CSS

The project uses TailwindCSS for styling. Most styling can be found inline but the small amount that has been abstracted away has been placed in '*./src/index.css*'. But, as per the documentation on reusing styles, it is advised that you keep this to a minimum:

"*…you can use Tailwind's @apply directive to extract repeated utility patterns to custom CSS classes when a template partial feels heavy-handed.*

Avoiding premature abstraction

Whatever you do, ***don't use @apply just to make things look "cleaner"****. Yes, HTML templates littered with Tailwind classes are kind of ugly. Making changes in a project that has tons of custom CSS is worse.*

If you start using @apply for everything, you are basically just writing CSS again and throwing away all of the workflow and maintainability advantages Tailwind gives you, for example:

- ***You have to think up class names all the time*** *— nothing will slow you down or drain your energy like coming up with a class name for something that doesn't deserve to be named.*
- ***You have to jump between multiple files to make changes*** *— which is a way bigger workflow killer than you'd think before co-locating everything together.*

- ***Changing styles is scarier*** *— CSS is global, are you sure you can change the min-width value in that class without breaking something in another part of the site?*
- ***Your CSS bundle will be bigger*** *— oof.*

If you're going to use @apply, use it for very small, highly reusable things like buttons and form controls — and even then only if you're not using a framework like React where a component would be a better choice."

Code Formatting

The project has been initialized with Prettier for code formatting upon save. It also has a plugin installed that will sort Tailwind classes upon save.

```
module.exports = {
    trailingComma: 'es5',
    tabWidth: 4,
    semi: false,
    singleQuote: true,
}

module.exports = {
  plugins: [require('prettier-plugin-tailwindcss')],
}


module.exports = {
  tailwindConfig: './tailwind.config.js',
}
```

"*./prettier.config.js*"

Testing

E2E Testing

End to end (E2E) testing is done using CypressJS (Cypress) and Github Actions (Features • GitHub Actions). Documentation specific to the merging of these two technologies can be found here: GitHub Actions | Cypress Documentation.

In the root folder you will find a '*main.yml*' file in '*./github/workflows*'. Whenever you push this repo to GitHub it will scan this file and execute the tests that it points to. These tests can be found in '*./cypress/e2e/*'. For more information on the technology specifics, see the original documentation above as well as comments found in the test files themselves as seen below:

```
// finds all links in the page that aren't
// mail or external links and clicks them
// and checks if they direct to the correct URL
// and checks that they don't load the 404 page
it("check all links to sites", () => {
  cy.viewport("macbook-15");
  cy.visit("/");
  cy.get('a:not([href*="mailto:"])')
     .not('[target*="_blank"]')
     .each((page) => {
     const href = page.prop("href");
     cy.visit(href);
     cy.url().should("include", href);
     cy.get('[data-cy="404"]')
     .should("not.exist");
     cy.go("back");
     });
});
```

Component Testing

Component testing is very similar but offers some different issues to overcome, the first of which is the fact that since we render a component in isolation none of the styling is rendered. To get around this I have created a plugin (‘*./cypress/plugins/tailwind.js*’) that runs before each test and loads the CSS file containing the TailwindCSS classes:

```
// A plugin to render CSS correctly when doing Component tests

before(() => {
   cy.exec('npx tailwindcss -i ./src/index.css -m').then(({stdout})
      => {
      if (!document.head.querySelector('#tailwind-style')) {
            const link = document.createElement('style')
            link.id = 'tailwind-style'
            link.innerHTML = stdout

            document.head.appendChild(link)
      }
   })
})
```

Grouped Imports

The route imports in ‘*./src/index.js*’ have been grouped together for simplicity. To do this we have a proxy file ‘*./src/routes.js*’ who’s only function is to import and export all routes. To keep with this convention, when a new route is created it will need to be imported into ‘*./src/routes.js*’ and then into ‘*./src/index.js*’ as seen below:

```
export { default as Login } from './routes/Login'
export { default as Register } from './routes/Register'
export { default as Dashboard } from './routes/Dashboard'
export { default as EditProfile } from './routes/EditProfile'
export { default as Contribute } from './routes/Contribute'
export { default as UserDirectory } from './routes/UserDirectory'
export { default as DataCommons } from './routes/DataCommons'
export { default as UserProfile } from './routes/UserProfile'
export { default as Recipe } from './routes/Recipe'
export { default as NewRecipeSubmission } from './routes/new-recipe/NewRecipeSubmission'
export { default as NewRecipeTerms } from './routes/new-recipe/NewRecipeTerms'
export { default as Ingredient } from './routes/Ingredient'
export { default as NotFound } from './routes/NotFound'
export { default as NewRecipeName } from './routes/new-recipe/NewRecipeName'
export { default as NewRecipeFork } from './routes/new-recipe/NewRecipeFork'
export { default as Terms } from './routes/Terms'
export { default as PrivacyPolicy } from './routes/Terms'
export { default as NotificationSettings } from './routes/Terms'
export { default as Guide } from './routes/Terms'
```

‘*./src/routes.js*’

```
// Routes
import {
 Login,
 Register,
 Dashboard,
 EditProfile,
 Contribute,
 UserDirectory,
 DataCommons,
 UserProfile,
 Recipe,
 NewRecipeSubmission,
 NewRecipeTerms,
 NotFound,
 NewRecipeName,
 NewRecipeFork,
 Ingredient,
 Terms,
} from "./routes";
```

‘*./src/index.js*’