# CS 4530: Fundamentals of Software Engineering

## Lesson 7.1 Testing User Interfaces

---

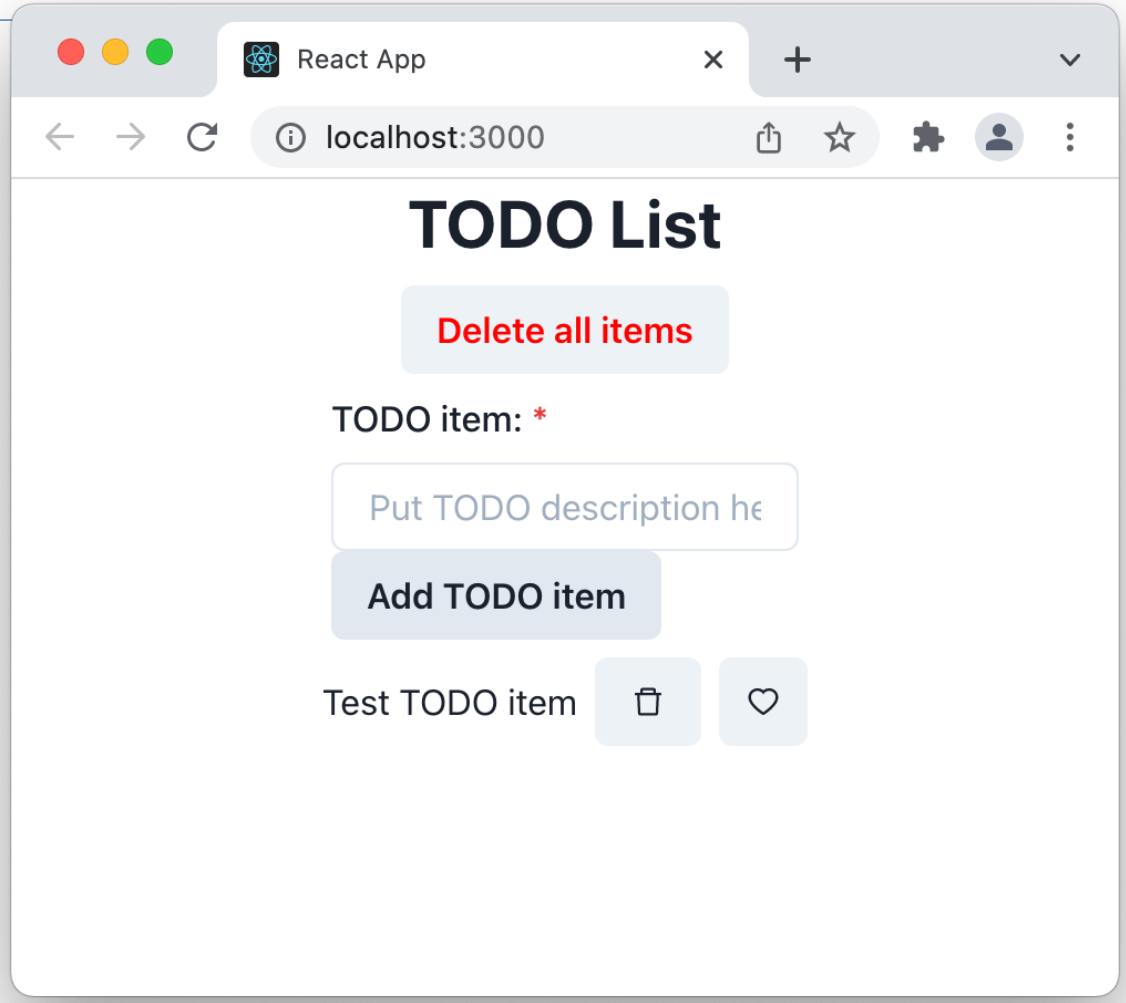
Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand
Khoury College of Computer Sciences

# Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - Be able to map the three core steps of a test (construct, act, check) to UI component testing
  - Understand the tradeoff between designing UIs for testability designing tests for UIs
  - Be able to write component-level test for React using Jest

