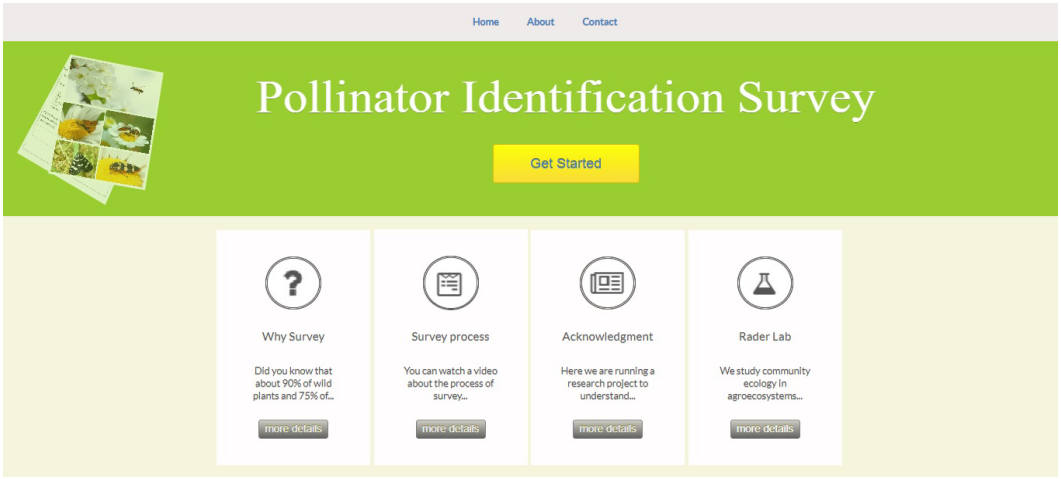


Pollinator Identification Survey

User guide
2018

An online presentation from this project may be found here:
https://www.youtube.com/watch?v=jHXCpOn1bd8&t=157s

Table of Contents

Supervisor

Dr. David Paul

Client

Earthwatch Institute
Key contact: Dr. Romina Rader
Email: rrader@une.edu.au
Website: http://au.earthwatch.org/
Ph: (03) 9016 7590
Lab : http://www.raderlab.com

Project Summary

The project is called the Pollinator Insects Online Survey Project (PIOSP). Its purpose is to develop a survey website as a part of PMP (Pollinators Monitoring Project). The PMP is being designed by multiple teams of varying skill levels, from Masters to Undergraduate students and consists of numerous components.

The focus of our COSC592 team is the online survey on pollinator insects. The participants will be required to undertake a short online survey, in which they will see a series of photos of pollinators on flowers and be asked to identify each. All potential participants should pass a validation question to prevent multiple attempts from the same users. Once the survey starts, it is expected that this survey will take between 5 and 15 minutes to complete. Participants will be able to undertake this survey from their computer or smartphone and will be able to complete it at any stage during the survey period. The data that results from the surveys will be used to generate an accuracy rate for human observers in the identification of five pollinator groups (bees, flies, beetles, wasps, moths/butterflies).

Getting Started

Please read the topics bellow to quickly get started with PIOSP app.
It requires just a couple minutes to see how it's easy to run.

Pre-requirements

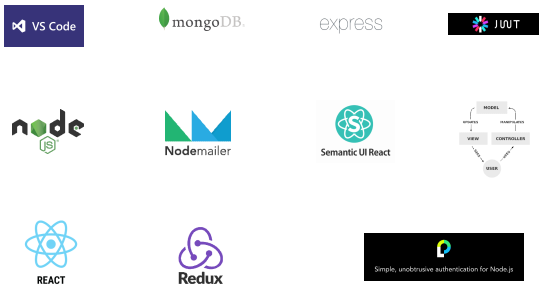
- **Server-side technology:**
  Express.js
  Node.js

- **Client-side:**
  React.js
  Redux
  Semantic UI React Library

- **Other Technology:**
  RESTAPI
  Json Web Token (JWT) for Authentication

- **Framework:**
  Node.js+ mongo + express Model-View-Controller (MVC) API

- **Database:**
  MongoDB



Server Node Server

1. Installing Node
    a. Windows and macOS
        1.a.1. Installing Node and NPM on Windows and macOS is straightforward because you can just use the provided installer:
        1.a.2. Download the required installer:
        1.a.3. Go to here
        1.a.4. Select the button to download the LTS build that is "Recommended for most users".
        1.a.5. Install Node by double-clicking on the downloaded file and following the installation prompts.
    b. Ubuntu 16.04
        The easiest way to install the most recent LTS version of Node 6.x is to use the package manager to get it from the Ubuntu binary distributions repository. This can be done very simply by running the following two commands on your terminal:

        curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
        sudo apt-get install -y nodejs

        Warning: Don't install directly from the normal Ubuntu repositories because they contain very old versions of node.

2. Install MongoDB from here

3. Clone this repository

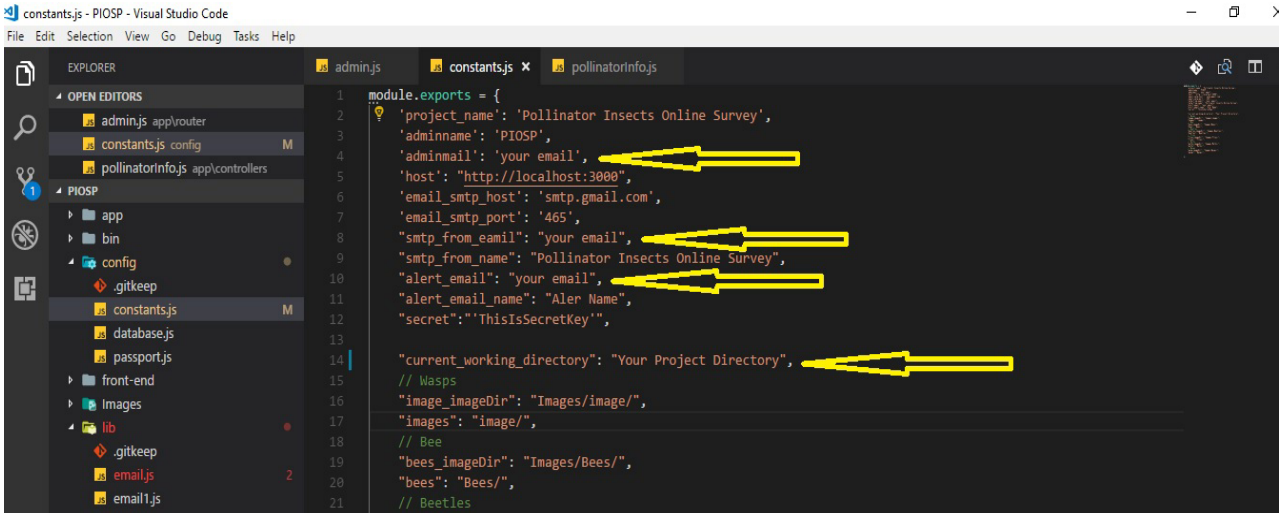
        $ git clone https://gitlab.une.edu.au/piosp/PIOSP

Note: You can download project folder or if you already have the project you don't need to clone the repository.

4. Change directories

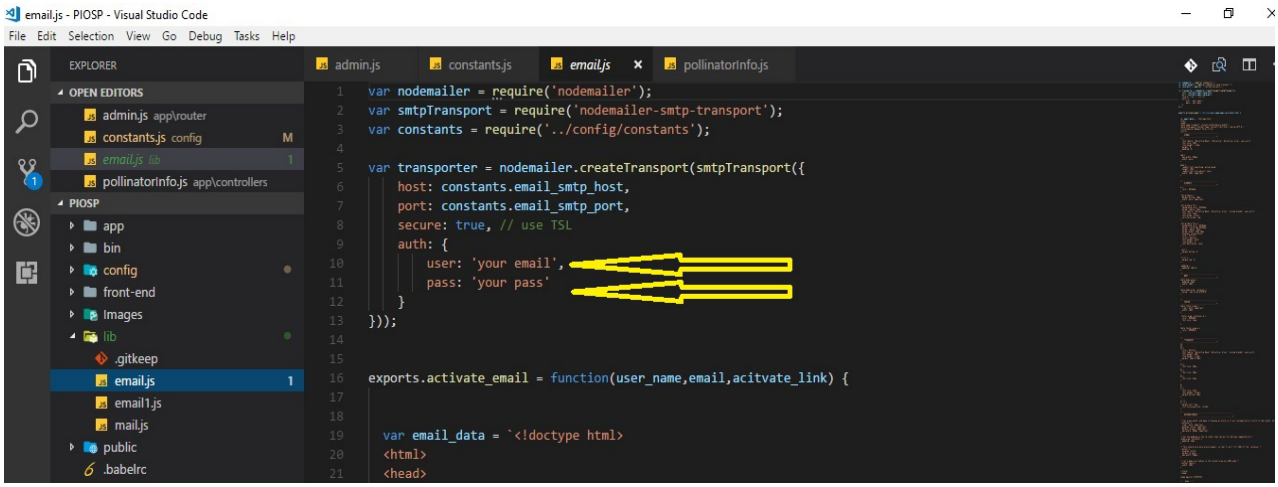
        $ cd PIOSP

5. In the project explorer go to the config folder and open constants.js file. Change this variable to your desire variable :



    Note: For email that you want to use here should turn on 'less secure app' option in your account, for example for gmail account you can do this here

6. Go to lib folder and open email.js file. Change this variable to your variable that you insert above :



6. Run MongoDB (Run in this port=> mongodb://127.1.1.0/)

5. In terminal Type:

    $ npm install
    $ node bin/www.js

You will see this message:

Now server is running in http://localhost:3000
You can test all routes and functionality by Postman(get here) or other API development environment.

