

Lifting State Up

"Lifting state up" is a pattern in React where you move the state from a child component to a common parent component that both the child and any other components needing access to that state share. This is done to:

- Share state between components: When multiple components need to access or modify the same state, keeping it in a child component limits accessibility.
- Centralize control: By lifting state to a parent, the parent can manage and distribute the state (and updates) to its children via props.
- Avoid prop drilling: Lifting state up reduces the need to pass state through multiple layers of components that don't directly use it.

When to Use It?

- When sibling components (or components at the same level) need to share or synchronize state.
- When a child's state affects another child or the parent itself.

How It Works

1. Identify the closest common ancestor of the components that need the state.
2. Move the state (and its update logic) to that parent component.
3. Pass the state and any state-updating functions to the children via props.

Application: Task Manager with Lifting State Up

A React application where two components—a **TaskForm** (to add tasks) and a **TaskList** (to display tasks)—share a list of tasks. We'll lift the `tasks`` state up to their parent component, `App``.

```
npx create-react-app task-manager
```

```
cd task-manager
```

```
npm start
```

Code Structure

App.js: Parent component managing the state.

TaskForm.js: Child component to add tasks.

TaskList.js: Child component to display tasks.

1. src/App.js (Parent Component)

```
import React, { useState } from 'react';
import TaskForm from './TaskForm';
import TaskList from './TaskList';
import './App.css';

function App() {
  // State is lifted up to App (common parent)
  const [tasks, setTasks] = useState([]);

  // Function to add a new task, passed to TaskForm
  const addTask = (task) => {
    setTasks([...tasks, { id: Date.now(), text: task, completed: false }]);
  };

  // Function to toggle task completion, passed to TaskList
  const toggleTask = (id) => {
    setTasks(tasks.map(task =>
      task.id === id ? { ...task, completed: !task.completed } : task
    ));
  };
}
```

```

return (
  <div className="App">
    <h1>Task Manager</h1>
    /* Pass addTask function to TaskForm */
    <TaskForm addTask={addTask} />
    /* Pass tasks and toggleTask to TaskList */
    <TaskList tasks={tasks} toggleTask={toggleTask} />
  </div>
);
}

```

```
export default App;
```

Note:

- State: `tasks`` (an array of task objects) is managed in `App``.
- Functions: `addTask`` and `toggleTask`` update the state and are passed as props to the children.
- Lifting Up: Instead of `TaskForm`` or `TaskList`` managing `tasks`` locally, `App`` holds it and shares it.

2. src/TaskForm.js (Child Component)

```

import React, { useState } from 'react';

function TaskForm({ addTask }) {
  const [input, setInput] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!input.trim()) return; // Ignore empty submissions
    addTask(input); // Call the parent's addTask function
    setInput(""); // Clear input
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Add a new task"
      />
      <button type="submit">Add Task</button>
    </form>
  );
}

export default TaskForm;

```

Note:

- Role: Allows the user to input a new task.
- State: Only manages the local `input` value, not the task list.
- Props: Receives `addTask` from `App` to update the shared `tasks` state.

3. src/TaskList.js (Child Component)

```
import React from 'react';
```

```
function TaskList({ tasks, toggleTask }) {  
  return (  
    <ul>  
      {tasks.map(task => (  
        <li key={task.id} style={{ textDecoration: task.completed ? 'line-through' : 'none' }}>  
          <input  
            type="checkbox"  
            checked={task.completed}  
            onChange={() => toggleTask(task.id)} // Call parent's toggleTask  
          />  
          {task.text}  
        </li>  
      ))}  
    </ul>  
  );  
}  
  
export default TaskList;
```

Note:

- Role: Displays the list of tasks and allows toggling their completion.
- Props: Receives `tasks` (the state) and `toggleTask` (to update the state) from `App`.
- No Local State: Relies entirely on the lifted state from the parent.

