Jonathan McCaffrey

CSCI 380

December 6, 2018

Writeup for Week 15 Assignment

**Current project hosted URL:** http://liontracks.teammccaffrey.com/

**Code hosted on GitHub:** <https://github.com/jmccaffrey42/csci380-project>

*This is the same writeup as week 14, however I’ve updated it and added a few new sections*

# Overview

For my project I am building a basic task management system called “Lion Tracks”. The architecture for this system is as follows:

Back-end Application

PHP

Server

Browser

Front-end Application

JavaScript/HTML/CSS

Project Database

MySQL

There is a web frontend application written almost entirely in JavaScript using a few libraries such as ReactJs and jQuery. This application creates DOM elements as necessary to render an interactive user interface allowing someone login, navigate to the board, manipulate lists of cards, and update cards in detail. This front-end application then talks to the server over HTTP using REST verbs such as GET, POST, DELETE and PUT to transfer state contained in objects encoded in JSON.

# Backend Service

The PHP service supports the following calls:

* **Card Lists**
  + GET /api/lists – Retrieves card lists
  + POST /api/lists – Create a new list
  + DELETE /api/lists/<id> – Delete a list
* **Cards**
  + GET /api/cards/<id> – Retrieve details about a card
  + POST /api/cards – Create a new card
  + PUT /api/cards – Update a card
  + DELETE /api/cards/<id> – Delete a card
* **Comments**
  + POST /api/comments – Create a comment
  + PUT /api/comments/<id> – Update a comment
  + DELETE /api/comments/<id> – Delete a comment
* **Auth**
  + POST /api/auth/login – Login
  + POST /api/auth/register – Create an account
* **Users**
  + GET /api/users – Retrieve all users
  + GET /api/users/<id> – Retrieve users by id
  + PUT /api/users/<id> – Update user

Calling the backend service with an HTTP client looks like this:

>> REQUEST

POST http://localhost:8000/api/comments

Content-Type: application/json

Accept: application/json

Cache-Control: no-cache

Authorization: 2e9feefe-de4f-4ce6-a49b-6feaac198cb2

{

"card\_id": "b2d5ade3-d94f-4fca-93f0-2fbe54eb5194",

"body": "Occaecati mollitia quia eligendi totam. Ut sunt officiis eum deserunt quaerat accusamus ad. Illum et dolorum sed debitis."

}

<< RESPONSE

HTTP/1.1 201 Created

Host: localhost:8000

Date: Sun, 09 Dec 2018 05:42:05 +0000

Connection: close

X-Powered-By: PHP/7.2.12

Cache-Control: no-cache, private

Date: Sun, 09 Dec 2018 05:42:05 GMT

Content-Type: application/json

X-RateLimit-Limit: 60

X-RateLimit-Remaining: 59

{

"body": "Occaecati mollitia quia eligendi totam. Ut sunt officiis eum deserunt quaerat accusamus ad. Illum et dolorum sed debitis.",

"card\_id": "b2d5ade3-d94f-4fca-93f0-2fbe54eb5194",

"user\_id": "2e9feefe-de4f-4ce6-a49b-6feaac198cb2",

"id": "aa62505f-738b-41a8-b4f5-562bdfa6addb",

"updated\_at": "2018-12-09 05:42:05",

"created\_at": "2018-12-09 05:42:05",

"user": {

"id": "2e9feefe-de4f-4ce6-a49b-6feaac198cb2",

"created\_at": "2018-11-26 03:01:28",

"updated\_at": "2018-11-26 03:01:43",

"email": "jmc@test2.com",

"name": "Jonathan McCaffrey"

}

}

The first thing the server does when receiving a request is check for an Authorization header. If this header is present, the server will lookup the associated user with the auth token provided and, if that user exists, set a request scoped variable with that user’s information. This allows code which runs later in the request lifecycle to call Auth::user() to see who is making the request. This is used when comments and cards are being created to ensure that the card owner is set correctly.

