Assignment 3: Callback Newsfeed

Due Monday, June 1st, at 11:59 PM

Overview:

The newsfeed is a classic component of modern websites. It usually contains posts from a variety of sources that serve to tell/show users what's going on. The most prevalent example of this is Facebook's newsfeed. When you view your Facebook newsfeed, you're able to see posts from all your friends, letting you know what's happening in their lives.

In this assignment, you're tasked with creating a dynamically updating newsfeed. Rather than serving textual content, this newsfeed will contain photos, music, and videos. You'll be interfacing with three APIs to deliver content: Flickr, SoundCloud, and YouTube. Users will input a textual description of a photo, song, or video. You'll access the three APIs in parallel, and give the user the option of posting from one of them.

Posts will be collected and stored in MongoDB, a flexible, document-based database. Any user who visits your site will be able to see all posts.

Unlike prior assignments, you'll be implementing a substantial part of the Node.js server that powers the application. Indeed, this server will access the aforementioned APIs and database. You'll also write some client-side logic to interface with your server.

Important client-side utilities:

Handlebars - A JavaScript library that allows you to render complex HTML templates and insert them into the DOM. It's a remedy for ugly `element.innerHTML = "[large HTML string]";` expressions.

Reference handout:
<https://docs.google.com/document/d/1W7DrKMonguxaPk03i81urvuaS2tq22N45HDchFRjeHE/edit?usp=sharing>
Further documentation: http://handlebarsjs.com/

jQuery - A JavaScript library that simplifies common JavaScript tasks, such as DOM selection/manipulation and animation.

Lectures on jQuery:
<https://docs.google.com/presentation/d/1wULDV4U4qRnI_N6KqAjcSP7972UDQrSLgfid06wSj04/edit?usp=sharing>
Further documentation: http://jquery.com/

Important server-side utilities:

You'll need to download/install both node and MongoDB using the instructions below.

Node.js - A JavaScript runtime environment for creating event-driven server and networking applications.

Installation instructions: <http://nodejs.org/download/>
Further documentation: http://nodejs.org/api/all.html

MongoDB - A database that uses JavaScript syntax to retrieve and store JSON information.

Installation instructions: <http://docs.mongodb.org/manual/installation/>
Reference handout:
<https://docs.google.com/document/d/19yV1x9qbsRC27Flba-3V5ejdTrChfO2h6d6UyzpEpX8/edit>
Further documentation: http://docs.mongodb.org/manual/tutorial/getting-started/

Note that lectures covered the fundamentals of server-side development with node and MongoDB but omitted full API descriptions that may prove necessary for this assignment. Freely consult the linked resources above when confused.

Overview of Starter Files:

We have provided the following starter files to help you get started. Take a look at the overview we gave in class to get a better idea of how everything fits together:
<https://docs.google.com/a/stanford.edu/presentation/d/17KufHu1JXcfArWJwe2GXKkLI2utl7UGKPrXHXMDynzA/edit?usp=sharing>

HTML:

views/index.html:
This file contains the HTML layout of the newsfeed application and all the Handlebars templates that you will need to complete the assignment. You don't need to modify this file, but feel free to add elements if you'd like to extend the baseline functionality.

CSS:

public/css/style.css:
This file contains the CSS styling for the application. You may modify this file, though it's not required for the assignment.

JavaScript libraries:

public/js/lib/handlebars.js:
This file contains the Handlebars library, which allows you to create templates and render them. Refer to the Handlebars handout (see "Important utilities" above) for more details. Do not modify this file.

public/js/lib/bootstrap.min.js:
This file contains the bootstrap javascript library that handles dynamic resizing of the page. Do not modify this file.

public/js/lib/jquery.min.js:
This file contains the jQuery library to simplify client-side scripting in this application. For a refresher on how to use jQuery, check out the lecture on jQuery. Do not modify this file.

Client-side JavaScript you're given:

public/js/main.js:
This file is the entry point to the application. It calls `MainView.render()`.

public/js/main-view.js:
This file exports the `MainView.render()` function, which calls `NewsfeedView.render()` and `SearchView.render()`.

public/js/search-view.js:
This file exports the `SearchView.render()` function, which will render the search results returned by the `SearchModel`.

public/js/handlebars-helpers.js:
This file exports functionality used to render the Handlebars templates.

