

Navigation

(Continued - See Archive from week 5
for code samples.)

Alternative <Route> API.

- **To-date:** `<Route path={...URL path...} component={ ComponentX} />`
- **Disadv.: We cannot pass props to the component.**
- **Alternative:**
`<Route path={...URL path...} render={...function....}>`
– **where function return a component.**
- **EX.: See /src/sample5/.**
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]} />;
  }}
/>
```

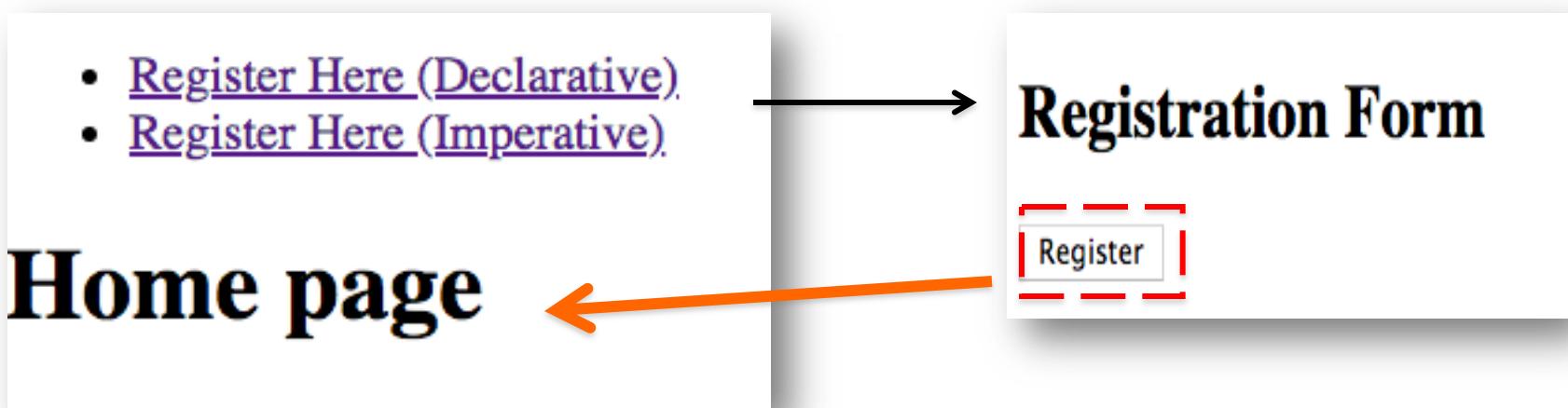
- **The <Route> component's own props object is the default parameter for the render function.**

```
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>
    </>
  );
};
```



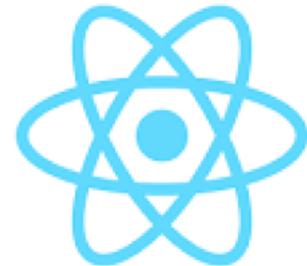
Programmatic Navigation.

- Performing navigation in JavaScript.
- Two options:
 1. Declarative – requires state; use <Redirect />.
 2. Imperative – requires withRouter() ; use this.props.history
- EX.: See /src/sample7/.



Summary

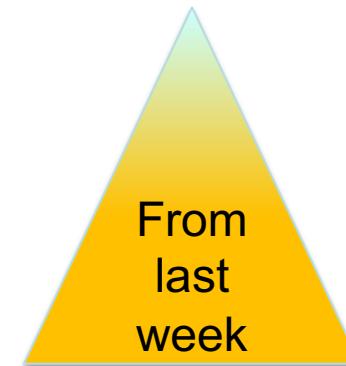
- **React Router (version 4) adheres to React principles:**
 - Declarative.
 - Component composition.
 - The event → state change → re-render
- **Main components - <BrowserRouter>, <Route>, <Redirect>, <Link>**
- **The withRouter() higher order component.**
- **Additional props:**
 - `props.match.params`
 - `props.history`
 - `props.location`



Design Patterns

(Continued)

Reusability.



- Techniques to make codes 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>
    )} />
  )
};
```

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

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

The Render Props - Sample App.