Note: If after running running command in step 5 you will receive message like the image below you need to install that module separately
by command (In Microsoft Windows, macOS, Ubuntu, Debian):

$ npm install --save <name of module>

Running Client Server

1. Open new terminal and go to your project directory

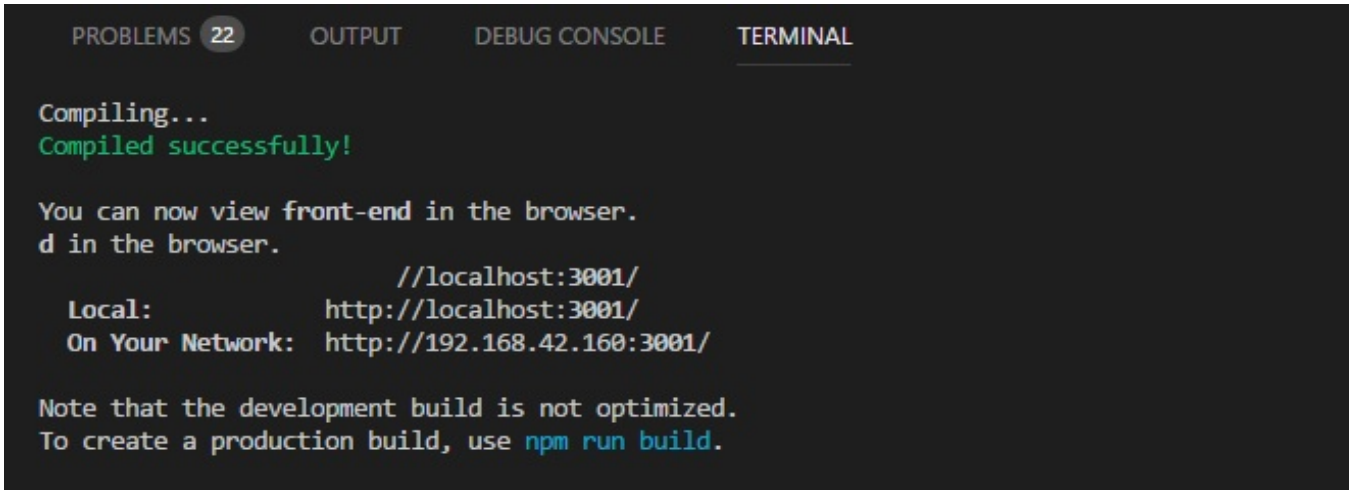
2. Change directories

    $ cd front-end

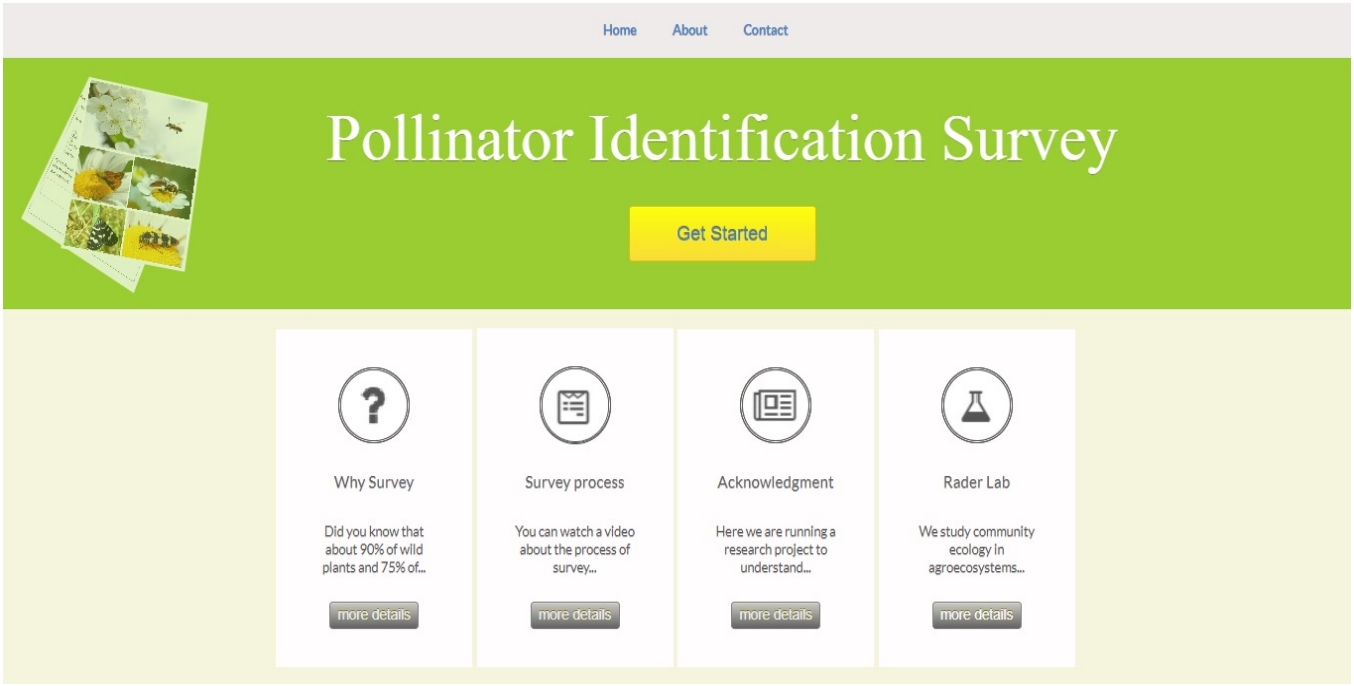
3. Type:

    $ npm install
    $ npm start

4. You will see this message:



Now the app is running https://localhost:3001 (if port 3001 is free)and it will open a browser and show the main page of the web app.



Note: If after running running command in step 3 you will receive message like the image below you need to install that module separately
by command (In Microsoft Windows, macOS, Ubuntu, Debian):

$ npm install --save <name of module>

```
node.js:201
        throw e; // process.nextTick error, or 'error' event on first tick
              ^
Error: Cannot find module './layer'
    at Function._resolveFilename (module.js:332:11)
    at Function._load (module.js:279:25)
    at Module.require (module.js:354:17)
    at require (module.js:370:17)
    at Object.<anonymous> (/home/downloadserver/node_modules/express/lib/router/route.js:6:13)
    at Module._compile (module.js:441:26)
    at Object..js (module.js:459:10)
    at Module.load (module.js:348:32)
    at Function._load (module.js:308:12)
    at Module.require (module.js:354:17)
```

