Stranger’s Things (Craiglist\_Clone) Requirements:

<https://github.com/andyouf/Learning2>

Project needs to look like the example project “News” as much as possible. I would like the JavaScript split into separate files like done with News where import/export modules were used. I used Wing framework in the Stranger’s Things project.

Please comment code where possible.

I am going to outline the project:

**Watch - Building a tiny CRUD Application**

**CRUD FUNCTIONS, AND BUILDING READ & DELETE**

<https://www.youtube.com/watch?v=_iF6P61QosM&feature=emb_title&ab_channel=FullstackAcademy>

#### BUILDING CREATE

<https://www.youtube.com/watch?v=jW-85_6VDtw&feature=emb_title&ab_channel=FullstackAcademy>

#### BUILDING UPDATE

<https://www.youtube.com/watch?v=ylL8UN_b_5c&ab_channel=FullstackAcademy>

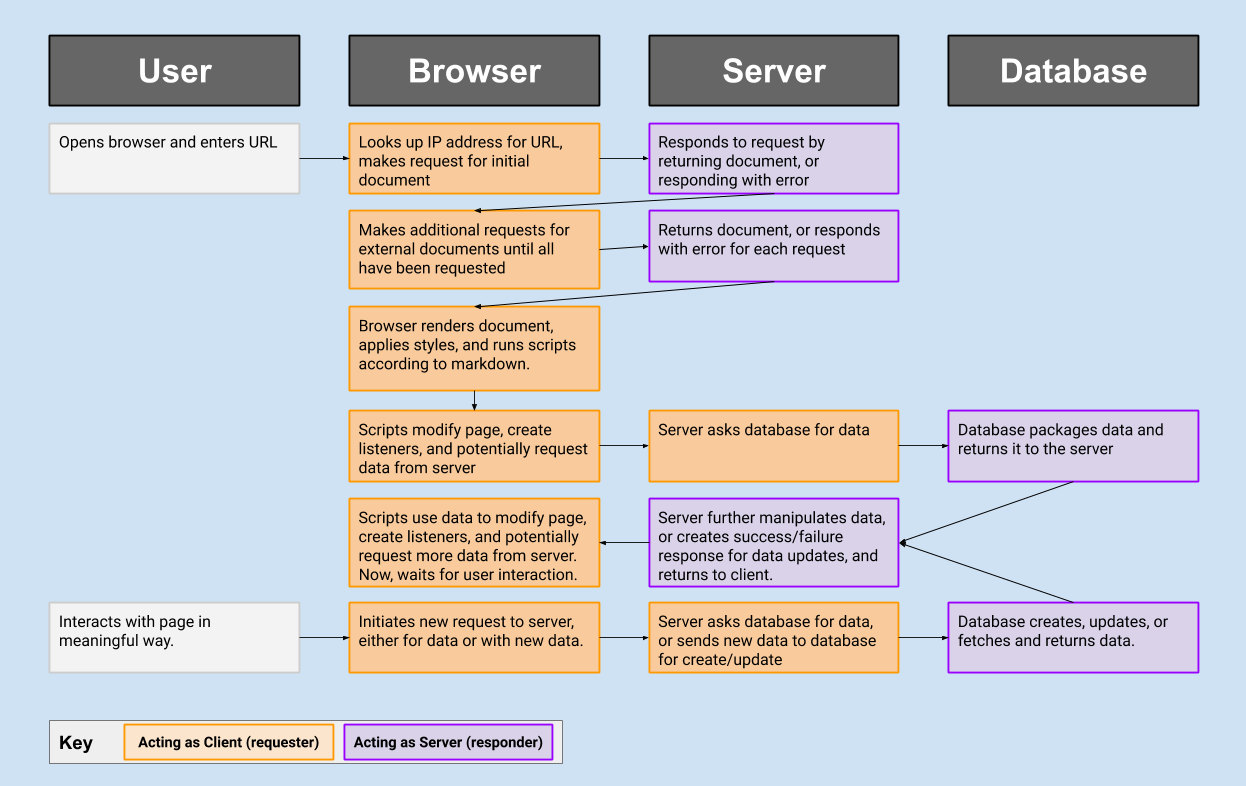
# History of a Page

The below diagram outlines how most web pages work, from the user typing in a URL, to page load and user interaction.

Read through it the way you would read a story. By the end of this project you will have built every thing possible on the front-end, and worked with an API which provides the back-end portion for you.

