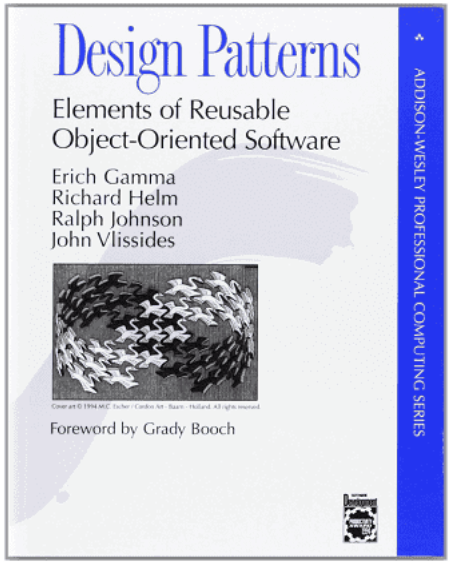


Design Patterns

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software **design**

Reusability & Separation of Concerns.

- **The DRY principle – Don't Repeat Yourself.**
- **Techniques to improve DRY(ness) (increase reusability):**
  1. **Inheritance** (is-a **relationships, e.g. Car is an automabile)**
  2. **Composition** (has-a **relationships, e.g. Car has an Engine)**

- **React favors composition.**
- **Core React composition Patterns:**
  1. **Containers.**
  2. **Render Props.**
  3. **Higher Order Components.**

Composition - Children

- **HTML is composable**

```
<div>
    <h2>Some Heading</h2>
    <ul>
        <li> . . . . . </li>
        <li> . . . . . </li>
        <li> . . . . . </li>
    </ul>
</div>
```

```
<div>
    <p>…….</p>
   <img ……. />
  <a href ……./>
<</div>
```

** has three children.**

- ** has two children; has three children**

The Container pattern.

All React components have a special <u>children</u> prop so that consumers can pass components directly by nesting them inside the jsx.

```
const Picture = (props) => {
  return (
    <div>
      <img src={props.src}/>
      {props.children}
    </div>
  )
}
```

```
const OtherComponent = props => {
  . . . . . .
  return (
    <div className='container'>
      <Picture src={picture.src}>
          // what is placed here is
          // passed as props.children
      </Picture>
    </div>
  )
}
```

- **When the** Picture **component renders, its** props.children **will <u>display what the consumer places</u> between the opening and closing tags of Picture.**
- **This <u>de-couples</u> the** Picture **component from its content and makes it** reusable**.**

```
const OtherComponent1 = props => {
  . . . . . .
  return (
    <div className='container'>
      <Picture src={picture.src}>
          <button>.......</button>
      </Picture>
    </div>
  )
}
```

```
const OtherComponent2 = props => {
  . . . . . .
  return (
    <div className='container'>
      <Picture src={picture.src}>
          <ul>. . . . . </ul>
      </Picture>
    </div>
  )
}
```

```
const OtherComponent3 = props => {
  . . . . . .
  return (
    <div className='container'>
      <Picture src={picture.src}>
          <ComplexComponewnt>
            . . . . . .
          </ComplexComponewnt>
      </Picture>
    </div>
  )
```

Image

Button

Image

List

Image

Complex Component

Picture is **composed** with other elements / components

The Render Prop pattern

- **Use the pattern to share logic between components.**
- **Dfn**.: A render prop is a function prop that a component uses to know what to render.

```
const SharedComponent = (props) => {
  ..........
  return (
    <div className="classX"
            onMouseOver={funcY} >
      { props.render() }
    </div>
  );
};
```

- SharedCoomponent **receives its render logic from the consumer, i.e.** SayHello.
- Prop name is arbitrary.

```
const SayHello = (props) => {
  ..........
  return (
    ........
    <SharedComponent render={() =>
      <span>Say Hello</span>
    } />
    ............
  );
};
```

```
<div className="classX"
        onMouseOver={funcY} >
  <span>Say Hello</span>
</div>
```

The Render Prop - Sample App.

