For Source Code

Github Link: https://github.com/obrm/react-best-practices-design-pattens/tree/master

Best Practices and Design Patterns in React.js for High-Quality Applications

Ori Baram · Follow

20 min read · Mar 30

Hey there, if you love using React.js like we do, you know it's an awesome library for creating user interfaces. It's simple, flexible, and super speedy. But as your app gets bigger, managing components, state, and logic becomes more of a challenge. To keep things running smoothly and make your code easy to understand, it's important to follow best practices and design patterns.

In this guide, we'll show you some of the most useful practices and patterns for working with React.js. We'll cover things like how to structure your folders, separate concerns, and design components. These tips will make your life as a developer a whole lot easier. They will help you build applications that are easy to maintain, scale, and perform well. We'll show you lots of real-life code examples to make things super clear and give you practical advice on how to put these best practices to work in your own projects. Plus there is a bonus at the end — a GitHub repo with all the code mentioned in this article!

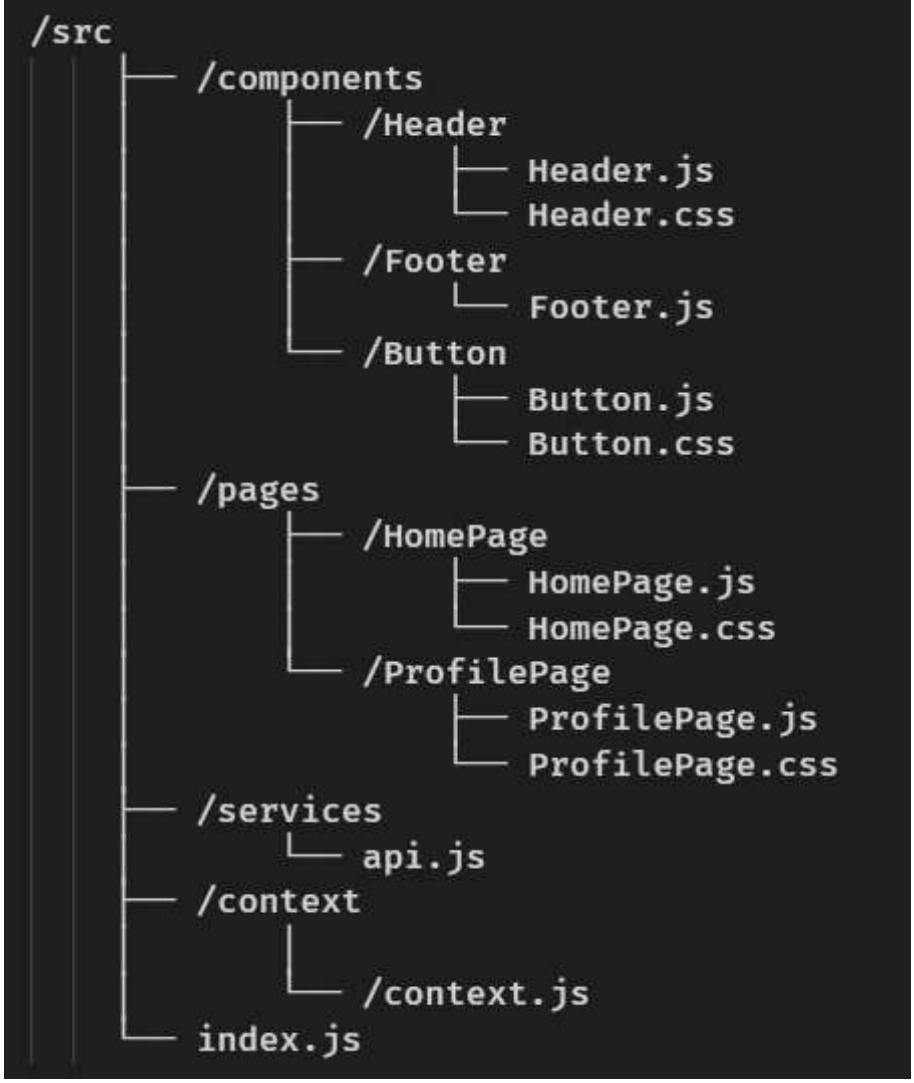
So, let's get started and make our React apps even better!

1. Best Practices for Building Scalable React Applications

If you are building a React application and want to make sure it stays organized and scalable, then you need to follow some best practices! In this section, we'll cover four essential best practices that will help you build maintainable, scalable, and efficient React applications. These practices include organizing your project files in a feature-based folder structure, keeping your components focused and small, using appropriate naming conventions, and separating the responsibilities of Pages and presentational components. By following these practices, you can create applications that are easier to navigate, understand, and maintain.

1.1. Folder Structure

Having an organized folder structure is super important if you want to keep your project hierarchy clear and make it easy to navigate. Check out this example of a feature-based folder structure:



Instead of organizing components by file types, we group them by features. This makes it a lot simpler to find and manage related files, like a component's JavaScript and CSS files.

