1. File Structure for React

a. Overview

- Link: https://chatgpt.com/share/67c09e39-e5cc-800d-ab99-9ee64a7711e6
- This document provides an overview of the directory structure for a React-based project.

```
/src
│── /assets          # Static assets (images, fonts, etc.)
│── /components      # Reusable UI components
│── /pages           # Page-level components
│── /layouts         # Layout components (e.g., Navbar, Sidebar, Footer)
│── /hooks           # Custom React hooks
│── /context         # React Context for state management
│── /utils           # Utility/helper functions
│── /services        # API calls and external services
│── /routes          # Route definitions
│── /styles          # Global and module styles (CSS, SCSS, Tailwind, etc.)
│── /types           # Interface
│── /config          # Configuration files (e.g., constants, environment settings)
│── App.js           # Main App component
│── package.json     # Project dependencies and scripts
│── .env             # Environment variables
```

b. Assests

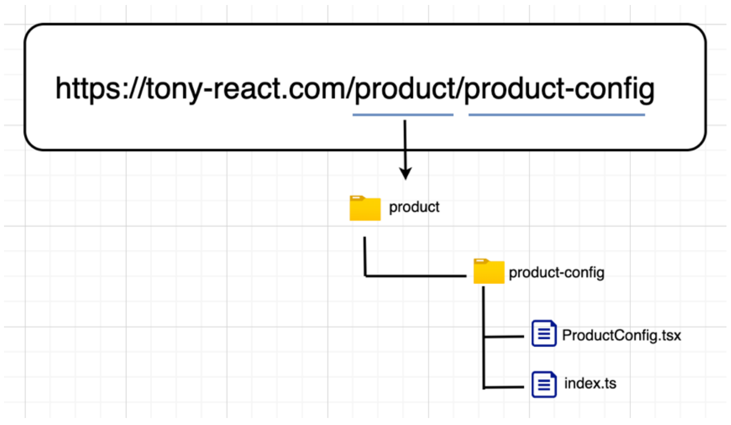
- Contains static assets such as images and fonts. These are files that do not change during the execution of the application.
- For example: background.img, timesnewroman.font, …

c. Components

- Components are one of the core **building blocks of React**. They have the same purpose as JavaScript functions and return HTML. Components make building a UI much easier. A UI is broken down into multiple individual pieces called components. You can work on components independently and then merge them into a parent component, your final UI.
- This place will contain the **components that appear on the page**. For example, in our project, it contains the table, the card, the button, the form, etc.

d. Pages

- The final folder that the pages folder indicates the route of the React application. Each file in this folder contains its route. A page can contain its sub folder. And every sub folder represents its route. This folder is confusing for you when you identify which components are in the pages folder, or the feature folder. So, I propose we reference ways the NextJS framework uses app router to separate components.

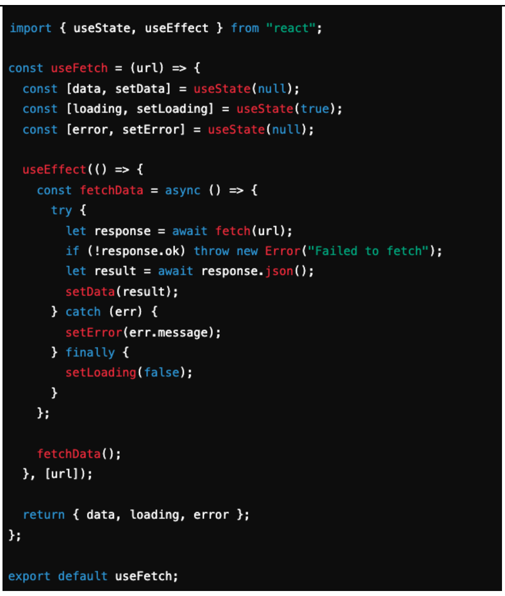


e. Layouts

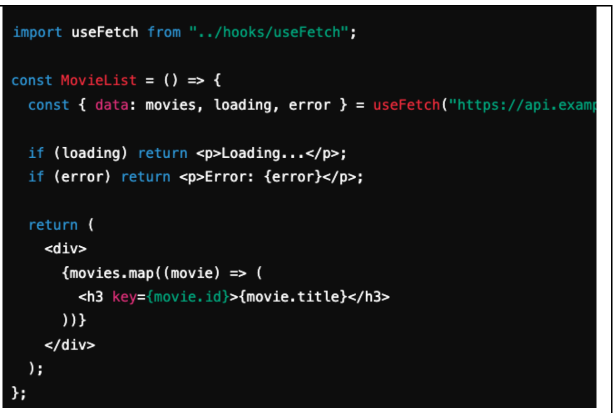
- The layouts folder contains dynamic layouts that you want to display based on your client's information. If your application only has a layout, then you can just place it in the components folder, but if you have multiple different layouts used across your application, this is a great place to store them.
- Example: navigation bar, the side bar

