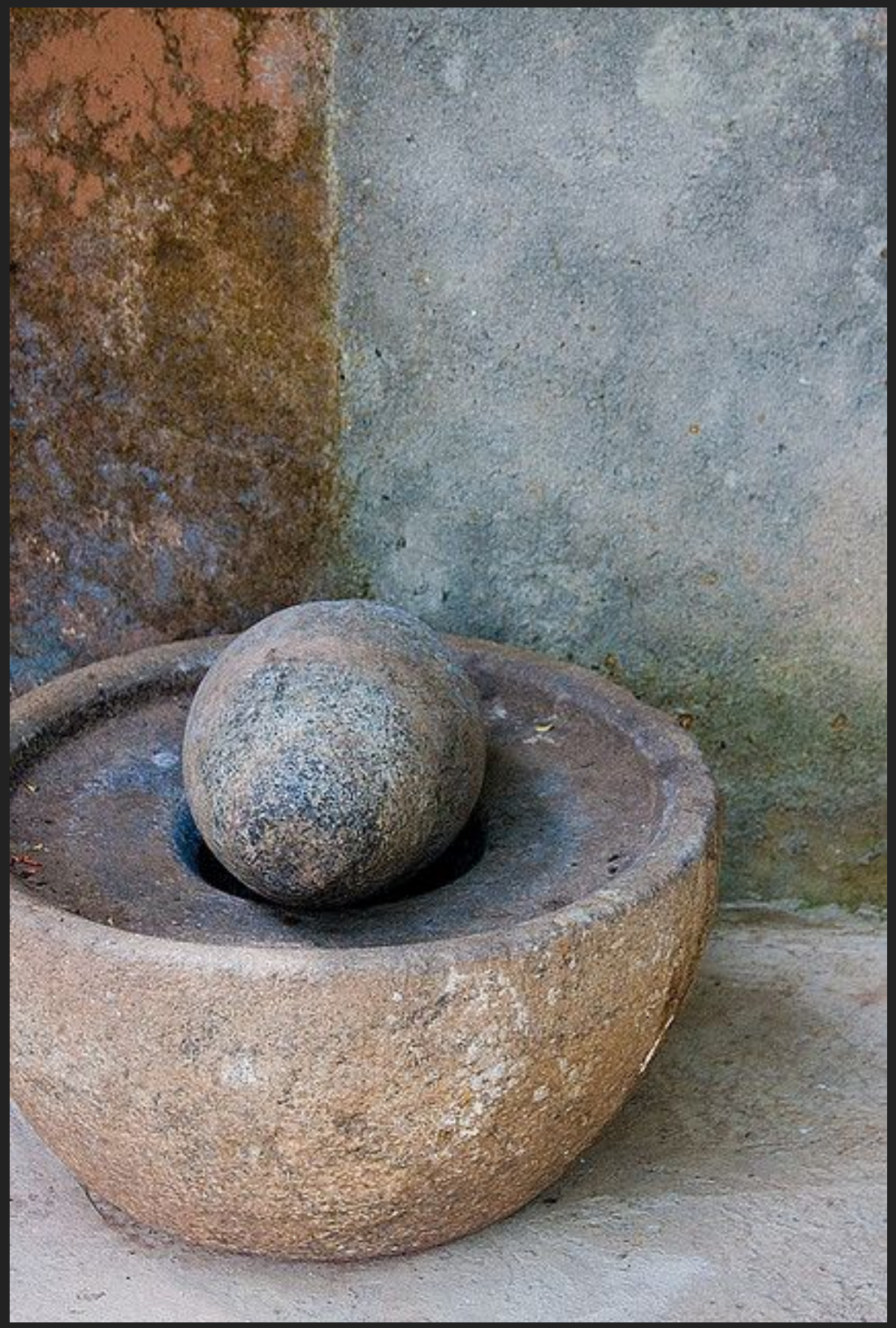
PRINCIPLES OF DEVELOPING

REACT APPLICATIONS

CAPABILITIES OF A FRONT-END APPLICATION

- Take control of screen space (real estate)
  - HTML/CSS
- Handle user interaction
  - Javascript
- Interaction with server - HTTP(S)
- State Management
- Handle modifications to DOM
- Can’t we do all the above using just HTML/CSS/JS ?

SEEN THIS?