1.2. Keep Components Small and Focused

When you're building an application, it's super important to create components that are easy to understand, maintain, and test. That's why it's a good idea to keep your components focused and small. If a component is starting to get too big, don't sweat it — just break it down into smaller, more manageable components!

For instance, let's say you have a UserProfile component that's starting to feel overwhelming. You could break it down into smaller components like ProfilePicture, UserName, and UserBio. This will make each component simpler to handle and reuse.

Check out this example:

```jsx
// UserProfile.js
import React from 'react';
import ProfilePicture from './ProfilePicture';
import UserName from './UserName';
import UserBio from './UserBio';

const UserProfile = ({ user }) => (
  <div className="user-profile">
    <ProfilePicture src={user.picture} />
    <UserName name={user.name} />
    <UserBio bio={user.bio} />
  </div>
);

export default UserProfile;
```

1.3. Naming Conventions

When you give your components, props, and state variables names that make sense, it helps other people (and future you!) understand your code more easily. Plus, it makes your code easier to maintain in the long run. Here are some tips to help you name things like a pro:

- For components, use PascalCase (like UserProfile.js)
- For variables and functions, use camelCase (like getUserData())
- And for constants, use UPPERCASE_SNAKE_CASE (like API_URL)

Note that this is very subjective and it up to you to decide by which naming convention you want to follow. The most important thing is to be consistent with your naming style.

1.4. Pages (container components) and Presentational Components

In React, there are two types of components: Pages (container components) and presentational components. Pages handle tasks like fetching data from external sources (like APIs), managing state and logic, and passing data down to the presentational components using props. Meanwhile, presentational components are responsible for rendering UI elements and displaying data passed down from their parent components. By separating these responsibilities, we can create more modular and reusable components.

Example — TodoApp Page:

```
// components/TodoApp.js (Container Component)
import React, { useState } from "react";
import TodoList from "./TodoList";

function TodoApp({ initialTodos }) {
  const [todos, setTodos] = useState(initialTodos);

  const handleToggle = (id) => {
    setTodos(
      todos.map((todo) =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
    );
  };

  return <TodoList todos={todos} onToggle={handleToggle} />;
}

export default TodoApp;
```

TodoList Component:

```
// components/TodoList.js (Presentational Component)
function TodoList({ todos, onToggle }) {
  return (
    <ul>
      {todos.map((todo) => (
        <li key={todo.id} onClick={() => onToggle(todo.id)}>
          {todo.completed ? <s>{todo.text}</s> : todo.text}
        </li>
      ))}
    </ul>
  );
}

export default TodoList;
```

Okay, so when it comes to React components, we can divide them into two types: Pages (or container components) and presentational components. Pages handle stuff like getting data from APIs and managing state and logic, while presentational components show the data that they get from their props and define how things look. By doing this, we can keep our code more organized, easier to understand, and simpler to test.

1.5. Keep it DRY with Array Mapping

Using an array of items and mapping over them can be a nifty trick to avoid code repetition in your React components. Imagine you're building a navigation bar with multiple links, each with its own title, path, and icon. Rather than writing out the same structure and code for each link, you can create an array of objects that hold all the necessary data and dynamically render them with a map function.

Check out this sample code that demonstrates how you can easily create an array of links and map over them to render a navigation bar:

```
const links = [
  { title: 'Home', path: '/home', icon: <HomeIcon /> },
  { title: 'About', path: '/about', icon: <AboutIcon /> },
  { title: 'Contact', path: '/contact', icon: <ContactIcon /> }
];

function Navbar() {
  return (
    <nav>
      <ul>
        {links.map((link) => (
          <li key={link.path}>
            <Link to={link.path}>
              {link.icon}
              <span>{link.title}</span>
            </Link>
          </li>
        ))}
      </ul>
    </nav>
  );
}
```

In the example above, we demonstrated how creating an array of items and mapping over them can help you avoid code repetition in your React components. This technique is not only useful for rendering navigation bars, but also for rendering forms with multiple input fields.

By creating an array of objects that contain the necessary data for each input field, you can map over them to render the input fields dynamically. This can greatly simplify your code and make it more maintainable, especially when dealing with forms that have many input fields.

Here's an example of this:

```
const inputs = [
  { label: 'First Name', type: 'text', name: 'firstName' },
  { label: 'Last Name', type: 'text', name: 'lastName' },
  { label: 'Email', type: 'email', name: 'email' }
];

function Form() {
  return (
    <form>
      {inputs.map((input) => (
        <div key={input.name}>
          <label htmlFor={input.name}>{input.label}</label>
          <input type={input.type} name={input.name} id={input.name} />
        </div>
      ))}
      <button type="submit">Submit</button>
    </form>
  );
}
```

This cool example shows how to use an array of input objects to create dynamic input fields in React. Each input object includes the label, type, and name for each input field. By using the map() function, you can easily loop over the array and render each input field dynamically with the data from each input object.

