CSS in React

There are many ways to style a React application, and there are lengthy debates about the best **styling strategy** and **styling approach**. We'll go over a few of these approaches without giving them too much weight. There will be some pro and con arguments, but it's mostly a matter of what fits best for developers and teams.

We will begin React styling with common CSS in React, but then explore two alternatives for more advanced **CSS-in-CSS** (**CSS Modules**) and **CSS-in-JS** (**Styled Components**) strategies. CSS Modules and Styled Components are only two approaches out of many in both groups of strategies. We'll also cover how to include scalable vector graphics (SVGs), such as a logo or icons, in our React application.

If you don't want to build common UI components (e.g. button, dialog, dropdown) from scratch, you can always pick a popular UI library suited for React, which provides these components by default. However, it is better for learning React if you try building these components before using a pre-built solution. As a result, we won't use any of the UI component libraries.

The following styling approaches and SVGs are pre-configured in `create-react-app`. If you're in control of the build tools (e.g. Webpack), they might need to be configured to import CSS or SVG files. Since we are using create-react-app, we can use these files as assets right away.

Cascading Style Sheet (CSS)

Common CSS in React is similar to the standard CSS you may have already learned. Each web application gives HTML elements a `class` (in React it's `className`) attribute that is styled in a CSS file later.

```
const App = () => {
  ...
```

```
  return (

    <div className="container">
      <h1 className="headline-primary">My Hacker Stories</h1>


      <SearchForm
        searchTerm={searchTerm}
        onSearchInput={handleSearchInput}
        onSearchSubmit={handleSearchSubmit}
      />

      {stories.isError && <p>Something went wrong ...</p>}

      {stories.isLoading ? (
        <p>Loading ...</p>
      ) : (
        <List list={stories.data} onRemoveItem={handleRemoveStory} />
      )}
    </div>
  );
};
```

src/App.js

The `<hr />` was removed because the CSS handles the border in the next steps. We'll import the CSS file, which is done with the help of the create-react-app configuration:

```
import React from 'react';
import axios from 'axios';

import './App.css';
```

This CSS file will define the two (and more) CSS classes we used in the App component. In your *src/App.css* file, define them like the following:

```css
.container {
  height: 100vw;
  padding: 20px;

  background: #83a4d4; /* fallback for old browsers */
  background: linear-gradient(to left, #b6fbff, #83a4d4);

  color: #171212;
}

.headline-primary {
  font-size: 48px;
  font-weight: 300;
```

```
  letter-spacing: 2px;
}
```

src/App.css

You should see the first stylings taking effect in your application when you start it again. Next, we will head over to the Item component. Some of its elements receive the `className` attribute too, however, we are also using a new styling technique here:

```
const Item = ({ item, onRemoveItem }) => (

  <div className="item">
    <span style={{ width: '40%' }}>

      <a href={item.url}>{item.title}</a>
    </span>

    <span style={{ width: '30%' }}>{item.author}</span>
    <span style={{ width: '10%' }}>{item.num_comments}</span>
    <span style={{ width: '10%' }}>{item.points}</span>
    <span style={{ width: '10%' }}>

      <button
        type="button"
        onClick={() => onRemoveItem(item)}

        className="button button_small"

      >
        Dismiss
      </button>
    </span>
  </div>
);
```

src/App.js

As you can see, we can also use the native `style` attribute for HTML elements. In JSX, style can be passed as an inline JavaScript object to these attributes. This way we can define dynamic style properties in JavaScript files rather than mostly static CSS files. This approach is called **inline style**, which is useful for quick prototyping and dynamic style definitions. Inline style should be used sparingly, however, as a separate style definition keeps the JSX more concise.

In your *src/App.css* file, define the new CSS classes. Basic CSS features are used. Advanced CSS features (e.g. nesting) from CSS extensions (e.g. Sass) are not included in this example, as they are optional configurations.

```css
.item {
  display: flex;
  align-items: center;
  padding-bottom: 5px;
}

.item > span {
  padding: 0 5px;
  white-space: nowrap;
  overflow: hidden;
  white-space: nowrap;
  text-overflow: ellipsis;
}

.item > span > a {
  color: inherit;
}
```

src/App.css

The button style from the previous component is still missing, so we'll define a base button style and two more specific specific button styles (small and large). One of the button specifications has been used, the other will be used in the next steps.

```css
.button {
  background: transparent;
  border: 1px solid #171212;
  padding: 5px;
  cursor: pointer;

  transition: all 0.1s ease-in;
}
```

```
.button:hover {
  background: #171212;
  color: #ffffff;
}

.button_small {
  padding: 5px;
}

.button_large {
  padding: 10px;
}
```