Friends List

Search

- **Jeff Herrera**

- **Michele Denis**

- **Annefleur Hop**

- **Brayden Rice**

Friends List

Search

- **Önal Kılıççı**

  onal.kilicci@example.com

- **Thomas Chen**

  thomas.chen@example.com

- **Magda Vieira**

  magda.vieira@example.com

- **Vilma Heikkila**

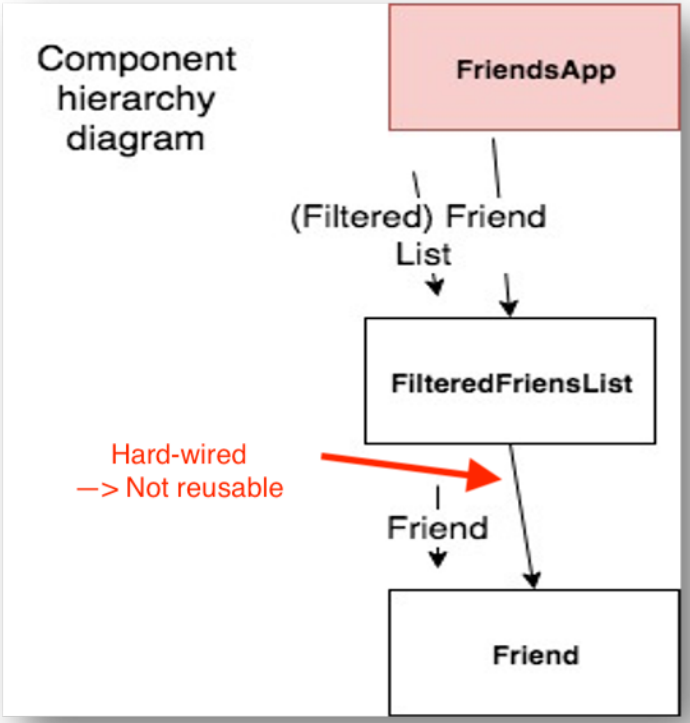
  vilma.heikkila@example.com

- **Herman Hessen**

  herman.hessen@example.com

- **Sandra Bell**

The Render Props - Sample App.



- **Solution: As well as passing the list of matching friends to**
- **, we also tell it <u>how to render a friend</u>**
- **Use a prop to communicate the 'how', i.e. a render prop**

```
<FilteredFriendList
  list={filteredList}
  render={(friend) => <FriendImage friend={friend} />}
/>
```

```
1   import React from "react";
2   You, 5 days ago • Initial structure
3   const FilteredFriendList = props => {
4     // console.log('Render of FilteredFriendList')
5     const friends = props.list.map(item => (
6       props.render(item)
7     ));
8     return <ul>{friends}</ul>;
9   };
10
11  export default FilteredFriendList;
12
```

```
<FilteredFriendList
  list={filteredList}
  render={(friend) => <FriendContact friend={friend} />}
/>
```

- FilteredFriendList **is no longer statically importing the component for rendering a friend.**
- **It receives this via the render prop.**
- **The friends array elements will be** Friend **components, e.g.** FriendContact, FriendImage

- **Without this pattern we would need a** FilteredFriendList **component for each use case, thus violating the DRY principle.**

- **The prop name is arbitrary;** render **is a convention.**

Reusability.

- **Core React composition Patterns:**
  1. **Containers**
  2. **Render Props**
  3. **Higher Order Components.**

- **HOC is a** function **that takes a component and returns an enhanced version of it.**