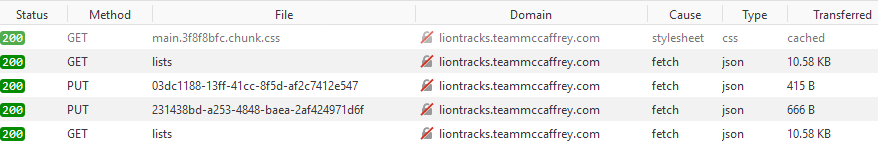
After auth, the request is passed on to [controller code](https://github.com/jmccaffrey42/csci380-project/blob/master/project-backend/app/Http/Controllers/CommentController.php) which pulls the relevant data from [model classes](https://github.com/jmccaffrey42/csci380-project/blob/master/project-backend/app/Comment.php) and executes the workflow of, for example, creating a new comment. These model classes issue a SELECT statement (or other appropriate query) to MySQL which collects the correct row in the table corresponding to the parameters given. If that row is found, it returns it to the PHP application which, in turn, encodes it in JSON and sends it back to the client as the HTTP response. The client interprets a “200” response code here as a correct login since this system doesn’t require a password.

# Frontend Client

The frontend client is a JavaScript application written using ReactJS and the Single Page Application (SPA) pattern. In this pattern, the client launches once per user session and all “pages” are managed within that single running instance. The client doesn’t refresh when you click to a new screen, and instead manipulates the DOM by creating and deleting elements on the fly. This allows for a more desktop-like experience and is ideal for websites which serve the function of full applications.

One major departure in this architecture is how sessions are handled. In this design, the backend does not keep track of session state in the traditional way but instead provides an API token mechanism to the client. The client issues a login call (POST /auth/login) and gets a token back. The client stores this token in [HTML5 Local Storage](https://www.w3schools.com/html/html5_webstorage.asp) so that you don’t have to re-login every page view. The client sets an Authorization HTTP header on API requests using this token, and the backend simply validates it before allowing the call to proceed.

As you click around and edit things on the page, you can see in your browser’s development console (under Network) that REST transactions are occurring to keep the server up-to-date with what you’re doing.



In the past we might have used the XMLHttpRequest API (also known as XHR) to handle these types of calls but pretty much all modern browsers (Edge, Chrome, Firefox, Safari, etc…) support the much improved [fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) which is what I’m using here. With that API we have full functionality to send PUT, POST, GET, DELETE and so on with any headers and body we want.

Navigating to different pages is handled via a routing system which reads the browser location using the [HTML5 History API](https://developer.mozilla.org/en-US/docs/Web/API/History_API) and determines which screen needs to be displayed. When a new screen is navigated to, the history API is used again to update the browser’s location to reflect what is being displayed.

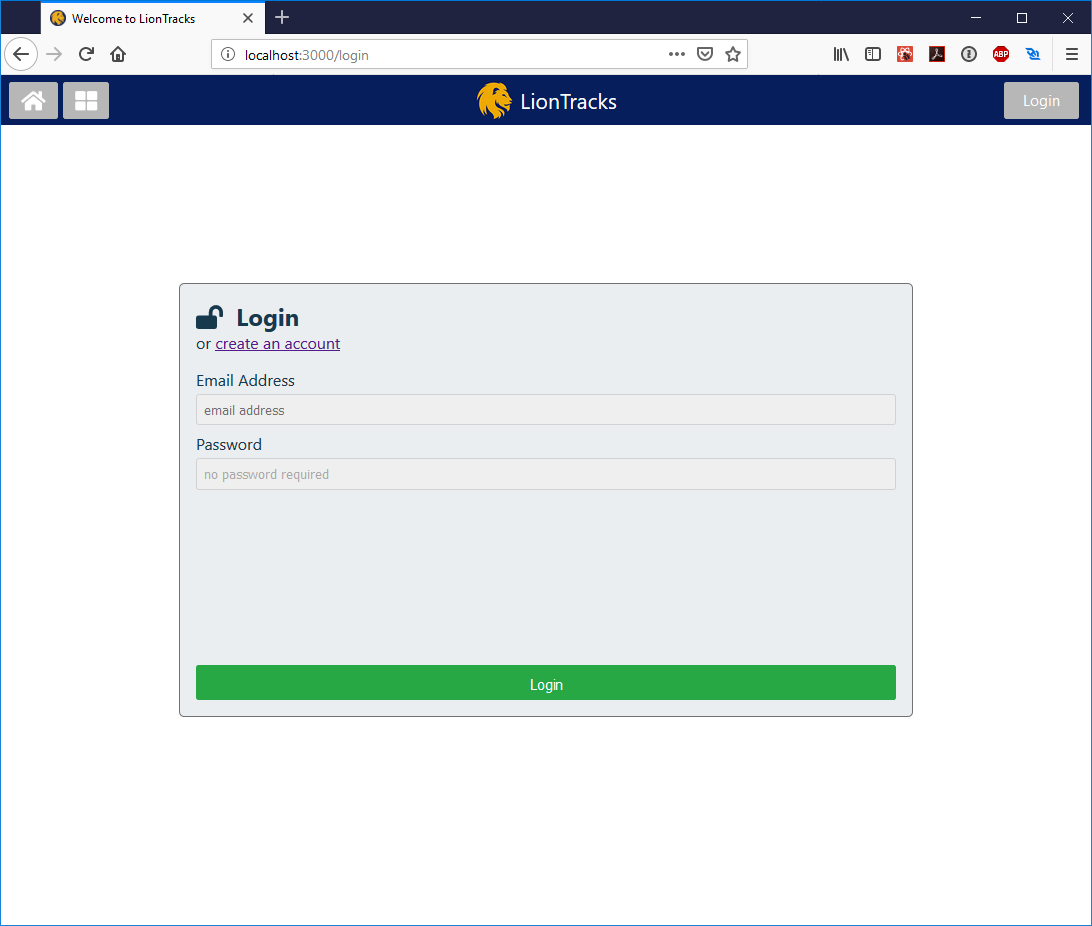
The frontend contains the following routes:

