CISC 327 Group 34 - CH

Assignment 5

November 25, 2024

Hayden Jenkins 20344883
Ryan Van Drunen 20331633
Madison MacNeil 20285877

Github Username – Name | Student Number

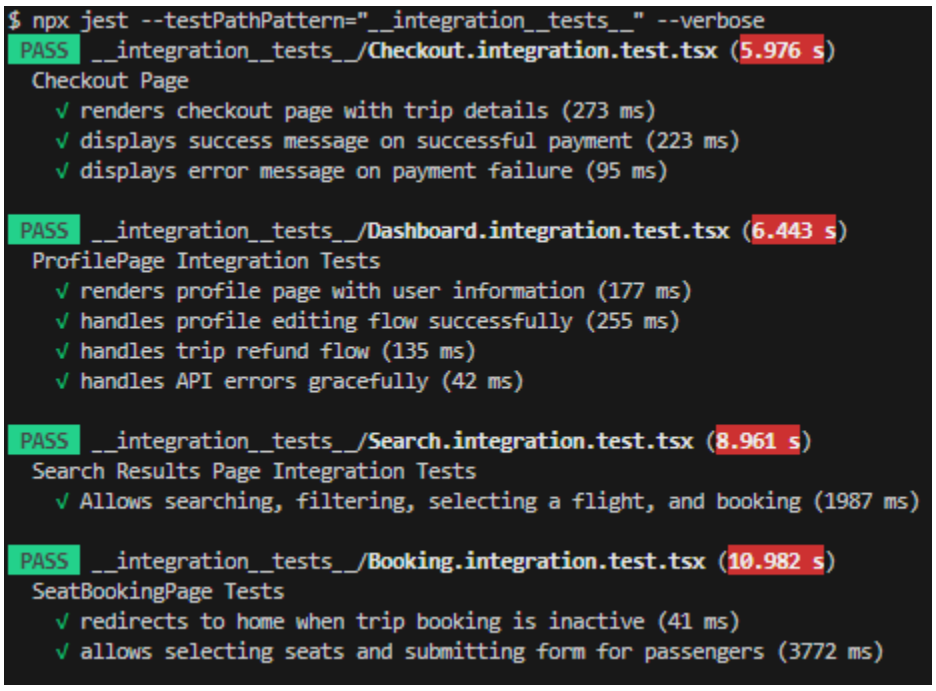
ryanvandrunen - Ryan Van Drunen | 20331633
hjenkins04/xonixdev - Hayden Jenkins | 20344883
madisonmacneil - Madison MacNeil | *20285877*

Task	Contributor
Checkout Integration Tests	Ryan Van Drunen
SeatBooking Integration Tests	Hayden Jenkins
Dashboard Integration Tests	Ryan Van Drunen
Search Filters Integration Tests	Madison MacNeil

How To Run Integration Tests

From the roam-frontend directory, run:

npx jest --testPathPattern="__integration__tests__"



Checkout Page Integration Test Cases

Renders checkout page with trip details

This test ensures that the Checkout component renders the trip details correctly by accessing the mock trip data set in the useTripStore. It checks that essential elements such as the "Complete Booking Payment" heading and passenger information (e.g., "John Doe") are visible on the page.

```
test("renders checkout page with trip details", async () => {
  render(<Checkout />);

  expect(screen.getByText("Complete Booking Payment")).toBeInTheDocument();
  expect(screen.getByText("John Doe")).toBeInTheDocument();
});
```

Displays success message on successful payment

This test validates the payment process for successful transactions. It simulates clicking the "Confirm and Pay" button, waits for a success message ("Success") to appear, and confirms that the "Home" button is displayed. Finally, it verifies navigation by checking if mockPush redirects to the "/dashboard" route.

```
test("displays success message on successful payment", async () => {
  render(<Checkout />);

  // Click the payment button
  const payButton = screen.getByText("Confirm and Pay");
  fireEvent.click(payButton);

  // Wait for and verify success message
  await waitFor(() => {
    expect(screen.getByText("Success")).toBeInTheDocument();
  });

  const homeButton = screen.getByText("Home");
  expect(homeButton).toBeInTheDocument();
  fireEvent.click(homeButton);

  // Verify that router.push was called
  await waitFor(() => {
    expect(mockPush).toHaveBeenCalledWith("/dashboard");
  });
});
```

