Implementing a RESTful Webservice API with Node.js

Introduction

In this assignment we are going to practice building a RESTful Webservice API with Node.js. Up to this point we've been working with a React application running on a browser. JavaScript applications running on browsers are great for creating dynamic user interfaces, but are limited in that they have no place to permanently store their application state as users interact with them. JavaScript Node.js applications can run outside the browser, on desktop machines, and have unfettered access to the file system, databases and network communication. They can complement React applications nicely. We are going to learn how to integrate JavaScript React applications running on a browser, with JavaScript applications running on a Node.js server.

Practice

To begin, copy ***/src/components/a7*** to ***/src/components/a8*** and modify ***App.js*** so all React assignment practice and build implementations are accessible. Confirm that all the practice examples and twitter screens still work.

Create a Node Application

To create a Node application, create a directory to contain the application, e.g., ***web-dev-node*** as shown below.

```
mkdir  web-dev-node     # create a new directory for the Node.js project
cd  web-dev-node        # change into the new directory
```

To initialize the project use ***npm***'s ***init*** command to create the project's ***package.json***. The ***package.json*** file contains the project's dependencies such as a list of other libraries the project needs to function, plus meta data such as the author and description. From within the project's root directory, use the ***init*** command as shown below.

```
npm  init
```

Accept all the default values by pressing ***Enter*** or ***Return*** key. List the content of the directory and confirm that you have a new ***package.json*** file.

Create a Hello World Express application

Node.js is a JavaScript interpreter that has become very popular and now has a large open source community sharing lots of useful libraries and tools written for the community. One of the most popular libraries, or modules, is the Express library which simplifies the creation of middleware services. We're going to use express to create an HTTP server to integrate with the React application we've been working on. To get started

we'll first install the library in our project. From the root directory of the project, use npm's install command as shown below. Once the library has installed, we can begin to implement the HTTP server.

```
npm install express
```

Open the project with IntelliJ (or your favorite IDE) and create a new file called ***server.js*** at the root of the project withh the following content. The JavaScript file creates an instance of the express library and uses it to declare an HTTP endpoint that listens on port 4000 for HTTP GET requests with the "/hello" URL pattern. If we point our browsers to the URL, the server will respond with "Hello World!".

```
const express = require('express');     // load the express library
const app = express();                  // create an instance of the library

app.get('/hello', (req, res) => {       // use express library to listen for URL pattern "/hello"
  res.send('Hello World!');             // respond with string "Hello World!"
});

app.listen(4000);                       // listen to port 4000
```

To start the server use ***node*** to run the ***server.js*** file as shown below. The server will listen indefinitely for HTTP requests on port 4000. Point your browser at http://localhost:4000/hello and confirm the server responds with ***Hello World!*** Stop the server with ***Ctrl+C***.

```
node  server.js                         // use node to run server.js which will
```

Configuring Cross Origin Resource Sharing (CORS)

The server is listening at port 4000 for HTTP requests. For security purposes, by default, servers will reject requests not coming from the same IP address, domain, or port. Since our React application runs on a Node server on port 3000, the server will refuse HTTP requests from our React application. The two applications are running on different origins and requests across origins is not allowed for security purposes. This policy is referred to as ***Cross Origin Resource Sharing*** (***CORS***). We can configure servers to allow requests from origins we know we can trust. The following configures ***CORS*** to allow requests from any origin. Later we'll narrow this to a specific URL when we deploy to remote servers such as ***Netlify*** or ***Heroku***.

```
const express = require('express');
const app = express();
app.use(function(req, res, next) {                          // intercept all HTTP requests with express library
  res.header("Access-Control-Allow-Origin", "*");           // allow requests from anywhere
  res.header("Access-Control-Allow-Headers",                // accept conten type set from request
     "Origin, X-Requested-With, Content-Type, Accept");
  res.header("Access-Control-Allow-Methods",                // allow HTTP request methods:
     "GET, POST, PUT, DELETE, OPTIONS");                    // GET, POST, PUT, DELETE, OPTIONS
  res.header("Access-Control-Allow-Credentials", "true");   // allow cookies for login
  next();                                                   // allow request to continue
});
app.get('/hello', (req, res) => {
```

```
  res.send('Hello World!');
});
```

