Web Programming

Workshop 05 - Supporting Interaction with React

Contents

Introduction

In the previous workshop, we used React to render a non-interactive mockup of Facebook's desktop layout. We also created a mock database and server.

In this workshop, we will make the mockup *interactive* and *dynamic*. We will connect the React components to the server, and connect specific React components to user input. In doing so, users will be able to post status updates, like status updates, and add comments to comment threads.

In addition, we will cover how to make your application support multiple pages in a modern way using `react-router`. You will use the same technique in your startup products to let the user navigate between multiple *screens* of your application.

Since this workshop is essentially a continuation of the previous workshop, we begin where we left off: at step 5. Note, if you already completed Workshop 04 then you already have the repository you need to begin work here.

Grading Rubric

- 20% commit `fb5`
- 20% commit `fb6`
- 15% commit `fb7`
- 45% commit `fb8`
- 15% Database and mock objects appropriately changed to support "Liking"/"Unliking" comments
- 15% Server method(s) added to handle Liking/Unliking comments
- 15% "Like"/"Unlike" link on comments triggers the server method, appropriately updates the "Like" counter, and appropriately toggles between "Like" and "Unlike"

Step 05: Making React Components Interact with the Mock Server

When we turned our React Components into separate JavaScript files, we went from the bottom-up. This time, we will start from the top-down, trickling server data through our React Components, beginning with `app.js`.

app/app.js

You only need to make one simple change to `app.js`: Change `<Feed />` to `<Feed user={4} />`. Note that `4` is the ID of the "John Vilk" user in the mock database. In our mockup, we are assuming that "John Vilk" logged in, and is the user currently using Facebook.

app/components/feed.js

The Feed component is now receiving a `user` prop with the ID of the active user. It will need to use this ID to request the Feed object for the user from the server.

This presents a problem for React components. When a React component is created, its `render()` method is immediately called, which constructs HTML for the webpage. We cannot render the Feed until it receives data from the server, but server requests are *asynchronous* – they are given a function that is called some time in the future with the result from the operation.

Here's an example of a *synchronous* operation:

```
[0,1].forEach((num) => {
  console.log("Processing " + num + "...");
});
// `map` has run before the next line runs
console.log("Processing complete.");
```

The above program would print:

```
Processing 0...
Processing 1...
Processing complete.
```

Here's an example of an *asynchronous* operation:

```
// addOnServer sends a number to a server, and
// receives that number with 1 added.
addOnServer(0, (result) => {
  console.log("Result: " + result);
});
// The below line runs before the callback function above does.
console.log("Server request sent.");
```

The above program would print:

```
Server request sent.
Result: 1
```

Asynchronous operations let your program soldier on while a request waits to be fulfilled. As a subtlety to asynchrony, your callback function will never run at the same

time as other JavaScript. It's difficult to explain in a workshop, but this Mozilla Development Network has a good explanation about JavaScript events and concurrency.

Back on track: As a result of asynchrony, React needs to render our Feed *before* we get data from the database. How can we do this?

We can do this by:

1. Rendering an "empty" Feed component.
2. Re-render the real Feed when data is available.

React facilitates this pattern with a special `state` field. While the `props` field is read-only, `state` can be modified via the `this.setState()` function. State changes trigger the component to re-render. In addition, you can set the initial value of `state` in the React component's constructor, which, like a Java constructor, runs when the component object is first created – and before the first call to `render()`.

Recall that a Feed object in the database has a single field, `contents`, that contains references to FeedItem objects. We can set `state` to `{ contents: [] }` in the constructor, which represents an empty feed, and write the `render()` method like normal. The Feed will be empty until the server response returns, which will change the `state` to include FeedItem data.

Note: If we wanted to get fancy, we could modify the Feed's `render()` function to display a loading animation until the server responds. You could, for example, set the state to `{ loaded: false; }` in the constructor, change `render` to create the animation when `loaded` is false, and set `loaded` to `true` once the server response comes back in.

Here's what our modified `Feed` class looks like, before we hook up the database:

```
export default class Feed extends React.Component {
  constructor(props) {
    // super() calls the parent class constructor --
    // e.g. React.Component's constructor.
    super(props);
    // Set state's initial value.
    // Note that the constructor is the ONLY place
    // you should EVER set state directly!
    // In all other places, use the `setState` method instead.
    // Setting `state` directly in other places will not
    // trigger `render()` to run, so your
    // program will have bugs.
    this.state = {
      // Empty feed.
      contents: []
    };
  }
```

```
  render() {
    return (
      <div>
        <StatusUpdateEntry />
        { Code for creating FeedItems }
      </div>
    )
  }
}
```

Note that I didn't write code for creating `FeedItem`s yet, because we haven't covered how to do that! Remember when we used `React.Children.map` to render a series of `Comment` components in `CommentThread`? We can do something similar here to render `FeedItem` components. Regular JavaScript arrays have a `map` function. Since a JavaScript array's `map` function is defined directly on the object, it only needs one argument: A function.

```
render() {
  return (
    <div>
      <StatusUpdateEntry />
      {this.state.contents.map(function(id) {
          return (
            <FeedItem activeUser={this.props.user} id={id} />
          );
      })}
    </div>
  )
}
```

But... this code is actually incorrect! JavaScript is a weird language with many problems, and we've just encountered one of them. Can you spot it?

...as it turns out, **the value of `this` changes inside of every function**. Our function argument refers to `this.props.user`, but since that function's `this` variable is different from `render`'s `this` variable, it will not return the value we expect.

There are multiple solutions to this dilemma, but the cleanest are JavaScript *arrow functions*. These function work as you'd expect: They inherit the `this` object of whomever defined them. In general, you should use arrow functions whenever you need to construct a one-time function as the argument to another function.

The arrow function version looks like this:

```
render() {
  return (
    <div>
      <StatusUpdateEntry />
```

```
      {this.state.contents.map((id) => {
          return (
            <FeedItem activeUser={this.props.user} id={id} />
          );
      })}
    </div>
  )
}
```

BUT THERE'S STILL ONE MORE ISSUE! Our new `render()` method *dynamically determines* how many `FeedItem` components to make, depending on the data it receives from the database. Whenever a `render()` method dynamically determines the order and number of HTML elements to make, React requires that you specify a `key` attribute *uniquely* identifying those HTML elements among its *siblings*.

By *dynamically determines*, we mean that the decision happens when the `render` methods runs. If a `render` method determines the *order* or *number* of HTML elements dynamically, `keys` are required. The opposite is *statically determines*, which is when `render` returns the same number and order of children each time it runs. You can tell *statically* – that is, by looking at the program text – the order and number of children that it will return every time it runs.

