TaskList App - React Documentation

This document provides a comprehensive guide to the TaskList React application. It covers React hooks, localStorage, JSON, state management, and walks through each component with detailed explanations of the code's logic and connections.

Table of Contents

Introduction to React Hooks

React Hooks are functions that let you "hook into" React state and lifecycle features from function components. They were introduced in React 16.8 to allow developers to use state and other React features without writing class components.

useState Hook

The `useState` hook allows functional components to manage state. It returns a pair: the current state value and a function to update it.

Basic Syntax:

```
const [state, setState] = useState(initialValue);
```

Example:

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

function Counter() {
  // Declare a state variable called "count" with initial value 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

In this example:

- `count` is the current state value
- `setCount` is the function to update the state
- `0` is the initial value for the state

useEffect Hook

The `useEffect` hook lets you perform side effects in functional components. It runs after every completed render by default, but can be configured to run only when certain values change.

Basic Syntax:

```
useEffect(() => {
  // Side effect code

  // Optional cleanup function
  return () => {
    // Cleanup code
  };
}, [dependencies]); // Dependencies array
```

Example:

```
import { useState, useEffect } from 'react';
```

```
function WindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    // Set up event listener when component mounts
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener('resize', handleResize);

    // Clean up event listener when component unmounts
    return () => {
      window.removeEventListener('resize', handleResize);
    };
  }, []); // Empty dependency array means this runs once on mount

  return <p>Window width: {width}px</p>;
}
```

In this example:

- The effect runs once when the component mounts
- It adds an event listener for window resizing
- When the component unmounts, the cleanup function removes the event listener

Working with localStorage

localStorage is a web API that allows you to store key-value pairs in the browser. It's commonly used with React to persist state between page reloads.

Basic Usage:

```
// Store data
localStorage.setItem('key', 'value');

// Retrieve data
const value = localStorage.getItem('key');

// Remove data
localStorage.removeItem('key');

// Clear all data
localStorage.clear();
```

Using localStorage with JSON:

Since localStorage only stores strings, we need to convert objects to JSON strings before storing them and parse them when retrieving:

```
// Storing an object
const user = { name: 'John', age: 30 };
localStorage.setItem('user', JSON.stringify(user));

// Retrieving the object
const storedUser = JSON.parse(localStorage.getItem('user'));
```

Example with React Hooks:

```
import { useState, useEffect } from 'react';

function PersistentCounter() {
  // Initialize state from localStorage or default to 0
  const [count, setCount] = useState(() => {
    const savedCount = localStorage.getItem('count');
    return savedCount ? parseInt(savedCount, 10) : 0;
  });

  // Update localStorage when count changes
  useEffect(() => {
    localStorage.setItem('count', count.toString());
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

This counter will persist its value even if the page is refreshed.

TaskList App Architecture

The TaskList app is a simple todo application with the following features:

- Add new tasks
- Edit existing tasks
- Delete tasks

- Clear all tasks
- Change theme

The application is composed of four main components:

1. **App**: The parent component that manages the main state
2. **Header**: Displays the app logo and theme selector
3. **AddTask**: Form for adding and updating tasks
4. **ShowTask**: List displaying all tasks with edit and delete functionality

Data Flow Diagram

```
App Component (Main State)
├── Header Component (Theme State)
├── AddTask Component (Task Input)
└── ShowTask Component (Task Display)
```

Component Breakdown

App Component

The App component is the main container for our application. It manages the primary state (taskList and current task) and passes data to child components.

```
import { useState, useEffect } from "react";
import AddTask from "./Components/AddTask";
import Header from "./Components/Header";
import ShowTask from "./Components/ShowTask";
import "./App.css";

export default function App() {
    const [taskList, setTaskList] = useState(
        JSON.parse(localStorage.getItem("taskList")) || []
    );

    const [task, setTask] = useState({});

    useEffect(() => {
        localStorage.setItem("taskList", JSON.stringify(taskList));
    }, [taskList]);

    return (
        <div className="App">
```

```
            <Header/>
            <AddTask taskList={taskList} setTaskList={setTaskList} task={task} setTask={setTask}/
            <ShowTask taskList={taskList} setTaskList={setTaskList} task={task} setTask={setTask}
        </div>
    );
}
```

Code Explanation:

1. **State Initialization**:
   - `taskList` : Array of tasks retrieved from localStorage (or empty array if none exists)
   - `task` : Object representing the currently selected task (empty by default)
2. **useEffect Hook**:
   - Runs whenever `taskList` changes
   - Stores the updated `taskList` in localStorage as a JSON string
   - This ensures task persistence between browser sessions
3. **Component Rendering**:
   - Renders the Header, AddTask, and ShowTask components
   - Passes state variables and setter functions as props to child components

Header Component

The Header component displays the application logo and provides theme switching functionality.

