

Sure Darshan! Here's your **enhanced and eye-catching documentation** for HOCs (Higher-Order Components) in React, with clear explanations, real-world analogies, emoji annotations, and zero content removed — only **additions and enhancements** made ☒.

🧠 Higher-Order Component (HOC) – Full Documentation Guide

Welcome to the ultimate HOC documentation! 🚀 This guide explains the **what**, **why**, and **how** of Higher-Order Components in your project, helping you **reuse logic**, **add behavior**, and keep components clean, smart, and DRY (Don't Repeat Yourself). 🧠

🧠 What is a Higher-Order Component?

A **Higher-Order Component (HOC)** is an advanced pattern in React for **reusing component logic without modifying the original component**.

In simple words: 🦋 An HOC is a *function* that takes a component and returns a new one with enhanced behavior.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

📦 HOCs act like wrappers or decorators 🏠 for your components, making them more powerful, smarter, or dynamic.

🎯 When Should You Use HOCs?

☒ When multiple components need:

- Badging or visual indicators 🏷️
 - Filtering, sorting, or logic transformation 📊
 - Authentication checks 🔒
 - Analytics or logging 📈
 - Lazy loading, suspense, or fallback UI 🛑
-

💎 HOCs in This Project

We use **two reusable HOCs**:

1. 📌 **withRestaurantBadges** – Adds dynamic badges to restaurant cards
2. 🔍 **withFilteredRestaurants** – Filters and sorts restaurant lists intelligently

Let's dive into each one. 🧩

1 withRestaurantBadges HOC

💡 What It Does

Enhances a `RestaurantCard` component by dynamically adding **badges** based on the restaurant's data.

🔧 Features:

Badge	Condition
🏠 Promoted	If <code>resData.info.promoted</code> is <code>true</code>
🌟 Top Rated	If <code>avgRating</code> <code>>= 4.5</code>
🟡 Low Rating	If <code>avgRating</code> <code>< 4.0</code> and <code>> 0</code>
⚡ Fast Delivery	If <code>deliveryTime</code> <code><= 25</code>
👤 Popular	If <code>totalRatings</code> <code>≥ 10K</code> (parsed from strings like "20K+", etc.)

🔗 Implementation

```
// src/hocs/withRestaurantBadges.js
import React from 'react';

const withRestaurantBadges = (WrappedComponent) => {
  return (props) => {
    const { resData } = props;
    const info = resData?.info;
    const badges = [];

    // --- Badge Logic ---
    if (info?.promoted) {
      badges.push({ text: 'Promoted 🏠', color: 'bg-black' });
    }
    if (info?.avgRating >= 4.5) {
      badges.push({ text: 'Top Rated 🌟', color: 'bg-green-700' });
    }
    if (info?.avgRating < 4.0 && info?.avgRating > 0) {
      badges.push({ text: 'Low Rating 🟡', color: 'bg-yellow-600' });
    }
    if (info?.sla?.deliveryTime <= 25) {
      badges.push({ text: 'Fast Delivery ⚡', color: 'bg-blue-600' });
    }

    // 👤 Popular logic - parse "10K+" or "20K+"
    try {
      if (info?.totalRatingsString) {
        const ratings = parseInt(info.totalRatingsString.replace(/\D/g, ''));
        if ((info.totalRatingsString.includes('K') && ratings >= 10) ||
          (!info.totalRatingsString.includes('K') && ratings > 10000)) {
          badges.push({ text: 'Popular 👤', color: 'bg-red-600' });
        }
      }
    } catch {}
  };
};
```

```

    }
  }
} catch (e) {
  console.error("Could not parse totalRatingsString", e);
}

return (
  <div className="relative">
    {badges.length > 0 && (
      <div className="absolute top-0 left-2 z-10 p-2 flex flex-col items-start gap-2">
        {badges.map((badge) => (
          <span key={badge.text} className={`text-white text-xs font-bold px-2 py-1 rounded-md shadow-lg ${badge.color}`}>
            {badge.text}
          </span>
        ))}
      </div>
    )}
    <WrappedComponent {...props} />
  </div>
);
};
};

export default withRestaurantBadges;

