Workshops Application

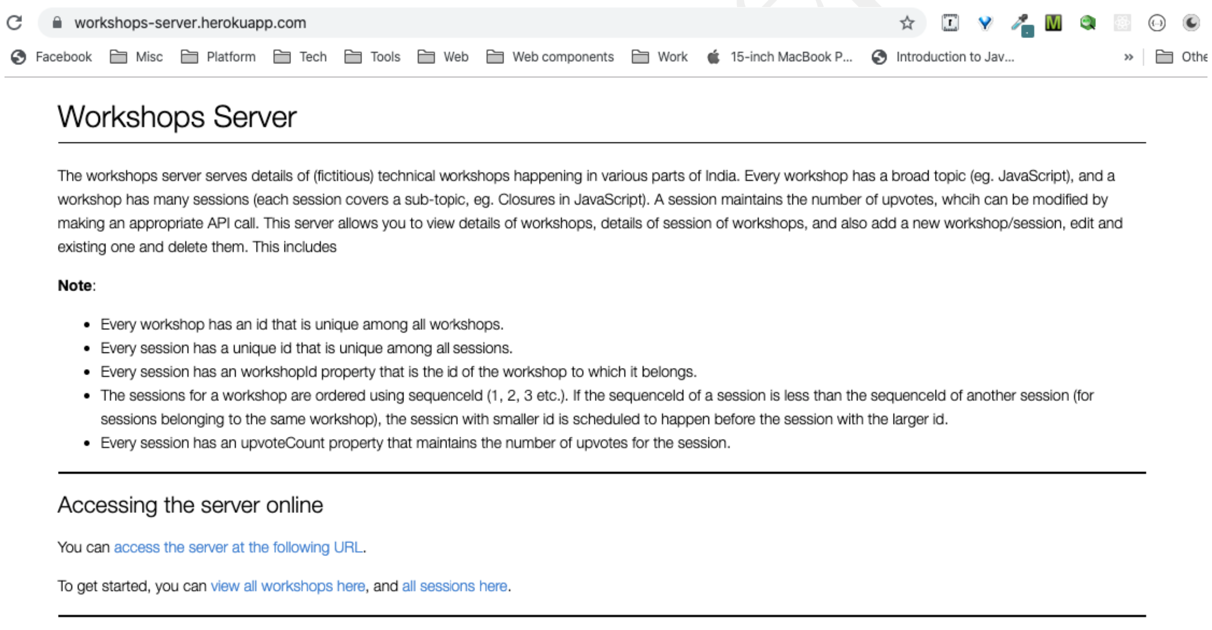
The Workshops application serves details of (fictitious) technical workshops hosted in various cities.

- Every workshop has a broad topic (eg. JavaScript)
- A workshop has many sessions.
  Each session covers a subtopic, eg. Functions in JavaScript.

You can view a list of workshops, details of every workshop, list sessions in a workshop, and also add a new session for a workshop.

Workshops Server

The server for the application, **along with documentation on the APIs** can be found here https://workshops-server.herokuapp.com/. You can go through the API documentation for details on how to get list of sessions, add a new session etc.



Workshops Server

The workshops server serves details of (fictitious) technical workshops happening in various parts of India. Every workshop has a broad topic (eg. JavaScript), and a workshop has many sessions (each session covers a sub-topic, eg. Closures in JavaScript). A session maintains the number of upvotes, whcih can be modified by making an appropriate API call. This server allows you to view details of workshops, details of session of workshops, and also add a new workshop/session, edit and existing one and delete them. This includes

Note:

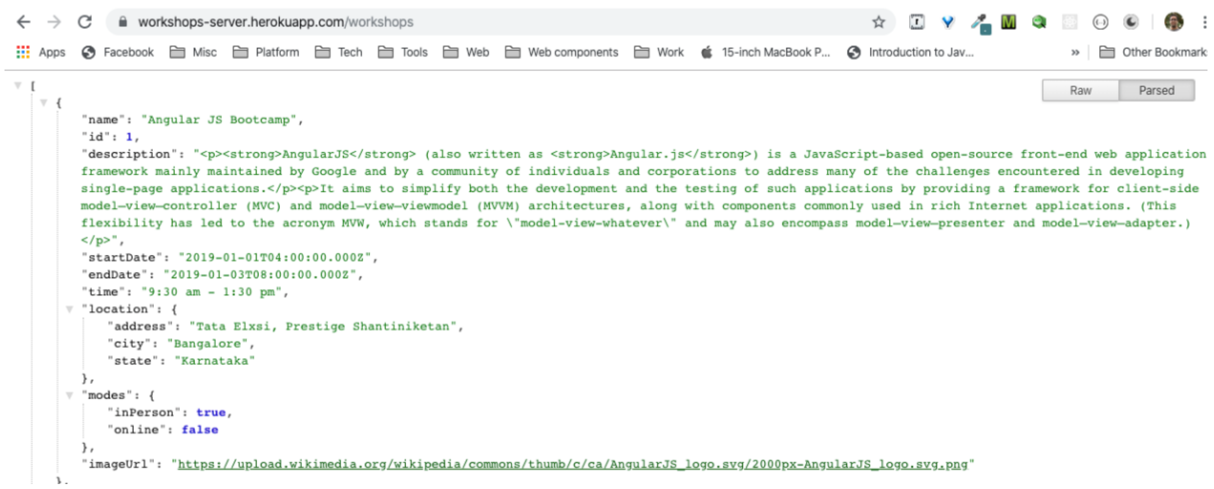
- Every workshop has an id that is unique among all workshops.
- Every session has a unique id that is unique among all sessions.
- Every session has an workshopId property that is the id of the workshop to which it belongs.
- The sessions for a workshop are ordered using sequenceId (1, 2, 3 etc.). If the sequenceId of a session is less than the sequenceId of another session (for sessions belonging to the same workshop), the session with smaller id is scheduled to happen before the session with the larger id.
- Every session has an upvoteCount property that maintains the number of upvotes for the session.

Accessing the server online

You can access the server at the following URL.

To get started, you can view all workshops here, and all sessions here.

For example, the list of all workshops can be found at https://workshops-server.herokuapp.com/workshops



```
[
  {
    "name": "Angular JS Bootcamp",
    "id": 1,
    "description": "<p><strong>AngularJS</strong> (also written as <strong>Angular.js</strong>) is a JavaScript-based open-source front-end web application
    framework mainly maintained by Google and by a community of individuals and corporations to address many of the challenges encountered in developing
    single-page applications.</p><p>It aims to simplify both the development and the testing of such applications by providing a framework for client-side
    model-view-controller (MVC) and model-view-viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications. (This
    flexibility has led to the acronym MVW, which stands for \"model-view-whatever\" and may also encompass model-view-presenter and model-view-adapter.)
    </p>",
    "startDate": "2019-01-01T04:00:00.000Z",
    "endDate": "2019-01-03T08:00:00.000Z",
    "time": "9:30 am - 1:30 pm",
    "location": {
      "address": "Tata Elxsi, Prestige Shantiniketan",
      "city": "Bangalore",
      "state": "Karnataka"
    },
    "modes": {
      "inPerson": true,
      "online": false
    },
    "imageUrl": "https://upload.wikimedia.org/wikipedia/commons/thumb/c/ca/AngularJS_logo.svg/2000px-AngularJS_logo.svg.png"
```

You may not see a neat display of JSON text. It's ok – you can install a JSON formatter Chrome extension if you like to view it neatly.

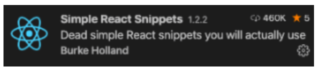
Supplied Files

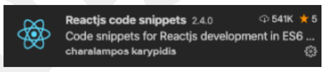
Some HTML, CSS and JavaScript code has been supplied to aid you to develop the application faster. These can be found in the supplied-files folder.

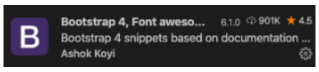
Extensions for VSCode

We shall be using the following extensions for VSCode. Make sure they are installed.

- **Simple React Snippets** by Burke Holland
- **Reactjs code snippets** by Charalampos Karypidis
- **Bootstrap 4, Font awesome 4 and 5 snippets** by Ashok Koyi


Simple React Snippets 1.2.2
460K ★ 5
Dead simple React snippets you will actually use
Burke Holland


Reactjs code snippets 2.4.0
541K ★ 5
Code snippets for Reactjs development in ES6 ...
charalampos karypidis

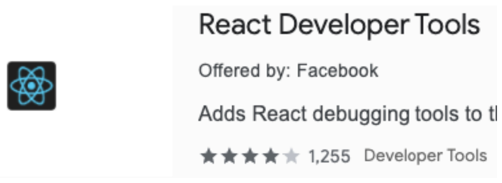

Bootstrap 4, Font aweso...
6.1.0 901K ★ 4.5
Bootstrap 4 snippets based on documentation ...
Ashok Koyi

Note: From the main menu selectView, then Extensions, and search for these extensions.

Extensions for Chrome

We shall be using the following extensions for Chrome. Make sure they are installed.

- **React Developer Tools** by Facebook
- **Redux DevTools** by remotedevio



Note: Open chrome://extensions in a Chrome tab, click on the **Menu icon on the top left corner**, then click **Open Chrome Web Store** at the **bottom of the Menu**, and search for these extensions.

Building the Workshops Application

Step 1: Setting up the basic React application

Create React App (CRA) is the tool supported by Facebook to scaffold React applications. It is very popular for setting up a general React application. We shall use it. You can find its documentation here - https://create-react-app.dev/

Various other tools are also popular, especially for special requirements (like building an app with server-side rendering). A list of popular alternatives to CRA can be found here.
https://reactjs.org/docs/create-a-new-react-app.html#recommended-toolchains

Let's set up the workshops-app React application using Create React App – you can either install the Node package `create-react-app` as a globally accessible package, or use `npx` to create the app using `create-react-app` without actually installing it as a global package.

1. **[Optional]**: Install CRA globally – since it is a global install you can run it from anywhere on your system. This is a one-time install. Once installed, you can create many React applications using it.

   ```
   npm install -g create-react-app
   ```

2. Navigate to the folder of your choice – we shall create the app here.

   ```
   cd /path/where/react/app/shall/be/created
   ```

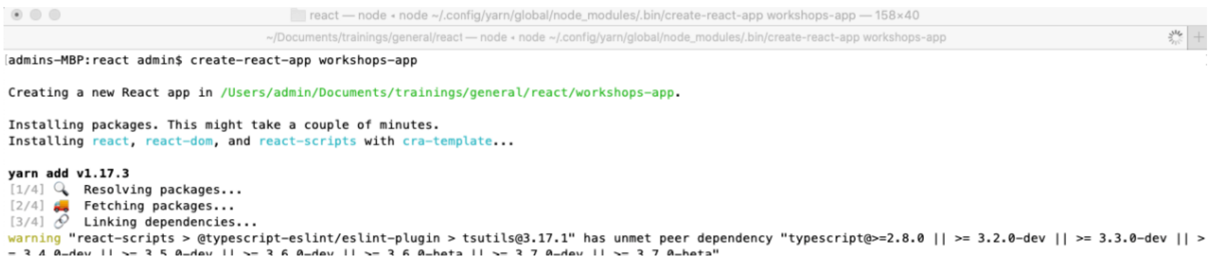
3. Run the command to create the app. You can use `create-react-app` directly if you installed it in 1. If not run it through `npx`.

   ```
   create-react-app workshops-app
   ```

   OR

   ```
   npx create-react-app workshops-app
   ```

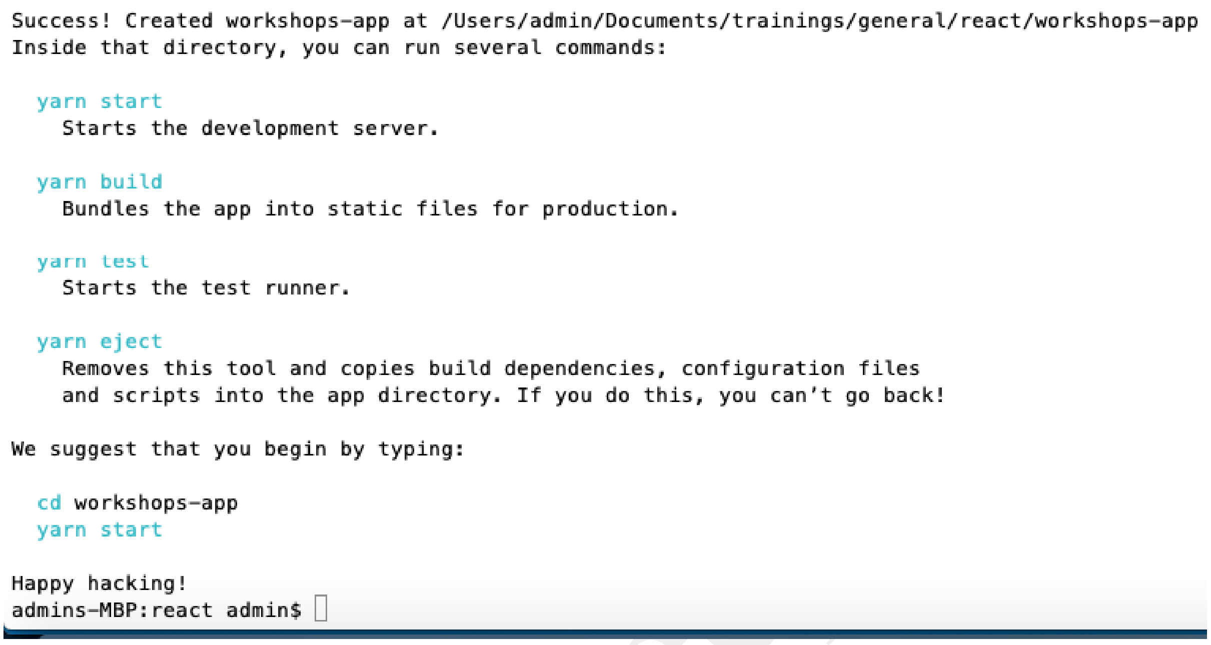
   You will see the project files and folders being created by CRA, and necessary Node packages being installed.



```
react — node ◂ node ~/.config/yarn/global/node_modules/.bin/create-react-app workshops-app — 158×40
~/Documents/trainings/general/react — node ◂ node ~/.config/yarn/global/node_modules/.bin/create-react-app workshops-app
admins-MBP:react admin$ create-react-app workshops-app

Creating a new React app in /Users/admin/Documents/trainings/general/react/workshops-app.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

yarn add v1.17.3
[1/4] 🔍  Resolving packages...
[2/4] 🚚  Fetching packages...
[3/4] 🔗  Linking dependencies...
warning "react-scripts > @typescript-eslint/eslint-plugin > tsutils@3.17.1" has unmet peer dependency "typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >
= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >= 3.6.0-beta || >= 3.7.0-dev || >= 3.7.0-beta"
```

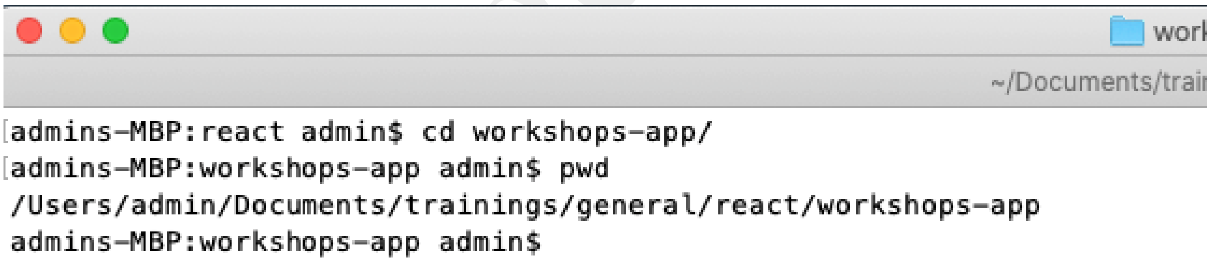
Note:You may face some permission issues if you use Powershell as your default shell in Windows. Simply choose Command Prompt (cmd) as the default shell, and it usually will sort out this issue. Within VSCode Terminal window, you can choose cmd as your default shell by choosing the Shell dropdown (displays the tab serial number, and currently executing process name), then "Select Default Shell". Form the shell choices that appear on top of the VSCode Window choose cmd instead of Powershell.