Client-side JavaScript you'll be implementing:

public/js/search-model.js:
This file exports the `SearchModel.search()` function, which makes an XMLHttpRequest to search the Flickr, SoundCloud, and YouTube APIs.

public/js/post-model.js:
This file exports the `PostModel` object, which contains functions that load, add, and remove posts from the newsfeed.

public/js/newsfeed-view.js:
This file exports the `NewsfeedView.render()` function, which will load and render all posts.

Server-side JavaScript you'll be implementing:

models/post.js
This file defines the schema of a post. It exports a model that can be used to create/read/update/delete posts in MongoDB.

lib/soundcloud.js

This file interacts with the SoundCloud API to retrieve and return search results. It includes a SoundCloud API key you should be using to make calls to the SoundCloud API

lib/youtube.js
This file interacts with the YouTube API to retrieve and return search results. It includes a YouTube API key you should be using to make calls to the YouTube API

lib/flickr.js
This file interacts with the Flickr API to retrieve and return search results. It includes a Flickr API key you should be using to make calls to the Flickr API.

routes/index.js
This file contains all the URL routes that the server exposes to the client. Clients can make HTTP requests to these routes to update models or get data back from the server.

Running the application:

Download/Install Node.js and MongoDB if you haven't already (see the 'Important server-side utilities' section).

To view your application, you'll need to start the server. In your terminal, `cd` to the project directory. Then, run the following commands:

```
npm install
sudo npm install -g nodemon
```

This will install all server dependencies into the `node_modules` folder. Now, start MongoDB (make sure you're in your project directory):

```
mongod --dbpath data
```

Keep a terminal open with `mongod` running. Start a new terminal and run the node server:

```
nodemon app.js
```

Note: if you see 'Error: failed to connect to [localhost:27017]', MongoDB isn't running. Make sure to run it using the `mongod` command above.

Keep a terminal open with `nodemon` running. You can now see the starter application at `localhost:3000` (or, equivalently, `127.0.0.1:3000`) in your browser.

For those who are curious, `nodemon` is a tool that runs node and automatically restarts it each time you make changes to your server (so you don't manually have to restart).

Terminology:

Post: In Callback Newsfeed, a post represents a photo, song, or video that's on the newsfeed. Each post contains three fields: `api` (which API the post is derived from), `title` (the title of the post), `source` (a Flickr, SoundCloud, or YouTube source URL), and `upvotes` (the number of upvotes the post has received).

Abstraction:

Callback Newsfeed's client-side JavaScript is divided into models, views and utilities, just as you had in assignment two. Refer back to Callback Yelp's handout if you'd like a refresher on the model-view pattern.

The server-side JavaScript is split into three API interfaces, all located in the lib directory, which you'll be responsible for completing. These API interfaces will be used in routes/index.js, which defines the URL routes (the external-facing HTTP API) exposed by the server.

Requirements:

We'll break down the requirements on a file-by-file basis. We highly recommend implementing the files in the same order that we explain the requirements in.

SoundCloud API (lib/soundcloud.js):

First, you'll need to write an interface to the SoundCloud API. Open up lib/soundcloud.js. Your job is to implement the `search()` function.

Note that you'll need to reference the SoundCloud API docs, just as you did with the Google Maps API from assignment two. You may find them at <https://developers.soundcloud.com/docs/api/reference>.

`search()` should do the following:

Make a `GET` request to the SoundCloud tracks endpoint (`SC_URL`) to search for tracks. You'll need to pass two query string arguments:

1. SoundCloud API client ID (`SC_CLIENT_ID`)
2. `q` (with value being the desired `query`)

Parse the results. Transform each track into a JavaScript object with the following format:

```
{
  'title': [track title],
```

```
  'source': SC_EMBED_URL + [track ID]
}
```

`SC_EMBED_URL` and `SC_CLIENT_ID` are constants that we've given you.

Store these objects in an array, and pass that array to the given `callback()`. Ensure that the first element of the array is the most relevant track (i.e. the track that came first in the SoundCloud API response).

If any error occurs, call `callback()` with a single parameter representing the error.

YouTube API (lib/youtube.js):

Next, write an interface to the YouTube API. Your job is to implement the `search()` function. You'll need to reference the YouTube API docs, located at <https://developers.google.com/youtube/v3/docs/search/list>.

`search()` should do the following:

Make a `GET` request to the YouTube search endpoint (`YT_URL`) to search for tracks. You'll need to pass four query string arguments:

1. `key` (with value `YT_API_KEY`)
2. The given search `query`
3. `part` (with value `'snippet'`)
4. `type` (with value `'video'`)

Parse the results. Transform each video into a JavaScript object with the following format:

```
{
  'title': [video title],
  'source': YT_EMBED_URL + [video ID]
}
```

`YT_EMBED_URL` and `YT_API_KEY` are constants that we have given you.