```

Usage

```

// src/components/Body.js
import { RestaurantCard } from './RestaurantCard';
import withRestaurantBadges from '../hocs/withRestaurantBadges';

const RestaurantCardWithBadges = withRestaurantBadges(RestaurantCard);

return (
  <RestaurantCardWithBadges resData={restaurant} />
);

```

2 withFilteredRestaurants HOC

What It Does

Enhances a component that receives a restaurant list by **filtering and sorting it** based on given config.

Functionality

☒ Filtering:

Filter	Description
<code>filterOpen</code>	Only includes restaurants that are currently open
<code>withDiscounts</code>	Only includes those with active <code>aggregatedDiscountInfoV3</code>

Sorting:

Supports sorting by:

- ★ `avgRating` (descending = best-rated first)
- ⌚ `deliveryTime` (ascending = fastest first)
- 💰 `costForTwo` (ascending = cheapest first)

Implementation

```
// src/hocs/withFilteredRestaurants.js
import React from 'react';

const withFilteredRestaurants = (config) => (WrappedComponent) => {
  return (props) => {
    const { restaurants } = props;
    let filteredList = [...restaurants];

    // --- Filtering ---
    if (config.filterOpen) {
      filteredList = filteredList.filter(res => res.info.isOpen);
    }
    if (config.withDiscounts) {
      filteredList = filteredList.filter(res =>
res.info.aggregatedDiscountInfoV3);
    }

    // --- Sorting ---
    if (config.sortBy) {
      const { key, ascending } = config.sortBy;
      filteredList.sort((a, b) => {
        let valA, valB;
        if (key === 'deliveryTime') {
          valA = a.info.sla?.[key];
          valB = b.info.sla?.[key];
        } else if (key === 'costForTwo') {
          valA = parseInt(a.info?.[key]?.match(/\d+/g));
          valB = parseInt(b.info?.[key]?.match(/\d+/g));
        } else {
          valA = a.info?.[key];
          valB = b.info?.[key];
        }

        if (valA < valB) return ascending ? -1 : 1;
        if (valA > valB) return ascending ? 1 : -1;
      });
    }

    return <WrappedComponent {...props} />;
  };
};
```

```
        return 0;
    });
}

return <WrappedComponent {...props} restaurants={filteredList} />;
};

export default withFilteredRestaurants;
```

Usage

```
// src/components/Body.js
import withFilteredRestaurants from '../hocs/withFilteredRestaurants';
import RestaurantList from './RestaurantList';

const FeaturedRestaurantList = withFilteredRestaurants({
  withDiscounts: true,
  filterOpen: true,
  sortBy: { key: 'avgRating', ascending: false },
})(RestaurantList);

return (
  <div className="featured-section">
    <h2 className="text-xl font-bold mb-4">🍷 Restaurants With Great Discounts!
  </h2>
    <FeaturedRestaurantList restaurants={listOfRestaurants} />
  </div>
);
```

Bonus Tips

- HOCs can be **composed** like:

```
const Enhanced = withA(withB(withC(Component)));
```

- Prefer **composition over inheritance** – a core React philosophy.
- Avoid stateful logic in HOCs (use hooks + context for state-sharing).


Summary


HOC Name	Purpose	Use Case Example
<code>withRestaurantBadges</code>	Add visual badges to cards	Highlight "Top Rated", "Popular", etc.


HOC Name	Purpose	Use Case Example
<code>withFilteredRestaurants</code>	Filter and sort restaurants	Show fast delivery, discount-only listings

recompose – Higher-Order Component Helpers for Scalable React Apps

What is **recompose**?

 **recompose** is a **React utility library** that provides a toolkit of **ready-to-use Higher-Order Components (HOCs)** and **functional composition helpers**, allowing you to:


- Reuse logic elegantly
- Compose HOCs cleanly
- Avoid deeply nested class components
- Write cleaner, declarative code 

Think of it like **Lodash for React components** 

Why Use **recompose**?

