CPSC 2650 Assignment 3: Testing React Apps

In this assignment, you'll write a set of unit and integration tests for the MDN Todo List React app using **Vitest** and **@testing-library/react**.

Learning goals:

By the end of this assignment, you should be able to:

- Configure and run Vitest tests in a Vite-powered React project
- Write meaningful unit and integration tests for React components
- Simulate user events and test state changes
- Query and assert DOM nodes using Testing Library

Setup:

Use the official MDN React Todo List app as the base project | GitHub repo: MDN Todo List

Task 0: Setup & Installation

1. Clone the base project:

```
git clone https://github.com/mdn/todo-react.git
cd todo-react
```

2. Install dependencies:

```
npm install
```

Note: If you get a permission error, you may need to use "`sudo npm install`"

3. Add testing tools:

```
npm install -D vitest jsdom @testing-library/react @testing-library/jest-dom
```

Note: The -D flag indicates that you want to install a development dependency.

4. In vite.config.js, add (as a separate object-value pair after plugins, base):

```
test: {
    globals: true,       // Enables test, expect, etc. globally
    environment: 'jsdom' // Needed for DOM in tests
}
```

❓ Check Your Understanding

- Why do we use the `-D` flag while installing the packages related to testing in Task 0, Step 3? Why not just install like every other package without the `-D` flag?

Task 1: Test Initial Rendering

This test ensures the app correctly renders the 3 todo items provided through props when it first loads.

- In a real app, this simulates loading saved tasks from a database or localStorage.
- By rendering the <App /> component with a test DATA array, you check whether each todo's name appears on the screen.
- This confirms that the component receives and displays props properly

Create a test file: `src/App.test.jsx`

1. Import tools:

```
import { render, screen } from '@testing-library/react';

import App from './App';
```

2. Write a test:

```
/*This needs to be copied over from main.jsx and used in the test*/

const DATA = [

  { id: 'todo-0', name: 'Eat', completed: true },

  { id: 'todo-1', name: 'Sleep', completed: false },

  { id: 'todo-2', name: 'Repeat', completed: false },

];


test('renders initial todos', () => {

    render(<App tasks={DATA} />);

    const eatElements = screen.getAllByText('Eat');

    expect(eatElements.length).toBeGreaterThan(0);
    const sleepElements = screen.getAllByText('Sleep');

    expect(sleepElements.length).toBeGreaterThan(0);

    const repeatElements = screen.getAllByText('Repeat');

    expect(repeatElements.length).toBeGreaterThan(0);

});
```

3. Run the test:

```
npx vitest run
```

❓ Check Your Understanding

- What is the purpose of this test?
- What would happen if we didn't pass any props to the App component in the test?
- Is this a unit or integration test? Why?

Task 2: Add Todo Test

Here, you'll simulate user interaction by typing into the input field and clicking the 'Add' button. This test ensures that the form works and that a new todo item appears on screen.

Before we simulate typing and form submission, we need to install another development dependency:

```
npm install -D @testing-library/user-event
```

```
/* Add this to top with the other imports*/
import userEvent from '@testing-library/user-event';


test('adds a new todo', async () => {
    render(<App tasks={DATA} />);
    const input = screen.getByRole('textbox');
    // Find the "Add" button by its label (case-insensitive match).
    const addButton = screen.getByRole('button', { name: /add/i });
    await userEvent.type(input, 'Eat a Pizza slice');
    await userEvent.click(addButton);

    const todoItems = screen.getAllByRole('listitem');
    const match = todoItems.find(item =>
                            item.textContent.includes('Eat a Pizza slice'));
    expect(match).toBeTruthy();
});
```

Run the test to make sure it passes:

```
npx vitest run
```

❓ Check Your Understanding

- What does this test simulate? Please be specific.
- What does **userEvent.type()** do in this test?
- How do we confirm that the new todo was added successfully?
- Is this a unit or integration test? Why?

Task 3: Toggle Completion Test

This step tests whether a todo item's completed state can be toggled by clicking its checkbox. You'll inspect the checkbox's checked state before and after the click to ensure it's updating properly.

```
 test('toggles a todo complete', async () => {
    render(<App tasks={DATA} />);
    const eatCheckbox = screen.getByLabelText('Eat');
    expect(eatCheckbox).toBeInstanceOf(HTMLInputElement);
    expect(eatCheckbox.checked).toBe(true); // Initially completed

    await userEvent.click(eatCheckbox);
    expect(eatCheckbox.checked).toBe(false); // Should now be unchecked
  });
```

Run the test to make sure it passes:

```
npx vitest run
```

❓ Check Your Understanding

- What does this test simulate? Please be specific.
- Why do we check the **checked** property of the checkbox?
- What happens if the checkbox isn't found by **getByLabelText()**?
- Is this a unit or integration test? Why?

Task 4: Filter Buttons

Now you'll test the 'Active' filter. When clicked, the app should hide completed tasks and only show todos that are not marked as done.

