React Context and Form Input

Embedded Components

- There will be some functionality that requires several layers of embedded components:

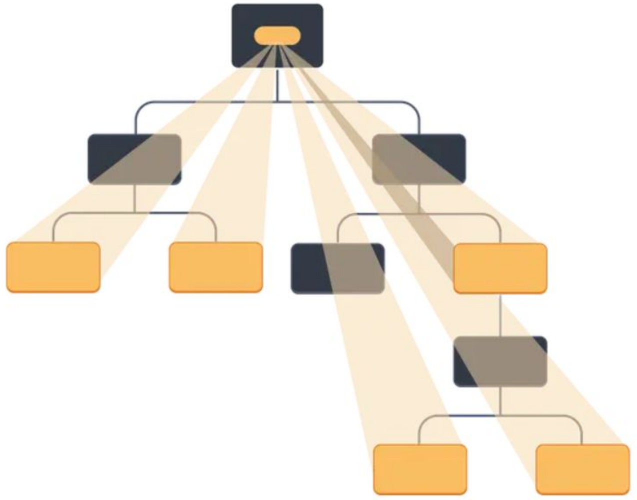
Image credit react.dev

In this case, using solely props and state variables will be an onerous exercise.

We will instead use a technique called Application Context

Context, User example

Our lecture example and the homework will focus on a User Context:

- Some site functionality is locked behind a password wall
- Other features are public, available to anyone on the internet
- For features locked behind a password wall, a component needs to know if a user is logged in
    - This component might be several layers beneath the App component, but it still needs to be aware of this information

Create a Context File

It’s good practice to encapsulate your context in a separate file:

UserContext.jsx

```
import { createContext } from 'react';

export const UserContext = createContext(null);
```

The createContext function specifies that “UserContext” will be information available to multiple components, and that its default value is null.

Configure App.jsx

We have to configure the "grandfather" component that is an ancestor of all the components who will share the data, in this case App.jsx:

```
import { useState, useEffect } from 'react';
import AppHeader from './components/AppHeader/AppHeader';
import AppFooter from './components/AppFooter';
import MainNav from './components/MainNav/MainNav';
import ViewManager from './components/ViewManager';
import { UserContext } from './context/UserContext';
import AuthService from './services/AuthService';
import axios from 'axios';

export default function App() {
  const [viewName, setViewName] = useState('LOGIN');
  const [user, setUser] = useState(null);
  // …
```

Import in the context file from the previous step

Make a state variable to hold the centralized context data

Configure App.jsx : Setup a Provider

To ensure that all descendant components have access to the user context, we use a Context Provider tag:

```
import { UserContext } from './context/UserContext';
// …
export default function App() {
    const [user, setUser] = useState(null);
// …

return (
    <div id="book-app">
      <UserContext.Provider value={user}>
                <AppHeader
                  title="Bookmark Manager"
                  onLogout={handleLogout}
                />
                <MainNav
                  viewName={viewName}
                  onNavChange={setViewName}
                />
                <ViewManager
                  viewName={viewName}
                  onLogin={handleLogin}
                />
      </UserContext.Provider>
      <AppFooter />
    </div>
```

- The “UserContext” is set to the value of the user state variable.
- AppHeader can access the UserContext (and any child components of AppHeader)
- ViewManager can access the UserContext (and any child components of ViewManager)
- AppFooter **does not** have access to the UserContext

Context Provider (this lecture) vs Just Props & State (previous lecture)

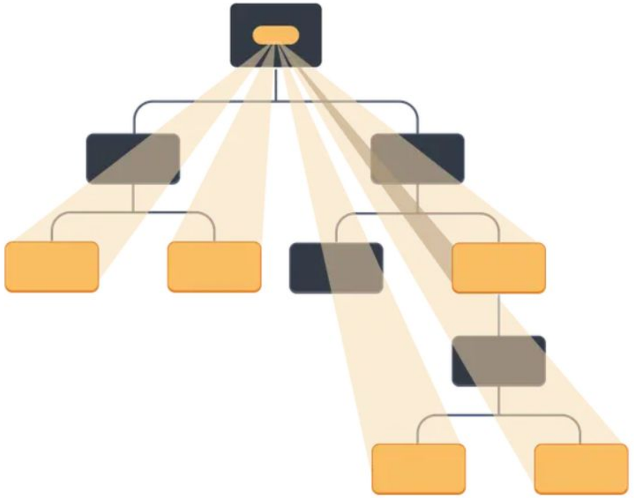
Image credit react.dev

Does your application have a user login and tracks information in a User object?
Use a User Context Provider for the user data.