```
import img from '../assets/react.svg';
import { useState, useEffect } from 'react';

export default function Header() {
    const [theme, setTheme] = useState("gTwo");

    useEffect(() => {
        document.documentElement.className = theme;
    }, [theme]);

    return (
        <header>
            <div className="logo">
                <img src={img} alt="logo" />
                <h1>TaskList</h1>
            </div>
```

```
            <div className="themeSelector">
                <span
                    className={`light ${theme === "light" ? "activeTheme" : ""}`}
                    onClick={() => setTheme("light")}
                ></span>
                <span
                    className={`medium ${theme === "medium" ? "activeTheme" : ""}`}
                    onClick={() => setTheme("medium")}
                ></span>
                <span
                    className={`dark ${theme === "dark" ? "activeTheme" : ""}`}
                    onClick={() => setTheme("dark")}
                ></span>
                <span
                    className={`gOne ${theme === "gOne" ? "activeTheme" : ""}`}
                    onClick={() => setTheme("gOne")}
                ></span>
                <span
                    className={`gTwo ${theme === "gTwo" ? "activeTheme" : ""}`}
                    onClick={() => setTheme("gTwo")}
                ></span>
                <span
                    className={`gThree ${theme === "gThree" ? "activeTheme" : ""}`}
                    onClick={() => setTheme("gThree")}
                ></span>
            </div>
        </header>
    );
}
```

Code Explanation:

1. **State Management**:
   - `theme` : Stores the current theme name (default is "gTwo")
2. **useEffect Hook**:
   - Applies the selected theme by setting a class on the document's root element
   - Runs whenever the `theme` state changes
3. **Theme Selection**:
   - Renders colored boxes that users can click to change the theme
   - Applies the "activeTheme" class to highlight the currently selected theme
   - Each theme option has an onClick handler that calls `setTheme` with the appropriate value

AddTask Component

The AddTask component provides a form for adding new tasks or updating existing ones.

```
export default function AddTask({taskList, setTaskList, task, setTask}) {
  function handleSubmit(e) {
    e.preventDefault();

    if (task.id) {
      const date = new Date();
      const updatedTaskList = taskList.map((todo) =>
        todo.id === task.id ? {
          id: task.id,
          name: task.name,
          time: `${date.toLocaleTimeString()} ${date.toLocaleDateString()}`
        } : todo
      );
      setTaskList(updatedTaskList);

    } else {
      const date = new Date();
      const newTask = {
        id: date.getTime(),
        name: e.target.task.value,
        time: `${date.toLocaleTimeString()} ${date.toLocaleDateString()}`
      };
      setTaskList([...taskList, newTask]);

    }
    setTask({});
  }

  return (
    <section className="addTask">
      <form onSubmit={handleSubmit}>
        <input
          onChange={(e) => setTask({...task, name: e.target.value})}
          value={task.name || ""}
          type="text"
          placeholder="Add Task"
          name="task"
          autoComplete="off"
          maxLength={25}
        />
        <button type="submit">{task.id ? "Update" : "Add"}</button>
      </form>
    </section>
```

```
    );
}
```

Code Explanation:

1. **Form Handling**:
   - `handleSubmit` : Processes form submission for both new tasks and updates
   - The function prevents the default form submission behavior with `e.preventDefault()`
2. **Task Update Logic**:
   - Checks if the current task has an ID (indicating it's an existing task being edited)
   - If editing: Maps through the task list and replaces the matching task with updated values
   - If adding new: Creates a new task object with a timestamp ID and adds it to the task list
   - Both operations update the timestamp to the current date and time
3. **Input Field**:
   - Value is bound to `task.name` (or empty string if undefined)
   - `onChange` handler updates the `task` state with each keystroke
   - The spread operator (`...task`) preserves existing task properties when updating
4. **Dynamic Button Text**:
   - Button displays "Update" when editing an existing task
   - Button displays "Add" when creating a new task

ShowTask Component

The ShowTask component displays the list of tasks and provides functionality to edit or delete them.