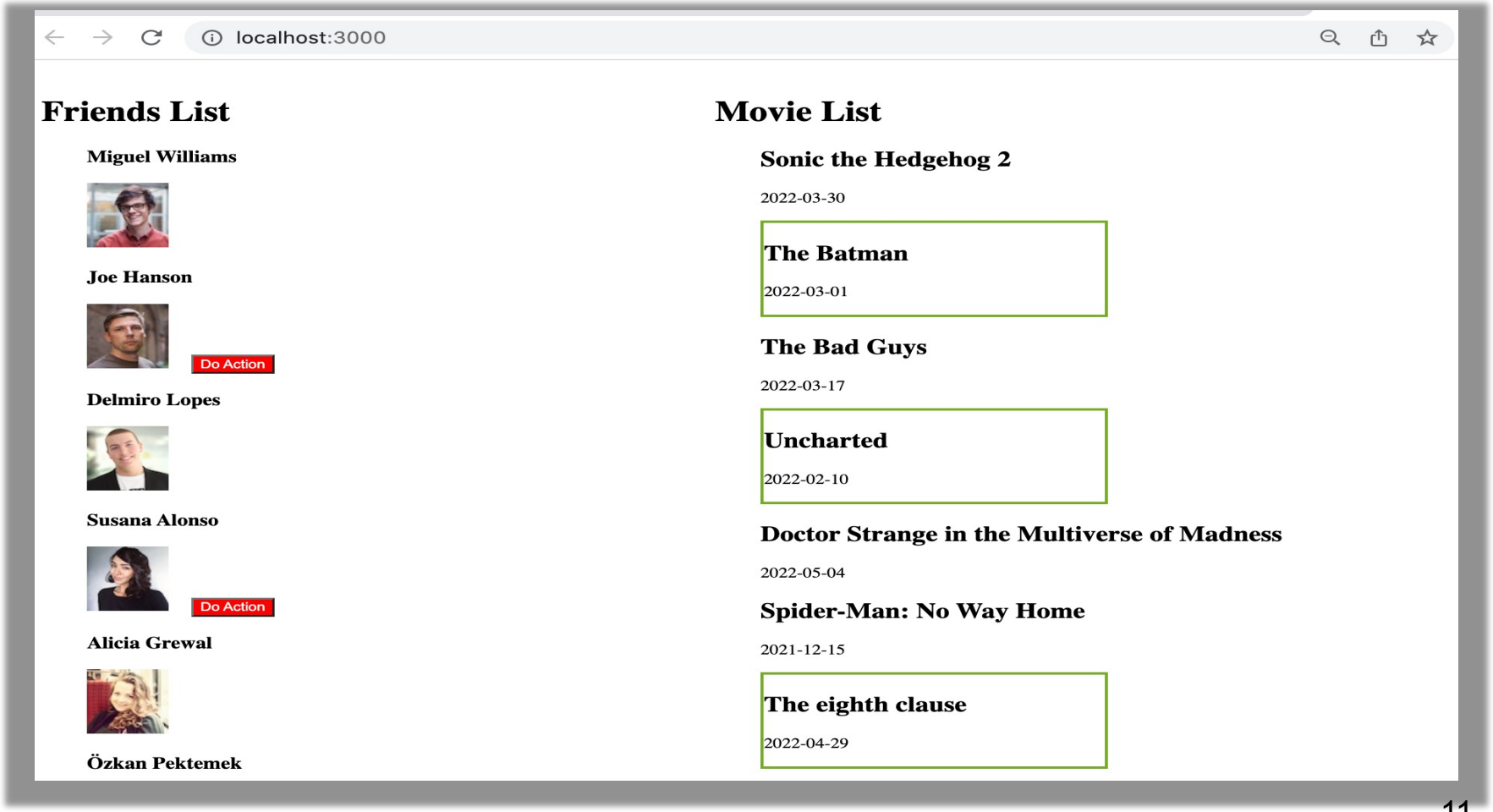
  function(Input Component) → Output component

  – **Enhancements could include:**
    - **Statefulness.**
    - **Props.**
    - **UI.**

- **Naming convention:** <u>with</u>XXXXXXXX**( )**

Higher Order Component - Sample App.

- **Objective: Make different types of components** *clickable*.



Higher Order Component.