Does your application only have two levels that share some data? (Parent / Child)
Just use props and state variables, like we saw in the previous lecture

Does your application need to share data across a hierarchy of components? (Parent / Child / Grandchild / Grand-Grand Child / Cousin)
Setup a Context Provider

Let’s implement a User Context provider

A bit more on Axios…

We will be using Axios again, let’s go over some extra Axios functionality before tackling Context:

```
axios.post (url, data)  .then(
    (response) => {
          console.log('Data created:', response.data);
      }
)
.catch(
    (error) => {
       console.log('Error creating data:', error);
     }
);
```

This is a **POST** request to a valid endpoint. The body of the request is captured by the variable **data**.

```
axios.put(url, updatedData).then(
   (response) => {
        console.log('Data updated:', response.data);
  }
)
.catch(
    (error) => {
        console.log('Error updating data:', error);
     }
);
```

We are now sending a **PUT** request, **updatedData** captures the body of the request

```
axios.delete(url).then(
    (response) => {
       console.log('Data deleted:', response.data);
     }
)
.catch(
   (error) => {
       console.log('Error deleting data:', error);
     }
);
```

We are now sending a **DELETE** request. DELETE requests have no body so only the endpoint’s URL is provided.

Form Processing

Let’s go over how to make a form that transmits data to our server back end:

- The form elements will be defined with JSX
- The form fields will be tied to state variables
- When the form is submitted, the state variables are collected and passed to the appropriate Axios service function

Form Processing (Storing the Form Data)

child

```
export default function ReservationForm({ reservation, setReservation, onReservationSubmit }) {
…
return (
    <form className={styles.reservationForm} onSubmit={onReservationSubmit}>
      <header>Reservation Details</header>
      <section>
        <label>Full Name:</label>
        <input type="text" value={reservation?.fullName ?? ''} onChange={(e) => setReservation( { ...reservation, fullName: e.target.value } )} />
      </section>
      <section>
        <label>Check-in Date:</label>
        <input type="date" value={reservation?.checkInDate ?? ''} onChange={(e) => setReservation( { ...reservation, checkInDate: e.target.value } )} />
      </section>
…
```

- The setReservation function, which is inherited as a prop, is called every time a field changes.
- We use the **spread operator** to update the object with the new value

parent

```
export default function AddReservation({ onSubmit }) {

const [reservation, setReservation] = useState(null);
…
return (
    <div>
      <h2>Add Reservation</h2>
      <ReservationForm
        reservation={reservation}
        setReservation={setReservation}
        onReservationSubmit={handleReservationSubmit}
      />
    </div>
  );
…
```

- Over on the parent side, notice how the setReservation function is passed in as a prop to ReservationForm
- setReservation’s purpose is to update the reservation state variable.

Form Processing (Pre-Populating form elements)

child

```
export default function ReservationForm({ reservation, setReservation, onReservationSubmit }) {
…
return (
  <form className={styles.reservationForm} onSubmit={onReservationSubmit}>
    <header>Reservation Details</header>
    <section>
      <label>Full Name:</label>
      <input type="text" value={reservation?.fullName ?? ''} onChange={(e) => setReservation( { ...reservation, fullName: e.target.value } )} />
    </section>
    <section>
      <label>Check-in Date:</label>
      <input type="date" value={reservation?.checkInDate ?? ''} onChange={(e) => setReservation( { ...reservation, checkInDate: e.target.value } )} />
    </section>
…
```

parent

```
export default function EditReservation({ reservationId, onSubmit }) {

const [reservation, setReservation] = useState(null);
…
return (
  <div>
    <h2>Add Reservation</h2>
    <ReservationForm
      reservation={reservation}
      setReservation={setReservation}
      onReservationSubmit={handleReservationSubmit}
    />
  </div>
 );
…
```

- The reservation object is passed in as a prop from the parent
- If there is a value present then populate it into the <input> tags
- **reservation?.fullName ??**
  We read this like so: is the reservation object null? If not, populate it with the fullName property, otherwise an empty string.

Note how the parent passes in the reservation object to the child's prop

Form Processing (Sending the form data to Axios)

child

```
export default function ReservationForm({ reservation, setReservation, onReservationSubmit }) {
…
return (
  <form className={styles.reservationForm} onSubmit={onReservationSubmit}>
   <header>Reservation Details</header>
   <section>
    <label>Full Name:</label>
    <input type="text" value={reservation?.fullName ?? ''} onChange={(e) => setReservation( { ...reservation, fullName: e.target.value } )} />
   </section>
   <section>
    <label>Check-in Date:</label>
    <input type="date" value={reservation?.checkInDate ?? ''} onChange={(e) => setReservation( { ...reservation, checkInDate: e.target.value } )} />
   </section>
…
```

When the form is submitted, we run the function **onReservationSubmit**, which is passed in as a prop from the parent

parent

```
export default function AddReservation({ onSubmit }) {

const [reservation, setReservation] = useState(null);

 function handleReservationSubmit(event) {
   event.preventDefault();

   ReservationService.createReservation(reservation)
    .then(() => {
     alert('Reservation created successfully!');
     onSubmit();
    })
    .catch((error) => {
     const message = error.response?.message || error.message;
     console.log('Error creating reservation:', message);
    });
 }
…
```

parent

```
…
return (
  <div>
   <h2>Add Reservation</h2>
   <ReservationForm
    reservation={reservation}
    setReservation={setReservation}
    onReservationSubmit={handleReservationSubmit}
   />
  </div>
 );
…
```

- Notice how the parent passes in its function, **handleReservationSubmit** as a prop to the child.
- When handleReservationSubmit is run, it reaches out to our Axios service to process the POST request.

Let’s implement our forms