|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Description** | **Use Case** | **Practice Tip** |
| **render()** | Renders a component for testing | To bring the component into the test environment | Start with **render(<MyComponent />)** and test if it's in the document |
| **screen.getByText()** | **💥 If found return element**  Finds element by text (throws if not found)  **💥 If nothing is found:**  It throws an error like:  ⚠️❗ | 1. You want to locate something based on its text — like paragraph text, labels, etc. 2. Check if a button, heading, or any text content exists | Render a form and test if a “Submit” button is present  **i) screen.getByText('Submit') or**  **ii) screen.getByText(/submit/i) (using regex to search name)** |
| **screen.getByRole()** | **💥 If found return element**  Finds element by accessibility role  **💥 If nothing is found:**  It throws an error like:  ⚠️❗ | 1. Useful for buttons, headings, checkboxes, etc. 2. Semantic role + accessible name 3. Complex components, broader matching 4. Element must have correct role/name | **Find a button by role:**  **screen.getByRole('button', { name: /submit/i }) or**  **screen.getByRole('button')or**  **screen.getByRole(textbox, { name: /name/i }) (by label)** |
| **screen.queryByRole('alert')** | **💥 If found return element**  Finds element by accessibility role  **💥 If nothing is found:**  return **null** | Use when the element might not be present, like:   1. Conditional rendering 2. Testing for absence | **expect(screen.queryByRole('alert')).not.toBeInTheDocument();** |
| **screen.queryByText()** | Like getByText, but returns null if not found (no error)  **💥 If nothing is found:**  return **null** | Check if element is **not** in the document | After submitting, check **expect(queryByText(/loading/i)).not.toBeInTheDocument()**  “I expect that this element is not in the DOM.” |
| **screen.getByLabelText()** | **💥 If found return element**  Finds form element associated with a <label>  **💥 If nothing is found:**  It throws an error like: ⚠️❗ | 1. Ideal for inputs and form fields 2. **basic form inputs** — simple and clean. | Test input field by label:  **screen.getByLabelText('Email')** |
| **screen.getByPlaceholderText()** | **💥 If found return element**  Finds element by placeholder text  **💥 If nothing is found:**  It throws an error like:⚠️❗ | Common for <input> or <textarea> | **screen.getByPlaceholderText('Enter your name')** |
| **screen.getByTestId()** | Finds element using data-testid attribute | When no better option is available | Add data-testid="submit-btn" and test with  **screen.getByTestId('submit-btn')** |
| **screen.findByText()** | Async version of getByText – returns a Promise | For async content (e.g. API data rendering) | **await scree.findByText('Loaded successfully')** |
| **fireEvent.click()** | fireEvent.click() simulates a click event on an element. It's a low-level method that directly triggers the click event on a DOM element. | It's useful when you need to simulate basic browser events, but it does not simulate real user behavior (such as mouse movement or focus). | **fireEvent.click(getByText('Submit'))** |
| **userEvent.click()** | userEvent.click() simulates a real user interaction — it mimics the behavior of a real user clicking on an element, including things like focus, mouse events, and more. | It's recommended over fireEvent.click() because it simulates user actions more realistically, ensuring better testing for how your application handles user interactions. | **await userEvent.click(screen.getByRole('button'))** |
| **userEvent.type()** | It automatically simulates **keypress events** for each character in the text and also handles things like **focus**, **input value updates**, and **events** like onChange. | Use userEvent.type() when you want to simulate a user typing specific text into an input field, just like they would in a real browser. | **await userEvent.type(input, 'myUsername');**    **expect(input).toHaveValue('myUsername');** |
| **userEvent.keyboard()** | Simulates a **keyboard event**, sending **one or more key events** (like keydown, keyup, etc.) to a target element. It’s more flexible than type() because you can simulate things like **modifier keys** (e.g., Shift, Ctrl, etc.) or even **complex key combinations**. | Use userEvent.keyboard() when you need to simulate a more complex keyboard interaction, like pressing a single key, multiple keys, or key combinations (like Ctrl + A). | **// Simulate typing**  **await userEvent.keyboard('myUsername');**  **// Simulate Backspace then typing '123'**  **await userEvent.keyboard('{Backspace}{Backspace}123');**  **expect(input).toHaveValue('myUser123');** |
| **waitFor()** | Waits for assertion to pass within timeout | For async assertions or state changes | **await waitFor(() => expect(screen.getByText('Done')).toBeInTheDocument())** |
| **within()** | Restricts queries to a specific element | When testing nested components | **within(screen.getByTestId('card')).getByText('Card Title')** |

**Practice Ideas**

1. **Simple Form Test**
   * Render a form with email and password inputs
   * Test typing, submission, and success message
2. **Todo List App**
   * Add todo
   * Mark as done
   * Remove item
   * Filter by status
3. **API Data Component**
   * Mock an API call
   * Wait for loading and check rendered data
   * Use findByText and waitFor
4. **Conditional Rendering**
   * Toggle content visibility on a button click
   * Use queryByText to check content is hidden
5. **Modal Component**
   * Open modal on button click
   * Test inside modal using within()