State Management

Sridhar Alagar

Photo Sharing App so far…

- Simple - set up on startup and static
  - Have a nice modular design of view components
  - Each unit independently fetches data

- Add in Session state, object creation and updating – gets complicated

- User adds photos or comments
  - Model data of one view changed by another view

- User logs out and login as a different user
  - Big change in model data viewed

Session state shared between frontend and backend

- Must be in sync between browser and server
  - who is logged in?

- Server should reject any requests from users not logged in

- Consider transitions of your photo app
  - Login - Not logged in to logged in
    - At app startup most models are not available (e.g. sidenav user list) but become available after login is completed
  - Logout - Logged in to not logged in
    - Requests to web server that worked before will now fail

Update models

- New users, photos, comments added – model change
  - Multiple users may be logged in at the same time

- Controller fetching model during startup may not work

- A view may need to be refreshed when a photo/comment added
  - One user may be viewing a photo, and another user may post a comment
    - New comment should show up on the view
    - Can be annoying to some users if changes are too frequent

Components are interested in some external events

- How to keep a modular design but allow controllers to be notified of things happening outside of it?
  - Example: a view component and an add component

- One option: Explicit communication interfaces in components
  - Pass callback functions around to components

- Better option: Listener/emitter pattern
  - Components registers interest (listen) and component detecting change signals (emit)

State management problem

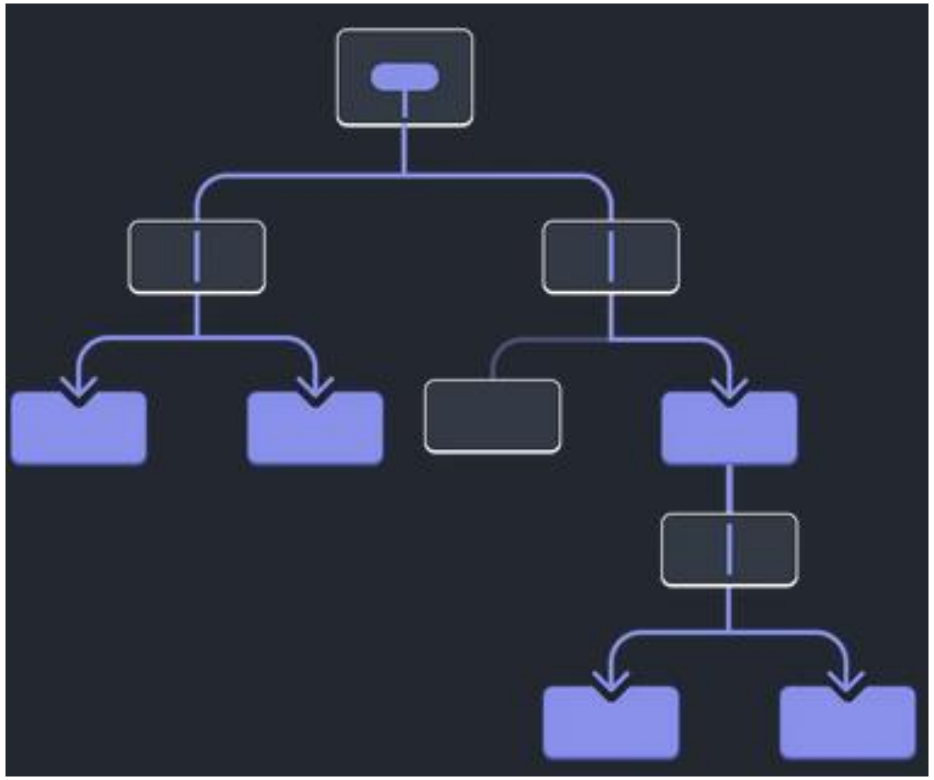
- State needs to shared by several components
  - Local vs global

- Components need to notified of the state changes

Managing global state in React - useState

- Lift shared states to root component

- Prop drill to child components
  - Some components can be deeply nested
  - Issues?