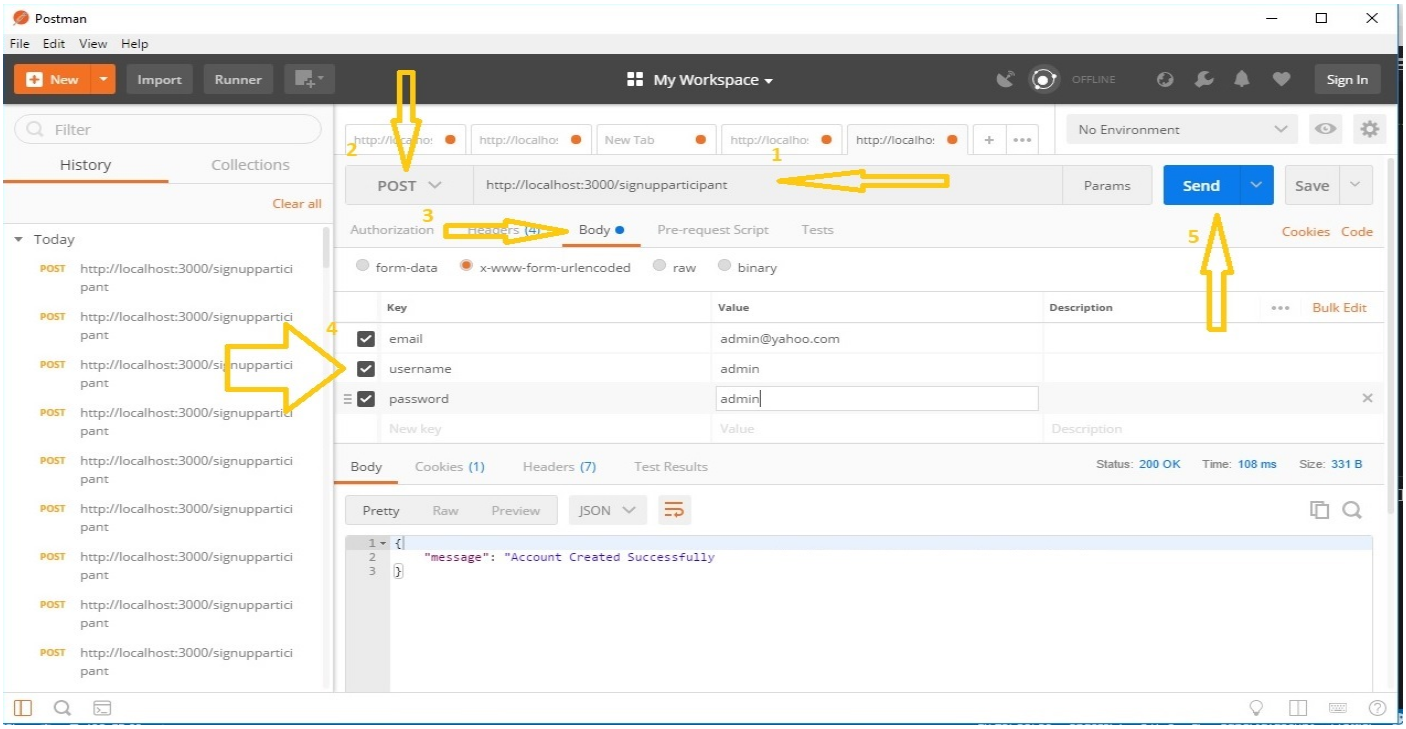
Register Admin

For registering main admin we have some steps to do:

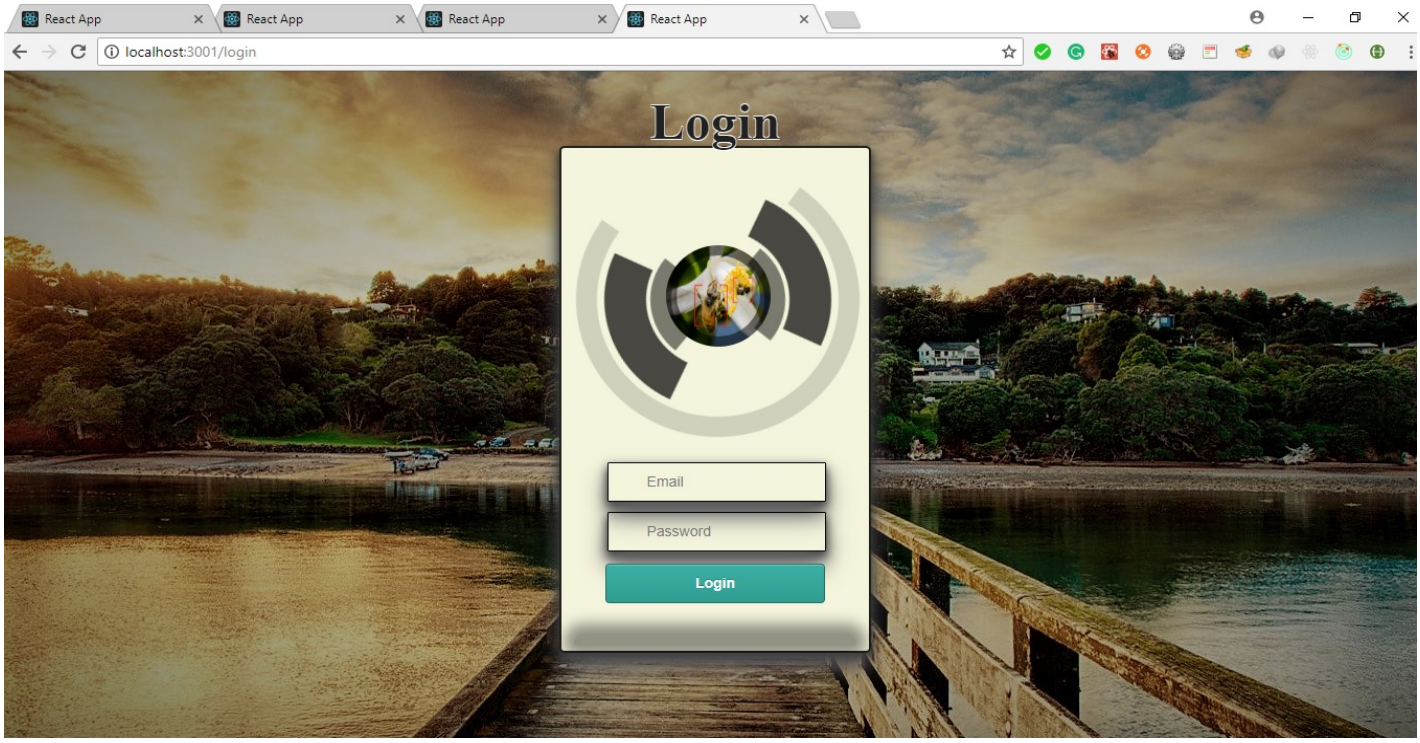
1. We have just one main admin who other sub admin doesn't have access to her/his account.
So we are registering first and main admin through postman.

2. After you install postman, type http://localhost:3000/signupparticipant, select post , in body add email, username and password and at the end press send.

Now Admin account created.



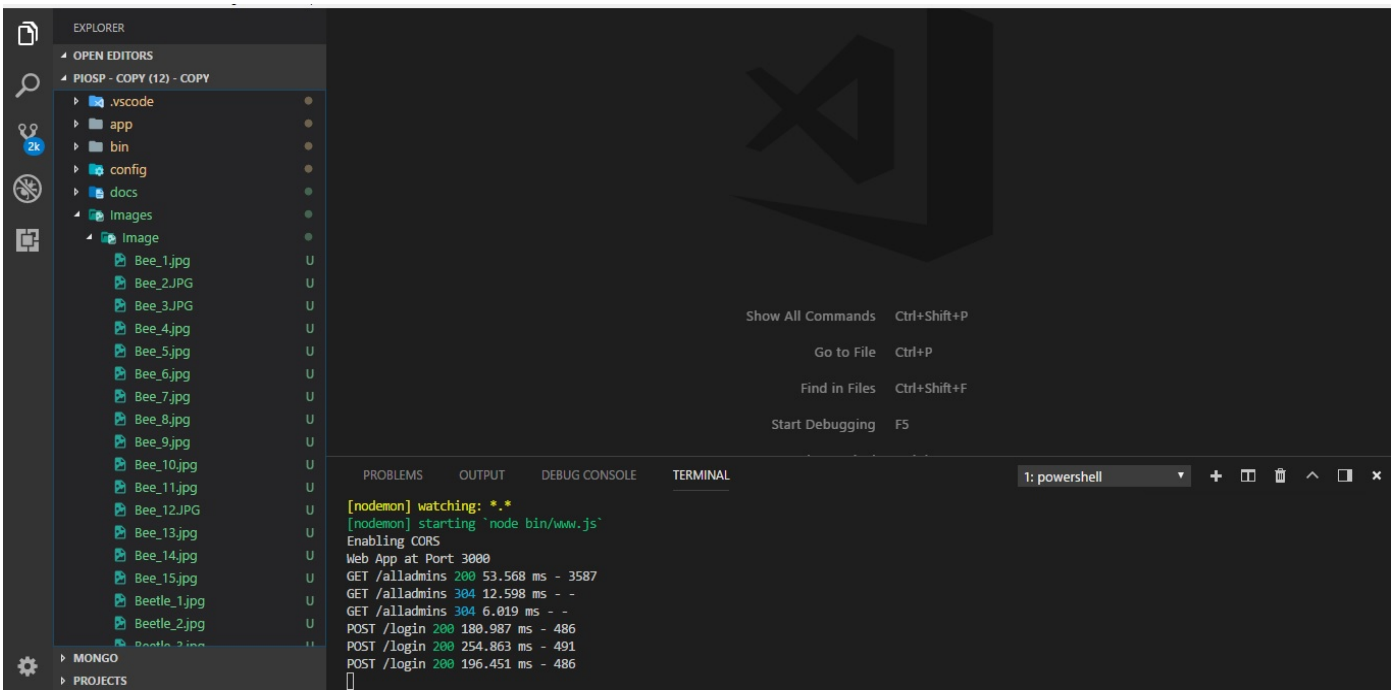
3. Now you can login to dashboard by http://localhost:3001/login


React App
React App
React App
React App
localhost:3001/login
Login
Email
Password
Login

