Advanced React Concepts

Table of Contents

1. React Hooks
    - useState
    - useEffect
2. React Router
3. Context API
4. Redux

React Hooks

React Hooks represent a revolutionary addition to React's ecosystem, introduced in React 16.8. At their core, Hooks are special functions that allow you to "hook into" React's internal features from functional components. They were created to solve several key challenges that developers faced with class components and to provide a more intuitive way to work with React's features.

Understanding React Hooks

What are Hooks? Hooks are JavaScript functions that establish a connection between a functional component and React's internal features. They provide a direct line to React's state management, lifecycle methods, and other core functionalities without requiring class components. Think of them as connection points that let you tap into React's power system.

Problems Solved by Hooks Before Hooks, developers faced several challenges:

1. Reusing stateful logic between components was difficult and often led to wrapper hell through render props and higher-order components
2. Complex components became hard to understand due to related logic being split across different lifecycle methods
3. Classes confused both people and machines - they were harder to optimize and made hot reloading less reliable
4. The `this` keyword in JavaScript classes was a constant source of confusion

Hooks solved these problems by:

- Allowing state management in functional components
- Providing a way to extract and reuse stateful logic without changing component hierarchy
- Grouping related logic together in a single place
- Making it possible to use React features without classes

How React Handles Hooks Internally React maintains an internal state for each component using a concept called the "fiber tree". When you use Hooks:

1. React creates a queue of Hook calls for each component
2. Each Hook call gets its own "memory cell" in the component
3. The order of Hook calls must be consistent between renders

4. React matches each Hook call to its corresponding state by maintaining a counter during rendering

This is why Hooks must be called:

- Only at the top level of your function
- In the same order every time
- Only within React functional components or custom Hooks

useState

The `useState` hook allows functional components to manage state.

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

function UserProfile() {
  const [name, setName] = useState('Vinod');
  const [city, setCity] = useState('Bangalore');

  return (
    <div>
      <h2>User Profile</h2>
      <p>Name: {name}</p>
      <p>City: {city}</p>
      <button onClick={() => setName('Shyam')}>Update Name</button>
      <button onClick={() => setCity('Shivamogga')}>Update City</button>
    </div>
  );
}
```

When React processes this component:

1. It allocates a memory cell for each state variable
2. Returns a pair of values: the current state and a function to update it
3. Preserves the state between re-renders
4. Triggers a re-render when the state updating function is called

useEffect

The `useEffect` hook handles side effects in functional components. It serves as a replacement for lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` combined into a single API.

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

function WeatherDashboard() {
  const [temperature, setTemperature] = useState(null);
  const [city] = useState('Bangalore');

  useEffect(() => {
    // This effect runs when the component mounts or city changes
    const fetchWeather = async () => {
      // Simulated API call
      const response = await fetch(`/api/weather/${city}`);
      const data = await response.json();
      setTemperature(data.temperature);
    };

    fetchWeather();

    // Cleanup function (equivalent to componentWillUnmount)
    return () => {
      // Cleanup code here (if needed)
    };
  }, [city]); // Dependency array

  return (
    <div>
      <h2>Weather in {city}</h2>
      {temperature ? (
        <p>Current temperature: {temperature}°C</p>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
}
```

React processes `useEffect` by:

1. Scheduling the effect to run after the render is committed to the screen
2. Checking the dependency array to determine if the effect should re-run
3. Running any cleanup function from the previous effect before running the new effect
4. Maintaining a separate effects list for each component

React Router

React Router is the standard routing library for React applications that enables dynamic, client-side routing. It allows you to build single-page applications (SPAs) with navigation without the page refreshing as the user navigates.

Understanding React Router

What is React Router? React Router is a collection of navigational components that compose declaratively with your application. It provides a way to map URLs to components, handle navigation state, and render different components based on the URL.