4. Verify the creation of your project! All's well that end well…well, at least if it ends this way ☺



```
Success! Created workshops-app at /Users/admin/Documents/trainings/general/react/workshops-app
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd workshops-app
  yarn start

Happy hacking!
admins-MBP:react admin$ 
```

5. Make sure to navigate to the project folder before proceeding.



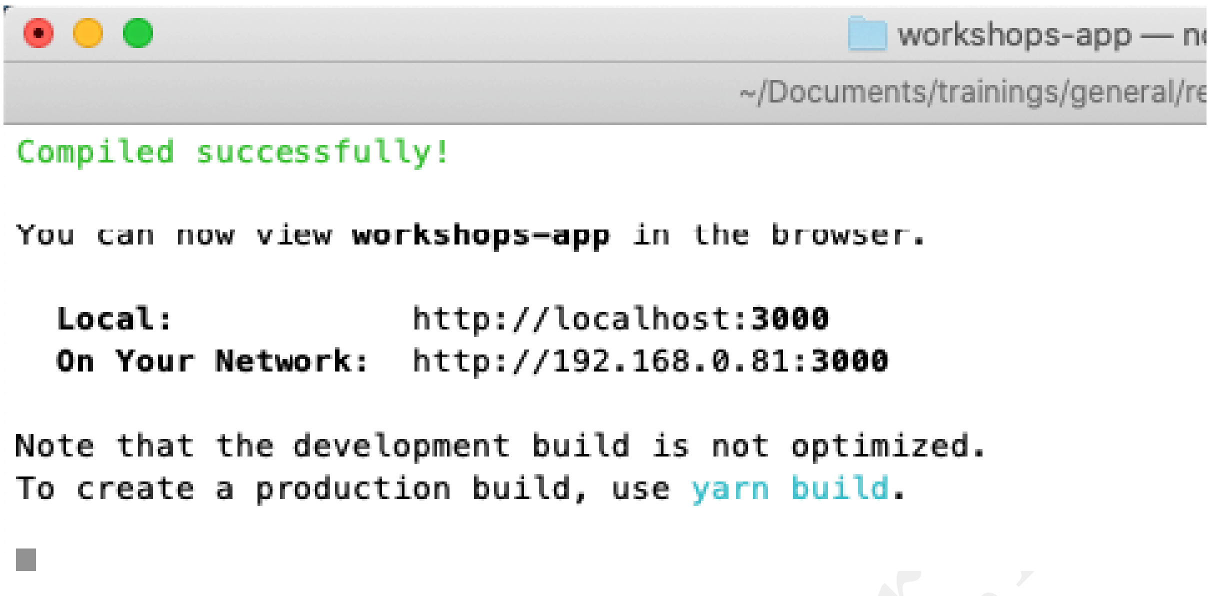
```
admins-MBP:react admin$ cd workshops-app/
admins-MBP:workshops-app admin$ pwd
/Users/admin/Documents/trainings/general/react/workshops-app
admins-MBP:workshops-app admin$
```

NOTE: You use cd instead of pwd on Windows to check the present working directory

6. The commands (npm scripts) supported by the app created by CRA can be found in package.json. To start the app from the project folder, we run the start script.

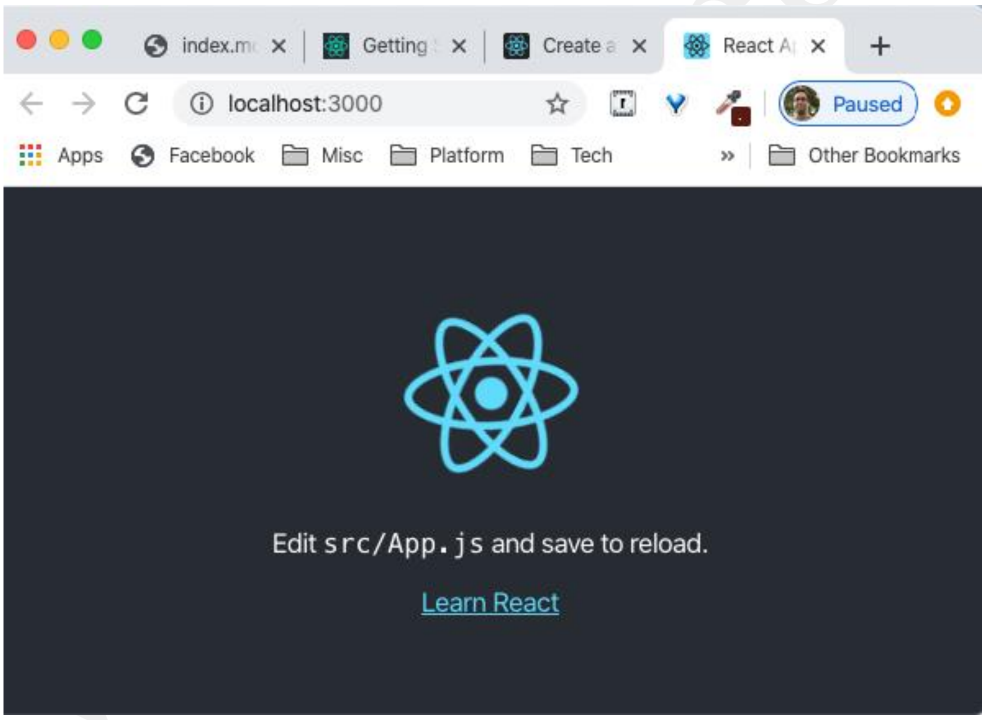
```
npm start
```

The script builds the application (using Webpack) and serves it using a web server on port 3000. You will see this once it builds and starts the server



```
Compiled successfully!

You can now view workshops-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.0.81:3000

Note that the development build is not optimized.
To create a production build, use yarn build.
```

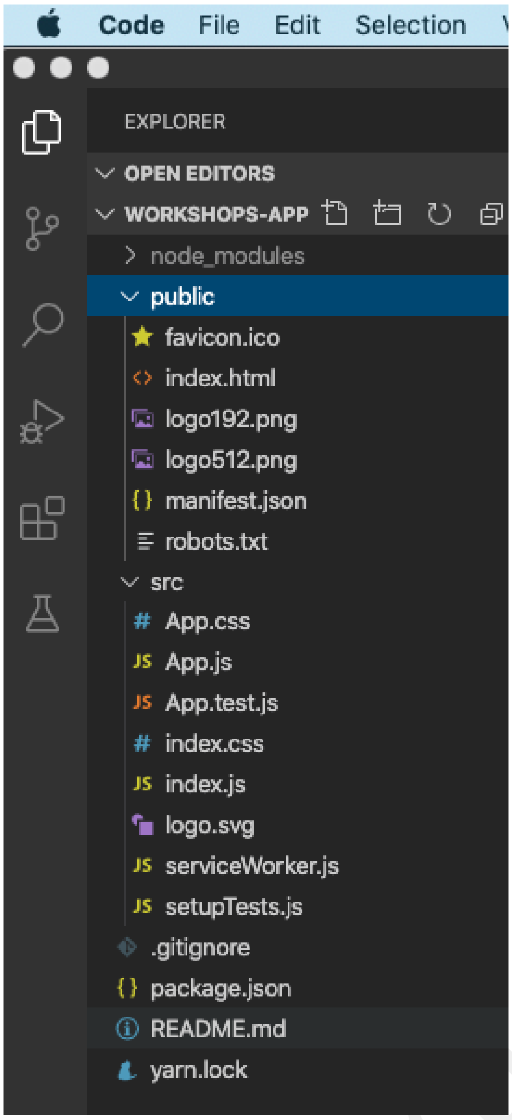
View it in a web browser on http://localhost:3000/



LEARNING: You have learnt how to setup a React app and run it. You need to do the setup only once - at the beginning of a project to set up the initial codebase for the application.

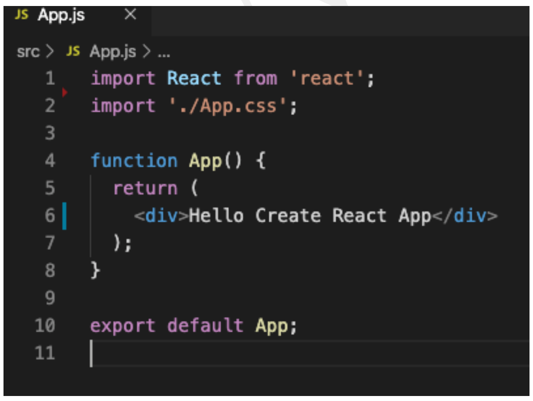
Step 2: Understanding the project files and folders

Open the newly created project folder in VSCode. Let's explore and understand the structure of the project.

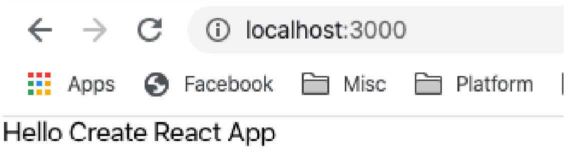


- **node_modules/** has the Node.js packages installed in the project. Three main Node.js packages were installed at the time of project creation
  - **React (react)** – The React core library
  - **React DOM (react-dom)** – The React DOM library
  - **React Scripts (react-scripts)** – A Node.js package that manages the workflow (build, test, serve etc.) – This internally uses Webpack for managing the workflows. You can find Webpack configurations within it.
- **public/** has the public assets, mainly the **index.html** of the Single Page Application (SPA) of the React app. The app is configured to serve this file.
- **src/** has the JavaScript and CSS files (application code). Index.js serves as the entry point for Webpack. It is the only file within this folder that is necessary – rest of the folder can be structured at will.
- **package.json** is the Node.js package file – it has npm script definitions, development and application dependencies and other details of the project
- **package-lock.json** and **yarn.lock** have exact versions of the installed Node packages. These files must fall under source control among other files in the project (committed to Git repository).
- **setupTests.js** is the file that runs before every unit test file. If configures unit tests before they can run.

CRA sets up development and other workflow using Webpack. It has Hot Module Replacement configured. Try making a change to index.js / App.js, or any of the included CSS files – you can see the browser refresh immediately.



```
import React from 'react';
import './App.css';

function App() {
  return (
    <div>Hello Create React App</div>
  );
}

export default App;
```



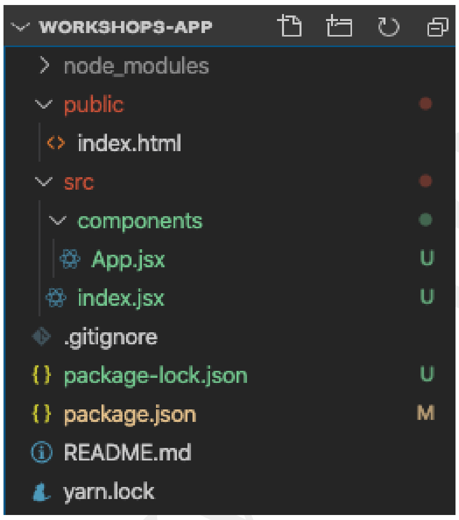
Hello Create React App

To keep things simple, let's remove all files in public and src folders except public/index.html, src/index.js and src/App.js. Let's also create a **src/components/** folder to house the components. Move App.js to within it components/ folder. As a convention, we name files with .jsx extension when they have JSX code within. Rename index.js to index.jsx, and components/App.js to components/App.jsx. Make sure to update the import path of App.jsx within index.jsx. The server needs to be restarted as it still searches for index.js and fails to start (use `Ctrl + C` to stop the server and `npm start` to start it again).