```
export default function ShowTask({taskList, setTaskList, task, setTask}) {
  function handleEdit(id) {
      const selectedTask = taskList.find((todo) => todo.id === id);
      setTask(selectedTask);
  }

  function handleDelete(id) {
      const updatedTaskList = taskList.filter((todo) => todo.id !== id);
      setTaskList(updatedTaskList);
  }

  return (
```

```
        <section className="showTask">
            <div className="head">
                <div>
                    <span className="title">Todo</span>
                    <span className="count">{taskList.length}</span>
                </div>
                <button className="clearAll" onClick={() => setTaskList([])}>Clear All</button>
            </div>

            <ul>
                {taskList.map((todo) => (
                    <li key={todo.id}>
                        <p>
                            <span className="name">{todo.name}</span>
                            <span className="time">{todo.time}</span>
                        </p>
                        <i onClick={() => handleEdit(todo.id)} className="bi bi-pencil-square"></i>
                        <i onClick={() => handleDelete(todo.id)} className="bi bi-trash"></i>
                    </li>
                ))}
            </ul>
        </section>
    );
}
```

Code Explanation:

1. **Task Editing**:
    - `handleEdit` : Finds the selected task by ID and sets it as the current task
    - This populates the form in the AddTask component for editing
2. **Task Deletion**:
    - `handleDelete` : Filters out the task with the specified ID from the task list
    - Uses the array `filter` method to create a new array without the deleted task
3. **Task List Display**:
    - Shows a counter with the total number of tasks
    - Maps through the task list to render each task as a list item
    - Each item displays the task name and timestamp
    - Icons provide edit and delete functionality
4. **Clear All Functionality**:

- Button that resets the task list to an empty array when clicked

Data Flow and State Management

The TaskList app demonstrates React's unidirectional data flow pattern. Here's how data flows through the application:

1. **Main State Management**:
   - The `App` component maintains the primary state (`taskList` and `task`)
   - This state is passed down to child components as props
2. **State Updates**:
   - Child components receive setter functions (`setTaskList` and `setTask`)
   - These functions allow child components to update the parent's state
   - When state changes in the parent, React re-renders affected components
3. **Data Persistence**:
   - The `useEffect` hook in the App component syncs the `taskList` state with localStorage
   - This ensures data persistence between page reloads
4. **Component Communication**:
   - When a user clicks the edit button in `ShowTask` , it calls `setTask` with the selected task
   - This updates the `task` state in the parent `App` component
   - The updated `task` is passed to `AddTask` , which populates its form for editing

Implementing Advanced State Management

For larger applications, you might want to use a more structured state management solution like Redux. Below is an example of how you could implement Redux in this application.

Redux Implementation Example

First, let's understand the core Redux concepts:

- **Store**: Central state container
- **Actions**: Objects describing what happened
- **Reducers**: Functions that determine how state changes in response to actions
- **Dispatch**: Method to send actions to the store

Here's how we could implement Redux for the TaskList app:

1. Install Required Packages:

```
npm install redux react-redux @reduxjs/toolkit
```

2. Create Task Slice (using Redux Toolkit):

```
// taskSlice.js
import { createSlice } from '@reduxjs/toolkit';

// Load initial state from localStorage
const getInitialState = () => {
  const savedTasks = localStorage.getItem('taskList');
  return savedTasks ? JSON.parse(savedTasks) : [];
};

const taskSlice = createSlice({
  name: 'tasks',
  initialState: {
    taskList: getInitialState(),
    currentTask: {}
  },
  reducers: {
    addTask: (state, action) => {
      const date = new Date();
      const newTask = {
        id: date.getTime(),
        name: action.payload,
        time: `${date.toLocaleTimeString()} ${date.toLocaleDateString()}`
      };
      state.taskList.push(newTask);
      localStorage.setItem('taskList', JSON.stringify(state.taskList));
    },
    updateTask: (state, action) => {
      const date = new Date();
      state.taskList = state.taskList.map(task =>
        task.id === action.payload.id ? {
          ...task,
          name: action.payload.name,
          time: `${date.toLocaleTimeString()} ${date.toLocaleDateString()}`
        } : task
      );
      state.currentTask = {};
      localStorage.setItem('taskList', JSON.stringify(state.taskList));
    },
    deleteTask: (state, action) => {
```