- A React app for viewing blog posts.
 - Suppose its views include:
 1. A view to display a post text followed by related comments.
 2. A view to display a post 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()}
    </>
  )
}
```

BlogPost told what to render after the blog text

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

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

The Render Props pattern

- Render prop function can be parameterized

```
const SharedComponent = (props) => {
  . . .
  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.

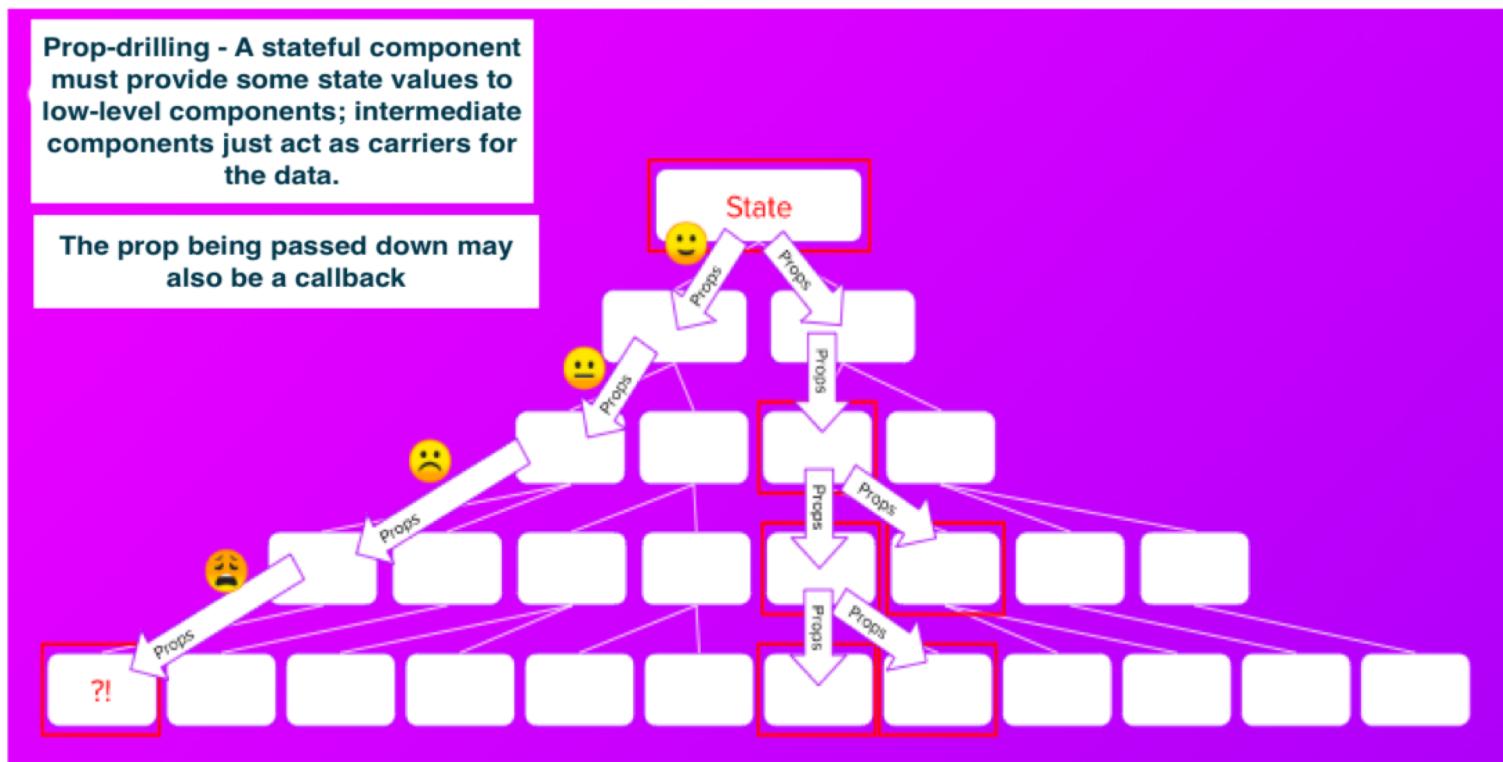
- Core React composition Patterns:
 1. Containers
 2. Render Props
 3. Higher Order Components
- HOC is a function that takes a component and returns an enhanced version of it.
 - Enhancements include:
 - Statefulness
 - Props
 - UI
- Ex – withRouter function.