Passing data deeply with useContext

- useContext lets a parent component make data available to any component in the tree below it
  - Think of teleporting data down to deeply nested component

- Three steps:
  1. Create the context
  2. Provide the context
  3. Use the context

- Some use cases for useContext
  1. Appearance – components need to know ‘dark’ or ‘light’ mode to render view
  2. User information – components need to know current logged in user

Get/set username using useContext

```
import { createContext, useContext, useState } from 'react';

const CurrentUserContext = createContext(null);

export default function MyApp() {
  const [currentUser, setCurrentUser] = useState(null);
  return (
    <CurrentUserContext.Provider
      value={{currentUser, setCurrentUser}}
    >
            <Form />
    </CurrentUserContext.Provider>
  );
}
```

Get/set username using useContext

```
function Form({ children }) {
  return ( … <LoginButton /> … );
}

function LoginButton() {
  const {currentUser, setCurrentUser} = useContext(CurrentUserContext);

  if (currentUser !== null) {
    return <p>You logged in as {currentUser.name}.</p>;
  }
  return (
      // code to authenticate user log in
      // set the username using setCurrentUser()
  );
}
```

Options for state management in React

- useContext, useReducer
  - When states gets complex with many handlers, consolidate using useReducer

- Redux – predictable state management for large scale application
- Simplified data flow
  1. Action: event triggers an action
  2. Dispatch: action is dispatched to the store
  3. Reducer: store passes the action and current state to the reducer, which returns a new state
  4. Update State: store updates its state with the new value
  5. Notify Components: components subscribed to the state re-render with the updated value

When to use Redux?

- Redux is useful when your app needs all of these:
  1. Caching state from a server
  2. UI state
  3. other complex data management on the client

- Not efficient for any just one of the above
- Other tools – Recoil, Zustand
  - Minimal boiler-plate code
  - Medium size application

