



You:

I'm about to give you a context so please remember this



ChatGPT:

Of course, I'm here to assist you. Please go ahead and provide the context, and I'll do my best to assist you based on the information you provide.



You:

I'm creating an E-commerce application called "ai-commerce". The goal is to use the help of Generative AI such as yourself to do most of the tasks. Following are some of the most important files, I want you to keep them in mind when I give you any component and I want you to use this information to write me some unit tests using jest and react-testing-library. The setup and everything else for testing is already done. I only expect a test file in return. Following are the files: `**** *src/context/AppContext.jsx* import React, { createContext, useEffect, useState } from "react"; import api from "../appwrite"; import ErrorPopup from "../components/ErrorPopup"; import LoadingOverlay from "../components/LoadingOverlay"; export const AppContext = createContext({ currentUser: null, loading: null, error: null, startSession: () => {}, endCurrentSession: () => {}, setLoading: () => {}, setError: () => {}, }); const AppProvider = ({ children }) => { const [currentUser, setCurrentUser] = useState(null); const [loading, setLoading] = useState(null); const [error, setError] = useState(null); /** * Run the useEffect on the first render of the app to initialize state based on whether a session already exists. i.e A user is logged in */ useEffect(() => { api.getAccount().then((user) => { console.log(user); setCurrentUser(user); }); }, []); const startSession = (user) => { setCurrentUser(user); }; const endCurrentSession = () => { setCurrentUser(null); }; return ( <AppContext.Provider value={{ currentUser, loading, error, startSession, endCurrentSession, setLoading, setError, }} > <div style={{ height: "100%", width: "100%", margin: "0px", padding: "0px", }} > {children} </div> <ErrorPopup message={error.message} reset={() => { setError(null); }} type={error.type} /> </div> <LoadingOverlay loadingMessage={loading} /> </AppContext.Provider> ); }; export default AppProvider; **** *src/appwrite.js* import { Client as Appwrite, Databases, Account, ID } from "appwrite"; import config from "../config"; let api = { sdk: null, provider: () => { if (api.sdk) { return api.sdk; } let appwrite = new Appwrite(); appwrite.setEndpoint(config.appwriteEndpoint).setProject(config.appwriteProjectId); const account = new Account(appwrite); const database = new Databases(appwrite); api.sdk = { database, account }; return api.sdk; }, createAccount: (email, password, name) => { return api.provider().account.create(ID.unique(), email, password, name); }, getAccount: () => { let account = api.provider().account; return account.get(); }, createSession: (email, password) => { return api.provider().account.createEmailSession(email, password); }, verifyEmail: () => { return api.provider().account.createVerification(location.origin + "/verifyEmail"); }, confirmVerification: (userId, secret) => { return api.provider().account.updateVerification(userId, secret); }, changePassword: (email) => { return api.provider().account.createRecovery(email, location.origin + "/changePassword"); }, verifyChangePassword: (userId, secret, password) => { return api.provider().account.updateRecovery(userId, secret, password, password); }, deleteCurrentSession: () => { return api.provider().account.deleteSession("current"); }, createDocument: (databaseId, collectionId, data) => { return api.provider().database.createDocument(databaseId, collectionId, ID.unique(), data); }, listDocuments: (databaseId, collectionId) => { return api.provider().database.listDocuments(databaseId, collectionId); }, updateDocument: (databaseId, collectionId, documentId, data) => { return api.provider().database.updateDocument(databaseId, collectionId, documentId, data); }, deleteDocument: (databaseId, collectionId, documentId) => { return api.provider().database.deleteDocument(databaseId, collectionId, documentId); }, }; export default api; **** *src/config.js* const config = { appName: import.meta.env.VITE_APP_NAME, appwriteEndpoint: import.meta.env.VITE_API_ENDPOINT, appwriteProjectId: import.meta.env.VITE_APPWRITE_PROJECT_ID, dbProductsId: import.meta.env.VITE_DB_PRODUCTS, collectionProductsId: import.meta.env.VITE_COLLECTION_PRODUCTS, collectionOrdersId: import.meta.env.VITE_COLLECTION_ORDERS, stripePubKey: import.meta.env.VITE_STRIPE_PUB_KEY, }; export default config; **** *src/App.jsx* import { RouterProvider } from "react-router-dom"; import { Elements } from "@stripe/react-stripe-js"; import { loadStripe } from "@stripe/stripe-js"; import AppProvider from "../context/AppContext"; import router from "../routes"; import "../App.scss"; import config from "../config"; const stripePromise = loadStripe(String(config.stripePubKey)); const App = () => { return ( <Elements stripe={stripePromise}> <AppProvider> <RouterProvider router={router} /> </AppProvider> </Elements> ); }; export default App; **** Now I want you to write me unit tests for the following component: **** *src/components/CartPanel.jsx* import React, { useContext } from "react"; import { Box, Button, Drawer, Typography } from "@mui/material"; import { dark } from "../themes"; import { AppContext } from "../context/AppContext"; import ProductCapsule from "../ProductCapsule"; import { CardElement, useElements, useStripe } from "@stripe/react-stripe-js"; import api from "../appwrite"; import config from "../config"; const CartPanel = ({ isOpen, onClose }) => { const { currentUser, setLoading, setError, startSession } = useContext(AppContext); const stripe = useStripe(); const elements = useElements(); const cartItems = Object.values(currentUser.cartItems); const mockStripePayment = async () => { // Simulate a Stripe payment request await new Promise((resolve) => setTimeout(resolve, 5000)); // Simulate a delay // Mock a successful payment response return { token: "mock-stripe-token" }; }; const handleCheckout = async () => { if (!stripe || !elements) return; try { setLoading("Processing Payment"); // Simulate a payment request const { token } = await mockStripePayment(); console.log("Received token:", token); const newOrder = { userEmail: currentUser.email, cartItems: JSON.stringify(currentUser.cartItems), totalAmount: cartItems.map((item) => item.amount).reduce((tot, cur) => (tot += cur)).toFixed(2), currency: cartItems[0].currency, }; await api.createDocument(config.dbProductsId, config.collectionOrdersId, newOrder); setError({ message: "Order placed successfully", type: "success" }); const newUser = { ...currentUser }; newUser.cartItems = []; startSession(newUser); } catch (error) { console.log(error.message); setError({ message: error.message, type: "error" }); } finally { setLoading(null); }; return ( <Drawer anchor="right" open={isOpen} onClose={onClose}> <div> Add cart UI and checkout items here </div> <Box backgroundColor={dark.card.backgroundSecondary} display="flex" flexDirection="column" sx={{ padding: "16px", width: "50vw", height: "100%" }} gap={1} > <Typography variant="h5" fontWeight="bold" color={dark.text.secondary}> Your Cart </Typography> <div> Add cart contents </div> <Box display="flex" flexDirection="column" alignItems="center" flex={1} p={2} > <Box display="flex" justifyContent="center" alignItems="center" flexDirection="column" width="100%" flex={1} gap={2} > {cartItems.length === 0 && ( <Typography variant="subtitle1" fontWeight={500} color={dark.text.secondary} > No items in the cart </Typography> )} {cartItems.map((item) => ( <ProductCapsule key={Math.random()} productName={item.name} quantity={item.quantity} productPrice={item.price} totalAmount={item.amount} currency={item.currency} /> ))} {cartItems.length > 0 && ( <Typography variant="h6" fontSize={18} fontWeight="bold" color={dark.text.primary} > Total Cost: " " <div> {cartItems.map((item) => item.amount).reduce((tot, cur) => (tot += cur)).toFixed(2)} </div> <div> {Object.values(cartItems)[0].currency} </div> </Box> </div> </Box> <div> <Button variant="filled" sx={{ backgroundColor: dark.accent.primary, color: dark.text.hover, minWidth: "60%", "& .Mui-disabled": { backgroundColor: dark.disabled.background, color: dark.disabled.text, } }} disabled={cartItems.length === 0} onClick={() => handleCheckout()} > Checkout </Button> </div> </div> </Drawer> ); }; export default CartPanel; ****`



