

# Agenda

- **Design patterns (Contd.)**
  - **The Provider Pattern**
- **Data Fetching and Caching**
  - **The react-query library**
- **Routing.**
  - **Protected routes and authentication.**

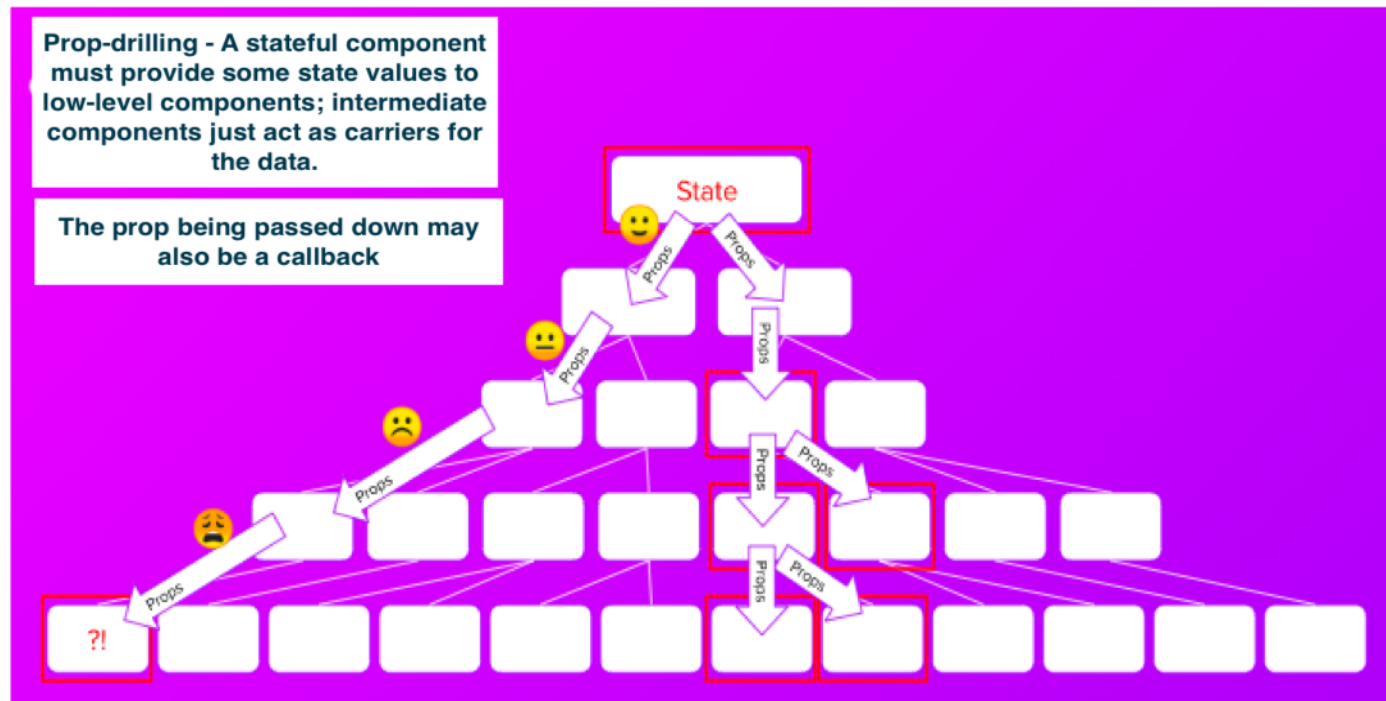