Using arrays and mapping over them is a great way to create dynamic content in your React components without repeating the same code over and over. This technique makes your components more scalable, reusable and easier to maintain.

2. Separation of Concerns

Separation of concerns is super important in software development. It's all about making sure each part of your app has one job to do. When it comes to React.js, separating the logic from the view can help make your code more maintainable, reusable, and scalable. So let's take a closer look at some examples and techniques to help us achieve this separation and make our React apps even better.

2.1. Custom Hooks

Custom hooks let you take stateful logic out of your components and use it across multiple components. Custom hooks are just JavaScript functions with a special naming convention — they have to start with "use" (like, useForm). These hooks can use other built-in or custom hooks, and they can export an object or an array with all the properties and methods you want to use in your components. Want to see some real-life examples of how custom hooks can make your React.js code even better? Let's dive in!

Example: Custom Hook for Form Input Handling

Imagine you've got a form with loads of input fields. Handling the state and input for each field can get pretty tricky and repetitive. But fear not — we can make things easier with a custom hook called useForm. This hook lets you extract the state and logic for all those fields and keep things neat and tidy.

First up, we use useState to create a values object that holds all our form input values. Then, we make a handleChange method that updates the values object whenever an input changes. Finally, we add a resetForm method that sets everything back to square one.

By popping values, handleChange, and resetForm in an array, we can use these properties and methods in all our components and make our form component way easier to maintain and reuse.

```
// hooks/useForm.js
import { useState } from 'react';

function useForm(initialValues) {
  const [values, setValues] = useState(
    initialValues,
  );

  const handleChange = (event) => {
    const { name, value } = event.target;
    setValues({ ...values, [name]: value });
  };

  const resetForm = () => {
    setValues(initialValues);
  };

  return [values, handleChange, resetForm];
}

export default useForm;
```

Now, we can use the useForm custom hook in our form component:

```
// components/LoginForm.js
import React from "react";
import useForm from "../hooks/useForm";
import ApiService from "../services/ApiService";

function LoginForm() {
  const initialValues = { email: "", password: "" };
  const [values, handleChange, resetForm] = useForm(initialValues);

  const handleSubmit = async (event) => {
    event.preventDefault();

    try {
      await ApiService.post("/register", values);
      resetForm();
    } catch (error) {
      console.error(error);
    }
  };

  return (
    <form onSubmit={handleSubmit}>

      <label htmlFor="email">Email:</label>
      <input
        type="email"
        name="email"
        value={values.email}
        onChange={handleChange}
      />

      <label htmlFor="password">Password:</label>
      <input
        type="password"
        name="password"
        value={values.password}
        onChange={handleChange}
      />

      <button type="submit">Log In</button>
    </form>
  );
}

export default RegistrationForm;
```

The useForm custom hook is the way to go if you want to handle form inputs in any React.js app with ease. Whether it's a login form or a registration form, you name it.

Another example: Custom Hook for Fetching Data

Another thing we can do with custom hooks is fetch data from an API. It's pretty common actually. All we need is to make a custom hook called useFetch that takes care of the state and side effects related to fetching data. Easy-peasy!

```
// hooks/useFetch.js
import { useState, useEffect } from 'react';

function useFetch(url, options) {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);
  const [isLoading, setIsLoading] =
    useState(false);

  useEffect(() => {
    const fetchData = async () => {
      setIsLoading(true);
      try {
        const response = await fetch(
          url,
          options,
        );
        const result = await response.json();
        setData(result);
        setError(null);
      } catch (error) {
        setError(error);
      } finally {
        setIsLoading(false);
      }
    };
  }, [url, options]);
  return { data, error, isLoading };
}

export default useFetch;
```

Now, we can use the useFetch custom hook in any component that requires data from an API:

```jsx
// components/UserList.js
import React from "react";
import useFetch from "../hooks/useFetch";

function UserList() {
  const { data, error, isLoading } = useFetch("https://api.example.com/users");

  if (isLoading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Error: {error.message}</div>;
  }

  return (
    <ul>
      {data.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default UserList;
```

Thanks to the useFetch custom hook, we can easily handle the state and logic for fetching data from the view in our UserList component. This makes our code way easier to maintain and reuse. With useFetch, we can keep all the state and side effects related to fetching data in one tidy spot, then use the resulting data in multiple components. This separation of concerns means we can easily change our data fetching logic without messing with the components that depend on it. All in all, it makes our code more maintainable and scalable.

2.2. Services

If you want to make your React apps more modular, using services is a great way to go. Basically, services are functions or classes that handle business logic like calling APIs, playing around with data, or other helpful tasks. You can import these services and use them in your components or custom hooks. It's an easy way to separate logic from the view and keep things organized.