Src: https://mx.pinterest.com/pin/521854675571762272/

REACT

- Was introduced in 2013
- Declarative UI
  - Developers ***describe*** the UI
  - Instead of ***imperative*** DOM manipulations
- Component based architecture
  - Components are self-contained, reusable, composable building blocks of UI
- JSX Syntax
  - A syntax that allowed developers to write HTML-like code within JavaScript
  - Blends UI and logic
- State management
  - Allows components to manage dynamic data internally
  - Support for state at various levels - component level or across many components or global
- Ecosystem and flexibility
  - React focuses on the “View” layer (in MVC), but integrates well with other libraries and frameworks
  - Compatibility with Typescript - easier to write type-safe and predictable code

JSX

- Javascript extended
  - https://react.dev/learn/writing-markup-with-jsx
- Describe what the UI should look like
- JSX produces React “elements”
- Its HTML code inside JS
- React transforms JSX to HTML and JS
  - Browsers do NOT understand JSX

REACT COMPONENT

- Components are self-contained, reusable, composable building blocks of UI
- Each component encapsulates its own logic, state, and rendering, promoting modularity and code reuse
- Components can be nested within other components to build complex interfaces
- Components are customisable by using Props
- Components promote separation of concerns
- The UI can be split into independent pieces. Each piece can be modelled as a component
- A react allocation is made up of a hierarchy (or tree) of components

CLASS COMPONENT

- React lets you define components as classes
- Must define the render() method
- It returns the markup

```
1  class Books extends React.Component {
2    render() {
3      return (
4        <ul>
5          <li>Eloquent Javascript</li>
6          <li>Javascript: Definitive guide</li>
7          <li>Javascript: The good parts</li>
8          <li>You don't know JS: Scope and closures</li>
9        </ul>
10     );
11   }
12 }
```

```
1  class ListBooks extends React.Component {
2    render() {
3      return (
4        <Books/>
5      )
6    }
7  }
```

FUNCTION COMPONENT

- React component is a JavaScript function that you can sprinkle with markup
- 1. Export component
- 2. Define function
- 3. Return markup
- 4. Use component

```
1  export default function Books() {
2    return (
3      <ul>
4        <li>Eloquent Javascript</li>
5        <li>Javascript: Definitive guide</li>
6        <li>Javascript: The good parts</li>
7        <li>You don't know JS: Scope and closures</li>
8      </ul>
9    )
10 }
```

```
1  export default function ListBooks() {
2    return (
3      <Books />
4    );
5  }
```

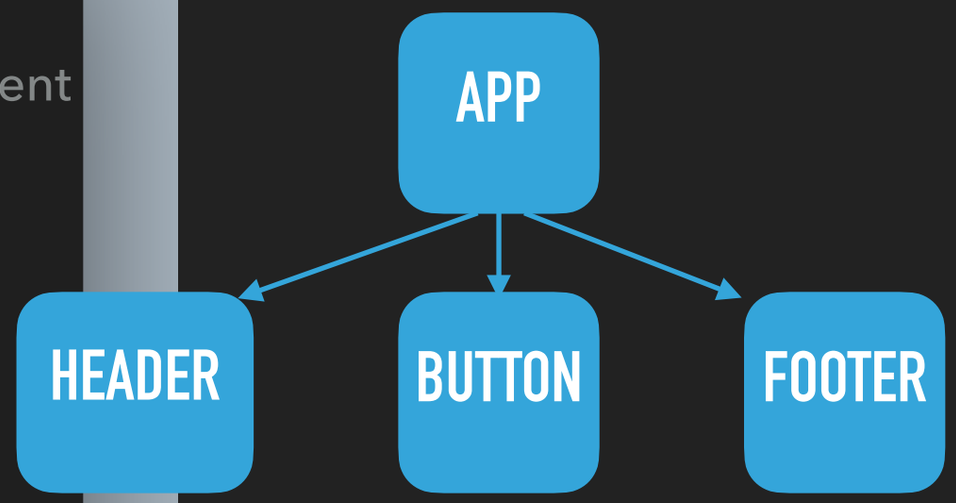
PROPS

- React components use props to communicate with each other
- Every parent component can pass some information to its child components
- Read-only
- Makes a component dynamic and customisable
- Any number of props
- Props can be of any type. Any js value can be passed
  - Objects, arrays, functions
- https://react.dev/learn/passing-props-to-a-component