Hello World React Client

Let's create a React component that can connect to the server we just created and retrieve the hello message at http://localhost:4000/hello. Copy the code below in ***/Practice/APIExamples/HelloApiClient.js***.

```
import React, {useEffect, useState} from "react";     // to handle local state variables such as hello below
const HelloApiClient = () => {
   const [hello, setHello] = useState('');            // declare local state variable and mutator
   useEffect(() => {                                  // when the component first loads
      fetch('http://localhost:4000/hello')            // send an HTTP request to this URL
         .then(response => response.text())           // parse the text in the HTTP response from server
         .then(text => setHello(text));               // set hello state variable with text from server
   }, []);                                            // don't force re-render because state changed
   return (
      <h1>{hello}</h1>                                // render hello message
   );
};
export default HelloApiClient;
```

Create a component where we can put all our API examples. Copy the code below into ***/Practice/APIExamples/index.js***.

```
import React from "react";
import HelloApiClient from "./HelloApiClient";       // load HelloApiClient declared above
const APIExamples = () => {
   return(
      <div>
         <HelloApiClient/>                           // just render HelloApiClient for now, more to come
      </div>
   );
};
export default APIExamples;
```

Include the new ***HelloApiClient*** component in the ***Practice*** component as shown below in ***/Practice/index.js***. Restart both the server and React application and confirm the ***Practice*** screen displays the ***Hello World!*** message from the server.

```
import APIExamples from "./APIExamples";             // import the APIExamples component
const Practice = () => {
   return(
      <div>
         <h1>Practice</h1>
         <Link to="/a8/twitter/home">Build</Link>
         <APIExamples/>                              // render the new component
         <ReduxExamples/>
```

```
      </div>
    )
};
export default Practice;
```

Interacting with REST APIs with React

React is great for creating dynamic front end applications easily deployed to anyone who has a browser that can run JavaScript, but because they run in a browser, they are limited by the security constraints of the browser such as no file access and limited network access. Node applications don't have any of the security constraints of React applications since they don't run in a browser and have full access to the services the operating system provides such as unfettered access to the file system, networks, databases etc. But because they don't run on a browser. Together, React and Node applications, they can complement their strengths and weaknesses. They can exchange HTTP messages to integrate into a single, seamless application. In this section we'll practice how to integrate React and Node. The React and Node applications are two halves of the same application client server application. React being the client, and Node being the server. The integration typically consists of implementing HTTP endpoints the React application can connect to. The endpoints implement tical use cases such as creating data, retrieving data, updating data, and deleting data. We usually refer to these operations as CRUD (Create Read Update Delete). The next several sections cover how to implement each of these operations to integrate a React and Node application that manage a simple data set.

Retrieving Data from an API

To practice retrieving data from a REST API, let's implement a simple movie service that manages a list of movies as an array of objects. The movies will be accessible through an HTTP GET request to URL ***/api/movies***. In the Node project, copy and paste the code below into a new file in ***/services/movies-service.js***.

```
let movies = [                                          // declare favorite movies
   {_id: '123', title: 'Aliens', rating: 4.5},
   {_id: '234', title: 'Terminator', rating: 4.8},
   {_id: '345', title: 'Avatar', rating: 4.7}
];
module.exports = (app) => {
   const getAllMovies = (req, res) => res.json(movies);    // respond with movies array
   app.get('/api/movies', getAllMovies);                   // listen for /api/movies URL request
};
```

Towards the end of ***server.js***, load the new movies service and pass it a reference of the express library.

```
require('./services/movies-service')(app);              // pass express instance to service

app.listen(4000);
```

In the React project, let's create a React client to retrieve the movies from the API and render them in an HTML list element. Copy and paste the code below in a new file ***APIExamples/MovieApiClient.js***.

```
import React, {useEffect, useState} from "react";
const MovieApiClient = () => {
   const [movies, setMovies] = useState([]);          // declare empty array local state variable movies
   useEffect(() =>                                    // when the component first loads
      fetch('http://localhost:4000/api/movies')       // send an HTTP request to this URL
         .then(response => response.json())            // parse the JSON in the HTTP response from server
         .then(movies => setMovies(movies))            // set ***movies*** state variable with movies from server
      , []);                                           // don't force re-render because state changed
   return(
      <div>
         <h2>Movies</h2>
         <ul className="list-group">                   // declare list
            {
               movies.map((movie) =>                   // iterate over movies array, for each movie
                  <li className="list-group-item"      // append a list item
                     key={movie._id}>                  // use movie's id as element's key
                     {movie.title} {movie.rating}      // render movie's title and stars
                  </li>
               )
            }
         </ul>
      </div>
   )
};
export default MovieApiClient;
```