Note: We have removed links to deleted files within index.jsx and components/App.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

The folder structure should look like this now.


```
WORKSHOPS-APP
  > node_modules
  v public
      index.html
  v src
    v components
        App.jsx           U
      index.jsx           U
    .gitignore
    package-lock.json     U
    package.json          M
    README.md
    yarn.lock
```

Step 3: Add Bootstrap and Font Awesome to the application

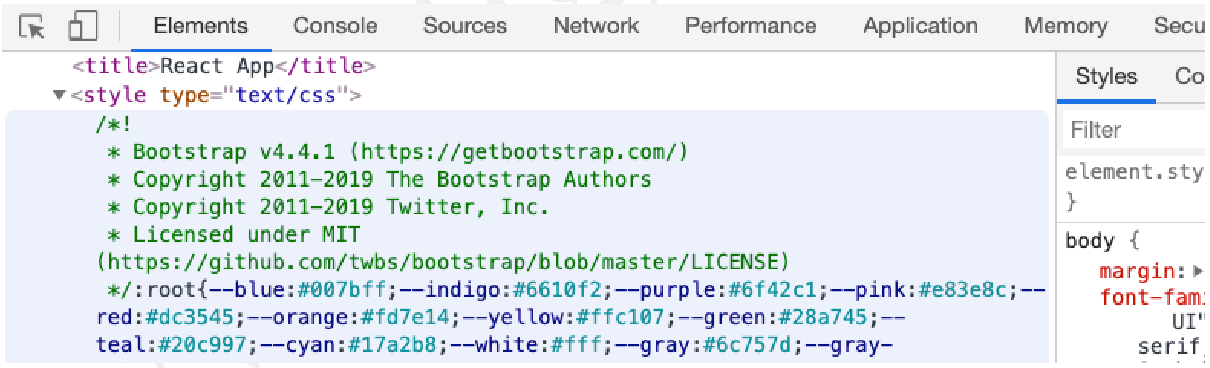
This application shall use Bootstrap for styles and Font Awesome for icons. From within the project folder install these libraries using npm

```
npm install bootstrap @fortawesome/fontawesome-free
```

The files for Bootstrap and Font Awesome will be installed in the node_modules/ folder under bootstrap/ and @fortawesome/fontawesome-free folders. To enable these in the application, the stylesheets of these libraries need to be imported. CRA uses Webpack, and Webpack allows import of CSS files too! (with necessary configuration).

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

import 'bootstrap/dist/css/bootstrap.min.css';
import '@fortawesome/fontawesome-free/css/all.min.css';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Absolute file path are with respect to node_modules/ folder – hence we begin with the bootstrap and @fortawesome folders while importing them. The style files that are imported get transformed in such a way that they end up as CSS within a style tag that is injected into the head of the index.html page (when it is served by the web server).



```
    <title>React App</title>
  ▼<style type="text/css">
      /*!
       * Bootstrap v4.4.1 (https://getbootstrap.com/)
       * Copyright 2011-2019 The Bootstrap Authors
       * Copyright 2011-2019 Twitter, Inc.
       * Licensed under MIT
      (https://github.com/twbs/bootstrap/blob/master/LICENSE)
       */:root{--blue:#007bff;--indigo:#6610f2;--purple:#6f42c1;--pink:#e83e8c;--
      red:#dc3545;--orange:#fd7e14;--yellow:#ffc107;--green:#28a745;--
      teal:#20c997;--cyan:#17a2b8;--white:#fff;--gray:#6c757d;--gray-
```

```
element.sty
}
body {
   margin: ▶
   font-fam
      UI"
   serif
```

LEARNING: You have learnt how external CSS files can be applied to your app. Adopt the same steps for any CSS you create and add in future.

EXERCISE: Try using some Bootstrap / Font Awesome class in App.js and make sure it is applied.

Step 4: Setting up navigation menu

Let's create a navigation menu for the app. Create **components/NavBar.jsx**. Create a Bootstrap-based Navbar for navigation in the app like so. We set URL **/** as home page route and **/workshops** to display a Workshops List page. These pages (components) shall be created in forthcoming steps and routing shall be set up.

NOTE:

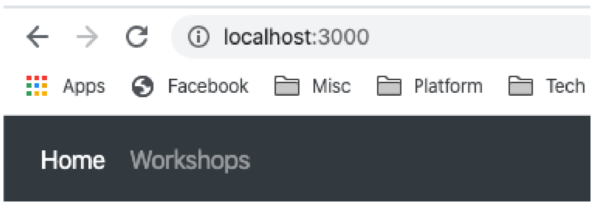
1. Use rsf to set up a basic Function-based component (React Stateless Function component) with necessary import and export.
2. Use b4-navbar-minimal-ul to set up the Navbar.
3. We have changed to dark theme – navbar-dark and bg-dark classes in nav element.
4. Use find and replace feature to replace all occurences of **class=** with **className=**
5. You may also use the supplied file (in supplied-files folder), which is provided for reference.

```
import React from 'react';

function NavBar(props) {
    return (
        <nav className="navbar navbar-expand navbar-dark bg-dark">
            <ul className="nav navbar-nav">
                <li className="nav-item active">
                    <a className="nav-link" href="/">Home</a>
                </li>
                <li className="nav-item">
                    <a className="nav-link" href="/workshops">Workshops</a>
                </li>
            </ul>
        </nav>
    );
}

export default NavBar;
```

Modify the App component - Remove all existing content from it and import and add a Navbar element to it.

```
import React from "react";
import NavBar from './NavBar';

function App() {
    return (
        <div>
            <NavBar />
        </div>
    );
}

export default App;
```

The app looks like this in the browser



Step 5: Creating and showing the About component

Create a new component **components/About.jsx**. You can use **b4-jumbotron-default** to set up the HTML (JSX). Since JSX is particular about self-closing syntax for tags without content, modify <hr> to hr />. Modify rest of content as per your wish – a sample is shown below. Find and replace all occurences of **class=** with **className=**.You may also use the supplied file.

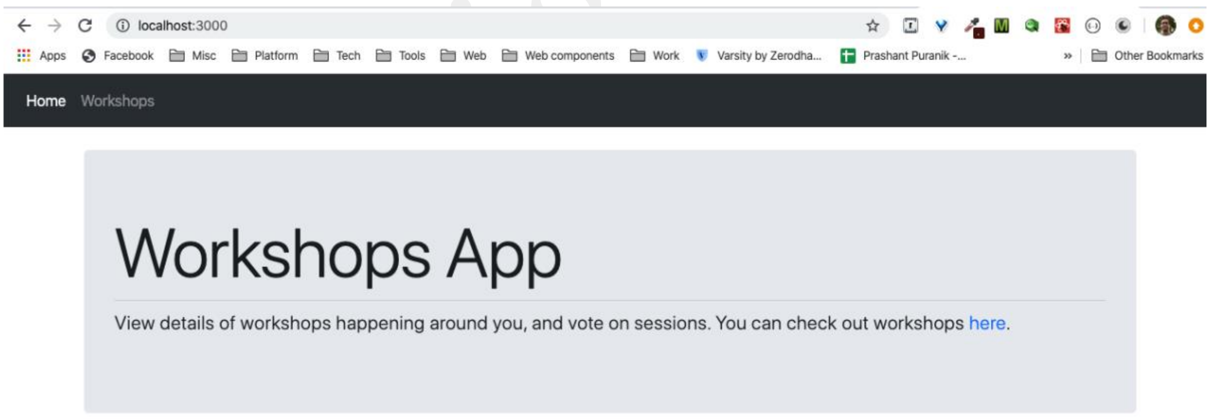
```
import React from 'react';

function About(props) {
    return (
        <div className="jumbotron">
            <h1 className="display-3">Workshops App</h1>
            <hr className="my-2" />
            <p className="lead">
                View details of workshops happening around you, and vote on sessions. You can
check out workshops <a href="/workshops">here</a>.
            </p>
        </div>
    );
}
```

```
export default About;
```

Modify the App component – Add an About element below the Navbar element.

```
import About from './About';

function App() {
  return (
    <div>
      <NavBar />
      <div className="container my-4">
        <About />
      </div>
    </div>
  );
}
```

Refresh the page and you will see this.


localhost:3000

Apps Facebook Misc Platform Tech Tools Web Web components Work Varsity by Zerodha... Prashant Puranik -... Other Bookmarks

Home Workshops

Workshops App

View details of workshops happening around you, and vote on sessions. You can check out workshops here.

Step 6: Scaffolding the WorkshopsList component

Create a new file - **components/WorkshopsList.jsx**. Since we shall have UI states, and workshops to be shown also possibly change (on pagination, filtering etc.), we shall create a class component. Define the initial state as an empty object.

NOTE:

1. Use rcc to set up a basic class-based component. Note that it imports Component from react package. We can refer to that class as Component now rather than React.Component.
2. We added a short message to see the WorkshopsList component renders fine when used.

```
import React, { Component } from 'react';

class WorkshopsList extends Component {
  state = {};

  render() {
    return (
      <div>
        WorkshopsList works!
      </div>
    );
  }
}

export default WorkshopsList;
```

Step 7: Setting up basic navigation

React Router is a popular package for setting up routing (i.e. URL navigation) in React applications. We shall use React Router v5 (react-router package) to set up navigation to home route / (to show About component) and workshops list route /workshops (to show WorkshopsList component).

1. **Setting up React Router**
   Install react-router-dom (from within the project folder)

   ```
   npm i react-router-dom
   ```

   NOTE: React Router has packages for navigation in web-based React applications (**react-router-dom**) and mobile apps built using React Native (**react-router-native**). Make sure you install and refer to documentation for react-router-dom. The 2 React router packages share a common package called react-router (which gets installed automatically when you install one of them).

2. **Watching for URL changes in the browser using <BrowserRouter /> component**
   React Router starts watching for URL changes (changes in the addresss of the browser's address bar) using a Router component. There are different types of Routers. For most

applications BrowserRouter will do (for older browsers that do not support HTML5 pushState API, HashRouter must be used).

Within index.jsx, set the App element now as a child of the BrowserRouter element. Now the App element and all its descendants will be able to use other Router components and perform routing related tasks (like navigation, or querying the current route's parameters etc.).

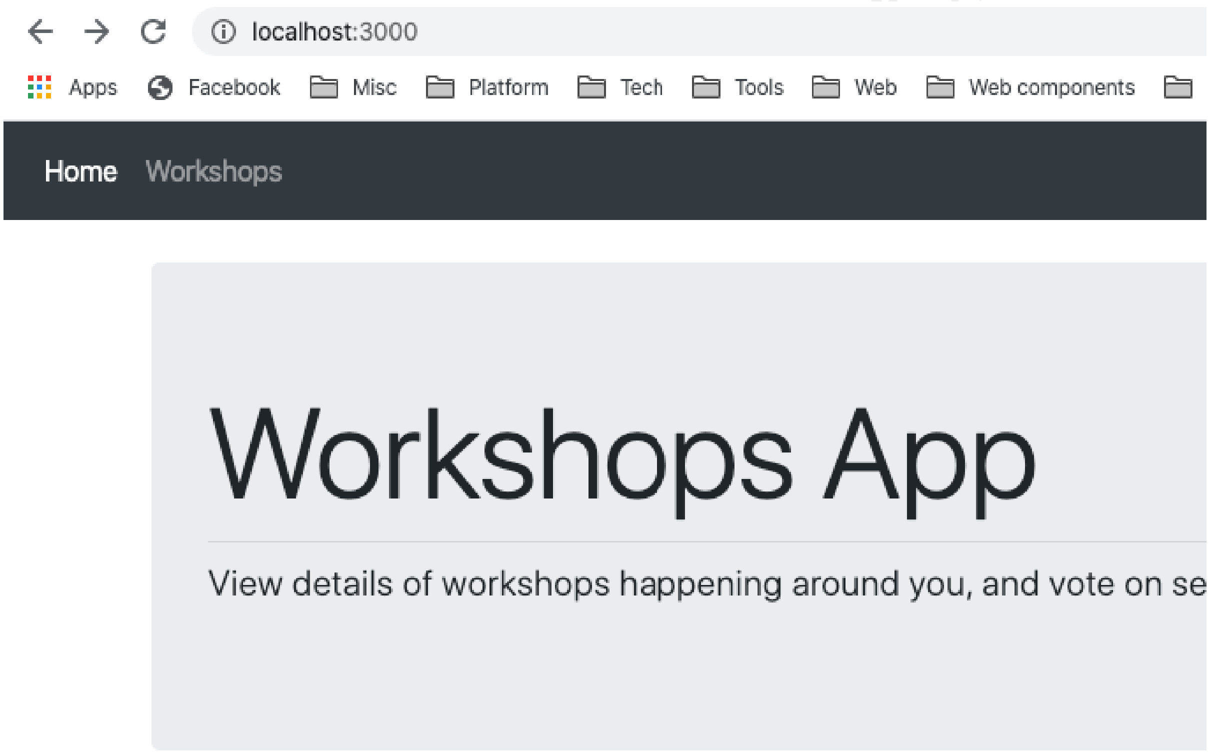
```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import App from './components/App';

import 'bootstrap/dist/css/bootstrap.min.css';
import '@fortawesome/fontawesome-free/css/all.min.css';

ReactDOM.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>,
  document.getElementById('root')
);
```

3. **Using <Route /> component to show a component when path matches browser URL**
Within components/App.jsx introduce a Route component to associate URL with the component to render. The Route component renders the component specified in the Route element when the URL changes, and it matches the one specified in the Route element. We introduce 2 Route elements like so (Note that we have removed the About element which was rendered directly within App).

```
import { Route } from 'react-router-dom';
import About from './About';
import WorkshopsList from './WorkshopsList';

function App() {
  return (
    <div>
      <NavBar />
```

```
        <div className="container my-4">
          <Route path="/" component={About} />
          <Route path="/workshops" component={WorkshopsList} />
        </div>
      </div>
    );
}

export default App;
```