ChatGPT:

To write unit tests for the CartPanel component using Jest and React Testing Library, you can follow the steps below. These tests will cover various aspects of the component, including rendering, user interactions, and state changes. I'll provide you with a test file that you can use:

```

javascript
// src/components/CartPanel.test.js

import React from "react";
import { render, screen, fireEvent, waitFor } from "@testing-library/react";
import { AppContext } from "../context/AppContext";
import CartPanel from "../CartPanel";

// Mock the useContext hook to provide values for testing
const mockContextValue = {
  currentUser: {
    email: "test@example.com",
    cartItems: {
      item1: {
        name: "Test Product 1",
        quantity: 2,
        price: 10.0,
        amount: 20.0,
        currency: "USD",
      },
    },
  },
  setLoading: jest.fn(),
  setError: jest.fn(),
  startSession: jest.fn(),
};

// Mock the useStripe and useElements hooks
jest.mock("@stripe/react-stripe-js", () => ({
  useStripe: () => ({
    // Mock any Stripe methods or properties you need
  }),
  useElements: () => ({
    // Mock any Elements-related methods or properties you need
  }),
}));

describe("CartPanel Component", () => {
  it("renders CartPanel component correctly", () => {
    render(
      <AppContext.Provider value={mockContextValue}>
        <CartPanel isOpen={true} onClose={() => {}} />
      </AppContext.Provider>
    );

    // Ensure that the cart items are displayed
    const productElement = screen.getByText("Test Product 1");
    expect(productElement).toBeInTheDocument();
  });

  it("handles checkout button click", async () => {
    render(
      <AppContext.Provider value={mockContextValue}>
        <CartPanel isOpen={true} onClose={() => {}} />
      </AppContext.Provider>
    );

    // Find and click the checkout button
    const checkoutButton = screen.getByText("Checkout");
    fireEvent.click(checkoutButton);

    // Ensure that the loading state is set
    expect(mockContextValue.setLoading).toHaveBeenCalledWith("Processing Payment");

    // Wait for the mockStripePayment to complete
    await waitFor(() => {
      expect(mockContextValue.setError).toHaveBeenCalledWith({
        message: "Order placed successfully",
        type: "success",
      });
    });

    // Ensure that startSession is called with the updated user
    expect(mockContextValue.startSession).toHaveBeenCalledWith({
      email: "test@example.com",
      cartItems: {},
    });

    // Ensure that the loading state is reset
  });
}

```

```

    expect(mockContextValue.setLoading).toHaveBeenCalledWith(null);
  });

  it("disables checkout button when cart is empty", () => {
    // Modify the context value to have an empty cart
    const emptyCartContextValue = { ...mockContextValue, currentUser: { ...mockContextValue.currentUser, cartItems: {} } }
    render(
      <AppContext.Provider value={emptyCartContextValue}>
        <CartPanel isOpen={true} onClose={() => {}} />
      </AppContext.Provider>
    );

    // Find the checkout button and ensure it's disabled
    const checkoutButton = screen.getByText("Checkout");
    expect(checkoutButton).toBeDisabled();
  });
});

```

In this test file:

1. We render the `CartPanel` component with a mocked `AppContext.Provider` to provide the necessary context values for testing.
2. We mock the `useStripe` and `useElements` hooks since we don't need to test Stripe functionality in this unit test.
3. In the first test, we check if the component renders correctly and if the cart items are displayed.
4. In the second test, we simulate a click on the "Checkout" button and ensure that the loading state is set, the payment is processed (mocked), and the relevant context functions are called.
5. In the third test, we modify the context to have an empty cart and verify that the "Checkout" button is disabled when the cart is empty.

You can run these tests using Jest, and they should help ensure that the `CartPanel` component behaves as expected. Make sure to adjust the context values and test cases according to your specific requirements.



got this error: Element type is invalid: expected a string (for built-in components) or a class/function (for composite components) but got: undefined. You likely forgot to export your component from the file it's defined in, or you might have mixed up default and named imports.



The error message you're encountering, "Element type is invalid," typically occurs when there's an issue with how components are imported or exported. The error message suggests that the component you're trying to render is either undefined or not properly exported from its source file. Here are some steps to troubleshoot and resolve this issue:

1. **Check Component Export:** Ensure that the `CartPanel` component is correctly exported from its source file (i.e., `CartPanel.jsx`). Make sure you have an export statement at the end of the file.

In `CartPanel.jsx`:

```

javascript
export default CartPanel;

```

2. **Verify Import Statements:** Double-check the import statement for the `CartPanel` component in your test file (`CartPanel.test.js`). Ensure that you are importing it correctly, and the file path is accurate.

In `CartPanel.test.js`:

```

javascript
import CartPanel from "../CartPanel"; // Adjust the path if necessary

```

3. **Check File Names and Paths:** Ensure that the file name and path to the `CartPanel` component match the actual file structure. If the file name or path is incorrect, you will encounter this error.
4. **Case Sensitivity:** JavaScript is case-sensitive. Ensure that the file names and import statements match the case of the actual file names. For example, "CartPanel" should match the case of the actual filename "CartPanel.jsx".
5. **Check for Typos:** Look for any typos or spelling mistakes in your import statements and component names.
6. **Clear Jest Cache:** Sometimes, Jest's cache can cause issues with imports. Try clearing Jest's cache by running the following command:

```

css
jest --clearCache

```

7. **Restart Development Server:** If you're running a development server (e.g., `npm start` or `yarn start`), try stopping and restarting it. Sometimes, this can help refresh the module cache.
8. **Check Dependencies:** Ensure that you have all the necessary dependencies installed, including `@testing-library/react`, `@testing-library/jest-dom`, and any other dependencies required for your project.