EXAMPLE

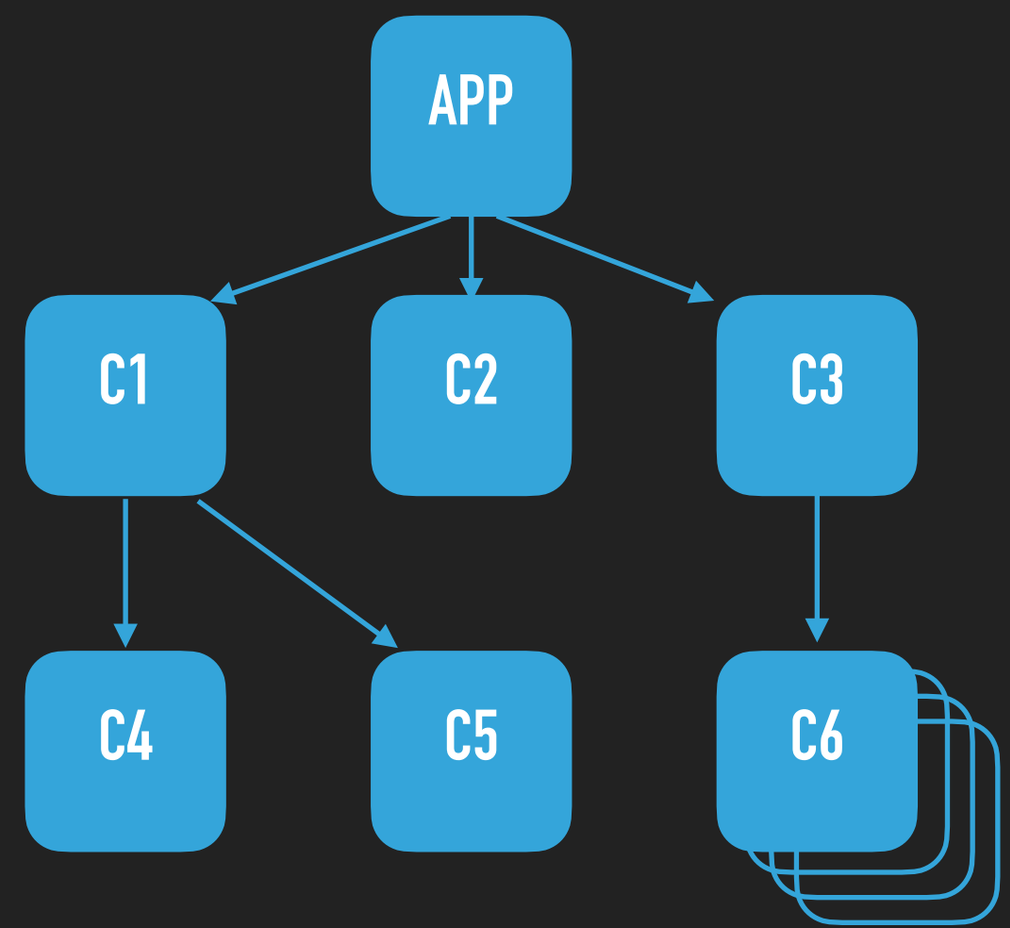
```
function Button({ label, onClick }) {
  return <button onClick={onClick}>{label}</button>;
}

function Header() {
  return <h1>Welcome to My App</h1>;
}

function Footer() {
  return <p>© 2024 My Company</p>;
}

function App() {
  return (
    <div>
      <Header />
      <Button label="Click Me" onClick={() => alert("Button clicked!")} />
      <Footer />
    </div>
  );
}
```

JSX element



COMPONENT TREE

- App Component is the root of the component tree
- C1 is instantiated only when its encountered in App's render()
- C1 is inserted in the DOM
- C1's render is not called again until:
  - Input props from App change
  - State of C1 changes



Same component, multiple instances

UNDERSTANDING STATE

- State is an object that holds some dynamic data
- State is mutable
- State is updated using specific methods (setState in this case)
  - Functional programming emphasizes immutable data structures
  - Ensures that the state is replaced, not modified in place
  - Immutability helps React efficiently compare previous and current states
- When state changes, the component re-renders

```
1  class Counter extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = { count: 0 };
5    }
6
7    increment = () => {
8      this.setState({ count: this.state.count + 1 });
9    };
10
11   render() {
12     return (
13       <div>
14         <p>Count: {this.state.count}</p>
15         <button onClick={this.increment}>Increment</button>
16       </div>
17     );
18   }
19 }
20
```

RE-RENDERING COMPONENTS

- A component can be rendered multiple times
- A component is re-rendered when
  - State of the component changes
  - Props change
  - Parent component is re-rendered
  - A few other cases
- What happens
  - Component redraws the UI with the changed state or props

STATE IN FUNCTIONAL COMPONENTS

- useState hook
- useState returns an array of two objects
- State object
- Function to change the state
- useState takes one param - initial value of state
- State object is managed by React. Component just uses it
- When state changes, the component re-renders