Test: Click the 'Active' filter and verify that only incomplete todos appear:

```
test('filters by active todos', async () => {
   render(<App tasks={DATA} />);
   await userEvent.click(screen.getByRole('button', { name: /active/i }));

   const listItems = screen.getAllByRole('listitem');

   expect(listItems.length).toBe(2);
   const listTexts = listItems.map(item => item.textContent);
   expect(listTexts.some(text => text.includes('Sleep'))).toBe(true);
   expect(listTexts.some(text => text.includes('Repeat'))).toBe(true);
   expect(listTexts.some(text => text.includes('Eat'))).toBe(false);
 });
```

Run the test to make sure it passes:

```
npx vitest run
```

❓ Check Your Understanding

- What does this test simulate? Please be specific.
- How does this test check which todos are visible?
- Why do we check for the presence/absence of specific task names?
- Is this a unit or integration test? Why?

Task 5: Delete Todo

This step checks whether the delete functionality works. After clicking a todo's 'Delete' button, it should no longer appear in the list.

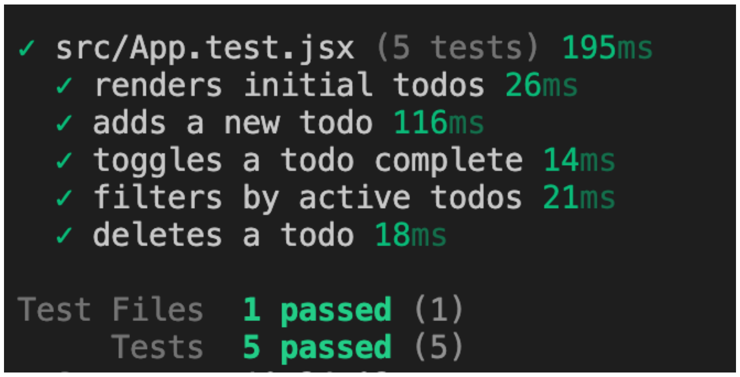
Test: Click the delete button and assert removal:

```
test('deletes a todo', async () => {
   render(<App tasks={DATA} />);
   const deleteBtn = screen
        .getAllByRole('button')
        .find(btn => btn.textContent.includes('Delete')
          && btn.textContent.includes('Eat'));
   await userEvent.click(deleteBtn);
   const listItems = screen.getAllByRole('listitem');
   const itemTexts = listItems.map(item => item.textContent);
   expect(itemTexts.some(text => text.includes('Eat'))).toBe(false);
 });
```

Run the test to make sure it passes:

```
npx vitest run
```

After completing Tasks 0-6, you should see the following after running the tests:



```
✓ src/App.test.jsx (5 tests) 195ms
  ✓ renders initial todos 26ms
  ✓ adds a new todo 116ms
  ✓ toggles a todo complete 14ms
  ✓ filters by active todos 21ms
  ✓ deletes a todo 18ms

Test Files  1 passed (1)
     Tests  5 passed (5)
```

❓ Check Your Understanding

- What does this test simulate? Please be specific.
- What's the key assertion in this test?
- How does the test identify the correct delete button?
- Is this a unit or integration test? Why?

Task 6: Write the following tests:

Now that you have some experience with writing tests, you will use that knowledge to implement the following tests from scratch:

- **Completed Filter**
  - Test that the “Completed” filter only shows tasks marked as complete.
  - What to test:
    1. After clicking the “Completed” button, only tasks with *completed: true* should be visible.
- **Mark All Todos Complete**
  - Test that a user can mark all incomplete todos as complete.
  - What to test:
    1. Click on all unchecked checkboxes.
    2. Confirm that the checked state of each checkbox becomes true.
- **Edit Button Visibility**
  - Test that each todo item shows an “Edit” button next to it.
  - What to test:
    1. That there are three visible “Edit” buttons when three todos are passed.

Task 7: Written responses and new `<Route path=”Task7”/>`

- Create a new `<Route>` called “Task 7” that can be accessed via the `<Navbar/>` and the path `/Task7`
  - This route should display answers to all the written response questions from Tasks 0 - 6. Paste screenshots where appropriate.
- Note that you need to install `react-router-dom` and refactor the app for routing to work as intended (refer to last week’s assignment).

Hand In:

Zip your assignment directory, submit your work in the Assignment 3 folder, and hand it in.

- When you zip your directory, remember to exclude the node_modules folder. The marker will download your submission, install the node modules and run your app.

Checklist:

- **[2.5 marks] Tasks 0-5** All required test cases for Steps 1-5 are implemented correctly
- **[4.5 marks] Task 6**: Student-Written Tests
    - Edit Button Visibility
        - Confirms each todo has a visible “Edit” button
    - Mark All Todos Complete
        - Loops through incomplete tasks and marks them complete
        - Asserts each checkbox is now checked
    - Completed Filter
        - Clicks the “Completed” filter
        - Asserts that only completed tasks are shown
- **[3 marks] Task 7: Reflection Questions for Tasks 0-6 and Routing**
    - Write brief answers to each question in the “Check your understanding” section after each task.
    - Created a separate Route to display answers as per instructions
    - Created and rendered <Navbar/> component to link <Home/> and <Task 7/>

Total: 10 marks

References and Resources:

1. https://vitest.dev
2. https://testing-library.com/docs/react-testing-library/intro/
3. https://github.com/mdn/todo-react
4. https://kentcdodds.com/blog/common-mistakes-with-react-testing-library
5. https://testingjavascript.com/