This is a complication of React's Virtual DOM, and its diffing algorithm. React will look at the return value from a `render()` call, and *diff* it with what it previously rendered. If an element in the previous return value and the current return value have the same `key` defined, React will *move* the element to the correct location. If an element in the current return value defines a `key` not seen in the previous return value, React will *create* a new instance of that Component for that element. If an element with a `key` goes missing in a future update, React *destroys* the component and does not reuse it. If you do not specify a `key`, React may re-use your component to render a new component, which may mess up any of the `state` you've set internally.

We did not have this problem before with `CommentThread` because `CommentThread` did not dynamically *create* React components. Instead, it dynamically inserted its children, which we *statically* specified in `FeedItem`, which means React was able to key them properly automatically.

That is a lot of information to absorb, but it comes down to two guidelines:

- **If your `render` method can create different amounts of HTML elements, you need to set a key.**
- **If your `render` method can create HTML elements in multiple orders, you need to set a key.**

…with an emphasis on the word 'create'. Shuffling your children around, whom (oddly enough, from a metaphorical standpoint) some other Component created in its `render` method, is fine. This subtlety is why React refers to the creators of components as *owners* instead of *parents*.

Thankfully, we can use the `FeedItem`'s `_id` property from the database to uniquely identify it. We can also pass the entire `FeedItem` object to the `FeedItem` Component through a prop called `data`:

```
render() {
  return (
    <div>
      <StatusUpdateEntry />
      {this.state.contents.map((feedItem) => {
          return (
            <FeedItem key={feedItem._id} data={feedItem} />
          );
      })}
    </div>
  )
}
```

That's right! Unlike HTML, which only supports strings/numbers, you can pass in arbitrary JavaScript values into components as props.

With the `render()` method written, it's time to talk with the "server"! If we define a method on our component called `componentDidMount`, React will call that method only once – after our component instance has `render()`ed for the first time. This is the perfect place to put a server call.

First, add the following line to the top of `feed.js` to pull in the `getFeedData` function:

```
import {getFeedData} from '../server';
```

Note: Notice how we put {} around `getFeedData`. We did not use `export default` on `getFeedData` in `server.js` – we used `export` on the function without `default`. This is the syntax you use to import non-default items from JavaScript modules.

`getFeedData` takes two arguments:

- The ID of the user whose feed you want to read.
- A callback function, which it will invoke with requested feed as an argument.

Add a `componentDidMount` function to the `Feed` component:

```
componentDidMount() {
  getFeedData(this.props.user, (feedData) => {
    // Note: setState does a *shallow merge* of
    // the current state and the new state. If
    // state was currently set to {foo: 3}, and
    // we setState({bar: 5}), state would then be
    // {foo: 3, bar: 5}. This won't be a problem here.
    this.setState(feedData);
  });
}
```

And now the Feed is all done! Now, we need to slightly adjust the other components so they handle the objects we've defined.

app/component/feeditem.js

`FeedItem` now receives a `FeedItem` object in the `data` prop, so we need to modify it to render the item appropriately. We also want to make it extensible, so we can add different types of `FeedItem`s later if we want (e.g. Advertisements).

Since the `data` prop is a `FeedItem` object, it has a `type` field that specifies what type of item it is. We can `switch` on this value, construct a different React element depending on its contents, and store it into a JavaScript variable. It will need a key, since we are choosing what to construct dynamically. Then, we can use that variable in the final element that we return from `render()`.

Here's a sketch of `FeedItem`'s new `render()` method, which does just that. There are holes we will fill in next:

```
render() {
  var data = this.props.data;
  var contents;
  switch(data.type) {
    case "statusUpdate":
      // Create a StatusUpdate. Dynamically created
      // React component: needs a key.
      // Keys only need to be unique among *siblings*,
      // so we can re-use the same key as the FeedItem.
      contents = (
        <StatusUpdate key={data._id}
                      author={data.contents.author}
                      postDate={data.contents.postDate}
                      location={data.contents.location}>
          {data.contents.contents}
        </StatusUpdate>
      );
      break;
    default:
      throw new Error("Unknown FeedItem: " + data.type);
  }

  return (
    <div className="fb-status-update panel panel-default">
      <div className="panel-body">
        {contents}
        <hr />
```

```
          <div className="row">
            <div className="col-md-12">
              <ul className="list-inline">
                <li>
                <a href="#">
                  <span className="glyphicon glyphicon-thumbs-up">
                  </span> Like</a>
                </li>
                <li>
                  <a href="#">
                  <span className="glyphicon glyphicon-comment">
                  </span>
                    Comment
                  </a>
                </li>
                <li>
                <a href="#">
                  <span className="glyphicon glyphicon-share-alt">
                  </span> Share</a>
                </li>
              </ul>
            </div>
          </div>
        </div>
        <div className="panel-footer">
          <div className="row">
            <div className="col-md-12">
              <a href="#">{like counter count} people</a>
                like this
            </div>
          </div>
          <hr />
          <CommentThread>
            { comments }
          </CommentThread>
        </div>
      </div>
    )
}
```

For the Like counter, recall that a `FeedItem` object has a `likeCounter` property, which has an array of `Users`. We can use replace "like counter count" with `data.likeCounter.length`, since all JavaScript arrays have a `length` property.

For the Comments, recall that Comment components have an `author` and a `postDate` prop. We can construct them using a `map` over the `Comment` objects from the

database. Since our version of Facebook does not support deleting comments from a `CommentThread`, we can use the comment's position in the thread as its `key`:

```
{
data.comments.map((comment, i) => {
  // i is comment's index in comments array
  return (
    <Comment key={i}
             author={comment.author}
             postDate={comment.postDate}>
      {comment.contents}
    </Comment>
  );
})
}
```

app/component/statusupdate.js

We only have to make one small adjustment to the `StatusUpdate` component. Previously, we passed in the *name* of the author as the `author` prop. Now, we are passing a User *object* as the `author` prop. Meaning: `this.props.author` is no longer the name of the user!

Change the `render()` method to use `this.props.author.fullName` instead of `this.props.author`.

app/component/commentthread.js

No changes need to be made to the `CommentThread` class.

app/component/comment.js

Like with `StatusUpdate`, change `this.props.author` to `this.props.author.fullName`.

Fixing Dates

After the above changes, your Facebook clone's feed is now interacting with your mock server, and does not contain any hardcoded mock data!

...but there's one, ugly problem: Dates are just numbers!

