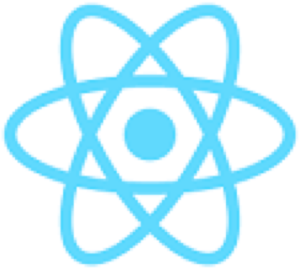


Navigation

(Continued)

(See Routing Samples Archive from earlier lecture for code samples.)

Alternative <Route> API.

- **To-date:** *<Route path={…URL path…} component={ ComponentX} />*
- **Disadv.: We cannot pass props to the mounted component.**
- **Alternative:**

  <Route path={…URL path…} render={…function….}>

  – **where** *function* **return the mounted component.**

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

  Objective: Pass usage data to the <Stats> **component from** sample4..

```
<Route path={`/inbox/:userId/statistics`} component={Stats} />
```

Alternative <Route> API.

```
<Route
  path={`/inbox/:id/statistics`}
  render={ (props) => {
    return <Stats {...props} usage={[5.4, 9.2]} />;
  }}
/>
```

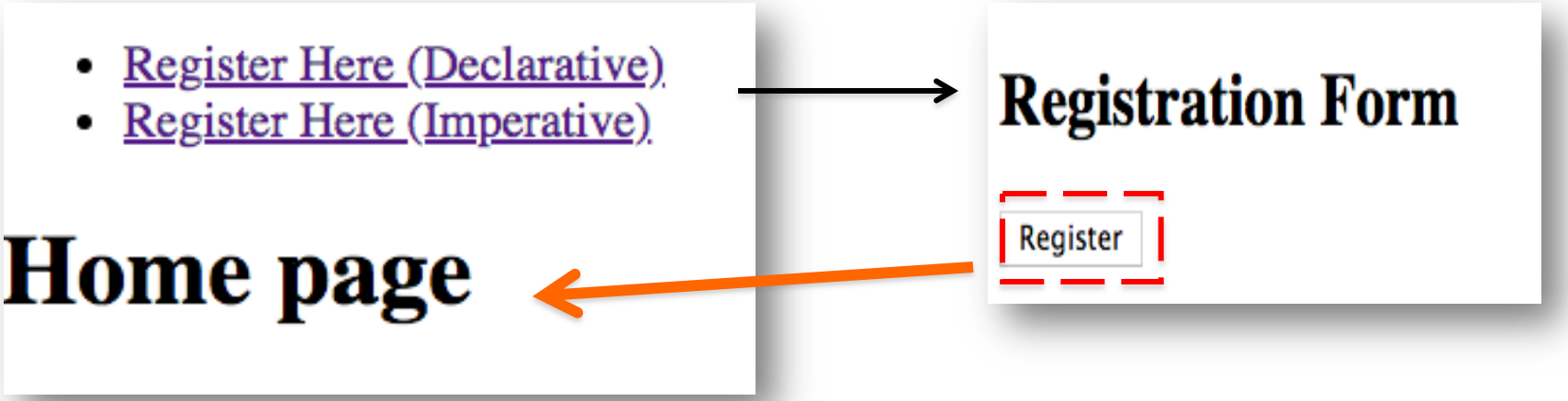
- **Render prop - callback function argument is the default props object for mounted component.**

```
const Stats = (props) => {
  return (
    <>
      <h3>Statistical data for user: {props.match.params.id}</h3>
      <h4>Emails sent (per day) = {props.usage[0]} </h4>
      <h4>Emails received (per day) = {props.usage[1]} </h4>
    </>
  );
};
```

Programmatic Navigation.