* / - The home page, not yet styled but will contain an overview of the LionTracks system
* /login – The Login form with a link to the registration page
* /register – The registration form to create a new account
* /board – The interactive task board. This board has lists which contain cards. You can drag and drop the cards and lists to reorder them and move the cards between lists. Clicking on a title lets you edit it and clicking on a card takes you to a detail page where it can be edited or commented on.

One interesting feature of the frontend is the drag and drop interaction on the board screen. This interaction takes advantage of the [HTML5 Drag and Drop APIs](https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API). These APIs allow someone to make very custom drag and drop workflows using JavaScript and pure DOM elements.

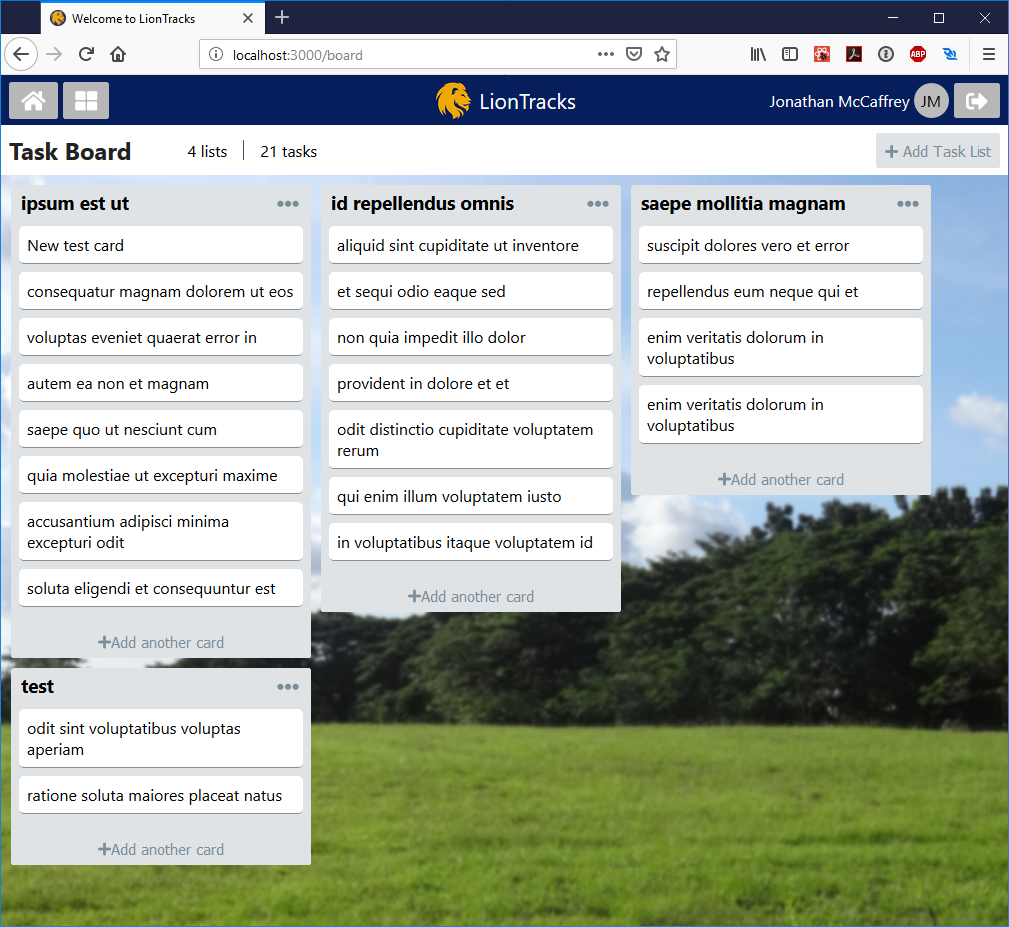
Here is a screenshot of these pages in their current state:

**Login Page**



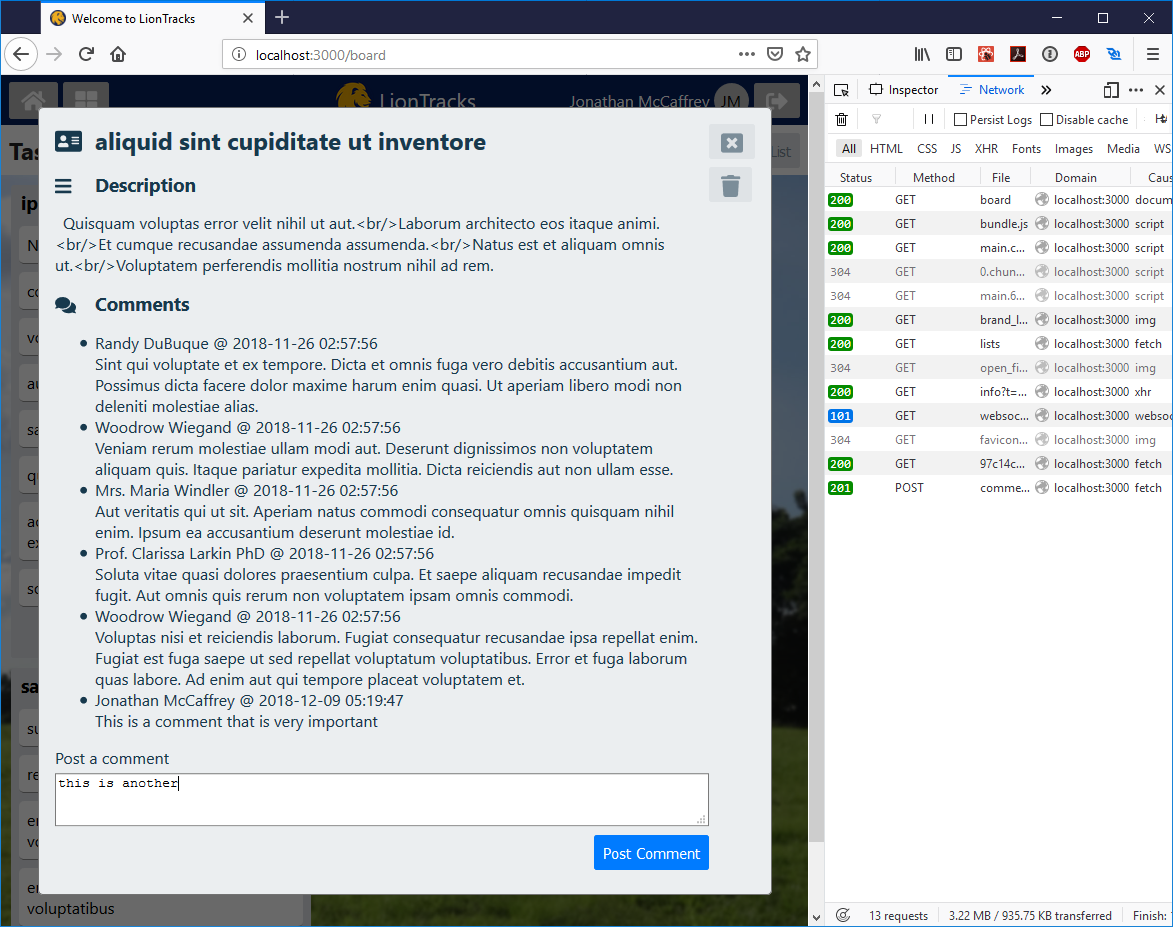
The login page captures the form onSumbit event, stops the form from actually submitting (as that would refresh the page), and then submits an API request to /auth/login with the email the user entered. If successful, the server sends back an auth token which is stored in memory (and backed up to local storage) for future use. Finally, on successful login, the screen sends the user to /board/ using the history API.