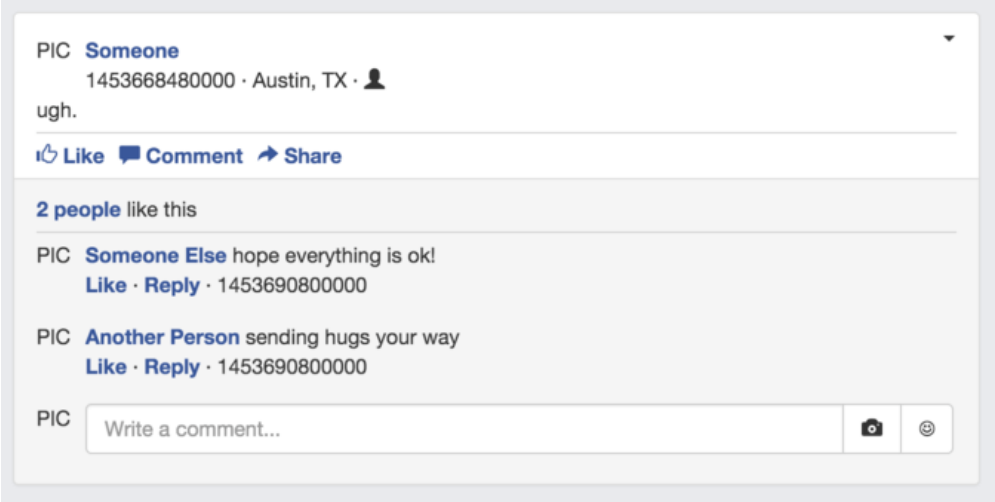

Figure 1: Data from Mock Server

We need to write a utility method that takes in the date in Unix time, and converts it into a suitable string. Facebook is fancy and formats the date differently depending on how far back into the past it is, but we'll settle for an old-fashioned MM/DD/YYYY HH:SS[AM/PM] display.

Fortunately, JavaScript provides a `Date` object, which does just that! There are other packages that are fancier, like Moment.js, but `Date` will work for this workshop.

The documentation for `Date` specifies that the constructor can take a value in Unix time, so let's use that. To format the `Date` object as a string for display in the Feed, we can use the `toLocaleString` method on the object to produce a string for the user's locale (for example, it will flip days and months for people in Europe automatically).

Create the file `app/util.js`, and define the following helper function in it:

```
/**
 * Converts Unix time (in ms since Jan 1 1970 UTC) to a
 * string in the local time zone.
 */
export function unixTimeToString(time) {
  return new Date(time).toLocaleString();
}
```

Then, in `app/components/comment.js`, import this function, and use it on `this.props.postDate`. Do the same thing for `this.props.postDate` for the `StatusUpdate` class.

Now, the Feed will look acceptable:

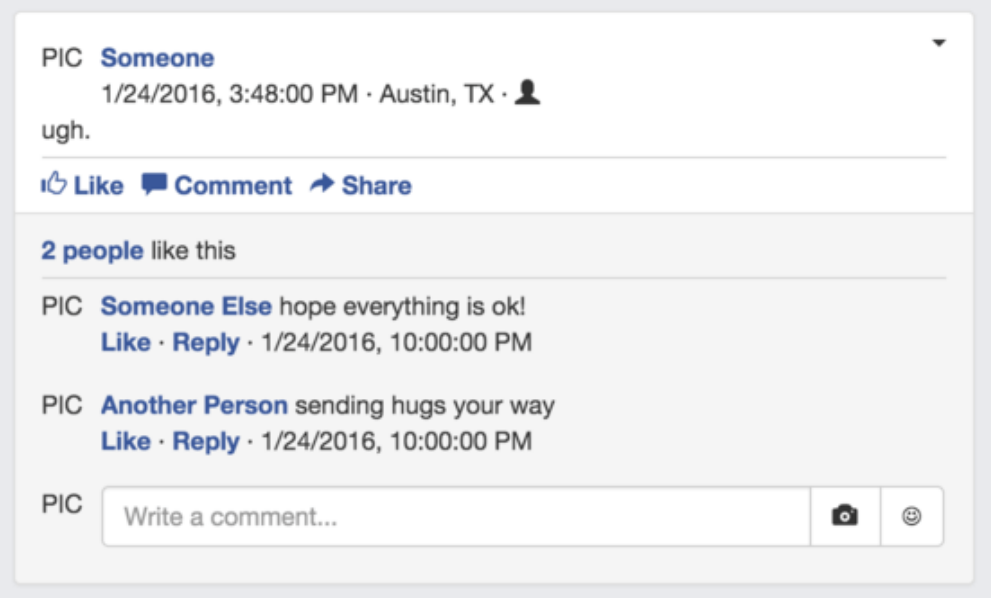

Figure 2: Fixing the Date

And if you ever wanted to spruce up the time display, you can change the helper function to behave differently if the status updates are less than 24 hours old.

Conclusion of Step

You did it! You've gotten through the *hardest two steps* in this workshop. We created a mock server, hooked up the mock server to our frontend, and waded through some complex JavaScript/React subtleties.

Adding interactivity to what you have built in the next step will be a snap!

`add app/util.js`, `commit` with message `fb5`, and `push` to GitHub

Step 06: User Interaction: Status Updates, Comments, & "Like"ing

Although our application is doing a bunch of cool things in the background to pull mock data from a database, it's completely boring without a way to interact with it.

Let's bring the status update entry, "Like" button, and comment boxes to life!

Status Updates

Before we start coding, let's think about the behavior we want.

When the user clicks the "Post" button, our Facebook clone should:

- Check that the post isn't empty. If it is, we ignore the click.
- If the post is not empty:
- Add a new FeedItem object into the database.
- Update the user's Feed object in the database to include the new status update.
  - On real Facebook, this would also probably update friends' feeds as well. But we're not mocking up friends right now.
- Update our Feed to show the new status update.

Let's handle the server-side first. We want a function called `addStatusUpdate` that takes in a user's ID, location, and the contents of the status update, and updates the database.

In `app/server.js`, add the following function, which posts the status update:

```
/**
 * Adds a new status update to the database.
 */
export function postStatusUpdate(user, location, contents, cb) {
  // If we were implementing this for real on an actual server,
  // we would check that the user ID is correct & matches the
  // authenticated user. But since we're mocking it, we can
  // be less strict.

  // Get the current UNIX time.
  var time = new Date().getTime();
  // The new status update. The database will assign the ID for us.
  var newStatusUpdate = {
    "likeCounter": [],
    "type": "statusUpdate",
    "contents": {
      "author": user,
      "postDate": time,
      "location": location,
      "contents": contents
    },
    // List of comments on the post
    "comments": []
  };

  // Add the status update to the database.
  // Returns the status update w/ an ID assigned.
```

```
  newStatusUpdate = addDocument('feedItems', newStatusUpdate);

  // Add the status update reference to the front of the
  // current user's feed.
  var userData = readDocument('users', user);
  var feedData = readDocument('feeds', userData.feed);
  feedData.contents.unshift(newStatusUpdate._id);

  // Update the feed object.
  writeDocument('feeds', feedData);

  // Return the newly-posted object.
  emulateServerReturn(newStatusUpdate, cb);
}
```

For the `StatusUpdateEntry` React component, we want to capture text changes as the user writes a status update, and use them to update the component's `state`. This is what React calls a "Controlled Component", and is the preferred way to structure input controls.