- **Performing navigation in JavaScript.**
- **Two options:**
  1. **Declarative – requires state; use** <Redirect />.
  2. **Imperative – requires** withRouter**() ; use** props.history

- **EX.: See** /src/sample8/**.**



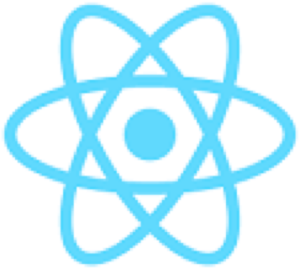
Summary

- **React Router (version 4) adheres to React principles:**
  - **Declarative.**
  - **Component composition.**
  - **The event → state change → re-render**

- **Main components - <BrowserRouter>, <Route>, <Redirect>, <Link>.**

- **The** withRouter() **higher order component.**

- **Additional props:**
  - **props.match.params**
  - **props.history**
  - **props.location**

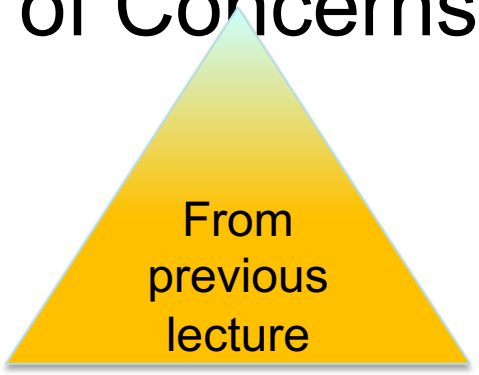


Design Patterns

(Continued)

Reusability & Separation of Concerns.

- **Techniq ues to make cods reusable:**
  1. **Inheritance**
  2. **Composition**



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

The Render Props pattern

- **Use the pattern to share logic between components.**
- **Dfn**.: "a component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic."

```
const SharedComponent = () => {
  return (
    <div className='classX'>
      {this.props.render()}
  </div>
  )
};
const SayHello = () => {
  return (
    <SharedComponent render={() => (
      <span>hello!</span>
    )} />
  )
}
```

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

```
<div className='classX'>
    <span>hello!</span>
</div>
```

The Render Props - Sample App.

- **A React app for viewing blog posts.**
  - **Suppose its views include:**
    1. **A view to display a post's text followed by related comments.**
    2. **A view to display a post's text followed by links to related / matching posts.**

Without Render Props pattern

```
const CommentList = (props) => {
  return (
    <div className='classX'>
      . . . map over comments array
    </div>
  )
};

const BlogPostAndComments = (props) => {
  . . . hooks and other logic . . .
  return (
    <>
      <TextBlock text={....} />
      <CommentList  />
    </>
  )
}
const BlogPostAndMatches = (props) => {
  . . . hooks and other logic . . .
  return (
    <>
      <TextBlock text={....} />
      <BlogMatches />
    </>
  )
}
```

Violates the DRY principle

With Render Props pattern

```
const BlogPost = (props) => {
  . . . hooks and other logic . . .
  return (
    <>
      <TextBlock text={} />
      {this.props.render()}
    </>
  )
}

const BlogPostAndComments = (props) => {
  return (
    <>
      <BlogPost
        render={() => <CommentList /> } />
    </>
  )
}

const BlogPostAndMatches = (props) => {
  return (
    <>
      <BlogPost
        render={() => <PostMatches /> } />
    </>
  )
}
```

BlogPost is told what to render after the blog text

The Render Props pattern

- Render prop function can be parameterized

```
const SharedComponent = (props) => {
  . . . get person data from API . . .
  return (
    <div className='classX'>
      {this.props.render(person.name)}
  </div>
  )
};
const SayHello = (props) => {
  return (
    <SharedComponent render={(name) => (
    <span>hello! {name}</span>
    )} />
  )
}
```

- SharedCoomponent **generates the parameters required by the render prop function.**

Reusability.