Example: API Service

Let's say we need to work with a RESTful API to do some CRUD operations on user data. To make things simpler, we can create a service that takes care of all of that. It's a handy way to communicate with the API and get our tasks done efficiently.

```
// services/apiService.js
const API_URL = 'https://api.example.com/users';

async function sendRequest(url, options) {
  const response = await fetch(url, options);
  return await response.json();
}

function createRequestOptions(method, body) {
  return {
    method,
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(body),
  };
}

async function fetchUsers() {
  return await sendRequest(API_URL);
}

async function createUser(user) {
  const options = createRequestOptions('POST', user);
  return await sendRequest(API_URL, options);
}

async function updateUser(userId, user) {
  const options = createRequestOptions('PUT', user);
  return await sendRequest(`${API_URL}/${userId}`, options);
}

async function deleteUser(userId) {
  const options = createRequestOptions('DELETE');
  await sendRequest(`${API_URL}/${userId}`, options);
}

export default {
  fetchUsers,
  createUser,
  updateUser,
  deleteUser,
};
```

Now, we can use the API service in our components or custom hooks:

```
// components/UserList.js
import React, { useEffect, useState } from "react";
import apiService from "../services/apiService";

function UserList() {
  const [users, setUsers] = useState([]);
  const [error, setError] = useState(null);
  const [isLoading, setIsLoading] = useState(false);

  useEffect(() => {
    const fetchData = async () => {
      setIsLoading(true);
      try {
        const data = await apiService.fetchUsers();
        setUsers(data);
        setError(null);
      } catch (error) {
        setError(error);
      } finally {
        setIsLoading(false);
      }
    };

    fetchData();
  }, []);

  if (isLoading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Error: {error.message}</div>;
  }

  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default UserList;
```

Thanks to the API service, we can keep all the API interaction logic separate from the view in our UserList component. That way, we can easily maintain and reuse our code without any fuss. It's a great way to make sure everything stays organized and easy to manage.

Example: Utility Service

Let's say we need to do some useful stuff like format dates, calculate sums, or make unique IDs. We can create a handy utility service that'll take care of all that. It's like our own little helper that we can call on whenever we need to get some useful work done.

```
// services/utilityService.js
function formatDate(date, format) {
  // Your date formatting logic here
}

function calculateSum(numbers) {
  return numbers.reduce((sum, number) => sum + number, 0);
}

function generateUUID() {
  // Your UUID generation logic here
}

export default { formatDate, calculateSum, generateUUID };
```

Cool! We can easily use our utility service in our components or custom hooks to make our code more efficient and reusable. It's a great way to keep things organized and make sure everything runs smoothly.

```
// components/TransactionList.js
import React from "react";
import utilityService from "../services/utilityService";

function TransactionList({ transactions }) {
  const totalAmount = utilityService.calculateSum(
    transactions.map((transaction) => transaction.amount)
  );

  return (
    <div>
      <ul>
        {transactions.map((transaction) => (
          <li key={transaction.id}>
            {transaction.description} - {utilityService.formatDate(transaction.date, "MM/DD/YYYY")}
          </li>
        ))}
      </ul>
      <div>Total Amount: {totalAmount}</div>
    </div>
  );
}

export default TransactionList;
```

By using the utility service, we've kept the utility logic separate from the view, making our TransactionList component easier to manage and reuse.

When to use a Custom Hook and when to use a Service Function

When deciding whether to use a custom hook or a service function, it's important to consider the specific use case and requirements of your application. **Custom hooks** are more appropriate when you need to **manage state or context**, or when you want to **reuse**

functionality across multiple components. On the other hand, service functions are better suited for reusable **logic that does not depend on state or context.**

In practice, you may find that some functionality fits better as a custom hook, while other functionality works better as a service function. For example, here are some more examples of when you might choose to use each approach:

Custom Hooks:

- **Managing complex state:** If you need to manage complex state across multiple components, a custom hook can be a great way to encapsulate that logic and make it reusable. For example, if you need to manage a user's preferences and match them with your product, a custom hook like `useUserSettings` could help track the state of the user's preferences and provide functionality for checking the match with your product. The custom hook utilizes `useState` to track the state of the user's preferences and the match with your product, and provides a function to check the match based on the user's preferences and your product's categories.

- **Context-dependent functionality:** A custom hook can help access the context from your application and provide reusable functionality. For example, if you need to perform user authentication and manage the user's state across multiple components, a custom hook like `useLoggedUser` could help you access the user's state and provide functionality for updating the state. The custom hook can utilize the `LoggedUserContext` to access the user's state and provides functions for setting a new user, setting an existing user, updating the user's state, etc.

Service Functions:

- **Task-specific functionality:** If you need to perform a specific task that doesn't depend on context or state, a service function can be a more appropriate approach. For example, if you need to validate data like an email address or a password, or formatting data such as currency values or dates, or sorting data such as a list of products or search results, service functions could provide these functionalities without requiring state or context.