To do this, we will add the `value` variable to the component's `state`, which will track the text the user has entered. Then, we will set the `value` of the `textarea` element to be `this.state.value`. Finally, we will add an *event listener function* to the `textarea` element, which will update `state.value` as the user enters text, which will re-render the `textarea` with the new text.

If you've written JavaScript before, you may object to this manner of handling user input. It sounds dreadfully inefficient and needlessly complex over the usual solution... but it's actually not! JavaScript is *extremely* fast, React's Virtual DOM will apply only a single DOM update each time the user types a character because it uses diffing, and this design avoids using awkward DOM APIs to look up and retrieve values from DOM elements directly. In addition, this design makes it easier to add additional validation or functionality to user input later down the line. (Examples: Search suggestions for searchboxes, URL previews for URLs put into the status update, @ mentions, ...)

With that in mind, let's jump in!

```
1  export default class StatusUpdateEntry extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = {
5        value: ""
6      };
7    }
8
9    /**
10    * Called when the user clicks the 'post' button.
```

```
11      * Triggers the `onPost` prop if the post isn't empty, and clears
12      * the component.
13      */
14     handlePost(e) {
15       // Prevent the event from "bubbling" up the DOM tree.
16       e.preventDefault();
17       // Trim whitespace from beginning + end of entry.
18       var statusUpdateText = this.state.value.trim();
19       if (statusUpdateText !== "") {
20         /* TODO: How do we send the post to the server
21                  + update the Feed? */
22         // Reset status update.
23         this.setState({value: ""});
24       }
25     }
26
27     /**
28      * Called when the user types a character into the status update box.
29      * @param e An Event object.
30      */
31     handleChange(e) {
32       // Prevent the event from "bubbling" up the DOM tree.
33       e.preventDefault();
34
35       // e.target is the React Virtual DOM target of the
36       // input event -- the <textarea> element. The textarea's
37       // `value` is the entire contents of what the user has
38       // typed in so far.
39       this.setState({value: e.target.value});
40     }
41
42     render() {
43       return (
44         <div className="fb-status-update-entry panel panel-default">
45           <div className="panel-body">
46             <ul className="nav nav-pills">
47               <li role="presentation" className="active">
48                 <a href="#">
49                   <span className="glyphicon glyphicon-pencil">
50                   </span> <strong>Update Status</strong></a>
51               </li>
52               <li role="presentation">
53                 <a href="#">
54                   <span className="glyphicon glyphicon-picture">
55                   </span> <strong>Add Photos/Video</strong></a>
```

```
            </li>
            <li role="presentation">
              <a href="#">
                <span className="glyphicon glyphicon-th">
                </span> <strong>Create Photo Album</strong></a>
            </li>
          </ul>
          <div className="media">
            <div className="media-left media-top">
              PIC
            </div>
            <div className="media-body">
              <div className="form-group">
                <textarea className="form-control"
                          rows="2"
                          placeholder="What's on your mind?"
                          value={this.state.value}
                          onChange={(e) => this.handleChange(e)} />
              </div>
            </div>
          </div>
          <div className="row">
            <div className="col-md-6">
              <div className="btn-group" role="group">
                <button type="button"
                        className="btn btn-default">
                  <span className="glyphicon glyphicon-camera">
                  </span>
                </button>
                <button type="button"
                        className="btn btn-default">
                  <span className="glyphicon glyphicon-user">
                  </span>
                </button>
                <button type="button" className="btn btn-default">
                  <span className="glyphicon glyphicon-heart">
                  </span>
                </button>
                <button type="button"
                        className="btn btn-default">
                  <span className="glyphicon glyphicon-pushpin">
                  </span>
                </button>
              </div>
            </div>
```

```
101               <div className="col-md-6">
102                 <div className="pull-right">
103                   <button type="button"
104                           className="btn btn-default">
105                     <span className="glyphicon glyphicon-user">
106                     </span> Friends <span className="caret"></span>
107                   </button>
108                   <button type="button"
109                           className="btn btn-default"
110                           onClick={(e) => this.handlePost(e)}>
111                     Post
112                   </button>
113                 </div>
114               </div>
115             </div>
116           </div>
117         </div>
118       )
119     }
120   }
```

Let's give a tour of what we've added to this class:

- **Line 2-7**: We added a constructor to set an initial value for `state.value`.
- **Line 14**: We added a function that is called when the user clicks the "Post" button.
- `e` is an Event object
- **Line 16**: `e.preventDefault()` prevents the change event from "bubbling" up the DOM tree to other elements. You generally want to call this in all of your event handlers.
- **Line 30**: We added a function that is called when the user types a character into the input field. It updates `state.value`.
- **Line 60**: We changed the `input` element so that its `value` is tied to `state.value`, and added an `onChange` arrow function that triggers `this.handleChange`.
- `onChange` is one of many events that you can subscribe to with a function.
- *Note: Existing React sample code would write 'onChange={this.handleChanged}. This does not work with ES6 classes, so we use an arrow function.*
- **Line 86**: We associated an arrow function with the `onClick` event on the "Post" button, which triggers `handlePost`.

If we wanted to hook up the other buttons (e.g. privacy settings, the different type of posts…), we would add handlers to those buttons which updated the Component's `state`, and then add logic to the `render()` method to display the correct thing depending on `state`.

But how should we actually handle *posting* the new status update? We could communicate with the server in `handlePost`, but we also need to tell the Feed to update itself

once the post finishes.

A simple solution is to add a property (`prop`) to `StatusUpdateEntry` called `onPost`. Like `onChange` and `onClick`, `onPost` will accept a function. We will call the function whenever a status update is posted. The `Feed` component can associate a function with `onPost`, and will handle sending the data to the server and updating itself.

Change Line 21 to `this.props.onPost(statusUpdateText);`.

Now, all we need to do is hook up `Feed` to `onPost`, hook up `onPost` to the server's `postStatusUpdate`, and then trigger the `Feed` to re-fetch its state once the status update posts:

```
1  export default class Feed extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = {
5        contents: []
6      };
7    }
8
9    refresh() {
10     getFeedData(this.props.user, (feedData) => {
11       this.setState(feedData);
12     });
13   }
14
15   onPost(postContents) {
16     // Send to server.
17     // We could use geolocation to get a location,
18     // but let's fix it to Amherst for now.
19     postStatusUpdate(4, "Amherst, MA", postContents, () => {
20       // Database is now updated. Refresh the feed.
21       this.refresh();
22     });
23   }
24
25   componentDidMount() {
26     this.refresh();
27   }
28
29   render() {
30     return (
31       <div>
32         <StatusUpdateEntry
33           onPost={(postContents) => this.onPost(postContents)} />
34         {this.state.contents.map((feedItem) => {
```

```
35            return (
36              <FeedItem key={feedItem._id} data={feedItem} />
37            )
38          })}
39        </div>
40      )
41    }
42  }
```

Notice that we've migrated logic from `componentDidMount` into a `refresh` method.