After performing these checks and making any necessary adjustments, try running your tests again. This should help resolve the "Element type is invalid" error. If you continue to encounter issues, please provide more details about your project setup, including folder structure and `package.json`, so that I can provide more specific guidance.



you forgot to mock this: "<CardElement />"

-----



You are correct; I apologize for the oversight. You should mock the `CardElement` component when rendering the `CartPanel` component in your test to prevent any unintended side effects. Here's how you can do it using Jest's `jest.mock`:

```
javascript
import React from "react";
import { render, screen, fireEvent, waitFor } from "@testing-library/react";
import { AppContext } from "../context/AppContext";
import CartPanel from "../CartPanel";

// Mock the useContext hook to provide values for testing
const mockContextValue = {
  // ... (mocked context values)
};

// Mock the useStripe and useElements hooks
jest.mock("@stripe/react-stripe-js", () => ({
  useStripe: jest.fn(), // Mock useStripe as an empty function
  useElements: jest.fn(), // Mock useElements as an empty function
  CardElement: () => <div data-testid="mock-card-element" />, // Mock CardElement as a div
}));

describe("CartPanel Component", () => {
  // ... (other test cases)

  it("renders CardElement component", () => {
    render(
      <AppContext.Provider value={mockContextValue}>
        <CartPanel isOpen={true} onClose={() => {}} />
      </AppContext.Provider>
    );

    // Verify that the CardElement component is rendered
    const cardElement = screen.getByTestId("mock-card-element");
    expect(cardElement).toBeInTheDocument();
  });
});
```

In this updated code, we mock the `CardElement` component as a simple `div` element using Jest's `jest.mock`. Additionally, we add a test case to ensure that the `CardElement` component is rendered in the `CartPanel` component. This prevents any unexpected behavior related to the Stripe-related components during testing.



you also forgot to mock this: `api.createDocument`

-----



You're absolutely right; I apologize for the oversight. To prevent actual API calls, you should mock the `api.createDocument` function. Here's how you can do it using Jest's `jest.mock`:

```
javascript
```

```
import React from "react";
import { render, screen, fireEvent, waitFor } from "@testing-library/react";
import { AppContext } from "../context/AppContext";
import CartPanel from "../CartPanel";

// Mock the useContext hook to provide values for testing
const mockContextValue = {
  // ... (mocked context values)
};

// Mock the useStripe and useElements hooks
jest.mock("@stripe/react-stripe-js", () => ({
  useStripe: jest.fn(), // Mock useStripe as an empty function
  useElements: jest.fn(), // Mock useElements as an empty function
  CardElement: () => <div data-testid="mock-card-element" />, // Mock CardElement as a div
}));

// Mock the api.createDocument function
jest.mock("../appwrite", () => ({
  createDocument: jest.fn(), // Mock createDocument as an empty function
}));

describe("CartPanel Component", () => {
  // ... (other test cases)

  it("handles checkout button click", async () => {
    // Mock the createDocument function to resolve immediately
    const mockCreateDocument = jest.fn().mockResolvedValue({});
    jest.mock("../appwrite", () => ({
      createDocument: mockCreateDocument,
    }));

    render(
      <AppContext.Provider value={mockContextValue}>
        <CartPanel isOpen={true} onClose={() => {}} />
      </AppContext.Provider>
    );

    // ... (rest of the test case)
  });
});
```

In this updated code, we:

1. Mock the `api.createDocument` function using Jest's `jest.mock`. We mock it as an empty function.
2. For the test case that handles the checkout button click, we further mock the `api.createDocument` function with a custom implementation using `jest.fn().mockResolvedValue({})` to make it resolve immediately without performing any actual API calls.

By mocking `api.createDocument` in this way, you ensure that the test case focuses solely on the component's behavior without involving actual API calls.



You:

now mock this component based on the context i had previously provided `import React, { useEffect, useState } from "react"; import { Snackbar, IconButton, Typography, Box } from "@mui/material"; import CloseIcon from "@mui/icons-material/Close"; import ErrorOutlineIcon from "@mui/icons-material/ErrorOutline"; import { dark } from "../themes"; const COLORS = { success: dark.notification.success, info: dark.notification.info, error: dark.notification.error, }; export default function ErrorPopup({ message, reset, type }) { const [show, setShow] = useState(!message); console.log(!message); useEffect(() => { const timer = setTimeout(() => { setShow(false); reset(); }, 15000); // Timeout of 15 seconds return () => { clearTimeout(timer); }; }, []); const handleAnimationEnd = () => { if (!show) { reset(); }; }; const getColor = (type) => { return COLORS[type]; }; return ( <Snackbar open={show} onClose={() => setShow(false)} autoHideDuration={15000} onExited={handleAnimationEnd} anchorOrigin={{ vertical: "top", horizontal: "center" }} // Move to top-center sx={{ backgroundColor: getColor(type), borderRadius: "8px", padding: "8px 12px", }} > <div style={{ display: "flex", alignItems: "center" }}> <ErrorOutlineIcon sx={{ color: dark.text.hover, }} /> <Typography variant="body1" color={dark.text.hover} sx={{ ml: 2 }}> {message} </Typography> <IconButton size="small" aria-label="close" onClick={() => { setShow(false); reset(); }} > <CloseIcon fontSize="small" sx={{ color: dark.text.hover, }} /> </IconButton> </div> </Snackbar> ); }`



ChatGPT:

To mock the `ErrorPopup` component for testing, you can create a mock version of it. Since this component relies on some state changes and effects, we'll need to account for those in our mock. Here's how you can do it:

```
javascript
import React from "react";

const MockedErrorPopup = ({ message, reset, type }) => {
  return (
```

```

    <div>
      {/* Mocked ErrorPopup content */}
      <p data-testid="mock-error-message">{message}</p>
    </div>
  );
};

export default MockedErrorPopup;

```

In this mock, we've created a simplified version of the `ErrorPopup` component that simply displays the `message` prop within a `p` element. We also added a `data-testid` attribute to the `p` element for easy querying in your tests.

Now, you can use this `MockedErrorPopup` component in your tests instead of the actual `ErrorPopup` component to isolate the testing of other components and logic that rely on it. When you render your component that uses `ErrorPopup`, replace it with `MockedErrorPopup`:

```

javascript
import { render, screen } from "@testing-library/react";
import YourComponentUsingErrorPopup from "../YourComponentUsingErrorPopup"; // Import your component

// Replace ErrorPopup with MockedErrorPopup in your component
jest.mock("../path/to/ErrorPopup", () => ({
  __esModule: true,
  default: MockedErrorPopup,
})));

describe("YourComponentUsingErrorPopup", () => {
  it("should render correctly", () => {
    // Render your component
    render(<YourComponentUsingErrorPopup />);

    // Query the mock error message
    const errorMessage = screen.getByTestId("mock-error-message");

    // Perform your assertions on errorMessage
    // ...
  });
  // ...
});

```

By mocking the `ErrorPopup` component with the `MockedErrorPopup`, you can focus on testing the behavior of your component that uses `ErrorPopup` without worrying about the intricacies of the `ErrorPopup` component itself.



can we not do this without mocking the entire component?