- **Template for a HOC:**

```
const withSomething = (InputComponent) => {
    return (
      (props) => {
          ….. additional behaviour ……..
          return (
              ….. additional JSX ….
               <InputComponent {...props} additionalProp={…}/>
               …… additional JSX ……
          );
      };   // end output component
    )
}. // end HOC
```

Higher Order Component - Sample App.

```
const withClickable = (Component) => {
  return ({ ...props }) => {
    const [clicked, setClicked] = useState(false);
    const onClick = () => setClicked(!clicked);
    return (
      <div onClick={onClick}>
        <Component {...props} clicked={clicked} />
      </div>
    );
  };
};

export default withClickable;
```

This HOC enhances the input component by:

1. Adding a state variable, controlled by an **onClick event handler.**

2. Passing the state value as a prop

3. Wrapping it in a div with properties.

Higher Order Component - Sample App.

```
4   const Friend = (props) => {
5
6     return (
7       <li>
8         <h3>{` ${props.friend.name.first} ${props.friend.name.last}`}</h3>
9         <img src={props.friend.picture.medium} />
10        {props.clicked ? <button>Do Action</button> : <></>}      You, 4 ho
11      </li>
12    );
13  };
14
15  export default withClicker(Friend);
```

The input component leverages the enhancements provided by a higher function via the additional props it offers.

```
<ClickableFriend  friend={friend} />
```

The app uses the enhanced component, not the input component

Summary.

- **Objectives – Reusability, Separation of Concerns (Single Responsibility), DRY.**
- **Benefits - Maintainability, Understandability, Extendability, Adaptability.**
- **Approach – Apply design patterns.**

- **React App.**
  - **Composition.**
  - **Patterns – Container, Render Prop, Higher Order Component.**