f. Hooks

- Hooks provide access to states for functional components while creating a React application. It allows you to use state and other React features without writing a class. Placing them in a dedicated directory allows for easy access and reuse across components throughout your application.
- For example: write the API call, …



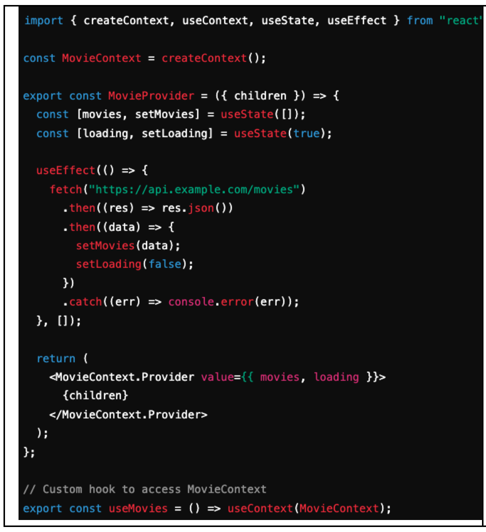
```
import { useState, useEffect } from "react";

const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        let response = await fetch(url);
        if (!response.ok) throw new Error("Failed to fetch");
        let result = await response.json();
        setData(result);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
};

export default useFetch;
```



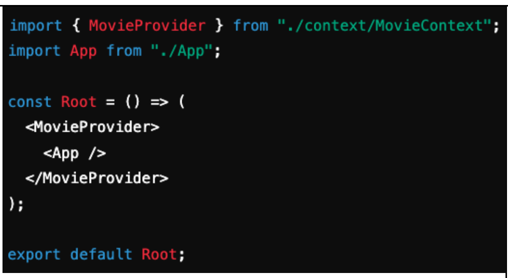
```
import useFetch from "../hooks/useFetch";

const MovieList = () => {
  const { data: movies, loading, error } = useFetch("https://api.examp

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <div>
      {movies.map((movie) => (
        <h3 key={movie.id}>{movie.title}</h3>
      ))}
    </div>
  );
};
```

g. Contexts

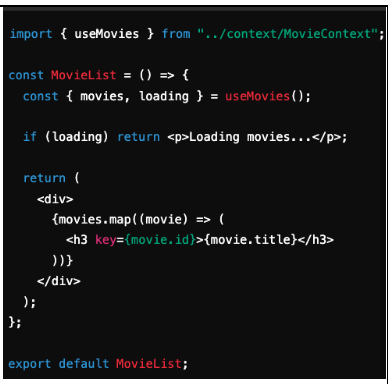
- Contexts are used to share state across multiple components without having to pass props manually through each level of the component tree.
- For example: authentication sharing, if you're building a Netflix-like platform, you might need global movie state..



```
import { createContext, useContext, useState, useEffect } from "react'

const MovieContext = createContext();

export const MovieProvider = ({ children }) => {
  const [movies, setMovies] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch("https://api.example.com/movies")
      .then((res) => res.json())
      .then((data) => {
        setMovies(data);
        setLoading(false);
      })
      .catch((err) => console.error(err));
  }, []);

  return (
    <MovieContext.Provider value={{ movies, loading }}>
      {children}
    </MovieContext.Provider>
  );
};

// Custom hook to access MovieContext
export const useMovies = () => useContext(MovieContext);
```



```
import { MovieProvider } from "./context/MovieContext";
import App from "./App";

const Root = () => (
  <MovieProvider>
    <App />
  </MovieProvider>
);

export default Root;
```



```
import { useMovies } from "../context/MovieContext";

const MovieList = () => {
  const { movies, loading } = useMovies();

  if (loading) return <p>Loading movies...</p>;

  return (
    <div>
      {movies.map((movie) => (
        <h3 key={movie.id}>{movie.title}</h3>
      ))}
    </div>
  );
};

export default MovieList;
```

h. Utils (Utilization)

- Utility functions, such as date formatting or string manipulation, are stored in this directory. These functions are typically used across multiple components or modules and serve to centralize commonly used logic.