# **React Contexts**

The Provider pattern

# The Provider pattern – When?

- **Use cases:**

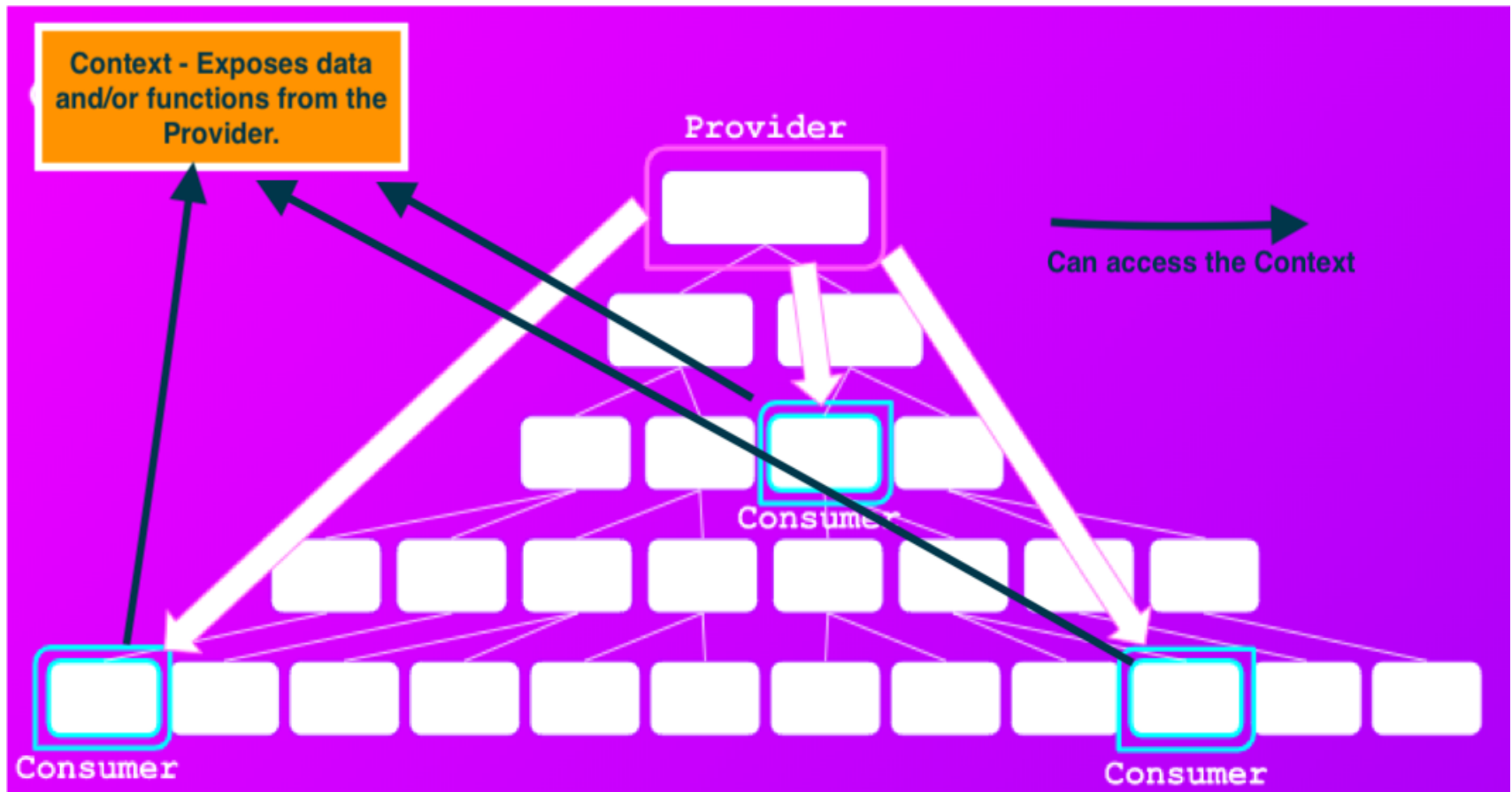
1. Sharing data/state **with multiple components**, i.e. global data, **e.g. favourite movies**.
2. **To avoid** prop-drilling.