Now, if you go to http://localhost:8080 and post a status update, it should work!

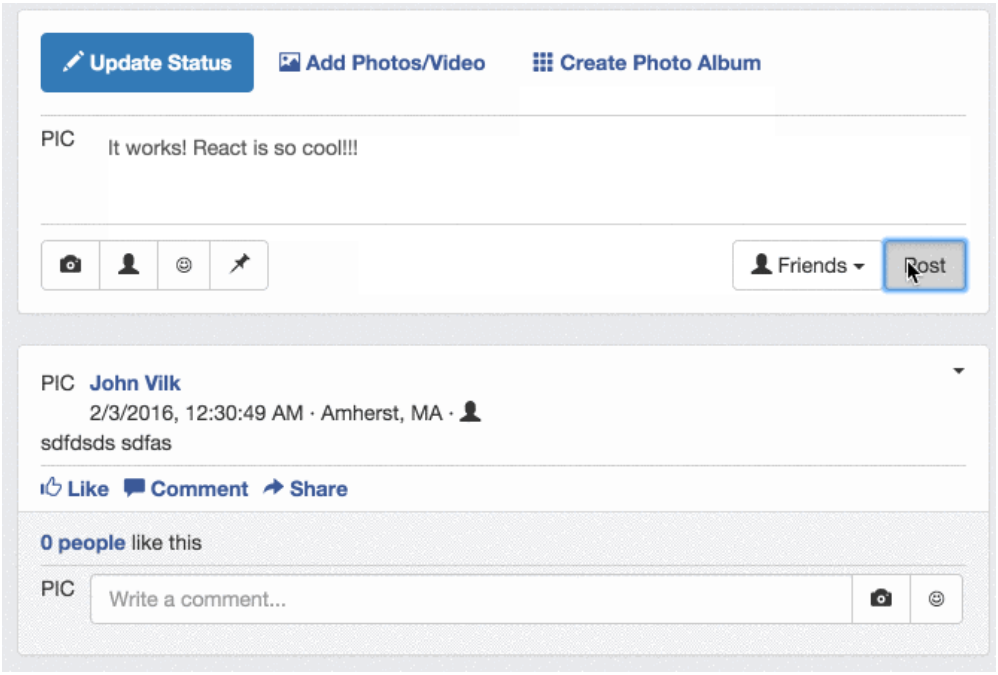

Figure 3: Posting a Status Update

Also, if you refresh the page, or close the browser and return, your "status updates" should remain there!

If something tragic happens, there's always the "Reset Mock DB" button in the left pane that we've added. Clicking that button resets the database to the initial set of mock objects we defined earlier.

Multiline Status Updates

You may notice that multiline status updates don't quite work. They display as a single line! Fortunately, fixing the situation is easy.

In `FeedItem`, change the children of `StatusUpdate` from `data.content.content` to:

```
{data.contents.contents.split("\n").map((line) => {
  return (
    <p>{line}</p>
  );
})}
```

This method takes the status update, and:

- splits the status update on the newline (\n) character.
- so "line 1\nline2\nline3" becomes ["line 1", "line 2", "line 3"]
- maps each line to a `p` element, which begins a new paragraph on a new line.

Now, every new line of a status update should appear on its own line. Try it out! If you Inspect a multiline post on real Facebook using Chrome's Developer Tools, you'll notice that they use `p`, too. It looks like everything is working great!

...but if you open up Chrome's Web Developer Tools, you'll see a warning from React:

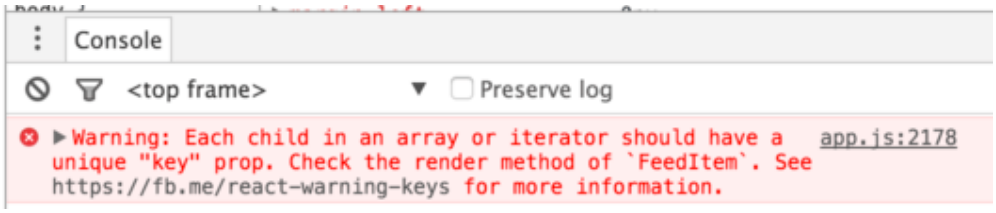

Figure 4: React Key Warning

We're creating a dynamic number of `p` elements, so we need to specify a unique `key` on each.

Change the code to:

```
{data.contents.contents.split("\n").map((line, i) => {
  // Note: 'i' is the index of line in data.contents.contents.
  return (
    <p key={"line" + i}>{line}</p>
  );
})}
```

Now, if you refresh the page, the warning goes away.

Comments

Ideally, when a user posts a comment, our application will only update the item that the user commented on. It does not make too much sense to refresh the entire Feed when the comment thread is localized to one part of the page.

To add a new comment to the Facebook clone, we need to add:

- A server function to add a comment to a post, which returns the updated *FeedItem* object with the new comment.
- Changes to `CommentEntry` that update its `state` in response to user input, similar to our changes to `StatusUpdateEntry`.
- A `onPost` prop on `CommentEntry`, which is triggered when a new comment is posted.
- Changes to `FeedItem` that let it change its contents after a comment is posted.

The server function is relatively straightforward, and reuses a helper function we defined for retrieving FeedItem objects:

```
/**
 * Adds a new comment to the database on the given feed item.
 * Returns the updated FeedItem object.
 */
export function postComment(feedItemId, author, contents, cb) {
  // Since a CommentThread is embedded in a FeedItem object,
  // we don't have to resolve it. Read the document,
  // update the embedded object, and then update the
  // document in the database.
  var feedItem = readDocument('feedItems', feedItemId);
  feedItem.comments.push({
    "author": author,
    "contents": contents,
    "postDate": new Date().getTime()
  });
  writeDocument('feedItems', feedItem);
  // Return a resolved version of the feed item so React can
  // render it.
  emulateServerReturn(getFeedItemSync(feedItemId), cb);
}
```

Since this server function updates the comment thread *and* returns the updated FeedItem object, our React code only needs to perform a single server request!

The changes to `CommentEntry` are mostly simple, but note that a comment gets entered when the user hits the "enter" key. The `onChanged` event, which we used for `StatusUpdateEntry`, only fires when the user does something that would ordinarily change the visual state of the input box – that is, the text contained within it. A

one-line text box does not change its value when the "enter" key is hit, so `onChanged` does not fire when "enter" is hit. We will have to use a different event to capture this key!

