

CSC309  
Development on the Web

Assignment 5  
Web Application on Sharing Economy

Title  
DogTaker



Instructor  
Ahmed Mashiyat

Team Members:

Parastoo Abtahi	999036617	c5abtahi
Jonghyo Kim	998960824	c4kimjon
Douglas Chong	998892756	c4chongd
Shubham Dhond	1000982715	g5dhonds

2015 - 11 - 19

## Description

DogTaker is a Sharing Economy web application that aims to connect dog owners with volunteers who are interested in taking care of a dog (dog lovers). Users can sign up for free and indicated whether they are a dog lover, dog owner, or both. Once logged in, users need to indicate their location and update other information on their profile.

The system enables dog owners to find a dog lover to take care of their dog, for a few hours or even days, while they are away. They can do so by posting a request on their profile and allowing any dog lover to respond to the public post. After the request is sent, the dog lover has the opportunity to accept the request, depending on their availability during the requested time. If the dog lover accepts the request, the dog owner can contact them directly in order to arrange the time and place.

After the process, the dog owner will have the opportunity to rank the dog lover and leave a comment for future dog owners who might be interested in contacting that person. The dog lover will also have the chance to rank the dog lover and comment on their experience with the dog. The website is hosted for free on Heroku at the URL <http://arcane-atoll-3703.herokuapp.com/>.

## Detailed Design

This web application consists of various components such as the welcome page, login page, registration page, main home page, and user profile page. Each page is described in detail below.

### A. Welcome Page

This page includes the logo of the application and a brief description of the purpose of DogTaker. Users who already have an account can click the “Login” button to be directed to the login page. New users can click the “Sign up” button to create a new account.

### B. Login Page

The login page allows the users who already have an account to enter their email and password. New users will also be able to login on this page using their Facebook account. An error will be prompted upon submission if the user and password provided does not correspond to a user in the database.

### C. Sign Up Page

This page allows new users to create an account in order to login to the site. The users are asked to enter their email which is necessary for communication among dog lovers and dog owners, after a request has been confirmed. Location is also a required field for registration, because it is used by dog owners to search for a volunteer in their neighbourhood. Various security measures have been implemented for this page to eliminate any vulnerabilities, which are described in the security section.

### D. Main Home Page

The main home page includes a list of all DogTaker users, with each email linked to that person's profile page. The map in the home page shows all user locations as pins. The map is zoomed in on the area where the current user resides. This feature allows a dog owner to be able to quickly look through all the dog lovers living in their neighborhood. Below the map, there is a request form that allows the user to send a public request to all the other users, with a specific start and end date. In this page, the user can also search based on username or location. Clicking the logo at the left hand side of any page within this web application brings back the user to this page.

### E. User Profile Page

On the left hand side, the user name and profile picture are displayed. Below the profile picture, is a list of all requests made by that user and the status of each request. A list of all comments and ratings made regarding that user can be found in the middle section of the page. The user name, email, and description are also shown on the right section of the page.

### F. Profile Edit Page

If the user has administrative privileges or they would like to edit their own profile information, they can use the "edit" button in the information box on the right hand side of the User Profile Page. This will guide the user into a different UI in which they will be able to modify the information.

The pages described above encapsulate a set of features and UI elements such as admin, user authentication, profiling, social network, search and recommendation. These features are described in detail below:

### A. Administrative Privileges

The first user created in the database will be the super administrative user. This user is able to assign administrative privileges to other users, as well as delete and edit other users. An administrator account has been created with email [admin@admin.com](mailto:admin@admin.com) and password Admin22@.

## B. User Authentication

Users can login by either creating a new account in DogTaker or using their facebook login. This features makes it easy for first-time users to try out our web application without having to go through the sign-up process. Since not all Facebook users have specified their location in their Facebook profile, this information is not being used by our application. Therefore, users will be prompted in order to allow our application to detect their location. Various security measures have been taken into account for the user authentication process, as described in the security section.

## C. Profiling

### I. Profile Image

The profile image is a key feature in our application. This allows dog owners to upload an image of their dog on to their profile, which will attract dog lovers who are interested in that particular breed or type of dog. In the case where the user would not like to upload an image, the default will be used.

### II. Profile Information

The profile information includes the following information: username, email, description. User email is a required field as it is a primary source of contact between dog lovers and dog owners once a request has been accepted.

### III. Map View

The map view on the home page of the application displays the locations of all users. This was implemented using the google maps api and the places library. For each user we allow them to input a location on registration which is then geocoded into latitude and longitude values which are stored in the database. These values are used to create the markers on the map.