After this project we will move on to building our own back-ends, and be in complete control of our own data.

(I put this on GitHub since it cuts off)



# What do we need to know?

* The Web
  + What are [HTTP Messages](https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages)
    - What is a header? What typically goes in a header?
    - What is a body?
    - What are status codes? What are some typical status codes?
  + What are the main [HTTP Methods](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods)
    - GET, POST, PUT, DELETE, PATCH are the most common
    - We've been using GET all along (a basic fetch request defaults to GET)
    - POST is usually used when you want the server to create a new resource
    - PATCH & PUT are for partial modification / replacement of an existing resource
    - DELETE is for deletion of an existing resource
* Using fetch, Part 2:
* fetch('https://url.com/', {
* method: "POST",
* headers: {
* 'Content-Type': 'application/json',
* 'Authorization': 'Bearer XXXXXXXXXX',
* },
* body: JSON.stringify({
* // our data here
* })
* });
* What are [APIs](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction)
* What is CRUD?
  + There are four basic things we want to do with data, it's encapsulated in the acronym CRUD
  + Create - make new data
  + Read - read existing data
  + Update - make changes to existing data
  + Destroy - remove existing data

# Stranger's Things

## Building A Front-End, now with CRUD

We are going to be consuming another API, which is hosted at [Stranger's Things API](https://strangers-things.herokuapp.com/docs/).

This one is different, in that it allows us to do more than request data... it also allows us to create data! In order to do this we will have to learn about CRUD (an acronym for Create, Read, Update, Destroy), how to improve our fetch game, and about user authentication via JSON Web Tokens (JWT).

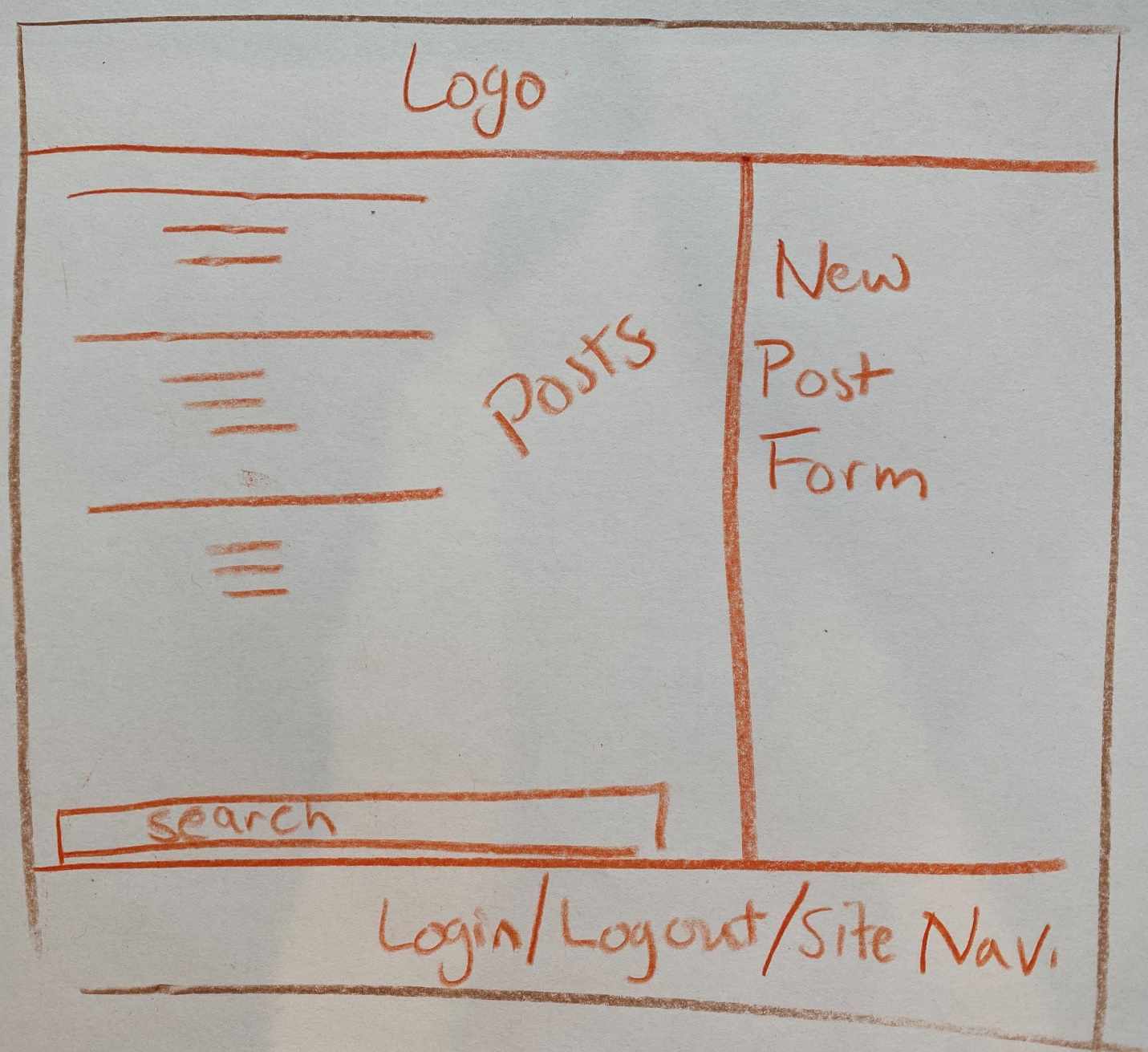
By the end of this you'll have a deeper grasp of what we can do on the front end, and start to have an idea of what's going on inside of servers when we interact with them.