-----



Certainly, if you want to test the `ErrorPopup` component without mocking the entire component, you can do so. However, keep in mind that the actual `ErrorPopup` component may have some dependencies like `useState`, `useEffect`, and `Snackbar`, which may lead to unwanted side effects when rendering it directly in your tests. To mitigate this, you can use React Testing Library's `render` function to isolate the testing of the `ErrorPopup` component while minimizing its side effects.

Here's an example of how you can test the `ErrorPopup` component without mocking the entire component:

```

javascript
import React from "react";
import { render, screen } from "@testing-library/react";
import ErrorPopup from "../ErrorPopup"; // Import the actual ErrorPopup component

describe("ErrorPopup Component", () => {
  it("should render with the provided message", () => {
    const message = "Test error message";
    const reset = jest.fn(); // Mock the reset function
    const type = "error";

    // Render the ErrorPopup component
    render(<ErrorPopup message={message} reset={reset} type={type} />);

    // Query elements within the rendered component
    const errorMessage = screen.getByText(message);
    const closeButton = screen.getByRole("button", { name: "close" });

    // Assert that the error message is displayed

```

```

expect(errorMessage).toBeInTheDocument();

// Simulate a click on the close button
fireEvent.click(closeButton);

// Ensure that the reset function is called when the close button is clicked
expect(reset).toHaveBeenCalledTimes(1);
});

// Add more test cases as needed
});

```

In this test case:

1. We import the actual `ErrorPopup` component.
2. We render the `ErrorPopup` component with the desired props (message, reset function, and type).
3. We use React Testing Library to query and assert elements within the rendered component. In this case, we check that the error message is displayed and that the close button works as expected.

This approach allows you to test the `ErrorPopup` component without mocking the entire component, focusing on its behavior and interactions while minimizing potential side effects.



You:

similarly test the following component: `import React from "react"; import { Box, Card, CardContent, CircularProgress, Typography, } from "@mui/material"; import { dark } from "../themes"; const LoadingOverlay = ({ loadingMessage }) => { console.log("In here"); return ( <Box style={{ display: "flex", justifyContent: "center", alignItems: "center", position: "fixed", top: 0, left: 0, width: "100%", height: "100%", backgroundColor: "rgba(0, 0, 0, 0.5)", // Semi-transparent overlay zIndex: 9999, }} > <Card style={{ maxWidth: 400, padding: "8px", textAlign: "center", backgroundColor: dark.card.background, }} > <CardContent> <Typography variant="body1" color={dark.text.hover} sx={{ mb: 2 }}> {loadingMessage && loadingMessage.length > 0 ? loadingMessage : "Loading ..."} </Typography> <CircularProgress sx={{ color: dark.accent.highlight }} /> </CardContent> </Card> </Box> ); }; export default LoadingOverlay;`



ChatGPT:

Certainly! Here's how you can test the `LoadingOverlay` component using React Testing Library:

```

javascript
import React from "react";
import { render, screen } from "@testing-library/react";
import LoadingOverlay from "../LoadingOverlay"; // Import the actual LoadingOverlay component

describe("LoadingOverlay Component", () => {
  it("should render with the provided loading message", () => {
    const loadingMessage = "Loading data...";

    // Render the LoadingOverlay component
    render(<LoadingOverlay loadingMessage={loadingMessage} />);

    // Query elements within the rendered component
    const loadingMessageElement = screen.getByText(loadingMessage);
    const circularProgressElement = screen.getByRole("progressbar");

    // Assert that the loading message is displayed
    expect(loadingMessageElement).toBeInTheDocument();

    // Assert that the circular progress indicator is displayed
    expect(circularProgressElement).toBeInTheDocument();
  });

  it("should render with the default loading message if none is provided", () => {
    // Render the LoadingOverlay component without a loading message
    render(<LoadingOverlay />);

    // Query the default loading message element
    const defaultLoadingMessageElement = screen.getByText("Loading ...");

    // Assert that the default loading message is displayed
    expect(defaultLoadingMessageElement).toBeInTheDocument();
  });

  // Add more test cases as needed
});

```



In these test cases:

1. We import the actual `LoadingOverlay` component.
2. In the first test case, we render the `LoadingOverlay` component with a specific loading message and then use React Testing Library to query and assert elements within the rendered component. We check that the loading message and the circular progress indicator are displayed as expected.
3. In the second test case, we render the `LoadingOverlay` component without providing a loading message, and then we check that the default loading message ("Loading ...") is displayed.

You can add more test cases as needed to cover different scenarios or behaviors of the `LoadingOverlay` component.



You:

```
similarly test the following component: import { AccountCircle, AccountCircleOutlined, ShoppingBasketOutlined, ShoppingCart, ShoppingCartOutlined, } from "@mui/icons-material"; import { AppBar, Box, IconButton, ListItemText, Menu, MenuItem, Toolbar, Typography, } from "@mui/material"; import React, { useContext, useState } from "react"; import { AppContext } from "../context/AppContext"; import { dark } from "../themes"; import api from "../appwrite"; import config from "../config"; const Navbar = ({ isCartOpen, handleCartToggle }) => { const [anchorEl, setAnchorEl] = useState(null); const { currentUser, endCurrentSession, setLoading, setError } = useContext(AppContext); const handleMenuOpen = (event) => { setAnchorEl(event.currentTarget); }; const handleMenuClose = () => { setAnchorEl(null); }; const onChangePassword = () => { setLoading(""); api.changePassword(currentUser.email).then((res) => { if (res) { setError({ message: "Click on the link sent to your email to reset the password", type: "info", }); } else { setError({ message: "Some error occurred", type: "error" }); }) .catch((err) => setError({ message: err.message, type: "error" })) .finally(() => setLoading(null)); }; const onLogout = () => { setLoading("Logging out"); api.deleteCurrentSession().then(() => { endCurrentSession(); }) .catch((err) => setError({ message: err.message, type: "error" })) .finally(() => setLoading(null)); }; return ( <AppBar position="static" sx={{ backgroundColor: dark.card.background, }} > <Toolbar sx={{ boxShadow: "0 4px 8px rgba(255, 255, 255, 0.3)" }} > <Box gap={2} display="flex" justify-content="left" align-items="center" sx={{ flexGrow: 1 }} > <ShoppingBasketOutlined sx={{ fontSize: 36, color: dark.accent.primary }} /> <Typography sx={{ fontSize: 24, color: dark.accent.highlight, fontWeight: 500 }} > {config.appName} </Typography> </Box> <IconButton color="inherit" onClick={handleCartToggle} > {isCartOpen ? ( <ShoppingCart sx={{ color: dark.text.hover, fontSize: 32 }} /> ) : ( <ShoppingCartOutlined sx={{ color: dark.text.hover, fontSize: 32 }} /> ) } </IconButton> <IconButton color="inherit" onClick={handleMenuOpen} > {anchorEl ? ( <AccountCircle sx={{ color: dark.text.hover, fontSize: 32 }} /> ) : ( <AccountCircleOutlined sx={{ color: dark.text.hover, fontSize: 32 }} /> ) } </IconButton> <Menu anchorEl={anchorEl} open={Boolean(anchorEl)} onClose={handleMenuClose} sx={{ color: dark.text.primary, }} > <MenuItem disabled> <ListItemText primary={ <Typography variant="subtitle1" sx={{ fontWeight: "bold" }} style={{ color: dark.background }} > {currentUser.name} </Typography> } disableTypography /> </MenuItem> <MenuItem onClick={onLogout}>Logout</MenuItem> </Menu> </Toolbar> </AppBar> ); }; export default Navbar;
```



