sourcemind

SOFTWARE ARCHITECTURE

What is Software Architecture

- Software architecture refers to the high-level design of a software system, which includes its components, their relationships, and the principles and guidelines governing their organization and interaction. It provides a blueprint for the construction of a software system, ensuring that it meets the desired functionality, performance, and quality attributes.

Architecture vs Design Patters

- Architecture refers to the overall structure and organization of a system, while a design pattern is a specific reusable solution to a commonly occurring problem in software design. In other words, architecture is the high-level plan for a system, while design patterns are more detailed, tactical solutions for specific issues that arise within that system. Architecture is concerned with the big picture, while design patterns focus on the implementation details.

sourcemind

vs

Bad Architecture Examples

Step by Step

1. Gather and understand the requirements: The first step in software architecture is to gather and understand the requirements of the system that needs to be designed. This includes understanding the stakeholders, their needs, ar the context in which the system will operate.
2. Create the initial architecture: Once the requirements are understood, the architect creates an initial architecture for the system. This includes defining the main components, their interactions, and the overall structure of the system.
3. Evaluate the architecture: The architecture is evaluated to ensure that it meets the requirements and that it is feasible implement. This evaluation may include reviewing the architecture with stakeholders, prototyping, or conducting simu
4. Refine the architecture: Based on the evaluation, the architecture is refined to improve its quality attributes, such as performance, scalability, and maintainability. This may involve making trade-offs between competing attribut such as performance and cost.
5. Document the architecture: The architecture is documented to communicate it to stakeholders and developers. This documentation includes diagrams, descriptions, and other artifacts that describe the architecture and its compon
6. Implement the architecture: Once the architecture is documented, the developers can begin implementing the system The architecture provides guidance for the development team on how to structure the code and components.
7. Test and validate the system: The system is tested and validated to ensure that it meets the requirements and that it i reliable and performs as expected.
8. Maintain and evolve the system: After the system is deployed, it will need to be maintained and evolved over time. The architecture provides a foundation for future changes and improvements to the system.

sourcemind

To design a React application architecture, consider the following:

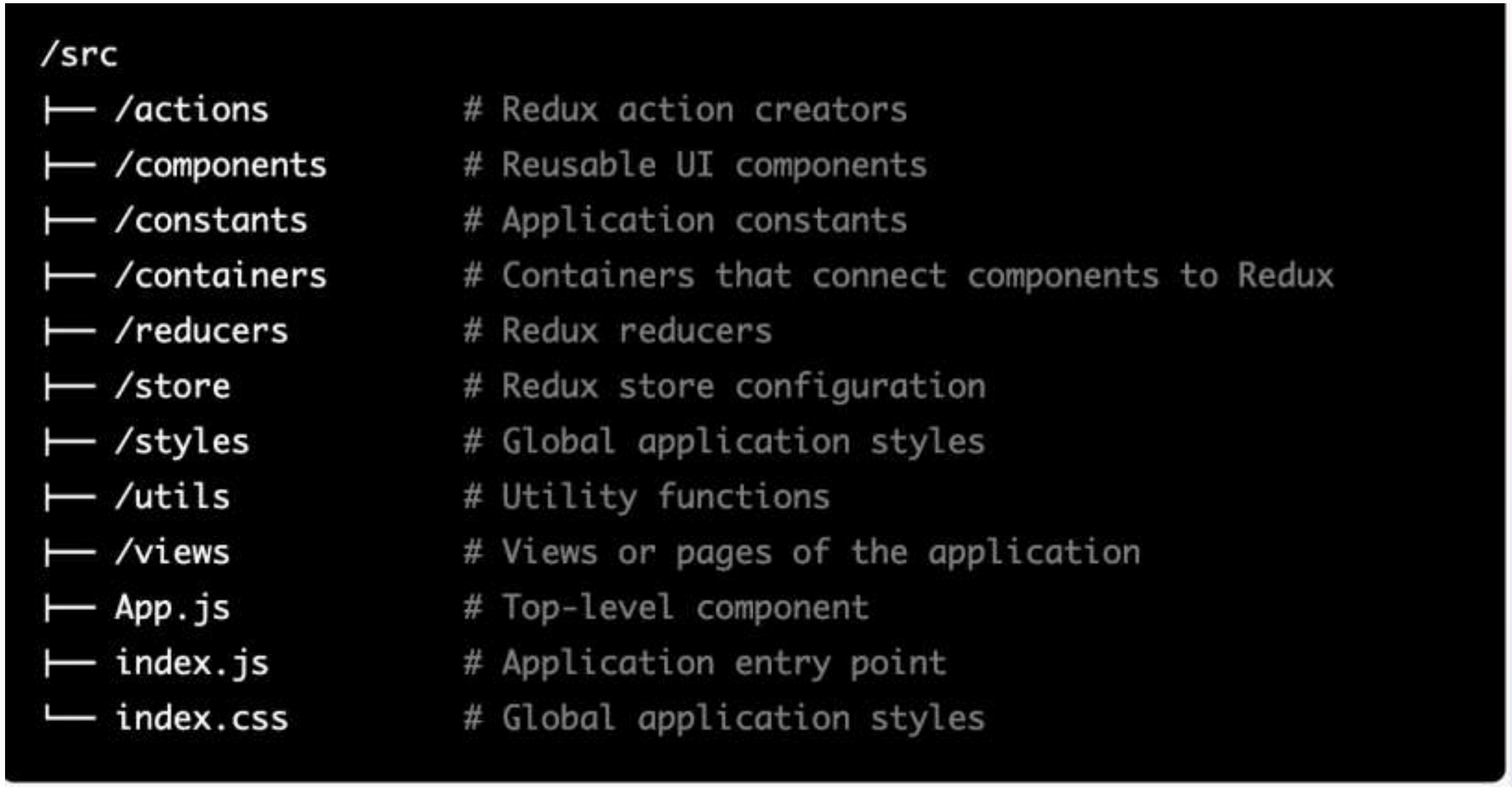
1. Decide on the state management approach, such as Redux or the React Context API.
2. Choose a routing library, such as React Router.
3. Divide the application into smaller components, grouping similar components together and separating them into reusable modules.
4. Use container and presentational components to separate logic and UI.
5. Utilize higher-order components and render props for code reuse.
6. Make sure to follow best practices for styling with CSS-in-JS libraries like Styled Components or CSS Modules.
7. Consider performance optimizations, such as code splitting and lazy loading.
8. Set up a testing framework like Jest and Enzyme to ensure proper testing coverage.
9. Set up a continuous integration/continuous deployment pipeline to streamline the development and deployment process.
10. Consider using libraries and frameworks that complement React, such as TypeScript, Next.js, or GraphQL.

sourcemind

Here are some examples of React design architectures:

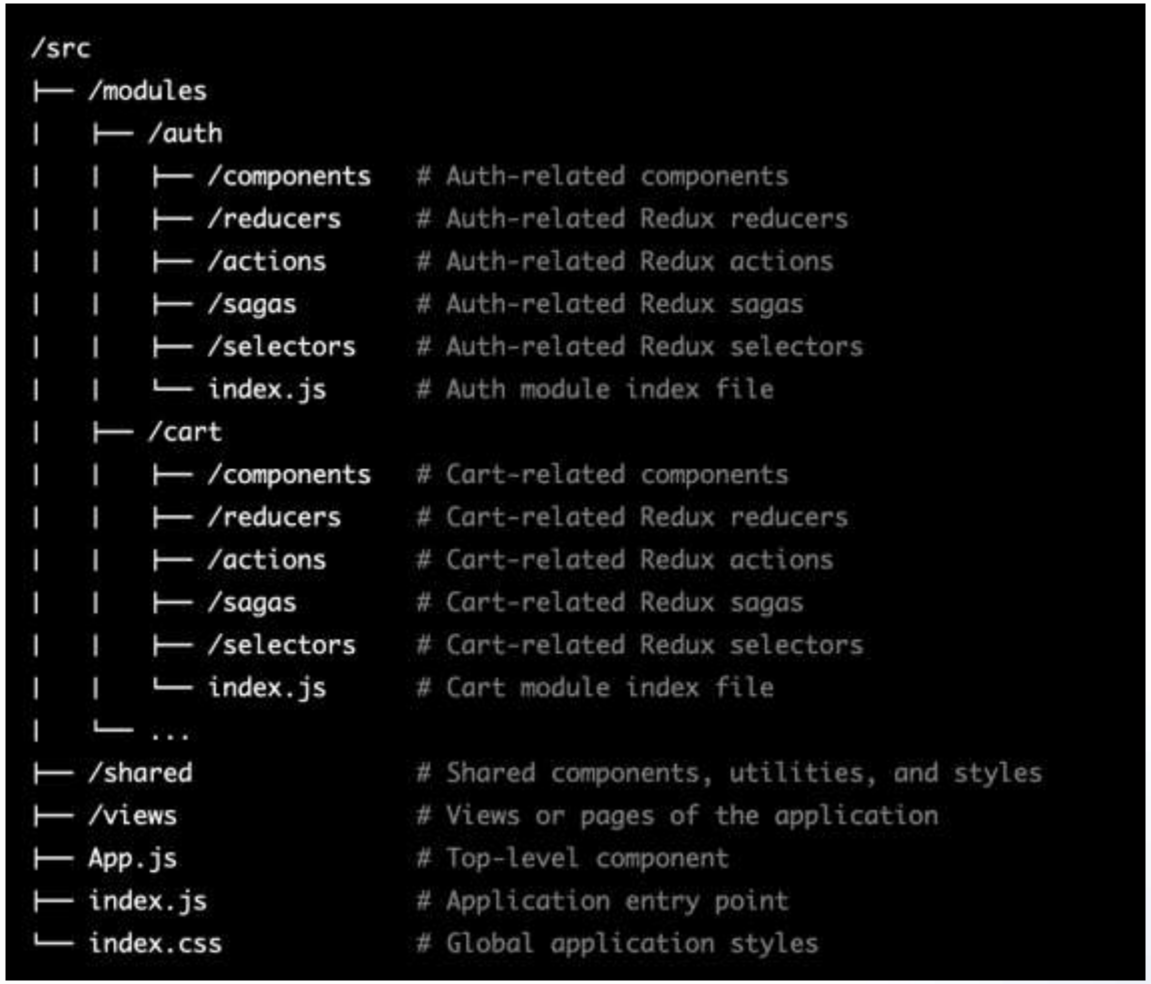
1. Flux - a popular and simple architecture that consists of unidirectional data flow, and can be used with React's state management library, Redux.
2. Redux - a predictable state container, that manages application state in a central store and can be used with React to simplify data flow and reduce boilerplate code.
3. MVC - Model-View-Controller architecture can be used with React by separating the application into the model (data) view (React components), and controller (application logic).
4. Atomic design - a methodology for designing components, where components are divided into atoms (individual UI e molecules (groups of atoms), organisms (groups of molecules), templates (arrangement of organisms), and pages.
5. Domain-driven design - a methodology for designing architecture around a specific business domain, focusing on business logic, context, and behavior.
6. Micro Frontends - an architecture that involves breaking down a large application into smaller, independently deployable frontends, which can be developed by separate teams with separate technology stacks.
7. Serverless Architecture - an architecture where the backend is made up of serverless functions that can be scaled up and down as needed, and can be used with React to create fast and scalable applications.

