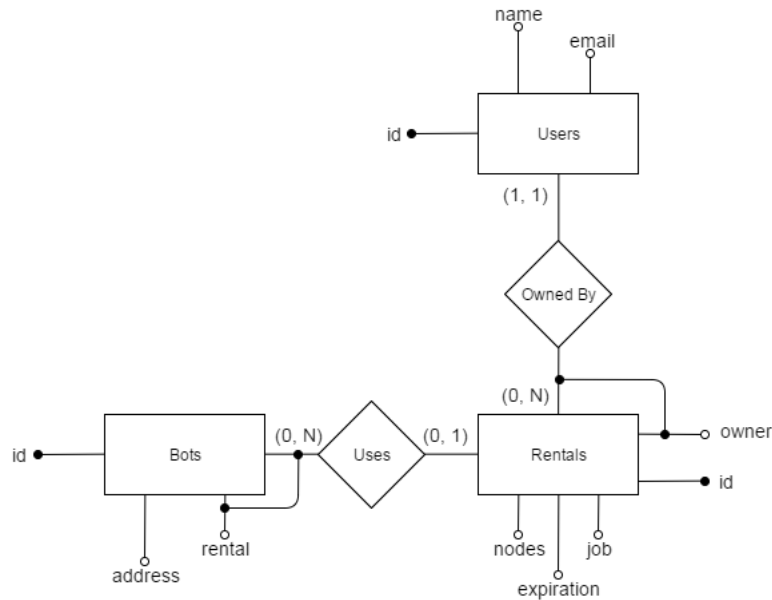
The application is a web based browser app with both client and server code.

2 Database Design

2.1 Description

The database consists of three tables - users, bots and rentals. The relationships are fairly straightforward, and there is a huge potential for implementing more functionality into the application.

2.2 E/R Diagram



2.3 Schemas and Constraints

Constraints:

users(uid, name, email)

bots(bid, address, rental)

rentals(rid, owner, nodes, job, expiration)

Functional dependencies:

$uid \rightarrow uid, name, email$

$bid \rightarrow bid, address, rental$

$rid \rightarrow rid, owner, nodes, job, expiration$

All relations are in BCNF.

Schema:

users:

uid: A unique 32 character string that represents a hexadecimal number intended to identify specific users.

name: A string that is up to 60 characters representing a non unique moniker for a user.

email: A unique string up to 60 characters that represents a user's email (can be null).

bots:

bid: A unique 32 character string that represents a hexadecimal number intended to identify specific bots.

address: A 15 character string representing the IP address of a bot.

rental: A unique 32 character string that represents a hexadecimal number intended to identify a specific rental that this bot is associated with (can be null however, if a rental is deleted the bots referencing that rental will null the rental value).

rentals:

rid: A unique 32 character string that represents a hexadecimal number intended to identify specific rentals.

owner: A unique 32 character string that represents a hexadecimal number intended to identify the specific user that owns this rental. Cannot be null, if the owner is deleted, all rentals associated with them are as well.

nodes: An integer representing the number of bots in the rental.

job: A string up to 16 characters that represents the task of the bots used by the rental.

expiration: A date representing when the rental should be terminated.

The following SQL commands are used to generate the database schema:

```
1 CREATE TABLE IF NOT EXISTS users (
2   uid CHAR(32),
3   name VARCHAR(60),
4   email VARCHAR(60) UNIQUE,
5   PRIMARY KEY (uid)
6 );
7 CREATE TABLE IF NOT EXISTS rentals (
8   rid CHAR(32),
9   owner CHAR(32),
10  job CHAR(16),
11  nodes INT,
12  expiration DATE,
13  PRIMARY KEY (rid),
14  FOREIGN KEY (owner)
15    REFERENCES users(uid)
16    ON DELETE CASCADE
17 );
18 CREATE TABLE IF NOT EXISTS bots (
19   bid CHAR(32),
20   address CHAR(15),
21   rental CHAR(32),
22   PRIMARY KEY (bid),
23   FOREIGN KEY (rental)
24     REFERENCES rentals(rid)
25     ON DELETE SET NULL
26 );
```