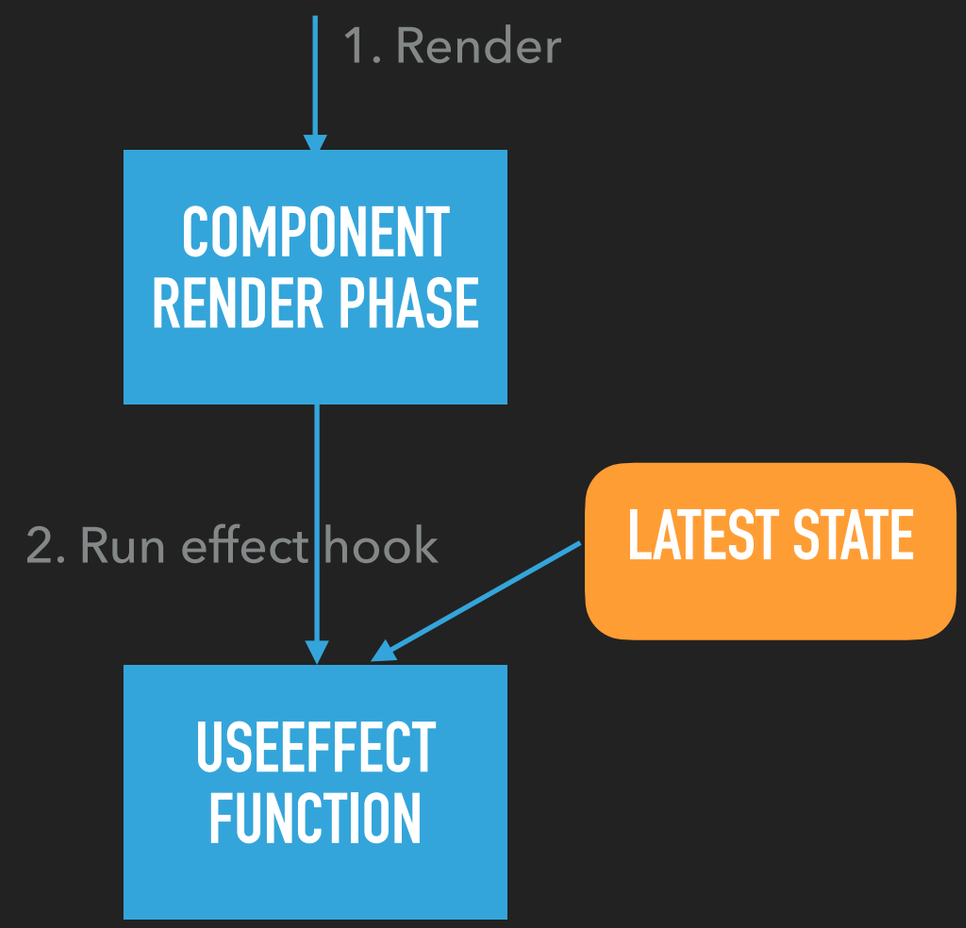
```
1  function Counter() {
2    const [count, setCount] = React.useState(0);
3
4    return (
5      <div>
6        <p>Count: {count}</p>
7        <button onClick={() => setCount(count + 1)}>Increment</button>
8      </div>
9    );
10 }
11
```

COMPONENTS – RENDER, STATE AND PROPS

- Component renders JSX
- JSX describes the UI
- The description of UI uses Props and state
- User interaction or other events changes state
- Component re-renders with the changed state

useEffect() HOOK

- Side effects - Mutations to state, timers, subscriptions, etc
- Side effects to happen outside of the render phase
- Use useEffect() hook instead of the function body.



useEffect SYNTAX AND USAGE

- useEffect(**()=>{}**, [dependencies])
- () => {} - Function that gets executed.
- Can cause side effects
- [dependencies] - Prop or state variables (or any variable)
- Empty dependency list [] makes the effect run only once

FUNCTION
() => {}

This function is executed after the render phase, **IF** the dependencies have changed.

ARRAY OF DEPENDENCIES

The effect function is run only if one or more dependencies change.

UNDERSTANDING USEEFFECT HOOK

- API calls are primarily made using useEffect
- Response to API call is stored in state
- When state changes, the component re-renders

```typescript
interface RemoteObject {
  id: string;
  name: string;
  data: any;
}

interface LocalObject {
  id: string;
  name: string;
  data: any;
  brand: string;
}
function Objects() {
  const [state, setState] = useState([] as LocalObject[]);

  useEffect(() => {
    const fetchObjects = async () => {
      const url = "https://api.restful-api.dev/objects";
      const response = await fetch(url);
      const remoteObjects: RemoteObject[] = await response.json();
      // From name, extract brand
      const localObjects: LocalObject[] = remoteObjects.map((remoteObject) => {
        let brand = remoteObject.name.replace(/ .*/, ''); // Extract first word
        return { ...remoteObject, brand };
      })
      setState(localObjects);
    }
    fetchObjects();
  }, [])

  console.log(state)
  return (
    <div>
      {state.map((object) => <p key={object.id}>{object.name} - {object.brand}</p>)}
    </div>
  )
}
```

ARCHITECTURE OF REACT APPS

SINGLE COMPONENT APPLICATION

```
<Component>
  Network requests
  State management
  Domain logic
  Render
</Component>
```

Container component

Ref: https://martinfowler.com/articles/modularizing-react-apps.html#ApartFromTheUserInterface