- Example: Format Date, convert number, debouncing.

i. Services

- In this directory, you'll find files responsible for handling API calls and other services. Services keep the implementation details of interacting with external resources, provide separation of concerns and code maintainability.
- Example: API call
- Concern: Hooks vs Services

Feature	/services (API Call)	/hooks (API + State management)
Purpose	Hanldes API calls only	Handles API + state logic
Returns	Function of that fetchs data	{data, loading, error
State Manegement	NO	Loading and error case
Usage	Called manually inside components	Automatically handles fetching
Best	Reusable API functions: API across application: user login or sign out	Clean components: API for user history, User Information

j. Routes (Routing page)

- The routes/ folder is used to define and organize the application's routes when using React Router. Instead of defining all routes inside App.js, keeping them in a separate routes/ folder makes the project modular, scalable, and easier to manage.

k. Styles

- This directory contains CSS or other styling files used to define the visual appearance of your application. In this folder, styles of different components are stored.

l. Types

- You can use this folder to contain interfaces, types that you use to define on components. This folder is easy to use to share interfaces across components.
- Example: interface for the API

m. Configs (chắc không xài)

- This folder contains a configuration file where we store environment variables in config.js. We will use this file to set up multi-environment configurations in your application. Ex- Environment Configuration, WebPack Configuration, Babel Configuration, etc.

2. File Structure for FastAPI (python)

a. Overview

- Link: https://chatgpt.com/share/67c13a81-ba08-800d-8469-718c8ca8f814
- The structure of the project would be similar to the structure below.

b. App

i. API

1. Routes

- This folder will handle the API routing. Each file name is associated with its model and schema.
- For example, the user.py will handle all the API related functionality for their models.
- The file in this folder will have the name with the format: <name of model>_route.py

2. dependencies.py

3. __init__.py

ii. Cores

- This folder will handle all the core features like the security, the database connection set up, and the utilisation, such as the format date, converting numbers, …

1. __init__.py

iii. Models

- This will handle the table of the SQL, we use in the project and define all the attributes in the table.
- For example. user.py: In this file, we will have the username: (Column: String(30))

1. __init__.py

iv. Schemas

- In this folder, we handle all cases of the fetching API and sending the API to the front end.

```python
from pydantic import BaseModel, EmailStr

class UserBase(BaseModel):
    username: str
    email: EmailStr

class UserCreate(UserBase):
    password: str

class UserResponse(UserBase):
    id: int

    class Config:
        orm_mode = True
```

1. __init__.py

v. Services

- In this folder, we will handle all the business logic for the API to send back to the databases or the front end application.
- The file in this folder will be <name of the models>_service.py

1. __init__.py

vi. Main.py

- Run the project

vii. __init__.py

c. .env

- Define the environment for the project

d. Requirements.txt

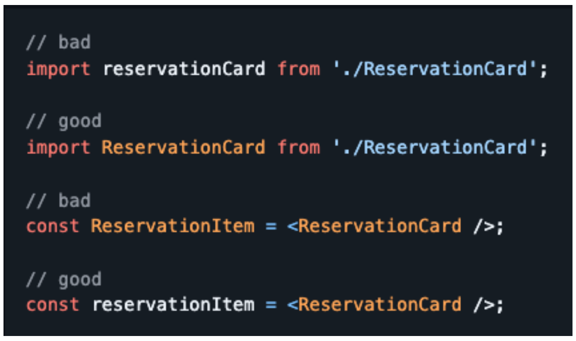
- The requirements for all the libraries used in the project are at or higher defined version.