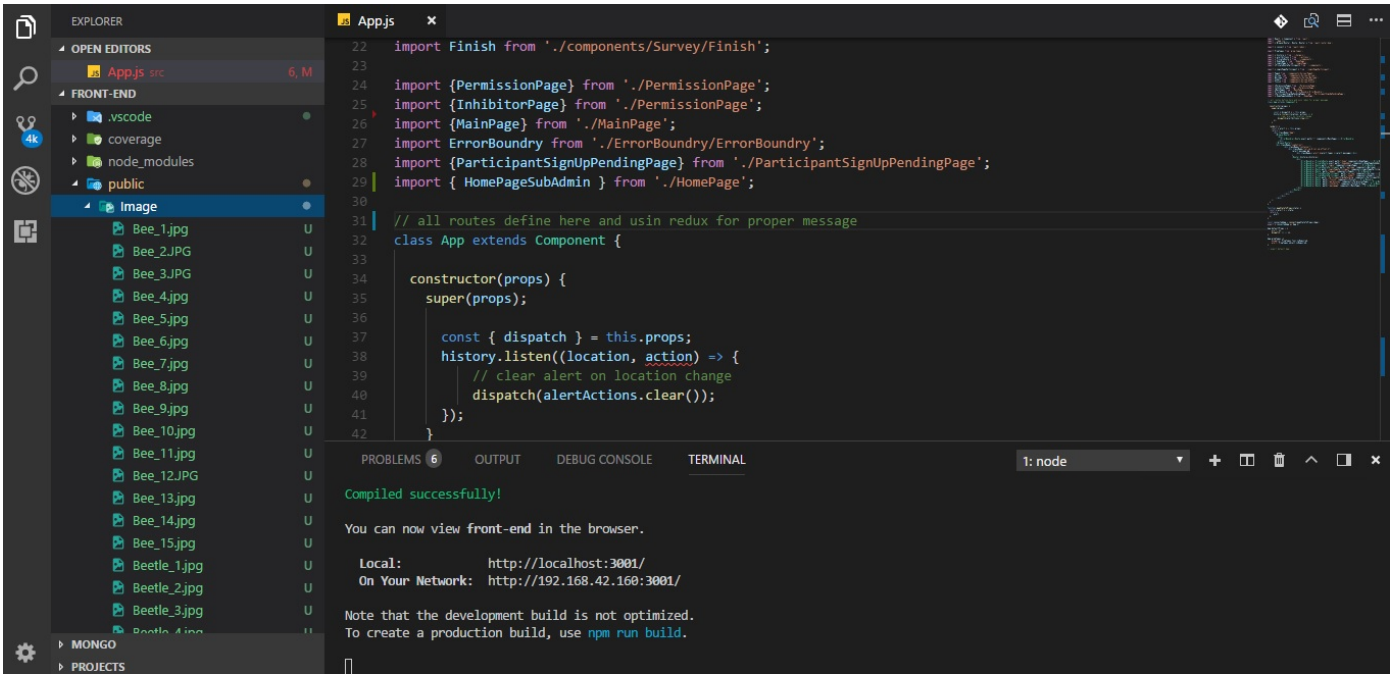
Filling Pollinator PhotoTable

For filling the pollinator table for the first time we have some steps to do:

1. Make sure all photos are in this directory in the Node server : PIOSP/Images/Image

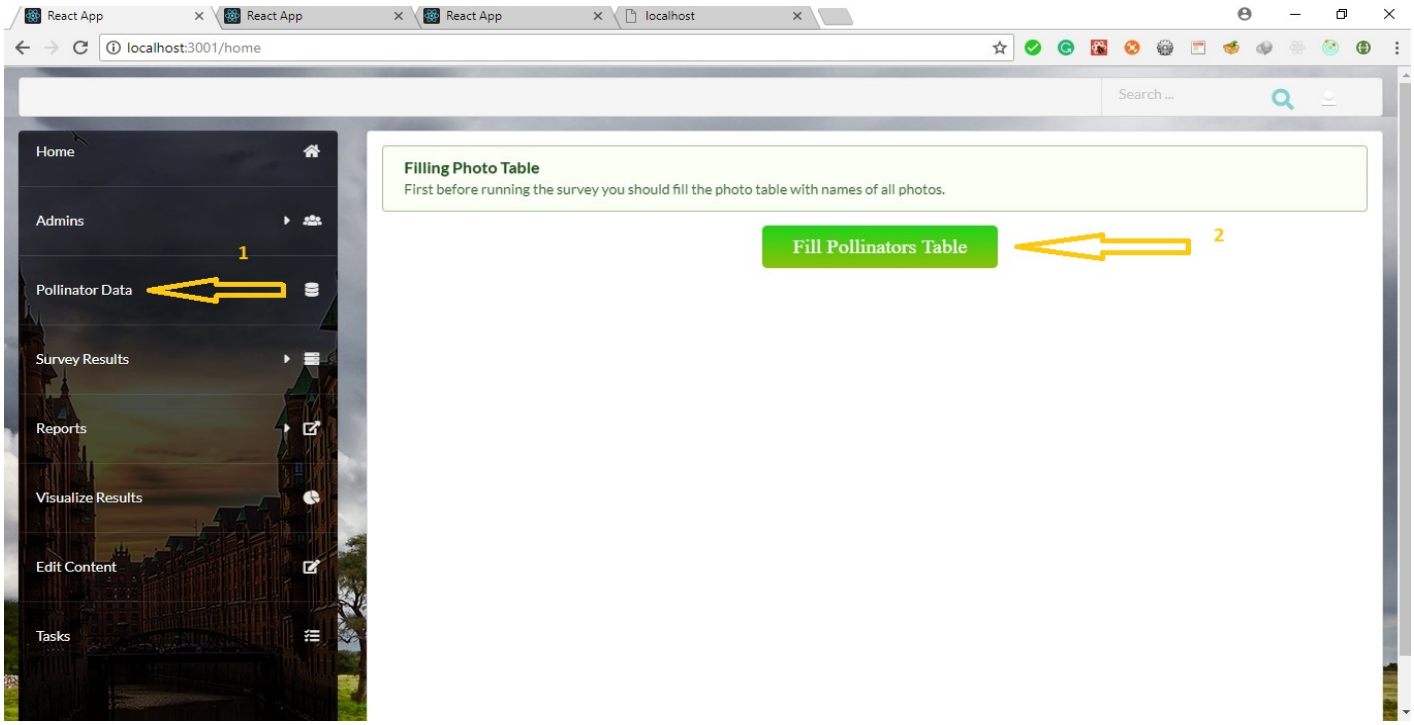


2. Make sure all photos are in this directory in the Client server : front-end/public/Image



Note: There is other foloder 'img' that includes photos for background , main page, logo, covers, .etc

3. Login to dashboard and from menu choose 'Pollinator Data' and click on 'Fill Pollinator Table'.



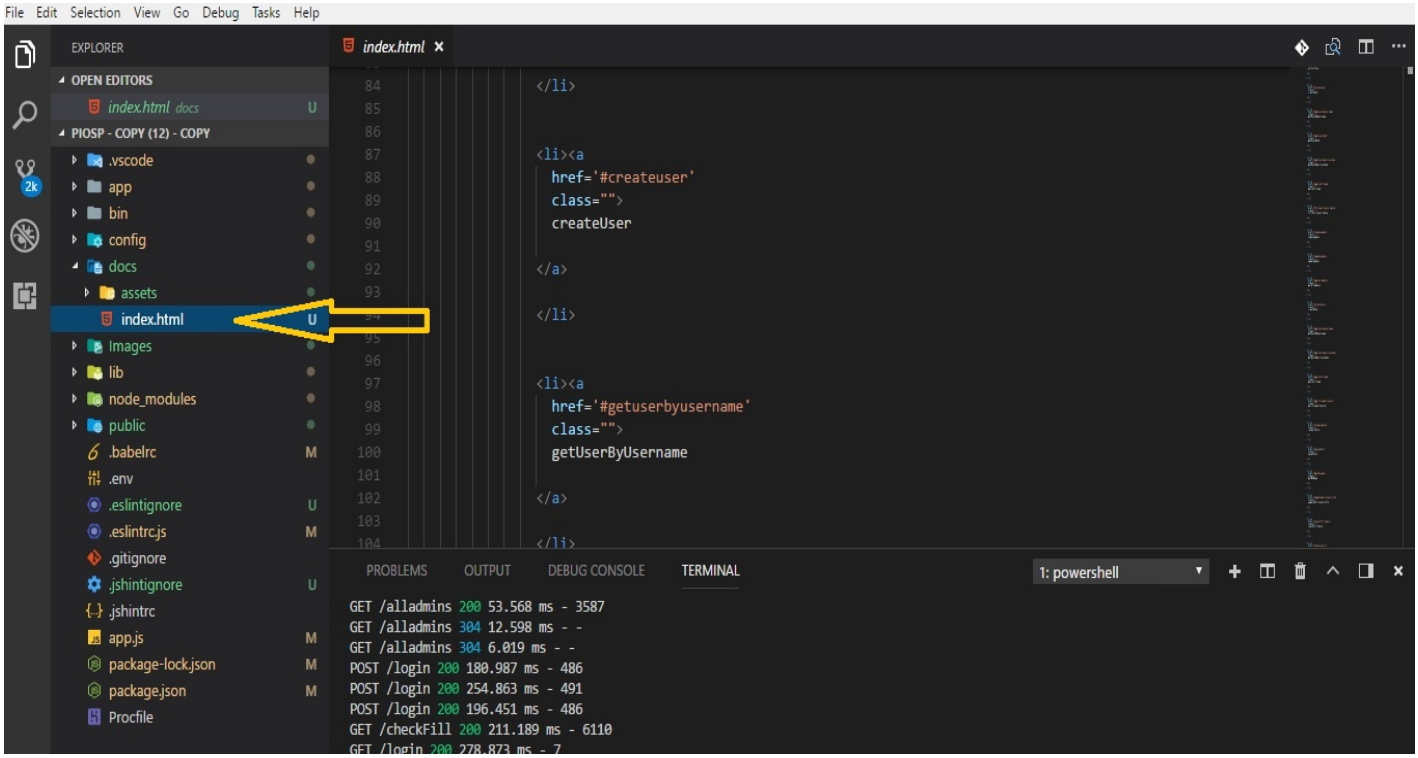
It will fills Pollinator table with pollinators name for select and render in survey.
It will check and if you click several times table will not refill and it prevents from duplicate in showing photos and mistake in several clicks.