Displays error message on payment failure

This test evaluates how the component handles payment failures. By mocking FetchBookingCheckout to throw an error, it simulates a failed payment process. The test ensures that an appropriate error message ("Failed") is displayed on the page.

```
test("displays error message on payment failure", async () => {
  // Mock FetchBookingCheckout to reject
  (FetchBookingCheckout as jest.Mock).mockRejectedValueOnce(
    new Error("Payment failed")
  );

  render(<Checkout />);

  const payButton = screen.getByText("Confirm and Pay");

  await act(async () => {
    fireEvent.click(payButton);
  });

  await waitFor(() => {
    expect(screen.getByText("Failed")).toBeInTheDocument();
  });
});
```

Dashboard Integration Test Cases

Renders profile page with user information

This test verifies the ProfilePage component correctly fetches and displays the user's profile information. It ensures that the API call fetchUserInfo is made with the correct user GUID and the resulting data is rendered on the page.

```
test("renders profile page with user information", async () => {
  render(<ProfilePage />);

  await waitFor(() => {
    expect(screen.getByText("John Doe")).toBeInTheDocument();
    expect(screen.getByText("john.doe@example.com")).toBeInTheDocument();
  });

  expect(fetchUserInfo).toHaveBeenCalledWith("user123");
});
```

Handles profiles editing flow successfully

This test validates the functionality of the profile editing feature. It simulates user interactions such as clicking the "Edit Profile" button, filling in the form fields, and submitting the updated profile information. The test confirms that the fetchUpdate API is called with the correct updated data and verifies the appearance of the success message.

```
test("handles profile editing flow successfully", async () => {
  (fetchUpdate as jest.Mock).mockResolvedValueOnce({});

  render(<ProfilePage />);

  // Click edit profile button
  const editButton = screen.getByText("Edit Profile");
  fireEvent.click(editButton);

  // Fill in the form
  const firstNameInput = screen.getByPlaceholderText("First Name*");
  const lastNameInput = screen.getByPlaceholderText("Last Name*");
  const emailInput = screen.getByPlaceholderText("Email Address*");
  const phoneInput = screen.getByPlaceholderText("Phone Number*");
  const dobInput = screen.getByPlaceholderText("Date of Birth*");
  const addressInput = screen.getByPlaceholderText("Street Address*");
  const provinceInput = screen.getByPlaceholderText("Province*");
  const postalCodeInput = screen.getByPlaceholderText("Postal Code*");

  fireEvent.change(firstNameInput, { target: { value: "Jane" } });
  fireEvent.change(lastNameInput, { target: { value: "Smith" } });
  fireEvent.change(emailInput, {
    target: { value: "jane.smith@example.com" },
  });
  fireEvent.change(phoneInput, { target: { value: "1234567890" } });
  fireEvent.change(dobInput, { target: { value: "1990/01/01" } });
  fireEvent.change(addressInput, { target: { value: "123 Main St" } });
  fireEvent.change(provinceInput, { target: { value: "Ontario" } });
  fireEvent.change(postalCodeInput, { target: { value: "A1A 1A1" } });

  // Submit form
  const submitButton = screen.getByText("Save");
  fireEvent.click(submitButton);

  // Verify success modal
  await waitFor(() => {
    expect(screen.getByText("Account Updated")).toBeInTheDocument();
  });

  expect(fetchUpdate).toHaveBeenCalledWith(
    "user123",
    "Jane",
    "Smith",
    "jane.smith@example.com",
    "1234567890"
  );
});
```

Handles trip refund flow

This test ensures that the trip refund feature works as intended. It simulates clicking the refund button for a trip, confirming the action, and displaying the success message after the API call. It also validates that the RemoveTripByGuid API is called with the correct trip GUID.

```
test("handles trip refund flow", async () => {
  (RemoveTripByGuid as jest.Mock).mockResolvedValueOnce({});

  render(<ProfilePage />);

  // Wait for trips to load
  await waitFor(() => {
    expect(screen.getByText("John Doe")).toBeInTheDocument();
  });

  // Click refund trip button
  const refundButton = screen.getByTestId("cancel-icon");
  fireEvent.click(refundButton);

  // Confirm refund
  const confirmButton = screen.getByText("Confirm");
  fireEvent.click(confirmButton);

  // Verify loader appears
  await waitFor(() => {
    expect(screen.getByText("Success")).toBeInTheDocument();
  });

  // Verify API call
  expect(RemoveTripByGuid).toHaveBeenCalledWith("trip123");
});
```