Fortunately, we can register a function with the `onKeyUp` event. This event fires whenever the user stops hitting a key on the keyboard (`onKeyDown` fires when the user first hits the key, `onKeyPress` fires while the user is *holding down* the key, and `onKeyUp` fires when the user lifts his finger off of the key.) We can check if the key is the "enter" key, and submit the comment to the server if it is non-empty.

The resulting code is as follows:

```
import React from 'react';

export default class CommentEntry extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: ""
    };
  }

  handleChange(e) {
    this.setState({ value: e.target.value });
  }

  handleKeyUp(e) {
    if (e.key === "Enter") {
      var comment = this.state.value.trim();
      if (comment !== "") {
        // Post comment
        this.props.onPost(this.state.value);
        this.setState({ value: "" });
      }
    }
  }

  render() {
    return (
      <div>
        <div className="media-left media-top">
          PIC
        </div>
        <div className="media-body">
          <div className="input-group">
            <input type="text"
```

```
35                      className="form-control"
36                      placeholder="Write a comment..."
37                      value={this.state.value}
38                      onChange={(e) => this.handleChange(e)}
39                      onKeyUp={(e) => this.handleKeyUp(e)} />
40                <span className="input-group-btn">
41                  <button className="btn btn-default" type="button">
42                    <span className="glyphicon glyphicon-camera"></span>
43                  </button>
44                  <button className="btn btn-default" type="button">
45                    <span className="glyphicon glyphicon-heart">
46                    </span>
47                  </button>
48                </span>
49              </div>
50            </div>
51          </div>
52      )
53    }
54  }
```

Some quick notes:

- **Line 15:** The `e` object for `handleKeyUp` is a `KeyboardEvent` object
- **Line 16:** While the documentation for `KeyboardEvent.key` states that it is not supported in many browsers, React patches `KeyboardEvent` so it works.

One restriction of our current design is that we do not support multiline comments. While we are not going to perform that change in this workshop, you could support multiline comments by:

- Replacing the `input` box with a one-row-high `textarea`.
- Adding logic to support linebreaks when the user hits "shift+enter", and entering the comment then the user hits "enter".
- The `KeyboardEvent` object has a `shiftKey` property that you can check.
- Changing `render()` to wrap lines of text in keyed `<p>` elements.

Next, we need to modify `CommentThread` so it passes a function to `CommentEntry`'s `onPost` property. We can add an `onPost` property to `CommentThread`, and pass its value along to `CommentEntry`:

```
export default class CommentThread extends React.Component {
  render() {
    return (
      <ul className="media-list">
        {React.Children.map(this.props.children, function(child) {
          return (
            <li className="media">
```

```
              {child}
            </li>
          )
        })}
        <li className="media">
          <CommentEntry onPost={this.props.onPost} />
        </li>
      </ul>
    )
  }
}
```

Finally, we need to change `FeedItem` to use our new server method, and to provide a `onPost` function to `CommentThread`. Currently, `FeedItem` uses its `data` property to render itself. A React component should never change its properties, but we need to change the data once a new comment is posted!

To support this functionality, we need to change the `FeedItem` to read from its *state* when it renders, and initialize the state to the JSON object passed in via the `data` property:

```
export default class FeedItem extends React.Component {
  constructor(props) {
    super(props);
    // The FeedItem's initial state is what the Feed passed to us.
    this.state = props.data;
  }

  handleCommentPost(commentText) {
    // Post a comment as user ID 4, which is our mock user!
    postComment(this.state._id, 4, commentText, (updatedFeedItem) => {
      // Update our state to trigger a re-render.
      this.setState(updatedFeedItem);
    });
  }

  render() {
    // Render using data from state.
    var data = this.state;
    // The rest of this method is the same as before!
    // Except you need to change the CommentThread element
    // to hook up an onPost function:
    // <CommentThread onPost={(commentText) => this.handleCommentPost(commentText)}>
  }
}
```

Now, if you refresh `http://localhost:8080`, you'll be able to post comments as "John

Vilk"!

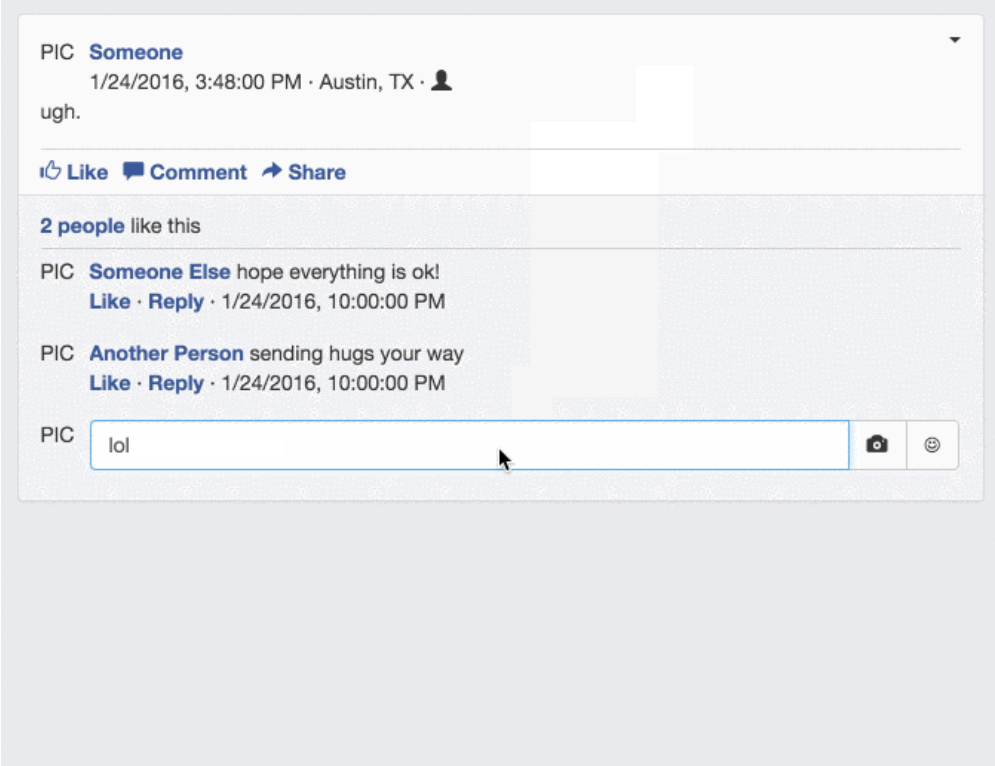

Figure 5: Facebook Comment

"Like" Button

Next, we need to hook up functionality to the "Like" button. The "Like" button will let users "Like" and "Unlike" items on Facebook, so we need two server methods: One for "Like", and another for "Unlike".