CONTAINER – PRESENTATIONAL COMPONENTS

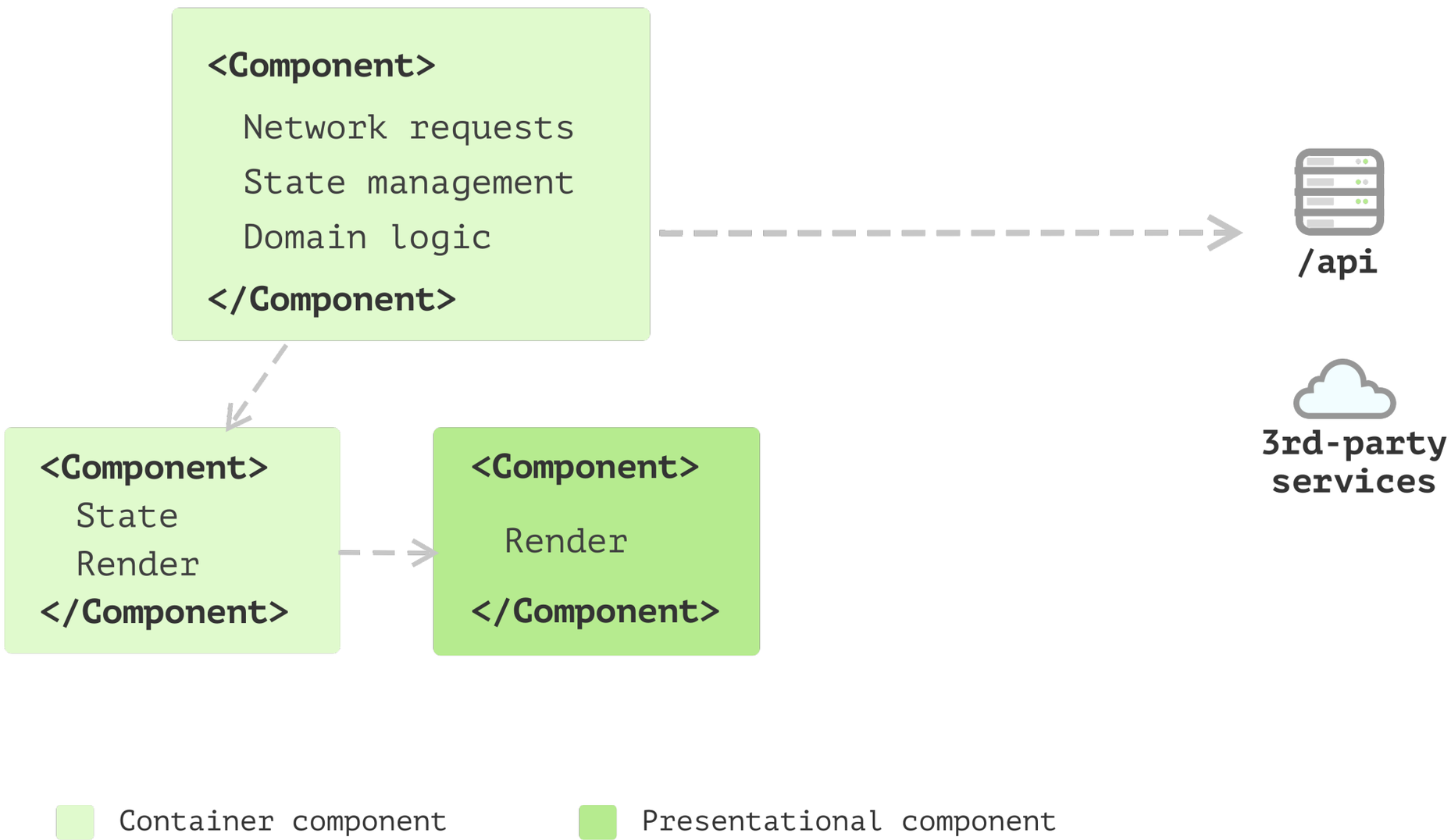
- Container Components
  - Concerned with how things work.
  - Manage state, handle logic, and interact with APIs or Redux stores.
  - Pass props to presentational components.
- Presentational Components
  - Concerned with how things look.
  - Focus on rendering UI based on props received.
  - Stateless and reusable.

CONTAINER – PRESENTATIONAL COMPONENTS

```
1  function UserListContainer() {
2    const [users, setUsers] = useState([]);
3    const [loading, setLoading] = useState(true);
4
5    useEffect(() => {
6      fetch('/api/users')
7        .then((res) => res.json())
8        .then((data) => {
9          setUsers(data);
10         setLoading(false);
11       });
12   }, []);
13
14   if (loading) return <p>Loading...</p>;
15
16   return <UserList users={users} />;
17 }
```

```
1  function UserList({ users }) {
2    return (
3      <ul>
4        {users.map((user) => (
5          <li key={user.id}>{user.name}</li>
6        ))}
7      </ul>
8    );
9  }
```

MULTIPLE COMPONENT APPLICATION



Ref: https://martinfowler.com/articles/modularizing-react-apps.html#ApartFromTheUserInterface

STATE MANAGEMENT USING HOOKS

- Apart from rendering, what are the other things the components are doing?
  - network requests
  - converting data into different shapes (types)
  - State management
- Can it be moved out?