In ***APIExamples***, import ***MovieApiClient*** as shown below. Restart the server and React applications and confirm the list of movies renders.

```
import React from "react";
import HelloApiClient from "./HelloApiClient";
import MovieApiClient from "./MovieApiClient";      // import MovieApiClient
const APIExamples = () => {
   return(
      <div>
         <MovieApiClient/>                          // render list of movies from the server
         <HelloApiClient/>
      </div>
   );
};
export default APIExamples;
```

Deleting Data from an API

Let's now practice removing a movie using the API. First we'll define the API in the movies service as shown below. Copy and paste the code below into ***/services/movie-service.js***.

```
   const deleteMovie = (req, res) => {                // handle delete HTTP request
      const id = req.params['mid'];                   // read mid parameter from the URL
      movies = movies.filter(m => m._id !== id);      // filter out movie whose ID is mid
```

```
    res.json(movies);                          // respond with movies with missing deleted movie
  };
  app.delete('/api/movies/:mid', deleteMovie); // invoke handler if delete HTTP request
```

Update ***APIExamples/MovieApiClient.js*** by adding a delete button to each movie in the list and when clicked, notify the server to delete the movie. The server will respond with the updated movie list. Restart the server and React application and confirm you can delete movies from the list. Refresh the browser after deleting a movie to confirm the list of movies is persisted on the server. You can reset the list of movies by restarting the server.

```
const deleteMovie = (movie) =>                          // handle delete button click
  fetch(`http://localhost:4000/api/movies/${movie._id}`, {  // fetch movies from API on server
    method: 'DELETE'                                    // use DELETE HTTP method
  })
    .then(response => response.json())                  // parse JSON response from server
    .then(movies => setMovies(movies));                 // set local movies with movies from server
...
<li className="list-group-item" key={movie._id}>
  {movie.title} {movie.rating}
  <button onClick={() => deleteMovie(movie)}            // new delete button passes movie handler
          className="btn btn-danger float-end">
    Delete
  </button>
</li>
```

Sending Data to an API

Let's now practice sending data to an API. The data coming and going to and from the server and the client applications is formatted as ***JSON*** (***JavaScript Object Notation***). As shown in earlier examples, Node express services can respond with JSON, but unfortunately they wont know what to do with JSON data we might send it. We'll need to rely on an additional library that knows how to parse a JSON string into an equivalent JSON object instance. In previous examples we used HTTP GET and DELETE to retrieve and remove data from the server. We could use HTTP GET to send data to the server by encoding the data as part of the URL, but these have a size limit and are generally not secure. A better option is to use HTTP POST and encode the data in the HTTP message's ***body***.

Configuring Body Parser

Let's install the body parser library so we can parse JSON strings from the body of HTTP messages. Install the body parser library as follows.

```
npm i body-parser
```

In ***server.js***, load and configure the body parser library right after loading the express library, as shown below.

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');              // load body parser library
```

```
app.use(bodyParser.urlencoded({ extended: false }));    // encode special characters
app.use(bodyParser.json());                             // register JSON body parser middleware
...
```

In ***movies-service.js***, configure an HTTP POST handler to process a '/api/movies' URL pattern. The handler will parse the movie from the request's body and append it to the movies array. The array will be send back to the client as a response.

```
const createMovie = (req, res) => {             // handler for HTTP POST to '/api/movies' URL
   const movie = req.body;                      // parse movie from HTTP request's body
   movies = [...movies, movie];                 // append movie to end of movies array
   res.json(movies);                            // respond with movies array
}
app.post('/api/movies', createMovie);           // listen for HTTP POST to '/api/movies' URL, notify handler
```

On the clilent side, add the following

```
const MovieApiClient = () => {                                  // initialize local movie state variable
const [movie, setMovie] = useState({title: '', rating: 2.5});   // handle movie title input field change
const onMovieTitleChange = (event) =>                           // update local state variable movie's title
   setMovie({...movie, title: event.target.value});             // handle create movie click event
const createMovieClickHandler = () =>                           // send HTTP message to server URL /api/movies
   fetch('http://localhost:4000/api/movies', {                  // use HTTP POST
      method: 'POST',                                           // embed movie in message body as a string
      body: JSON.stringify(movie),                              // tell server string is formatted
      headers: {                                                // as JSON
         'content-type': 'application/json'
      }
   })                                                           // parse movies as JSON array from server
      .then(response => response.json())                        // response and update local movies array copy
      .then(movies => setMovies(movies));
```

Further down in the same ***MovieApiClient*** component, add a ***Create*** button to create a movie, and an input field for the ***title*** as shown below.