Store these objects in an array, and pass that array to the given `callback()`. Ensure that the first element of the array is the most relevant video (i.e. the video that came first in the YouTube API response).

If any error occurs, call `callback()` with a single parameter representing the error.

Flickr API (lib/flickr.js):

Write an interface to the Flickr API. Your job is to implement the `search()` function. You'll need to reference the Flickr API docs, located at <https://www.flickr.com/services/api/flickr.photos.search.html>.

`search()` should do the following:

Make a `GET` request to the Flickr API (`FLICKR_URL`) to search for photos. You'll need to pass seven query string arguments:

1. Our Flickr API key (`FLICKR_API_KEY`)
2. The given search `query`
3. `method` (with value `'flickr.photos.search'`)
4. `format` (with value `'json'`)
5. `media` (with value `'photos'`)
6. `sort` (with value `'relevance'`)
7. `nojsoncallback` (with value `1`)

Parse the results. Transform each photo into a JavaScript object with the following format:

```
{
  'title': [photo title],
  'source': 'https://farm' + [photo farm] + '.staticflickr.com/' +
    [photo server] + '/' + [photo ID] + '_' + [photo secret] + '_z.jpg'
}
```

Note: the `source` parameter is dervied from the documentation at <https://www.flickr.com/services/api/misc.urls.html>.

Store these objects in an array, and pass that array to the given `callback()`. Ensure that the first element of the array is the most relevant photo (i.e. the photo that came first in the Flickr API response).

If any error occurs, call `callback()` with a single parameter representing the error.

Testing APIs

To test each API individually, enter your terminal and ensure that you are in the assignment three project directory. Create a node REPL (read-evaluate-print loop) by executing the following command in the terminal:

```
node
```

Once the REPL fully loads (you'll see a > prompt), execute the following statements:

```
var api = require('./lib/[api_name].js');
api.search("Yellow Flicker Beat", function(error, results) {
  console.log(error, results)
});
```

Replace `[api_name]` with the api that you want to test (`youtube`, `soundcloud`, or `flickr`). Replace `"Yellow Flicker Beat"` with your desired search query. Check to see if the function returns logical search results. Ensure that `error === null`.

Joint API (routes/index.js):

Now that you've implemented all the individual APIs, you'll need to tie them together into one. When a user searches for a string, your job is to simultaneously query all three APIs. You should take the single most relevant result from each API (for a total of three results), and return a JSON array to the client.

To accomplish this, define a new route that handles a `GET` request on `/search`. In the callback, query all three APIs by calling the functions you wrote in lib/soundcloud.js, lib/youtube.js, and lib/flickr.js.

The client will make requests to `/search?query=[search query here]`. You can access the `query` variable using `request.query.query`.

The API requests should be simultaneous; that is, your code should have the following form:

```
soundcloud.search(query, function() { ... });
youtube.search(query, function() { ... });
flickr.search(query, function() { ... });
```

Do not nest the callbacks. The following is incorrect and will not be accepted:

```
soundcloud.search(query, function() {
  // ...

  youtube.search(query, function() {
    // ...

    flickr.search(query, function() { ... });
  });
});
```

Now, in each callback function, get the first (most relevant) result, and annotate it by setting

one key-value pair:

```
result.api = /* a string representing which API the result was
                from (i.e. 'youtube', 'soundcloud', or 'flickr'). */;
```

The client uses this property to display the result on the front-end.

Append the result to an array. When you've gotten all three results in the `results` array, send them back to the client by calling `response.json(200, results)`. Note: some APIs may return no results for certain search terms; in this case, `results` will contain fewer than three entries.

Keep in mind the three callback functions could be called in any order, as the APIs may return at different times. One of these callbacks will be called last. Within each callback, you need to assess whether this is the last callback called (i.e. by keeping a counter) and, if so, return the response.

Throw an error if any of the callbacks fail.

At this point, take a moment to assess whether your code is working. Make requests to the `/search` endpoint to see whether you're getting back reasonable data. You can accomplish this with the `curl` command-line utility. Note that the response should contain three or fewer search results (one from each API, if that API returned results), and they should be formatted in JSON.