Here are other SQL commands that I use in the app:

```
1 /* Inserting tuples. */
2 INSERT INTO rentals (rid, owner, job, nodes, expiration) VALUES (rid, uid, "", 0, YYYY-MM-DD);
3
4 INSERT INTO bots (bid, address, rental) VALUES (bid, "", rid);
5
6 INSERT INTO users (uid, name, email) VALUES (uid, "", "");
7 /* ——— */
8
```

```

9  /* Deletion of individual tuples. */
10 DELETE
11 FROM bots
12 WHERE bid=bid;
13
14 DELETE
15 FROM users
16 WHERE uid=uid;
17
18 DELETE
19 FROM rentals
20 WHERE rid=rid;
21 /* —— */
22
23 /* Deletion of all tuples. */
24 DELETE FROM rentals;
25
26 DELETE FROM users;
27
28 DELETE FROM bots;
29 /* —— */
30
31 /* Saving tuples. */
32 UPDATE bots
33 SET address="",
34     rental=rid
35 WHERE bid=bid;
36
37 UPDATE users
38 SET name="",
39     email=""
40 WHERE uid=uid;
41
42 UPDATE rentals
43 SET owner=uid,
44     nodes=0,
45     job="",
46     expiration=YYYY-MM-DD
47 WHERE rid=rid;
48 /* —— */
49
50 /* Searching tuples. */
51 /* These are complicated and I generate them with javascript. */
52 /* Here is that. */
53
54 /* Searching for bots. */
55 var b = '(bots B) ';
56 var r = '(rentals R) ';
57 if (req.query.job) {
58     r = '((SELECT * FROM rentals E WHERE E.job LIKE\'' + req.query.job + '%\' ) AS R) ';
59     b = '(' + b + 'JOIN' + r + 'ON B.rental=R.rid) '
60 }
61 if (req.query.uid) {
62     var u = '((SELECT * FROM' + r + 'WHERE R.owner LIKE\'' + req.query.uid + '%\' ) AS U) '
63     b = '(' + b + 'JOIN' + u + 'ON B.rental=U.rid) '
64 }
65 b = 'SELECT bid, address, rental FROM ' + b;
66 /* */
67

```

```

68  /* Searching for users. */
69  var b = '(users B) ';
70  var r = ' (rentals R) ';
71  if (req.query.job) {
72      r = ' ((SELECT * FROM rentals E WHERE E.job LIKE \'%\' + req.query.job + \'%\') AS R) ';
73      b = '(' + b + 'JOIN' + r + 'ON B.uid=R.owner) '
74  }
75  if (req.query.rid) {
76      var u = ' ((SELECT * FROM' + r + 'WHERE R.rid LIKE \'%\' + req.query.rid + \'%\') AS U) '
77      b = '(' + b + 'JOIN' + u + 'ON B.uid=U.owner)'
78  }
79  b = 'SELECT uid, name, email FROM ' + b;
80  /* */
81
82  /* — */
83
84  /* Counting tuples. */
85  SELECT COUNT(uid) AS Users FROM users;
86
87  SELECT COUNT(bid) AS Bots FROM bots;
88
89  SELECT COUNT(rid) AS Rentals FROM rentals;
90  /* — */
91
92  /* Counting the number of rentals and nodes associated with every type of job. */
93  SELECT *
94  FROM (
95      (SELECT DISTINCT job AS Job,
96          COUNT(job) AS Rentals
97      FROM rentals
98      GROUP BY Job) AS R)
99  JOIN (
100      (SELECT DISTINCT job AS Job,
101          SUM(nodes) AS Nodes
102      FROM rentals
103      GROUP BY Job) AS N) ON N.Job=R.Job;
104  /* — */
105
106  /* Counting the number of rentals and nodes each user has, and display both their name and id. */
107  SELECT UserID,
108      name AS Username,
109      Rentals,
110      Nodes
111  FROM (users U)
112  JOIN (
113      (SELECT *
114      FROM ((
115          (SELECT DISTINCT OWNER AS UserID,
116              COUNT(OWNER) AS Rentals
117          FROM rentals
118          GROUP BY UserID) AS R)
119      JOIN (
120          (SELECT DISTINCT OWNER AS UserID2,
121              SUM(nodes) AS Nodes
122          FROM rentals
123          GROUP BY UserID2) AS N) ON N.UserID2=R.UserID)) AS D) ON D.UserID=U.uid;
124  /* — */