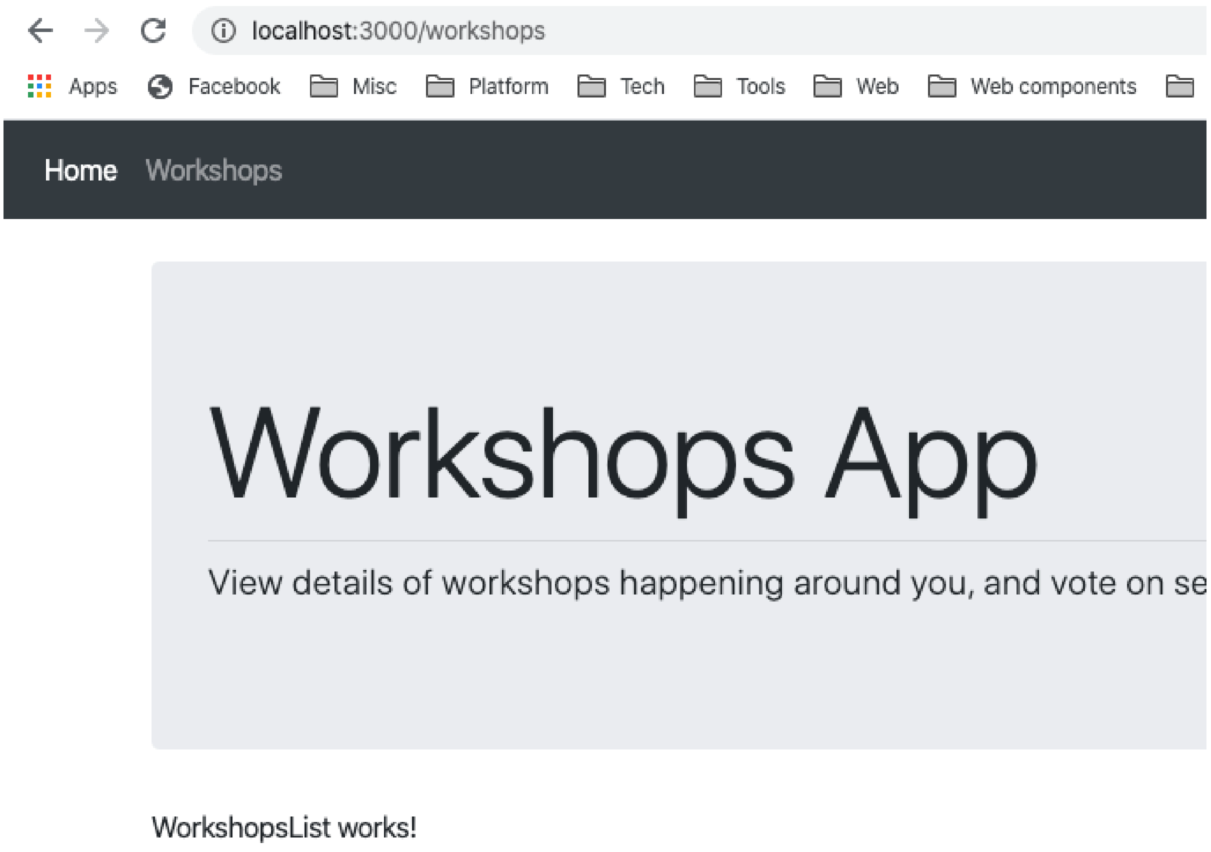
Refresh the page and you will see this on the home route – http://localhost:3000



Workshops App

View details of workshops happening around you, and vote on se

Click the Workshops link – The URL changes to http://localhost:3000/workshops and we see both About and WorkshopsList.



Workshops App

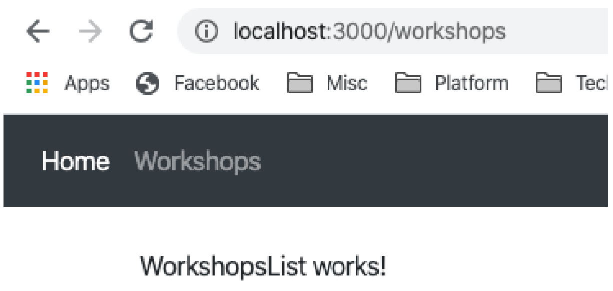
View details of workshops happening around you, and vote on se

WorkshopsList works!

4. **Preventing prefix match of path using exact={true} prop setting of <Route />**
   You may be wondering why About showed up! Well, by default the Route matches happens using a "prefix" match by default – since http://localhost:3000 is a prefix of http://localhost:3000/workshops (rather / is a prefix of /workshops), both Route paths are matched and both components show up. To overcome this we have the exact prop which must be set to the Boolean value true (however since true is the default value of a prop in React, we may omit setting the value of the exact prop).

```
<div className="container my-4">
    <Route path="/" component={About} exact={true} />
    <Route path="/workshops" component={WorkshopsList} />
</div>
```

Now, the /workshops behaves like expected.



WorkshopsList works!

EXERCISE: A Route element renders an instance of the component as its descendant, if path matches the current URL. This is the reason why the element shows up on path match. Check this out in the Components tab enabled by React Dev Tools in Chrome.

5. Using <Switch /> to show only one Route's component

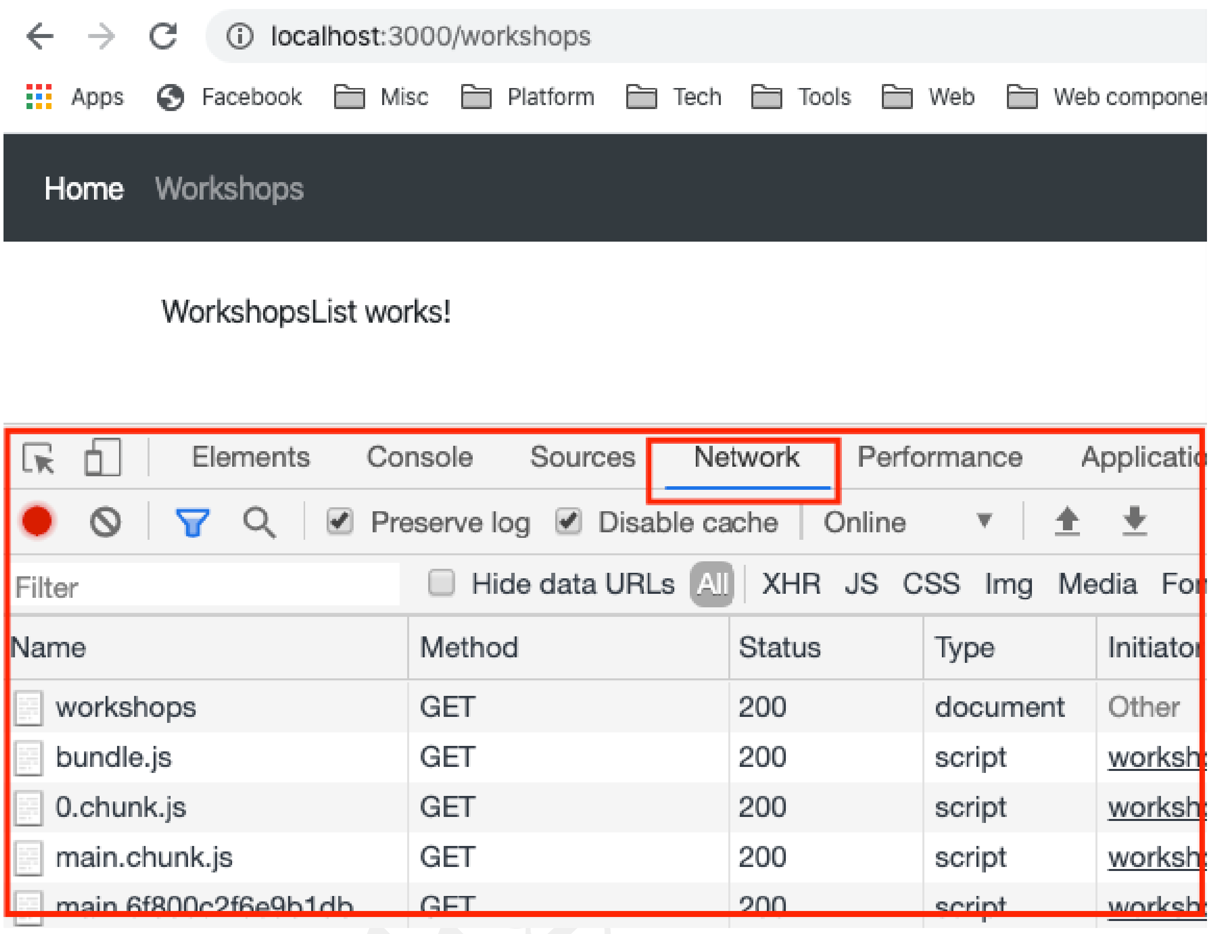
An alternative to using exact in this case is the Switch component of React Router. It behaves like switch-case in a programming language. You can enclose Route elements as children within a Switch element. When the URL changes, the Route elements are compared for path match one-by-one in source order (i.e. starting from the first Route child within Switch). If one of the Route's path matches the URL, its component is displayed and the rest of Route children are not checked (they are ignored). Thus only one component renders.

IMPORTANT: When using Switch, we must make sure the more specific route is above the more general one. For example, in our case the Route with the /workshops path must be rendered before the Route with the / path (if not, the component that matches /workshops, i.e. WorkshopsList component will never be rendered).

```
import { Switch, Route } from 'react-router-dom';


function App() {
  return (
    <div>
      <NavBar />
      <div className="container my-4">
        <Switch>
          <Route path="/workshops" component={WorkshopsList} />
          <Route path="/" component={About} />
        </Switch>
      </div>
    </div>
  );
}


export default App;
```

6. Using <Link /> component to prevent server requests (and consequent browser refresh)

We are building a Single Page Application (SPA) using CRA. Webpack bundles our assets (HTML, CSS, JS etc.). These are served when the first "page" of our app loads, and contain everything that's required for the application to function (except data which a component may require). Our NavBar uses anchor elements (links) which by default result in change in address and a request to the server to fetch the linked resource. Thus clicking on the NavBar links right now refreshes the page (since assets are again fetched) – you can view this in the Network tab.



WorkshopsList works!

Name	Method	Status	Type	Initiator
workshops	GET	200	document	Other
bundle.js	GET	200	script	worksh
0.chunk.js	GET	200	script	worksh
main.chunk.js	GET	200	script	worksh
main.6f800c2f6e9b1db	GET	200	script	worksh

To change the URL on click of the NavBar links, while preventing fetching these assets again, we use <Link to="" /> component of React Router instead of anchor component, i.e. <a href="" />. Link component internally uses HTML5 pushState API when available and sets up the URL path normally. On older browsers it results in hash-based URL. Note that we use the **to** prop of Link, instead of the **href** prop of anchor (a) element. Your components/NavBar.jsx should have the following changes.

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


function NavBar(props) {
  return (
    <nav className="navbar navbar-expand navbar-dark bg-dark">
      <ul className="nav navbar-nav">
        <li className="nav-item active">
          <Link className="nav-link" to="/">Home</Link>
        </li>
        <li className="nav-item">
          <Link className="nav-link" to="/workshops">Workshops</Link>
```

```
            </li>
        </ul>
    </nav>
  );
}
```

Verify that the app does not refresh the pages when navigating.

NOTE:
1. For links that take the user outside of the application (to a different website, i.e. domain) we would obviously use anchor tags (a).
2. <Link /> renders an anchor element (that does not initiate page refresh) as a child and also forwards any props set on it to that anchor element. For example, setting a CSS class on Link element would result in the class being set on the rendered anchor child.

7. **Using <NavLink /> component to highlight the "active" link**
   Right now, the NavBar highlights the Home link (in bright white color) irrespective of which page is currently active (this happens because the class "active" is present in the list item (li) having the home link. Note that Bootstrap allows us to add the class "active" to the anchor element instead of li too. We would like to apply class "active" if the Link path ("to" value) matches the current URL Since the class would be forwarded to the anchor element rendered as child of Link element, this would solve the problem.

   The <NavLink /> component does exactly that – when a CSS class name is set as value of the activeClassName prop, it sets it up on the rendered anchor child if the NavLink path ("to" value) matches the current URL. In addition we would require exact={true} to enable exact match (instead of prefix match). Let's make the following changes to the NavBar.

```
import { NavLink } from 'react-router-dom';

function NavBar(props) {
  return (
    <nav className="navbar navbar-expand navbar-dark bg-dark">
      <ul className="nav navbar-nav">
        <li className="nav-item">
          <NavLink className="nav-link" to="/" activeClassName="active"
exact={true}>Home</NavLink>
        </li>
        <li className="nav-item">
          <NavLink className="nav-link" to="/workshops" activeClassName="active"
exact={true}>Workshops</NavLink>
```

```
            </li>
        </ul>
    </nav>
  );
}
```

Step 8: Creating a service to make Ajax requests to the backend

In order to make Ajax requests we shall use the popular axios library.

Reference: https://github.com/axios/axios

It is promise-based – the methods get(), put(), post() and delete() that it defines on an HTTP client object, all return promises. These resolve with the response from the server, or reject with the error information (in case Ajax request fails). Let's install axios (make sure to run from the project folder).

```
npm install axios
```

As a good practice let's create a **services** folder within src/. Any files that house code to talk to the backend APIs will be created within this folder. Let's create a workshops.js file that will have logic to talk to the backend for workshops related APIs.

We export and object that shall contain methods to fetch all workshops, a workshop's details given the workshop's id, get sessions for a workshop with given id, add a new session for a workshop etc.

We keep the Workshops API base URL separate as it will be re-used across methods. Import axios and write a method to fetch all workshops. Note the use of template strings – the string passed to axios.get() is quoted using backticks in order to use string interpolation.

In case of success, axios resolves with a response object. The data property of this object has the HTP response body (data sent by server) – hence we return another promise with that as the resolved value – this makes the data readily available to the consumer of the API.