3. Coding Convention React

- **Link: https://github.com/UETCodeCamp/jsx-style-guide**

a. Naming

- File name: Use Pacal Case
  - Ex: Footer.tsx, NavBar.tsx
- Reference name: Use Pascal Case for React components and camelCase for their instances



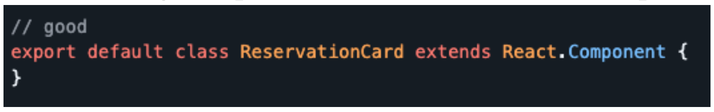
```jsx
// bad
import reservationCard from './ReservationCard';

// good
import ReservationCard from './ReservationCard';

// bad
const ReservationItem = <ReservationCard />;

// good
const reservationItem = <ReservationCard />;
```

- All the coefficients in the code must be changed to the string with the CONSTANT variable. (remove magic string).

- Variable:
    - The CONSTANT variable follows the rule. EX: CONSTANT_VARIABLE = 123.
        - All the characters must be capitalised
        - Using the underscore "_" to separate the word.
    - The boolean variable and function must have the "is<name>". Ex: isHidden, isSorted().
    - The getter and setter must have the "set" and "get" in the name.

b. Declaration

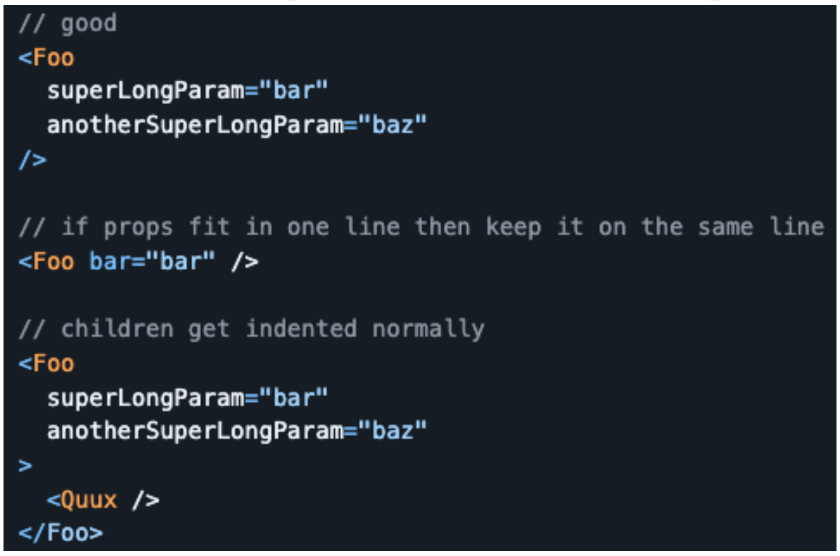
- Do not use displayName for naming components. Instead, name the component by reference.



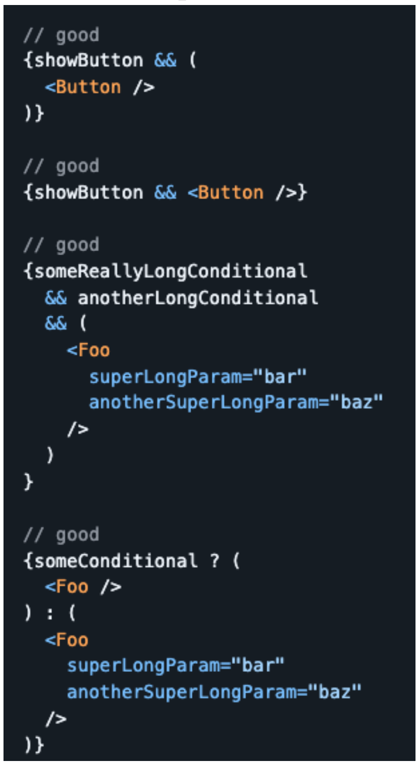
```
// good
export default class ReservationCard extends React.Component {
}
```

c. Alignment

- If the props of the component fit in one line, we can keep them in one line. When the number of the props is large, we need to follow the convention.
- If we use the condition with the components, we follow the rule
    - If we can keep it to one line, keep it
    - If we need to separate to a new line, and the components need to be in the parentheses ()