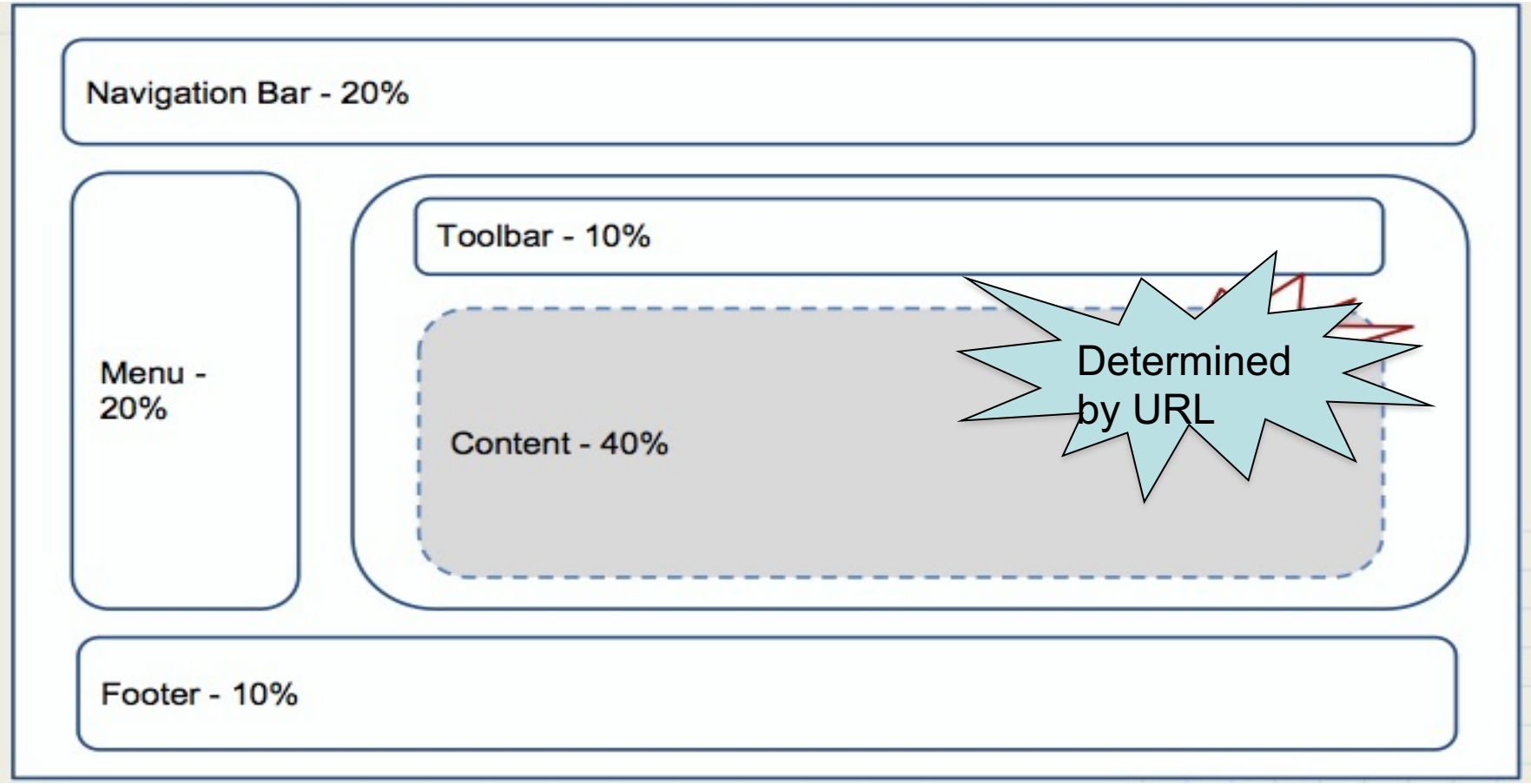
- **(More on patterns later.)**

Navigation

(Continued)

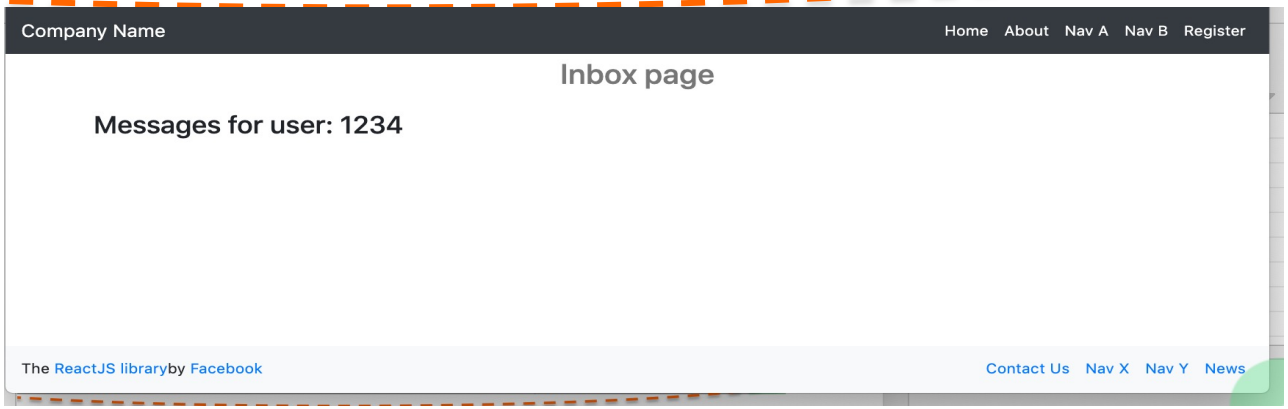
(See Archive from earlier lecture for code samples.)