# The Provider pattern – How?

- **React Implementation steps:**
  1. **Declare a component for managing the data to be shared – the Provider component.**
  2. **Create a context construct and associate it with the Provider.**
  3. **Place the data to be shared inside the context.**
  4. **Use composition to integrate Provider and consumer(s).**
  5. **Use a hook to enable consumers access the context's data.**
- **Contexts – the glue for the provider pattern in React.**
  - **Enables a component to share its data (and behaviour) with subordinates, without the need for prop drilling.**
  - **Provider component manages the context.**
  - **Consumer accesses the context with useContext hook**

# 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>
10  );
11 };
12 export default ContextProvider
```

- **We associate the Context with the Provider component using `<contextName.Provider>`.**
- **The values object declares the context's content. .**
  - **Can be functions as well as (state) data.**

# The Provider pattern – Implementation.

- Integrate (Compose) the Provider with the rest of the app using the Container pattern

```
const App = () => {  
  return (  
    <BrowserRouter>  
      <ContextProvider>  
        . . . . .  
      </ContextProvider>  
    </BrowserRouter>  
  );  
};  
  
ReactDOM.render(<App />, document.getElementById("root"));
```

- The Provider's children (and their children) will now be able to access the context.

# The Provider pattern – Implementation.

- **The context consumer uses the useContext hook.**  
contextRef = useContext(ContextName)  
// contextRef points at context's values object.
  - Use contextRef.key **to access an element inside the context.**

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

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

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

};
```



# The Provider pattern – Implementation.

- **For better separation of concerns, have multiple context instead of a 'catch all' context.**

```
const App = () => {  
  return (  
    <BrowserRouter>  
      <ContextProviderA>  
        <ContextProviderB>  
          . . . . .  
        </ContextProviderB>  
      </ContextProviderA>  
    </BrowserRouter>  
  );  
};
```

# The Provider pattern.

- **When NOT to use Contexts:**
  1. **To avoid 'shallow' prop drilling.**
    - **Prop drilling is faster for 'shallow' cases.**
  2. **To save state that should be kept local to a component, e.g. web form inputs.**
  3. **For large object - monitor performance and refactor as necessary.**

# **Data Fetching & Caching.**

# SPA State (Data)

- **Client state (aka App State).**
  - e.g. Menu selection, UI theme, Text input, logged-in user id.
  - **Characteristics:**
    - **Client-owned; Not shared; Not persisted (across sessions); Up-to-date.**
    - **Accessed synchronously.**
    - **useState() hook**
    - **Management - Private to a component or Global state (Context).**

# SPA State (Data)

- **Server state (The M in MVC).**
  - e.g. list of ‘discover’ movies, movie details, friends.
  - **Characteristics:**
    - **Persisted remotely. Shared ownership.**
    - **Accessed asynchronously → Impacts User experience.**
    - **Can change without client’s knowledge → Client can be ‘out of date’.**
    - **useState + useEffect hooks.**

# SPA Server State.

- **Server state characteristics (contd.).**
  - **Management options:**
    1. **Private to a component →**
      - **Good separation of concerns.**
      - **Unnecessary re-fetching.**
    2. **Global state (Context).**
      - **No unnecessary re-fetching.**
      - **Fetching data before its required.**
      - **Poor separation of concerns.**
    3. **3<sup>rd</sup> party library – e.g. Redux**
      - **Same as 2 above.**
- **We want the best of 1 and 2, if possible.**

# Sample App.

[Home](#)

## Movie List

- [The Conjuring: The Devil Made Me Do It](#)
- [Cruella](#)
- [Wrath of Man](#)
- [The Unholy](#)
- [Spiral: From the Book of Saw](#)
- [A Quiet Place Part II](#)
- [Army of the Dead](#)
- [Mortal Kombat](#)
- [Godzilla](#)

[Home](#)

## Movie Details

```
{
  "adult": false,
  "backdrop_path": "/6MKr3KgOLmzOP6MSuZERO41Lpkt.jpg",
  "belongs_to_collection": {
    "id": 837007,
    "name": "Cruella Collection",
    "poster_path": null,
    "backdrop_path": null
  },
  "budget": 200000000,
  "genres": [
    {
      "id": 35,
      "name": "Comedy"
    },
    {
      "id": 80,
      "name": "Crime"
    }
  ],
  "homepage": "https://movies.disney.com/cruella",
  "production_companies": [
```

- Both pages make HTTP Request to a web API (TMDB)

# Sample App – The Problem.

The screenshot shows a web browser at localhost:3000 displaying a 'Movie List' page. The page has a search bar and a list of movie titles. A network inspector is open on the right, showing a list of HTTP requests. Red arrows point from the movie titles to the corresponding requests in the network log.