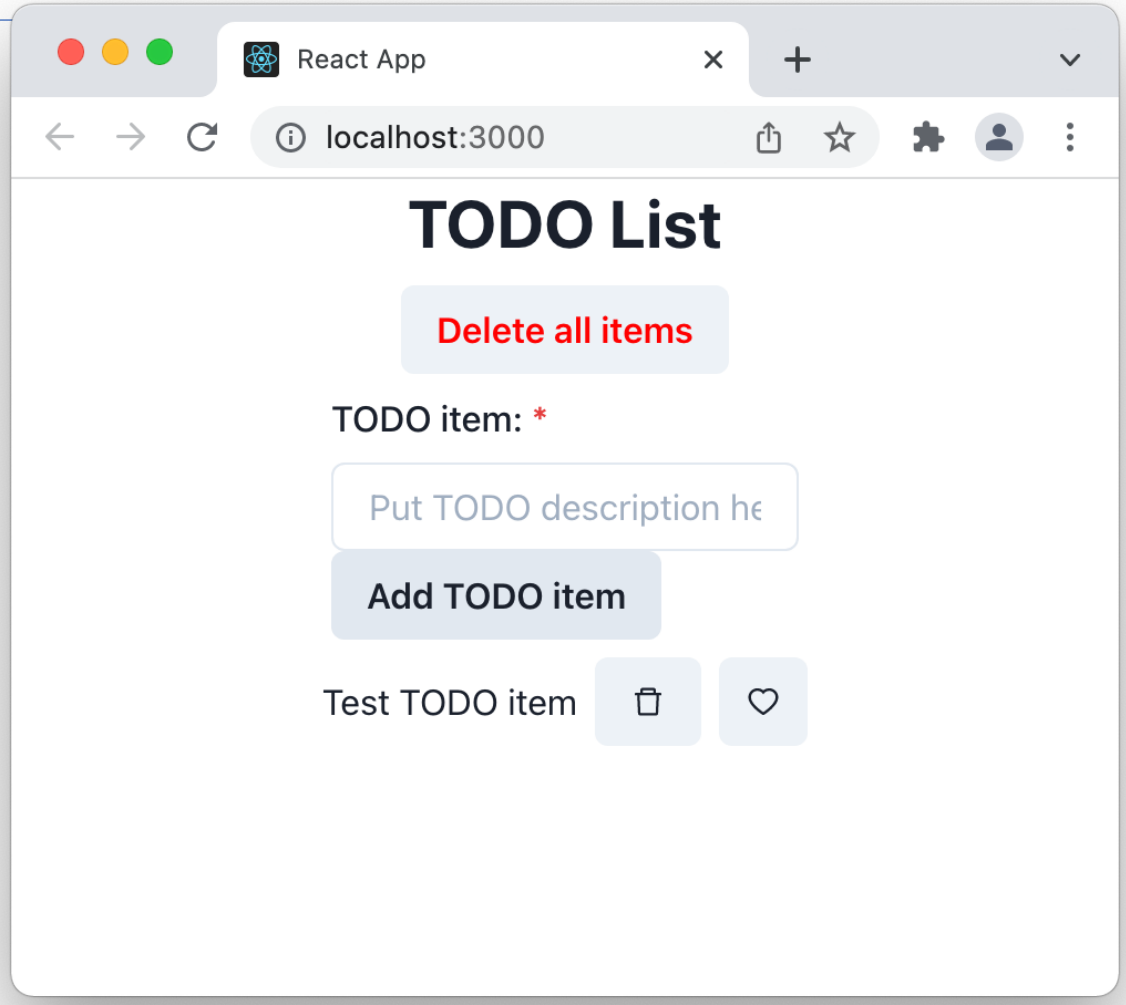
# How do we test this TODO App?

```
export const TodoApp = () => {
  const [items, setItems] = useState<TodoItem[]>([]);
  const { register, handleSubmit, reset } = useForm<FormContents>();
  const addTodoItem = useCallback(
      (contents: FormContents) => {
        if (contents.itemDesc) {
          const newItem = { title: contents.itemDesc, id: nanoid() };
          setItems((oldItems) => oldItems.concat(newItem));
          reset();
        }
      },
      [setItems, reset]
  );
  const onSubmit = handleSubmit(addTodoItem);
  return (
      <VStack>
        <Heading>TODO List</Heading>
        <Box>
          <Button color="red" data-testid="deleteAllButton" onClick={deleteAllItems}>
            Delete all items
          </Button>
        </Box>
        <form onSubmit={onSubmit}>
          <FormControl isRequired>
            <FormLabel>TODO item:</FormLabel>
            <Input
                placeholder="Put TODO description here"
                {...register("itemDesc")}
            />
          </FormControl>
          <Button type="submit">
            Add TODO item
          </Button>
        </form>
        {items.map((theItem) => (
            <TodoItemComponent
                item={theItem}
                key={theItem.id}
                deleteItem={() => {
                  setItems((oldItems) => oldItems.filter((i) => i !== theItem));
                }}
            />
        ))}
      </VStack>
  );
};
```



