Client Side Hydration

When the user first loads the page, we need to show some list items right away. We'll learn how to in this lesson!

WE'LL COVER THE FOLLOWING

Server side

We first want to get to a point where we have our initial set of data. As soon as the window loads, let's pull data from the server and display it as a series of tweets.

To know if we've succeeded, the server should have some data.

We'll write a short script to do so:

```
function loadTestData() {
  const sampleData = [];
  const sampleDataSize = 20;
  for (let i = 0; i < sampleDataSize; i++) {
    const message = getRandomString({
      length: getRandomInteger({min: 10, max: 150}),
      includeSpaces: true
    });
    const firstName = getRandomString({
      length: getRandomInteger({min: 3, max: 7}),
      includeSpaces: false
    });
    const lastName = getRandomString({
      length: getRandomInteger({min: 3, max: 7}),
      includeSpaces: false
    });
    const handle = '@' + getRandomString({
      length: getRandomInteger({min: 4, max: 8}),
      includeSpaces: false
    });
    sampleData.push({
      tweet: {
```

```
        name: `${firstName} ${lastName}`,
        message, handle
      }
    });
  }
  for (const data of sampleData) {
    // Do nothing with result
    api.post(HOST + 'tweets', data, () => {});
  }
}
```

Hopefully, this gives a good set of data. I've changed the `getRandomString` function slightly to include spaces only if specified, since we don't want things like handles to have spaces. We do want messages to, however.

If you're interested in the server changes, here they are:

```
class Database {
  constructor() {
    this.tweets = [];
  }

  query({lastTweetId, pageSize}) {
    if (!lastTweetId) {
      return this.tweets.slice(0, pageSize);
    }
    for (let i = 0; i < this.tweets.length; i++) {
      const currentTweet = this.tweets[i];
      if (currentTweet.id === lastTweetId) {
        return this.tweets.slice(i + 1, i + 1 + pageSize);
      }
    }
    return [];
  }

  insert(tweet) {
    this.tweets.push({
      tweet,
      id: getRandomString({length: 50}),
      timestamp: (new Date()).getTime()
    });
  }
}
```

The only thing I want you to note is that the `insert` function adds data to the request instead of putting in the database directly. Namely, we give it a unique ID (this has to be done on the server to avoid conflicts with others) and the current time as received by the server.

Fetching and rendering tweets

Now, for the actual hydration on the client, we just call `get` with a pageSize.

Now, for the actual hydration on the client, we just call `get` with a pageSize, and then pass the relevant fields to a template we created.

The hydration looks like this:

```
const DEFAULT_PAGE_SIZE = 5;
const DEFAULT_SORT_ORDER = 'recent';

function onNewTweets(data) {
  // TODO create tweet and render it
}

function hydrate() {
  const params = {
    pageSize: DEFAULT_PAGE_SIZE,
    sortOrder: DEFAULT_SORT_ORDER
  }
  api.get(HOST + 'tweets', params, onNewTweets);
}

loadTestData();
hydrate();
```

Pretty simple.

As for the template, we have to move that whole chunk of HTML to JavaScript to populate it dynamically. Some people are extremely opposed to this trend in the web – and it is a trend now. One of the biggest complaints about a popular framework, AngularJS, was that it coupled the HTML and JavaScript too much. React went one step further and just had all the HTML reside within JavaScript files. So many websites these days are dynamically populated that there's little static HTML on them (maybe just the footer with links to "about," "career," etc.), in which case it can make sense for convenient templating.