**Movie List**

Search

- [The Conjuring: The Devil Made Me Do It](#)
- [Cruella](#)
- [Wrath of Man](#)
- [The Unholy](#)
- [Spiral: From the Book of Saw](#)
- [A Quiet Place Part II](#)
- [Army of the Dead](#)
- [Mortal](#)
- [Godzill](#)
- [Endang](#)
- [Tom Cl](#)

Slide 15

**Network**

Filter: XHR JS CSS Img Media Font Doc WS Manifest Other Has blocked cookies

Blocked Requests

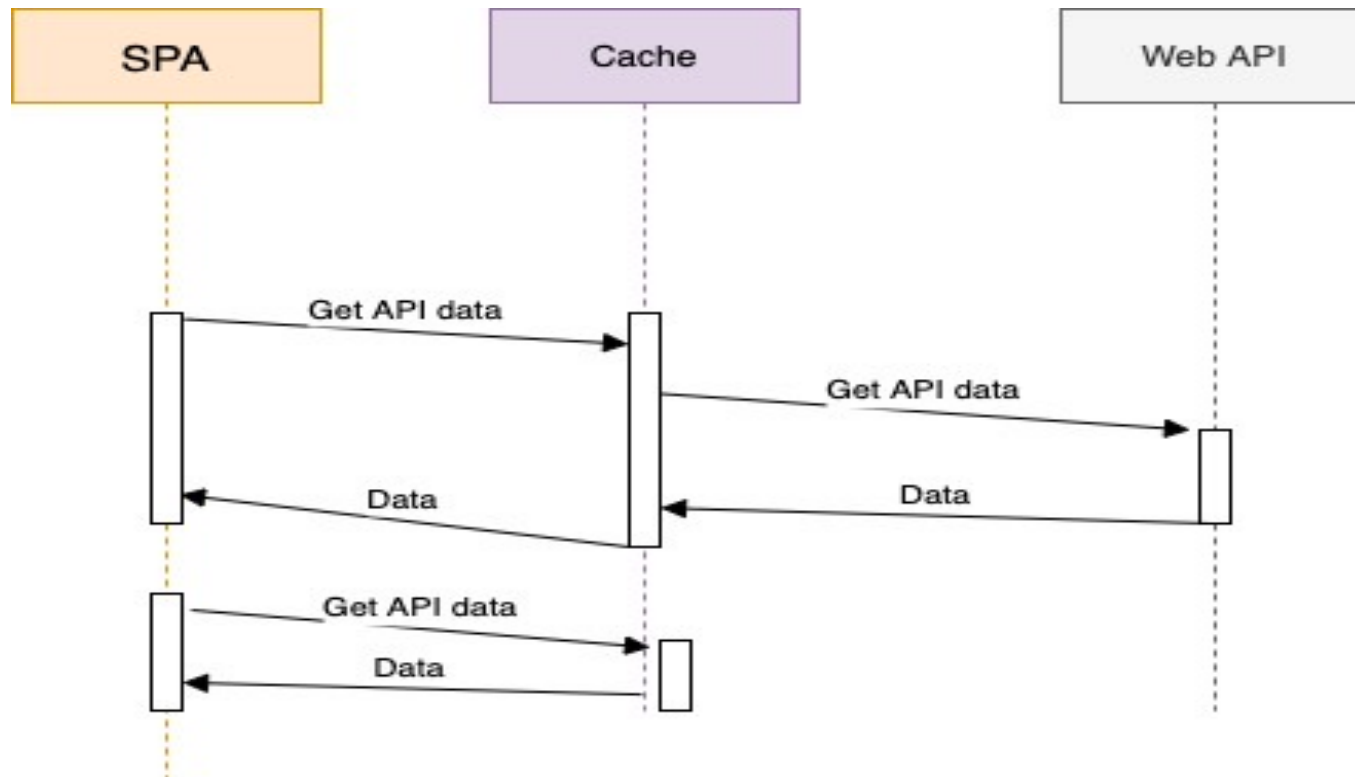
Name	Status	Type	Initiator	Size	Time	Waterfall
movie?api_key=...	200	fetch	VM24:1	(di...	1 ...	
423108?api_key...	200	fetch	VM24:1	1.5...	30...	
movie?api_key=...	200	fetch	VM24:1	(di...	1 ...	
423108?api_key...	200	fetch	VM24:1	(di...	1 ...	
movie?api_key=...	200	fetch	VM24:1	(di...	1 ...	
337404?api_key...	200	fetch	VM24:1	1.4...	27...	
movie?api_key=...	200	fetch	VM24:1	(di...	1 ...	