```
const MovieApiClient = () => {
...
return(
   <div>
      <h2>Movies</h2>
      <ul className="list-group">
         <li className="list-group-item">
            <button                                             // create movie button
               onClick={createMovieClickHandler}                // notify handler when clicked
               className="btn btn-success float-end">
               Create
            </button>
            <input className="form-control"                     // movie title text input field
                  value={movie.title}                           // value bound to local movie object's title
```

```
                onChange={onMovieTitleChange}              // notify handler when field value changes
                style={{width: "70%"}}/>
        </li>
```

Updating an API

```
const saveMovie = (req, res) => {                          // handler for HTTP PUT to '/api/movies/:mid' URL
   const newMovie = req.body;                              // parse movie from HTTP request's body
   const movieId = req.params['mid'];                      // parse movieId from HTTP request parameters
   movies = movies.map(movie =>                            // recreate movies array
      movie._id === movieId ? newMovie : movie);           // replace old movie with new version
   res.json(movies);                                       // respond with movies array
}
app.put('/api/movies/:mid', saveMovie);                    // listen for HTTP PUT to '/api/movies' URL, notify handler
```

Add an ***Edit*** button for each of the movies to select the movie to edit as shown below. Clicking the button sets the current movie state variable which renders in the movie title text field implemented earlier.

```
{
   movies.map((movie) =>
      <li className="list-group-item" key={movie._id}>
         {movie.title} {movie.stars}
         <button onClick={() => setMovie(movie)}
               className="btn btn-primary float-end ms-2">
            Edit
         </button>
```

Add a ***Save*** button next to the movie title input field that invokes a ***saveMovie*** function as shown below.

```
<ul className="list-group">
   <li className="list-group-item">
      <input className="form-control"
            value={movie.title}
            onChange={onMovieTitleChange}
            style={{width: "70%"}}/>
      <button
         onClick={saveMovie}
         className="btn btn-primary ms-2 float-end">
         Save
      </button>
```

Right before the return statement, implement the ***saveMovie*** function shown below. The function should send an HTTP PUT message to the ***saveMovie*** HTTP Webservice endpoint created earlier in this section. The ID of the movie is appended at the end of the URL for the Webservice to identify which movie to update. The updated movie is embedded as a string in the body of the message. Restart the client and server and confirm you can edit the movie titles.

```
const saveMovie = () =>
   fetch(`http://localhost:4000/api/movies/${movie._id}`, {
      method: 'PUT',
      body: JSON.stringify(movie),
      headers: {
         'content-type': 'application/json'
      }
   })
      .then(response => response.json())
      .then(movies => setMovies(movies));
return(
   <div>
      <h2>Movies</h2>
```

Build

The implementation of our Twitter clone thus far uses data files such as ***who.json*** and ***tweets.json*** hosted in the client React application. Creating new tweets and profile changes are stored in an application state managed through reducers. These changes are not permanent and are lost if the browser reloads. The only way for the React client application to store state changes permanently is to rely on server integration. For the build part of this assignment, we will integrate the current Twitter React client application with a new set of Webservices to store new tweet, tweet updates, as well as profile updates. To begin, copy the ***tweets.json*** from the React project to the new Node project we've been practicing with. Put the JSON file in ***/data/tweets.json***.

Retrieving Tweets from a REST API

Let's start with retrieving an initial set of tweets stored in ***/data/tweets.json***. To make the data available, let's create a set of Webservices in a new file ***/services/tweeter-service.js***. Copy and paste the code below into ***tweeter-service.js***.

```
const tweets = require('../data/tweets.json');     // load tweets.json

module.exports = (app) => {                        // export a function that accepts an express instance

   const findAllTweets = (req, res) => {           // handle HTTP GET request for all tweets
      res.json(tweets);                            // respond with tweets array imported earlier
   }

   app.get('/api/tweets', findAllTweets);          // listen for HTTP GET request to /api/tweets, handle request
};
```

Towards the end of ***/server.js*** file, require the new service and pass it ***app***. Services need to be loaded after all other network configurations such as the CORS configuration done earlier as shown below.