**Board Page**



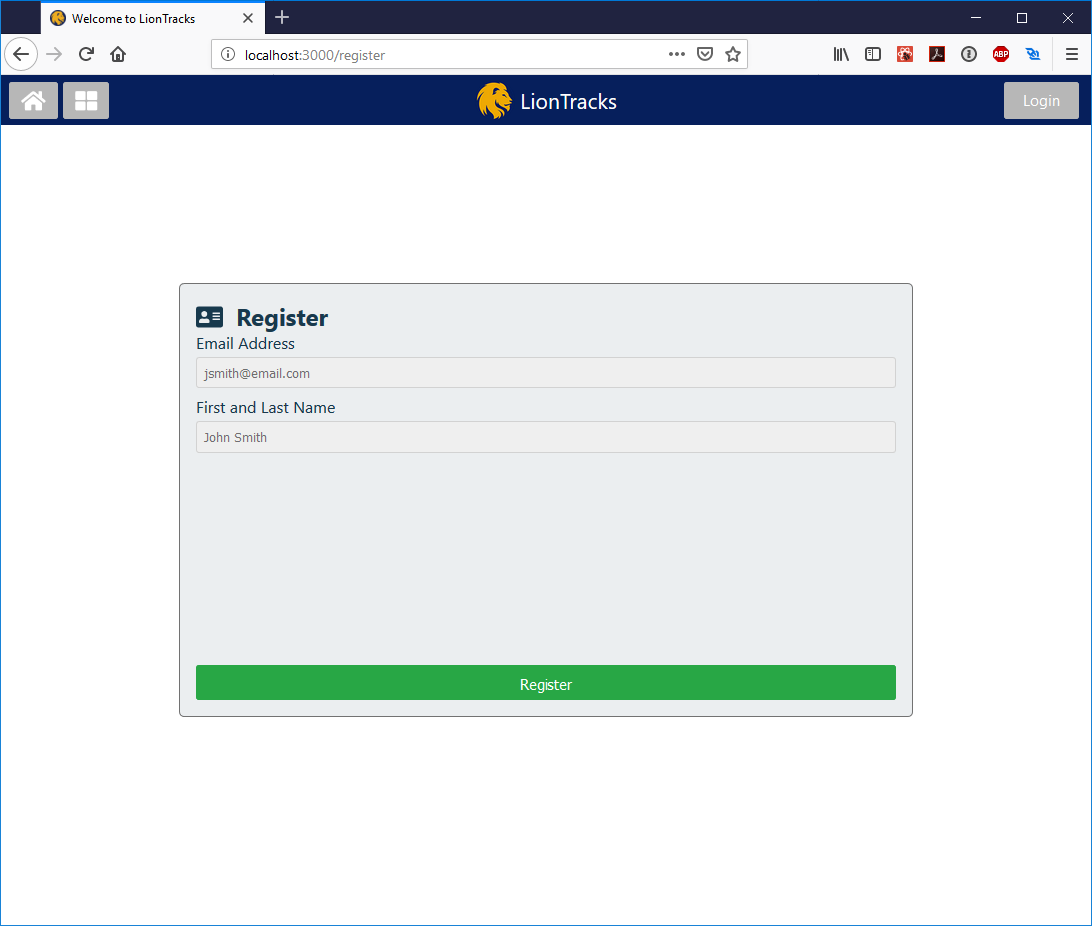
On this page users can create lists and cards, and then arrange them as they see fit. This screen heavily uses a newer CSS feature called [flexbox](https://css-tricks.com/snippets/css/a-guide-to-flexbox/) for all of the layout which made these lists of cards very easy to get right.