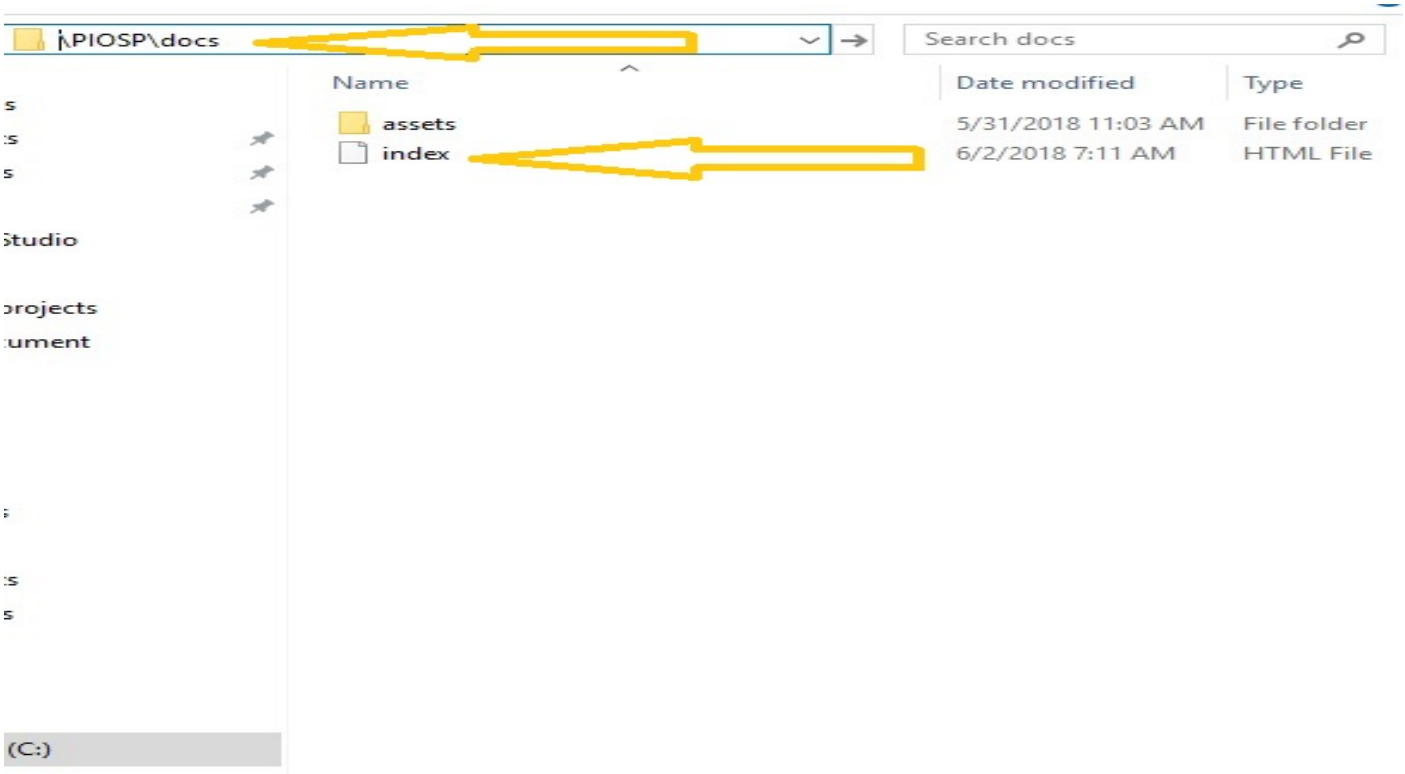
Node Server Modules

There is a html file in folder in 'docs' folder in the Node Server directory that includes all names, description, prameters and returns of each module and functions. Also includes routers comment. This file documented modules and functions in Node Server.

All these functions and modules commented inside as well.
It can open without running application.

Node Server Size : 71.1 MB





Client Server Components

AdminListPage: Work with list of Admins , can delete, update or register new admin

App.js: Include all routes

components: Include sub folders, stylesheets and Survey
Survey: Includes these components :

- **FillingTable:** filling the pollinator table when we are running the server for the first time
- **Finish:** last page of survey
- **NotFound:** Route not included in routing
- **Page2:** second page of survey, In this page we are receiving name and email of participant and sending through server, after this page we have participant pending page
- **Submit:** after participant answers questions, survey will redirect to this page for last part, in this page participants rate their experience about survey and give information about their pollination research
- **Survey:** render after welcome page, main part of survey, show questions and photos and send answers through server
- **Welcome:** first page after participants click on the link that have been sent to them, after clicking on the start seurvey button page will redirect to first page of survey that shows questions

ErrorBoundry: React new sterategy for handling components error

HomePage: customized dashbord component for main admin and sub admin, main admin have access to list of admins and can update, delete or add admins.

index.js: all components routed inside app component and rendering in a single html page, all wrapping in store for using in redux

LoginPage: login page ,user credential will go through action in redux

LoginPageParticipant: this component using jwt in action function and, in action we check the json web token make sure participant is authenticated to atthend the survey

MainPage: render the main page of component, this page shows at the main url http://localhost:3001/ , http://localhost:3001/mainpage

Pagination: reusable component that used several parts of survey, when we need to fetch high number of records to increase fetching speed

ParticipantSignUpPendingPage: after page 2 when participants submit their name and email, they will redirect to this page as a pending page

PermissionPage: Including inhibitor, permission and newuser components for prevent attending more than one time , If participant attended the survey before this inhibitor component will render, It show message to the participant for their previous attend and , they can not go to the survey questions page

RegisterPage: render register page and admin can register sub admins

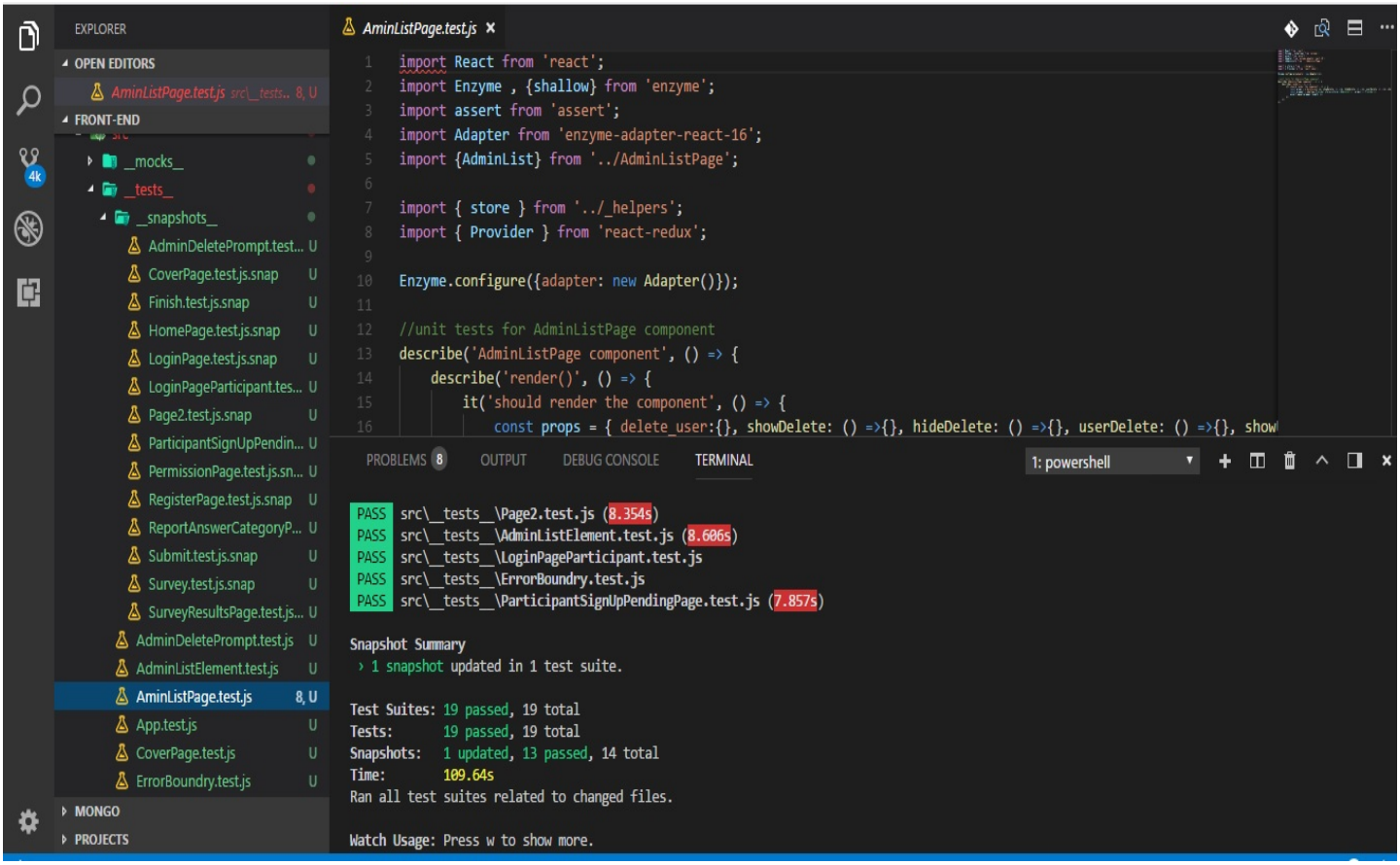
ReportAnswerCategoryPage: includes several components with list of admins and answers that we can have query between list create csv file