ChatGPT:

To test the `Navbar` component, you can follow a similar approach as before. Here's how you can write tests for the `Navbar` component using React Testing Library:

```
javascript
import React from "react";
import { render, screen, fireEvent } from "@testing-library/react";
import Navbar from "../Navbar"; // Import the actual Navbar component

describe("Navbar Component", () => {
  it("should render with the app name and icons", () => {
    const isCartOpen = false;
    const handleCartToggle = jest.fn(); // Mock the cart toggle function

    // Render the Navbar component
    render(<Navbar isCartOpen={isCartOpen} handleCartToggle={handleCartToggle} />);

    // Query elements within the rendered component
    const appNameElement = screen.getByText("YourAppNameHere"); // Replace with your app name
    const cartIconElement = screen.getByRole("button", { name: "Open Cart" });
    const userIconElement = screen.getByRole("button", { name: "Open User Menu" });

    // Assert that the app name and icons are displayed
    expect(appNameElement).toBeInTheDocument();
    expect(cartIconElement).toBeInTheDocument();
    expect(userIconElement).toBeInTheDocument();
  });

  it("should call handleCartToggle when the cart icon is clicked", () => {
    const isCartOpen = false;
    const handleCartToggle = jest.fn(); // Mock the cart toggle function

    // Render the Navbar component with the mocked handleCartToggle function
    render(<Navbar isCartOpen={isCartOpen} handleCartToggle={handleCartToggle} />);

    // Query and click the cart icon
    const cartIconElement = screen.getByRole("button", { name: "Open Cart" });
    fireEvent.click(cartIconElement);

    // Assert that handleCartToggle is called
  });
});
```



```

    expect(handleCartToggle).toHaveBeenCalledTimes(1);
  });

  // Add more test cases as needed
});

```

In these test cases:

1. We import the actual Navbar component.
2. In the first test case, we render the Navbar component and use React Testing Library to query and assert elements within the rendered component. We check that the app name and icons are displayed as expected.
3. In the second test case, we render the Navbar component with a mocked handleCartToggle function and simulate a click on the cart icon. We then assert that handleCartToggle is called once when the cart icon is clicked.

You can add more test cases to cover different aspects of the Navbar component's behavior and interactions as needed.



You:

you completely forgot to wrap it in context



ChatGPT:

You're right; I apologize for the oversight. To test the Navbar component properly, it should be wrapped in a context provider to provide the necessary context values. Here's an updated version of the test that includes wrapping the component in the AppContext provider:

```

javascript
import React from "react";
import { render, screen, fireEvent } from "@testing-library/react";
import Navbar from "../Navbar"; // Import the actual Navbar component
import { AppContext } from "../context/AppContext"; // Import the AppContext

describe("Navbar Component", () => {
  it("should render with the app name and icons", () => {
    const isCartOpen = false;
    const handleCartToggle = jest.fn(); // Mock the cart toggle function
    const currentUser = {
      name: "John Doe", // Replace with your user data
    };

    // Render the Navbar component wrapped in the AppContext provider
    render(
      <AppContext.Provider value={{ currentUser }}>
        <Navbar isCartOpen={isCartOpen} handleCartToggle={handleCartToggle} />
      </AppContext.Provider>
    );

    // Query elements within the rendered component
    const appNameElement = screen.getByText("YourAppNameHere"); // Replace with your app name
    const cartIconElement = screen.getByRole("button", { name: "Open Cart" });
    const userIconElement = screen.getByRole("button", { name: "Open User Menu" });

    // Assert that the app name and icons are displayed
    expect(appNameElement).toBeInTheDocument();
    expect(cartIconElement).toBeInTheDocument();
    expect(userIconElement).toBeInTheDocument();
  });

  it("should call handleCartToggle when the cart icon is clicked", () => {
    const isCartOpen = false;
    const handleCartToggle = jest.fn(); // Mock the cart toggle function
    const currentUser = {
      name: "John Doe", // Replace with your user data
    };

    // Render the Navbar component wrapped in the AppContext provider
    render(
      <AppContext.Provider value={{ currentUser }}>
        <Navbar isCartOpen={isCartOpen} handleCartToggle={handleCartToggle} />
      </AppContext.Provider>
    );

    // Query and click the cart icon
    const cartIconElement = screen.getByRole("button", { name: "Open Cart" });

```

```

    fireEvent.click(cartIconElement);

    // Assert that handleCartToggle is called
    expect(handleCartToggle).toHaveBeenCalledTimes(1);
  });

  // Add more test cases as needed
});

```

In these updated test cases, the Navbar component is wrapped in an `AppContext.Provider`, providing the necessary context values, including `currentUser`. This ensures that the component receives the expected context and allows for proper testing of its behavior.