For instance, if you search for 'yellow flicker beat', you should get back the following results (your order may be different):

```
[
  {
    "title": "Lorde - Yellow Flicker Beat (Hunger Games)",
    "source": "http://www.youtube.com/embed/3PdILZ_1P74",
    "api": "youtube"
  },
  {
    "title": "Lorde -  Yellow Flicker Beat",
    "source":
"https://w.soundcloud.com/player/?url=http%3A%2F%2Fapi.soundcloud.com%2Ftra
cks%2F169903379",
    "api": "soundcloud"
  },
  {
    "title": "red, orange, yellow flicker beat sparking up my heart.",
```

```
    "source":
"https://farm4.staticflickr.com/3928/15237088889_97d162b9ed_z.jpg",
    "api": "flickr"
  }
]
```

`SearchModel` (public/js/search-model.js):

At this point, you'll be able to start interacting with the APIs from the client-side. In public/js/search-model.js, you'll need to implement `SearchModel.search()`.

`SearchModel.search()` should make a request to the `/search` endpoint, passing in the `query` parameter via query-string.

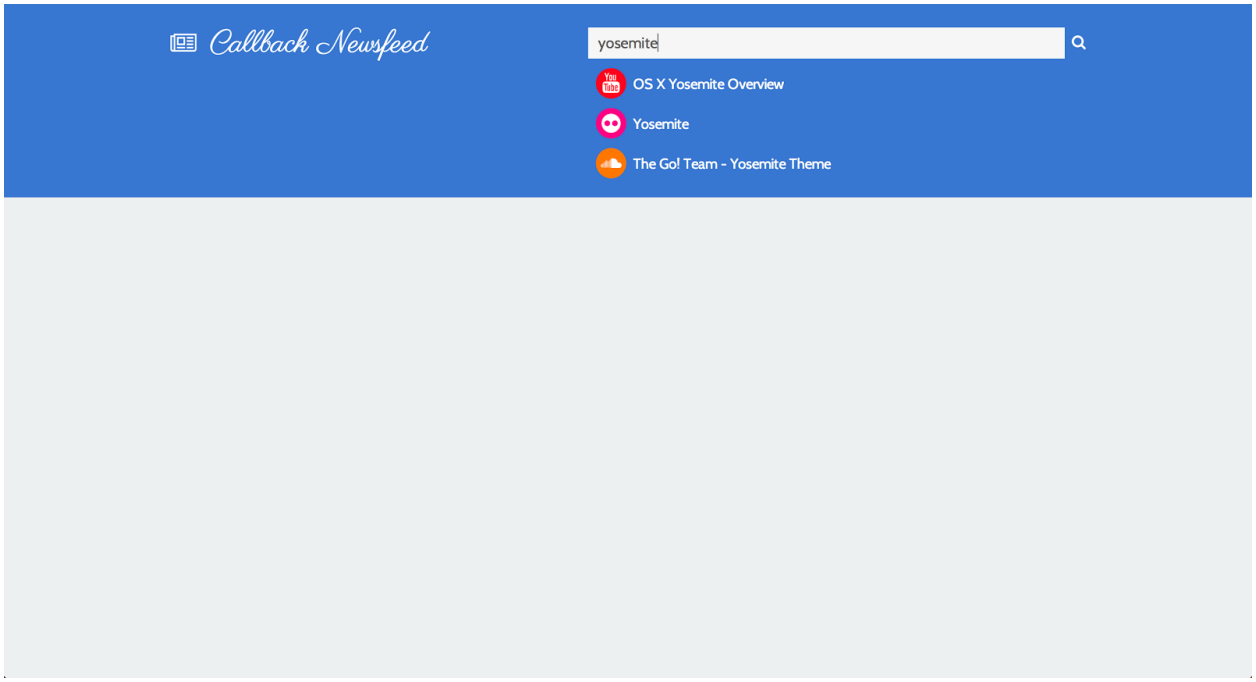
During this step, you should call `encodeURIComponent()` on the `query` parameter before appending it to the URL. This will escape any special characters in the query so that it can be sent to the server correctly. See the first answer at <http://stackoverflow.com/questions/75980/best-practice-escape-or-encodeuri-encodeuricomponent> (look at the section under `encodeURIComponent()`).

Call the callback, passing in the array of results returned by the `/search` endpoint. Recall that the first argument to the callback is the error that occurred, if any.

`SearchView` (public/views/search-view.js):

When the search form is submitted, the `SearchView` uses the `SearchModel` to query the server. It displays the API results in a dropdown and allows the user to pick which one he/she would like to post. We've given you the implementation of `SearchView`. You can examine it if you'd like, but you don't need to modify the file.