SurveyResultsPage: we have a dropdown list to show number of correct and incorrect answers base on category, to visualize we have options between doughnut chart, bar chart , line chart ,and pie chart

_actions, _components, _constants, _helpers, _reducers, _services : folder for redux implementation, we are using redux for managing admin data and some part of results data
__mocks__ : mock some function to use in test with jest
__tests__: test functions

Tests Client Server

We have used jest and enzyme for unit test and shallow test.
Also we have using snapshot to watch each change in selected components.



Client Side Development Instructions

For the project to build, **these files must exist with exact filenames**:

* `public/index.html` is the page template;
* `src/index.js` is the JavaScript entry point.

You can delete or rename the other files.

You may create subdirectories inside `src`. For faster rebuilds, only files inside `src` are processed by Webpack.
You need to **put any JS and CSS files inside `src`**, otherwise Webpack won't see them.

Only files inside `public` can be used from `public/index.html`.
Read instructions below for using assets from JavaScript and HTML.

You can, however, create more top-level directories.
They will not be included in the production build so you can use them for things like documentation.

Available Scripts

In the project directory, you can run:

`npm start`

Runs the app in the development mode.
Open http://localhost:3000 to view it in the browser.

The page will reload if you make edits.
You will also see any lint errors in the console.

`npm test`

Launches the test runner in the interactive watch mode.

`npm run build`

Builds the app for production to the `build` folder.
It correctly bundles React in production mode and optimizes the build for the best performance.

The build is minified and the filenames include the hashes.
App is ready to be deployed!

`npm run eject`

Note: this is a one-way operation. Once you `eject`, you can't go back!

If you aren't satisfied with the build tool and configuration choices, you can `eject` at any time. This command will remove the single build dependency from your project.

Instead, it will copy all the configuration files and the transitive dependencies (Webpack, Babel, ESLint, etc) right into your project so you have full control over them. All of the commands except `eject` will still work, but they will point to the copied scripts so you can tweak them. At this point you're on your own.

You don’t have to ever use `eject`. The curated feature set is suitable for small and middle deployments, and you shouldn’t feel obligated to use this feature. However we understand that this tool wouldn’t be useful if you couldn’t customize it when you are ready for it.

Supported Browsers

By default, the generated project uses the latest version of React.

All test shows that it's working on all browsers.
You can refer [to the React documentation](https://reactjs.org/docs/react-dom.html#browser-support) for more information about supported browsers.

Supported Language Features and Polyfills

This project supports a superset of the latest JavaScript standard.
In addition to [ES6](https://github.com/lukehoban/es6features) syntax features, it also supports:

* [Exponentiation Operator](https://github.com/rwaldron/exponentiation-operator) (ES2016).
* [Async/await](https://github.com/tc39/ecmascript-asyncawait) (ES2017).
* [Object Rest/Spread Properties](https://github.com/sebmarkbage/ecmascript-rest-spread) (stage 3 proposal).
* [Dynamic import()](https://github.com/tc39/proposal-dynamic-import) (stage 3 proposal)
* [Class Fields and Static Properties](https://github.com/tc39/proposal-class-public-fields) (part of stage 3 proposal).
* [JSX](https://facebook.github.io/react/docs/introducing-jsx.html) and [Flow](https://flowtype.org/) syntax.

Syntax Highlighting in the Editor

To configure the syntax highlighting in your favorite text editor, head to the [relevant Babel documentation page](https://babeljs.io/docs/editors) and follow the instructions. Some of the most popular editors are covered.

Displaying Lint Output in the Editor

>Note: this feature is available with `react-scripts@0.2.0` and higher.
>It also only works with npm 3 or higher.

Some editors, including Sublime Text, Atom, and Visual Studio Code, provide plugins for ESLint.

They are not required for linting. You should see the linter output right in your terminal as well as the browser console. However, if you prefer the lint results to appear right in your editor, there are some extra steps you can do.

You would need to install an ESLint plugin for your editor first. Then, add a file called `.eslintrc` to the project root:

```js
{
  "extends": "react-app"
}
```

Now your editor should report the linting warnings.

Note that even if you edit your `.eslintrc` file further, these changes will **only affect the editor integration**. They won't affect the terminal and in-browser lint output. This is because Create React App intentionally provides a minimal set of rules that find common mistakes.

If you want to enforce a coding style for your project, consider using [Prettier](https://github.com/jlongster/prettier) instead of ESLint style rules.

Debugging in the Editor

This feature is currently only supported by [Visual Studio Code](https://code.visualstudio.com) and [WebStorm](https://www.jetbrains.com/webstorm/).

Visual Studio Code and WebStorm support debugging out of the box with Create React App. This enables you as a developer to write and debug your React code without leaving the editor, and most importantly it enables you to have a continuous development workflow, where context switching is minimal, as you don't have to switch between tools.

Installing a Dependency

The generated project includes React and ReactDOM as dependencies. It also includes a set of scripts used by Create React App as a development dependency. You may install other dependencies (for example, React Router) with `npm`:

```sh
npm install --save react-router
```

Alternatively you may use `yarn`:

```sh
yarn add react-router
```

This works for any library, not just `react-router`.

Importing a Component

This project setup supports ES6 modules thanks to Babel.
While you can still use `require()` and `module.exports`, we encourage you to use [`import` and `export`](http://exploringjs.com/es6/ch_modules.html) instead.

For example:

`Button.js`

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

class Button extends Component {
  render() {
    // ...
  }
}

export default Button; // Don't forget to use export default!
```

`DangerButton.js`

```js
import React, { Component } from 'react';
import Button from './Button'; // Import a component from another file

class DangerButton extends Component {
  render() {
    return <Button color="red" />;
  }
}

export default DangerButton;
```

Named exports are useful for utility modules that export several functions. A module may have at most one default export and as many named exports as you like.

Code Splitting

Instead of downloading the entire app before users can use it, code splitting allows you to split your code into small chunks which you can then load on demand.

This project setup supports code splitting via [dynamic `import()`](http://2ality.com/2017/01/import-operator.html#loading-code-on-demand). The `import()` function-like form takes the module name as an argument and returns a [`Promise`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) which always resolves to the namespace object of the module.

Here is an example:

`moduleA.js`

```js
const moduleA = 'Hello';

export { moduleA };
```

`App.js`

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

class App extends Component {
  handleClick = () => {
    import('./moduleA')
      .then(({ moduleA }) => {
        // Use moduleA
      })
      .catch(err => {
        // Handle failure
      });
  };
```

```
  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Load</button>
      </div>
    );
  }
}

export default App;
```

This will make `moduleA.js` and all its unique dependencies as a separate chunk that only loads after the user clicks the 'Load' button.

You can also use it with `async` / `await` syntax if you prefer it.

Also check out the [Code Splitting](https://reactjs.org/docs/code-splitting.html) section in React documentation.