# Record/Replay Tools Enable Browser-Based Testing

- Tools like Selenium automate testing apps in the browser by recording interactions, replaying them, checking that result visually matches
- Strengths of this approach:
  - “Easy”
  - End-to-end
- Weaknesses of this approach:
  - Brittle – tests break when UI changes
  - Impossible to unit-test
  - Slow



# Write UI component tests just like any other test

*Follow the generic testing model from Lesson 5.1:*

- Construct the situation:
  - Set up SUT to get the state ready
  - [Optional: Prepare collaborators]
- Apply the operation inputs.
- Check the outputs, verify the state change, handle the behavior
  - Handle exceptions,
  - Time-Out to handle nontermination,
  - Post-check with collaborators.

1: Render component into a testing DOM tree

2: Interact with the rendered component

3: Check the rendered result

# UI Testing Libraries make Component Tests Lightweight

- Render components into a “virtual DOM”
  - Just like browser would, but no browser
- Interact with components by “firing events” like a user would
  - Click, enter text, etc. on DOM nodes, just like a user would in a browser
- Inspect components that are rendered
  - Tests specify how to “find” a component in that virtual DOM

“Testing Library”
https://testing-library.com
Compatible with many UI libraries and many testing frameworks

# Rendering Components in Virtual DOM

```
let renderedComponent: RenderResult;
beforeEach(() => {
  renderedComponent = render(<TodoApp />);
});
```

- The *render* function prepares our component for testing:
  - Creates a virtual DOM
  - Instantiates our component, mounts it in DOM
  - Mocks all behavior of the core of React
  - We use the *RenderResult* returned by *render* to interact with the component

https://testing-library.com/docs/react-testing-library/api#render

# Inspecting Rendered Components: TestIDs

**SUT**

```
<Button color="red" data-testid="deleteAllButton" onClick={deleteAllItems}>
  Delete all items
</Button>
```

**Test**

```
let renderedComponent: RenderResult;
beforeEach(() => {
  renderedComponent = render(<TodoApp />);
  let deleteAllButton = renderedComponent.getByTestId("deleteAllButton")
});
```

First approach to inspect rendered components: add `data-testid` to component, use `getByTestId`

# Inspecting Rendered Components: ARIA Role

**SUT**

```
<Button type="submit">
  Add TODO item
</Button>
```

**Test**

```
let renderedComponent: RenderResult;
beforeEach(() => {
  renderedComponent = render(<TodoApp />);
let newItemButton = renderedComponent.getByRole("button",
                                                {name: "Add TODO item"});
});
```

The ARIA role of a DOM component indicates how a screen-reader or other assistive device will represent the interface to an end-user. Chakra-UI provides the roles on all of its components out-of-the-box.

# 3 Tiers for Inspecting Rendered Components

1. How every user interacts with your app
2. How some users interact with your app
3. How only your test interacts

- Just like “good tests use public APIs”, good UI tests interact like a user would

# 3 Tiers for Inspecting Rendered Components

- Queries that reflect how every users interacts with your app
  - byRole – Using accessibility tree
  - byLabelText – Using label on form fields
  - byPlaceHolderText – Using placeholder text on form field
  - byText – By exact text in an element
  - byDisplayValue – By current value in a form field
- Queries that reflect how some users interact with your app
  - byAltText – By alt text, usually not presented to sighted users
  - byTitle - By a “title” attribute, usually not presented to sighted users
- Queries that have nothing to do with how a user interacts with app
  - byTestId