```
import axios from 'axios';


export default {
  apiBaseUrl: 'https://workshops-server.herokuapp.com',
  getWorkshops() {
     return axios.get( `${this.apiBaseUrl}/workshops` ).then( response => response.data );
  }
}
```

Step 9: Fetching list of workshops within the WorkshopsList component and displaying it.

We shift focus back to WorkshopsList.jsx. We shall maintain state object with the following properties

- status : maintains the loading status. Possible values are
    - 'FETCHING_WORKSHOPS'
    - 'FETCHED_WORKSHOPS'
    - 'ERROR_FETCHING_WORKSHOPS'
- workshops – Set to list of workshops once they are fetched from the server
- error – Set to the error object in case there was an error fetching list of workshops

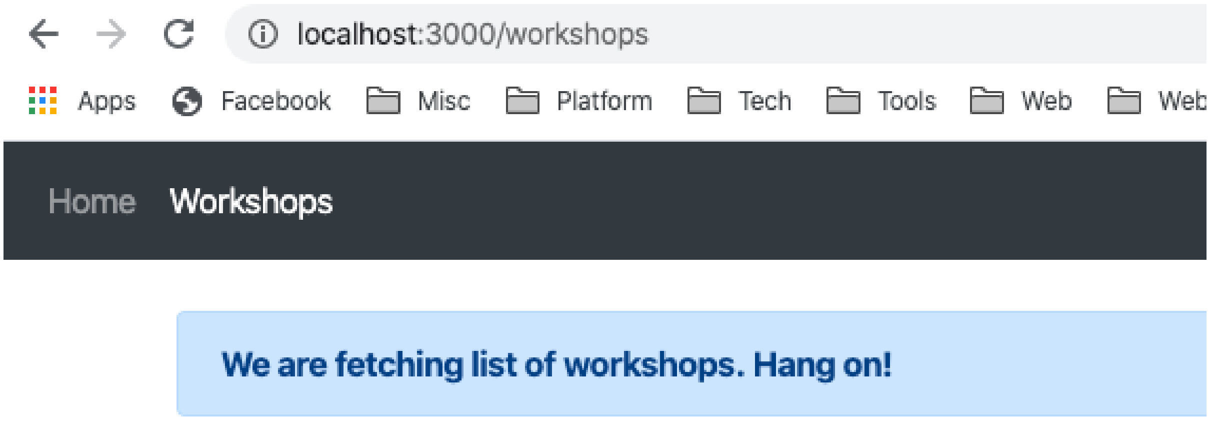
Let's fetch the list of workshops from within componentDidMount (use cdm to scaffold it). Before calling the service getWorkshops() method, we set state like so, so that the UI can re-render with an appropriate loading message (use b4-alert-dismissible for the alert widget markup) when the Ajax call goes out.

As a best practice we have created a string constant FETCHING_WORKSHOPS because it shall be used in more than one place (while setting state and in a switch-case in render)

```
import React, { Component } from 'react';

const FETCHING_WORKSHOPS = 'FETCHING_WORKSHOPS';

class WorkshopsList extends Component {
  state = {};

  render() {
    const { status } = this.state;

    let el = null;

    switch( status ) {
      case FETCHING_WORKSHOPS:
        el = (
          <div className="alert alert-primary alert-dismissible fade show" role="alert">
            <button type="button" className="close" data-dismiss="alert" aria-
label="Close">
              <span aria-hidden="true">&times;</span>
              <span className="sr-only">Close</span>
```

```
                </button>
                <strong>We are fetching list of workshops. Hang on!</strong>
              </div>
            );
            break;
        }

        return el;
    }

    componentDidMount() {
        this.setState({
            status: FETCHING_WORKSHOPS
        });
    }
}

export default WorkshopsList;
```

The component sets state on page load and the /workshops route should show this.



We are fetching list of workshops. Hang on!

Let's call getWorkshops(). If it succeeds we set the workshops on the state and status to 'FETCHED_WORKSHOPS'. If it fails, we set the error on the state and status to 'ERROR_FETCHING_WORKSHOPS'.

```
import WorkshopsService from '../services/workshops.js';

const FETCHING_WORKSHOPS = 'FETCHING_WORKSHOPS';
const FETCHED_WORKSHOPS = 'FETCHED_WORKSHOPS';
const ERROR_ETCHING_WORKSHOPS = 'ERROR_FETCHING_WORKSHOPS';

class WorkshopsList extends Component {
    componentDidMount() {
        this.setState({
            status: FETCHING_WORKSHOPS
        });

        WorkshopsService.getWorkshops()
            .then(workshops => this.setState({
                status: FETCHED_WORKSHOPS,
                workshops
            }))
            .catch(error => this.setState({
                status: ERROR_FETCHING_WORKSHOPS,
                error
            }));
    }
}
```

You can use async-await instead of directly working with the promise API if you prefer so.

```
async componentDidMount() {
        this.setState({
            status: FETCHING_WORKSHOPS
        });

        try {
            const workshops = await WorkshopsService.getWorkshops()
            this.setState({
                status: FETCHED_WORKSHOPS,
                workshops
            });
        } catch( error ) {
            this.setState({
                status: ERROR_FETCHING_WORKSHOPS,
                error
            });
        }
    }
```

The render() method shall be called when we set state after success/failure of the service method call. We show the list of workshops (on success) and error message (on failure). You may use the supplied files 01-WorkshopsList-workshop-cards.jsx to set up the UI for the success case – you will however need to map through the workshops to generate a column for each workshop - make changes to it as shown below.

We shall destructure state properties into variables.

```
const { status, workshops, error } = this.state;
```

On success, we render the list of workshops.

```
case FETCHED_WORKSHOPS:
el = (
    <div className="row">
        {
            workshops.map(workshop => (
                <div className="col-4 d-flex" key={workshop.id}>
                    <div className="card w-100 my-3 d-flex flex-column">
                        <div className="card-body">
                            <div className="card-img-container d-flex flex-column justify-content-center">
                                <img className="card-img-top w-50 d-block mx-auto"
src={workshop.imageUrl} alt={workshop.description} />
                            </div>
                            <h4 className="card-title">{workshop.name}</h4>
                            <div className="card-text">
                                <div>
                                    <span>{workshop.startDate}</span> -
<span>{workshop.endDate}</span>
                                </div>
                                <div>
                                    <span>{workshop.time}</span>
                                </div>
                                <div className="my-3">
                                    {workshop.description}
                                </div>
                            </div>
                        </div>
                    </div>
                </div>
            ))
        }
    </div>
);
break;
```

On failure, we show an appropriate alert message

```
case ERROR_FETCHING_WORKSHOPS:
  el = (
    <div className="alert alert-danger alert-dismissible fade show" role="alert">
      <button type="button" className="close" data-dismiss="alert" aria-label="Close">
        <span aria-hidden="true">&times;</span>
        <span className="sr-only">Close</span>
      </button>
      <strong>{error.message}</strong>
    </div>
  );
  break;
```

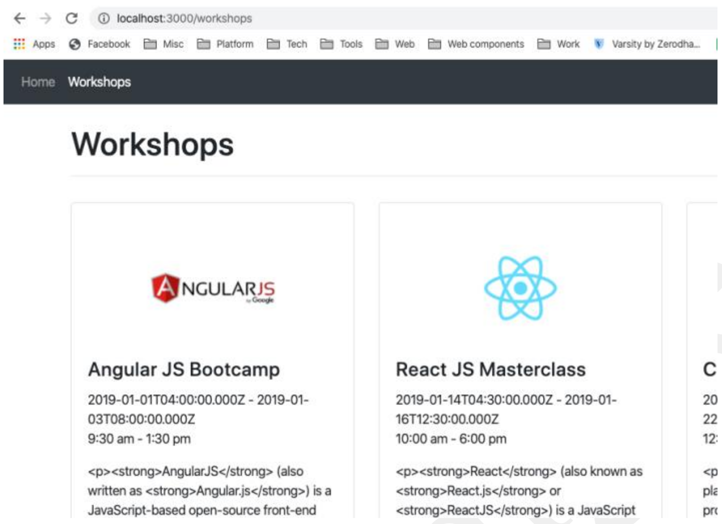
Also import the CSS file for WorkshopsList (supplied).

```
import './WorkshopsList.css';
```

Add a heading for the component which shows up in all states. You can do so by rendering el as a child like so.

```
return (
        <div>
        <h1>Workshops List</h1>
       <hr />
       {el}
        </div>
);
```

You should be able to see the WorkshopsList component function with different UI for different states.



Workshops

Angular JS Bootcamp
2019-01-01T04:00:00.000Z - 2019-01-03T08:00:00.000Z
9:30 am - 1:30 pm
<p><strong>AngularJS</strong> (also written as <strong>Angular.js</strong>) is a JavaScript-based open-source front-end

React JS Masterclass
2019-01-14T04:30:00.000Z - 2019-01-16T12:30:00.000Z
10:00 am - 6:00 pm
<p><strong>React</strong> (also known as <strong>React.js</strong> or <strong>ReactJS</strong>) is a JavaScript

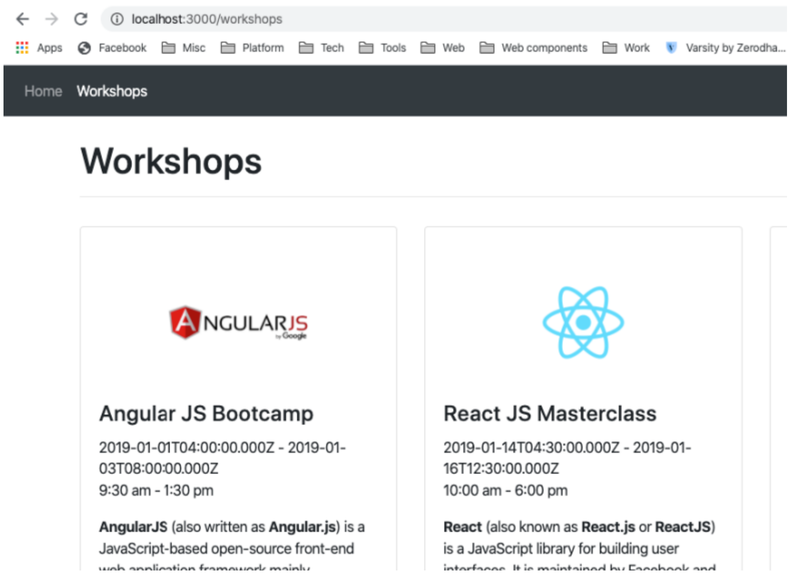
The workshop descriptions are HTML strings. However since interpolated text is set as innerText and not innerHTML, the HTML tags are escaped and appear literally. In React we interpolate HTML strings like so. Make sure to remove the string interpolated as child within the div.

```
<div className="my-3" dangerouslySetInnerHTML={{__html:
workshop.description}}></div>
```

Reference: https://reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml

Note: Be extremely careful when interpolating HTML strings – if they come from user input (eg. user comments on a website, articles posted in a blog site etc., they can be exploited for script injection by hackers. It is to mitigate such unintended side-effects that React makes it difficult to interpolate HTML strings.

After this change you can see it appear as HTML.



Workshops

Angular JS Bootcamp
2019-01-01T04:00:00.000Z - 2019-01-03T08:00:00.000Z
9:30 am - 1:30 pm
AngularJS (also written as **Angular.js**) is a JavaScript-based open-source front-end

React JS Masterclass
2019-01-14T04:30:00.000Z - 2019-01-16T12:30:00.000Z
10:00 am - 6:00 pm
React (also known as **React.js** or **ReactJS**) is a JavaScript library for building user

Step 10: Using a third-party library for displaying formatted date

We can build a custom utility function to format dates as per the format we require. It is also idiomatic in React to build out a component instead that accepts the raw date string as prop, and required format, and renders the formatted date value. Even better, let's use a third-party component that does exactly this.

Reference: https://www.npmjs.com/package/react-moment

Let's use React Moment that gives us a component to display formatted dates. Under the hood it uses the hugely popular Moment.js library for working with dates.

```
npm install --save moment react-moment
```

Now change the <span />s rendering the start and end dates to <Moment />s.

```
<div>
   <Moment interval={0} format="MMM D YYYY">
      {workshop.startDate}
   </Moment>
   {" - "}
   <Moment interval={0} format="MMM D YYYY">
      {workshop.endDate}
   </Moment>
</div>
```

Note: By default, the time updates every 60 seconds. We set the interval prop to 0 to disable automatic update.

Step 11: Creating an <Alert /> component

This entire step is an exercise. We have seen a Bootstrap alert being used in multiple places (loading and error state messaging). Create a custom Alert component in a **components/utils/Alert.jsx** file. We have chosen to create a separate utils/ folder to house this component as it potentially can be used in other projects too, and it helps to keep such components together. Alert takes as props…

- **message** (string): the message to display – any string
- **theme** (string): the class to use (can be only one of these - 'primary' | 'secondary' | 'success' | 'danger' | 'warning' | 'info' | 'light' | 'dark'.

Since there is no state associated with it, you can create it as a function component. Use PropTypes to validate the props. **Note that you will need to install prop-types and import it. You can use rscp to scaffold a function component with PropTypes setup.**

For documentation on PropTypes refer https://reactjs.org/docs/typechecking-with-proptypes.html

The following array will help you set the PropTypes for the theme prop.

```
[
    'primary',
    'secondary'
    'success',
    'danger',
    'warning',
    'info',
    'light',
    'dark'
]
```

You will need to set up a CSS class conditionally as per the theme prop. To set up a CSS classes conditionally, often using template string (backtick-quoted string) as the className prop value helps, since we can interpolate variables, use the conditional operator (the ternary ? : operator), logical and operator (&&) etc. within it. For example, in this case you can easily set up the class based on theme this way (you can of course think of other ways to do the same).

```
<div className={`alert alert-${props.theme} alert-dismissible fade
show`}>
```

Finally, use the Alert component in place of the alert messages in WorkshopsList

```
<Alert theme="primary" message="We are fetching list of workshops. Hang
on!" />
```

NOTE: You can refer to a sample implementation of Alert component within supplied files to verify your work.

Step 12: Toggling (hiding an showing alternatively) workshop descriptions

The workshops descriptions take too much space on screen. We would like to give an option of hiding/showing them. Add a button, clicking which shall toggle descriptions.

```
<h1>
        Workshops
    <div className="float-right">
          <button className="btn btn-sm btn-primary">
               Show details
      </button>
    </div>
</h1>
```

We introduce a boolean state property showDescriptions that dictates if descriptions are shown or not. Initially it is true, indicating descriptions shall be shown.

```
state = {
        showDescriptions: true
};
```

Next we add a method called toggleDescriptions() that changes this state property. Make sure to create using class properties syntax, i.e. arrow function syntax for class methods – this automatically binds a version of the method for every WorkshopsList object that is created, and ensures that the function context ("this") is bound to the WorkshopsList object.

```
toggleDescriptions = () => {
        this.setState(
        curState => ({
          showDescriptions: !curState.showDescriptions
        })
        );
}
```

Within render(), destructure showDescriptions for ease of use.