At this point, if you visit the app in your browser, you should be able to search and see results, as depicted in the image below:



Clicking on a result doesn't actually create a post yet. That's what you'll be working on next.

Post Schema (models/post.js)

In order to store posts, you first need to create a Mongoose schema to interface with MongoDB. See the MongoDB reference handout <https://docs.google.com/document/d/19yV1x9qbsRC27Flba-3V5ejdTrChfO2h6d6UyzpEpX8/edit?usp=sharing> if you need a refresher.

Create a post schema in models/post.js with four fields:

`api` - A string representing which API the post comes from
    (`'flickr'`, `'youtube'`, or `'soundcloud'`)
`source` - The source URL of the API resource
`title` - The title of the post
`upvotes` - The number of upvotes the post has

Then, set `module.exports` to a Mongoose model corresponding to this schema.

Post API (routes/index.js):

Your server must provide the client with an interface to create/read/update/delete posts. To accomplish this, define four routes in routes/index.js:

- A `GET` request to `/posts` should retrieve all posts from the database and send a JSON

array to the client (via `response.json()`).
- A `POST` request to `/posts` should add a post to the database. The client will pass in `api`, `source`, and `title` parameters in the request body (accessible via `request.body.api`, `request.body.source`, and `request.body.title`). Validate that all three parameters are provided. Then, create and store a post in MongoDB (set `upvotes` to 0). Send the newly-created post to the client using `response.json()`.
- A `POST` request to `/posts/remove` should delete a post. The client will pass in an `id` parameter in the request body representing the post to delete. Send an empty response body back to the client.
- A `POST` request to `/posts/upvote` should upvote a post. The client will pass in an id parameter in the request body representing the post to upvote. Send the updated post back to the client using `response.json()`.

`PostModel` (public/js/post-model.js):

Now that you've implemented the server-side routes, you'll need to write the client model that interfaces with them. Specifically, implement the four functions in public/js/post-model.js to get a list of posts, add a post, delete a post, and upvote a post.

To accomplish this, make AJAX requests to the correct server-side paths and parse the response. Make sure to call the given `callback()`. This code will be very similar to what you wrote in `EntryModel` in assignment two.

`NewsfeedView` (public/js/newsfeed-view.js):

Now that you've written a model to encapsulate your data, you must write a view to render that data. Note that, for this application, rendering posts is expensive. Each post is an API widget, and that widget makes requests to external services. As a result, we want to avoid re-rendering posts as much as possible.

`NewsfeedView.render()` should retrieve all posts, calling `NewsfeedView.renderPost()` for each one. Pass in `false` for the `updateMasonry` argument.

After calling `NewsfeedView.renderPost()` for each retrieved post, `NewsfeedView.render()` should do the following (simply copy-paste this code):

```
$newsfeed.imagesLoaded(function() {
  $newsfeed.masonry({
    columnWidth: '.post',
    itemSelector: '.post'
  });
});
```

This updates the newsfeed grid display.

`NewsfeedView.renderPost()` should render the `#newsfeed-post-template` Handlebars template, passing in `post` as template data. The resulting HTML must be prepended to the `$newsfeed` element.

Note that you should **not** replace `$newsfeed`'s inner HTML (i.e. don't call `$newsfeed.html()` in any way), as this will re-render the entire `$newsfeed` element. Instead, call the `$` function, passing in the post's HTML. Assign this to a variable called `$post` (this specific name is required, as it's used in code snippets below). `$post` represents the element that needs to be added to `$newsfeed`. Then, call `$newsfeed.prepend()`, passing the `$post` you just created. This will circumvent the need to re-render all posts.

For each post, you'll need to attach two event listeners:

- When the `.remove` button is clicked, remove the post using `PostModel.remove()`. In the callback, copy-paste the following code:

  ```
  $newsfeed.masonry('remove', $post);
  $newsfeed.masonry();
  ```

  This will remove the post from the DOM and update the grid display.

- When the `.upvote` button is clicked, upvote the post using `PostModel.upvote()`. Update the upvote count of the corresponding `.post` element in the DOM. Do not fully re-render the post; only change it's upvote count.