- Every navigation to the Home page triggers an HTTP request to TMDB.
- Similarly for the Detail page.
- Both pages use useEffect and useState hooks.



# Sample App – The Solution. .

- Cache the API data locally in the browser.
- Caches are in-memory datastores with high performance, low latency.
- Helps reduce the workload on of the backend for read intensive workloads.



# Caching (General).

- **Caches are key-value datastores.**
  - key1: value, key2: value, .....
  - **Keys must be unique.**
  - **Value can be any serializable data type – JS Object, JS array, Primitive.**
- **Cache hit – The requested data is in the cache.**
- **Cache miss - The requested data is not in the cache.**
- **Caches have a simple interface:**
  - serializedValue = cache.get(key)
  - cache.delete(key)
  - cache.purge()
- **Cache entries should have a time-to-live (TTL).**

# The react-query library

- **3<sup>rd</sup> party JavaScript (React) caching library.**

- **Provides a set of hooks.**

e.g. `const { data, error, isLoading, isError } =`

`useQuery(key, queryFunction);`

- data – from the cache or returned by the API.
  - error – error response from API.
  - isLoading(boolean) – true while waiting for API response.
  - isError (boolean) – true when API response is an error status.
- **It causes a component to re-render on query completion.**
- **It relaces your useState and useEffect hooks.**

# The query key.

- *“Query keys can be as simple as a string, or as complex as an array of many strings and nested objects. As long as the query key is serializable, and **unique to the query's data** .....*”

e.g. `const { ....., } =  
 useQuery( ["movie", { id: 123456 }], getMovie);`

`export const getMovie = async (args) => {  
 const [ prefix, { id: id } ] = args.queryKey;  
 .... Do HTTP GET using movie id of 123456`

# react-query DevTools.

- Allows us observe the current state of the cache data store – great when debugging.

The screenshot displays the react-query DevTools interface. At the top, a browser window shows 'localhost:3000'. Below the browser, a 'Movie List' application is visible with a search bar and a list of movies including 'Cruella' and 'The Conjuring: The Devil Made Me Do It'. The DevTools interface is open, showing a query in a 'fresh' state. The query is '[ "discover" ]'. The status bar indicates 'fresh (1)', 'fetching (0)', 'stale (0)', and 'inactive (2)'. The 'Query Details' panel shows the query name 'discover' and its status 'fresh'. The 'Data Explorer' panel shows the query results: 'page: 1', 'results: 20 items', and 'total\_pages: 500'. A 'Close' button is visible at the bottom left of the DevTools panel.