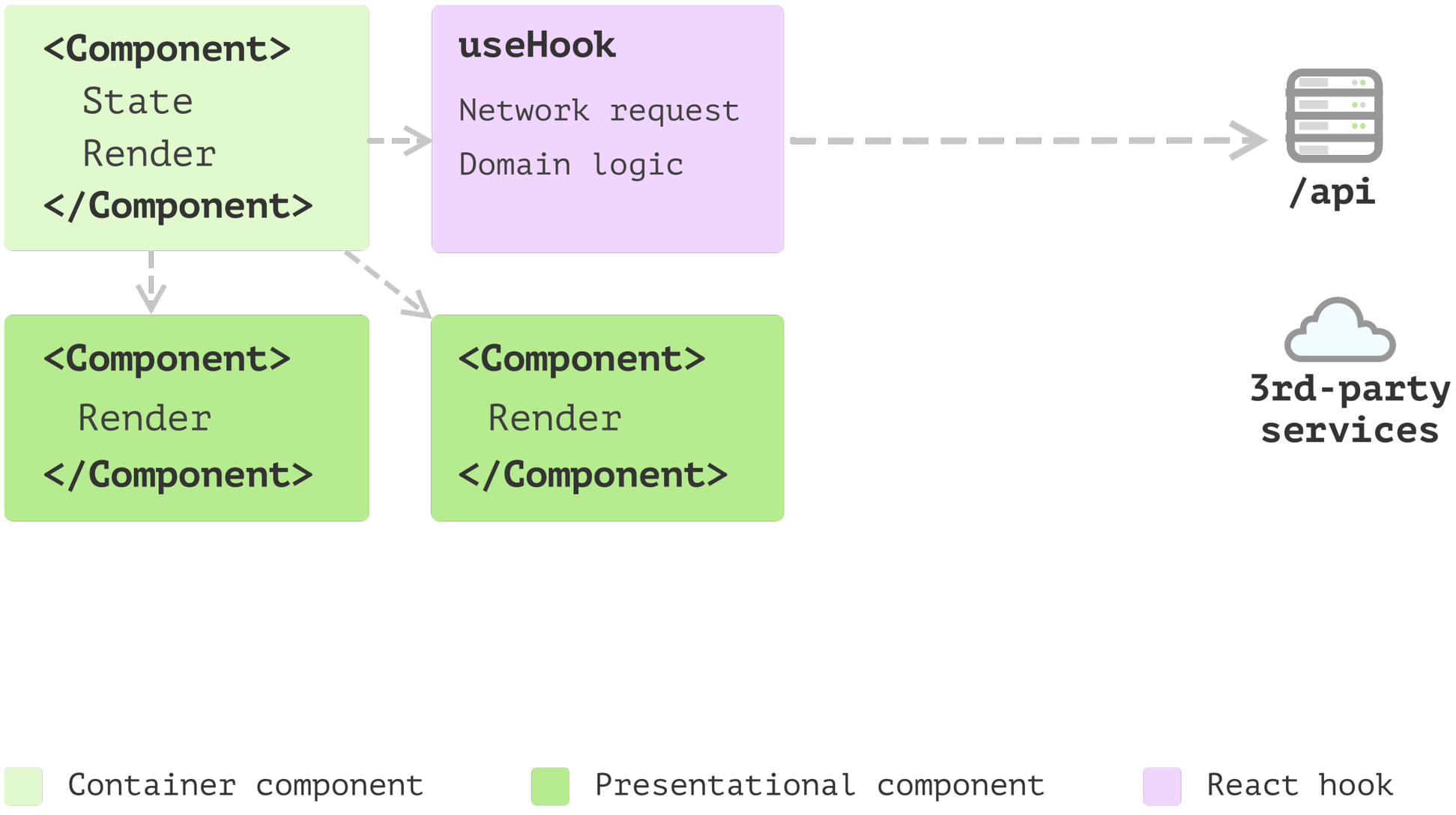
STATE MANAGEMENT USING HOOKS

```
1  function useUsers() {
2    const [users, setUsers] = useState([]);
3    const [loading, setLoading] = useState(true);
4
5    useEffect(() => {
6      fetch('/api/users')
7        .then((res) => res.json())
8        .then((data) => {
9          setUsers(data);
10         setLoading(false);
11       });
12   }, []);
13
14   return { users, loading };
15 }
```

```
1  function UserListContainer() {
2    const { users, loading } = useUsers();
3
4    if (loading) return <p>Loading...</p>;
5
6    return <UserList users={users} />;
7  }
```

```
1  function UserList({ users }) {
2    return (
3      <ul>
4        {users.map((user) => (
5          <li key={user.id}>{user.name}</li>
6        ))}
7      </ul>
8    );
9  }
```