Adding a Stylesheet

This project setup uses [Webpack](https://webpack.js.org/) for handling all assets. Webpack offers a custom way of “extending” the concept of `import` beyond JavaScript. To express that a JavaScript file depends on a CSS file, you need to **import the CSS from the JavaScript file**:

`Button.css`

```css
.Button {
  padding: 20px;
}
```

`Button.js`

```js
import React, { Component } from 'react';
import './Button.css'; // Tell Webpack that Button.js uses these styles

class Button extends Component {
  render() {
    // You can use them as regular CSS styles
    return <div className="Button" />;
  }
}
```

This is not required for React but many people find this feature convenient. You can read about the benefits of this approach [here](https://medium.com/seek-ui-engineering/block-element-modifying-your-javascript-components-d7f99fcab52b). However you should be aware that this makes your code less portable to other build tools and environments than Webpack.

In development, expressing dependencies this way allows your styles to be reloaded on the fly as you edit them. In production, all CSS files will be concatenated into a single minified `.css` file in the build output.

If you are concerned about using Webpack-specific semantics, you can put all your CSS right into `src/index.css`. It would still be imported from `src/index.js`, but you could always remove that import if you later migrate to a different build tool.

Post-Processing CSS

This project setup minifies your CSS and adds vendor prefixes to it automatically through [Autoprefixer](https://github.com/postcss/autoprefixer) so you don’t need to worry about it.

For example, this:

```css
.App {
  display: flex;
  flex-direction: row;
  align-items: center;
}
```

becomes this:

```css
.App {
  display: -webkit-box;
  display: -ms-flexbox;
  display: flex;
  -webkit-box-orient: horizontal;
  -webkit-box-direction: normal;
      -ms-flex-direction: row;
          flex-direction: row;
  -webkit-box-align: center;
      -ms-flex-align: center;
          align-items: center;
}
```

If you need to disable autoprefixing for some reason, [follow this section](https://github.com/postcss/autoprefixer#disabling).

Adding a CSS Preprocessor (Sass, Less etc.)

Generally, we recommend that you don’t reuse the same CSS classes across different components. For example, instead of using a `.Button` CSS class in `<AcceptButton>` and `<RejectButton>` components, we recommend creating a `<Button>` component with its own `.Button` styles, that both `<AcceptButton>` and `<RejectButton>` can render (but [not inherit](https://facebook.github.io/react/docs/composition-vs-inheritance.html)).

Following this rule often makes CSS preprocessors less useful, as features like mixins and nesting are replaced by component composition. You can, however, integrate a CSS preprocessor if you find it valuable. In this walkthrough, we will be using Sass, but you can also use Less, or another alternative.

First, let’s install the command-line interface for Sass:

```sh
npm install --save node-sass-chokidar
```

Alternatively you may use `yarn`:

```sh
yarn add node-sass-chokidar
```

Then in `package.json`, add the following lines to `scripts`:

```diff
  "scripts": {
+    "build-css": "node-sass-chokidar src/ -o src/",
+    "watch-css": "npm run build-css && node-sass-chokidar src/ -o src/ --watch --recursive",
     "start": "react-scripts start",
     "build": "react-scripts build",
     "test": "react-scripts test --env=jsdom",
```

>Note: To use a different preprocessor, replace `build-css` and `watch-css` commands according to your preprocessor’s documentation.

Now you can rename `src/App.css` to `src/App.scss` and run `npm run watch-css`. The watcher will find every Sass file in `src` subdirectories, and create a corresponding CSS file next to it, in our case overwriting `src/App.css`. Since `src/App.js` still imports `src/App.css`, the styles become a part of your application. You can now edit `src/App.scss`, and `src/App.css` will be regenerated.

To share variables between Sass files, you can use Sass imports. For example, `src/App.scss` and other component style files could include `@import "./shared.scss";` with variable definitions.

To enable importing files without using relative paths, we add the `--include-path` option to the command in `package.json`.

```
"build-css": "node-sass-chokidar --include-path ./src --include-path ./node_modules src/ -o src/",
"watch-css": "npm run build-css && node-sass-chokidar --include-path ./src --include-path ./
node_modules src/ -o src/ --watch --recursive",
```

This will allow you to do imports like

```scss
@import 'styles/_colors.scss'; // assuming a styles directory under src/
@import 'nprogress/nprogress'; // importing a css file from the nprogress node module
```

At this point you might want to remove all CSS files from the source control, and add `src/**/*.css` to your `.gitignore` file. It is generally a good practice to keep the build products outside of the source control.

As a final step, you may find it convenient to run `watch-css` automatically with `npm start`, and run `build-css` as a part of `npm run build`. You can use the `&&` operator to execute two scripts

sequentially. However, there is no cross-platform way to run two scripts in parallel, so we will install a package for this:

```sh
npm install --save npm-run-all
```

Alternatively you may use `yarn`:

```sh
yarn add npm-run-all
```

Then we can change `start` and `build` scripts to include the CSS preprocessor commands:

```diff
  "scripts": {
    "build-css": "node-sass-chokidar src/ -o src/",
    "watch-css": "npm run build-css && node-sass-chokidar src/ -o src/ --watch --recursive",
-   "start": "react-scripts start",
-   "build": "react-scripts build",
+   "start-js": "react-scripts start",
+   "start": "npm-run-all -p watch-css start-js",
+   "build-js": "react-scripts build",
+   "build": "npm-run-all build-css build-js",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
```

Now running `npm start` and `npm run build` also builds Sass files.

Why `node-sass-chokidar`?

`node-sass` has been reported as having the following issues:

- `node-sass --watch` has been reported to have *performance issues* in certain conditions when used in a virtual machine or with docker.

- Infinite styles compiling

- `node-sass` has been reported as having issues with detecting new files in a directory

`node-sass-chokidar` is used here as it addresses these issues.

Adding Images, Fonts, and Files

With Webpack, using static assets like images and fonts works similarly to CSS.

You can **`import` a file right in a JavaScript module**. This tells Webpack to include that file in the bundle. Unlike CSS imports, importing a file gives you a string value. This value is the final path you can reference in your code, e.g. as the `src` attribute of an image or the `href` of a link to a PDF.

To reduce the number of requests to the server, importing images that are less than 10,000 bytes returns a [data URI](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/

Data_URIs) instead of a path. This applies to the following file extensions: bmp, gif, jpg, jpeg, and png. SVG files are excluded due to [#1153](https://github.com/facebookincubator/create-react-app/issues/1153).

Here is an example:

```js
import React from 'react';
import logo from './logo.png'; // Tell Webpack this JS file uses this image

console.log(logo); // /logo.84287d09.png

function Header() {
  // Import result is the URL of your image
  return <img src={logo} alt="Logo" />;
}

export default Header;
```

This ensures that when the project is built, Webpack will correctly move the images into the build folder, and provide us with correct paths.

This works in CSS too:

```css
.Logo {
  background-image: url(./logo.png);
}
```

Webpack finds all relative module references in CSS (they start with `./`) and replaces them with the final paths from the compiled bundle. If you make a typo or accidentally delete an important file, you will see a compilation error, just like when you import a non-existent JavaScript module. The final filenames in the compiled bundle are generated by Webpack from content hashes. If the file content changes in the future, Webpack will give it a different name in production so you don’t need to worry about long-term caching of assets.

Please be advised that this is also a custom feature of Webpack.

Using the `public` Folder

>Note: this feature is available with `react-scripts@0.5.0` and higher.

Changing the HTML

The `public` folder contains the HTML file so you can tweak it, for example, to [set the page title](#changing-the-page-title).
The `<script>` tag with the compiled code will be added to it automatically during the build process.

Adding Assets Outside of the Module System

You can also add other assets to the `public` folder.

Note that we normally encourage you to `import` assets in JavaScript files instead.
For example, see the sections on [adding a stylesheet](#adding-a-stylesheet) and [adding images and fonts](#adding-images-fonts-and-files).
This mechanism provides a number of benefits:

* Scripts and stylesheets get minified and bundled together to avoid extra network requests.
* Missing files cause compilation errors instead of 404 errors for your users.
* Result filenames include content hashes so you don't need to worry about browsers caching their old versions.

However there is an **escape hatch** that you can use to add an asset outside of the module system.

If you put a file into the `public` folder, it will **not** be processed by Webpack. Instead it will be copied into the build folder untouched. To reference assets in the `public` folder, you need to use a special variable called `PUBLIC_URL`.

Inside `index.html`, you can use it like this:

```html
<link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
```

Only files inside the `public` folder will be accessible by `%PUBLIC_URL%` prefix. If you need to use a file from `src` or `node_modules`, you'll have to copy it there to explicitly specify your intention to make this file a part of the build.

When you run `npm run build`, Create React App will substitute `%PUBLIC_URL%` with a correct absolute path so your project works even if you use client-side routing or host it at a non-root URL.

In JavaScript code, you can use `process.env.PUBLIC_URL` for similar purposes:

```js
render() {
  // Note: this is an escape hatch and should be used sparingly!
  // Normally we recommend using `import` for getting asset URLs
  // as described in "Adding Images and Fonts" above this section.
  return <img src={process.env.PUBLIC_URL + '/img/logo.png'} />;
}
```

Keep in mind the downsides of this approach:

* None of the files in `public` folder get post-processed or minified.
* Missing files will not be called at compilation time, and will cause 404 errors for your users.
* Result filenames won't include content hashes so you'll need to add query arguments or rename them every time they change.