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

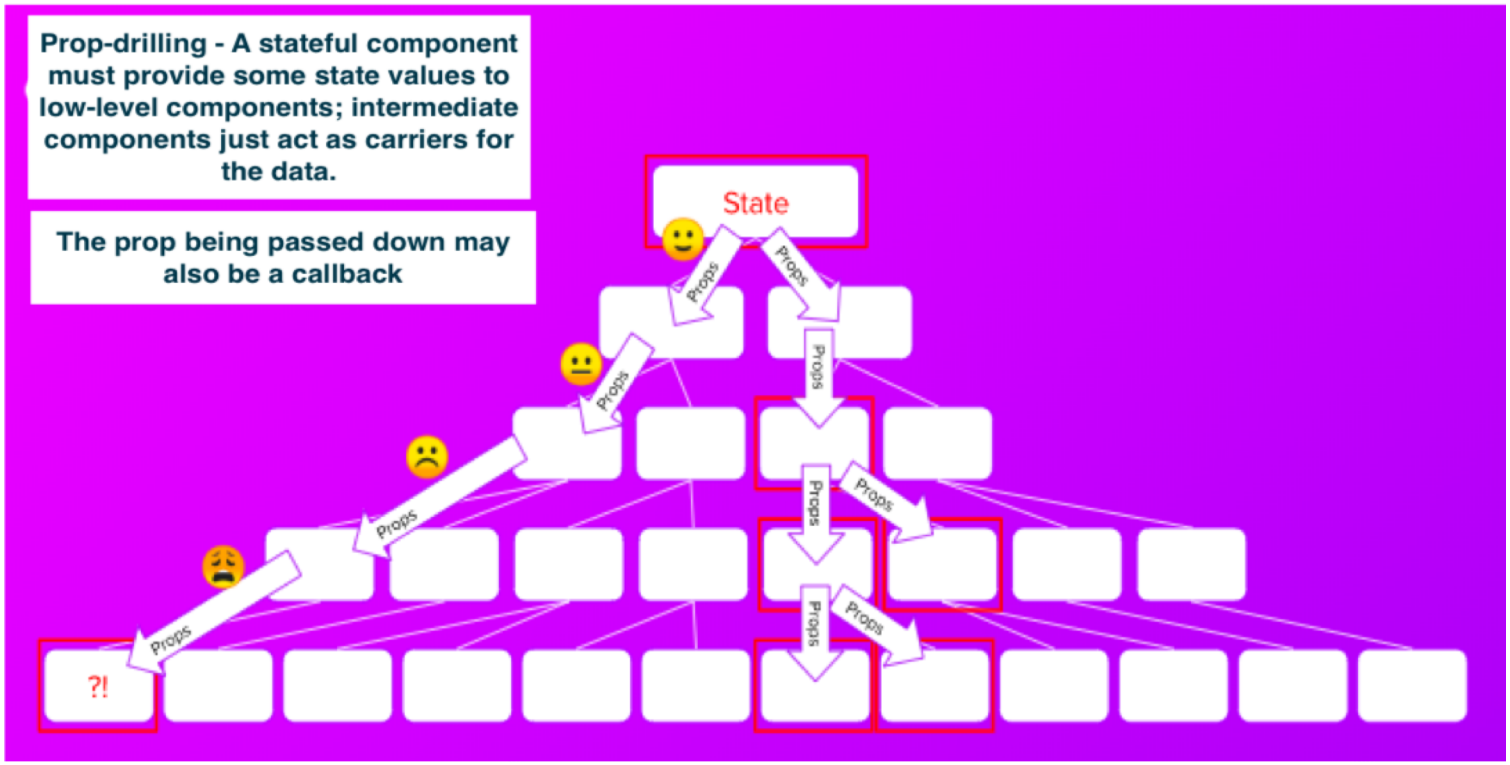
- **HOC is a function that takes a component and returns an enhanced version of it.**
  - **Enhancements include:**
    - **Statefulness**
    - **Props**
    - **UI**
- **Ex –** withRouter **function.**

The Provider pattern

React Contexts

The Provider pattern – When?