```
...
require('./services/tweets-service')(app);    // load tweets service
```

```
app.listen(4000);
```

Now that we have an HTTP Webservice endpoint, let's try using it from the React client application. In the React project, create a twitter service file to interact with the server's twitter service in /src/services/twitterService.js, as shown below.

```
const TWEET_API = 'http://localhost:4000/api/tweets';    // base URL of API

export const fetchAllTweets = (dispatch) =>              // function to fetch tweets and notify reducer
   fetch(TWEET_API)                                      // asynchronously sends HTTP GET request to URL
      .then(response => response.json())                 // parse JSON body from response
      .then(tweets =>                                    // parsed tweets from server
         dispatch({                                      // notify reducer
            type: 'fetch-all-tweets',                    // action
            tweets                                       // send tweets from server to reducer
         })
      );
```

The ***fetchAllTweets*** function sends an asynchronous HTTP message to the server and registers a callback function to receive a reponse. The browser sends the request on behalf of JavaScript and notifies our code through the callback with the response from the server. The string data embedded in the response's body consists of a stream that can be parsed into its JSON representation with ***response.json()***. The resulting JSON, tweets, can finally be dispatched to the reducer for further processing.

Now that we can fetch tweets from the server, let's use the functionality to populate the initial set of tweets when the ***TweetList*** component first loads. We'll use the useEffect hook to detect when the ***TweetList*** component first loads, and invoke the ***fetchAllTweets*** function to go fetch the tweets from the server. We'll pass a reference of dispatch so it can notify the reducer when the tweets come back from the server. Update your ***TweetList*** component as shown below.

```
import React, {useEffect} from "react";                                  // import useEffect to handle load event
import {useDispatch, useSelector} from "react-redux";                    // import useDispatch to notify reducer
import TweetListItem from "./TweetListItem";
import {fetchAllTweets} from "../../../../services/tweetService";        // import fetchAllTweets service

const selectAllTweets = (state) => state.tweets.tweets;

const TweetList = () => {
   const tweets = useSelector(selectAllTweets);
   const dispatch = useDispatch();                                       // get dispatch from hook
   useEffect(() => fetchAllTweets(dispatch), [])                         // on load call fetch all tweets
```

When the server responds with the tweets, we notify reducers with tweets as part of the action. The reducer will create a new state with the tweets from the server as shown below. Restart the serer and client applications and confirm the tweets are rendered from the server.

```
const tweets = (state = initialState, action) => {
```

```
  switch (action.type) {
    case 'fetch-all-tweets':                    // handle fetch-all-tweets event
      return({                                  // set tweets from server
        tweets: action.tweets
      })
      break;
    case 'like-tweet':
```

Posting new Tweets

To create new tweets, we'll POST them to the server to store in its tweets array. They will persist there as long as the server is running. Next assignment will store them permanently using a database. For now let's take a look at how a client sends data to the server and how the server can process that data. In ***/services /tweet-service.js***, implement a function to handle HTTP POST messages sent from the client that include new tweets in their body, as shown below.

```
  ...
  const postNewTweet = (req, res) => {          // handle HTTP POST event
    const newTweet = {                          // initialize new tweet instance
      _id: (new Date()).getTime() + '',
      "topic": "Web Development",
      "userName": "ReactJS",
      "verified": false,
      "handle": "ReactJS",
      "time": "2h",
      "avatar-image": "../../../images/react-blue.png",
      "logo-image": "../../../images/react-blue.png",
      "stats": {
        "comments": 123,
        "retweets": 234,
        "likes": 345
      },
      ...req.body,                              // include/override with tweet posted by client
    }
    tweets = [
      newTweet,                                 // append new tweet at beginning of array
      ...tweets
    ];
    res.json(newTweet);                         // send new array back to the client
  }

  app.post('/api/tweets', createTweet);         // listen for HTTP POST and notify handler
  ...
```

In the React application, add function ***postNewTweet*** in ***tweetService.js*** as shown below. Use the function to POST an HTTP message to the server, including the new tweet embedded in the message's body.