More: https://testing-library.com/docs/queries/about

# Acting on Rendered Components: *userEvent*

- Testing Library provides userEvent.<event> methods
  - ```
    userEvent.type(newItemTextField, "Write a better test input");
    userEvent.click(newItemButton);
    ```
    Also: change, keyDown, keyUp, etc
- These methods simulate user behavior:
  - Before clicking: MouseOver, MouseMove, MouseDown, MouseUp
  - Type will click the text box, then provide characters one-at-a-time
    ```

# Example Test: Unit Test TodoItemComponent

**Goals: Test that item title is rendered, test that clicking on delete button calls deleteItem**
**Strategy: Render component, find the item title, find the delete button. Click the button.**

```
export const TodoItemComponent: React.FunctionComponent<{
  item: TodoItem;
  deleteItem: () => void;
}> = ({ item, deleteItem }) => {
...
 return (
     <HStack>
        <Text data-testid='todoItem'>{item.title}</Text>
        <Button onClick={deleteItem} aria-label="delete">
          <AiOutlineDelete />
        </Button>
        {likeButton}
      </HStack>
   );
};
```

# Example Test: Unit Test TodoItemComponent

**Goals: Test that item title is rendered, test that clicking on delete button calls deleteItem**
**Step 1: Setup – Render the component with a todo item and a mock delete handler**

```
let itemTitleText: string;
let renderedComponent: RenderResult;
let mockDeleteItem = jest.fn();
beforeEach(() => {
  itemTitleText = "Some Todo Item";
  renderedComponent = render(
    <TodoItemComponent
      item={{ title: itemTitleText, id: 'someID' }}
      deleteItem={mockDeleteItem}
    />
  );
  mockDeleteItem.mockClear();
});
```

# Testing for Item Text: Is the itemTitleText in the component?

```
export const TodoItemComponent: React.FunctionComponent<{
  item: TodoItem;
  deleteItem: () => void;
}> = ({ item, deleteItem }) => {
...
 return (
     <HStack>
        <Text data-testid='todoItem'>{item.title}</Text>
        <Button onClick={deleteItem} aria-label="delete">
          <AiOutlineDelete />
        </Button>
        {likeButton}
      </HStack>
   );
};
```

```
it("Displays the item title exactly as specified", () => {
  expect(renderedComponent.getByText(itemTitleText))
        .toBeDefined();
});
```