The Provider pattern

React Contexts

The Provider pattern – When?

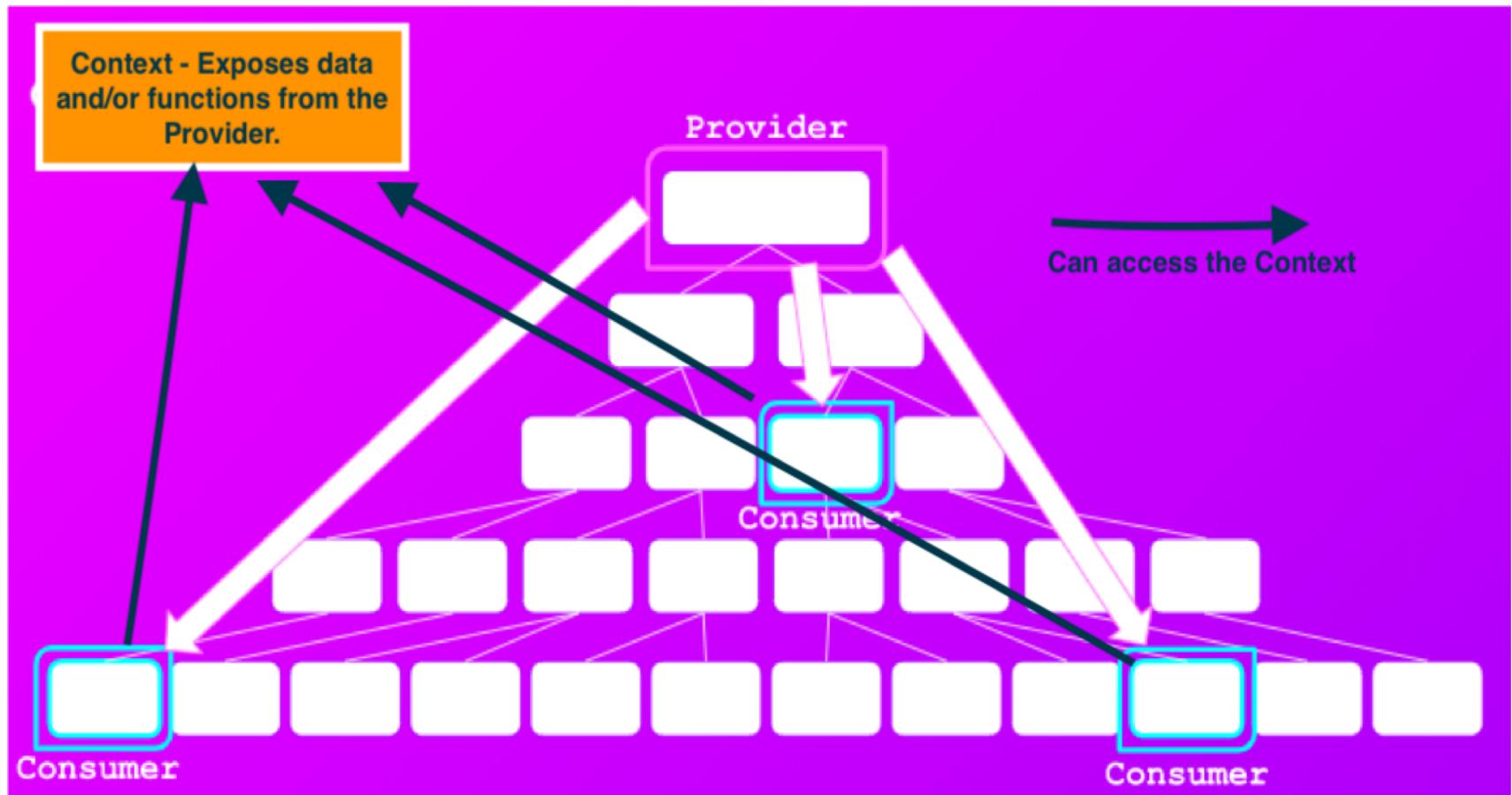
- **Use the pattern for:**
 - Sharing global state, e.g. Web API data.
 - To avoid prop-drilling.