```
export const postNewTweet = (dispatch, newTweet) =>   // function to post tweet to server
  fetch(TWEET_API, {                                  // send message to tweet API
```

```
    method: 'POST',                          // use HTTP POST
    body: JSON.stringify(newTweet),          // embed tweet in body as a string
    headers: {
      'content-type': 'application/json'     // tell server string in body is formatted as JSON
    }
  })
    .then(response => response.json())       // parse JSON response from server
    .then(tweet =>                           // actual tweet stored in server
      dispatch({                             // send tweet to reducer
        type: 'create-tweet',                // event to add new tweet at beginning of tweets
        tweet                                // actual tweet
      })
    );
```

In ***WhatsHappening/index.js***, refactor ***tweetClickHandler*** to use the new ***postNewTweet*** service as shown below. Restart both the client and server applications and confirm you can create new tweets. Refresh the browser and confirm that the tweets you created are persisted in the server.

```
import {postNewTweet, fetchAllTweets}        // import postNewTweet service
  from "../../../../services/tweetService";

const WhatsHappening = () => {

  const tweetClickHandler = () => {
    dispatch({                               // replace old dispatch
      type: 'create-tweet',
      tweet: {
        tweet: whatsHappening
      }
    });
    postNewTweet(dispatch, {
      tweet: whatsHappening
    });
  }
```

Deleting Tweets

Implement the delete HTTP handler in ***/services/tweets-service.js*** as shown below. This will allow the React client to remove tweets from the server with an HTTP delete request.

```
  const deleteTweet = (req, res) => {          // handles HTTP DELETE request
    const id = req.params['id'];               // parse ID from URL parameter :id
    tweets = tweets.filter(tweet => tweet._id !== id);  // filter out the deleted tweet
    res.sendStatus(200);                       // respond with success status
  }
  app.delete('/api/tweets/:id', deleteTweet);  // listen for HTTP DELETE request and notify function
```

In the React client application, implement the ***deleteTweet*** function below in ***/src/services/tweetService.js***. This function will allow the React client to send an HTTP DELETE request to the server. The ID of the tweet we are deleting will be encoded at the end of the URL.

```
export const deleteTweet = (dispatch, tweet) =>   // function to send HTTP request to delete tweet
   fetch(`${TWEET_API}/${tweet._id}`, {           // encode tweet ID at end of the URL
      method: 'DELETE'                            // use HTTP DELETE method
   }).then(response => dispatch({                 // ignore response, just dispatch to reducer
      type: 'delete-tweet',                       // send reducer a 'delete-tweet' event
      tweet                                       // pass tweet to be deleted
   }));
```

In the ***TweetListItem.js*** component, where we render each tweet, refactor the ***deleteTweetClickHandler*** to use the new ***deleteTweet*** service instead of just notifying the reducer. The reducer will be notified by the ***deleteTweetClickHandler*** after receiving acknowledgement from the server that the tweet has been deleted.

```
import {deleteTweet}                              // import deleteTweet from service
      from "../../../../services/tweetService";

const TweetListItem = ({tweet}) => {
   const dispatch = useDispatch();
   const deleteTweetClickHandler = () => {
      dispatch({type: 'delete-tweet', tweet});    // don't notify reducer right away, instead
      deleteTweet(dispatch, tweet);               // use new deleteTweet function to notify server
   }
   …
}
```

Updating a Tweet

Implement the likeTweet HTTP handler in ***/services/tweets-service.js*** as shown below. This will allow the React client to update tweets from the server with an HTTP PUT request.

```
   const likeTweet = (req, res) => {              // handles HTTP PUT request
      const id = req.params['id'];                // parse ID from URL parameter :id
      tweets = tweets.map(tweet => {              // iterate through the tweets
         if (tweet._id === id) {                  // if we found the tweet we want to change
            if (tweet.liked === true) {           // toggle the tweets liked property
               tweet.liked = false;
               tweet.stats.likes--;               // and update its likes property
            } else {
               tweet.liked = true;                // toggle the tweets liked property
               tweet.stats.likes++;               // and update its likes property
            }
            return tweet;                         // replace the updated tweet
         } else {                                 // otherwise
            return tweet;                         // keep the old tweet
         }
      });
```

```
    res.sendStatus(200);                        // respons with success status
  }
  app.put('/api/tweets/:id/like', likeTweet);   // listen for HTTP PUT request and notify function
```

In the React client application, implement the ***likeTweet*** function below in ***/src/services/tweetService.js***. This function will allow the React client to send an HTTP PUT request to the server. The ID of the tweet we are updating will be encoded in the end of the URL.