- **Use the pattern for:**
  - **Sharing global data, e.g. Web API data.**
  - **To avoid prop-drilling.**

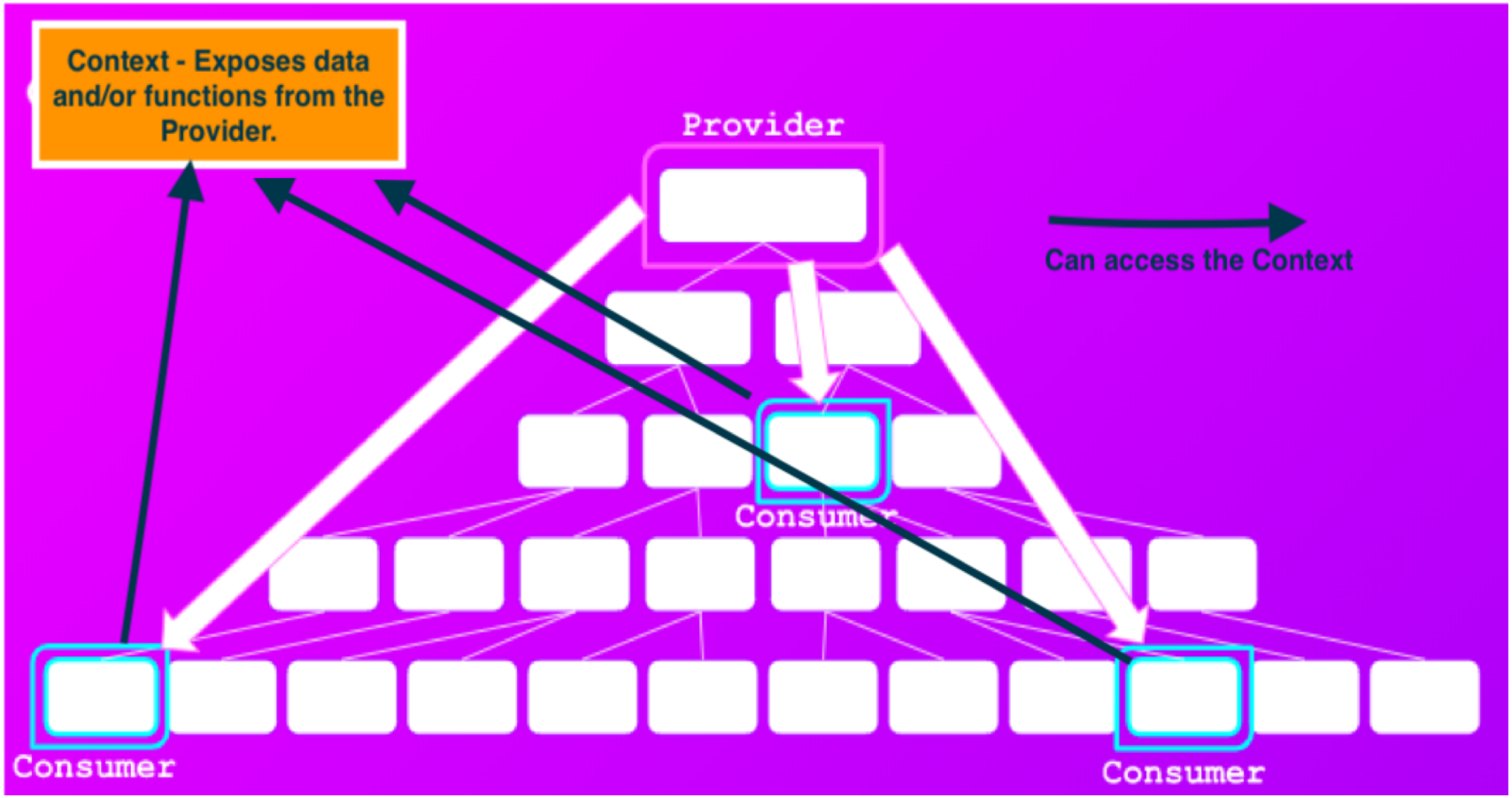


The Provider pattern – How?