Photo App current Model Data Handling

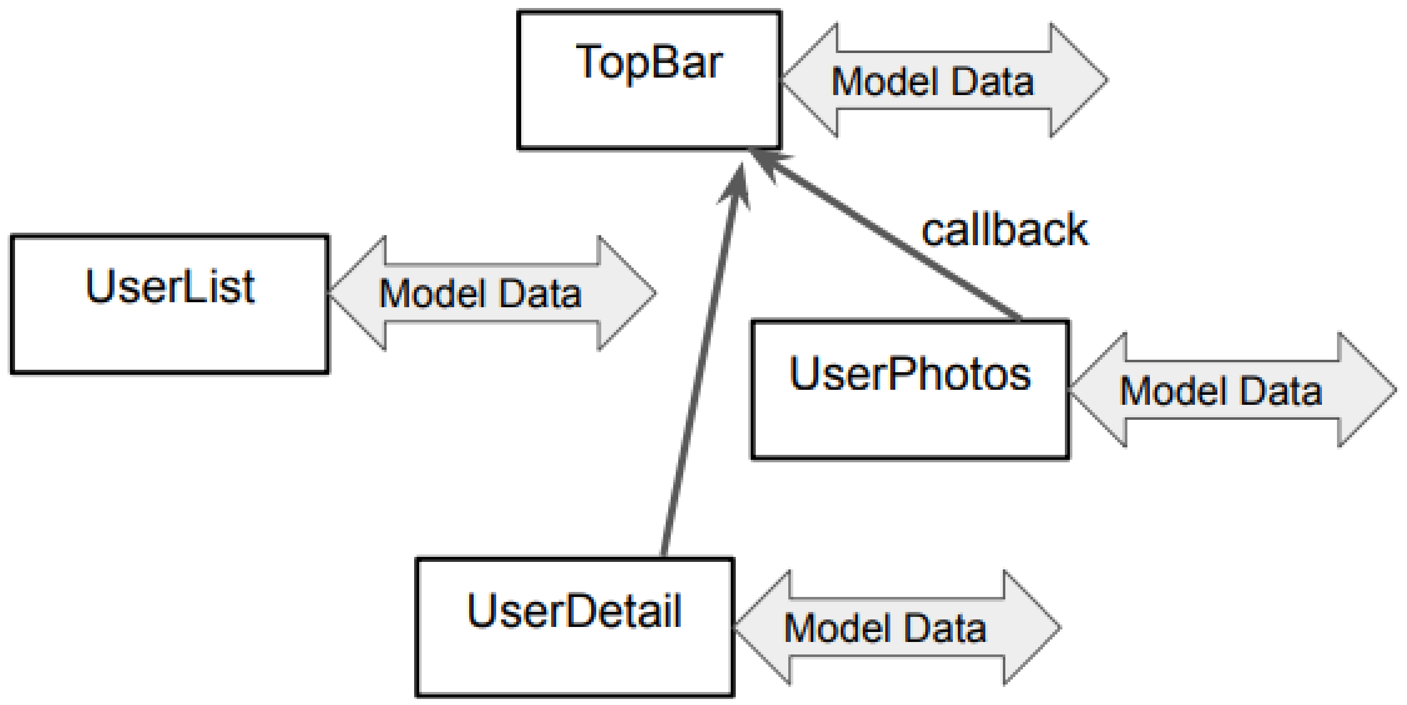
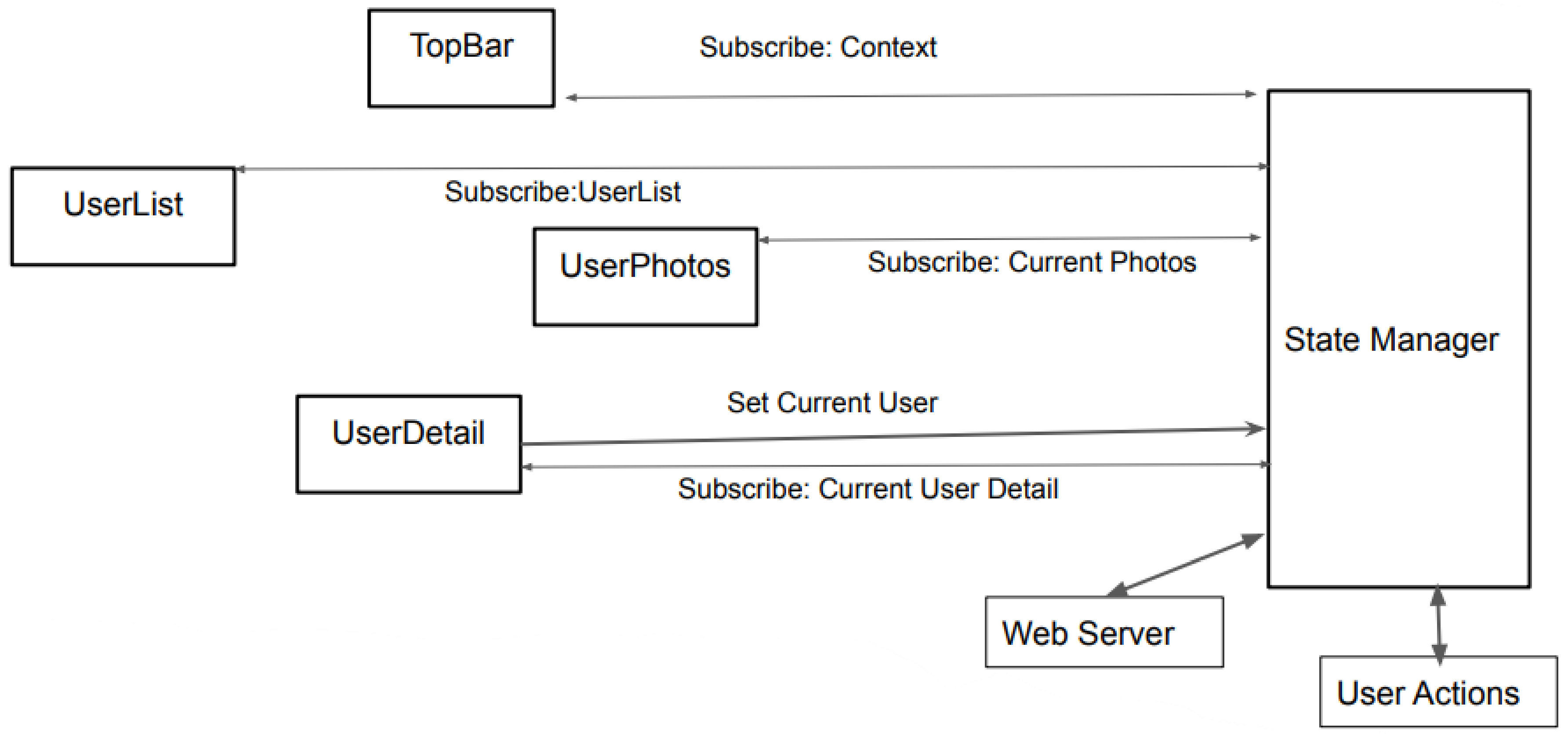


Photo App with state management



Dealing with other model changes

What happens if another user adds a photo or comment?

1. Do nothing. Easy, but not a good idea
   - Won’t see the change till there is refresh

2. Poll: periodically check for changes or just refresh the model

3. Push notifications: server pushes model changes as soon as they occur
   - User sees the changes immediately
   - Easy to implement using websockets

Photo App with sessions and input

- App needs to track who is logged in
  - Ideally held in some state store
  - OK to keep in PhotoShare component (useState, useContext)

- Redirect in router when user is not logged in

```
    userIsLoggedIn ?
            <Route path="/users/:id" component={UserDetail} />
    :
            <Redirect path="/users/:id" to="/login-register" />
```

- Need to inform component when to refresh their models
  - State management is ideal: OK to use callbacks

Sources

1. CS142 Lectures
2. React - managing state
3. Changelog - when and when not to reach for redux
4. Real Time Notification System with Node.js and WebSockets