Key Features and Benefits:

1. **Dynamic Routing**: Routes are rendered as regular React components, allowing for dynamic route matching

2. **Nested Routes**: Support for complex application layouts with nested routes
3. **Route Parameters**: Easy handling of dynamic URL parameters
4. **Navigation State**: Built-in history management and URL synchronization
5. **Client-Side Routing**: Smooth transitions without full page reloads

Core Components

1. **BrowserRouter**:
    - Provides routing functionality using HTML5 history API
    - Keeps UI in sync with the URL
2. **Routes and Route**:
    - `Routes`: The container for all route definitions
    - `Route`: Defines the mapping between URL paths and components
3. **Link and NavLink**:
    - `Link`: Creates navigation links
    - `NavLink`: Special version of Link that can be styled when active

Implementation Example

```
import {
  BrowserRouter,
  Routes,
  Route,
  Link,
  NavLink,
  useParams,
  useNavigate
} from 'react-router-dom';

// Main App Component
function App() {
  return (
    <BrowserRouter>
      <div>
        <nav>
          {/* Navigation with styled active links */}
          <NavLink
            to="/"
            style={({ isActive }) => ({
              color: isActive ? '#007bff' : '#000'
            })}
          >
            Home
          </NavLink>
          <NavLink to="/users">Users</NavLink>
          <NavLink to="/about">About</NavLink>
        </nav>
```

```
          <Routes>
            <Route path="/" element={<Home />} />
            <Route path="/about" element={<About />} />
            {/* Nested Routes Example */}
            <Route path="/users" element={<Users />}>
              <Route index element={<UsersList />} />
              <Route path=":id" element={<UserDetails />} />
            </Route>
            {/* 404 Route */}
            <Route path="*" element={<NotFound />} />
          </Routes>
        </div>
      </BrowserRouter>
    );
}

// Example of a component using route parameters
function UserDetails() {
  const { id } = useParams();
  const navigate = useNavigate();
  const [user, setUser] = useState(null);

  useEffect(() => {
    // Example: Fetch user data based on ID
    const fetchUser = async () => {
      try {
        // Simulated API call
        const response = await fetch(`/api/users/${id}`);
        const data = await response.json();
        setUser(data);
      } catch (error) {
        // Handle error and redirect to users list
        navigate('/users');
      }
    };

    fetchUser();
  }, [id, navigate]);

  if (!user) return <div>Loading...</div>;

  return (
    <div>
      <h2>User Details</h2>
      <p>Name: {user.name}</p>
      <p>City: {user.city}</p>
      <button onClick={() => navigate(-1)}>Go Back</button>
    </div>
  );
}

// Example of a protected route
function PrivateRoute({ children }) {
  const isAuthenticated = useAuth(); // Custom hook for auth state

  return isAuthenticated ? children : <Navigate to="/login" />;
}
```

Advanced Features

1. **Route Parameters**:
   - Access URL parameters using `useParams` hook
   - Example: `/users/:id` can match `/users/123`
2. **Programmatic Navigation**:
   - Use `useNavigate` hook for programmatic routing
   - Navigate forward, backward, or to specific routes
3. **Protected Routes**:
   - Implement authentication-based routing
   - Redirect unauthorized users
4. **Route Configuration**:

```
// Example of route configuration object
const routes = [
  {
    path: '/',
    element: <Home />,
  },
  {
    path: 'users',
    element: <Users />,
    children: [
      { path: ':id', element: <UserDetails /> },
    ],
  },
];
```

Best Practices

1. **Organized Route Structure**:
   - Keep routes organized in a logical hierarchy
   - Use nested routes for related content
2. **Loading States**:
   - Handle loading states during navigation
   - Provide feedback during data fetching

3. **Error Boundaries**:
   - Implement error boundaries for route components
   - Handle 404 cases with catch-all routes
4. **Code Splitting**:
   - Use lazy loading for route components
   - Improve initial load performance

Context API

The Context API is a powerful feature in React that provides a way to share values between components without explicitly passing props through every level of the component tree. It's designed to solve the problem of prop drilling and share data that can be considered "global" for a tree of React components.