```
const { status, workshops, error, showDescriptions } = this.state;
```

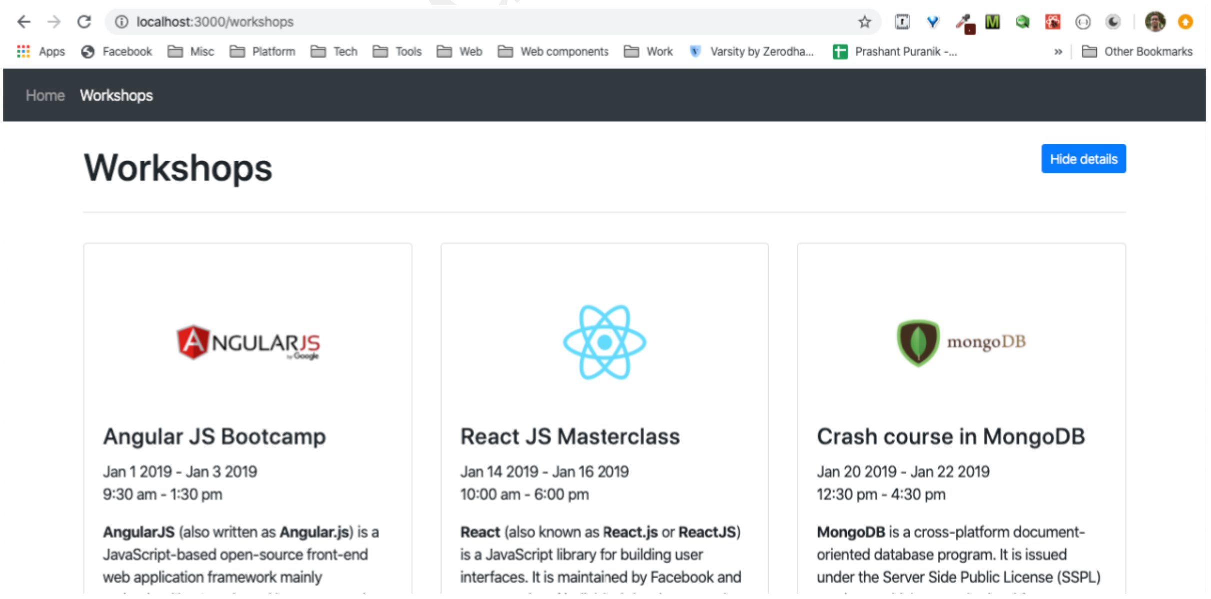
Have the button call toggleDescriptions( ) when clicked. Also update the button text to show the right message based on whether descriptions are shown or hidden

```
<button class="btn btn-sm btn-primary"
onClick={this.toggleDescriptions}>
        { showDescriptions ? 'Hide details' : 'Show details' }
</button>
```

Render the descriptions only when showDescriptions is true.

```
{
   showDescriptions && (
       <div className="my-3" dangerouslySetInnerHTML={{__html:
workshop.description}}></div>
   )
}
```

You should now be able to toggle descriptions using the button

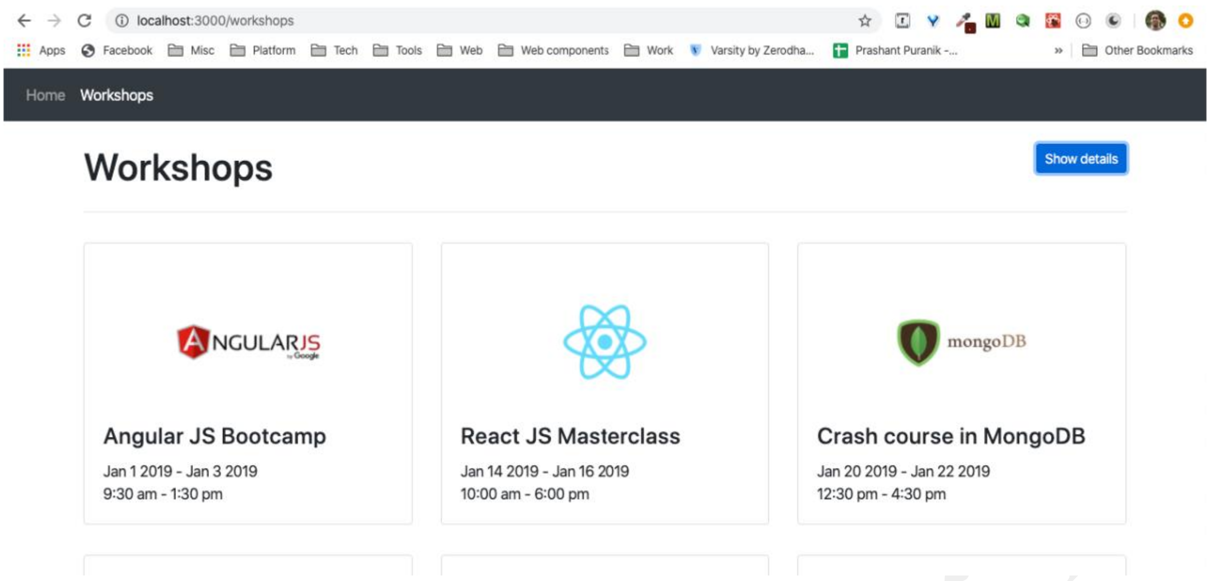


Workshops

Angular JS Bootcamp
Jan 1 2019 - Jan 3 2019
9:30 am - 1:30 pm
AngularJS (also written as **Angular.js**) is a JavaScript-based open-source front-end web application framework mainly

React JS Masterclass
Jan 14 2019 - Jan 16 2019
10:00 am - 6:00 pm
React (also known as **React.js** or **ReactJS**) is a JavaScript library for building user interfaces. It is maintained by Facebook and

Crash course in MongoDB
Jan 20 2019 - Jan 22 2019
12:30 pm - 4:30 pm
MongoDB is a cross-platform document-oriented database program. It is issued under the Server Side Public License (SSPL)



Step 13: Creating <WorkshopDetails /> component and navigating to it

We shall show the details of a workshop in a separate page when the user clicks on a card in the workshops list component. This new page shall make use of a WorkshopDetails component to show the details of a workshop.

Create a new file - **components/WorkshopDetails.jsx**, and set up a basic UI so we can see the component works when it loads.

We shall show the component right below the NavBar when the card for it within WorkshopsList is clicked. Since it appears below the NavBar (replacing the WorkshopsList), the Route for it should be added in App component. Let's say the URL **path it matches is /workshops/:id** where **:id is a placeholder for a path fragment** (in React Router, we match path fragments using a colon (:) followed by a variable (the variable name can be any valid JS variable name). A variable is created internally and is set to the actual path fragment when we navigate to this path - more on this later!. Make this change in the **App component**.

NOTE:
1. We have kept the most specific /workshops/:id path above the /workshops path.
2. Code for imports shall not be shown henceforth.

```
<Switch>
        <Route path="/workshops/:id" component={WorkshopDetails} />
    <Route path="/workshops" component={WorkshopsList} />
    <Route path="/" component={About} />
</Switch>
```

EXERCISE: Can you guess what would happen if the /workshops/:id and /workshops Routes were in reverse order?

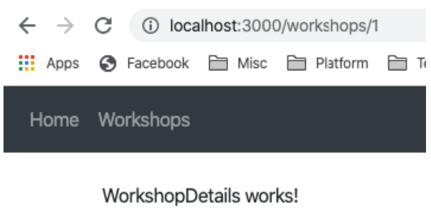
Set the cards in **WorkshopsList** component within a Link that links to the route for the WorkshopDetails with respective workshop id. The card markup is not shown.

```
<div className="col-4 d-flex">
        <Link to={`/workshops/${workshop.id}`} className="card-link-container">
                <div className="card">…</div>
        </Link>
</div>
```

To set right the link styles, add the classes text-reset text-decoration-none w-100 my-3 d-flex flex column to the Link and remove w-100 my-3 d-flex flex-column from the card.

QUESTION: On which element are these classes set internally?

Click to make sure the app navigates to /workshops/:id where id is the id of the workshop that was clicked. The WorkshopDetails component should render.



WorkshopDetails works!

Step 14: Fetching details of a workshop and showing it in the WorkshopDetails component

This entire step is an exercise. Since we shall have UI states, we shall create a class component. Define the initial state as an empty object.

Define a method getWorkshopById(id) in services/workshops.js, that accepts an id (number) and returns the details of the workshop from the URL https://workshops-server.herokuapp.com/workshops/:id where :id is to be substituted with an actual workshop's id – eg. https://workshops-server.herokuapp.com/workshops/1

Within WorkshopDetails class, make a call to getWorkshopsById(1) – this shall always fetch details of workshop with id = 1. We shall set it right in a later step.

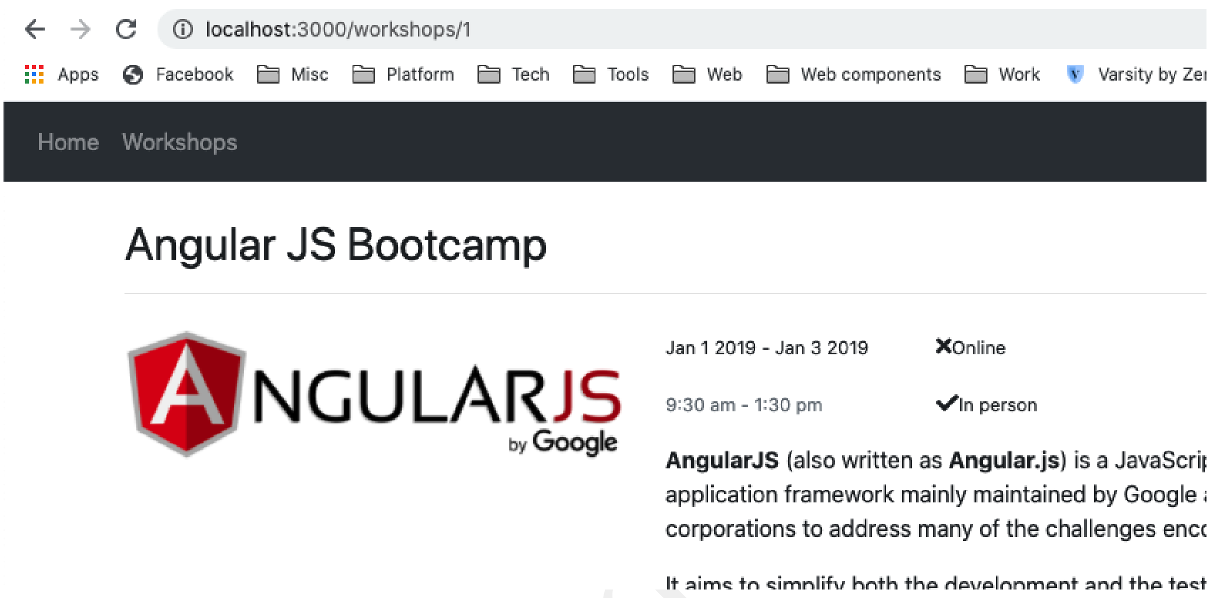
Make sure state now has a property called **workshop**, which has the details of the workshop as an object. Maintain UI states and show appropriate messages as the UI updates. Use the following constants for **status** property of state…

```
const FETCHING_WORKSHOP_DETAILS = 'FETCHING_WORKSHOP_DETAILS';
const FETCHED_WORKSHOP_DETAILS = 'FETCHED_WORKSHOP_DETAILS';
const ERROR_FETCHING_WORKSHOP_DETAILS =
'ERROR_FETCHING_WORKSHOP_DETAILS';
```

For setting up details of a workshop **you can use the supplied file 01-WorkshopDetails.jsx**. Note that since there is just a single workshop object whose details need to be shown, there is no requirement for iteration (using map() etc.) like in the case of WorkshopsList.

Also make sure to display a cross mark (fa fa-times) when a workshop mode (Online / In person) is not available, or check mark (fa fa-check) when a workshop mode is available.

The UI when workshop details are fetched and rendered should look like this.



Angular JS Bootcamp

Jan 1 2019 - Jan 3 2019 ✗Online

9:30 am - 1:30 pm ✓In person

AngularJS (also written as **Angular.js**) is a JavaScrip
application framework mainly maintained by Google
corporations to address many of the challenges enc

It aims to simplify both the development and the test

Step 15: Using Route-rendered component's props to show the details of the right workshop

Any component rendered as a result of a match with a <Route /> is passed 3 props

- **history** – this object has methods for programmatically navigating to back -back(), forward – forward(), by any number of steps in the browser's navigation history stack – go(), and to specified routes – push()
- **location** – this object has details of various parts of the URL – eg. search string (search property), hash fragment (hash property) etc.
- **match** – this object contains information about how a <Route /> path matched the URL. For example, when the URL is /workshops/2, and <WorkshopDetails /> show up on match against <Route path="/workshops/:id" component={WorkshopDetails} />, then match.params (which has values of actual path parameters) will have an id property set to 2 (the matched "id").

Reference: https://reacttraining.com/react-router/web/api/Route/route-props

We shall use the **match** prop with its **params.id** property to get the id of the workshop that appears in the URL. Thus, we fetch and display the right workshop.

If you used promises directly then modify like so.

```
const id = this.props.match.params.id;
WorkshopsService.getWorkshopById( id )
```

```
        .then( … )
```

Or, if you used async…await, then

```
const id = this.props.match.params.id;
const workshop = await WorkshopsService.getWorkshopById( id );
```

Step 16: Fetching list of sessions along with workshop details

Modify the service method getWorkshopById(id) by adding a second argument - getWorkshopById(id, includeSessions = false). When includeSessions is true it makes the following API request that fetches details of workshop with given id, along with list of sessions for it. Make sure to substitute :id with the id passed to the method.

http://workshops-server.herokuapp.com/workshops/:id?_embed=sessions

If includeSessions is false (the default is false in case second argument is not passed), then only workshop details are fetched (as before).

Alternatively, we can also make a to the same URL but supply a config object with the search params, when making a call to axios.get( ).

```
getWorkshopById( id, includeSessions ) {
     const config = includeSessions ? { params: { _embed: 'sessions' } } : {};
     return axios.get( `${this.apiBaseUrl}/workshops/${id}`, config ).then( response =>
response.data );
  }
```

Step 17: Displaying a list of sessions using a <SessionsList /> component

This entire step is an exercise. Build a function component <SessionsList /> that accepts the list of Sessions as a prop (called **sessions**) and renders it. Name your component file as SessionsList.jsx.

For setting up the list of sessions **you can use the supplied file 01-SessionsList.jsx**, and copy over and import the styles from the supplied file **SessionsList.css**. The li element with CSS class **list-group-item** in 01-SessionsList.jsx must be iterated over using map( ).

Set sessions prop to a default value of [ ] (empty array). Use PropTypes to validate the **sessions** prop – it should be an array of objects, where each object has the following properties

```
id: number
workshopId: number
sequenceId: number
name: string
speaker: string
```

```
duration: number
level: string (one of "Basic" | "Intermediate" | "Advanced")
abstract: string
upvoteCount: number
```

Have the sessions rendered below the workshop details (add this in the UI that is rendered once workshop details are fetched successfully).

```
<SessionsList sessions={workshop.sessions} />
```

Step 18: Using styled-components library to set up component styles

When adding SessionsList.css, you may have noticed that the styles for the cross (.fa.fa-times) and check mark (.fa.fa-check) were affected. We could change the selector in SessionsList.css to scope the selector and have it change only the voting part of sessions. We will take an alternate approach – we shall first style using inline styles. After that we shall a more powerful alternative – a CSS in JS library called styled-components to do the same.

First, we remove the CSS import (shown commented here instead).