4. src/App.css (Optional Styling)

```
.App {
  text-align: center;
  max-width: 600px;
  margin: 0 auto;
  padding: 20px;
}

form {
  margin-bottom: 20px;
}

input[type="text"] {
  padding: 5px;
  margin-right: 10px;
}

button {
  padding: 5px 10px;
}

ul {
  list-style: none;
  padding: 0;
}

li {
  display: flex;
  align-items: center;
  margin: 10px 0;
}

input[type="checkbox"] {
  margin-right: 10px;
}
```

How Lifting State Up Works Here

1. Before Lifting:

- `TaskForm` might have its own `tasks` state and add tasks locally.
- `TaskList` would need its own `tasks` state or prop drilling from `TaskForm`, leading to duplication or complexity.
- No way for `TaskList` to reflect `TaskForm`'s changes without manual synchronization.

2. After Lifting:

- ``tasks`` state is moved to ``App``, the common parent.
- ``TaskForm`` sends new tasks to ``App`` via ``addTask``.
- ``TaskList`` receives the updated ``tasks`` and can toggle them via ``toggleTask``.
- Both components stay in sync because ``App`` owns and updates the single source of truth (``tasks``).

Flow

- User types "Buy milk" in ``TaskForm`` and clicks "Add Task".
- ``TaskForm`` calls ``addTask('Buy milk')``.
- ``App`` updates ``tasks`` with a new task object and re-renders.
- ``TaskList`` receives the updated ``tasks`` prop and displays "Buy milk".
- User checks the checkbox in ``TaskList``, calling ``toggleTask(id)``.
- ``App`` updates the ``completed`` status, and ``TaskList`` reflects the change.

Context API

The Context API is a built-in feature in React that allows you to share state (or data) across components without having to pass props manually through every level of the component tree—a problem known as "prop drilling."

It's particularly useful when multiple components at different nesting levels need access to the same data, such as theme settings, user authentication etc.

Key Concepts

- Context: A way to create a "global" state that can be accessed by any component in the tree.
- Provider: A component that supplies the context value to its descendants.
- Consumer: A component that subscribes to the context value (though modern React uses hooks like `useContext` instead of the older Consumer syntax).

How It Works

1. Create a context using `React.createContext()`.
2. Wrap the component tree (or a part of it) with a `Provider` and pass the state/value.
3. Use the `useContext` hook (or Consumer) in any child component to access the context value.

When to Use It?

- When state needs to be shared across many components, especially if they're not direct parent-child relationships.
- As an alternative to "lifting state up" when the state needs to be accessed by deeply nested components or when prop drilling becomes cumbersome.

Redoing the Task Manager Example with Context API

Refactoring previous `Task Manager` application to use the Context API instead of lifting state up with props.

The `tasks` state and its update functions (`addTask` and `toggleTask`) will be managed in a context, accessible to `TaskForm` and `TaskList` without passing props explicitly.

1. src/TaskContext.js (Context Definition)

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

// Create the Task Context
const TaskContext = createContext();

// Custom hook to use the context (optional, for convenience)
export const useTaskContext = () => {
  const context = useContext(TaskContext);
  if (!context) {
    throw new Error('useTaskContext must be used within a TaskProvider');
  }
  return context;
};

// Task Provider component to wrap the app
export const TaskProvider = ({ children }) => {
  const [tasks, setTasks] = useState([]);

  const addTask = (task) => {
    setTasks([...tasks, { id: Date.now(), text: task, completed: false }]);
  };

  const toggleTask = (id) => {
    setTasks(tasks.map(task =>
      task.id === id ? { ...task, completed: !task.completed } : task
    ));
  };
};
```

```

// Value object to share via context
const value = {
  tasks,
  addTask,
  toggleTask,
};

return (
  <TaskContext.Provider value={value}>
    {children}
  </TaskContext.Provider>
);
};

```

- TaskContext: Created with `createContext()` to hold the shared state.
- TaskProvider: A component that manages the `tasks` state and provides it (along with `addTask` and `toggleTask`) to its children.
- useTaskContext: A custom hook to simplify consuming the context in functional components.

2. src/App.js (Using the Provider)