When to Use the `public` Folder

Normally we recommend importing [stylesheets](#adding-a-stylesheet), [images, and fonts](#adding-images-fonts-and-files) from JavaScript.
The `public` folder is useful as a workaround for a number of less common cases:

* You need a file with a specific name in the build output, such as [`manifest.webmanifest`](https://developer.mozilla.org/en-US/docs/Web/Manifest).

* You have thousands of images and need to dynamically reference their paths.
* You want to include a small script like [`pace.js`](http://github.hubspot.com/pace/docs/welcome/) outside of the bundled code.
* Some library may be incompatible with Webpack and you have no other option but to include it as a `<script>` tag.

Note that if you add a `<script>` that declares global variables, you also need to read the next section on using them.

Adding Bootstrap

You don’t have to use [React Bootstrap](https://react-bootstrap.github.io) together with React but it is a popular library for integrating Bootstrap with React apps. If you need it, you can integrate it with Create React App by following these steps:

Install React Bootstrap and Bootstrap from npm. React Bootstrap does not include Bootstrap CSS so this needs to be installed as well:

```sh
npm install --save react-bootstrap bootstrap@3
```

Alternatively you may use `yarn`:

```sh
yarn add react-bootstrap bootstrap@3
```

Import Bootstrap CSS and optionally Bootstrap theme CSS in the beginning of your ```src/index.js``` file:

```js
import 'bootstrap/dist/css/bootstrap.css';
import 'bootstrap/dist/css/bootstrap-theme.css';
// Put any other imports below so that CSS from your
// components takes precedence over default styles.
```

Import required React Bootstrap components within ```src/App.js``` file or your custom component files:

```js
import { Navbar, Jumbotron, Button } from 'react-bootstrap';
```

Now you are ready to use the imported React Bootstrap components within your component hierarchy defined in the render method. Here is an example [`App.js`](https://gist.githubusercontent.com/gaearon/85d8c067f6af1e56277c82d19fd4da7b/raw/6158dd991b67284e9fc8d70b9d973efe87659d72/App.js) redone using React Bootstrap.

Adding a Router

Create React App doesn't prescribe a specific routing solution, but [React Router](https://reacttraining.com/react-router/) is the most popular one.

To add it, run:

```sh
npm install --save react-router-dom
```

Alternatively you may use `yarn`:

```sh
yarn add react-router-dom
```

To try it, delete all the code in `src/App.js` and replace it with any of the examples on its website.

Proxying API Requests in Development

>Note: this feature is available with `react-scripts@0.2.3` and higher.

People often serve the front-end React app from the same host and port as their backend implementation.
For example, a production setup might look like this after the app is deployed:

```
/            - static server returns index.html with React app
/todos       - static server returns index.html with React app
/api/todos   - server handles any /api/* requests using the backend implementation
```

Such setup is **not** required. However, if you **do** have a setup like this, it is convenient to write requests like `fetch('/api/todos')` without worrying about redirecting them to another host or port during development.

To tell the development server to proxy any unknown requests to your API server in development, add a `proxy` field to your `package.json`, for example:

```js
  "proxy": "http://localhost:4000",
```

This way, when you `fetch('/api/todos')` in development, the development server will recognize that it's not a static asset, and will proxy your request to `http://localhost:4000/api/todos` as a fallback. The development server will **only** attempt to send requests without `text/html` in its `Accept` header to the proxy.

Conveniently, this avoids [CORS issues](http://stackoverflow.com/questions/21854516/understanding-ajax-cors-and-security-considerations) and error messages like this in development:

```
Fetch API cannot load http://localhost:4000/api/todos. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:3000' is therefore not allowed access. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
```

Keep in mind that `proxy` only has effect in development (with `npm start`), and it is up to you to ensure that URLs like `/api/todos` point to the right thing in production. You don't have to use the `/api` prefix. Any unrecognized request without a `text/html` accept header will be redirected to the specified `proxy`.

The `proxy` option supports HTTP, HTTPS and WebSocket connections.
If the `proxy` option is **not** flexible enough for you, alternatively you can:

* [Configure the proxy yourself](#configuring-the-proxy-manually)
* Enable CORS on your server ([here's how to do it for Express](http://enable-cors.org/server_expressjs.html)).
* Use [environment variables](#adding-custom-environment-variables) to inject the right server host and port into your app.

Invalid Host Header" Errors After Configuring Proxy

When you enable the `proxy` option, you opt into a more strict set of host checks. This is necessary because leaving the backend open to remote hosts makes your computer vulnerable to DNS rebinding attacks. The issue is explained in [this article](https://medium.com/webpack/webpack-dev-server-middleware-security-issues-1489d950874a) and [this issue](https://github.com/webpack/webpack-dev-server/issues/887).

This shouldn't affect you when developing on `localhost`, but if you develop remotely like [described here](https://github.com/facebookincubator/create-react-app/issues/2271), you will see this error in the browser after enabling the `proxy` option:

>Invalid Host header

To work around it, you can specify your public development host in a file called `.env.development` in the root of your project:

```
HOST=mypublicdevhost.com
```

If you restart the development server now and load the app from the specified host, it should work.

If you are still having issues or if you're using a more exotic environment like a cloud editor, you can bypass the host check completely by adding a line to `.env.development.local`. **Note that this is dangerous and exposes your machine to remote code execution from malicious websites:**

```
# NOTE: THIS IS DANGEROUS!
# It exposes your machine to attacks from the websites you visit.
DANGEROUSLY_DISABLE_HOST_CHECK=true
```

We don't recommend this approach.

Configuring the Proxy Manually

>Note: this feature is available with `react-scripts@1.0.0` and higher.

If the `proxy` option is **not** flexible enough for you, you can specify an object in the following form (in `package.json`).

You may also specify any configuration value [`http-proxy-middleware`](https://github.com/chimurai/http-proxy-middleware#options) or [`http-proxy`](https://github.com/nodejitsu/node-http-proxy#options) supports.
```js
{
  // ...
  "proxy": {
    "/api": {
      "target": "<url>",
      "ws": true
      // ...
    }
  }
  // ...
}
```

All requests matching this path will be proxies, no exceptions. This includes requests for `text/html`, which the standard `proxy` option does not proxy.

If you need to specify multiple proxies, you may do so by specifying additional entries. Matches are regular expressions, so that you can use a regexp to match multiple paths.
```js
{
  // ...
  "proxy": {
    // Matches any request starting with /api
    "/api": {
      "target": "<url_1>",
      "ws": true
      // ...
    },
    // Matches any request starting with /foo
    "/foo": {
      "target": "<url_2>",
      "ssl": true,
      "pathRewrite": {
        "^/foo": "/foo/beta"
      }
      // ...
    },
    // Matches /bar/abc.html but not /bar/sub/def.html
    "/bar/[^/]*[.]html": {
      "target": "<url_3>",
      // ...
    },
    // Matches /baz/abc.html and /baz/sub/def.html
    "/baz/.*/.*[.]html": {
      "target": "<url_4>"
      // ...
    }
  }
  // ...
}
```

Debugging Tests in Chrome

Add the following to the `scripts` section in your project's `package.json`
```json
"scripts": {
    "test:debug": "react-scripts --inspect-brk test --runInBand --env=jsdom"
  }
```
Place `debugger;` statements in any test and run:
```bash
$ npm run test:debug
```

This will start running your Jest tests, but pause before executing to allow a debugger to attach to the process.

Open the following in Chrome
```
about:inspect
```

After opening that link, the Chrome Developer Tools will be displayed. Select `inspect` on your process and a breakpoint will be set at the first line of the react script (this is done simply to give you time to open the developer tools and to prevent Jest from executing before you have time to do so). Click the button that looks like a "play" button in the upper right hand side of the screen to continue execution. When Jest executes the test that contains the debugger statement, execution will pause and you can examine the current scope and call stack.

>Note: the --runInBand cli option makes sure Jest runs test in the same process rather than spawning processes for individual tests. Normally Jest parallelizes test runs across processes but it is hard to debug many processes at the same time.

Debugging Tests in Visual Studio Code

Debugging Jest tests is supported out of the box for [Visual Studio Code](https://code.visualstudio.com).

Use the following [`launch.json`](https://code.visualstudio.com/docs/editor/debugging#_launch-configurations) configuration file:
```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Debug CRA Tests",
      "type": "node",
      "request": "launch",
      "runtimeExecutable": "${workspaceRoot}/node_modules/.bin/react-scripts",
      "args": [
        "test",
        "--runInBand",
        "--no-cache",
        "--env=jsdom"
      ],
      "cwd": "${workspaceRoot}",
```

```
      "protocol": "inspector",
      "console": "integratedTerminal",
      "internalConsoleOptions": "neverOpen"
    }
  ]
}
```