```
// import './SessionsList.css';
```

Define an object instead with the styles for .fa (copy from SessionsList.css and modify). The style object in React requires kebab-cased CSS properties to be written in camelCase instead. Also, all property values are strings (quoted).

```
const fa = {
        fontSize: '2em',
        cursor: 'pointer',
        color: '#aaa'
}
```

Add this object as an inline style for the elements that represent the Font Awesome icons.

```
<div className="d-flex flex-column align-items-center">
        <i style={fa} className="fa fa-caret-up"></i>
        <span>{upvoteCount}</span>
        <i style={fa} className="fa fa-caret-down"></i>
</div>
```

Refresh the page – you will see the styles applied only for the voting buttons. The .fa.fa-times and .fa.fa-check icons are not affected.

One disadvantage of inline styling is we cannot set the hover styles similarly. To overcome this and other limitations of inline styles (when compared to external stylesheets), we use "CSS in JS" libraries. These libraries help us define hover styles, media queries etc. in JS. One such popular library is styled-components. This library allows us to define styles using plain CSS syntax. Let's install and use it.

```
npm install styled-components
```

Reference: https://styled-components.com/docs

When using styled-components, styles are defined using the ES2015 tagged template literals syntax, or as style objects. In the CSS in JS approach, we define styles in a JS file - define a SessionsList.css.jsx with the following code

```
import styled from 'styled-components';


export const VotingButton = styled.i`
   font-size: 2em;
   cursor: pointer;
   color: #aaa;

   &:hover {
      color: black;
   }
`;
```

The CSS styles are wrapped in a tagged template literal, with tag styled.<HTMLElementType>. This results in creation of a component that wraps the native React equivalent of the native HTMLElementType. In the above case we have created a wrapper component (VotingComponent) over the italic component. A uniquely named class is automatically generated and it is added to the VotingComponent. The & represents the element being styled (italic element in this case) – and we thus add pseudo-class styles.

Import the VotingButton component in SessionsList and use it in place of the normal italic component – note that we do not require the class fa anymore and that object can be safely deleted.

```
import { VotingButton } from './SessionsList.css';
```

Note: The imported file is actually a .jsx file (the .jsx file extension is not required and hence omitted).

```
<div className="d-flex flex-column align-items-center">
         <VotingButton className="fa fa-caret-up"></VotingButton>
         <span>{upvoteCount}</span>
         <VotingButton className="fa fa-caret-down"></VotingButton>
</div>
```

Refresh the page – you will see the styles applied only for the voting buttons. The .fa.fa-times and .fa.fa-check icons are not affected in this approach either.

Step 19: Set up CSS classes for the session level badges based on level of the session

This entire step is an exercise. You need to add a class to the badge element based on the session level.

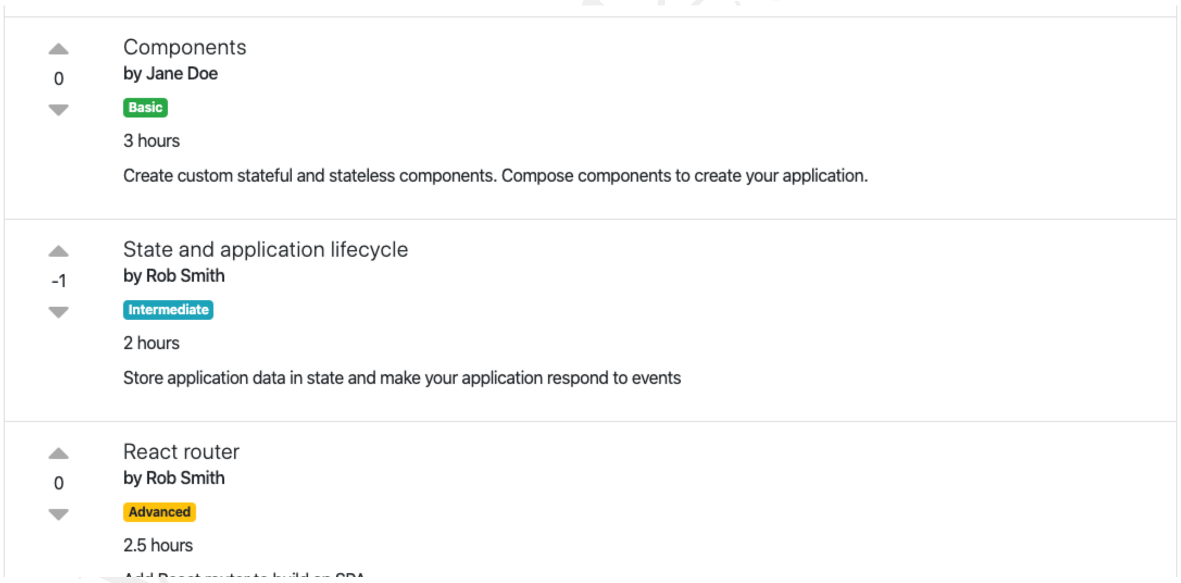
```
<span className="badge">{session.level}</span>
```

Additional CSS class to be applied based on session level

```
Basic: badge-success
Intermediate: badge-info
Advanced: badge-warning
```

Tip: It is better to create a helper function that returns the class to be applied when passed the session level **OR**, you may also maintain an object map of session levels to classes. In either case, set the class using a template string (you can call methods within ${}).

At the end of this the session badges should appear so.


0
Components
by Jane Doe
Basic
3 hours
Create custom stateful and stateless components. Compose components to create your application.
-1
State and application lifecycle
by Rob Smith
Intermediate
2 hours
Store application data in state and make your application respond to events
0
React router
by Rob Smith
Advanced
2.5 hours

Step 20: Registering vote on click of upvote / downvote buttons

Every session is displayed with a upvote / downvote button. Create a **services/sessions.js** file. We define service methods that can vote a session up or down given the session's id (the request is a PUT request but no data needs to be sent in HTTP request body).

```
import axios from 'axios';
```

```
export default {
    apiBaseUrl: 'https://workshops-server.herokuapp.com',
    upvote( id ) {
        return axios.put( `${this.apiBaseUrl}/sessions/${id}/upvote` ).then( response =>
response.data );
    },
    downvote( id ) {
        return axios.put( `${this.apiBaseUrl}/sessions/${id}/downvote` ).then( response =>
response.data );
    }
}
```

We next implement voting functionality for sessions in <SessionsList />. Since SessionsList a function component, it must receive the method(s) to upvote/downvote on sessions from its parent, i.e. <WorkshopDetails /> component.

First, define these methods on WorkshopDetails – we accept the index of the session to vote up/down instead of the session object or its id. This makes it easy to update the session once a vote is registered in the backend.

The backend returns the entire session details once a vote is registered – this will have the updated vote count. **Note how we set state once that happens** – state is set to a **new object** which copies over workshop details, and create a **new array for sessions**, and copies over all sessions except the session being voted on (that session's updated data is copied over instead).

This strategy of carefully creating new object/array makes it straightforward to compare old state and new state and find out what exactly changed (by a simple equality comparison using ===) wherever required.

```
onVoteChange = ( idx, updatedSession ) => {
    this.setState(
        curState => ({
            workshop: {
                ...curState.workshop,
                sessions: [
                    ...curState.workshop.sessions.slice( 0, idx ),
                    updatedSession,
                    ...curState.workshop.sessions.slice( idx + 1 )
                ]
            }
        })
    )
}

upvote = ( idx ) => {
    return SessionsService
            .upvote( this.state.workshop.sessions[idx].id )
            .then( updatedSession => this.onVoteChange( idx, updatedSession ) )
            .catch( error => alert( `Something went wrong while registering the vote
(${error.message})` ) );
}

downvote = ( idx ) => {
    return SessionsService
            .downvote( this.state.workshop.sessions[idx].id )
            .then( updatedSession => this.onVoteChange( idx, updatedSession ) )
            .catch( error => alert( `Something went wrong while registering the vote
(${error.message})` ) );
}
```

Now pass the methods as props to SessionsList.

```
<SessionsList sessions={workshop.sessions} upvote={this.upvote}
 downvote={this.downvote} />
```

Accept the props in SessionsList (we destructure below).

```
function SessionsList( { sessions, upvote, downvote } ) {
        ...
}
```

Accept array index when iterating using map( )

```
sessions.map( ( session, idx ) => ( ... ) )
```

Set the vote up/down event handlers - pass the index of session (within the sessions array) when they are called.

```
<div className="voting-widget d-flex flex-column align-items-center">
        <i className="fa fa-caret-up" onClick={() => upvote( idx )}></i>
        <span>{session.upvoteCount}</span>
        <i className="fa fa-caret-down" onClick={() => downvote( idx )}></i>
</div>
```

EXERCISE: Modify propTypes to include upvote and downvote

Step 21: Creating a generic <VotingWidget /> and using it

This entire step is an exercise. The voting functionality of session is one that can be useful in diverse places in the app. For example if we wanted to upvote/downvote workshops, we would require similar setup. Create a file **components/utils/VotingWidget.jsx** that defines a generic VotingWidget. It accepts the following as props and renders the UI.

- **upvoteCount** – the upvote count
- **upvote, downvote** – functions to be called on click of upvote and downvote buttons

Move the styles for voting widget (into a style object within VotingWidget.jsx / a separate VotingWidget.css.jsx), and make use of it in the new VotingWidget component.

Use the VotingWidget component in place of the present voting UI in SessionsList component, passing in the required props.

NOTE: You can refer to a sample implementation of VotingWidget within supplied files to verify your work.

Step 22: Adding a service method to fetch workshops page-by-page
Add a new method to the workshops service object that shall fetch only results on a given page. The backend API to do this is
http://workshops-server.herokuapp.com/workshops/?_page=:page&limit=:limit

For example to fetch the 3rd page where every page has 5 results, you make a call to
http://workshops-server.herokuapp.com/workshops/?_page=3&limit=5

This method accepts the page number (page) and number of results per page (limit), fetches only results for that page, and returns a promise that resolves with the results for that page (workshops), and total number of workshops (total). Note that the backend returns the total number of workshops (overall in the database, and not only for that page) in a custom HTTP header called x-total-count – we need to extract this detail using the Axios response object's headers property (an object with keys being HTTP headers, and values the HTTP header values).

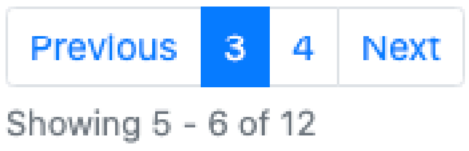
```
getWorkshopsByPage( page, limit ) {
    return axios.get( `${this.apiBaseUrl}/workshops?_page=${page}&_limit=${limit}` )
            .then( response => {
                console.log( response.headers );
                return {
                    workshops: response.data,
                    total: parseInt( response.headers['x-total-count'] )
                };
            })
}
```

Step 23: Creating a generic <Pagination /> component and using it to paginate workshops in <WorkshopsList />

The WorkshopsList component fetches all the workshops from the server in one go. In a real application this is not practical – you may have thousands if not millions of results, and it is impractical to fetch all in one go. So we use techniques like pagination and infinite scrolling to fetch results in batches.

Let's implement the pagination feature as a separate utility component, and use it within WorkshopsList to fetch the workshops page-by-page.

Let's imagine what such a generic Pagination component would look and work like.



What could the props for this component be? The component displays the page numbers around the current page, could possibly take in the number of such page numbers to display (let's call these *pagination items*), the *total* number of results (eg. 12 in the figure above), and number of results per page (eg. 2 in the figure above - let's call this the *page size*). The Previous and Next buttons should result in the previous and next set of pagination items being displayed.

The Pagination component should handle the click on pagination items (and the Next and Previous buttons). Also, the component including Pagination as a child component should be able to react to changes in the page number – for this reason we also need to be able to pass a function that will be called back when user clicks on a new pagination item (including Previous and Next links). We will pass this callback function the page number selected by the user (see currentPage in description that follows). This will enable the component using Pagination to fetch results for the newly selected page from the backend.

Taking all this into consideration, the **props** for the Pagination component would be

- pageSize (number)
- total (number)
- numPaginationItems (number)

- onPageChange (function)

The Pagination component maintains the currentPage (number) in its **state**. The pagination item for the current page is highlighted in the UI.

The logic for generating the right pagination items is not really important to understanding the component itself – hence we use a partial implementation of the Pagination component in supplied files. Copy utils/Pagination.jsx from supplied files, over to the utils/ folder in the app.

We can now use Pagination in WorkshopsList. Import the following in WorkshopsList.jsx.

```
import Pagination from './utils/Pagination';
import WorkshopsService from '../services/workshops.js';
```

Add the following 2 properties to state, and 2 class properties which are used to configure the Pagination widget.

```
  state = {
     showDescriptions: true,
     total: 0,
     currentPage: 1
  };

  pageSize = 2;
  numPaginationItems = 2;
```

We add a method that will be called back by the Pagination widget – it receives the current page selected by the user and sets it up on the state.

```
onPageChange = ( { currentPage } ) => {
         this.setState( { currentPage } );
}
```

We add the following helper method for gathering pagination related data from multiple sources into one (an object which is returned by the method).

```
getPaginationData() {
         return {
         total: this.state.total,
         currentPage: this.state.currentPage,
         pageSize: this.pageSize,
         numPaginationItems: this.numPaginationItems,
         onPageChange: this.onPageChange
```

```
        };
}
```

Create a separate method that fetches the workshops based on currentPage (in state). Since this method uses await, we need to mark it as an asynchronous method.

Note: We factor this code out into a separate method since both componentDidMount() and componentDidUpdate() will need to fetch workshops.

```
async fetchWorkshops() {
    this.setState({
        status: FETCHING_WORKSHOPS
    });

    try {
        const data = this.getPaginationData();
        const { workshops, total } = await WorkshopsService.getWorkshopsByPage(
data.currentPage, data.pageSize );
        this.setState({
            status: FETCHED_WORKSHOPS,
            workshops,
            total
        });
    } catch( error ) {
        this.setState({
            status: ERROR_FETCHING_WORKSHOPS,
            error
        });
    }
}
```

The componentDidMount() method looks like so now.