```

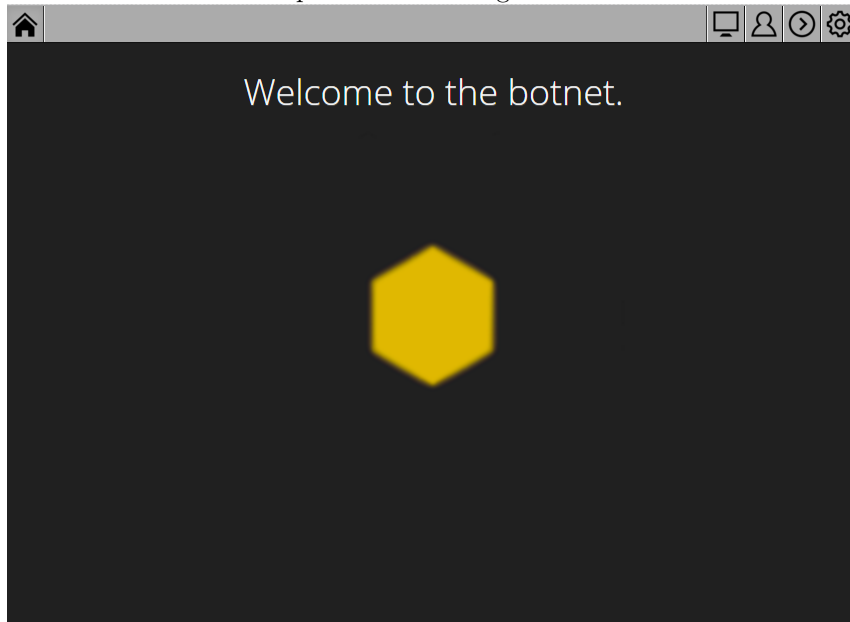
2.4 Notes

3 Application Design

3.1 Pages/Navigation

This is the home page for the application.

The header bar at the top is used for navigation.



3.1.1 Header bar

There are five buttons on the header bar.

On the far left is the home button, which navigates back to the home page.

On the far right are the remaining four buttons. From left to right these navigate to the bots, users, rentals and advanced pages.

