

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 automobile)
 2. **Composition** (has-a relationships, e.g. Car has an Engine)
- **React favors 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>
```

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

<div> has three children.

- **<div> has two children; 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.

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



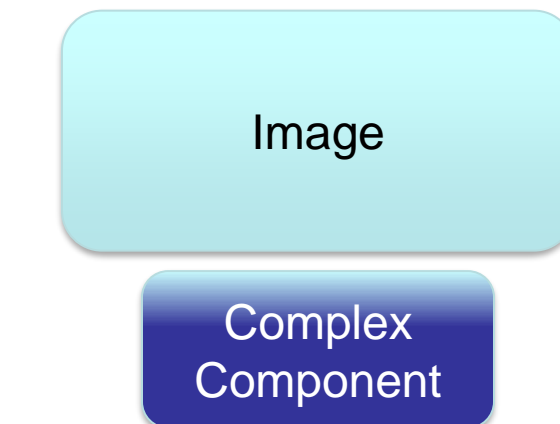
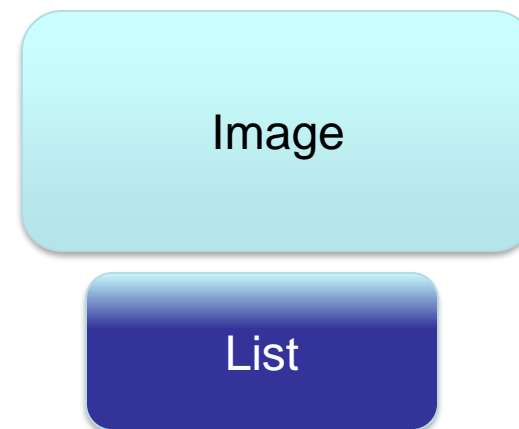
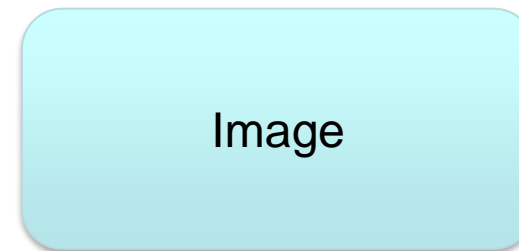
```
3  const Container = () => {  
4    // .. code ...  
5    return (  
6      <div className="container" >  
7        <Picture src={anImageRef}>  
8          // JSX here is bound to  
9          // props.children of Picture  
10       </Picture>  
11     </div>  
12   );  
13 };
```

- The container determines what Picture renders,
- This de-couples 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}>
        <ComplexComponent>
          . . . . .
        </ComplexComponent>
      </Picture>
    </div>
  )
}
```



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 generate part of its rendered output.