In larger React apps, you often find yourself repeating logic like:

- Conditional rendering
- State management
- Lifecycle methods
- Prop mapping or enhancement
- Performance optimization (e.g., `shouldComponentUpdate`)

Instead of writing repetitive boilerplate, **recompose** helps you compose functionality like building blocks  using pure functions.

Key Features

Feature	HOC / Utility	Description
Add local state	<code>withState</code>	Injects state & setter like <code>useState</code>
Add handlers (logic)	<code>withHandlers</code>	Adds handler props (pure functions)
Add lifecycle methods	<code>lifecycle</code>	Access to <code>componentDidMount</code> , etc.
Conditionally render	<code>branch</code> , <code>renderIf</code>	Render different components based on props
Optimize rendering	<code>pure</code> , <code>onlyUpdateForKeys</code>	Prevent unnecessary re-renders
Transform props	<code>mapProps</code> , <code>withProps</code>	Modify or inject props

Feature	HOC / Utility	Description
Compose multiple HOCs	<code>compose()</code>	Cleanly combine multiple HOCs in one call

Installation

```
npm install recompose
# or
yarn add recompose
```

Example Use Cases

1. Add State to a Functional Component

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

const enhance = useState('count', 'setCount', 0);

const Counter = ({ count, setCount }) => (
  <div>
    <p>Count: {count}</p>
    <button onClick={() => setCount(n => n + 1)}>Increment</button>
  </div>
);

export default enhance(Counter);
```

2. Add Custom Handlers (like methods)

```
import { withHandlers } from 'recompose';

const enhance = withHandlers({
  onClick: props => () => {
    alert(`Hello ${props.name}!`);
  },
});

const HelloButton = ({ onClick }) => (
  <button onClick={onClick}>Say Hello</button>
);

export default enhance(HelloButton);
```

3. Lifecycle in Functional Component 🧬

```
import { lifecycle } from 'recompose';

const enhance = lifecycle({
  componentDidMount() {
    console.log('Component mounted 🧬');
  },
});

const Dummy = () => <div>I'm alive!</div>;

export default enhance(Dummy);
```

4. Combine HOCs Using `compose`

```
import { compose, withProps, withState } from 'recompose';

const enhance = compose(
  withProps({ name: 'Darshan' }),
  withState('count', 'setCount', 0)
);

const MyComponent = ({ name, count, setCount }) => (
  <div>
    <h2>Hello {name}</h2>
    <p>Count: {count}</p>
    <button onClick={() => setCount(count + 1)}>+1</button>
  </div>
);

export default enhance(MyComponent);
```

💧 Power Combo

Combine `branch`, `renderComponent`, and `compose` to render a loading spinner:

```
import { branch, renderComponent, compose } from 'recompose';

const Loading = () => <div>Loading...</div>;

const enhance = compose(
  branch(
    props => props.isLoading,
    renderComponent(Loading)
  )
);
```



```
);  
  
const DataComponent = ({ data }) => <div>Data: {data}</div>;  
  
export default enhance(DataComponent);
```

⊘ When *Not* to Use

- You’re using React Hooks: Hooks now handle most of this logic natively.
- New projects: **recompose** is no longer actively maintained.
- You need class-based state logic: consider **useState**, **useEffect**, etc.

☑ **Best for:**

- Legacy projects (React <16.8)
- Component libraries with functional HOC pipelines
- Clean abstraction of logic in very large codebases

🔍 Summary

☑ Pros	⚠ Cons
Clean, declarative HOC composition	No longer maintained officially
Abstracts repetitive logic	Not needed with React Hooks
Great for large/legacy React apps	Adds minor learning curve

📄 Resources

- 🔗 [GitHub Repo](#)
- 📖 [Recompose Docs](#)
- 👤 [Dan Abramov's Thinking on HOCs](#)