You:

```

similarly write tests for this component: import { Box, Typography } from "@mui/material"; import React from "react"; import { dark } from "../themes";
const ProductCapsule = ({ quantity, productName, productPrice, totalAmount, currency, }) => { return ( <Box display="flex" p={2} maxWidth="100%"
minWidth="80%" width="100%" sx={{ backgroundColor: dark.card.background, borderRadius: "8px", p: 0 }} > <Box display="flex"
flexDirection="column" alignItems="center" minWidth="7%" maxWidth="7%" width="7%" sx={{ backgroundColor: dark.accent.highlight, m: 0, p: "6px
8px", borderRadius: "8px", }} > <Typography variant="subtitle2" fontSize={14}> QTY. </Typography> <Typography display="flex" flex={1}
justifyContent="center" alignItems="center" fontSize={16} fontWeight="bold" whiteSpace="nowrap" overflow="hidden" textOverflow="ellipsis" py="4px"
> {quantity} </Typography> </Box> <Box display="flex" flexDirection="column" flex={1} alignItems="center" minWidth="43%" maxWidth="43%"
width="43%" whiteSpace="nowrap" overflow="hidden" textOverflow="ellipsis" sx={{ m: 0, p: "6px 8px", }} > <Typography width="100%"
textAlign="center" variant="subtitle2" fontSize={14} color={dark.text.secondary} > Product Name </Typography> <Typography fontSize={16}
fontWeight={500} textAlign="center" py="4px" color={dark.text.primary} > {productName} </Typography> </Box> <Box display="flex"
flexDirection="column" alignItems="center" width="25%" whiteSpace="nowrap" overflow="hidden" textOverflow="ellipsis" sx={{ m: 0, p: "6px 8px", }} >
<Typography variant="subtitle2" fontSize={14} color={dark.text.secondary} > Price per item </Typography> <Typography fontSize={16} fontWeight=
{500} py="4px" color={dark.text.primary} > {`$${productPrice.toFixed(2)} ${currency}`} </Typography> </Box> <Box display="flex"
flexDirection="column" alignItems="center" width="25%" whiteSpace="nowrap" overflow="hidden" textOverflow="ellipsis" sx={{ m: 0, p: "6px 8px", }} >
<Typography variant="subtitle2" fontSize={14} color={dark.text.secondary} > Total </Typography> <Typography fontSize={16} fontWeight={500}
py="4px" color={dark.text.primary} > {`$${totalAmount.toFixed(2)} ${currency}`} </Typography> </Box> </Box> ); }; export default ProductCapsule;

```



ChatGPT:

To test the `ProductCapsule` component, you can use `React Testing Library`. Here's how you can write tests for the `ProductCapsule` component:

```

javascript
import React from "react";
import { render, screen } from "@testing-library/react";
import ProductCapsule from "../ProductCapsule"; // Import the actual ProductCapsule component
import { dark } from "../themes"; // Import the dark theme

describe("ProductCapsule Component", () => {
  it("should render with product details", () => {
    const quantity = 3;
    const productName = "Sample Product";
    const productPrice = 10.99;
    const totalAmount = 32.97;
    const currency = "USD";

    // Render the ProductCapsule component with props
    render(
      <ProductCapsule
        quantity={quantity}
        productName={productName}
        productPrice={productPrice}
        totalAmount={totalAmount}
        currency={currency}
      />
    );

    // Query elements within the rendered component
    const quantityElement = screen.getByText(`QTY. ${quantity}`);
    const productNameElement = screen.getByText(productName);
    const pricePerItemElement = screen.getByText(`${productPrice.toFixed(2)} ${currency}`);
    const totalElement = screen.getByText(`${totalAmount.toFixed(2)} ${currency}`);

    // Assert that the product details are displayed
    expect(quantityElement).toBeInTheDocument();
    expect(productNameElement).toBeInTheDocument();
    expect(pricePerItemElement).toBeInTheDocument();
    expect(totalElement).toBeInTheDocument();
  });

  it("should render with proper styles", () => {
    const quantity = 3;

```

```

const productName = "Sample Product";
const productPrice = 10.99;
const totalAmount = 32.97;
const currency = "USD";

// Render the ProductCapsule component with props
render(
  <ProductCapsule
    quantity={quantity}
    productName={productName}
    productPrice={productPrice}
    totalAmount={totalAmount}
    currency={currency}
  />
);

// Query elements with expected styles
const productCapsule = screen.getByRole("box");
const quantityBox = screen.getByRole("box", { name: "QTY." });
const productNameBox = screen.getByRole("box", { name: "Product Name" });
const pricePerItemBox = screen.getByRole("box", { name: "Price per item" });
const totalBox = screen.getByRole("box", { name: "Total" });

// Assert that the elements have the expected styles
expect(productCapsule).toHaveStyle(`background-color: ${dark.card.background}`);
expect(quantityBox).toHaveStyle(`background-color: ${dark.accent.highlight}`);
expect(productNameBox).toHaveStyle(`color: ${dark.text.secondary}`);
expect(pricePerItemBox).toHaveStyle(`color: ${dark.text.primary}`);
expect(totalBox).toHaveStyle(`color: ${dark.text.primary}`);
});

// Add more test cases as needed
});

```

In these test cases:

1. We import the actual ProductCapsule component and the dark theme.
2. In the first test case, we render the ProductCapsule component with specific props and use React Testing Library to query and assert elements within the rendered component. We check that the product details are displayed correctly.
3. In the second test case, we render the ProductCapsule component with props and then query elements that are expected to have specific styles. We use the toHaveStyle matcher to assert that the elements have the expected styles.

You can add more test cases to cover different scenarios or styles for the ProductCapsule component as needed.



write tests according to the very first tests you wrote



Certainly! Here are tests for the ProductCapsule component following a similar structure to the initial tests:

```

javascript
import React from "react";
import { render, screen } from "@testing-library/react";
import ProductCapsule from "../ProductCapsule"; // Import the actual ProductCapsule component
import { dark } from "../themes"; // Import the dark theme

describe("ProductCapsule Component", () => {
  it("should render with product details", () => {
    const quantity = 3;
    const productName = "Sample Product";
    const productPrice = 10.99;
    const totalAmount = 32.97;
    const currency = "USD";

    // Render the ProductCapsule component with props
    render(
      <ProductCapsule
        quantity={quantity}
        productName={productName}
        productPrice={productPrice}
        totalAmount={totalAmount}

```

```

        currency={currency}
      />
    );

    // Query elements within the rendered component
    const quantityElement = screen.getByText(`QTY. ${quantity}`);
    const productNameElement = screen.getByText(productName);
    const pricePerItemElement = screen.getByText(`${productPrice.toFixed(2)} ${currency}`);
    const totalElement = screen.getByText(`${totalAmount.toFixed(2)} ${currency}`);

    // Assert that the product details are displayed
    expect(quantityElement).toBeInTheDocument();
    expect(productNameElement).toBeInTheDocument();
    expect(pricePerItemElement).toBeInTheDocument();
    expect(totalElement).toBeInTheDocument();
  });

  it("should render with proper styles", () => {
    const quantity = 3;
    const productName = "Sample Product";
    const productPrice = 10.99;
    const totalAmount = 32.97;
    const currency = "USD";

    // Render the ProductCapsule component with props
    render(
      <ProductCapsule
        quantity={quantity}
        productName={productName}
        productPrice={productPrice}
        totalAmount={totalAmount}
        currency={currency}
      />
    );

    // Query elements with expected styles
    const productCapsule = screen.getByRole("box");
    const quantityBox = screen.getByRole("box", { name: "QTY." });
    const productNameBox = screen.getByRole("box", { name: "Product Name" });
    const pricePerItemBox = screen.getByRole("box", { name: "Price per item" });
    const totalBox = screen.getByRole("box", { name: "Total" });

    // Assert that the elements have the expected styles
    expect(productCapsule).toHaveStyle(`background-color: ${dark.card.background}`);
    expect(quantityBox).toHaveStyle(`background-color: ${dark.accent.highlight}`);
    expect(productNameBox).toHaveStyle(`color: ${dark.text.secondary}`);
    expect(pricePerItemBox).toHaveStyle(`color: ${dark.text.primary}`);
    expect(totalBox).toHaveStyle(`color: ${dark.text.primary}`);
  });

  // Add more test cases as needed
});

```

These test cases follow a similar structure to the initial tests. The first test checks if the product details are rendered correctly, and the second test checks if the elements have the expected styles based on the provided props and the dark theme.