Handles API errors gracefully

This test checks how the component handles API failures, specifically when fetchTrips throws an error. It ensures that errors are logged to the console appropriately and that the application does not crash.

```
test("handles API errors gracefully", async () => {
  const error = new Error("API Error");
  (fetchTrips as jest.Mock).mockRejectedValueOnce(error);

  customRender(<ProfilePage />);

  await waitFor(() => {
    expect(console.error).toHaveBeenCalledWith(
      "Error fetching trips:",
      error
    );
  });
});
```

Search Result Integration Test Case

This integration-test tests the different components of the search results page to ensure correct interactions across the different functions and sub-components. Ensuring that all components work together to achieve consistent functionality and that data is passed correctly from one component to the next.

Renders the Search Box with Expected Values

```
//Confirm search input values render correctly
await waitFor(() => {
    expect(screen.getByTestId("departure-city-button")).toHaveTextContent("JFK");
    expect(screen.getByTestId("departure-city-button")).toHaveTextContent("New York");
    // Check arrival city button text
    expect(screen.getByTestId("arrival-city-button")).toHaveTextContent("LAX");
    expect(screen.getByTestId("arrival-city-button")).toHaveTextContent("Los Angeles");
    // Check departure date button text
    expect(screen.getByTestId("departure-date-button")).toHaveTextContent("Sun");
    expect(screen.getByTestId("departure-date-button")).toHaveTextContent("October");
    expect(screen.getByTestId("departure-date-button")).toHaveTextContent("20");
    // Check travelers and class button text
    expect(screen.getByTestId("travellers-button")).toHaveTextContent("2");
});
```

This section of the integration test ensures that the searchbox renders with the expected values as are passed to it from SearchData. Specifically, it ensures that the Departure City and Arrival City input boxes contain the text of both their IATA code and city name. It also confirms the display of the search date in words (“Sun”, “October”) for the data passed to it.

Confirms That Flights of All Price Points Render Before Filters Are Applied

```
// === TESTING FLIGHT RESULTS W/O FILTERS ===
await waitFor(() => {
    // Make sure "American Airlines" is rendered
    expect(screen.getAllByText("American Airlines")).toBeInTheDocument;

    //Make sure flights of all prices are rendered
    expect(screen.queryAllByText(/300/i)).toBeInTheDocument;
    expect(screen.queryByText(/900/i)).toBeInTheDocument();
    expect(screen.queryByText(/200/i)).toBeInTheDocument();
});
```

The second section of the search results integration test confirms that flights of all prices render as results upon the initial search. Specifically, flights of “$200”, “$300”, and “$900”.

Filters Flights by Price, Confirms no Flights >$200 Render Anymore

```
// === TESTING FILTERING ===
const airlineFilterButton = screen.getByTestId("filter-button-1");
await user.click(airlineFilterButton);

const dropdownList = await screen.findByTestId('dropdown-list');
await waitFor(() => {
    expect(dropdownList).toBeInTheDocument();
});

const dropdownOption = screen.getByTestId("dropdown-selection-0");
await user.click(dropdownOption);

await user.click(searchButton)

await waitFor(() => {
    expect(screen.queryByText('300')).not.toBeInTheDocument();
    expect(screen.queryByText('900')).not.toBeInTheDocument();
});
```

The third section of the search results integration test applies the maximum price filter, by clicking the filter button and selecting the option ‘$200’ from the dropdown. The search button is clicked and the test confirms that flights of prices above $200 no longer render as search results.

Confirms Accurate Rendering of Expanded Information from Selected Flight

```
await waitFor(() => {
    const laxElements = screen.getAllByText('LAX'); // Get all elements containing 'LAX'
    expect(laxElements[1]).toBeInTheDocument();

    const jfkElements = screen.getAllByText('JFK'); // Get all elements containing 'LAX'
    expect(laxElements[1]).toBeInTheDocument();

    expect(screen.getByText((content, element) => {
        return content.includes('John F.') && content.includes('Kennedy International Airport');
    })).toBeInTheDocument();

    expect(screen.getByText((content, element) => {
        return content.includes('Los Angeles') && content.includes('International Airport');
    })).toBeInTheDocument();

    expect(screen.getByTestId("right-arrow")).toBeInTheDocument()
    expect(screen.getByText(/Price: \$200/)).toBeInTheDocument()
    expect(screen.getByText(/Duration: 5h/)).toBeInTheDocument()
    expect(screen.getByText(/Baggage: 2 bags/)).toBeInTheDocument()
});
```

The fourth section of the search results integration test selects a flight by clicking on it and confirms that the expanded view of that flight information contains all the expected values of the chosen flight.