Understanding Context API

What is Context? Context provides a way to pass data through the component tree without having to pass props manually at every level. Think of it as a broadcast system where a parent component can send data that any child component can tune into, regardless of how deep it is in the component tree.

When to Use Context:

- Theme data (dark/light mode)
- User authentication data
- Language preferences
- Global UI state
- Any data that needs to be accessed by many components at different nesting levels

Benefits of Context:

1. Eliminates prop drilling
2. Makes code more maintainable
3. Centralizes state management
4. Simplifies component composition

Implementation Patterns

Basic Context Setup

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

// 1. Create Context
const UserContext = createContext();

// 2. Create Provider Component
function UserProvider({ children }) {
  const [user, setUser] = useState({
    name: 'Vinod',
    city: 'Bangalore',
    preferences: {
      theme: 'light',
      language: 'en'
    }
  });

  // Value object can contain both state and functions
  const value = {
    user,
    setUser,
    updatePreference: (key, value) => {
      setUser(prev => ({
        ...prev,
        preferences: {
          ...prev.preferences,
          [key]: value
        }
      }));
    }
  };

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

// 3. Custom Hook for Using Context
function useUser() {
  const context = useContext(UserContext);
  if (context === undefined) {
    throw new Error('useUser must be used within a UserProvider');
  }
  return context;
}
```

Using Context in Components

```
// Parent App Component
function App() {
  return (
    <UserProvider>
      <Header />
      <MainContent />
      <Footer />
    </UserProvider>
  );
}

// Deeply Nested Component
```

```
function UserProfile() {
  const { user, setUser } = useUser();

  const handleUpdateUser = () => {
    setUser({
      name: 'Shyam',
      city: 'Shivamogga',
      preferences: user.preferences
    });
  };

  return (
    <div>
      <h3>User Profile</h3>
      <p>Name: {user.name}</p>
      <p>City: {user.city}</p>
      <button onClick={handleUpdateUser}>
        Update User
      </button>
    </div>
  );
}

// Another Component Using Context
function ThemeToggle() {
  const { user, updatePreference } = useUser();
  const currentTheme = user.preferences.theme;

  return (
    <button
      onClick={() => updatePreference('theme', currentTheme === 'light' ? 'dark' :
'light')}
    >
      Switch to {currentTheme === 'light' ? 'Dark' : 'Light'} Mode
    </button>
  );
}
```

Advanced Patterns

Multiple Contexts

```
// Separate contexts for different concerns
const ThemeContext = createContext();
const AuthContext = createContext();

function AppProviders({ children }) {
  return (
    <ThemeContext.Provider value={/* theme values */}>
      <AuthContext.Provider value={/* auth values */}>
        {children}
      </AuthContext.Provider>
    </ThemeContext.Provider>
  );
}
```

Context with Reducer

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

const UserStateContext = createContext();
const UserDispatchContext = createContext();

const userReducer = (state, action) => {
  switch (action.type) {
    case 'UPDATE_USER':
      return {
        ...state,
        ...action.payload
      };
    case 'UPDATE_PREFERENCE':
      return {
        ...state,
        preferences: {
          ...state.preferences,
          [action.payload.key]: action.payload.value
        }
      };
    default:
      throw new Error(`Unhandled action type: ${action.type}`);
  }
};

function UserProvider({ children }) {
  const [state, dispatch] = useReducer(userReducer, {
    name: 'Vinod',
    city: 'Bangalore',
    preferences: {
      theme: 'light',
      language: 'en'
    }
  });

  return (
    <UserStateContext.Provider value={state}>
      <UserDispatchContext.Provider value={dispatch}>
        {children}
      </UserDispatchContext.Provider>
    </UserStateContext.Provider>
  );
}
```

Best Practices

1. **Context Organization**:
    - Split contexts by domain/concern
    - Keep context values focused and specific
    - Consider performance implications
2. **Performance Optimization**:

```
// Memoize context values
const value = useMemo(() => ({
  user,
  setUser
}), [user]);
```