Like with `CommentEntry`, we will have these functions return an updated `likeCounter`, so our application only needs to make one server request when the user hits "Like":

```
/**
 * Updates a feed item's likeCounter by adding the
 * user to the likeCounter. Provides an updated likeCounter
 * in the response.
 */
export function likeFeedItem(feedItemId, userId, cb) {
  var feedItem = readDocument('feedItems', feedItemId);
```

```
  // Normally, we would check if the user already
  // liked this comment. But we will not do that
  // in this mock server. ('push' modifies the array
  // by adding userId to the end)
  feedItem.likeCounter.push(userId);
  writeDocument('feedItems', feedItem);
  // Return a resolved version of the likeCounter
  emulateServerReturn(feedItem.likeCounter.map((userId) =>
                        readDocument('users', userId)), cb);
}

/**
 * Updates a feed item's likeCounter by removing
 * the user from the likeCounter.
 * Provides an updated likeCounter in the response.
 */
export function unlikeFeedItem(feedItemId, userId, cb) {
  var feedItem = readDocument('feedItems', feedItemId);
  // Find the array index that contains the user's ID.
  // (We didn't *resolve* the FeedItem object, so
  // it is just an array of user IDs)
  var userIndex = feedItem.likeCounter.indexOf(userId);

  // -1 means the user is *not* in the likeCounter,
  // so we can simply avoid updating
  // anything if that is the case: the user already
  // doesn't like the item.
  if (userIndex !== -1) {
    // 'splice' removes items from an array. This
    // removes 1 element starting from userIndex.
    feedItem.likeCounter.splice(userIndex, 1);
    writeDocument('feedItems', feedItem);
  }
  // Return a resolved version of the likeCounter
  emulateServerReturn(feedItem.likeCounter.map((userId) =>
                        readDocument('users', userId)), cb);
}
```

Next, we need to change `FeedItem` so that clicking the "Like" button triggers a server event. We also need to change the `render()` method so that the "Like" button switches to a "Unlike" button if the user has already liked the item.

```
  /**
   * Triggered when the user clicks on the 'like'
   * or 'unlike' button.
   */
```

```
handleLikeClick(clickEvent) {
  // Stop the event from propagating up the DOM
  // tree, since we handle it here. Also prevents
  // the link click from causing the page to scroll to the top.
  clickEvent.preventDefault();
  // 0 represents the 'main mouse button' --
  // typically a left click
  // https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/button
  if (clickEvent.button === 0) {
    // Callback function for both the like and unlike cases.
    var callbackFunction = (updatedLikeCounter) => {
      // setState will overwrite the 'likeCounter'
      // field on the current state, and will keep
      // the other fields in-tact. This is called a
      // shallow merge:
      // https://facebook.github.io/react/docs/component-api.html#setstate
      this.setState({likeCounter: updatedLikeCounter});
    };

    if (this.didUserLike()) {
      // User clicked 'unlike' button.
      unlikeFeedItem(this.state._id, 4, callbackFunction);
    } else {
      // User clicked 'like' button.
      likeFeedItem(this.state._id, 4, callbackFunction);
    }
  }
}

/**
 * Returns 'true' if the user liked the item.
 * Returns 'false' if the user has not liked the item.
 */
didUserLike() {
  var likeCounter = this.state.likeCounter;
  var liked = false;
  // Look for a likeCounter entry with userId 4 -- which is the
  // current user.
  for (var i = 0; i < likeCounter.length; i++) {
    if (likeCounter[i]._id === 4) {
      liked = true;
      break;
    }
  }
  return liked;
```

```
}

render() {
  var likeButtonText = "Like";
  if (this.didUserLike()) {
    likeButtonText = "Unlike";
  }
  // Skipping the first part of this method, which is unchanged.
      <a href="#" onClick={(e) => this.handleLikeClick(e)}>
        <span className="glyphicon glyphicon-thumbs-up"></span>
          {likeButtonText}
      </a>
  // The rest of this method is unchanged
}
```

Now, you can "Like" status updates!

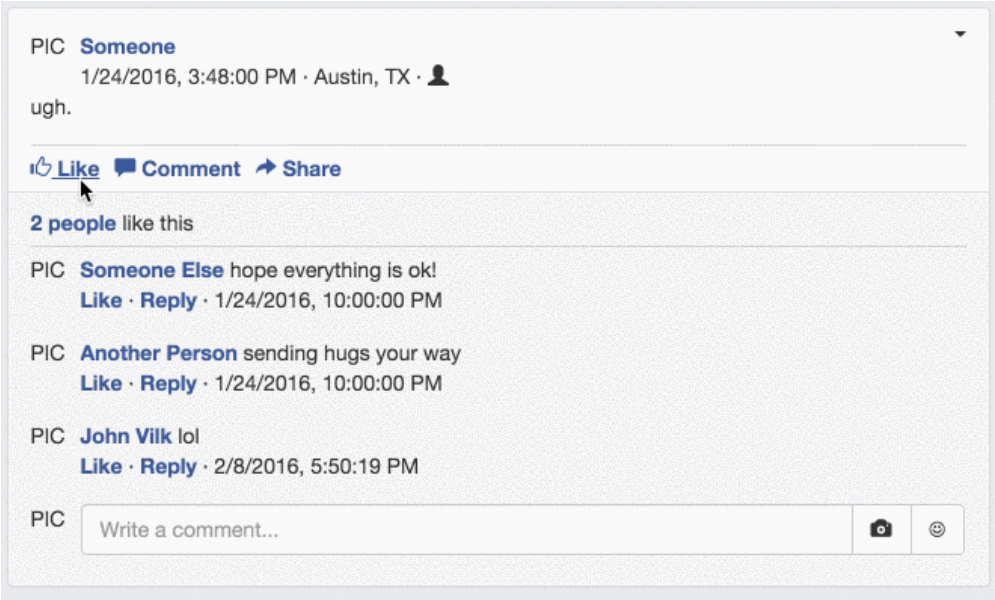

Figure 6: Facebook Like

With this change in place, you've completed step 6!

`commit` your changes with message `fb6`, and `push` them to GitHub.

Step 07: Supporting Multiple "Pages"

Right now, we've covered enough of React to have a working, interactive Facebook Feed. What we have not talked about so far is supporting *multiple pages* in our Facebook application.

One way to support multiple pages is to have multiple HTML files… but since we put most of our application's HTML into React Component, that does not make sense for this course.

Another, more modern way to support multiple pages is to use a *client-side routing library* to render the correct page. A routing library uses the data in #-based links to direct your application to render the appropriate content. For example, the url `http://localhost:8080/#/profile/1` could contain a user's profile. The benefit of a URL that begins with `#` is that it does not cause the browser to reload the HTML file, which lets your application respond to the request immediately.

Now that you know about them, you'll notice `#`-based links all over the web. For example, Google's homepage is https://www.google.com/webhp?hl=en . If you enter a search query, like "nachos", you get directed with the google URL ending with ?hl=en#hl=en&q=nachos. Since the URL contains a `#`, Google didn't have to reload the page to give you your search results. But because the URL changed, I can now link you to the search results for nachos! Mmm…

There is an excellent routing library for React programs called `react-router`. You tell it what `#` URLs point to what React components, and it does the rest!