Home

## Movie List

Search

- [Cruella](#)
- [The Conjuring: The Devil Made Me Do It](#)

fresh (1) fetching (0) stale (0) inactive (2)

Filter Sort by Status > Last Upd: ↑ Asc

1 [ "discover" ]

0 [ "movie", { "id": "337404" } ]

0 [ "movie", { "id": "423108" } ]

Query Details

"discover" fresh

Observers: 1

Last Updated: 09:40:07

Actions

Refetch Invalidate Reset Remove

Data Explorer

▼ Data 4 items

page: 1

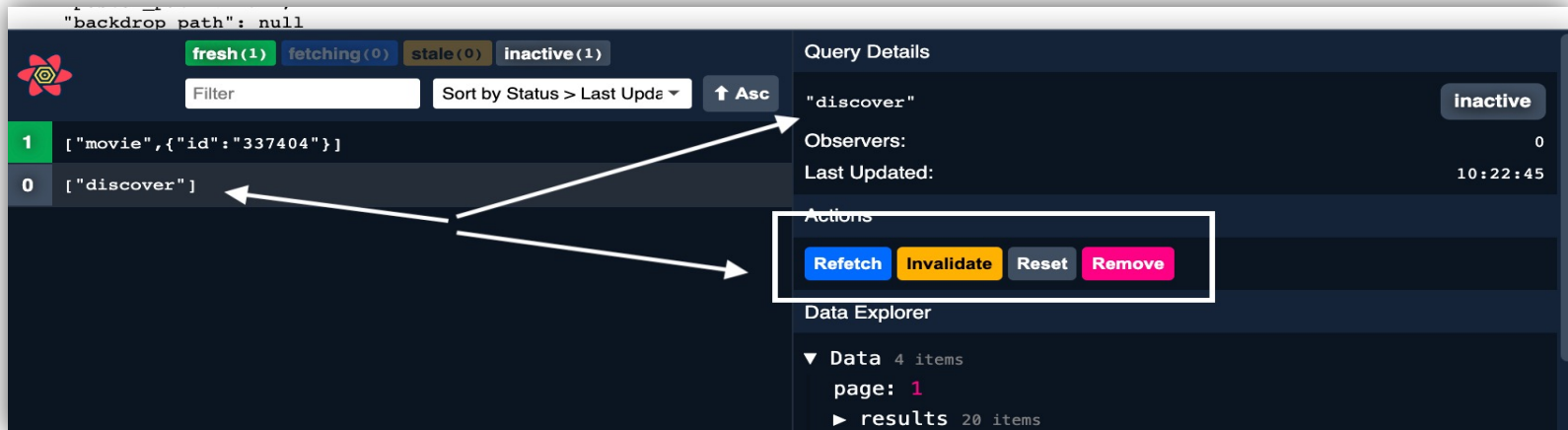
► results 20 items

total\_pages: 500

Close

# react-query DevTools.

- Allows us manipulate cache entries.



- Refresh – force cache to re-request (update) data from web API immediately.
- Invalidate – Set entry as 'stale'. Cache will request update from web API when required by the SPA.
- Reset – only applies when app can update data.
- Remove – remove entry from cache immediately.