Folder Structure



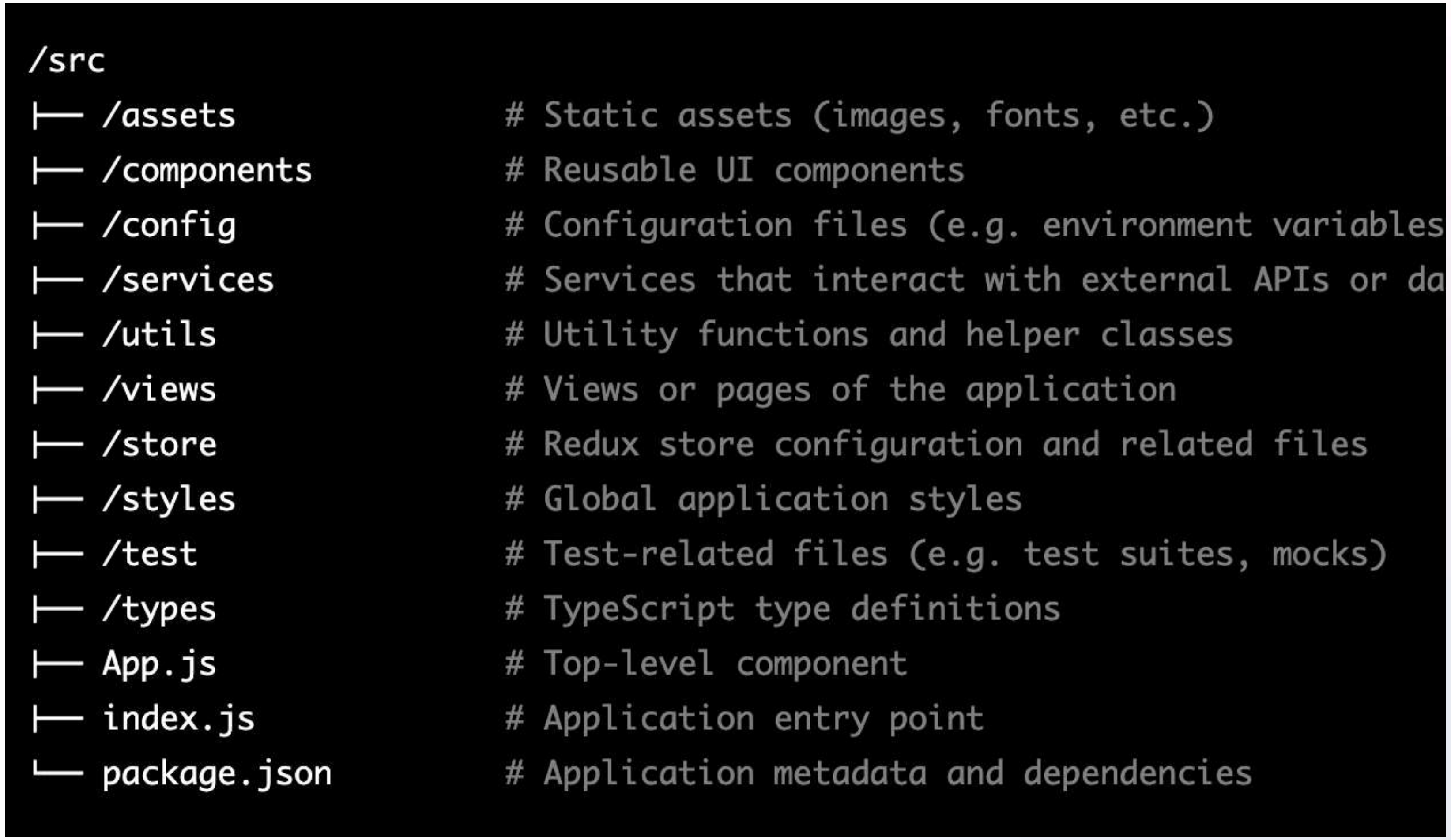
```
/src
├── /actions          # Redux action creators
├── /components       # Reusable UI components
├── /constants        # Application constants
├── /containers       # Containers that connect components to Redux
├── /reducers         # Redux reducers
├── /store            # Redux store configuration
├── /styles           # Global application styles
├── /utils            # Utility functions
├── /views            # Views or pages of the application
├── App.js            # Top-level component
├── index.js          # Application entry point
└── index.css         # Global application styles
```