## D. Social Network

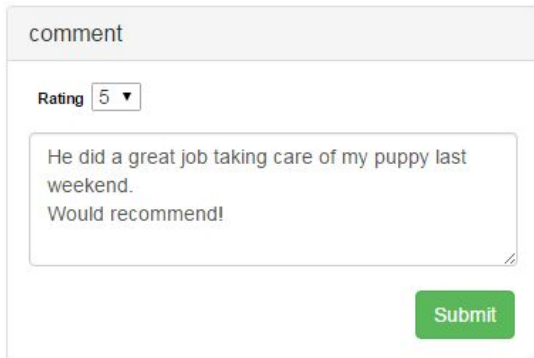
### I. Sending a Request

Users can send a request for a dog lover at specified date. This request will appear on their profile page and dog lovers can choose to accept this request. After the request is accepted,

the request will show as accepted on the user's profile page and show by which user it was accepted by.

## II. Rate and Comment

The rating and comment system allows each user to rate others on our website.



Each user can view others' profile pages, and leave a comment and a rating under the chosen user. Any user can view the comments and rating information associated with each user that are listed under the figure shown above.

## E. Search and Recommendation

In order to implement an efficient search engine for our user database, we decided to make use of the *search-index* module.

The search bar on the top of the page consists of the search field and the search term. The user can choose which search field to search on from the drop down, then enter the search term to search the index for the entered search term.

The search term entered must be at least two characters long, and it will search for terms beginning with the entered search term. As for the location field, it must be searched using the 'Location' dropdown menu as it uses a different route to search the index. Similarly to the Location field in the Register page, it autofills the location that is recognized by Google Maps. Once the query is made, it will return all users within the 100km radius of the searched location. The search functionalities are encapsulated in the routes `/searchresults` and `/searchlocation`.

## Security

Multiple security features were implemented in this project to account for vulnerabilities.

### A. Password restrictions

The first security feature we decided to implement was to set restrictions on the password for the users. These restrictions will prevent users from choosing a short or trivial passwords that are prone to dictionary brute force attacks. On the registration page of the website, we have implemented frontend checks that validate these requirements.

The characteristics of a good password required by our website:

- be at least 8 characters long
- contain at least one number
- contain at least one uppercase letter
- contain at least one lowercase letter
- not contain the username

We have added the following code to *global.js*, to check the above requirements upon the submission of the registration page:

```
....
    if(password.val().length < 8){
        registerErrors.text("Your password must be at least 8 characters long.");
        registerErrors.show();
        return;
    } else if(!hasAt.test($('#username').val())) {
        registerErrors.text("Not a valid email. Must have @");
        registerErrors.show();
        return;
    } else if(password.val().indexOf($('#username').val())>-1){
        registerErrors.text("Your password may not contain your username to better secure your account.");
        registerErrors.show();
        return;
    } else if(!hasNumber.test(password.val())){
        registerErrors.text("Your password must contain at least one number.");
        registerErrors.show();
        return;
    } else if(!hasCapital.test(password.val())){
        registerErrors.text("Your password must contain at least one capital letter.");
        registerErrors.show();
        return;
    } else if(!hasLower.test(password.val())){
        registerErrors.text("Your password must contain at least one lowercase letter.");
        registerErrors.show();
        return;
    }
}
....
```

## B. Password Hash Algorithm

The other security prevention we used is the password hash/salt algorithm within the passport-local-mongoose module. This prevents any hackers from accessing user passwords as they will not be stored in plain-text in the database. When a request is sent from the frontend user, in order to register, *setPassword* is called with the user password, as shown below:

```
schema.methods.setPassword = function (password, cb) {  
  if (!password) {  
    return cb(new BadRequestError(options.missingPasswordError));  
  }  
  var self = this;  
  
  options.passwordValidator(password, function(err) {  
    if (err) {  
      return cb(err);  
    }  
    crypto.randomBytes(options.saltlen, function(err, buf) {  
      if (err) {  
        return cb(err);  
      }  
  
      var salt = buf.toString(options.encoding);  
      pbkdf2(password, salt, function(err, hashRaw) {  
        if (err) {  
          return cb(err);  
        }  
        self.set(options.hashField, new Buffer(hashRaw, 'binary').toString(options.encoding));  
        self.set(options.saltField, salt);  
        cb(null, self);  
      });  
    });  
  });  
};
```

Then this method generates a salt of length *options.saltlen*, using the crypto module, and passes it onto the *pbkdf2* function which generates a hash according to the salt given.

```
options.saltlen = options.saltlen || 32;  
options.iterations = options.iterations || 25000;  
options.keylen = options.keylen || 512;
```

```

...

var pbkdf2 = function(password, salt, callback){
  if(crypto.pbkdf2.length >= 6){
    crypto.pbkdf2(password, salt, options.iterations, options.keylen, options.digestAlgorithm, callback);
  } else {
    crypto.pbkdf2(password, salt, options.iterations, options.keylen, callback);
  }
};

