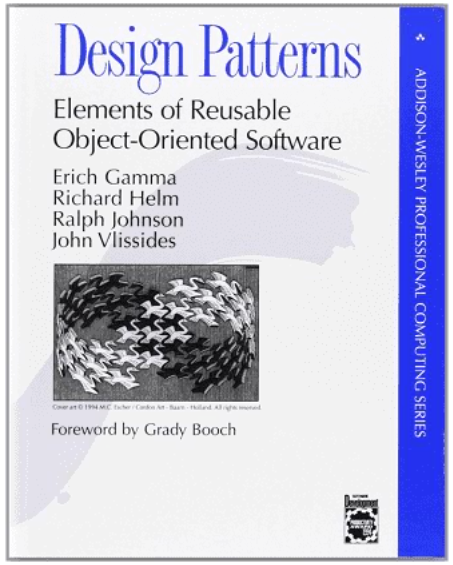


Design Patterns

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 favours composition.**
- **Core React composition Patterns:**
  1. **Container.**
  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>
```

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

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

** has three children.**

The Container pattern.

All React components have a special children prop. It allows a consumer (container) to pass other components to it by nesting them inside the tsx.

index.tsx

```tsx
const App: React.FC = () => {
  return (
    <DemoComponent>
      <p>This is a child element.</p>
    </DemoComponent>
  );
};
```

demoComponent.tsc.tsx