The Provider pattern – How?

- **React Implementation steps:**
 1. **Declare a component for managing global state – the Provider component.**
 2. **Make the state accessible by other components.**
 - **Using a Context in React.**
 3. **Use component composition to integrate the Provider with consumer components.**
 4. **Consumer uses useContext hook to access Provider's Context.**
- **A Context provides a way to pass data through the component tree without having to pass props down manually at every level.**

The Provider pattern – React Contexts.



The Provider pattern – Implementation

- **Declare the Provider component:**

```
0  export const SomeContext = React.createContext(null)
1
2  const ContextProvider = props => {
3      . . . Use useState and useEffect hooks to
4      . . . initialize global state variables
5      return (
6          <SomeContext.Provider
7              value={{ key1: value1, . . . }} >
8              {props.children}
9          </SomeContext.Provider>
0      );
1  };
2  export default ContextProvider
```

- **We associate the Context with the Provider component using <contextName.Provider>.**
- **The values object declare what is accessible by consumer.**
 - 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.**

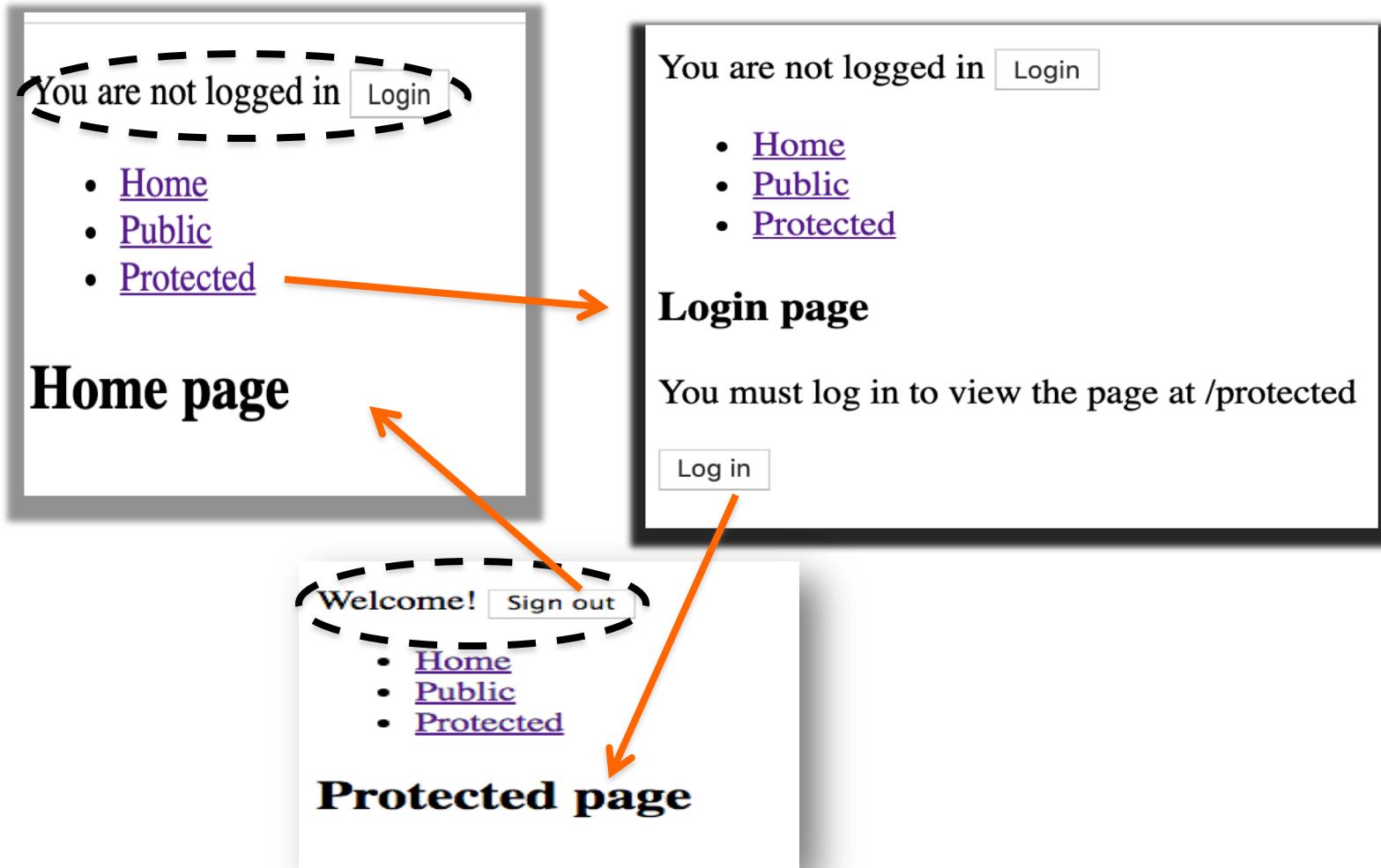
The Provider pattern.