```
export const likeTweet = (dispatch, tweet) =>
  fetch(`${TWEET_API}/${tweet._id}/like`, {
    method: 'PUT'
  })
  .then(response =>
    dispatch({
      type: 'like-tweet',
      tweet
  }));
```

In the ***TweetStats.js*** component, where we render each number of likes, refactor the ***likeClickHandler*** to use the new ***likeTweet*** service instead of just notifying the reducer. The reducer will be notified by the ***likeTweet*** after receiving acknowledgement from the server that the tweet has been updated.

```
import {likeTweet} from "../../../../services/tweetService";

const TweetStats = ({tweet}) => {
  const dispatch = useDispatch();
  const likeClickHandler = () => {
    dispatch({type: 'like-tweet', tweet});
    likeTweet(dispatch, tweet);
  };
  ...
}
```

Deploying a client server application

Up to this point you should have a React and Node application interacting together as a single client server application running on your local computer. In this section we are going to deploy both applications to remote public servers so anyone can use or view our client server application. Deploying the React application should be as simple as committing and pushing your changes to Netlify as you have been doing thus far, but this time around your React application will be depending on a server application that we have not yet deployed. Let's first work on deploying the Node server application and then revisit come back to deploying the React application.

Deploying a Node Server application

First let's make sure we don't accidentally deploy unnecessary files such as IDE specific files in .idea and Node library files in node_modules. At the root of the project create file ***.gitignore***. Yes that's a period in front of ***gitignore*** because it's a ***hidden*** file. Copy and paste the content below into ***.gitignore***.

```
node_modules
.idea
```

Then initialize the directory as a GIT repository using the ***init*** command as shown below. Add and commit all the files to the new local code repotisoty. Once you have a local git repository, you are ready to deploy the project to a remote server.

```
git init
git add .
git commit -am "first commit"
```

Deploying a Node server application to Heroku

Thus far we have learned how to deploy React applications to Netlify. In this section we are going to learn how to deploy a Node server application to ***Heroku***. Head over to Heroku.com and create a free account. Make sure you remember the login credential since you'll need them shortly.

Install Heroku's CLI (Command Line Interface)

Heroku relies on a program you need to install in your computer, the command line interface or CLI. To install on macOS, type the following at the command line

```
brew tap heroku/brew && brew install heroku
```

On Windows, download the installer using the link below, or follow the latest instructions.

https://cli-assets.heroku.com/heroku-x64.exe

Windows installers may display a warning titled “Windows protected your PC”. To run the installation when this warning is shown, click “More info”, verify the publisher as “salesforce.com, inc”, then click the “Run anyway” button. In the Choose Components window, select all the components, click Next, then Install, then Close

Once you've installed the CLI, you can test if it worked by typing the following at the command line:

```
heroku --version                                    # thats a double dash
```

Login to heroku using the same credentials you used to create the heroku online account. Use the command shown below. Press any key and the CLI will open a browser window where you can login.

```
heroku login
heroku: Press any key to open up the browser to login or q to exit:
```

Challenge (required for graduates)

As a challenge, create ***/services/profile-service.js*** to persist changes in the profile. Changes users make to the profile in the ProfileScreen should be persisted in the server through the following API:

Method	URL	Handler	Description
GET	/api/profile	getCurrentProfile(req, res)	Retrieves the current profile which initially must be the same JSON used in the previous assignment. Current profile is stored in the server in a variable called ***profile***
PUT	/api/profile	updateCurrentProfile(req, res)	Updates the current ***profile*** variable in the server's memory. The new profile is included in the body of the request

In the React application, create a service client in ***/services/profileService.js*** that can send the HTTP request above. Implement each of the requests in similarly called functions in ***profileService.js***. Refactor the ***ProfileScreen*** and ***EditProfile*** to use the new services to interact with the Node server. When the ***EditProfile*** screens displays it should send the GET request above to retrieve the current profile and populate the form. When the user clicks the ***Save*** button it should send the PUT request above to send the updates to the server and save it.

Deliverables

As a deliverable, make sure you complete the ***Practice***, ***Build*** and ***Challange*** (if graduate student) sections of this assignment. All previous assignments must be accessible from the root context of your Website. Add, commit and push all work to your repository and confirm the Website works on Netlify.