Technologies for Web and Social Media - Hamsterchan

Daniel Kartin

Introduction

This website was built using EmberJS, and the accompanying server was built using node.js with express,. The database is a noSQL mongoDB cluster hosted on atlas. There is also a dash of ajax, for making the *POST* request.

I will not put any pictures of the website in this paper, as one can simply visit the links below to see it live, or download the source code from the supplied github repositories.

Website links

When loading the client site for the first time, the server will take quite a while before responding, as it is sleeping, and when it does load, it should reloaded after a few seconds to make sure that the server has been woken up as well.

The client website can be seen at https://web-miniproject-client.herokuapp.com. The server site can be seen at https://web-miniproject-server.herokuapp.com/api/posts.

Repository links

The github repo for the server can be found at https://github.com/totalfreak/web-miniproject-server.

While the github repo for the client can be found at The github repo for the server can be found at https://github.com/totalfreak/web-miniproject-client.

Emberjs

Emberjs is a javascript framework much akin to angular and react, but with a much better looking mascot:

Figure 1: The Emberjs mascot, Tomster

Mostly I went with Emberjs instead of Angular due to the documentation, syntax, and ease of use.

Purpose

The purpose of this webpage is for allowing people who share a passion for hamsters, to with a title, image, and description take part in the hamster culture. I initially begun work on implementing comments, but after much fiddling couldn't get it to work with the database schema. The web page allows for anonymous posting of content, hence the content might be disturbing if let run wild, some forms of moderation would be needed, if anyone but Georgios got into the website.

The Technologies

Starting off with the frontend, Emberjs makes it quick to create something tangible, as it creates routing automatically, and has many addons that enables functionality like sass, scss, or basically any other library or technology you could think of. Emberjs pages are consisting of templates that are written in the hbs (handlebars) format, and accompanying component code, models and routes.

For creating the main page, where the newest 50 posts are being displayed, a sort of for loop was created, where we iterate over each post in the data, from ember.data. And creates the post taking in the `post.id`, `post.image`, `post.title`, and `post.text` to fill in each post with content.

```
{{#each this.model as |post|}}
     <div class="post_container">
         <p class="post_id">Id: {{post.id}}</p>
         <img src="{{post.image}}" class="post_image"
         style='max-width: 200px; max-height: 200px;'>
         h1 class="post_title">{{post.title}}</h1>
         <h3 class="post_text">{{post.text}}</h3>
     </div>
```

And ember even allows for else statements, where if no posts were found, it shows an empty post, urging the user to create the first post.

```
{{else}}
 <div class="post_container">
     <img src="image url" class="post_image"
     style='max-width: 200px;max-height: 200px;'>
     <h1 class="post_title">No posts have been made yet</h1>
     <h3 class="post_text">Hurry up and create the first post</h3>
 </div>
{{/each}}
```

So far it has only been about showing the content of the database on the frontend, but how does one make a new post, you might ask. Why, with the new post button of course. A simple form, saving the values in their desired variables, to be used in the ajax request.

```
Title: {{input value=title}} <br>
Image url: {{input value=image}} <br>
Text: {{textarea value=text}} <br>
<input type="submit" {{action "sendRequest" title image text}}>
```

The actual ajax request here, with the title, image, and text from the new post button template.

```
if(title.length > 5 && text.length > 20) {
    this.get('ajax').request('https://web-miniproject-
 server.herokuapp.com/api/posts', {
        method: 'POST',
        data: {
        post: {
            title: title,
            image: image,
            text: text
         }
     }
 });
```

On the server side, the nodejs server is equipped with express for the app, mongoose for mongodb, and body parser for json, with which I made the post schema, and allowed it to save the incoming data as posts, and send back the newest 50 posts.

The function called on the *Post* schema, gets every post in the database, limits it to 50 and sorts it in reverse, so that the newest post would be the first element in the returned array.

```
Post.find(function(err, posts) {
    if (err) {
        res.send(err);
    }
    res.json({posts: posts});
}).limit(50).sort({_id:-1});
```

When creating a new post, the code is essentially the same, but with a `post.save` instead of `Post.`find, and no limit and sorting.

When navigating to https://web-miniproject-server.herokuapp.com/api/posts, the server hook returns those 50 posts mentioned before, and can be seen in neat `json` format. Just remember that the server might be sleeping.

The server as mentioned uses express to run and listen on the specified port, being 4500, and brings in body parser to enable for json encoding and decoding. This mean that everything the server is, is tied to being an express app, bundling in CORS headers, body parser and even setting up the route.

Back to the client though, when it receives the json from the get request, it has to use it inside ember. Ember has something called emberData, which is their way of handling database management and plugging it into the their own way of working. `DS.Model` is the imported emberData schema handler. Here it basically recreates the schema from the server, with the addition of line 5, where work had begun on getting comments into database.

```
1. application.Post = DS.Model.extend({
2.   title: DS.attr('string'),
3.   image: DS.attr('string'),
4.   text: DS.attr('string'),
5.   comments: DS.hasMany('comment')
6. });
```