```

The saltlen, iterations, and keylen options mentioned above can be increased in order to make it more difficult to decrypt the user passwords. Here is an example of the data generated in mongodb after the functions execute:

```

    "salt" : "a5f8c659a1158e4c9196ba4b6973ec413f220028ea94fd8f382f8fee44d23a2b",
    "hash" : "910ee92595528ca23237283a53f581c910baa7c3835dd087be0d697b11286b
240fd1433b5a563ecda6a9bf1e443a8eb1741561554ebdb21b98305fce0890426a3a821541d75bf825775a4c7
d6d4fa734c5156f934c8a463542488b1905e09af76dc971fb0aeb3f17cae31708658b2674e3f839968ed93fea4
ac7d5fedde05ce9b18e08657bf474938a7985f89199964313af21f348a0b89ab7b57e6f5c8eab90189f131a404
918227b934229ddac6f2120183bdbcbde1cd672f34588c5c60d36a0b3fe08f87c715e3e8b37d1bf18193a862b
4e8358319c97b437060c93dd95e1184b6cecee0ebb89a650ed1d62a0c8b754d0ace1988f0dc24e37a9efec51b
9de09e3ca2e4cbc1b0953d7a8ea72c9aaa873bd342e65f8f12a137f159aa6803e060fb1caa2039fdf900df7a536
c13b87446748ccce46d4cd78c1ef90b0883520eb60821dd7b291c542fd22718720449aaddcd9a6c1fdace86bf
80fe6fce00a679b61ace4d97fe83c9c5caa8e90bf23949d91c2c0b8deb8f834e371c815316455be5e504665b4c8
ce86a3ec34ff5eb5a426b9ec3ebfab57a837445e14abddb05d14c39aad66b56a593354284022d9d2444e10560
8c159259b6acf80dbc83b851013101cc87331cdd483b623ad6c918ff97f2250b1cb73fa2fa4e058724a412bcd5
e0a96c14264f123451eb0afeb365518da7ec50a25916f99eae8c17860110b2b3f",
    "username" : "testaccount@gmail.com",

```

## Performance

Various optimizations were implemented to improve the performance of the application.

### A. Optimizations

The first performance improvement we made to our application was to implement the use of minified CSS and JavaScript files. To minify the CSS and JavaScript files we used the grunt-contrib-uglify and grunt-contrib-cssmin node modules to create grunt tasks which minify the public CSS and JavaScript files used by our application. The gruntfile.js file is shown below:

```

grunt.initConfig({
  pkg: grunt.file.readJSON('package.json'),

```



```

    uglify: {
      development: {
        options: {
          paths: ['public/javascripts/']
        },
        files: {
          'public/javascripts/global.min.js': 'public/javascripts/global.js',
          'public/javascripts/locationField.min.js': 'public/javascripts/locationField.js',
          'public/javascripts/map.min.js': 'public/javascripts/map.js'
        }
      }
    },
    cssmin: {
      development: {
        options: {
          paths: ['public/stylesheets/']
        },
        files: {
          'public/stylesheets/style.min.css': 'public/stylesheets/style.css'
        }
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-cssmin');

```

The next performance optimization we made was the use of GZIP to compress the HTTP payloads of our server responses which are decompressed in the browser. To implement the GZIP compression we used the compression node module. The code snippet which configures the application to use this module is shown below.

```

...

var compression = require('compression');
app.use(compression);

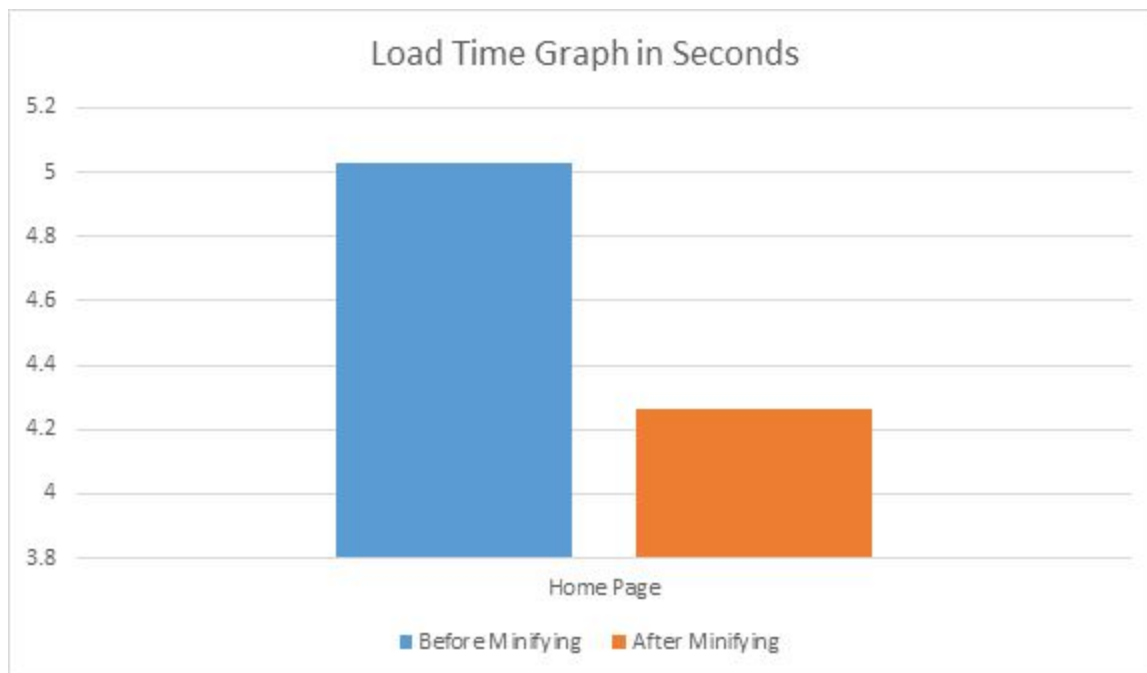
...

