Mastering

React Design

Patterns

Elevate Your Coding Game

SWIPE TO LEARN

RAJ BHENSDADIYA

- Get ready to explore React design patterns with advanced, production-ready examples.
- Whether you're a seasoned pro or new to React, this carousel post will empower you to excel in the world of React!

RAJ BHENSDADIYA

The Singleton Pattern

- **What is it?**
  - The Singleton Pattern ensures a class has only one instance and provides a global point of access.

- **Why use it?**
  - For managing application-wide resources like configuration or global state.

- **Real-world example:** In a complex React app, a single instance of a configuration manager ensures consistent settings.

JS singleton.js

```
import { useState } from 'react';

const ConfigManager = () => {
  const [config, setConfig] = useState({});
  const setConfigValue = (key, value) => {
    setConfig({ ...config, [key]: value });
  };
  const getConfigValue = (key) => {
    return config[key];
  };
  return {
    setConfigValue,
    getConfigValue,
  };
};

const App = () => {
  const configManager = ConfigManager();
  // Setting configuration values
  configManager.setConfigValue('apiUrl', 'https://api.example.com');
  // Retrieving configuration values
  const apiUrl = configManager.getConfigValue('apiUrl');
  return (
    <div>
      <h1>My React App</h1>
      <p>API URL: {apiUrl}</p>
    </div>
  );
};

export default App;
```

The Factory Pattern

- **What is it?**
  - Creates objects without specifying the exact class, allowing for dynamic object creation.

- **Why use it?**
  - Ideal for creating complex components dynamically.

- **Real-world example:** Dynamically generating form elements based on configuration.

JS factory.js

```
import React from 'react';

const InputField = ({ label }) => (
  <div>
    <label>{label}</label>
    <input type="text" />
  </div>
);

const TextArea = ({ label }) => (
  <div>
    <label>{label}</label>
    <textarea rows="4" cols="50"></textarea>
  </div>
);

const createFormElement = (type, label) => {
  switch (type) {
    case 'text':
      return <InputField label={label} />;
    case 'textarea':
      return <TextArea label={label} />;
    default:
      return null;
  }
};

const DynamicForm = () => {
  return (
    <form>
      {createFormElement('text', 'Name')}
      {createFormElement('textarea', 'Description')}
    </form>
  );
};

export default DynamicForm;
```

The Composite Pattern

- **What is it?**
  - Composes objects into tree structures to represent part-whole hierarchies.
- **Why use it?**
  - Building complex UIs with nested components.

- **Real-world example:** Creating a tree-like folder structure for file management.

JS composite.js

```
import React from 'react';

const File = ({ name }) => <div>{name}</div>;

const Folder = ({ name, children }) => (
  <div>
    <strong>{name}</strong>
    <div style={{ marginLeft: '20px' }}>{children}</div>
  </div>
);

const FileSystem = () => (
  <Folder name="Root">
    <Folder name="Documents">
      <File name="document1.txt" />
      <File name="document2.txt" />
    </Folder>
    <Folder name="Images">
      <File name="image1.jpg" />
      <File name="image2.jpg" />
    </Folder>
  </Folder>
);

export default FileSystem;
```

The Component Pattern

- **What is it?**
  - Organizes components into a hierarchy for reusability.

- **Why use it?**
  - Create UIs with individual, interchangeable parts.

- **Real-world example:** Building a customizable dashboard with reusable widgets.

JS component.js

```jsx
import React from 'react';

const Widget = ({ title, content }) => (
  <div className="widget">
    <h2>{title}</h2>
    <div className="widget-content">{content}</div>
  </div>
);

const WeatherWidget = () =>
      <Widget
        title="Weather Widget"
        content="Today's weather: Sunny"
      />;

const StockWidget = () =>
      <Widget
        title="Stock Widget"
        content="Stock price: $100"
      />;

const CustomDashboard = () => (
  <div className="dashboard">
    <WeatherWidget />
    <StockWidget />
  </div>
);

export default CustomDashboard;
```

The Observer Pattern

- **What is it?**
  - Defines a one-to-many relationship between objects.
- **Why use it?**
  - Keep your components in sync without tight coupling.

- **Real-world example:** Implementing a chat application with real-time updates.

JS observer.js

```
import React, { useState, useEffect } from 'react';

const ChatApp = () => {
  const [messages, setMessages] = useState([]);
  const [newMessage, setNewMessage] = useState('');

  const sendMessage = () => {
    // Send message to the server
    // Server broadcasts the message to all clients
  };

  useEffect(() => {
    // Listen for incoming messages from the server
    // Update the messages state when a new message arrives
  }, []);

  return (
    <div>
      <div className="chat-messages">
        {messages.map((message, index) => (
          <div key={index}>{message}</div>
        ))}
      </div>
      <input
        type="text"
        value={newMessage}
        onChange={(e) => setNewMessage(e.target.value)} />
      <button onClick={sendMessage}>Send</button>
    </div>
  );
};

export default ChatApp;
```

The Render Props Pattern

- **What is it?**
    - Shares code between components using a prop with a function value.

- **Why use it?**
    - Reuse logic in multiple components.

- **Real-world example:** Implementing a data fetching component with render prop.

JS renderProps.js

```
import React, { useState, useEffect } from 'react';

const DataFetcher = ({ render }) => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // Fetch data from an API
    // Update the data and loading state
  }, []);

  return render(data, loading);
};

const App = () => (
  <DataFetcher
    render={(data, loading) => (
      <div>
        {loading ? (
          <p>Loading...</p>
        ) : (
          <ul>
            {data.map((item) => (
              <li key={item.id}>{item.name}</li>
            ))}
          </ul>
        )}
      </div>
    )}
  />
);

export default App;
```

The Higher Order Component (HOC) Pattern

- **What is it?**
  - A function that takes a component and returns a new component.

- **Why use it?**
  - Reuse component logic and enhance functionality.

- **Real-world example:** Adding authentication to a protected route.

JS HOC.js

```
import React, { useState, useEffect } from 'react';

// HOC function to add authentication logic to a component
const withAuth = (WrappedComponent) => {
  return function AuthComponent(props) {
    const [isAuthenticated, setIsAuthenticated] = useState(false);

    useEffect(() => {
      // Check if the user is authenticated
      // (e.g., by checking a token)
      const isAuthenticated = checkAuthentication();
      setIsAuthenticated(isAuthenticated);
    }, []);

    return isAuthenticated
           ? <WrappedComponent {...props} />
           : <div>Please log in to access this page.</div>;
  };
};

// Usage of the HOC to protect a route
const ProtectedRoute = () => {
  return <div>This is a protected route.</div>;
};

const AuthProtectedRoute = withAuth(ProtectedRoute);

export default AuthProtectedRoute;
```

I'm Raj.

Software Engineer;
passionate for building and shipping scalable software.

I am open for full-stack development projects. (JavaScript, Python, AI/ML, DevOps)

Let's connect and expand our professional network together.

I'm excited to collaborate, exchange ideas, and contribute to impactful projects.