src/App.css

Apart from styling approaches in React, naming conventions (CSS guidelines) are a whole other topic. The last CSS snippet followed BEM rules by defining modifications of a class with an underscore (`_`). Choose whatever naming convention suits you and your team. Without further ado, we will style the next React component:

```
const SearchForm = ({ ... }) => (

  <form onSubmit={onSearchSubmit} className="search-form">

    <InputWithLabel ... >
      <strong>Search:</strong>
    </InputWithLabel>

    <button
      type="submit"
      disabled={!searchTerm}

      className="button button_large"
    >
      Submit
    </button>
  </form>
);
```

src/App.js

We can also pass the `className` attribute as a prop to React components. For instance, we can use this option to pass the SearchForm component a flexible style with a `className` prop from a range of predefined classes from a CSS file.

Lastly, style the InputWithLabel component:

```
const InputWithLabel = ({ ... }) => {
  ...

  return (
    <>

      <label htmlFor={id} className="label">

        {children}
      </label>
       
      <input
        ref={inputRef}
        id={id}
        type={type}
        value={value}
        onChange={onInputChange}

        className="input"

      />
    </>
  );
};
```

src/App.js

In your *src/App.css* file, add the remaining classes:

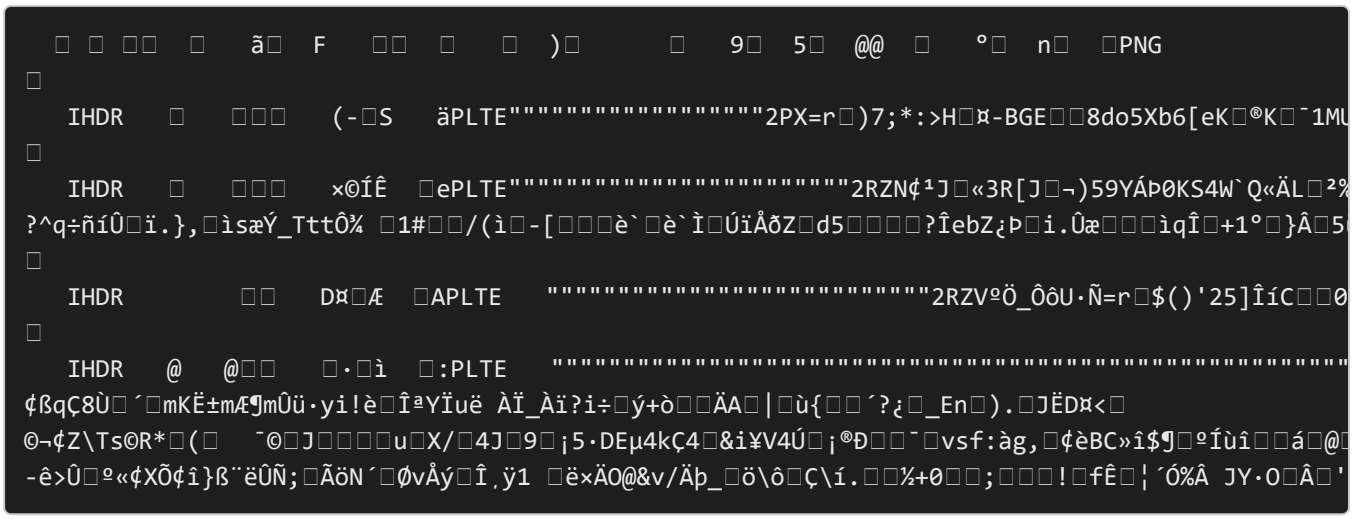
```css
.search-form {
  padding: 10px 0 20px 0;
  display: flex;
  align-items: baseline;
}

.label {
  border-top: 1px solid #171212;
  border-left: 1px solid #171212;
  padding-left: 5px;
  font-size: 24px;
}

.input {
  border: none;
  border-bottom: 1px solid #171212;
  background-color: transparent;

  font-size: 24px;
}
```

src/App.css

For simplicity, we styled elements like label and input individually in the *src/App.css* file.

> However, in a real application it may be better to define these elements once in the *src/index.css* file globally. As React components are split into multiple files, sharing style becomes a necessity.

This is the basic CSS most of us have already learned, written with an inline style that is more dynamic. Without CSS extensions like Sass (Syntactically Awesome Style Sheets) inline styles can become burdensome, though, because features like CSS nesting are not available in native CSS.



Exercises:

- Confirm the changes from the last section.
- Read more about CSS stylesheets in create-react-app.
- Read more about Sass in create-react-app for taking advantage of more advanced CSS features like nesting.
- Try to pass `className` prop from App to SearchForm component, either with the value `button_small` or `button_large` and use this as `className` for the button element.