Routes to the Appropriate Next Page in the Program

```
    //Confirm use of Book My Ticket Now button
    const bookTicketButton = screen.getByText('Book My Ticket Now');
    expect(bookTicketButton).toBeInTheDocument();
    await fireEvent.click(bookTicketButton);

    expect(mockPush).toHaveBeenCalledWith('/seat-booking');
```

In the final section of the search results integration test, click the ‘Book My Ticket Now’ button in the bottom right corner of the flight information expanded view and confirm that the button routes the user (when logged in) to the seat-booking page.

Seat Booking Integration Test Case

Redirects to home when trip booking is inactive

The first test case ensures that if the session state has marked the current trip booking as inactive and the user visits the seat booking page, they are redirected to the home page.

```
test("redirects to home when trip booking is inactive", async () => {
  act(() => {
    const { setTripData } = useTripStore.getState();
    setTripData((prev) => ({
      ...prev,
      trip_booking_active: false,
    }));
  });

  render(<SeatBookingPage />);

  // Assert: Ensure that router.replace was called with "/"
  await waitFor(() => {
    const { tripData } = useTripStore.getState();
    expect(tripData.trip_booking_active).toBe(false);
    expect(mockReplace).toHaveBeenCalledWith("/");
  });
});
```

Allows selecting seats and submitting form for passengers

The second test, which is much larger and more complex, tests the different components and logic within the seat selection form components, aswell as their child components. It ensures the correct behavior of the rendering logic, passenger form completion and collection, and state synchronization. Aswell as the Seat reservation logic for return flights and multiple passengers. The test simulates user interactions to complete passenger forms, handles multiple passengers by resetting and storing data independently, and ensures transitions to the checkout page after all seats are reserved and passenger details are entered.

Set initial trip state data and render the page, checking that the page is loaded correctly with all the sub-components and state metadata.

```
// Arrange: Set initial trip data in the store
act(() => {
  const { setTripData } = useTripStore.getState();
  setTripData(mockTripData);
});

await waitFor(() => {
  const { tripData } = useTripStore.getState();
  expect(tripData.trip).not.toBeNull();
  expect(tripData.trip?.guid).toEqual("trip-guid");
});

// Arrange: Render SeatBookingPage and verify initial state
const user = userEvent.setup();
render(<SeatBookingPage/>);
const { tripData: initialTripData } = useTripStore.getState();

// Assert: Ensure initial trip states are set correctly
expect(initialTripData.trip_booking_active).toBe(true);
expect(initialTripData.trip_purchased).toBe(false);

// Assert: Ensure seat booking page and initial elements are present
await waitFor(() => {
  expect(screen.getByTestId("seat-booking")).toBeInTheDocument();
  expect(fetchFlightSeats).toHaveBeenCalledWith("flight-guid");
  expect(screen.getByTestId("seat-1-available")).toBeInTheDocument();
});
```

Simulate the first seat selection and ensure that the form header results correctly match the initial search criteria.

```
// Act: Select the first seat
const seatDeparture1 = screen.getByTestId("seat-1-available");
await act(async () => {
  fireEvent.click(seatDeparture1);
});

// Assert: Ensure booking form is displayed
expect(screen.getByTestId("booking-form-column")).toBeInTheDocument();

// Assert Form Header Details Match
await waitFor(() => {
  // Check departure city button text
  expect(screen.getByTestId("departure-city-button")).toHaveTextContent("JFK");
  expect(screen.getByTestId("departure-city-button")).toHaveTextContent("New York");
  expect(screen.getByTestId("departure-city-button")).toHaveTextContent("JFK Airport");
  // Check arrival city button text
  expect(screen.getByTestId("arrival-city-button")).toHaveTextContent("LAX");
  expect(screen.getByTestId("arrival-city-button")).toHaveTextContent("Los Angeles");
  expect(screen.getByTestId("arrival-city-button")).toHaveTextContent("LAX Airport");
  // Check departure date button text
  expect(screen.getByTestId("departure-date-button")).toHaveTextContent("Mon");
  expect(screen.getByTestId("departure-date-button")).toHaveTextContent("December");
  expect(screen.getByTestId("departure-date-button")).toHaveTextContent("25");
  // Check travelers and class button text
  expect(screen.getByTestId("traveller-class-button")).toHaveTextContent("2");
});
```