```
      state.taskList = state.taskList.filter(task => task.id !== action.payload);
      localStorage.setItem('taskList', JSON.stringify(state.taskList));
    },
    clearTasks: (state) => {
      state.taskList = [];
      localStorage.setItem('taskList', JSON.stringify(state.taskList));
    },
    setCurrentTask: (state, action) => {
      state.currentTask = action.payload;
    }
  }
});

export const { addTask, updateTask, deleteTask, clearTasks, setCurrentTask } = taskSlice.actions;
export default taskSlice.reducer;
```

3. Create Redux Store:

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import taskReducer from './taskSlice';
import themeReducer from './themeSlice';

export const store = configureStore({
  reducer: {
    tasks: taskReducer,
    theme: themeReducer
  }
});
```

4. Connect Redux to React:

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Provider } from 'react-redux';
import { store } from './store';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
```

```
    </React.StrictMode>
  );
```

5. Update Components to Use Redux:

App Component:

```
import AddTask from "./Components/AddTask";
import Header from "./Components/Header";
import ShowTask from "./Components/ShowTask";
import "./App.css";

export default function App() {
    return (
        <div className="App">
            <Header/>
            <AddTask/>
            <ShowTask/>
        </div>
    );
}
```

AddTask Component with Redux:

```
import { useSelector, useDispatch } from 'react-redux';
import { addTask, updateTask, setCurrentTask } from '../taskSlice';

export default function AddTask() {
  const dispatch = useDispatch();
  const currentTask = useSelector(state => state.tasks.currentTask);

  function handleSubmit(e) {
    e.preventDefault();

    if (currentTask.id) {
      dispatch(updateTask({
        id: currentTask.id,
        name: currentTask.name
      }));
    } else {
      dispatch(addTask(e.target.task.value));
    }

    e.target.reset();
  }
```

```
    return (
      <section className="addTask">
        <form onSubmit={handleSubmit}>
          <input
            onChange={(e) => dispatch(setCurrentTask({...currentTask, name: e.target.value}))}
            value={currentTask.name || ""}
            type="text"
            placeholder="Add Task"
            name="task"
            autoComplete="off"
            maxLength={25}
          />
          <button type="submit">{currentTask.id ? "Update" : "Add"}</button>
        </form>
      </section>
    );
  }
```

ShowTask Component with Redux:

```
  import { useSelector, useDispatch } from 'react-redux';
  import { deleteTask, setCurrentTask, clearTasks } from '../taskSlice';

  export default function ShowTask() {
    const dispatch = useDispatch();
    const taskList = useSelector(state => state.tasks.taskList);

    function handleEdit(id) {
      const selectedTask = taskList.find(task => task.id === id);
      dispatch(setCurrentTask(selectedTask));
    }

    function handleDelete(id) {
      dispatch(deleteTask(id));
    }

    return (
        <section className="showTask">
            <div className="head">
                <div>
                    <span className="title">Todo</span>
                    <span className="count">{taskList.length}</span>
                </div>
                <button className="clearAll" onClick={() => dispatch(clearTasks())}>Clear All</butt
            </div>

            <ul>
                {taskList.map((todo) => (
```

```
                    <li key={todo.id}>
                        <p>
                            <span className="name">{todo.name}</span>
                            <span className="time">{todo.time}</span>
                        </p>
                        <i onClick={() => handleEdit(todo.id)} className="bi bi-pencil-square"></i>
                        <i onClick={() => handleDelete(todo.id)} className="bi bi-trash"></i>
                    </li>
                ))}
            </ul>
        </section>
    );
}
```

This Redux implementation centralizes state management and makes data flow more predictable, which can be beneficial as applications grow larger and more complex.

This documentation provides a comprehensive explanation of the TaskList app, covering React hooks, localStorage, state management, and the logical connections between components. By understanding these concepts and how they work together, even those with limited coding knowledge can grasp the architecture and functionality of the application.