You can add more test cases to cover additional scenarios or behavior of the ProductCapsule component as needed.



You:

similarly write for this component import React, { useContext, useEffect, useState } from "react"; import { Card, CardContent, Typography, CardMedia, Button, } from "@mui/material"; import { dark } from "../themes"; import QuantitySelector from "../QuantitySelector"; import { AppContext } from "../context/AppContext"; const ProductCard = ({ productId, productName, description, imageUrl, price, currency, }) => { const { currentUser, startSession } = useContext(AppContext); const [quantity, setQuantity] = useState(0); useEffect(() => { if (Object.values(currentUser.cartItems).length === 0) { setQuantity(0); }, [currentUser]; const addItemToCart = () => { setQuantity((q) => q + 1); const newUser = { ...currentUser }; const newCartItems = { ...newUser.cartItems }; if (productId in newCartItems) { newCartItems[productId].quantity += 1; newCartItems[productId].amount += price; } else { newCartItems[productId] = { name: productName, quantity: 1, amount: price, price: price, currency: currency, }; newUser.cartItems = { ...newCartItems }; startSession(newUser); }; const removeItemFromCart = () => { setQuantity((q) => q - 1); const newUser = { ...currentUser }; const newCartItems = { ...newUser.cartItems }; if (productId in newCartItems) { if (quantity > 1) { newCartItems[productId].quantity -= 1; newCartItems[productId].amount -= price; } else { delete newCartItems[productId]; } } newUser.cartItems = { ...newCartItems }; startSession(newUser); }; return ( <Card sx={{ borderRadius: 2, boxShadow: "0 2px 4px rgba(0, 0, 0, 0.1)", display: "flex", flexDirection: "column", justifyContent: "space-between", background: dark.card.backgroundSecondary, height: "400px", width: "300px", minWidth: "250px", cursor: "pointer", // Add cursor pointer on hover transition: "box-shadow 0.3s", // Add transition effect "&:hover": { boxShadow: "0 4px 8px rgba(250, 250, 250, 0.3)", // Whitish backdrop shadow on hover }, }} > <CardMedia component="img" height="0" image={imageUrl} title={productName} sx={{ height: "50%", objectFit: "cover" }} // Set the image proportion as desired /> <CardContent sx={{ height: "45%", overflowY: "scroll", "-ms-overflow-style": "none", // Hide scrollbar in IE and Edge "scrollbar-width": "none", // Hide scrollbar in Firefox "&::-webkit-scrollbar": { width: "0.5em", // Adjust scrollbar width as needed }, "&::-webkit-scrollbar-thumb": { backgroundColor: "transparent", // Hide scrollbar thumb }, }} > <Typography variant="h6" component="div" color={dark.text.primary}> {productName} </Typography> <Typography variant="body2" color={dark.text.secondary}> {description} </Typography> </CardContent> <CardContent sx={{ display: "flex", justifyContent: "space-between", alignItems: "center", fontWeight: "bold", position: "relative", color: dark.text.primary, px: 2, pt: 2, height: "5%", paddingBottom: 0, }} > {`\${price.toFixed(2)} \${currency}`} {quantity === 0 ? ( <Button variant="text" sx={{

```
color: dark.accent.highlight }} onClick={() => addItemToCart()} > Add to Cart </Button> ) : ( <QuantitySelector quantity={quantity} addItemToCart={() => addItemToCart()} removeItemFromCart={() => removeItemFromCart()} /> ) </CardContent> </Card> ); }; export default ProductCard;
```



Sure, here are tests for the ProductCard component:

```
javascript
import React from "react";
import { render, screen, fireEvent } from "@testing-library/react";
import ProductCard from "../ProductCard"; // Import the actual ProductCard component
import { dark } from "../themes"; // Import the dark theme
import { AppContext } from "../context/AppContext"; // Import the AppContext for mocking context

describe("ProductCard Component", () => {
  // Define a mock context value for testing
  const mockContextValue = {
    currentUser: {
      cartItems: {},
    },
    startSession: () => {},
  };

  it("should render with product details", () => {
    const productId = "123";
    const productName = "Sample Product";
    const description = "Product Description";
    const imageUrl = "sample-image.jpg";
    const price = 10.99;
    const currency = "USD";

    // Render the ProductCard component with props and mock context
    render(
      <AppContext.Provider value={mockContextValue}>
        <ProductCard
          productId={productId}
          productName={productName}
          description={description}
          imageUrl={imageUrl}
          price={price}
          currency={currency}
        />
      </AppContext.Provider>
    );

    // Query elements within the rendered component
    const productNameElement = screen.getByText(productName);
    const descriptionElement = screen.getByText(description);
    const priceElement = screen.getByText(`$${price.toFixed(2)} ${currency}`);
    const addToCartButton = screen.getByText("Add to Cart");

    // Assert that the product details are displayed
    expect(productNameElement).toBeInTheDocument();
    expect(descriptionElement).toBeInTheDocument();
    expect(priceElement).toBeInTheDocument();
    expect(addToCartButton).toBeInTheDocument();
  });

  it("should add and remove items from the cart", () => {
    const productId = "123";
    const productName = "Sample Product";
    const description = "Product Description";
    const imageUrl = "sample-image.jpg";
    const price = 10.99;
    const currency = "USD";

    // Render the ProductCard component with props and mock context
    render(
      <AppContext.Provider value={mockContextValue}>
        <ProductCard
          productId={productId}
          productName={productName}
          description={description}
          imageUrl={imageUrl}
          price={price}
          currency={currency}
        />
      </AppContext.Provider>
    );
  });
});
```

```

);

// Query the add to cart button
const addToCartButton = screen.getByText("Add to Cart");

// Click the add to cart button twice
fireEvent.click(addToCartButton);
fireEvent.click(addToCartButton);

// Query the quantity selector
const quantitySelector = screen.getByText("Quantity: 2");

// Assert that the quantity is displayed as 2
expect(quantitySelector).toBeInTheDocument();

// Query the remove button
const removeButton = screen.getByText("Remove");

// Click the remove button once
fireEvent.click(removeButton);

// Assert that the quantity is displayed as 1
expect(quantitySelector).toHaveTextContent("Quantity: 1");
});

// Add more test cases as needed
});

```

These test cases cover rendering the product details and adding/removing items from the cart in the ProductCard component. You can add more test cases to cover additional scenarios or behavior if needed.