Enter the form fields for the first passenger, submit the form and validate that the trip store is updated correctly.

```
// Act: Fill in passenger form
fireEvent.change(screen.getByTestId("form-field-first-name"), { target: { value: firstPassengerDetails.firstName } });
fireEvent.change(screen.getByTestId("form-field-middle-name"), { target: { value: firstPassengerDetails.middleName } });
fireEvent.change(screen.getByTestId("form-field-last-name"), { target: { value: firstPassengerDetails.lastName } });
fireEvent.change(screen.getByTestId("form-field-prefix"), { target: { value: firstPassengerDetails.prefix } });
fireEvent.change(screen.getByTestId("form-field-passport-number"), { target: { value: firstPassengerDetails.passportNumber } });
fireEvent.change(screen.getByTestId("form-field-known-traveller-number"), { target: { value: firstPassengerDetails.knownTravellerNumber } });
fireEvent.change(screen.getByTestId("form-field-email"), { target: { value: firstPassengerDetails.email } });
fireEvent.change(screen.getByTestId("form-field-phone"), { target: { value: firstPassengerDetails.phone } });
fireEvent.change(screen.getByTestId("form-field-address"), { target: { value: firstPassengerDetails.address } });
fireEvent.change(screen.getByTestId("form-field-apt-number"), { target: { value: firstPassengerDetails.aptNumber } });
fireEvent.change(screen.getByTestId("form-field-province"), { target: { value: firstPassengerDetails.province } });
fireEvent.change(screen.getByTestId("form-field-zip"), { target: { value: firstPassengerDetails.zipCode } });
fireEvent.change(screen.getByTestId("form-field-emerg-first-name"), { target: { value: firstPassengerDetails.emergFirstName } });
fireEvent.change(screen.getByTestId("form-field-emerg-last-name"), { target: { value: firstPassengerDetails.emergLastName } });
fireEvent.change(screen.getByTestId("form-field-emerg-email"), { target: { value: firstPassengerDetails.emergEmail } });
fireEvent.change(screen.getByTestId("form-field-emerg-phone"), { target: { value: firstPassengerDetails.emergPhone } });


// Act: Submit the form for the first passenger
const nextPassengerButton = screen.getByTestId("next-passenger-button");
await act(async () => {
  fireEvent.click(nextPassengerButton);
});

// Assert: Verify the first passenger's data is stored correctly
const { tripData: firstPassengerFirstFlightData } = useTripStore.getState();
let firstPassenger = firstPassengerFirstFlightData.trip?.passengers[0];
expect(firstPassenger?.name).toBe(firstPassengerDetails.firstName);
expect(firstPassenger?.middle).toBe(firstPassengerDetails.middleName);
expect(firstPassenger?.last).toBe(firstPassengerDetails.lastName);
expect(firstPassenger?.prefix).toBe(firstPassengerDetails.prefix);
expect(firstPassenger?.passport_number).toBe(firstPassengerDetails.passportNumber);
expect(firstPassenger?.known_traveller_number).toBe(firstPassengerDetails.knownTravellerNumber);
expect(firstPassenger?.email).toBe(firstPassengerDetails.email);
expect(firstPassenger?.phone).toBe(firstPassengerDetails.phone);
expect(firstPassenger?.street_address).toBe(firstPassengerDetails.address);
expect(firstPassenger?.apt_number).toBe(firstPassengerDetails.aptNumber);
expect(firstPassenger?.province).toBe(firstPassengerDetails.province);
expect(firstPassenger?.zip_code).toBe(firstPassengerDetails.zipCode);
expect(firstPassenger?.emerg_name).toBe(firstPassengerDetails.emergFirstName);
expect(firstPassenger?.emerg_last).toBe(firstPassengerDetails.emergLastName);
expect(firstPassenger?.emerg_email).toBe(firstPassengerDetails.emergEmail);
expect(firstPassenger?.emerg_phone).toBe(firstPassengerDetails.emergPhone);
```