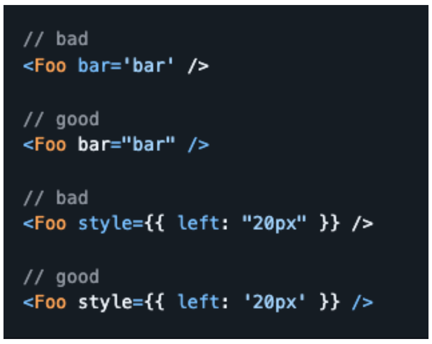
```
// good
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
/>

// if props fit in one line then keep it on the same line
<Foo bar="bar" />

// children get indented normally
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
>
  <Quux />
</Foo>
```



```
// good
{showButton && (
  <Button />
)}

// good
{showButton && <Button />}

// good
{someReallyLongConditional
  && anotherLongConditional
  && (
    <Foo
      superLongParam="bar"
      anotherSuperLongParam="baz"
    />
  )
}

// good
{someConditional ? (
  <Foo />
) : (
  <Foo
    superLongParam="bar"
    anotherSuperLongParam="baz"
  />
)}
```

d. Quotes

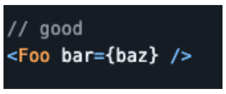
- The props related to the HTML we are using the '', other will be "".



```
// bad
<Foo bar='bar' />

// good
<Foo bar="bar" />

// bad
<Foo style={{ left: "20px" }} />

// good
<Foo style={{ left: '20px' }} />
```

e. Spacing

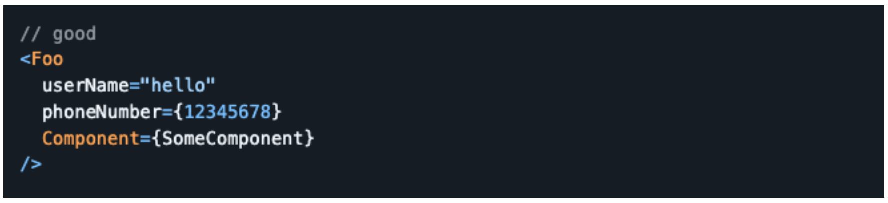
- We write the components like this:
  - If we have to initialise the value, do not pad JSX curly braces with spaces



```
// good
<Foo bar={baz} />
```

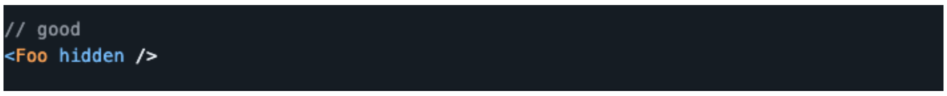
f. Props

- Follow the rule:
  - Always use camelCase for prop names, or PascalCase if the prop value is a React component.



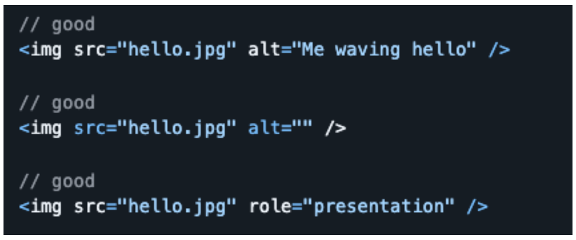
```
// good
<Foo
  userName="hello"
  phoneNumber={12345678}
  Component={SomeComponent}
/>
```

  - Omit the value of the prop when it is explicitly true.



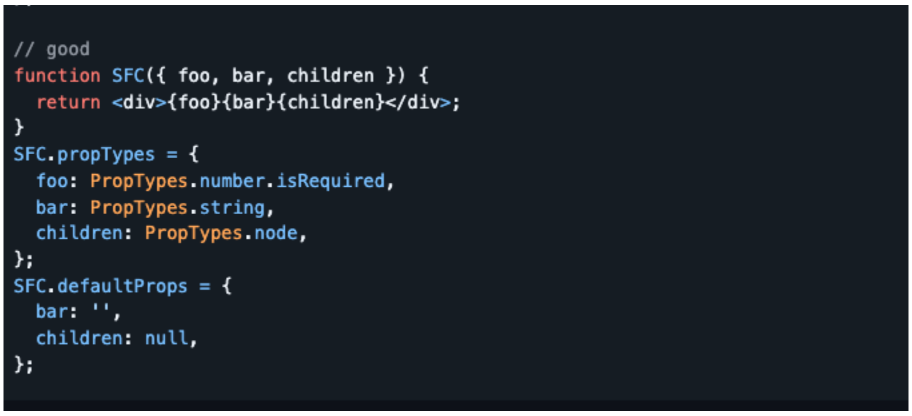
```
// good
<Foo hidden />
```

  - Always include an alt prop on <img> tags. If the image is presentational, alt can be an empty string or the <img> must have role="presentation".



```
// good
<img src="hello.jpg" alt="Me waving hello" />

// good
<img src="hello.jpg" alt="" />

// good
<img src="hello.jpg" role="presentation" />
```