```

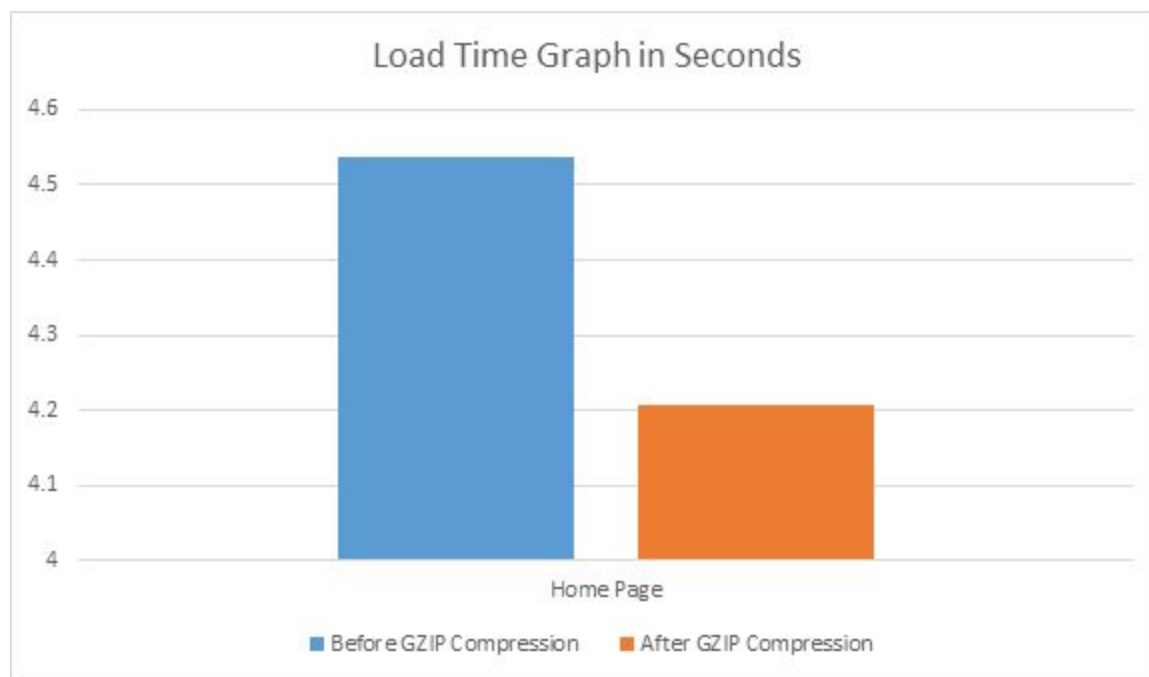
## B. Improvement

To test the improvement of our optimizations made above we used the load times of the landing page after user login as this is the page where every user will visit. The load times were obtained using the google chrome developer tools.

Before minifying the JS and CSS files.	After minifying-d of JS and CSS files.
4.97 s	3.80 s
5.14 s	4.16 s
4.94 s	4.61 s
5.00 s	4.44 s
5.09 s	4.31 s
Average : 5.028 s	Average: 4.264 s



Before implementing GZIP compression.	After implementing GZIP compression.
4.80 s	3.79 s
4.70 s	4.39 s
4.67 s	4.55 s
4.11 s	4.40 s
3.41 s	3.91 s
Average : 4.538 s	Average: 4.208 s



## Video

You can view our YouTube video here:

<https://www.youtube.com/watch?v=XsmVQRWjni0>