- **When not to use Contexts:**
 1. **Don't use Context to avoid drilling props down just one or two layers. Prop drilling is faster for a couple of layers.**
 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.**

Authentication and Protected Routes

(See archive for full source code)

Objective



Protected Routes.

- Not native to React Router.
- We need a custom solution:
- Solution elements:
 1. 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='/protected' component={ ProtectedPage } />
  <Redirect from='*' to='/' />
</Switch>
...
```

Declarative

Protected Routes

- **Solution elements (Contd.):**

2. Toggle the app's authentication status header based on login status:

```
5  const BaseAuthHeader = props => {
6    const { history } = props;
7    const signOutHandler = () => authenticator.signout(() => history.push("/"));
8    return authenticator.isAuthenticated ? (
9      <p>
10        | Welcome! <button onClick={signOutHandler}>Sign out</button>
11      </p>
12    ) : (
13      <p>
14        | You are not logged in{" "}
15        <button onClick={() => history.push("/login")}>Login</button>
16      </p>
17    );
18  };
19  export default withRouter(BaseAuthHeader);
20
```

Protected Routes

- **Solution elements (Contd.):**
 3. **<LoginPage>** – For a valid login it redirects to the protected view the user tried to access pre-authentication (`props.location.state`); otherwise it redisplays the Login page:

```
5  const LoginPage = props => {
6    const [loggedInStatue, setLoggedInStatus] = useState(false);
7    const login = () => {
8      authenticator.authenticate("user1", "pass1", status =>
9        setLoggedInStatus(status)
10     );
11   };
12   const { from } = props.location.state || { from: { pathname: "/" } };
13   if (loggedInStatue === true) {
14     return <Redirect to={from} />;
15   }
16   return (
17     <>
18     . . . . .
19     <button onClick={login}>Log in</button>
20   );
}
```

Protected Routes

- **Solution elements (Contd.):**
 4. **Authenticator – checks submitted credentials and performs appropriate action (callback parameter), if successful.**

```
1  class Authenticator {  
2    constructor() {  
3      this.isAuthenticated = false;  
4    }  
5    this.users = [...  
6    ];  
7  }  
8  
9  
10   authenticate(username, password, cb) {  
11     setTimeout(() => {  
12       const validUser = this.users.find(  
13         u => u.username === username && u.password === password  
14       )  
15       this.isAuthenticated = validUser ? true : false  
16       cb(this.isAuthenticated);  
17     }, 100);  
18   }  
19 }
```

Protected Routes

- **Solution elements (Contd.):**
 - 5. **<PrivateRoute>** – checks the user's authentication status:
 - **Authenticated** – Returns a normal **<Route>** component.
 - **Unauthenticated** – Returns a **<Redirect>** to the login page.

```
<PrivateRoute path="/protected" component={ ProtectedPage } />
```

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



A tooltip for the 'props.location' object is displayed, showing its properties:

- pathname: "/protected"
- search: ""
- hash: ""
- state: undefined
- key: "3gs8f0"

The 'key' property is highlighted in red.

Protected Routes

- **Recommendation – Store the current user in a context.**