- **React Implementation steps:**
  1. **Declare a component for managing global data – the Provider component.**
  2. **Make the data accessible by other components (consumers).**
  3. **Use component composition to integrate the Provider with consumer components.**

- Contexts **– the grue behind provider pattern in React.**
  - **A** Context **provides a way to pass data through the component hierarchy without having to pass props down manually at every level.**
  - **Provider component creates/manages the context.**
  - **Consumer accesses context with** useContext **hook**

The Provider pattern – React Contexts.



The Provider pattern – Implementation

- **Declare the Provider component:**

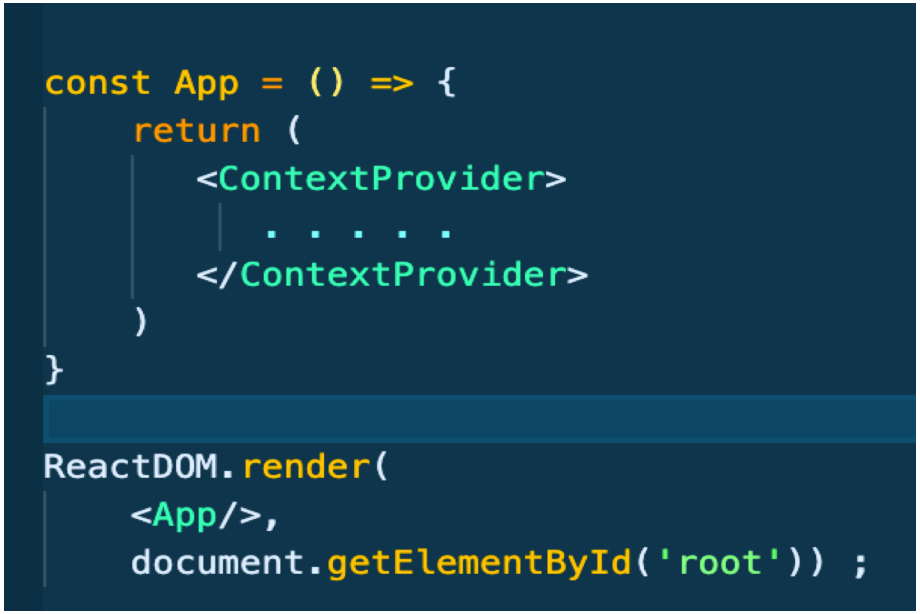
```
export const SomeContext = React.createContext(null)

const ContextProvider = props => {
  . . . Use useState and useEffect hooks to
  . . . initialize global state variables
  return (
    <SomeContext.Provider
         value={{ key1: value1,. . . . }} >
      {props.children}
    </SomeContext.Provider>
  );
};
export default ContextProvider
```

- **We associate the** Context **with the** Provider **component using** <contextName.Provider>.
- **The** values object **declare what is accessible by consumers.**
  - **Functions as well as state data can be values.**

The Provider pattern – Implementation.

- **Integrate (Compose) the Provider with the rest of the app**



```
const App = () => {
    return (
        <ContextProvider>
            . . . . .
        </ContextProvider>
    )
}

ReactDOM.render(
    <App/>,
    document.getElementById('root')) ;
```

- **The Provider's children will now be able to access the shared context.**

The Provider pattern – Implementation.

- **The Consumer accesses the context with** useContext **hook.**

```
import React, { useContext } from "react";
import {SomeContext} from '......'

const ConsumerComponent = props => {
  const context = useContext(SomeContext);

  . . . access context values with 'context.keyX'

};
```

- **The context keys match those of the** values **object exposed by the Provider component.**

The Provider pattern.

- **When not to use Contexts:**
  1. **Don't use Context to avoid drilling props down just one layer. Prop drilling is faster for this case.**
  2. **Avoid using Context to save state that should be kept locally, e.g. web form inputs should be in local state.**
  3. **If you pass an object as your context value, monitor performance and refactor as necessary.**

useReducer hook

(See archive for full source code)

useReducer hook

- **“useReducer** is usually preferable to **useState** when you have complex state logic that involves non-primitive state values or when the next state is computed from the previous one.”