Typical Web app layout



Persistent elements/components

- **Use cases: Site-wide Headers. Footers. Side menus.**



Persistent elements/components

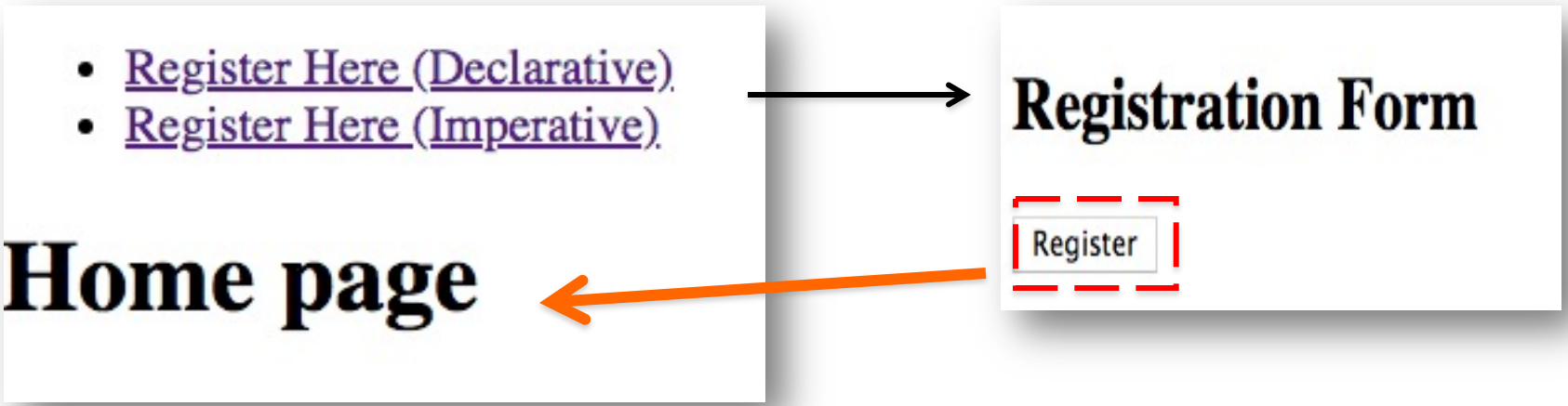
- **Ref.** src/sample6

```
class Router extends Component {
    render() {
        return (
            <BrowserRouter>
              <div>
                <Header/>
                <div className="container">
                    <Switch>
                        <Route path='/about' component={ About } />
                        <Route path='/register' component={ Register } />
                        <Route path='/contact' component={ Contact } />
                        <Route path='/inbox/:userId' component={ Inbox } />
                        <Route exact path='/' component={ Home } />
                        <Redirect from='*' to='/' />
                    </Switch>
                </div>
                <Footer />
              </div>
            </BrowserRouter>
        )
    }
}
```

Programmatic Navigation.

- **Performing navigation in JavaScript.**
- **Two options:**
  1. **Declarative – requires state; use** <Navigate /> element.
  2. **Imperative – use the hook**

- **EX.: See** /src/sample7/**.**

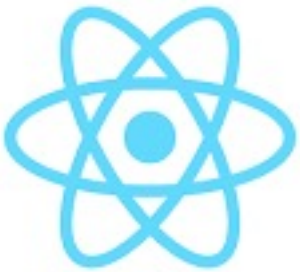


Summary

- **React Router package adheres to React principles:**
  - **Declarative.**
  - **Component composition.**
  - **The event → event handler → re-render**

- **Package's main components - <BrowserRouter>, <Route>, <Navigate>, <Link>.**

- **Package hooks – useParams, useNavigate, useLocation.**



Custom Hooks

Custom Hooks.

- **Custom Hooks let you extract component logic into** reusable **functions.**
- **Improves code readability and modularity.**

Example:

```
const BookPage = props => {
  const isbm = props.isbn;
  const [book, setBook] = useState(null);
  useEffect(() => {
    fetch(
    `https://api.for.books?isbn=${isbn}`)
    .then(res => res.json())
    .then(book => {
      setBook(book);
    });
  }, [isbn]);
  . . . .rest of component code . . . .
}
```

Objective – Extract the book-related state code into a custom hook.

Custom Hook Example.

Solution:

```
const useBook = isbn => {
  const [book, setBook] = useState(null);
  useEffect(() => {
     fetch(
    `https://api.for.books?isbn=${isbn}`)
    .then(res => res.json())
    .then(book => {
      setBook(book);
    });
  }, [isbn]);
   return [book, setBook];
};
```

```
const BookPage = props => {
  const isbm = props.isbn;
  const [book, setBook] = useBook(isbn);

  . . . .rest of component code . . . .
}
```

- **Custom Hook is an ordinary function BUT can only be called from a React component function.**
- **Prefix hook function name with** use **to leverage linting support.**
- **Function can return any collection type (array, object), with any number of entries.**