You:

now write for this component: import React from "react"; import { Box, IconButton, Typography } from "@mui/material"; import { KeyboardArrowLeftOutlined, KeyboardArrowRightOutlined, } from "@mui/icons-material"; import { dark } from "../themes"; const QuantitySelector = ({ quantity, addItemToCart, removeItemFromCart }) => { const handleIncrement = () => { addItemToCart(); }; const handleDecrement = () => { removeItemFromCart(); }; return ( <Box sx={{ borderRadius: "8px", backgroundColor: dark.card.background, padding: "5px", }} > <IconButton onClick={handleDecrement} sx={{ padding: 0 }}> <KeyboardArrowLeftOutlined sx={{ fontSize: 24, backgroundColor: dark.accent.highlight, color: dark.background, borderRadius: "8px", }} /> </IconButton> <Typography variant="body2" component="span" sx={{ fontSize: 16, fontWeight: 500, padding: "2px 16px", }} > {quantity} </Typography> <IconButton onClick={handleIncrement} disabled={quantity === 10} sx={{ padding: 0 }} > <KeyboardArrowRightOutlined sx={{ fontSize: 24, backgroundColor: dark.accent.highlight, color: dark.background, padding: 0, borderRadius: "8px", }} /> </IconButton> </Box> ); }; export default QuantitySelector;



ChatGPT:

Certainly, here are tests for the QuantitySelector component:

```

javascript
import React from "react";
import { render, screen, fireEvent } from "@testing-library/react";
import QuantitySelector from "../QuantitySelector"; // Import the actual QuantitySelector component
import { dark } from "../themes"; // Import the dark theme

describe("QuantitySelector Component", () => {
  it("should render with initial quantity and buttons", () => {
    const quantity = 3;

    // Render the QuantitySelector component with props
    render(
      <QuantitySelector
        quantity={quantity}
        addItemToCart={() => {}}
        removeItemFromCart={() => {}}
      />
    );

    // Query elements within the rendered component
    const decrementButton = screen.getByRole("button", { name: "Decrement" });
    const quantityText = screen.getByText(quantity.toString());
    const incrementButton = screen.getByRole("button", { name: "Increment" });

    // Assert that the initial quantity and buttons are displayed
    expect(decrementButton).toBeInTheDocument();
    expect(quantityText).toBeInTheDocument();
    expect(incrementButton).toBeInTheDocument();
  });
});

```



```

});

it("should increment the quantity when the increment button is clicked", () => {
  const quantity = 3;
  const addItemToCart = jest.fn();

  // Render the QuantitySelector component with props
  render(
    <QuantitySelector
      quantity={quantity}
      addItemToCart={addItemToCart}
      removeItemFromCart={() => {}}
    />
  );

  // Query the increment button
  const incrementButton = screen.getByRole("button", { name: "Increment" });

  // Click the increment button
  fireEvent.click(incrementButton);

  // Assert that addItemToCart is called once
  expect(addItemToCart).toHaveBeenCalledTimes(1);
});

it("should decrement the quantity when the decrement button is clicked", () => {
  const quantity = 3;
  const removeItemFromCart = jest.fn();

  // Render the QuantitySelector component with props
  render(
    <QuantitySelector
      quantity={quantity}
      addItemToCart={() => {}}
      removeItemFromCart={removeItemFromCart}
    />
  );

  // Query the decrement button
  const decrementButton = screen.getByRole("button", { name: "Decrement" });

  // Click the decrement button
  fireEvent.click(decrementButton);

  // Assert that removeItemFromCart is called once
  expect(removeItemFromCart).toHaveBeenCalledTimes(1);
});

it("should disable the increment button when quantity is 10", () => {
  const quantity = 10;

  // Render the QuantitySelector component with props
  render(
    <QuantitySelector
      quantity={quantity}
      addItemToCart={() => {}}
      removeItemFromCart={() => {}}
    />
  );

  // Query the increment button
  const incrementButton = screen.getByRole("button", { name: "Increment" });

  // Assert that the increment button is disabled
  expect(incrementButton).toBeDisabled();
});

// Add more test cases as needed
});

```

These test cases cover rendering the quantity and buttons, incrementing and decrementing the quantity, and disabling the increment button when the quantity reaches 10 in the QuantitySelector component. You can add more test cases to cover additional scenarios or behavior if needed.



do the same for this component: import React, { useContext, useEffect } from "react"; import { AppContext } from "../context/AppContext"; import { Navigate, useNavigate } from "react-router-dom"; const ProtectedRoute = ({ children }) => { const { currentUser } = useContext(AppContext); const navigate = useNavigate(); useEffect(() => { if (!currentUser) { navigate("/auth"); } }, [currentUser]); return currentUser ? <>{children}</> : <Navigate to="/auth" />; }, export default ProtectedRoute;



Here are tests for the ProtectedRoute component:

```

javascript
import React from "react";
import { render, act } from "@testing-library/react";
import { MemoryRouter } from "react-router-dom";
import ProtectedRoute from "../ProtectedRoute"; // Import the actual ProtectedRoute component
import { AppContext } from "../context/AppContext"; // Import the AppContext

describe("ProtectedRoute Component", () => {
  it("should render children when currentUser is available", () => {
    const currentUser = { id: 1, username: "testuser" };

    // Render the ProtectedRoute component with currentUser
    const { getByText } = render(
      <MemoryRouter>
        <AppContext.Provider value={{ currentUser }}>
          <ProtectedRoute>
            <div>Protected Content</div>
          </ProtectedRoute>
        </AppContext.Provider>
      </MemoryRouter>
    );

    // Assert that the children are rendered
    expect(getByText("Protected Content")).toBeInTheDocument();
  });

  it("should navigate to '/auth' when currentUser is not available", async () => {
    const currentUser = null;

    // Render the ProtectedRoute component with currentUser as null
    const { container } = render(
      <MemoryRouter initialEntries={['/protected']}>
        <AppContext.Provider value={{ currentUser }}>
          <ProtectedRoute>
            <div>Protected Content</div>
          </ProtectedRoute>
        </AppContext.Provider>
      </MemoryRouter>
    );

    // Assert that the navigation to '/auth' occurs
    await act(async () => {
      expect(container.innerHTML).toBe("");
    });
  });
});

```

These test cases cover rendering the children when currentUser is available and navigating to "/auth" when currentUser is not available in the ProtectedRoute component.