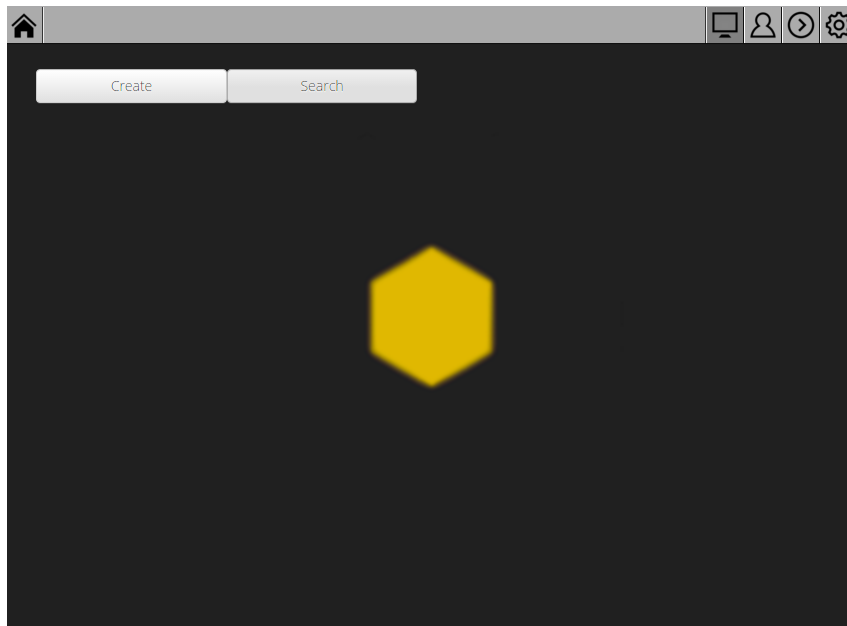


3.1.2 Bots, Users, Rentals

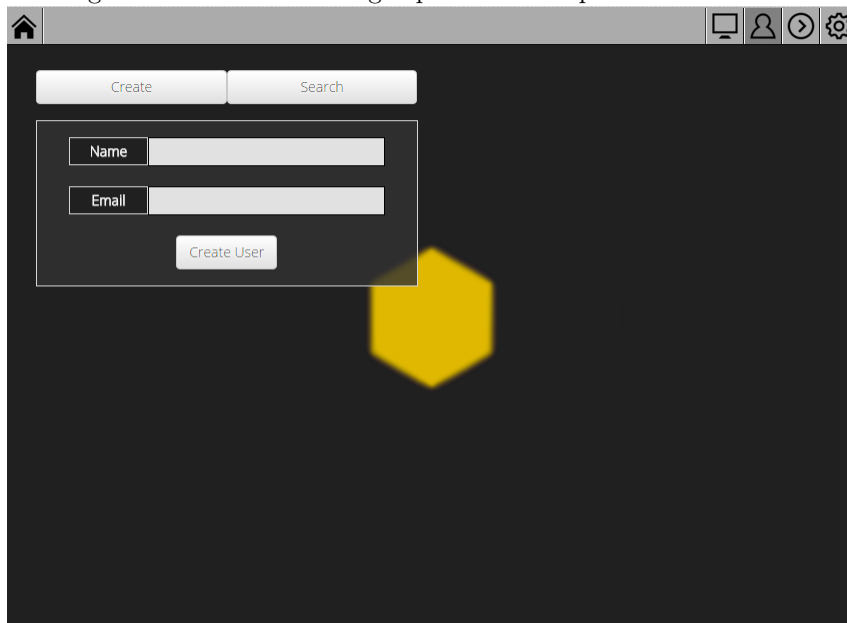
The bots, users and rentals pages are very similar.

When navigating to any of them, they all have the same default page.

This is what it looks like with empty database tables.



Clicking the create button brings up a form to input attribute values to create a tuple in a relation.



Clicking the search button brings up a form to input attribute values to filter tuples in a relation.

When a relation has tuples, these are displayed on the right.

Search will filter these displayed tuples.

More details on the specific operation of search and create are in section 3.2.2.

Each page allows for the selection of specific tuples by clicking on one of the boxes on the right. This will bring up a special modification menu which allows you to edit attributes or delete the tuple from the relation.

ID	User	Nodes	Job	Expiration
5678095ecb7e4e0aa50de5e7e8d010b4	c45471dae6ca44249d076c2e1cb479d6	4	ddos	2016-10-06
5c1d9f7bd59847ca9bbc42ba185004a2	ff8c1f8160ff4940b26c442e1b9e4af8	2	ddos	2016-06-22
6aae7256e79244deb57dfacb7948b279	a935a001294c4d11839a69df226c44f5	4	ddos	2016-08-16
9d3c2e36b972413b932bd2d595b05a4a	c45471dae6ca44249d076c2e1cb479d6	2	ddos	2016-01-15

Certain relations may have extra options besides saving and deleting when you select them. For instance, selecting a rental allows you to select the View User button, which will navigate to the users page and select that user (the form can be seen above).

Similarly, selecting a user provides 'Create Rental' and 'View Rentals.' Create rental will navigate to the rental creation page and fill in the user id for you. View rentals navigates to the rental search page, filtering only to rentals with the selected user's id (form seen below).

The screenshot shows a web application interface. At the top, there is a navigation bar with a home icon, a monitor icon, a user icon, a refresh icon, and a settings icon. Below the navigation bar, there is a main content area. On the left, there is a form with two tabs: 'Create' and 'Search'. The 'Create' tab is active. The form contains three input fields: 'ID' (with value 'a935a001294c4d11839a69df226c44f5'), 'Name' (with value 'user 1'), and 'Email' (with value 'a935a001294c4d11839a69df226c44f5@website.cc'). Below these fields are four buttons: 'Create Rental', 'View Rentals', 'Save User', and 'Delete User'. On the right, there is a list of three users, each with a dark background and white text. The first user has ID 'a935a001294c4d11839a69df226c44f5', Name 'user 1', and Email 'a935a001294c4d11839a69df226c44f5@website.com'. The second user has ID 'c45471dae6ca44249d076c2e1cb479d6', Name 'user 0', and Email 'c45471dae6ca44249d076c2e1cb479d6@website.com'. The third user has ID 'ff8c1f8160ff4940b26c442e1b9e4af8', Name 'user 2', and Email 'ff8c1f8160ff4940b26c442e1b9e4af8@website.com'.

3.1.3 Advanced

The advanced page provides customized query functionality.