# The Mantra

## Data-flow

Remember, like all applications we've developed so far, the overall flow looks like this:

1. Fetch initial data and build application state
2. Build initial interface from application state
   1. Attach data to interface where necessary
3. Set up all event listeners for interface
   1. Event listeners might transform interaction into API requests (**THIS IS NEW**)
4. User interaction triggers listeners, which in turn update data in multiple places:
   1. Pass update requests to API (**THIS IS NEW**)
   2. API returns success or failure for update
   3. On success, update state to reflect change
   4. On failure, render useful error message to user
5. If state changed, re-render changed portions of interface



# Suggested Development Path

While the goal is for you to work through the details of a larger-scale application on your own, and to read and absorb the documentation provided by the API we are using, I have provided some thoughts on the path you might take through the different requirements below.

#### REGISTER / LOGIN / LOGOUT

Since so much of the app requires users to be able to be logged in, starting here is probably prudent.

First you need to create a form which sends the correct data to the backend. Your register form should have a place to enter a username, a place to enter a password, and a place to enter password confirmation.

Perhaps you want to set a min property on password and username length, and you should definitely make them required on the form inputs.

If the form is filled out sufficiently, you'll need to make a correct AJAX request to the back end (see documentation). On successful user creation, you'll be given a token.

That token will need to be stored somewhere, perhaps localStorage, and sent with all future requests. Any fetch request with a valid token is considered authenticated, and the user corresponding to the token is assumed by the API to be the one making the requests.

Once you can register, you should offer a form to be able to login, and a button to be able to log out.

Remember, the presence of the token in localStorage should be how your front end treats the user as logged in/logged out, for all decisions it needs to make regarding rendering.

It might be useful for you to create some helper functions around this:

* logIn - which sets the token in localStorage
* logOut - which clears the token in localStorage
* isLoggedIn - which lets you know if there's a current user logged in

And also, since you'll need them for fetch requests:

* makeHeaders - which creates a headers object with our without the bearer token depending on what is in localStorage

For now, there are some "testing" API routes that let you see if you're currently logged in. You can hit them with fetch with headers (both logged in and logged out), and if you're setting them correctly you'll see the different responses you'll get.

You should also provide feedback on the form if the user provides incorrect credentials, as well as if the user tries to provide bad usernames or passwords.

#### POST FORM

You should make a form for users to make new listings. The fields for the form should match the fields that the API expects, and the submit button should be intercepted so that you can create the right fetch request.

This form could live in an aside that you show only to logged in users, or on submit you could show a modal that requires a person to sign in/sign up before you create the post. If you choose to show the form for logged in users only, make sure to update the interface whenever the user logs in... not only on page load.

The returned object is the new post. Since your state will keep track of all available posts, it might be wise to add the returned one to that array and call a re-render when it comes back.

#### POSTS VIEW

Next you should work on providing a pleasant view for all active posts.

As the app loads you should fetch the initial posts, and populate them into the element which holds the posts.

When you make a GET request to /api/posts, if you don't send a token, the API will only provide you with all posts. If you do, the posts made by the active user will also have the messages on them included.

There's an additional field provided by the API, isAuthor, which is only true for posts made by the user represented by the auth token.Make sure to suppress/show certain functionality for posts based on whether or not the current user is in fact the author.