- **Based on the Redux 3rd party library for managing application state.**
  - **The M part of MVC.**

- **Signature:**

  const [state, ***dispatch***] = useReducer(***reducer***, initial state)

useReducer elements.

- **reducer – A custom function for mutating the state**

  reducerFunction(state, action)

  - **Returns the new state value.**
  - **Cannot modify the current state, just make a copy and change the copy.**

- **Reducer** actions **– An object:** { actionType, payload }
  - **Reducer function uses an action's** Type **property to determine the nature of state change required. The** payload **is the data associated with the state change.**

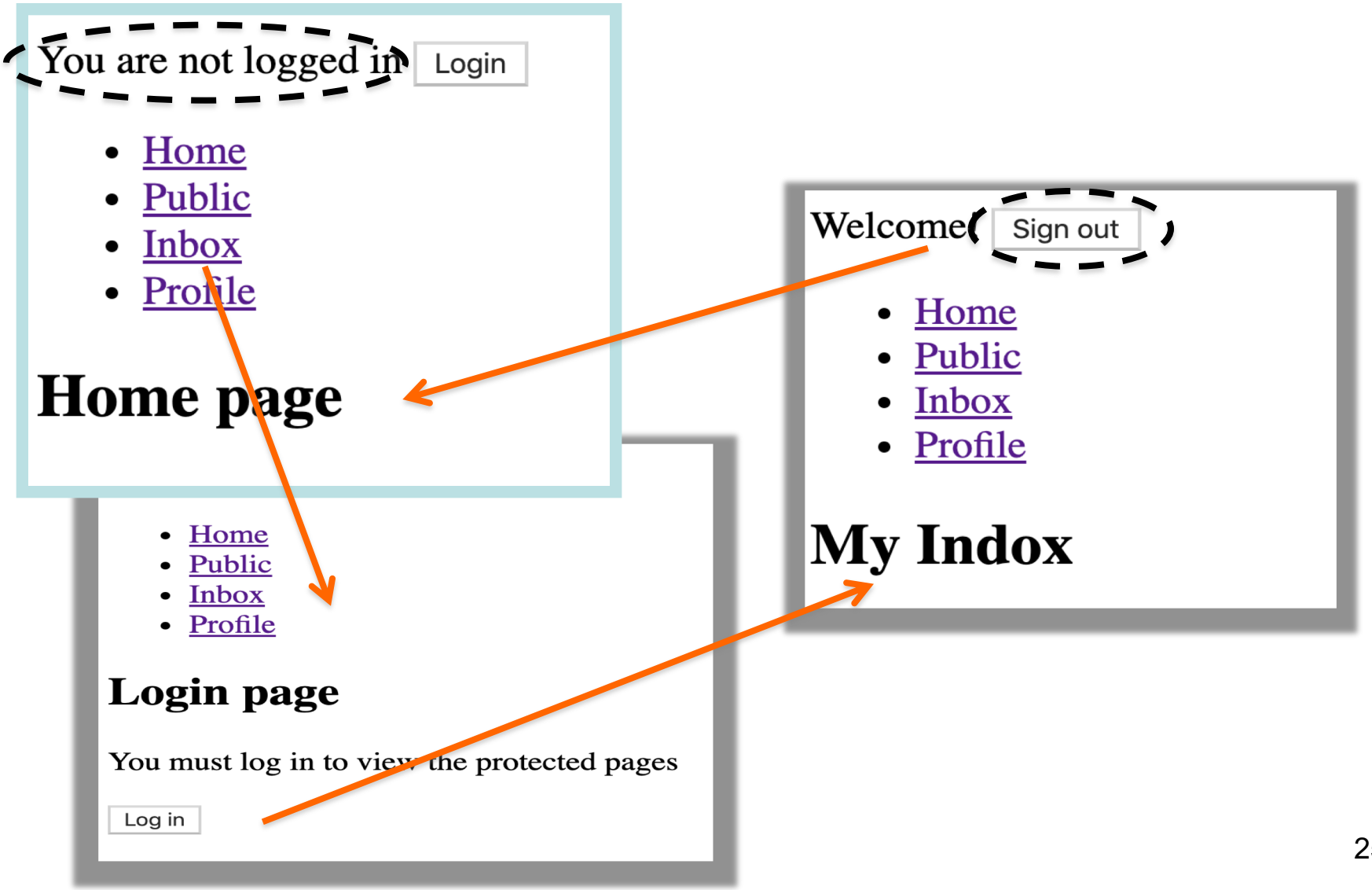
- **dispatch - function to send/dispatch actions to the reducer.**