A query can be made by filling in the query text box and clicking submit. The result of the query will appear in a table below. Clicking clear will clear the text box and the result table.

There are some buttons that will generate a query for you to perform certain tasks (submit must still be clicked to send the query).

Create tables will create the users, bots and rentals table if they do not exist. Delete tables will destroy the users, bots and rentals table (they must be created again after this to be used).

Generate data will generate some randomized but valid tuples in all the tables. You can use this multiple times to create large amounts of test data. Delete data will generate a query to delete all tuples in each table.

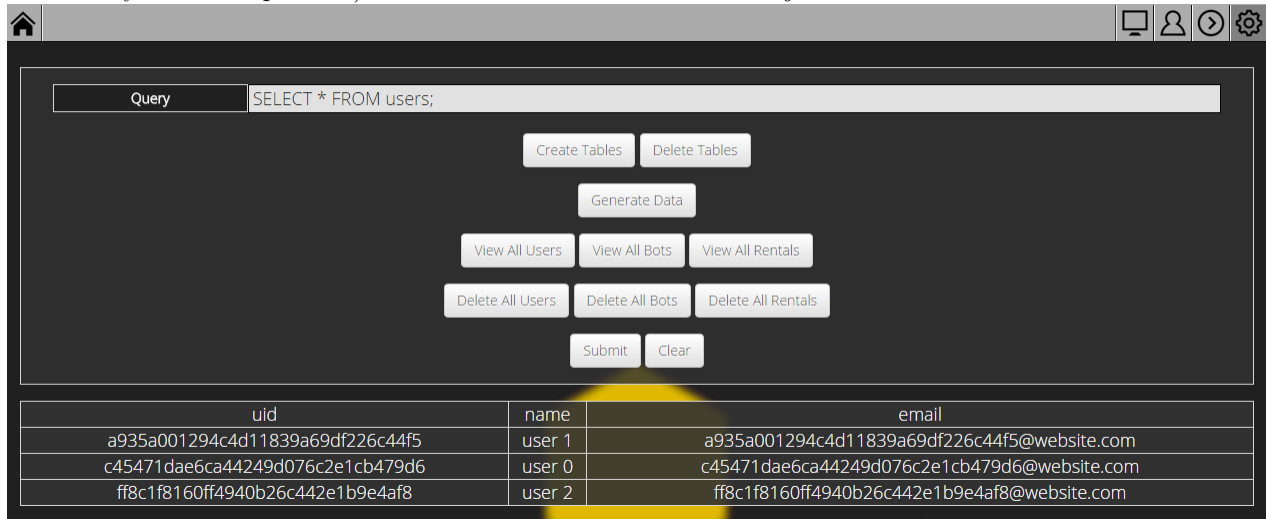
The view buttons allow for quick viewing of the regular tables.

The delete all buttons allow you to quickly purge all tuples from a table.

The screenshot shows a web application interface. At the top, there is a navigation bar with a home icon, a monitor icon, a user icon, a refresh icon, and a settings icon. Below the navigation bar, there is a main content area. On the left, there is a form with a 'Query' text box. Below the text box are two buttons: 'Submit' and 'Clear'. To the right of the 'Query' text box, there are several buttons arranged in a grid. The first row contains 'Create Tables', 'Delete Tables', 'View All Users', 'View All Bots', and 'View All Rentals'. The second row contains 'Generate Data', 'Delete Data', 'Count Users', 'Count Bots', 'Count Rentals', and 'Count All'. The third row contains 'Delete All Users', 'Delete All Bots', 'Delete All Rentals', 'Count User Rentals/Nodes', and 'Count Job Rentals/Nodes'.

Here is an example of what happens if you type `SELECT * FROM users;` (or click view all users, which au-

tomatically fills the input box) and then click submit. The data is just some randomized data.



uid	name	email
a935a001294c4d11839a69df226c44f5	user 1	a935a001294c4d11839a69df226c44f5@website.com
c45471dae6ca44249d076c2e1cb479d6	user 0	c45471dae6ca44249d076c2e1cb479d6@website.com
ff8c1f8160ff4940b26c442e1b9e4af8	user 2	ff8c1f8160ff4940b26c442e1b9e4af8@website.com

3.1.4 Notes

I will provide more details on setting up the app in the readme file.