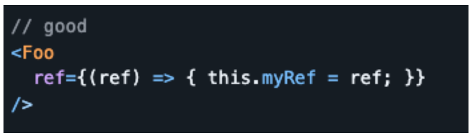
  - Always define explicit defaultProps for all non-required props.



```
// good
function SFC({ foo, bar, children }) {
  return <div>{foo}{bar}{children}</div>;
}
SFC.propTypes = {
  foo: PropTypes.number.isRequired,
  bar: PropTypes.string,
  children: PropTypes.node,
};
SFC.defaultProps = {
  bar: '',
  children: null,
};
```

g. Refs

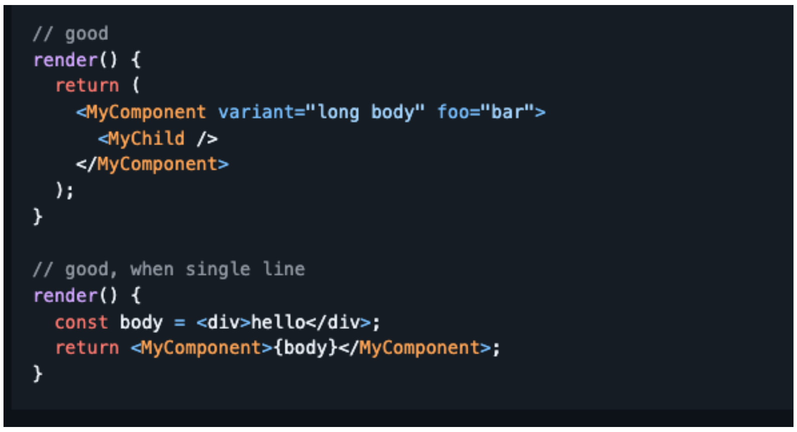
- Always use ref callbacks.



```
// good
<Foo
  ref={(ref) => { this.myRef = ref; }}
/>
```

h. Perentheses

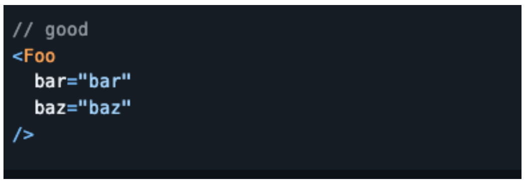
- Wrap JSX tags in parentheses when they span more than one line.



```
// good
render() {
  return (
    <MyComponent variant="long body" foo="bar">
      <MyChild />
    </MyComponent>
  );
}

// good, when single line
render() {
  const body = <div>hello</div>;
  return <MyComponent>{body}</MyComponent>;
}
```

i. Tags

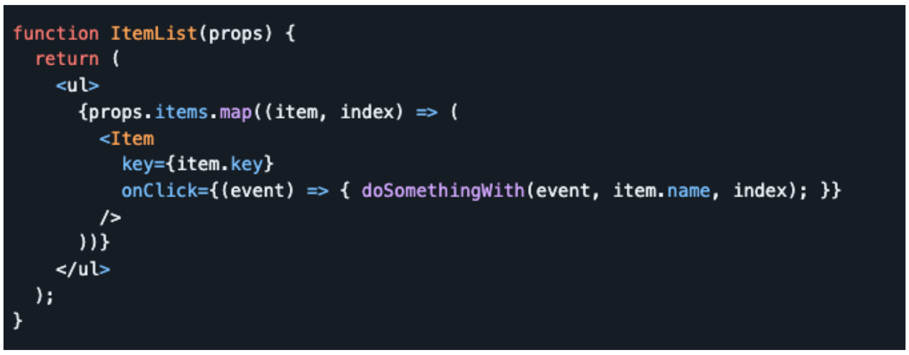
- Always self-close tags that have no children
- If your component has multiline properties, close its tag on a new line



```
// good
<Foo
  bar="bar"
  baz="baz"
/>
```