```

import React from 'react';
import { TaskProvider } from './TaskContext';
import TaskForm from './TaskForm';
import TaskList from './TaskList';
import './App.css';

function App() {
  return (
    <TaskProvider>
      <div className="App">
        <h1>Task Manager with Context API</h1>
        <TaskForm />
        <TaskList />
      </div>
    </TaskProvider>
  );
}

export default App;

```

Note:

Wraps the app with `TaskProvider` instead of managing state locally.

TaskForm` and `TaskList` no longer receive props from App; they'll get data from the context.

3. src/TaskForm.js (Consuming Context)

```
import React, { useState } from 'react';
import { useTaskContext } from './TaskContext';

function TaskForm() {
  const [input, setInput] = useState("");
  const { addTask } = useTaskContext(); // Access addTask from context

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!input.trim()) return;
    addTask(input);
    setInput("");
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Add a new task"
      />
      <button type="submit">Add Task</button>
    </form>
  );
}

export default TaskForm;
```

Note:

Uses `useTaskContext` to get `addTask` directly from the context instead of receiving it as a prop.

- State: Still manages its local `input` state, but relies on context for shared `tasks`.

4. src/TaskList.js (Consuming Context)

```
import React from 'react';
import { useTaskContext } from './TaskContext';

function TaskList() {
  const { tasks, toggleTask } = useTaskContext(); // Access tasks and toggleTask from context

  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id} style={{ textDecoration: task.completed ? 'line-through' : 'none' }}>
          <input
            type="checkbox"
            checked={task.completed}
            onChange={() => toggleTask(task.id)}
          />
          {task.text}
        </li>
      ))}
    </ul>
  );
}

export default TaskList;
```

Note:

Uses `useTaskContext` to get `tasks` and `toggleTask` from the context instead of props.

5. src/App.css

```
.App {
  text-align: center;
  max-width: 600px;
  margin: 0 auto;
  padding: 20px;
}

form {
  margin-bottom: 20px;
}

input[type="text"] {
  padding: 5px;
  margin-right: 10px;
}
```

```
button{
  padding: 5px 10px;
}

ul{
  list-style: none;
  padding: 0;
}

li{
  display: flex;
  align-items: center;
  margin: 10px 0;
}

input[type="checkbox"]{
  margin-right: 10px;
}
```

How It Works with Context API

1. State Management: The ``TaskProvider`` in ``TaskContext.js`` holds the ``tasks`` state and update functions (``addTask``, ``toggleTask``).
2. Context Provider: ``App`` wraps its children with ``<TaskProvider>``, making the context value available to all descendants.
3. Context Consumption: ``TaskForm`` and ``TaskList`` use ``useTaskContext`` to access the shared state and functions directly, bypassing prop passing.

Flow

- User types "Buy milk" in ``TaskForm`` and submits.
- ``TaskForm`` calls ``addTask('Buy milk')`` from the context.
- ``TaskProvider`` updates ``tasks`` and re-renders its children.
- ``TaskList`` sees the updated ``tasks`` from the context and displays "Buy milk".
- User toggles the checkbox, calling ``toggleTask(id)`` from the context.
- ``TaskProvider`` updates the ``completed`` status, and ``TaskList`` reflects it.

Comparison: Lifting State Up vs. Context API

Aspect	Lifting State Up	Context API	
-----	-----	-----	
State Location	In the parent component (e.g., `App`).	In a context provider (e.g., `TaskProvider`).	
Access Method	Passed via props to children.	Accessed via `useContext` anywhere in the tree.	
Prop Drilling	Required if children are deeply nested.	Avoided; context is global within provider scope.	
Scalability	Works well for small apps; cumbersome for large ones.	Better for larger apps with many consumers.	
Setup Complexity	Simpler, no extra API needed.	Requires context creation and provider setup.	