```tsx
const DemoComponent: React.FC<DemoComponentProps> = ({ children }) => {
  return (
    <div>
      <h1>Full Stack 2</h1>
      {children}
    </div>
  );
};
```

- **See: sampleChildren.tsx in routingSamples**
- **The container determines what** DemoComponent **renders,**
- **This de-couples the** DemoComponent **from its content and makes it reusable.**

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

Image

Button

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

Image

List

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

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

Image

Complex Component

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 generate part of its rendered output.

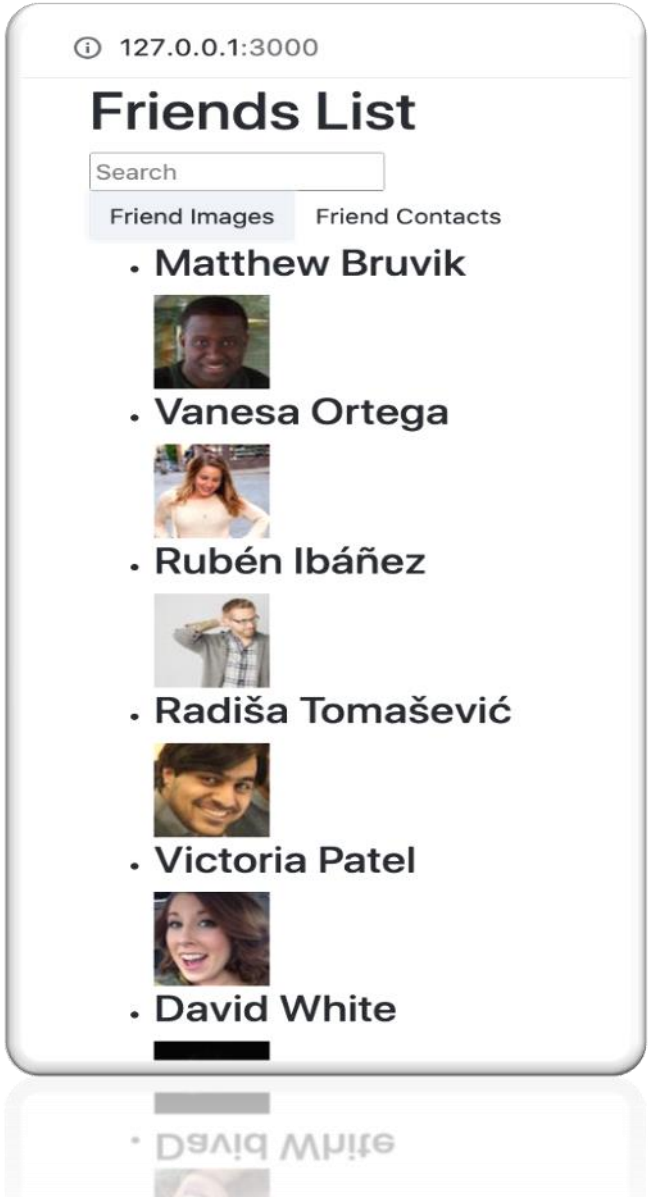
```
const SharedComponent = (props) => {
  ..........
  return (
    <div className="classX"
                onMouseOver={funcY} >
      { props.render() }
    </div>
  );
};
```

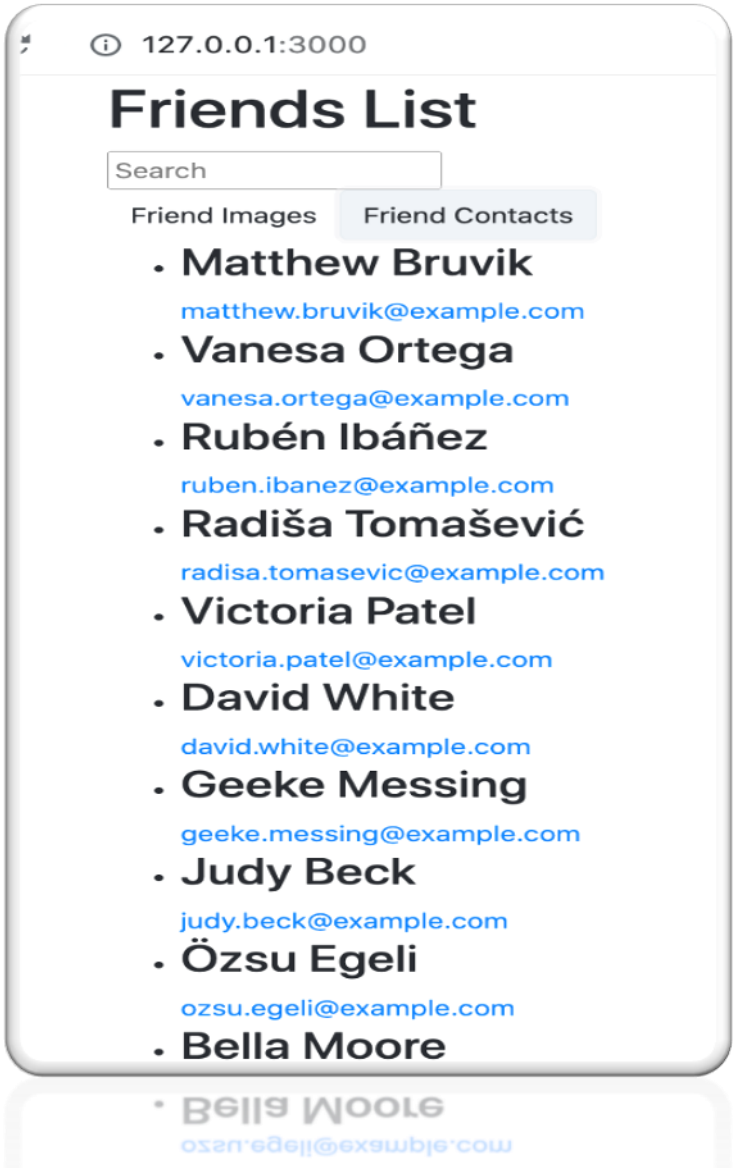
- SharedComponent **receives its render logic from the consumer, i.e.** SayHello.
- Prop name is arbitrary.

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

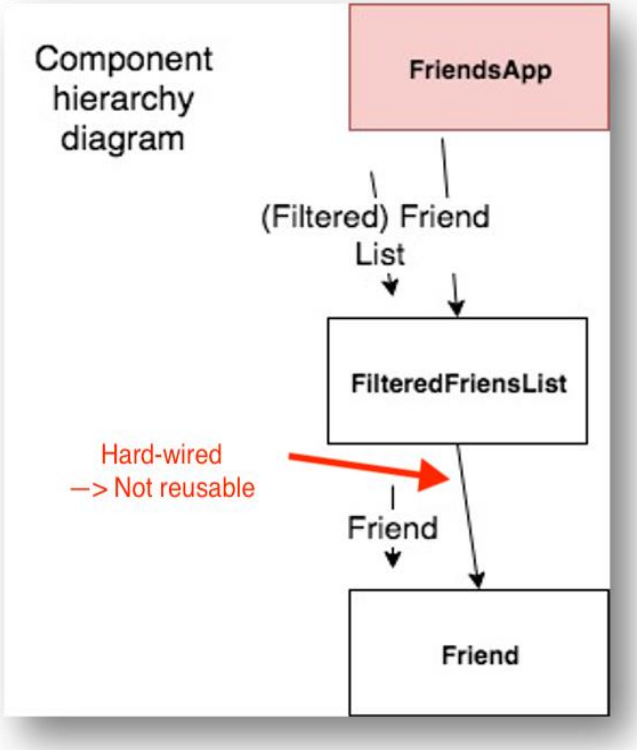
```
<div className="classX"
          onMouseOver={funcY} >
  <span>Say Hello</span>
</div>
```

The Render Prop - Sample App.





The Render Props - Sample App.



- **Updates to design:**

1. FriendsApp **passes a render-prop to** FilteredFriendList**, indicating** how **Friends should be rendered.**
2. **Remove static import of Friend component type from** FilteredFriendList**.**

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

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

```
const FilteredFriendList:React.FC<FilteredFriendListProps>  = ({list, render}) => {
  const friends = list.map((item) =>
    render(item)
  );
  return <ul>{friends}</ul>;
};
```

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

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

Custom Hooks

Custom Hooks.

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

Example:

```
const BookPage: React.FC<BookPageProps> = ({ isbn }) => {
  const [book, setBook] = useState<any>(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: **useBook.tsx**

```tsx
const useBook = (isbn: string) => {
  const [book, setBook] = useState<Book | null>(null);

  useEffect(() => {
    if (isbn) {
      fetch(`https://api.for.books?isbn=${isbn}`)
        .then(res => res.json())
        .then((data: Book) => {
          setBook(data);
        });
    }
  }, [isbn]);

  return book;
};
```

```tsx
const BookPage: React.FC<BookPageProps> = ({ isbn }) => {
  const book = useBook(isbn);

  return (
    <div>
      {/* Render book details here */}
      {book ? (
        <div>
          <h1>{book.title}</h1>
          <p>{book.author}</p>
          {/* Add more book details as needed */}
```

- **Custom Hook is an ordinary function BUT should 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.**