Ensure that the seat is marked as reserved, and select a new available seat for the second passenger.

```
await waitFor(() => {
  expect(screen.getByTestId("seat-1-reserved")).toBeInTheDocument();
  expect(screen.getByTestId("seat-2-available")).toBeInTheDocument();
});

const seatDeparture2 = screen.getByTestId("seat-2-available");
await act(async () => {
  fireEvent.click(seatDeparture2);
});
```

Repeat form submission and validation tests again for the second passenger… Submit the form again and ensure that the current flight is changed to the return flight.

```
// Submit the form for the second passenger
const returnFlightButton = screen.getByTestId("book-return-flight-button");
await act(async () => {
  fireEvent.click(returnFlightButton);
});

// Assert Form Header Details Match
await waitFor(() => {
  // Check departure city button text
  expect(screen.getByTestId("departure-city-button")).toHaveTextContent("LAX");
  expect(screen.getByTestId("departure-city-button")).toHaveTextContent("Los Angeles");
  expect(screen.getByTestId("departure-city-button")).toHaveTextContent("LAX Airport");
  // Check arrival city button text
  expect(screen.getByTestId("arrival-city-button")).toHaveTextContent("JFK");
  expect(screen.getByTestId("arrival-city-button")).toHaveTextContent("New York");
  expect(screen.getByTestId("arrival-city-button")).toHaveTextContent("JFK Airport");
  // Check departure date button text
  expect(screen.getByTestId("departure-date-button")).toHaveTextContent("Sun");
  expect(screen.getByTestId("departure-date-button")).toHaveTextContent("December");
  expect(screen.getByTestId("departure-date-button")).toHaveTextContent("31");
  // Check travelers and class button text
  expect(screen.getByTestId("traveller-class-button")).toHaveTextContent("2");
});
```

Select an available seat for the first passenger and ensure that the form details are prefilled from the previous departure flight.

```
// Act: Select the first seat
const seatReturn1 = screen.getByTestId("seat-1-available");
await act(async () => {
  fireEvent.click(seatReturn1);
});

await waitFor(() => {
  expect(screen.getByTestId("form-field-first-name")).toHaveValue(firstPassengerDetails.firstName);
  expect(screen.getByTestId("form-field-middle-name")).toHaveValue(firstPassengerDetails.middleName);
  expect(screen.getByTestId("form-field-last-name")).toHaveValue(firstPassengerDetails.lastName);
  expect(screen.getByTestId("form-field-prefix")).toHaveValue(firstPassengerDetails.prefix);
  expect(screen.getByTestId("form-field-passport-number")).toHaveValue(firstPassengerDetails.passportNumber);
  expect(screen.getByTestId("form-field-known-traveller-number")).toHaveValue(firstPassengerDetails.knownTravellerNumber);
  expect(screen.getByTestId("form-field-email")).toHaveValue(firstPassengerDetails.email);
  expect(screen.getByTestId("form-field-phone")).toHaveValue(firstPassengerDetails.phone);
  expect(screen.getByTestId("form-field-address")).toHaveValue(firstPassengerDetails.address);
  expect(screen.getByTestId("form-field-apt-number")).toHaveValue(firstPassengerDetails.aptNumber);
  expect(screen.getByTestId("form-field-province")).toHaveValue(firstPassengerDetails.province);
  expect(screen.getByTestId("form-field-zip")).toHaveValue(firstPassengerDetails.zipCode);
  expect(screen.getByTestId("form-field-emerg-first-name")).toHaveValue(firstPassengerDetails.emergFirstName);
  expect(screen.getByTestId("form-field-emerg-last-name")).toHaveValue(firstPassengerDetails.emergLastName);
  expect(screen.getByTestId("form-field-emerg-email")).toHaveValue(firstPassengerDetails.emergEmail);
  expect(screen.getByTestId("form-field-emerg-phone")).toHaveValue(firstPassengerDetails.emergPhone);
});
```

Repeat the same thin again for the second passenger and ensure you are now redirected to the checkout page.

```
// Act: Submit form
const CheckoutButton = screen.getByTestId("checkout-button");
await act(async () => {
  fireEvent.click(CheckoutButton);
});
```