**Card Detail**

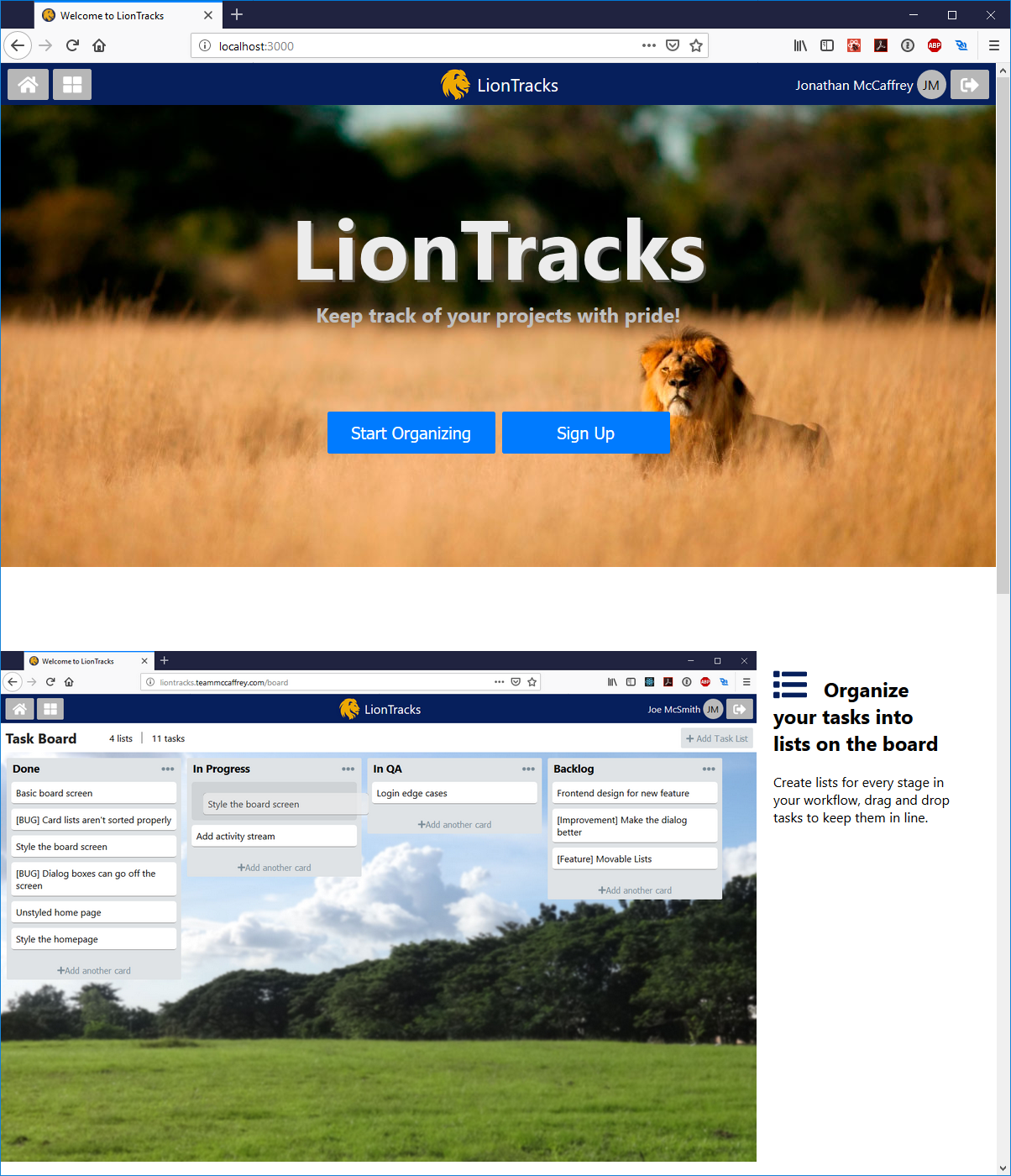


On this page the user can edit the card by clicking on either the title or the description and filling in what they want. When they press enter or click somewhere else, the change will be saved to the server by a PUT call. In the screenshot, the network debugger is open so we can see the REST transactions that the frontend is making with the backend as I post comments on the task.

**Register Page**



**Index Page**



I went for a basic marketing page layout here with a top jumbotron and some screenshots teasing the app below. The goal of this page is to get the user to click Sign Up and convert them into a registered account. Once they complete that step they can begin using the board.