3. **Error Handling**:

```
function useUser() {
  const context = useContext(UserContext);
  if (context === undefined) {
    throw new Error('useUser must be used within a UserProvider');
  }
  return context;
}
```

4. **Testing Considerations**:

```
// Wrapper for testing components that use context
const TestWrapper = ({ children }) => (
  <UserProvider>
    {children}
  </UserProvider>
);
```

Common Pitfalls to Avoid

1. **Over-using Context**:
    - Don't use context for values that could be passed as props
    - Consider component composition before reaching for context
2. **Context Value Structure**:
    - Keep context values simple and focused
    - Split complex contexts into smaller, more focused ones

3. **Performance Issues**:
    - Avoid putting too much data in a single context
    - Use context splitting and memoization when needed

Redux

Redux is a predictable state container for JavaScript applications that helps manage global application state in a consistent and maintainable way. It implements a unidirectional data flow pattern and is commonly used with React, although it can be used with any UI library.

Understanding Redux

What is Redux? Redux provides a centralized store for state management that follows strict rules to ensure predictable state updates. It's particularly useful for applications with complex state logic or when many components need to access and modify the same state.

Core Principles:

1. **Single Source of Truth**: The entire application state is stored in a single store
2. **State is Read-Only**: State can only be modified through dispatched actions
3. **Changes are Made with Pure Functions**: Reducers must be pure functions

When to Use Redux:

- Complex state logic across multiple components
- Large-scale applications
- Frequent state updates
- Need for state persistence
- Complex data flows
- Team collaboration on state management

Core Concepts

1. Store

The store is the heart of Redux, holding the complete state tree of your application.

```
import { configureStore } from '@reduxjs/toolkit';

const store = configureStore({
  reducer: {
    user: userReducer,
    preferences: preferencesReducer
  }
});
```

2. Actions

Actions are plain JavaScript objects that describe what happened in your application.

```
// Action Types
const UPDATE_USER = 'user/updateUser';
const UPDATE_PREFERENCES = 'user/updatePreferences';

// Action Creators
const updateUser = (name, city) => ({
  type: UPDATE_USER,
  payload: { name, city }
});

const updatePreferences = (preferences) => ({
  type: UPDATE_PREFERENCES,
  payload: preferences
});

// Async Action Creator (using Redux Thunk)
const fetchUserData = (userId) => {
  return async (dispatch) => {
    try {
      const response = await fetch(`/api/users/${userId}`);
      const userData = await response.json();
      dispatch(updateUser(userData.name, userData.city));
    } catch (error) {
      dispatch(setError(error.message));
    }
  };
};
```

3. Reducers

Reducers specify how the application's state changes in response to actions.

```
// Initial State
const initialState = {
  user: {
    name: 'Vinod',
    city: 'Bangalore'
  },
  preferences: {
    theme: 'light',
    language: 'en'
  },
  loading: false,
  error: null
};

// User Reducer
const userReducer = (state = initialState, action) => {
  switch (action.type) {
    case UPDATE_USER:
      return {
        ...state,
        user: {
          ...state.user,
          ...action.payload
        }
      };
    case UPDATE_PREFERENCES:
      return {
        ...state,
        preferences: {
          ...state.preferences,
          ...action.payload
        }
      };
    default:
      return state;
  }
};
```

Implementation Patterns

Basic Redux Setup with React

```
import { Provider, useSelector, useDispatch } from 'react-redux';
import { configureStore } from '@reduxjs/toolkit';

// Root Component
function App() {
  return (
    <Provider store={store}>
      <div>
        <UserProfile />
        <PreferencesPanel />
      </div>
    </Provider>
  );
}

// Component using Redux
function UserProfile() {
  const dispatch = useDispatch();
  const user = useSelector(state => state.user);
  const loading = useSelector(state => state.loading);

  const handleUpdateUser = () => {
    dispatch(updateUser('Shyam', 'Shivamogga'));
  };
  if (loading) return <div>Loading...</div>;

  return (
    <div>
      <h2>User Profile</h2>
      <p>Name: {user.name}</p>
      <p>City: {user.city}</p>
      <button onClick={handleUpdateUser}>
        Update User
      </button>
    </div>
  );
}
```