# Summary

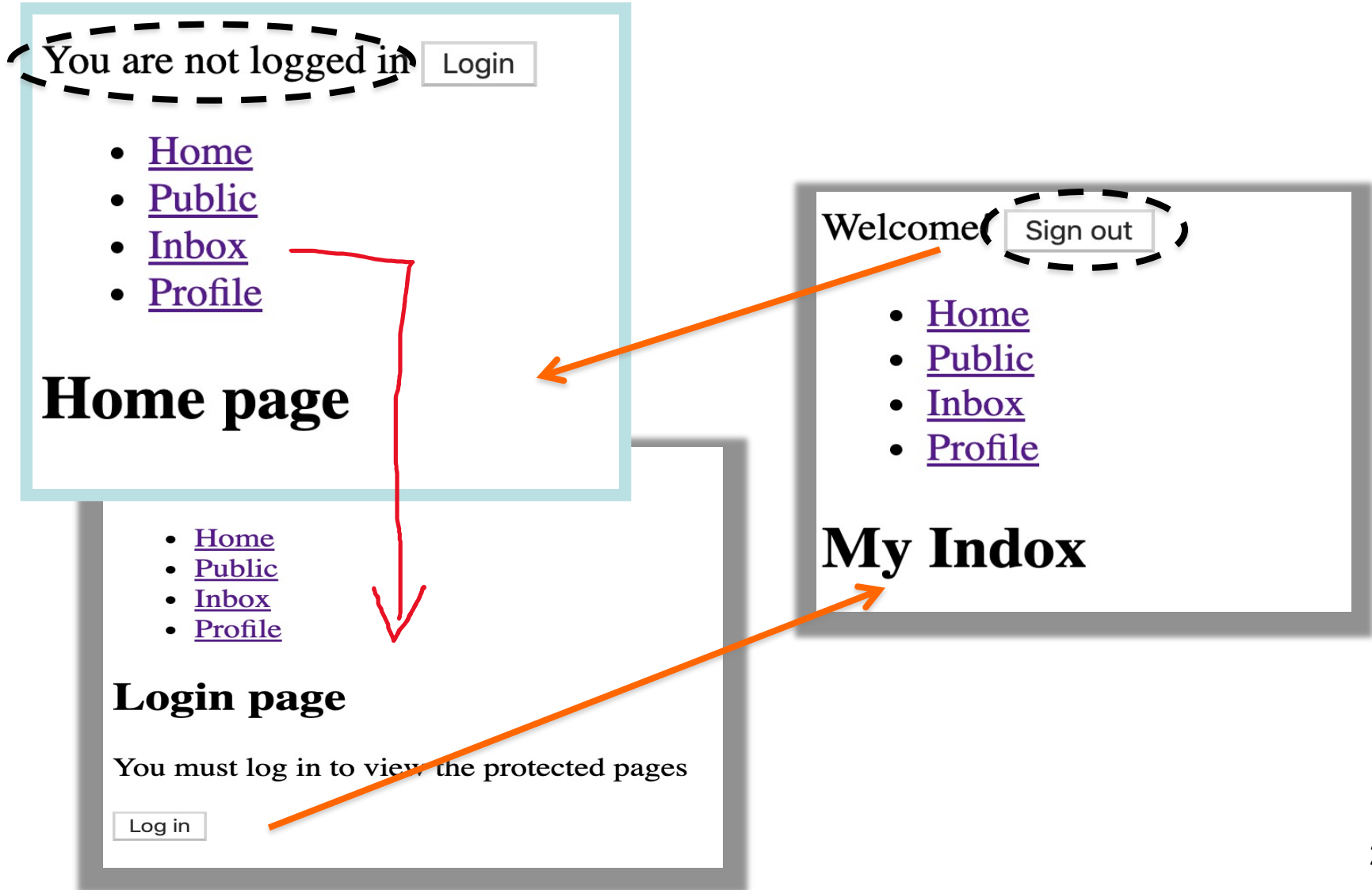
- **State Management - The M in MVC**
- **State:**
  1. **Client/App state.**
  2. **Server state.**
- **Cache server state locally in the browser.**
  - **Reduces unnecessary HTTP traffic → Reduce page loadtime**
  - **Be aware of cache entry staleness → Use TTL.**
- **The react-query library**
  - **A set of hooks for cache interaction.**

# **Authentication and Protected/Private Routes**

(See Routing samples Archive)




# Objective



# 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 - Implementation

- Solution elements: The AuthContext.

```
import React, { useState, useEffect, createContext } from "react";

export const AuthContext = createContext(null);

const AuthContextProvider = (props) => {
  const [user, setUser] = useState({ username: null, password: null });

  const authenticate = (username, password) => {
    // .... Validation user credentials somehow .....
    setUser({ username, password });
  };

  const isAuthenticated = user.username !== null ? true : false;
  const signout = () => {
    setTimeout(() => setUser({ username: null, password: null }), 100);
  };

  return (
    <AuthContext.Provider
      value={{
        isAuthenticated,
        authenticate,
        signout,
      }}
    >
      {props.children}
    </AuthContext.Provider>
  );
};


export default AuthContextProvider;
```

# Protected Routes - Implementation

- 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 };
```



```
{pathname: "/inbox", search: "", key: "0pfafo"}  
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  | };  
25
```