3.2 Client Side Operation

3.2.1 Design

The client side functions by using a monumental amount of javascript. I use nodejs, backbone, jquery and underscore extensively. Further bootstrap and some custom css is used to provide style sheets.

The webpage uses backbone routing, jquery and underscore to navigate around the app by generating and injecting html into the page.

Backbone is also used to easily store and parse database information and to generate http requests to the server when working with data from the database.

3.2.2 Create/Search

Both create/search functions of the pages generate attributes through user input forms, which are sent to the server, which then generates a query.

Create simply takes user input and generates an instance of the tuple using the supplied data. There isn't any sanitization of user input (besides some html constraints and placeholders to suggest the correct form). Invalid entries simply provide an error alert. Empty entries use a default value.

Search is more complex. It uses a combination of SQL queries and underscore filtering. Simple filters, that

are available immediately from a tuple (such as filtering rentals by job type) are done client side with underscore. More complex searches such as searching for users that have rentals of a specific job type (ie searching by job type on the user page) will send the attributes to the server, which then generates an appropriate SQL query and returns the result. Note that partial entries into search forms work (ie you can search for part of a userID and it returns all related results containing that part).

3.2.3 Advanced

The advanced page literally takes the text in the query form and sends it as an SQL query to be executed by the server.

The buttons on the page (besides submit and clear) simply insert a hardcoded query into the form.

Results from the server are parsed into a table and displayed, certain actions don't necessarily return meaningful responses from the sql databases (INSERT, UPDATE) so there is just a success message displayed (if it was a success). Errors display an error message.

Queries can be chained and the table of results are horizontally appended.

I have hardcoded several interesting queries for ease of use.

3.3 Server Side Operation

3.3.1 Design

The server is implemented using express to create an http server, and a node module for mysql to interact with the mysql database.

Requests are received and processed by server.js and routes.js.

3.4 Directory/File Descriptions

- /app - This is the root directory.
 - /node_modules - Directory containing node modules used by the server.
 - /body-parser - Used to simplify client requests to the server.
 - /express - Used to run http server.
 - /mysql - Used to interact with mysql server.
 - /uuid - Used to generate unique IDs for tuples.
 - /public - Directory containing the client side portion of the app.
 - /css - Contains styles sheets.
 - bootstrap.css - Contains default bootstrap styles.

- bootstrap.css - More bootstrap styles.
- styles.css - My own modifications on style sheets.
- /img - Contains images used by the client.
- /js - Contains javascript files to run the client side app.
 - /collections - Contains backbone collection definitions.
 - bots.js - Bot collection.
 - rentals.js - Rental collection.
 - users.js - User collection.
 - /lib - Contains javascript libraries that I used.
 - backbone-min.js - Backbone
 - jquery-min.js - JQuery
 - underscore-min.js - Underscore
 - /models - Contains backbone model definitions.
 - bots.js - Bot model.
 - rentals.js - Rental model.
 - users.js - User model.
 - /views - Contains backbone view definitions.
 - advanced.js - Javascript used to run the advanced page.
 - botinfo.js - Javascript used to generate the bot info markup.
 - header.js - Javascript used to generate and run the header.
 - home.js - Javascript used to generate the home page.
 - managebots.js - Javascript used to generate and run the manage bots page.
 - managerentals.js - Javascript used to generate and run the manage rentals page.
 - manageusers.js - Javascript used to generate and run the manage users page.
 - rentalinfo.js - Javascript used to generate the rental info markup.
 - userinfo.js - Javascript used to generate the user info markup.
- main.js - The router definition. Allows for page navigation and the generation of client http requests to the server.
- /tpl - Contains html templates for all the pages.
 - advanced.html - Markup for corresponding page.
 - botinfo.html - Markup for info sections to be duplicated and injected into pages.
 - header.html - Markup for corresponding page.
 - home.html - Markup for corresponding page.
 - managebots.html - Markup for corresponding page.
 - managerentals.html - Markup for corresponding page.
 - manageusers.html - Markup for corresponding page.

- rentalinfo.html - Markup for info sections to be duplicated and injected into pages.
- userinfo.html - Markup for info sections to be duplicated and injected into pages.
- index.html - The main page. Loads everything for the client, and provides a space to have page templates injected.
- config.js - Configuration options for the server and database.
- routes.js - Contains functions used for generating SQL queries and other server responses.
- server.js - Server code, sets up server options and defines routes to be used based on client requests.