Using Redux Toolkit (Modern Redux)

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

// Create async thunk
export const fetchUserData = createAsyncThunk(
  'user/fetchUserData',
  async (userId) => {
    const response = await fetch(`/api/users/${userId}`);
    return response.json();
  }
);

// Create slice
const userSlice = createSlice({
  name: 'user',
  initialState: {
    name: 'Vinod',
    city: 'Bangalore',
    loading: false,
    error: null
  },
  reducers: {
    updateUser: (state, action) => {
      const { name, city } = action.payload;
      state.name = name;
      state.city = city;
    }
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchUserData.pending, (state) => {
        state.loading = true;
      })
      .addCase(fetchUserData.fulfilled, (state, action) => {
        state.loading = false;
        state.name = action.payload.name;
```

```
          state.city = action.payload.city;
        })
        .addCase(fetchUserData.rejected, (state, action) => {
          state.loading = false;
          state.error = action.error.message;
        });
    }
  });
```

Advanced Patterns

1. Middleware

Middleware provides a way to interact with actions before they reach the reducer.

```
// Custom logging middleware
const loggingMiddleware = (store) => (next) => (action) => {
  console.log('Previous State:', store.getState());
  console.log('Action:', action);
  const result = next(action);
  console.log('New State:', store.getState());
  return result;
};

// Adding middleware to store
const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(loggingMiddleware)
});
```

2. Selectors

Selectors are functions that help extract and compute derived data from the Redux store.

```
import { createSelector } from '@reduxjs/toolkit';

// Basic selector
const selectUser = state => state.user;

// Memoized selector
const selectUserDetails = createSelector(
  [selectUser],
  (user) => ({
    fullName: `${user.name} from ${user.city}`,
    isFromBangalore: user.city === 'Bangalore'
  })
);
```

```
// Using selectors in components
function UserInfo() {
  const userDetails = useSelector(selectUserDetails);
  return (
    <div>
      <p>{userDetails.fullName}</p>
      {userDetails.isFromBangalore && <span>Bangalore User!</span>}
    </div>
  );
}
```

Best Practices

1. **State Structure**:
   - Keep state normalized
   - Avoid deeply nested state
   - Use proper state slicing
2. **Performance Optimization**:

```
// Use memoized selectors
const selectFilteredUsers = createSelector(
  [selectUsers, selectFilter],
  (users, filter) => users.filter(user => user.city === filter)
);
```

3. **Action Handling**:
   - Use action creators
   - Implement proper error handling
   - Handle loading states
4. **Redux Toolkit Usage**:
   - Use createSlice for reducing boilerplate
   - Utilize built-in immer integration
   - Implement proper TypeScript integration

Common Pitfalls to Avoid

1. **Mutation of State**:

```
// Wrong
state.user.name = action.payload;

// Right (without Redux Toolkit)
return {
  ...state,
  user: {
    ...state.user,
    name: action.payload
  }
};
```

2. **Unnecessary State**:

   - Don't store derived data
   - Don't store props in Redux
   - Don't store component state in Redux

3. **Performance Issues**:

   - Avoid frequent updates to large state objects
   - Use proper memoization
   - Implement proper state normalization

Testing Redux

```
// Testing reducers
describe('userReducer', () => {
  it('should handle UPDATE_USER', () => {
    const initialState = {
      name: 'Vinod',
      city: 'Bangalore'
    };
    const action = updateUser('Shyam', 'Shivamogga');
    const nextState = userReducer(initialState, action);
    expect(nextState.name).toBe('Shyam');
    expect(nextState.city).toBe('Shivamogga');
  });
});

// Testing async actions
test('fetchUserData', async () => {
  const store = configureStore({ reducer: userReducer });
  await store.dispatch(fetchUserData('123'));
  const state = store.getState();
  expect(state.loading).toBe(false);
  expect(state.name).toBeDefined();
});
```