In any case, I don't want to take up space with the template in the JavaScript file. Just know that there's a function `createTweet` that returns a string, which represents the HTML we wrote earlier with the values set to the parameters.

```
function createTweet({name, handle, message}) {
  const template = `
    <div class="tweet">
      ...
      <div class="tweet__main__message">
        ${message}
      </div>
      ...
    </div>
  `;
```

```
  return template;
}
```

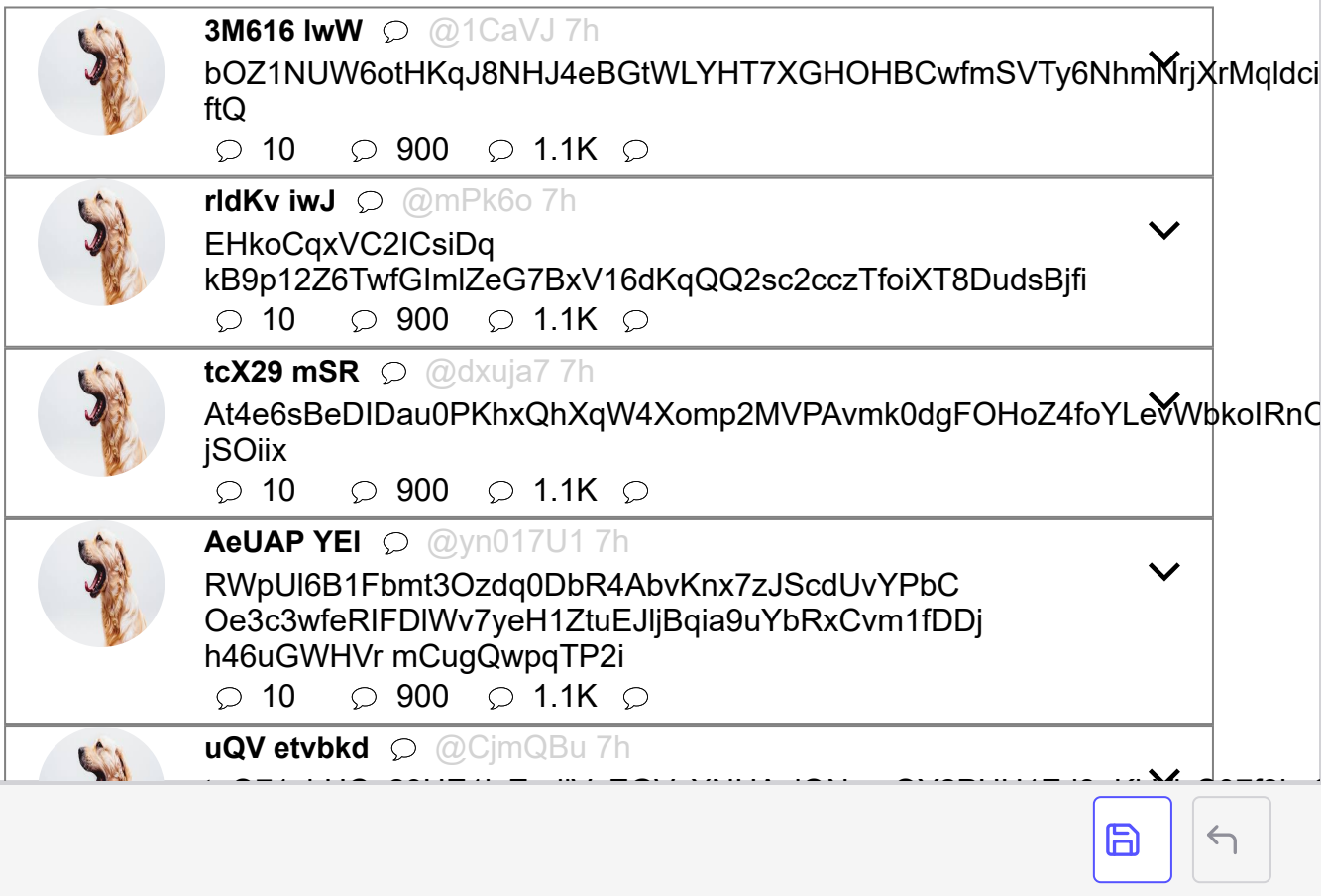
Putting that all together, we get:

Output

JavaScript

HTML

CSS (SCSS)