#### DELETE BUTTON

Your posts should have a way for the active user to delete them, only if isAuthor is true for the post. Go ahead and add that now.

You'll also have to add a click handler to make that DELETE request. For each post, the delete handler will need a way to recover the post.\_id to help form the correct URL for the request.

On successful delete, make sure to remove that post from the page as well as from the array that is holding all posts in state.

#### MESSAGES FORM

For any post, you should add a form to send a message to the post author, only if there is a logged in user and the logged in user is not the one who made it.

The message form really only needs a text input, and a button to create the message.

Again, like the delete button, the submit handler will need a way to know how to form the correct URL so that the API responds, so make sure you're recovering it from the post element, if you're attaching it as data to begin with.

#### LOADING THE USER OBJECT ON PAGE LOAD

On page load, if there is a user logged in, you can make a GET request to /api/users/me and be given a user object. It will have all messages they've received, as well as all posts they've made (with messages partitioned by post).

It would be most useful to do this as part of your bootstrapping whenever the page is loaded.

#### SEARCH FORM

As part of your listings, add a little search form. Listen to the user typing into the field, and filter the listings in your state based on that. Re-render the filtered listings.

One thing that is tempting is to replace the state with the results of the filter, but this will mean that if the user deletes the filter, or chooses to filter a new term, that you'd have to re-fetch the original list of postings.

Instead, keep a searchTerm in your state, and when you render, render the postList (or whatever you call it) after the filter is applied. Below is a **very loose** attempt at providing some structure.

let state = {

posts: [],

searchTerm: '',

}

function renderPostList(postList) {

// renders, doesn't care about what's in postList

}

function postMatches(post, text) {

// return true if any of the fields you want to check against include the text

// strings have an .includes() method

}

let filteredPosts = state.posts.filter(post => postMatches(post, state.searchTerm));

renderPostList(filteredPosts) // only renders posts which pass postMatches

# Stretch Goals

#### QUICKER DESIGN, WITH FRAMEWORKS

Here, we have two choices when it comes to styling:

* Roll your own, use this as a chance to practice Grid, Flex, and your basic CSS
* Use a framework. Feel free to use one of the ones from before, or to explore an entirely new one.
  + <https://picnicss.com/>
  + <https://kbrsh.github.io/wing/>
  + <http://getskeleton.com/>

#### PLUGINS GIVE US EXTRA FUNCTIONALITY

jQuery has a number of useful plugins, and one that could be great for this project is [autocomplete](https://jqueryui.com/autocomplete/)

#### ANIMATIONS

Check out this CSS animation framework: [animate.css](https://daneden.github.io/animate.css/). It's really lovely, and a few targeted animations will make your application pop.

#### FONTS

We have on a number of occasions utilized [Google Fonts](https://fonts.google.com/). Feel free to add a few to your project, or to find fonts hosted elsewhere.

Additionally, icon fonts allow you to add icons to your application, and common icons increase your user's ability to onboard to your application much more quickly. Check out [Material Icons](https://material.io/resources/icons/?style=baseline) (used by adding <link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons" /> to your <head>), or [Font Awesome](https://fontawesome.com/).

# Advice

## The Great API Synthesis Project

At this point you've built a number of front-ends. They've been interactive, featured real live data, and have looked great.

The one thing we haven't done is allowed people to sign up for accounts, and create data personal to them. The point of this project is to provide you with the tools necessary to do that, and some time to hack away at it.

### MORE STRUGGLE

This project will test your abilities, and give you new skills relevant to your future career. Most applications have users, and most applications feature some amount of CRUD.

Despite the utility of this project, and the importance of the concepts, it will be hard. It's important to keep a good attitude about the difficulty level. You've made it far enough into the program that moments of personal struggle are more useful than ever.

The author of this project believes in you. You've got this.

### DON'T BE STUCK FOREVER

Remember, productive struggle feels sometimes like thrashing. It's very possible to worry that you'll never recover right up until the moment where you do.

However, it's also possible to be headstrong and ignore the signs that you're truly stuck. If you've used your time productively:

* trying multiple approaches with your code
* reading the documentation
* searching for similar problems via **google** or **stackoverflow**

Then it's time to rely on your community. You have classmates and an instructional team that are here for you, feel free to reach out to them!

# How do I know I'm done?

You will be done when you can check off all of the following:

1. You have a fully deployed website on netlify, and can share a link to it with the class
2. You meet all the criteria in the project description