`useContext` is a hook provided by React that allows functional components to consume values from the React context. React context provides a way to pass data through the component tree without having to pass props down manually at every level. `useContext` simplifies the process of consuming these values within functional components.

Here's a breakdown of how `useContext` works:

1. **Create a Context:**
   First, you need to create a context using the `React.createContext()` method. This method returns a context object.

   ```jsx
   const MyContext = React.createContext(defaultValue);
   ```

   `defaultValue` is an optional parameter that represents the default value of the context. This value is only used when a component does not have a matching provider above it in the tree.

2. **Provide the Context Value:**
   Wrap the part of your component tree where you want to provide the context value with a `Context.Provider`. This provider accepts a `value` prop which is the value that will be propagated to the consuming components.

   ```jsx
   <MyContext.Provider value={/* some value */}>
     {/* Your component tree */}
   </MyContext.Provider>
   ```

3. **Consume the Context Value:**
   To access the context value within a functional component, you can use the `useContext` hook passing the context object as an argument.

   ```jsx
   const value = useContext(MyContext);
   ```

   `value` will now hold the current value of the context. Whenever the context value changes, the component using `useContext` will re-render to reflect the new value.

Here's a simple example:

```jsx
import React, { useContext } from 'react';
```

```
// Create a context
const ThemeContext = React.createContext('light');

// Component that consumes the context value
function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button style={{ background: theme }}>Themed Button</button>;
}

// Component tree with Context.Provider
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedButton />
    </ThemeContext.Provider>
  );
}

export default App;
```

In this example, `ThemedButton` consumes the `ThemeContext` value using `useContext`, and it renders a button styled based on the provided theme. The `App` component provides the context value by wrapping `ThemedButton` with `ThemeContext.Provider` and setting its value to `'dark'`.

Certainly! Here's a more detailed example of how you can use `useContext` in React:

Suppose you have a simple application where you want to manage the currently logged-in user throughout the component tree. You can use React context to provide this information down the component tree without manually passing it through props at each level.

First, let's define a context for the user:

```jsx
// UserContext.js
import React from 'react';

const UserContext = React.createContext();

export default UserContext;
```

Now, let's create a component to provide the user context:

```jsx
// UserProvider.js
import React, { useState } from 'react';
import UserContext from './UserContext';

const UserProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  const login = (username) => {
    setUser(username);
  };

  const logout = () => {
    setUser(null);
  };

  return (
    <UserContext.Provider value={{ user, login, logout }}>
      {children}
    </UserContext.Provider>
  );
};

export default UserProvider;
```

In this provider component, we maintain the state of the current user using `useState`. We also provide methods `login` and `logout` to update the user state accordingly. We wrap our `children` (the components we want to have access to this context) with `UserContext.Provider`, passing the necessary values through the `value` prop.

Now, let's create a component that consumes this user context:

```jsx
// Profile.js
import React, { useContext } from 'react';
import UserContext from './UserContext';

const Profile = () => {
  const { user, logout } = useContext(UserContext);
```

```
  return (
    <div>
      <h2>Welcome, {user}!</h2>
      <button onClick={logout}>Logout</button>
    </div>
  );
};

export default Profile;
```

In this `Profile` component, we use `useContext` to access the `UserContext`. We extract `user` and `logout` from the context object and use them to display the user's name and provide a logout button.

Finally, let's compose our application:

```jsx
// App.js
import React from 'react';
import UserProvider from './UserProvider';
import Profile from './Profile';

const App = () => {
  return (
    <UserProvider>
      <Profile />
    </UserProvider>
  );
};

export default App;
```

Now, when you render `App`, it will render `Profile`, which will have access to the user context. When you click the logout button in `Profile`, it will update the user state in the context, and `Profile` will re-render accordingly. This way, the user information is managed centrally and accessible throughout the component tree without prop drilling.