Module based Structure



```
/src
├── /modules
|   ├── /auth
|   |   ├── /components    # Auth-related components
|   |   ├── /reducers      # Auth-related Redux reducers
|   |   ├── /actions       # Auth-related Redux actions
|   |   ├── /sagas         # Auth-related Redux sagas
|   |   ├── /selectors     # Auth-related Redux selectors
|   |   └── index.js       # Auth module index file
|   ├── /cart
|   |   ├── /components    # Cart-related components
|   |   ├── /reducers      # Cart-related Redux reducers
|   |   ├── /actions       # Cart-related Redux actions
|   |   ├── /sagas         # Cart-related Redux sagas
|   |   ├── /selectors     # Cart-related Redux selectors
|   |   └── index.js       # Cart module index file
|   └── ...
├── /shared                # Shared components, utilities, and styles
├── /views                 # Views or pages of the application
├── App.js                 # Top-level component
├── index.js               # Application entry point
└── index.css              # Global application styles
```

Another example



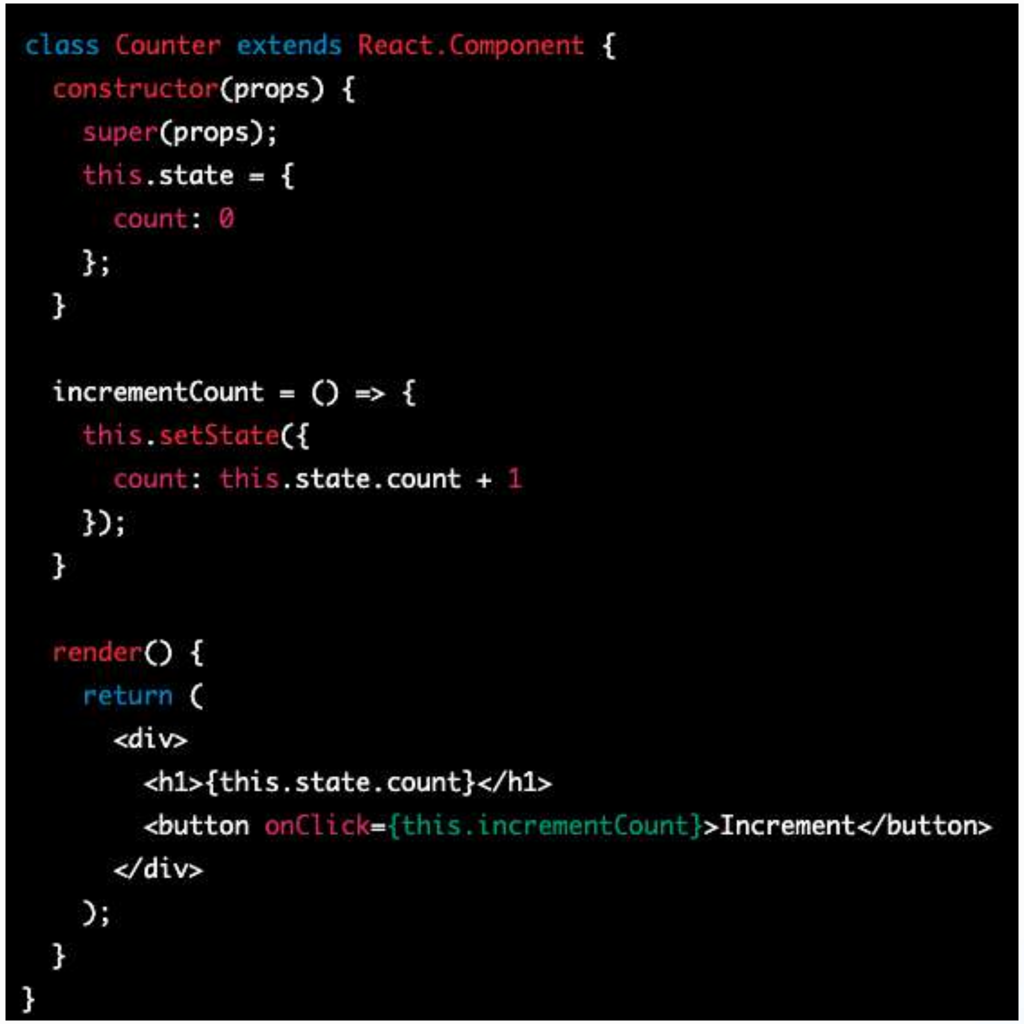
```
/src
├── /assets             # Static assets (images, fonts, etc.)
├── /components         # Reusable UI components
├── /config             # Configuration files (e.g. environment variables
├── /services           # Services that interact with external APIs or da
├── /utils              # Utility functions and helper classes
├── /views              # Views or pages of the application
├── /store              # Redux store configuration and related files
├── /styles             # Global application styles
├── /test               # Test-related files (e.g. test suites, mocks)
├── /types              # TypeScript type definitions
├── App.js              # Top-level component
├── index.js            # Application entry point
└── package.json        # Application metadata and dependencies
```