- **Standalone functionality:** If you need to provide functionality that can be used across multiple applications or platforms, a service function can be a good way to encapsulate that logic and make it reusable. For example, if you need to provide a function for image processing, or file storage, or email sending, or payment processing, service functions could provide these functionalities in a way that can be used across multiple applications or platforms.

3. Component Design Patterns

When you're building React apps, it's important to make components that are easy to work with, reusable, and don't give you a headache when you're trying to update them. That's

where **design patterns** come in. In this section, we're going to check out some design patterns that can help you make flexible and reusable components. These patterns will help you divide big components into smaller, more manageable pieces, making them easier to understand, test, and maintain.

3.1. Higher-Order Components (HOCs)

Higher-Order Components (HOCs) are pretty neat! They're functions that take a component and return a new component with more features or behaviors. Basically, they let you reuse component logic and handle things like authentication or data fetching.

For example, you can use an HOC to check if a user is authenticated and then wrap a component that needs authentication with it. If the user is authenticated, the wrapped component will show up. But if not, they'll be sent to the login page. Here's an example to give you an idea of what it looks like:

```
function requireAuth(Component) {
  return function AuthenticatedComponent(props) {
    const isAuthenticated = checkAuth();

    if (!isAuthenticated) {
      return <Redirect to="/login" />;
    }

    return <Component {...props} />;
  };
}

const WrappedComponent = requireAuth(MyComponent);
```

Alright, so in this example we got this function called requireAuth, which gets a component as input and returns a new one that checks if the user is logged in. If not, it takes the user to the login page. Otherwise, the wrapped component is rendered with the extra props from the HOC.

Now, we can also use HOCs to wrap a component that needs some data fetching. Basically, the HOC will fetch the data and pass it down to the component as props.

Check it out:

```
function withDataFetching(Component) {
  return function DataFetchingComponent(props) {
    const [data, setData] = useState(null);
    const [isLoading, setIsLoading] = useState(false);
    const [error, setError] = useState(null);

    useEffect(() => {
      setIsLoading(true);

      fetch(props.url)
        .then((response) => response.json())
        .then((data) => {
          setIsLoading(false);
          setData(data);
        })
        .catch((error) => {
          setIsLoading(false);
          setError(error);
        });
    }, [props.url]);

    return (
      <Component {...props} data={data} isLoading={isLoading} error={error} />
    );
  };
}

const WrappedComponent = withDataFetching(MyComponent);
```

So in this example, the withDataFetching function takes a component and gets data from a URL to pass it down as props. If there's an error, it goes into the error prop, and if the data's still loading, the isLoading prop is set to true.

There are plenty of other rad **HOCs** you can use with React too! One of them is **memo**. Instead of causing unnecessary re-renders even if a component's props remain the same, thereby slowing down your React app, **memo** allows you to optimize your component so that it only re-renders if the props it receives have changed, making your app more efficient and improving performance. This can be a real game-changer for optimizing performance in your React app.

But there are also some other **HOCs** that are commonly used in popular React libraries. For example, there's **connect** from the react-redux library, which connects a component to the **Redux store**, so it can access the store's state and dispatch functions. This means you can easily create components that react to changes in the Redux store. Another one is **withFirebase** from the **react-redux-firebase** library, which provides a **Firebase** instance to a component, so it can interact with the Firebase backend.

HOCs are pretty cool because they let you reuse component logic and deal with problems that affect the whole app. It's like killing two birds with one stone! HOCs make it so your components are nice and tidy, they're easy to take care of, they're focused and easy to maintain.

3.2. Render Props

Render props are a fancy term in React that lets you share code between components by passing a function as a prop to a child component, and it can be more flexible than HOCs in some cases.

This function, when called, returns a React element (usually JSX). The idea behind render props is to let the parent component take care of rendering some part of the component. So, the child component can focus on providing the necessary functionality, while the parent component determines how the rendered content should look like.

Let's see an example of how render props work in real life:

Imagine you're building an online store app, and you need to display a list of products in different parts of your app. You want to fetch the products and display them in different ways, such as a grid view or a list view. Here's how you can achieve this using the render props pattern:

First, create a ProductFetcher component that fetches the product data and accepts a render prop (a function):

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

const ProductFetcher = ({ render }) => {
  const [products, setProducts] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    // Fetch products from API (using a service or directly making an API call)
    fetchProducts().then((fetchedProducts) => {
      setProducts(fetchedProducts);
      setIsLoading(false);
    });
  }, []);

  if (isLoading) {
    return <div>Loading...</div>;
  }

  return render(products);
};

export default ProductFetcher;
```

In this example, the ProductFetcher component is responsible for fetching the products and managing the loading state. It accepts a render prop, which is a function, and calls it with the products as an argument.

Now, let's create two separate components for displaying the products in grid and list views:

```
// ProductGrid.js
import React from 'react';

const ProductGrid = ({ products }) => {
  return (
    <div className="product-grid">
      {products.map((product) => (
        <div key={product.id} className="product-grid-item">
          <img src={product.image} alt={product.name} />
          <h2>{product.name}</h2>
          <p>${product.price}</p>
        </div>
      ))}
    </div>
  );
};

export default ProductGrid;
```

```
// ProductList.js
import React from 'react';

const ProductList = ({ products }) => {
  return (
    <ul className="product-list">
      {products.map((product) => (
        <li key={product.id} className="product-list-item">
          <img src={product.image} alt={product.name} />
          <h2>{product.name}</h2>
          <p>${product.price}</p>
        </li>
      ))}
    </ul>
  );
};

export default ProductList;
```

Finally, use the ProductFetcher component and pass the appropriate render function to display the products in different formats:

```
import React from 'react';
import ProductFetcher from './ProductFetcher';
import ProductGrid from './ProductGrid';
import ProductList from './ProductList';

const App = () => {
  return (
    <div>
      <h1>Grid View</h1>
      <ProductFetcher render={(products) => <ProductGrid products={products} />} />

      <h1>List View</h1>
      <ProductFetcher render={(products) => <ProductList products={products} />} />
    </div>
  );
};

export default App;
```

So, in this example, we have the ProductFetcher component that fetches and manages the product data. The ProductGrid and ProductList components are responsible for displaying the products in different formats. With the help of the render prop, the ProductFetcher component fetches the product data, and the parent component (in this case, App) determines how the products should be displayed.

By using the **render props** pattern, we get some sweet benefits, like:

1. **Separation of concerns:** The ProductFetcher component is only responsible for fetching and managing the loading state of products. It doesn't have to worry about how the products are displayed.
2. **Reusability:** The ProductFetcher component can be used throughout the app to fetch and display products in different formats without much hassle.
3. **Flexibility:** The parent component can decide how the products should be displayed, giving us the freedom to customize the UI without tinkering with the fetching logic.

3.3. Compound Components

Compound components are a cool design trick that helps you create better and more flexible components. You can group together related components and control their behavior from a parent component, which can make your code much more organized and user-friendly.

One popular example of this is an accordion component, which usually has lots of panels that expand and collapse when you click them. With compound components, you can build a reusable accordion component that's both easy to customize and easy to use.

Let's check out an example of how to make an Accordion component using compound components:

```
function Accordion({ children }) {
  const [selectedIndex, setSelectedIndex] = useState(0);

  return (
    <div>
      {React.Children.map(children, (child, index) => {
        if (child.type.name === 'AccordionHeader') {
          return React.cloneElement(child, {
            key: index,
            isOpen: index === selectedIndex,
            onClick: () => setSelectedIndex(index),
          });
        }
        if (child.type.name === 'AccordionContent') {
          return React.cloneElement(child, {
            key: index,
            isOpen: index === selectedIndex,
          });
        }
        return null;
      })}
    </div>
  );
}

function AccordionHeader({ isOpen, onClick, children }) {
  return (
    <div onClick={onClick}>
      <h3>{children}</h3>
      {isOpen ? <i>-</i> : <i>+</i>}
    </div>
  );
}

function AccordionContent({ isOpen, children }) {
  return isOpen ? <div>{children}</div> : null;
}
```

In this example, we got this Accordion component, where we render a list of children and loop through them with React.Children.map to decide when to display them. We use React.cloneElement to give AccordionHeader and AccordionContent some extra props to control how they behave.

Here's how all of these component would look in a component called MyAccordion:

```
import { useState } from 'react';
import Accordion from './Accordion';
import AccordionHeader from './AccordionHeader';
import AccordionContent from './AccordionContent';

function MyAccordion({ items }) {
  const [selectedIndex, setSelectedIndex] = useState(0);

  return (
    <Accordion>
      {items.map((item, index) => (
        <div key={index}>
          <AccordionHeader
            isOpen={selectedIndex === index}
            onClick={() => setSelectedIndex(index)}
          >
            {item.header}
          </AccordionHeader>
          <AccordionContent isOpen={selectedIndex === index}>
            {item.content}
          </AccordionContent>
        </div>
      ))}
    </Accordion>
  );
}

export default MyAccordion;
```

You must agree with me that this component is way more comprehensive, easy to reason about and clean.

And if that's not enough for you, we also got a Tabs component that shows different content when you click on different tabs. Just like with the Accordion, we can use compound components to make it easier to build and use. Want to see an example of how it's done?

Check it out:

```
function Tabs({ children }) {
  const [activeTab, setActiveTab] = useState(0);

  return (
    <div>
      <div>
        {React.Children.map(children, (child, index) => {
          if (child.type.name === 'Tab') {
            return React.cloneElement(child, {
              key: index,
              isActive: index === activeTab,
              onClick: () => setActiveTab(index),
            });
          }
          return null;
        })}
      </div>
      {React.Children.map(children, (child, index) => {
        if (child.type.name === 'TabContent') {
          return React.cloneElement(child, {
            key: index,
            isActive: index === activeTab,
          });
        }
        return null;
      })}
    </div>
  );
}

function Tab({ isActive, onClick, children }) {
  return (
    <div onClick={onClick}>
      <h3 style={{ color: isActive ? 'red' : 'black' }}>{children}</h3>
    </div>
  );
}

function TabContent({ isActive, children }) {
  return isActive ? <div>{children}</div> : null;
}
```

In this example we got a Tabs component that displays a list of tabs with different content. We use React.cloneElement to give the Tab and TabContent components some extra props for controlling their behavior.

And here's how it will look combined all together:

```
import { useState } from 'react';
import Tabs from './Tabs';
import Tab from './Tab';
import TabContent from './TabContent';

function MyTabs({ tabs }) {
  const [activeTab, setActiveTab] = useState(0);

  return (
    <Tabs>
      {tabs.map((tab, index) => (
        <Tab
          key={index}
          isActive={activeTab === index}
          onClick={() => setActiveTab(index)}
        >
          {tab.title}
        </Tab>
      ))}
      {tabs.map((tab, index) => (
        <TabContent key={index} isActive={activeTab === index}>
          {tab.content}
        </TabContent>
      ))}
    </Tabs>
  );
}

export default MyTabs;
```

Compound components are really cool and can make your components super flexible and expressive. Instead of just having a bunch of separate components, you can group them together in a parent component to make things way more intuitive and user-friendly. Stuff like Accordions, Tabs, and Dropdowns can get pretty complicated in a web app, but with compound components, you can simplify everything and make it easy-peasy for your users to navigate.

3.4. Optimize Rendering with React.memo

React.memo, that we already mentioned, is like a secret weapon for optimizing functional components. It makes sure that the component only re-renders when the props change, which can be a huge performance boost, especially when you're dealing with complex or expensive-to-compute props. Basically, it helps you avoid unnecessary renders and keeps your app running smoothly.

Check out this example:

```
import React from 'react';

const UserProfileName = ({ name }) => (
  <h2 className="user-profile-name">{name}</h2>
);

export default React.memo(UserProfileName);
```

In this example, the UserProfileName component will only re-render when its name prop changes, avoiding unnecessary updates and improving performance.

3.5. Use React.lazy for Code Splitting and Lazy Loading

React.lazy is a really neat tool that can help speed up your app and make it more efficient. It's a built-in feature that allows you to split up your components and load them only when you actually need them. This is great if you have a really big app with tons of different components that aren't all needed right away. With React.lazy, you can pick and choose which components to load, and exactly when to load them. This means that your app can start up faster, use less memory, and generally be much more efficient. Plus, it's super easy to use — all you need to do is wrap your component in a special function, and React will take care of the rest.

Let's say you're building an e-commerce app with React. You have a product listing page that displays all the products in your inventory. Each product has an image, name, price, and description.

You also have a product details page that displays more information about a single product, such as reviews, ratings, and related products.

If you load all the code for the product details page on the product listing page, it could slow down the initial load time and cause your app to feel sluggish. That's where React.lazy comes in.

Here's how you can use React.lazy to lazily load the ProductDetails page only when the user clicks on a product:

First, you create a separate file for the ProductDetails component:

```
import React from "react";

function ProductDetails(props) {
  return (
    <div>
      <h1>{props.product.name}</h1>
      <img src={props.product.imageUrl} alt={props.product.name} />
      <p>{props.product.description}</p>
      <p>Price: {props.product.price}</p>
      <p>Rating: {props.product.rating}</p>
      {/* other details */}
    </div>
  );
}

export default ProductDetails;
```

Next, you wrap the ProductDetails component in a call to React.lazy:

```
const ProductDetails = React.lazy(() => import("./ProductDetails"));
```

This tells React to lazily load the ProductDetails component only when it's needed.

Finally, you use the ProductDetails component in your product listing page, and wrap it in a Suspense component to handle the loading state:

```
import React, { Suspense } from "react";

const ProductDetails = React.lazy(() => import("./ProductDetails"));

function ProductListing(props) {
  const [selectedProduct, setSelectedProduct] = useState(null);

  function handleProductClick(product) {
    setSelectedProduct(product);
  }

  return (
    <div>
      <h1>Product Listing Page</h1>
      {props.products.map((product) => (
        <div key={product.id} onClick={() => handleProductClick(product)}>
          <h2>{product.name}</h2>
          <img src={product.imageUrl} alt={product.name} />
          <p>Price: {product.price}</p>
        </div>
      ))}
      {selectedProduct && (
        <Suspense fallback={<div>Loading...</div>}>
          <ProductDetails product={selectedProduct} />
        </Suspense>
      )}
    </div>
  );
}

export default ProductListing;
```

When the user clicks on a product, the handleProductClick function sets the selectedProduct state to the clicked product. If selectedProduct is not null, the ProductDetails component is rendered inside a Suspense component.

The Suspense component displays a loading state until the ProductDetails component is loaded.

That's it! With React.lazy, you can lazily load components when you need them, making your app faster and more efficient.

3.6. Use Context and React.createContext for Global State Management

The **Context API** is a great way to share state across multiple components without having to pass props down through the component tree. This can greatly simplify your state management and make your code more maintainable.

Take a look at this sample:

```
import React, { createContext, useContext, useState } from 'react';

const ThemeContext = createContext();

const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

const useTheme = () => useContext(ThemeContext);

export { ThemeProvider, useTheme };
```

So, we have a ThemeContext and a ThemeProvider component that we wrap around the app. It's great because all components can use the useTheme hook to access and update the theme without passing it around like a hot potato.

Here's an example for this:

```
import React from 'react';
import { useTheme } from './ThemeContext';

const ThemeToggle = () => {
  const { theme, setTheme } = useTheme();

  const toggleTheme = () => {
    setTheme(theme === 'light' ? 'dark' : 'light');
  };

  return (
    <button onClick={toggleTheme}>
      Switch to {theme === 'light' ? 'dark' : 'light'} theme
    </button>
  );
};

export default ThemeToggle;
```

3.7. Reducers for Managing Complex State

Reducers are like a super handy tool that can help you manage complex state in a predictable and testable way. They're a functional programming pattern that works great with useReducer hook or libraries like Redux. Basically, you pass them the current state and an action, and they return the new state.

Example: Todo App

So, here's what you do first: make a function that's gonna be the reducer and the actions you want:

```
// todoReducer.js
export const ADD_TODO = 'ADD_TODO';
export const TOGGLE_TODO = 'TOGGLE_TODO';

export const todoReducer = (state, action) => {
  switch (action.type) {
    case ADD_TODO:
      return [...state, { id: Date.now(), text: action.text, completed: false }];
    case TOGGLE_TODO:
      return state.map(todo =>
        todo.id === action.id ? { ...todo, completed: !todo.completed } : todo
      );
    default:
      return state;
  }
};
```

Then, you can use the todoReducer with the useReducer hook. It's as simple as that:

```
// TodoApp.js
import React, { useReducer } from 'react';
import { todoReducer, ADD_TODO, TOGGLE_TODO } from './todoReducer';

const TodoApp = () => {
  const [todos, dispatch] = useReducer(todoReducer, []);

  const addTodo = text => {
    dispatch({ type: ADD_TODO, text });
  };

  const toggleTodo = id => {
    dispatch({ type: TOGGLE_TODO, id });
  };

  // Render the todos and form for adding new todos
};

export default TodoApp;
```

3.8. Use Keys When Rendering Lists

When you're rendering a list of stuff in React, make sure to give each item a unique key. This helps React work its magic and make things faster and smoother. Here's how you do it:

```
import React from 'react';

const TodoList = ({ todos, toggleTodo }) => (
  <ul>
    {todos.map(todo => (
      <li key={todo.id} onClick={() => toggleTodo(todo.id)}>
        {todo.completed ? <s>{todo.text}</s> : todo.text}
      </li>
    ))}
  </ul>
);

export default TodoList;
```

In this example, we give each item in the todos array a unique key, which helps React render things faster and avoid messing up the DOM, optimizing rendering and avoid unnecessary DOM updates.

By sticking to these best practices, you'll be able to create React apps that are way easier to manage, scale, and keep running smoothly. By breaking things down into smaller, more focused components, optimizing rendering, and only loading the things you actually need, and handling state efficiently, you'll be well on your way to creating a top-notch React app. So make sure to keep these tips in mind as you tackle your own React projects to ensure they'll last for the long haul.

Conclusion

To wrap it up, all the cool design patterns and best practices we've talked about are super important for making awesome **React.js** apps. If you want to write clean, easy-to-read code that's simple to maintain and scale up, then you gotta **organize your folders right**, split up your components into **custom hooks**, **services**, and **presentational parts**, use the **appropriate names** for things, and work with **HOCs**, **render props, compound components,** and **reducers**. And don't forget the basics like keeping your **components small**, optimizing your rendering with **React.memo**, code splitting with **React.lazy**, using the **Context API**, and using **keys** when you're doing list rendering. Following all these tips and tricks is the way to a robust and efficient application that both you, fellow developers and your users will love!

All the code mentioned in this article can be found in this repository.

Design Patterns | Reactjs | Best Practices | Code Organization | Code Scalability

Follow

Written by Ori Baram

256 Followers

A programmer & instructor in JavaScript, HTML, CSS, dedicated to learning, teaching, evolving, and sharing a passion for coding with the world.

More from Ori Baram