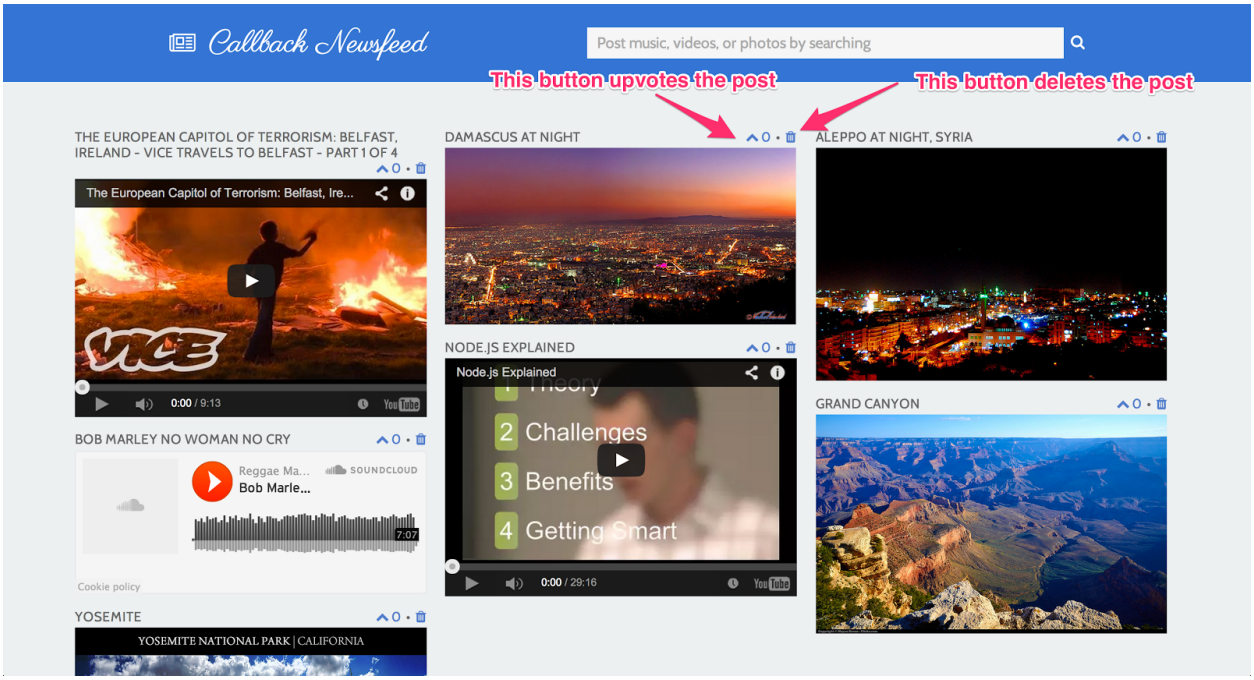
Note that when MongoDB creates a post document on the server, it adds an `_id` attribute representing that post's ID. You'll need to pass this ID into `PostModel`'s functions.

If any error occurs, display it in the `.error` div.

At the end of `NewsfeedView.renderPost()`, re-render the grid by copy-pasting this code:

```
if (updateMasonry) {
  $newsfeed.imagesLoaded(function() {
    $newsfeed.masonry('prepended', $post);
  });
}
```

Below is a diagram showing rendered posts, along with the buttons of interest:



Grading:

We will grade your application based on two criteria: the functionality of your application and proper JavaScript coding style.

Functionality (60%):
You must implement all the required tasks above. Follow the design pattern we've laid out for this assignment, and do not violate the abstractions. Decompose common functionality.

Given that you have done the above and your application works, you should receive full marks. We will deduct points for incomplete/omitted features, lack of robustness, and bad design.

Style (40%):
Your style will largely be graded on consistency. Follow the style already present in the starter files. If you work with JavaScript in industry, the team you'll be working with will likely have a baseline style established. You should always match this style. Consistency is everything.

Make sure to document your code. It's hard for us to stress this enough. The goal is to make your code so understandable that we can read it with ease. Any complex logic that you had to reason through should be clarified with inline comments. All functions should be documented: what does the function do, what are its parameters, what does it return or pass to a callback, and, if applicable, how does it handle errors?

Assignment length

The reference solution is approximately 280 lines of code across all six files (lines are those with semicolons, colons, or opening/closing braces; not including comments). Good solutions will likely range from 250 to 350 lines of code. Exactly how long the assignment will take depends on the person; our estimate is 8-15 hours, but some people may take more, and others may take less.

Honor code

All of your code for this assignment should be your own original work. Do not copy other students' work. If you referenced online sources, cite them as a comment in your code.

Submission

Visit <https://web.stanford.edu/class/cs42/cgi-bin/newsfeed.php> and submit a zip file of your entire piazza assignment directory (all starter files should be included in the zip). You may submit as many times as you'd like before the deadline; only your last submission will be retained.