Let's add a simple "profile page" to our application. The URL:

`http://localhost:8080/#/profile/1`

will contain the profile page for user with ID 1. To keep the workshop short, the page will be nothing fancy; it will only tell you what the user's ID is.

Open up `app/app.js`, and change it to add the following code. We'll define the missing Components in a bit:

```
import React from 'react';
import ReactDOM from 'react-dom';
import Feed from './components/feed';
import { IndexRoute, Router, Route, browserHistory } from 'react-router'

ReactDOM.render((
  <Router history={browserHistory}>
    <Route path="/" component={App}>
      {/* Show the Feed at / */}
      <IndexRoute component={FeedPage} />
      <Route path="profile/:id" component={ProfilePage} />
    </Route>
```

```
  </Router>
),document.getElementById('fb-feed'));
```

React-router's configuration is nothing but a set of special React components. This configuration says:

- At `/` (`http://localhost:8080/`), render `<App><FeedPage /></App>`
- At `/profile/(ID here)` (http://localhost:8080/#/profile/3 for user 3), render `<App><ProfilePage param={ {id: (ID here) } } /></App>`
- The `:id` notation tells React Router to stash that part of the URL into a field named `id` on a component property named `param`

There's a helpful guide to react-router, which links to many examples.

`App`, `ProfilePage`, and `FeedPage` are very simple components. Add these to `app.js`, above the call to `render()`:

```
/**
 * A fake profile page.
 */
class ProfilePage extends React.Component {
  render() {
    return (
      <p>This is the profile page for a user
         with ID {this.props.params.id}.</p>
    );
  }
}

/**
 * The Feed page. We created a new component just
 * to fix the userId at 4.
 */
class FeedPage extends React.Component {
  render() {
    return <Feed user={4} />;
  }
}


/**
 * The primary component in our application.
 * The Router will give it different child Components
 * as the user clicks around the application.
 *
 * If we implemented all of Facebook, this App would
 * also contain Component objects for the left and
```

```
 * right content panes.
 */
class App extends React.Component {
  render() {
    return (
      <div>{this.props.children}</div>
    )
  }
}
```

Now, if you go to `http://localhost:8080/`, you'll see the Facebook feed. If you go to `http://localhost:8080/#/profile/2`, you'll see a "profile page" for user 2.

Let's update the `StatusUpdate` and `Comment` components so they link to the profile page. In both files, add the following to the top of the file:

```
import {Link} from 'react-router';
```

In `app/components/statusupdate.js`, change:

```
<a href="#">{this.props.author.fullName}</a>
```

...into:

```
<Link to={"/profile/" + this.props.author._id}>
  {this.props.author.fullName}
</Link>
```

Next, in `app/components/comment.js`, change:

```
<a href="#">{this.props.author.fullName}</a>
```

...into:

```
<Link to={"/profile/" + this.props.author._id}>
  {this.props.author.fullName}
</Link>
```

If you reload `http://localhost:8080/` and click on someone's name in a comment or status update, you'll be brought to their "profile page"!

Hopefully, this simple example gives you an idea of how your startup product can grow to support multiple interactive pages.

`commit` your changes with message `fb7`, and `push` them to GitHub.

Step 08: Add Support for the Like Button on Comments

Now that we've done all this, it's time for you to apply the skills you've learned! We've hooked up a "Like" button on feed items, but now you need to add a "Like" button

to comments.

To do this, you'll need to:

- Add a `likeCounter` to Comment objects in the database.
- Update the comment-related server methods so the Comment objects they create properly contain the `likeCounter` field.
- Add a server method that lets a user like/unlike particular comments. It can return an updated comment object.
- To uniquely identify a comment, you can supply the ID of the FeedItem and the index of the comment.
- Render "Like" or "Unlike" on comments, depending on if the current user clicked the button.
- Trigger the appropriate server method when the user clicks the "Like"/"Unlike" button, and update the comment object.
- You'll either have to pass this information down to the comment, *or* pass the "like" click up to the FeedItem via a callback function. Your choice.

You may want to click the "Reset Mock DB" button when you make the database modification to remove old data from the database.

`commit` your changes with message `fb8`, and `push` them to GitHub.

With this complete, you've finished the workshop!

Tips & Tricks

Is something not working? Open up the Chrome Development Tools, and look at your console. Chrome will tell you when bad things are happening, and will link you to lines of your source code where the problem occurred.

Remember to try resetting the mock database if something weird is happening. If that fixes the problem, then you may have a bug in a server function!

Overview & ProTips

This workshop covered a lot of ground!

We started with mock data in the HTML directly. Then we moved it into React components directly. Then we moved it into React component properties. Then we moved it into a mock data store, with access via a mock server!

You've covered all of the skills needed to make a dynamic mockup of your application. Next time, we'll talk more about servers, which power things underneath your application.

ProTips

Here are some of the JavaScript and React subtleties that we went over in this workshop and the previous workshop.

JavaScript Modules:

- You can `export` multiple named items from a module.
- Examples: `export function foo() {};`, `export var bar = 4;`, `export class Baz { }`
- You can `import` named items from a module with `import { item1, item2, ... } from '../path/to/module';`
- Example: `import {foo, bar, baz} from './bat.js`;
- You can `export default` one item per module.
- Example: `export default class Foo { }`
- You can `import` a `default` export of another module with `import Foo from './bar';`
- NPM modules are `import`ed without a filesystem path; only use the name of the module.
- Example: `import {Router} from 'react-router';` It would be invalid to import from `'./react-router'`!

JavaScript:

- `function`s change the value of `this`. Use arrow functions when you need to pass a function as an argument to another function, as they preserve the value of `this`.

React components:

- Change `class` properties to `className`.
- Always include a closing tag on HTML elements in your React code.
- Don't modify `props`. `props` get set only once.
- Don't modify `state` directly. Use `setState`.
- ...except in the constructor. There, you can set `state` directly to its initial value.
- `setState` performs a *shallow merge* on the value you give it and the existing state.
- Example: If `state` is `{foo: 3}` and you call `setState({ bar: 5 })`, `state` will become `{ foo: 3, bar: 5}`.
- Set an `id` on HTML elements in `render()` when their order or number depend on data at runtime.
- The `id` only needs to be unique among elements returned from that `render()` function.
- When iterating over children, use `React.Children` methods

Submission

You must submit the URL of your **Workshop4and5** GitHub repository to Moodle. Visit Moodle, find the associated Workshop 5 activity, and provide your URL. Make sure your **Workshop4and5** repository is public so we can clone your repository and evaluate your work. **Submitting the URL for this assignment is part of completing the work.**