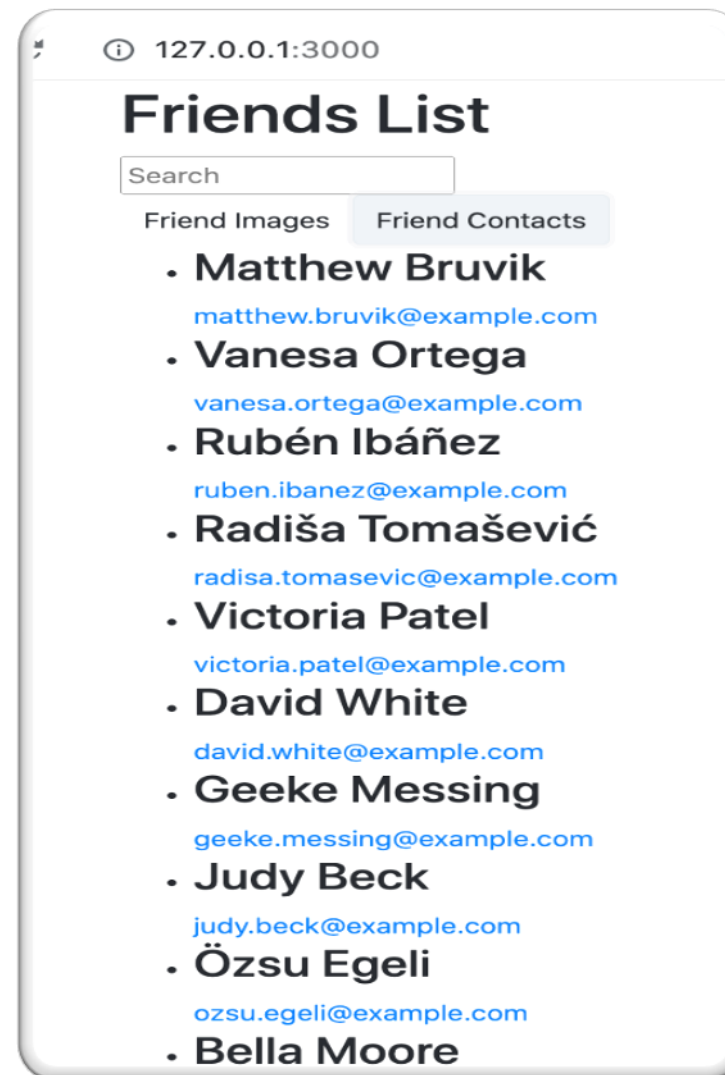
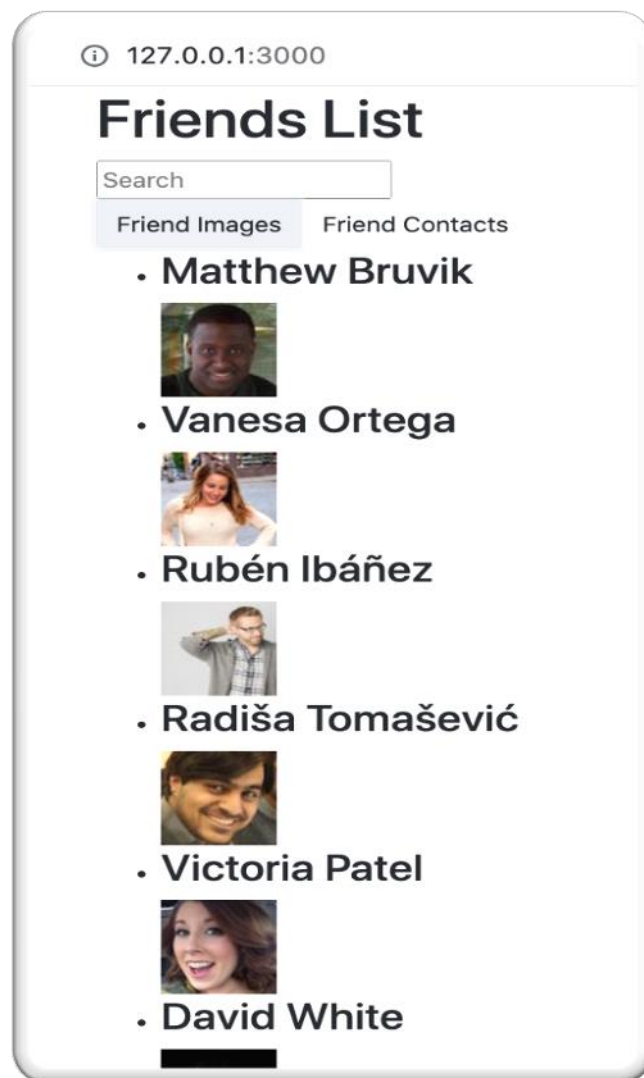
```
const SharedComponent = (props) => {  
  .....  
  return (  
    <div className="classX"  
      onMouseOver={funcY}>  
      { props.render() }  
    </div>  
  );  
};
```

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

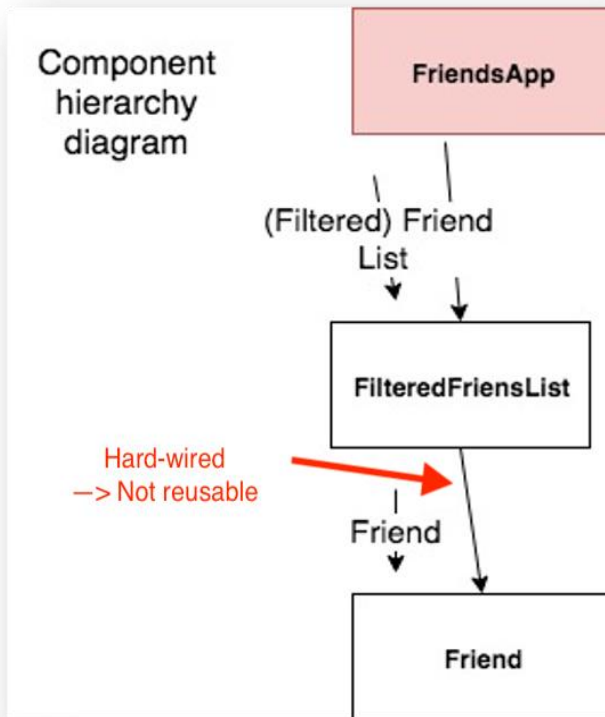
```
const SayHello = (props) => {  
  .....  
  return (  
    <SharedComponent render={() => {  
      <span>Say Hello</span>  
    }} />  
  );  
};
```

```
<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.**

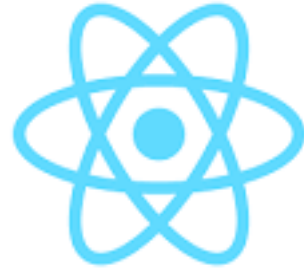

```
<FilteredFriendList  
  list={filteredList}  
  render={(friend) => <FriendImage friend={friend} />}  
/>
```

```
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} />}  
/>
```

- 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.



Custom Hooks

Custom Hooks.

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

Example:

```
const BookPage = props => {  
  const isbn = 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 isbn = props.isbn;  
  const [book, setBook] = useBook(isbn);  
  
  . . . rest of component code . . .  
}
```

- 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.