STATE MANAGEMENT WITH HOOKS

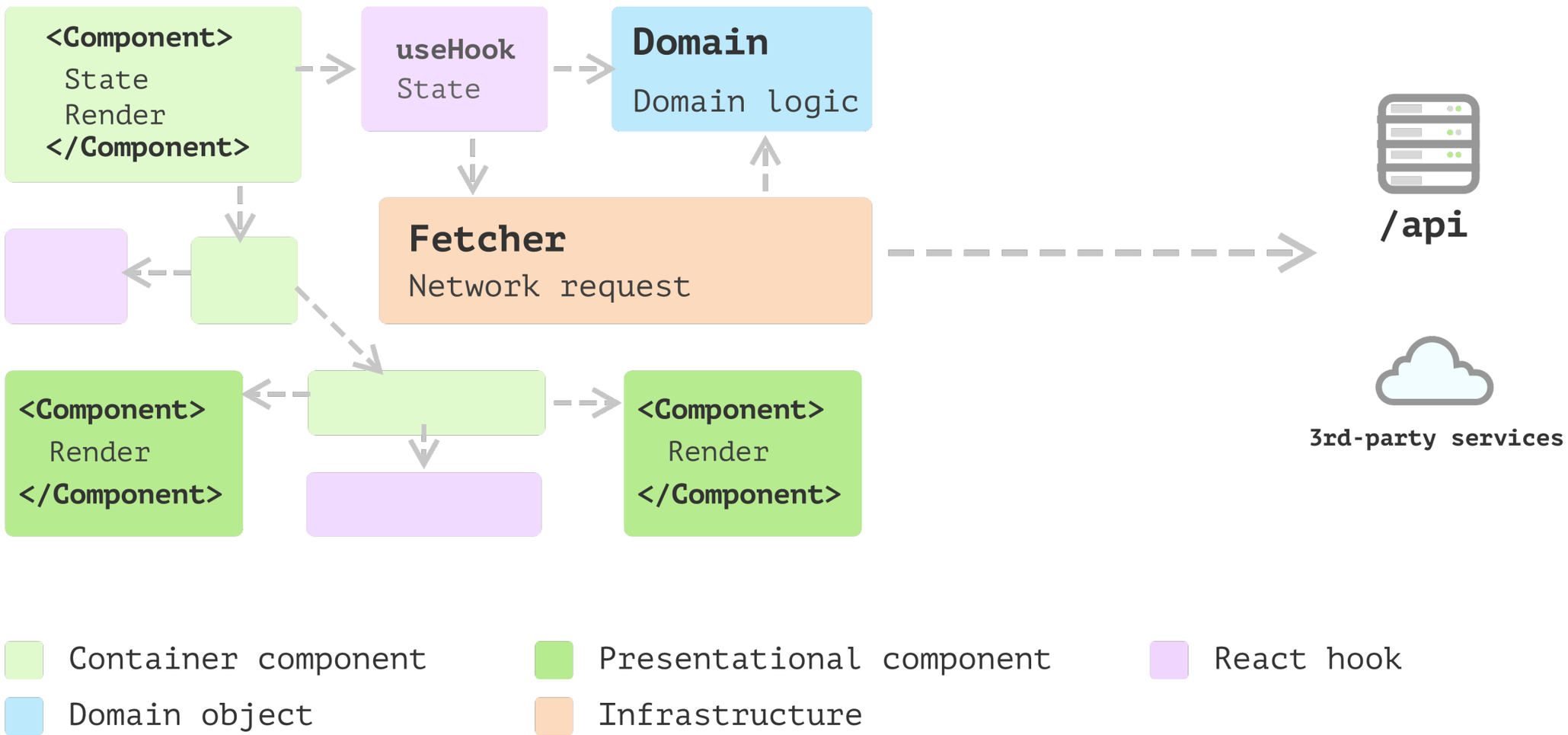


Ref: https://martinfowler.com/articles/modularizing-react-apps.html#ApartFromTheUserInterface