Bad architectural practices

1. Monolithic structure: This is where the entire application is contained within a single file or component. This can make it difficult to maintain and scale the application, as changes to one part of the code can have unintended consequences elsewhere. Instead, it's generally a good idea to use a modular, feature-based approach to organizing your code.
2. Poorly organized file structure: If your files are disorganized and scattered around the codebase, it can be difficult to find what you need and make changes efficiently. Instead, try to follow a consistent, well-structured file hierarchy that makes it easy to find what you need.

3. Over-reliance on global state: Using global state can be a useful tool for managing state in a React application, but if you rely too heavily on global state it can make it difficult to understand and debug your code. Instead, try to use local state wherever possible and use global state sparingly.
4. Tight coupling between components: If your components are tightly coupled, meaning they rely too heavily on each other's internal state and logic, it can make it difficult to modify or replace components without affecting other parts of the application. Instead, try to design your components to be modular and reusable, with clear boundaries and well-defined APIs.
5. Lack of separation of concerns: If your components are responsible for too many different things, such as rendering UI elements, handling state, and making API calls, it can make it difficult to test and maintain your code. Instead, try to separate your concerns into different layers or components that each have a specific responsibility. For example, you could have separate components for rendering UI elements, handling user input, and making API calls.

State Design Pattern in React

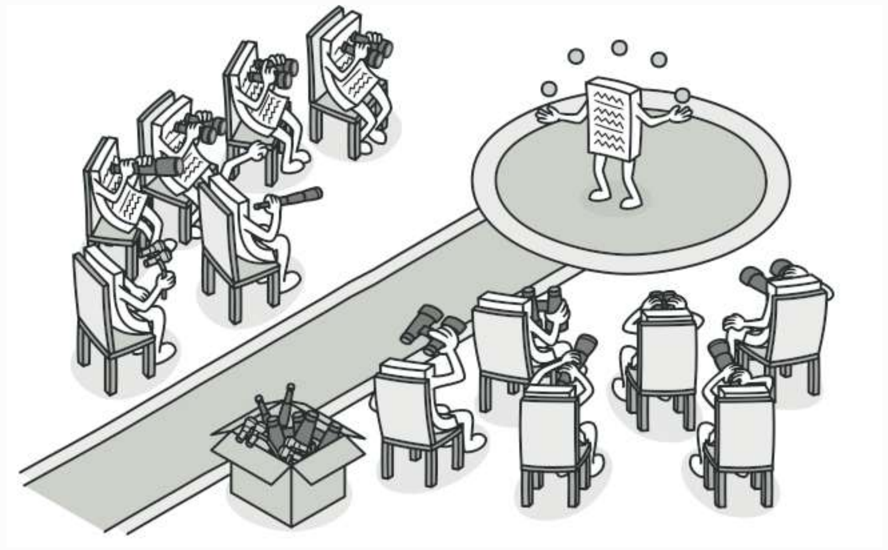


```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  incrementCount = () => {
    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    return (
      <div>
        <h1>{this.state.count}</h1>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

The State design pattern is a fundamental concept in React, where components can manage their own state and render their views based on that state. In React, state is an object that hol data that can change over time and trigger re-rendering of the component.

Behavioral patterns

3. Observer



Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

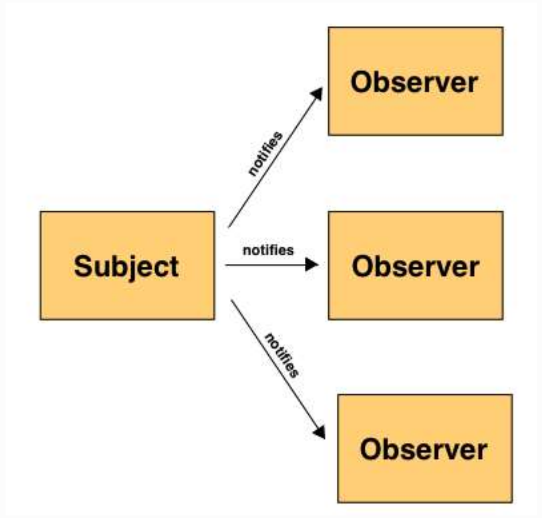
Design Pattern Link

JS Implementation

Observer Design Pattern in React

MobX is a popular library for state management in React applications. It can be used to implement the Observer des
pattern, which involves a subject that is being observed by one or more observers, and the observers are notified
 whenever the subject changes.

With MobX, you can create an observable store that acts as the subject, and components that observe the store and re-render whenever the store changes.



sourcemind

Example