Line breaking long tweets

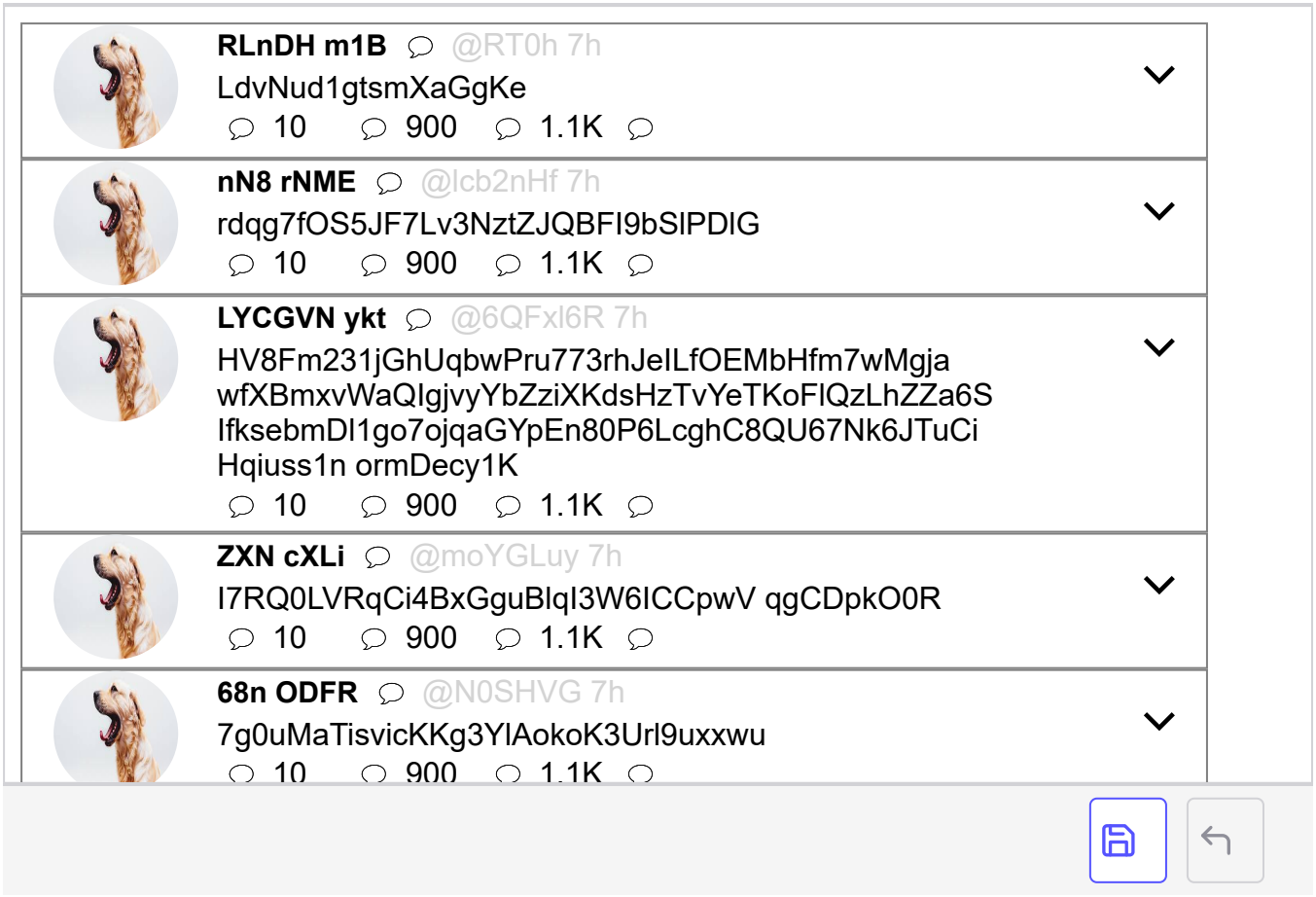
So, I'm not sure about what you see, but if you refresh enough, you'll get long outputs and the component styling breaks. We didn't rigorously test our CSS when we first applied styles – but that's okay. That's what generated test data is for! It looks like we're not breaking text that's too long. Thankfully, we have an easy fix for this!

Output

JavaScript

HTML

CSS (SCSS)



Now it looks indistinguishable to me from a typical Twitter discussion.

In the next lesson, we'll add a loading element to our list.