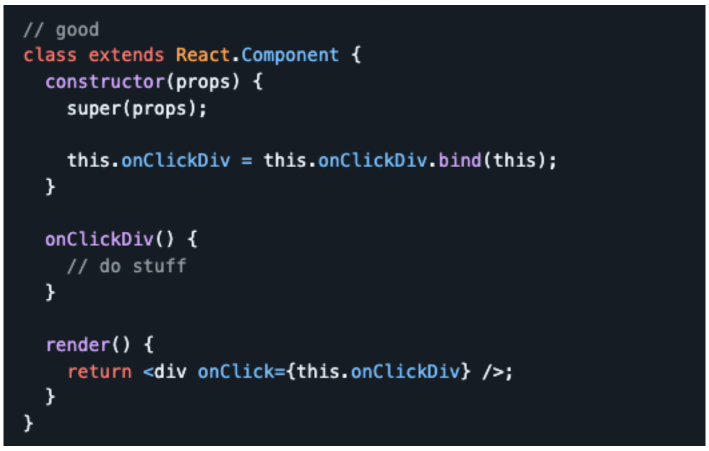
j. Methods

- Use arrow functions to close over local variables. It is handy when you need to pass additional data to an event handler. Although, make sure they do not massively hurt performance, in particular when passed to custom components that might be PureComponents, because they will trigger a possibly needless rerender every time.



```
function ItemList(props) {
  return (
    <ul>
      {props.items.map((item, index) => (
        <Item
          key={item.key}
          onClick={(event) => { doSomethingWith(event, item.name, index); }}
        />
      ))}
    </ul>
  );
}
```

- Bind event handlers for the render method in the constructor. Why? A bind call in the render path creates a brand new function on every single render. Do not use arrow functions in class fields, because it makes them challenging to test and debug, and can negatively impact performance, and because conceptually, class fields are for data, not logic.



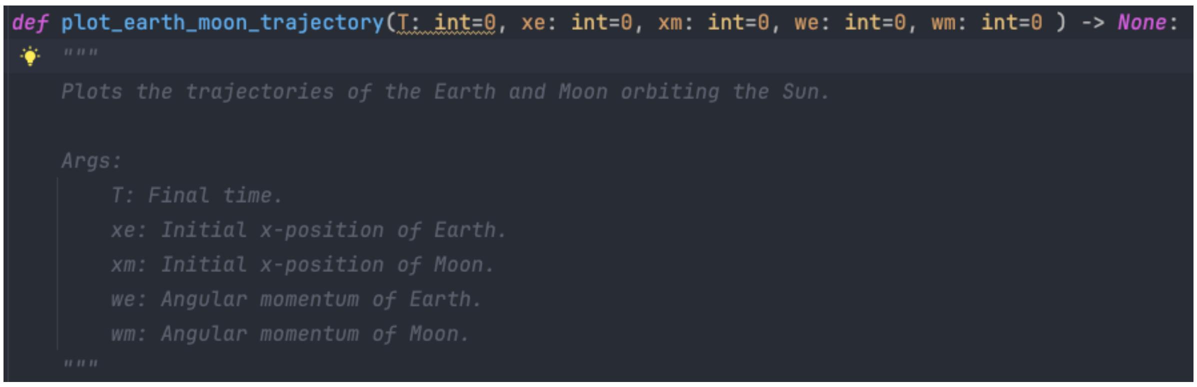
```
// good
class extends React.Component {
  constructor(props) {
    super(props);

    this.onClickDiv = this.onClickDiv.bind(this);
  }

  onClickDiv() {
    // do stuff
  }

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

- Be sure to return a value in your render methods

4. Coding Convention Python

- Follow the PEP8 (install the extension PEP8 to check)

- All functions must have custom parameters with a default value. All parameters must explicitly specify the type of value.
- All functions must have a description like this:
    - Function:



```python
def plot_earth_moon_trajectory(T: int=0, xe: int=0, xm: int=0, we: int=0, wm: int=0 ) -> None:
    """
    Plots the trajectories of the Earth and Moon orbiting the Sun.

    Args:
        T: Final time.
        xe: Initial x-position of Earth.
        xm: Initial x-position of Moon.
        we: Angular momentum of Earth.
        wm: Angular momentum of Moon.
    """
```

- The variable, function, file name are using the snake_case excepted CONSTANT and boolean variable:
    - long_variable, def long_function(self, …), long_file.py
    - CONSTANT_VARIABLE =…
    - Boolean variable: isTrue