  dispatcher(action)

Authentication and Protected/Private Routes

(See Archive from lecture 3 for sample code)

Objective


You are not logged in
Login
Home
Public
Inbox
Profile
Home page
Welcome
Sign out
Home
Public
Inbox
Profile
My Indox
Home
Public
Inbox
Profile
Login page
You must log in to view the protected pages
Log in

Protected Routes.

- **Not native to React Router.**
- **We need a custom solution.**
- **Solution outline: Clear, declarative style for declare views/pages requiring authentication:**

```
<Switch>
  <Route path="/public" component={PublicPage} />
  <Route path="/login" component={LoginPage} />
  <Route exact path="/" component={HomePage} />
  <PrivateRoute path="/inbox" component={Inbox} />
  <PrivateRoute path="/profile" component={Profile} />
  <Redirect from="*" to="/" />
</Switch>
```

Protected Routes.

- **Solution features:**

  1. **React Context to store current authenticated user.**
  2. **Programmatic navigation - to redirect unauthenticated user to login page.**
  3. **Remember user's intent before forced authentication.**

Protected Routes

- **Solution elements: The AuthContext.**

```
3   export const AuthContext = createContext(null);
4
5   const AuthContextProvider = (props) => {
6     const [isAuthenticated, setIsAuthenticated] = useState(false);
7
8     const authenticate = (username, password) => {…
13    };
14
15    const signout = () => {…
17    }
18
19    return (
20      <AuthContext.Provider
21        value={{
22          isAuthenticated,
23          authenticate,
24          signout,
25        }}
26      >
27        {props.children}
28      </AuthContext.Provider>
29    );
30  };
```

Protected Routes

- **Solution elements (Contd.): <PrivateRoute />**

```
<PrivateRoute path="/inbox" component={Inbox} />
```

```
5   const PrivateRoute = props => {
6     const context = useContext(AuthContext)
7     // Destructure props from <privateRoute>
8     const { component: Component, ...rest } = props;
9
10    return context.isAuthenticated === true ? (
11      <Route {...rest} render={props => <Component {...props} />} />
12    ) : (
13      <Redirect
14        to={{
15          pathname: "/login",
16          state: { from: props.location }
17        }}
18      />
19    );
20  };
21
```

```
{pathname: "/inbox", sear
key: "0pfafo"} i
  hash: ""
  key: "0pfafo"
  pathname: "/inbox"
  search: ""
  state: undefined
▶ __proto__: Object
```

Protected Routes

- **Solution elements (Contd.): <LoginPage>**

```
5  const LoginPage = props => {
6    const context = useContext(AuthContext)
7
8    const login = () => {
9      context.authenticate("user1", "pass1");
10   };
11   const { from } = props.location.state || { from: { pathname: "/" } };
12
13   if (context.isAuthenticated === true) {
14     return <Redirect to={from} />;
15   }
16   return (
17     <>
18       <h2>Login page</h2>
19       <p>You must log in to view the protected pages </p>
20       {/* Login web form  */}
21       <button onClick={login}>Log in</button>
22     </>
23   );
24 };
```