```
import React from "react"
import ReactDOM from "react-dom"
import { makeAutoObservable } from "mobx"
import { observer } from "mobx-react-lite"

class Timer {
    secondsPassed = 0

    constructor() {
        makeAutoObservable(this)
    }

    increaseTimer() {
        this.secondsPassed += 1
    }
}

const myTimer = new Timer()

// A function component wrapped with `observer` will react
// to any future change in an observable it used before.
const TimerView = observer(({ timer }) => <span>Seconds passed: {timer.secondsPassed}</span>)

ReactDOM.render(<TimerView timer={myTimer} />, document.body)

setInterval(() => {
    myTimer.increaseTimer()
}, 1000)
```

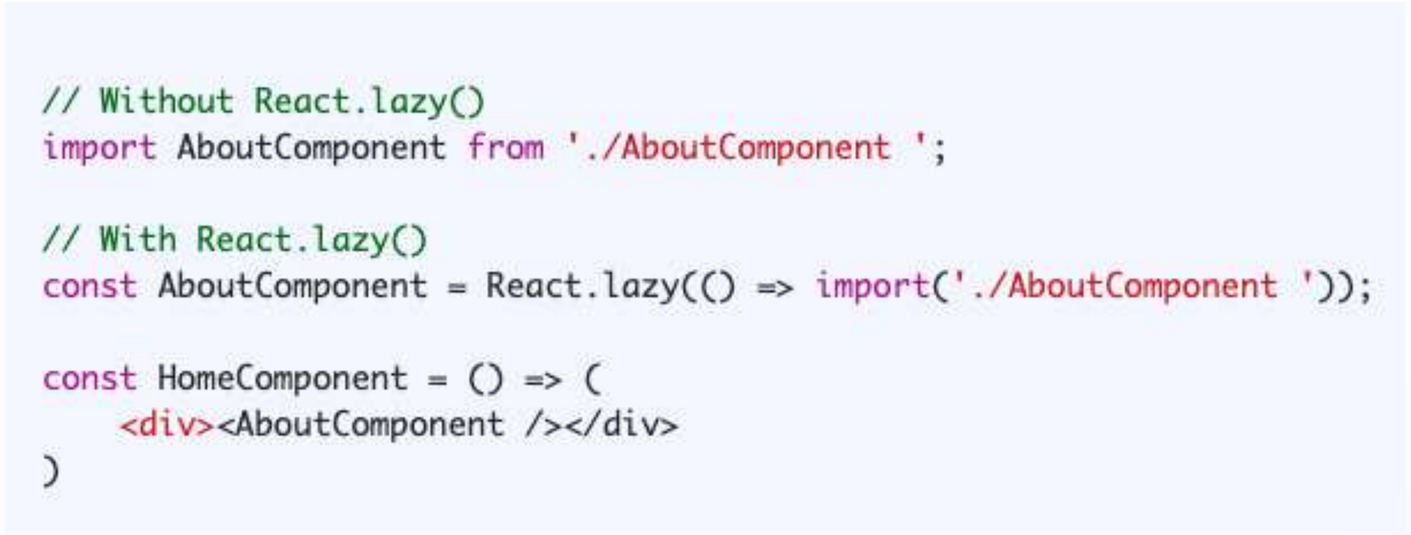
sourcemind

Lazy Loading Design pattern

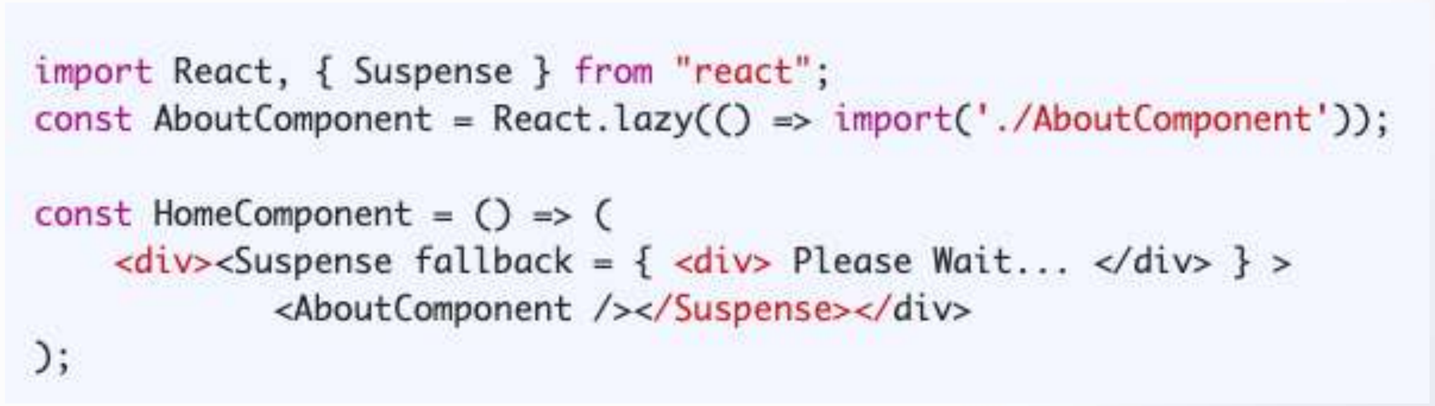
- Lazy loading is one of the most common design patterns used in web and mobile development. It is widely used with frameworks like Angular and React **to increase an application's performance by reducing initial loading time**.

- **In the earlier versions of React, lazy loading was implemented using third-party libraries. However, React introduced two new native features to implement lazy loading with the React v16.6 update.**

React.lazy()



```
// Without React.lazy()
import AboutComponent from './AboutComponent ';

// With React.lazy()
const AboutComponent = React.lazy(() => import('./AboutComponent '));

const HomeComponent = () => (
    <div><AboutComponent /></div>
)
```



```
import React, { Suspense } from "react";
const AboutComponent = React.lazy(() => import('./AboutComponent'));

const HomeComponent = () => (
    <div><Suspense fallback = { <div> Please Wait... </div> } >
            <AboutComponent /></Suspense></div>
);
```

- As discussed, lazy loading has many benefits. But overusing it can have a significant negative impact on your applications. So, it is essential to understand when we should use lazy loading and when we should not.
- You should not opt for lazy loading if your application has a small bundle size. There is no point in splitting a small bundle into pieces, and it will only increase coding and configuring effort.
- Also, there are special application types like e-commerce sites that can be negative impacted by lazy loading. For example, users like to scroll through quickly when searching for items. If you lazy load shopping items, it will break the scrolling speed and result in a bad user experience. So, you should analyze the company’s website usage before using lazy loading.