```
async componentDidMount() {
        this.fetchWorkshops();
}
```

The componentDidUpdate() method is similar, except we don't want the workshops to be fetched afresh unless currentPage changes in the state (the currentPage is updated via the callback passed to Pagination widget - other changes are immaterial when it comes to fetching workshops).

```
async componentDidUpdate( oldProps, oldState ) {
    if( oldState.currentPage !== this.state.currentPage ) {
        this.fetchWorkshops();
    }
}
```

Finally, we add the Pagination widget to WorkshopsList - this sets the Pagination working.

```
<div>
    <div className="clearfix">
        <h1>
            Workshops
            <div className="float-right">
                <button className="btn btn-sm btn-primary" onClick={this.toggleDescriptions}>
                    { showDescriptions ? 'Hide details' : 'Show details' }
                </button>
            </div>
        </h1>
    </div>
    <hr />
    <div className="col-12 clearfix">
        <Pagination {...this.getPaginationData()} className="float-right mb-2" />
    </div>
    {el}
</div>
```

Note: The Pagination component developed is a modified adaption of the one described in this article - https://www.digitalocean.com/community/tutorials/how-to-build-custom-pagination-with-react

Step 24: Creating a form to add a new session for a workshop

To components/ folder let's add the AddSession component that contains a form to add a new session to a workshop. Set this component up from the supplied files – AddSession.jsx.- the basic class component has been set up in this file.

Let's setup a tabbed navigation with 2 links on WorkshopDetails page – one which routes to SessionsList, and the other to AddSession. Since we shall be setting up routing, import the following in WorkshopDetails.jsx

```
import { NavLink, Route } from 'react-router-dom';
```

Change the anchor tags to Link elements (and href to to). We shall show the WorkshopDetails with SessionsList component tab open when user navigates to **/workshops/:id**, and WorkshopDetails with AddSession component tab open when user navigates to **/workshops/:id/add**. Since WorkshopDetails should be shown on both URLs, make sure the **exact** prop is NOT used in App.jsx.

In App.jsx… (make sure there is no exact prop set for WorkshopDetails)

```
<Route path="/workshops/:id" component={WorkshopDetails} />
```

The WorkshopDetails receives the **match prop** by virtue of being rendered by a Route match (along with history and location props).

- match.url - the URL that matched the matching component. For example, when the URL is the form /workshops/:id, it matches the WorkshopDetails component (via the Route in App.jsx), and the match prop in WorkshopDetails would be **'/workshops/:id'**. In order to set the URLs to match the SessionsList and AddSession components, we set up the corresponding **Route**s using this prop (it is preferable to hardcoding).
- match.path – the path that matched the matching component. This would be the exact address displayed in the address bar which loaded the matched component. For example, if we navigated to /workshops/2, then it would be '/workshops/2' (whereas match.url would be '/workshops/:id')

Note: We are supposed to use match.url when setting up Links for child components, and match.path when setting up Routes for child components. Following this principle makes sure the matched parts of the path (:id value) is available in the child components too (i.e. inside SessionsList and AddSession).

Firstly, use b4-nav-tabs-pills-ul to help set up the tabbed navigation.

```
<ul className="nav nav-tabs">
        <li className="nav-item">
        <NavLink className="nav-link" to={this.props.match.url} activeClassName="active"
exact={true}>Sessions</NavLink>
        </li>
    <li className="nav-item">
        <NavLink className="nav-link" to={`${this.props.match.url}/add`}
activeClassName="active" exact={true}>Add a session</NavLink>
        </li>
</ul>
```

Then, set up the Routes.

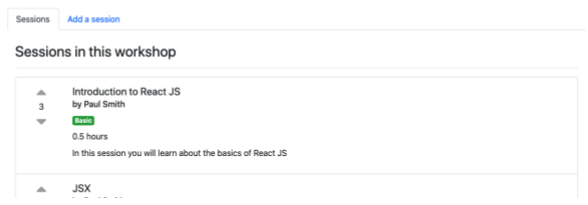
- We use the render prop of Route (instead of component) in the SessionsList case, as we would like to pass in props to SessionsList (you can imagine the render prop is like function that receives the Route props – history, location and match and returns the UI to render).

- The Route props may simply be passed on to the child component (the way we have done).
- Also make sure to add **exact** to the SessionsList Route so that it does not appear on /workshops/:id/add (as /workshops/:id is a prefix).

Reference: https://reacttraining.com/react-router/web/api/Route/render-func

```
<Route exact path={this.props.match.path} render={ routeProps => (
   <SessionsList sessions={workshop.sessions} upvote={this.upvote} downvote={this.downvote}
{...routeProps} />
)} />
<Route path={`${this.props.match.path}/add`} component={AddSession} />
```

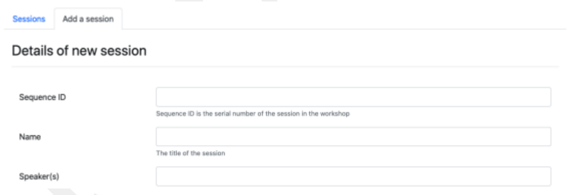
You should be able to navigate using the tabs like so.



Sessions in this workshop

3
Introduction to React JS
by Paul Smith
Basic
0.5 hours
In this session you will learn about the basics of React JS

JSX



Details of new session

Sequence ID
Sequence ID is the serial number of the session in the workshop
Name
The title of the session
Speaker(s)

Step 25: Setting up AddSession component

It has been set up as a class component because the form controls will be **controlled components** – i.e. we shall store user inputs in state properties, and set up the value prop of these form controls from the state properties.

First we setup state to have properties to hold user input values (which are set as and when user provides inputs). We also setup an errors property in state to hold the error messages (which are set after validation of inputs)

```
state = {
      values: {
          sequenceId: 0,
          name: '',
          speaker: '',
          duration: 0,
          level: '',
          abstract: ''
      },
      errors: {
          sequenceId: [],
          name: [],
          speaker: [],
          duration: [],
          level: [],
          abstract: []
      }
};
```

We make the form input controls as controlled components by adding an onChange listener (an updateValues( ) method) to update the state property corresponding to the input control being changed (the event object gives us necessary details. The implementation of this method is simplified because of the fact that the names of the inputs are same as the state properties for them – this makes updating the state very straightforward. Note that since the new state depends on the current state, we have used the function form of setState( ). Note how we carefully create a clone of the current state and change only portions that actually need change (the errors object is the exact same object as before, even "referentially").

```
updateValues = ( event ) => {
    const inputName = event.target.name;
    const inputValue = event.target.value;

    this.setState(
            ( curState ) => {
                return {
            ...curState,
            values: {
                        ...curState.values,
                        [inputName]: inputValue
            }
              };
        },
    );
}
```

In render, we destructure the input values from state.

```
const { sequenceId, name, speaker, duration, level, abstract } =
this.state.values;
```

We set the onChange listener and the value prop from the corresponding state property. Only one input control is shown here – **setup onChange and value similarly for all other input controls (except the buttons).**

```
<input type="text" className="form-control" name="sequenceId"
id="sequenceId" placeholder="" aria-describedby="sequenceHelpId"
onChange={this.updateValues} value={sequenceId} />
```

Let's validate the form inputs once an input changes. It may be overkill to validate each input on change of a single input – however there are cases when validating more than one input is necessary when an input changes (eg. a confirm password input exists along with password input). We adopt the simple approach of validating all inputs on change of any input.

First, create the validateForm( ) method and have it called once updateValues( ) changes the state.

```
validateForm = () => {

}

updateValues = ( event ) => {
    const inputName = event.target.name;
    const inputValue = event.target.value;

    this.setState(
            ( curState ) => {
                return {
        ...curState,
        values: {
                        ...curState.values,
                        [inputName]: inputValue
        }
          };
      },
      this.validateForm
    );
}
```

Next, set up an object to maintain errors in each input. Since there can be multiple errors in an input, we set these as arrays. We also destructure state properties into variables.

```
validateForm = () => {
   const errors = {
      sequenceId: [],
      name: [],
      speaker: [],
      duration: [],
      level: [],
      abstract: []
   };

         const { sequenceId, name, speaker, duration, level, abstract } = this.state.values;
}
```

Next, validate the input values according to business rules, and set up any errors encountered in the errors object. A sample is provided below.

```
if( sequenceId === '' ) {
   errors.sequenceId.push( 'Sequence ID is required' );
}
```

The supplied files has AddSession-validations.js that has the validations set up for you. It makes heavy use of regular expressions for validating the values. You can copy it over to the code within validateForm( ).

Setup the errors object on the state – the page re-renders with the error messages. For convenience, we also create an isValid( ) method that returns true if no input has errors, else false (the logic that sets this variable is left for you to ponder upon and understand).

```
isValid = () => {
      return Object.values( this.state.errors ).every( arr => arr.length === 0 );
   }
```

Inside render(), we destructure the errors – we rename the variables so that they do no clash with variable names we have destructured input values with.

```
const {
   sequenceId: sequenceIdErrs,
   name : nameErrs,
   speaker: speakerErrs,
   duration: durationErrs,
   level: levelErrs,
   abstract: abstractErrs
} = this.state.errors;
```

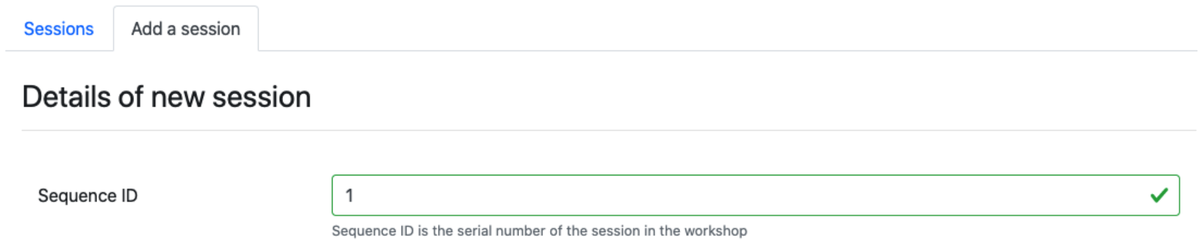
We disable default validation by the browser on form submission – this is done by adding a noValidate prop to the form element (this is a good practice) - notice that the prop is camelCased.

```
<form noValidate>
```

Next, we display the error messages for the inputs. This requires some Bootstrap classes to be conditionally assigned to indicate validity/invalidity. We show the code for only the sequenceId field. Setting up error message and display states for rest of inputs is left as an exercise.

```
<div className="col-sm-9">
        <input className={`form-control ${sequenceIdErrs.length === 0 ? 'is-valid' : 'is-invalid'}`}
type="text" name="sequenceId" id="sequenceId" placeholder="" aria-
describedby="sequenceHelpId" onChange={this.updateValues} value={sequenceId} />
        <small id="sequenceHelpId" className="text-muted">Sequence ID is the serial number
of the session in the workshop</small>
        <small className="invalid-feedback">
        {sequenceIdErrs.map(error => <div key={error}>{error}</div>)}
        </small>
</div>
```

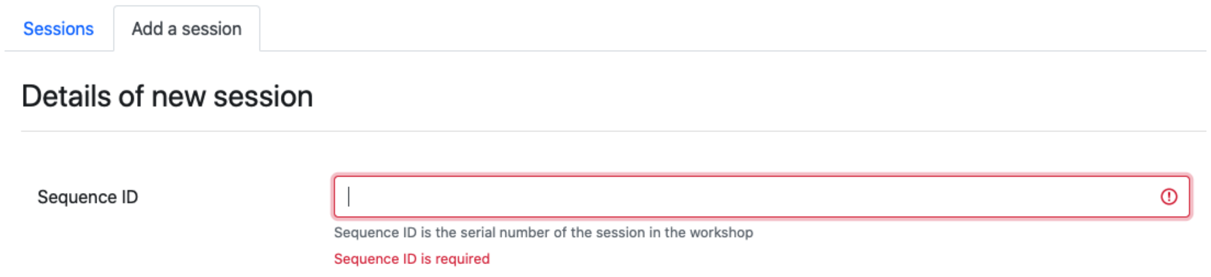
The input should function like so on valid and invalid inputs



Details of new session

Sequence ID 1

Sequence ID is the serial number of the session in the workshop



Details of new session

Sequence ID

Sequence ID is the serial number of the session in the workshop

Sequence ID is required

We disable the submit button if form is invalid (some input is invalid), and enable it otherwise.

```
<div className="form-group row">
        <div className="offset-sm-3 col-sm-9">
        <button type="submit" className="btn btn-primary mr-2" disabled={!this.isValid()}>Add
session</button>
            <button type="button" className="btn btn-danger">Cancel</button>
        </div>
</div>
```

When user fill in details and submits the form, we would need to submit the details to the backend. We define an addSession( ) method to do so and have it called on form submit. In it we first prevent the default form submission.

Inside render( )

```
<form noValidate onSubmit={this.addSession}>
```

And in a new method addSession( )

```
addSession = ( event ) => {
        event.preventDefault();
}
```

In services/sessions.js, define a method that adds a new session to the backend when passed the session details, along with the id of the workshop for which it is being added as a session. We make sure to send data as number data type, where expected so. Initially there are no votes on the new session, hence upvoteCount is set to 0.

```
addSession( session, workshopId ) {
   const sessionCopy = { ...session, workshopId: parseInt( workshopId ) };
   sessionCopy.sequenceId = parseInt( sessionCopy.sequenceId );
   sessionCopy.duration = parseFloat( sessionCopy.duration );
   session.upvoteCount = 0;

   return axios.post(
      `${this.apiBaseUrl}/sessions`,
      sessionCopy,
      {
         headers: {
            'Content-Type': 'application/json'
         }
```

```
        }
    ).then( response => response.data );
}
```

The axios.post() method accepts, the URL of the service endpoint, the data to be sent along with the outgoing POST request (which is converted to JSON string), and any headers to add to the request – here the 'Content-Type' is set to 'application/json' to indicate to the backend the format in which the data is being sent. The response data contains the session along with a unique id value that is generated when it is added to the backend.

We make use of the addSession( ) service method, in the AddSession component's addSession( ) method.

```
import SessionsService from '../services/sessions.js';
```

```
addSession = ( event ) => {
    event.preventDefault();

    SessionsService.addSession( this.state.values, this.props.match.params.id )
        .then(session => {
            alert( `The session has been added successfully (session id = ${session.id})` );
        })
        .catch( error => alert( `Something went wrong (${error.message})` ) );
}
```

You should now be able to add new sessions through the form.