BUSINESS MODELS

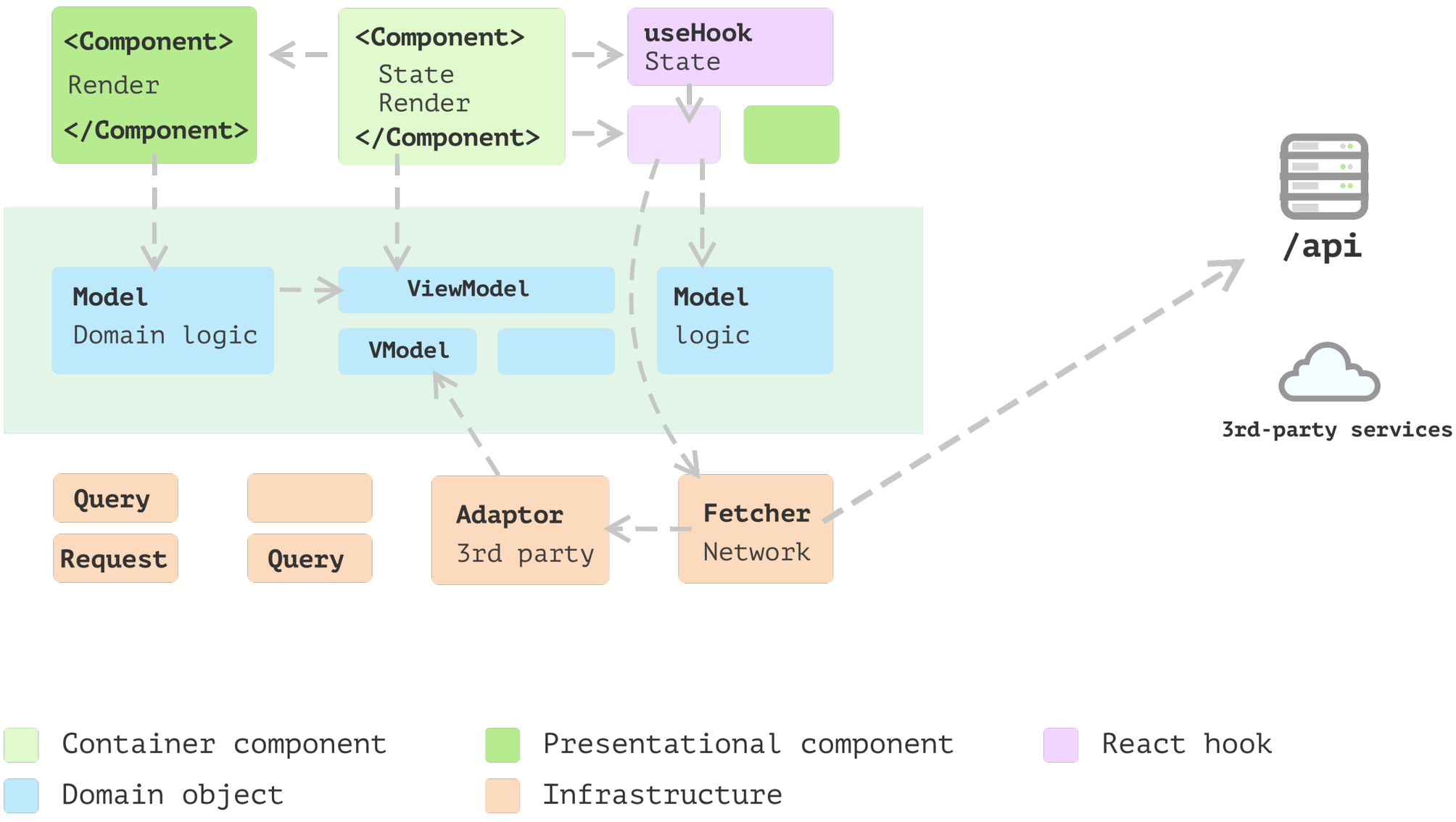
- A domain object corresponds to a real-world concept or entity in the problem domain
  - Examples: User, Order, Product, Invoice
- Encapsulates behaviour and data
  - Contains both data (attributes) and behaviour (methods) that define the entity's responsibilities and rules
  - Example: An Invoice domain object might have an addLineItem() method or a calculateTotal() method
- Domain objects are derived from the business rules and requirements
- Typically mapped to DB entities
- Typically, a lighter version, called DTO is sent as a response to API calls (/api/getUser)
- Where should domain objects (or business models) be defined in a React App ?

BUSINESS MODELS & DOMAIN LOGIC



Ref: https://martinfowler.com/articles/modularizing-react-apps.html#ApartFromTheUserInterface

LAYERED FRONTEND APPLICATION



Ref: https://martinfowler.com/articles/modularizing-react-apps.html#ApartFromTheUserInterface

LAYERED FRONTEND APPLICATION

- Presentation - Domain - Data layering
- Presentation layer
  - Responsible for presenting views and handling user interaction
- Domain layer
  - Responsible for representing data
  - Domain logic/business logic
  - Transformations from one form to another
- Data layer
  - Responsible for fetching data and network handling