```
it("Displays the item title exactly as specified", () => {
  expect(renderedComponent.getByTestId("todoItem"))
        .toHaveTextContent(itemTitleText);
});
```

Note the subtle distinction between these two tests

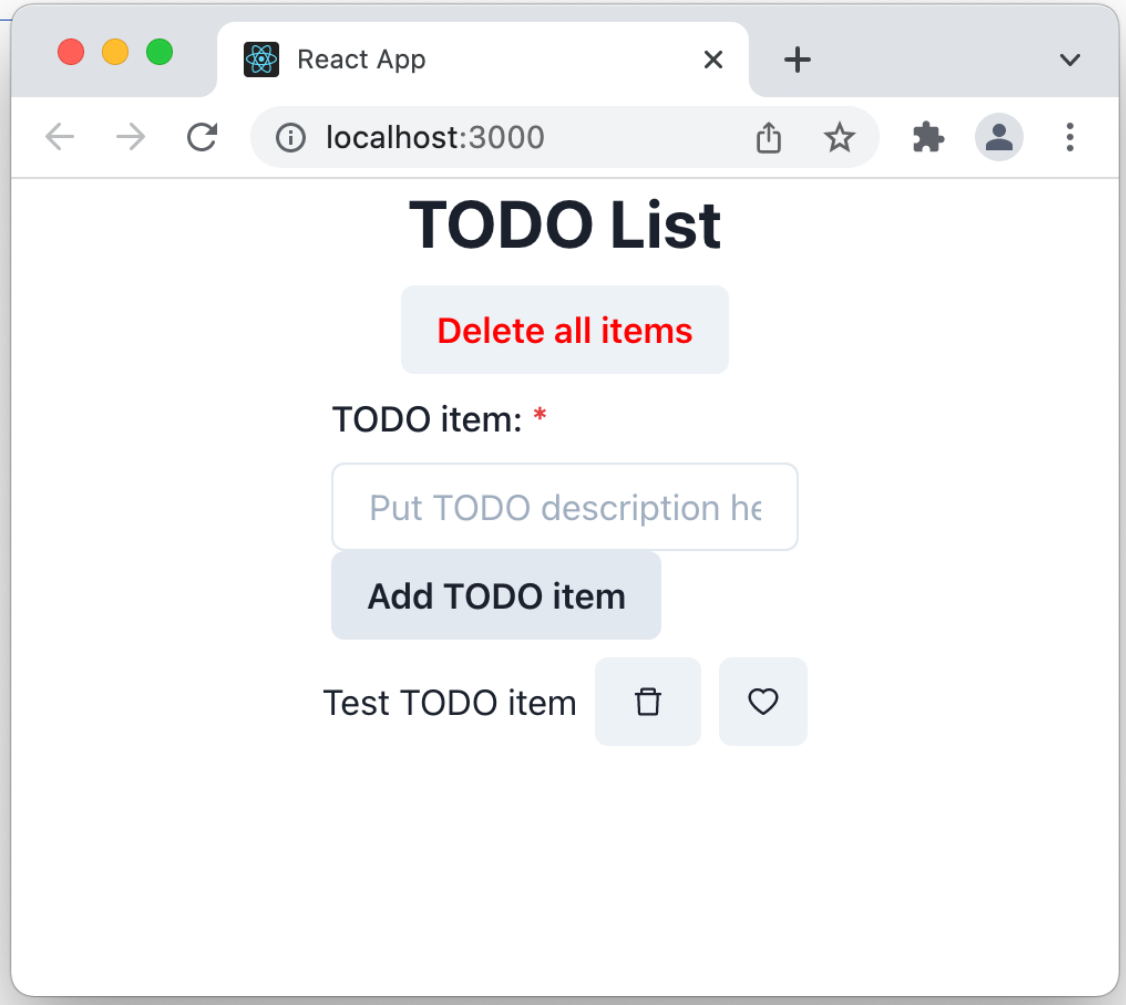
# Testing for item deletion

```
let itemTitleText: string;
let renderedComponent: RenderResult;
let mockDeleteItem = jest.fn();
beforeEach(() => {
  itemTitleText = "Some Todo Item";
  renderedComponent = render(
    <TodoItemComponent
      item={{ title: itemTitleText, id: 'someID' }}
      deleteItem={mockDeleteItem}
    />
  );
  mockDeleteItem.mockClear();
});
```

```
it("Calls the deleteItem handler when the delete button is clicked", () => {
  userEvent.click(renderedComponent.getByLabelText("delete"));
  expect(mockDeleteItem).toHaveBeenCalled();
});
```

# Testing the Todo App

- The Todo App has more interesting behaviors – creating new TodoItems
- Next example: how to test that a todo item is created when “Add TODO item” is clicked.



# Testing Todo App's add todo item

**SUT**

```
...
<form onSubmit={onSubmit}>
  <FormControl isRequired>
    <FormLabel>TODO item:</FormLabel>
    <Input placeholder="Put TODO description here" {...register("itemDesc")} />
  </FormControl>
  <Button type="submit">
    Add TODO item
  </Button>
</form>...
```

**Test**

```
beforeEach(() => {
  renderedComponent = render(<TodoApp />);
  newItemTextField = renderedComponent
                          .getByPlaceholderText(
                "Put TODO description here");
  newItemButton = renderedComponent
                      .getByRole("button", {
                                  name: "Add TODO item"
                      }) ;
});
```

# Testing To

```
beforeEach(() => {
  renderedComponent =
  newItemTextField =

  newItemButton = ren
                  .ge
});
it("Adds the specified todo item to the list", () => {
  userEvent.type(newItemTextField, "Write a better test input");
  userEvent.click(newItemButton);
  expect(renderedComponent.getByTestId("todoItem")).toHaveTextContent(
```

Warning: An update to *TodoApp* inside a ***test*** was not wrapped **in** act(...).

When testing, code that causes React state updates should be wrapped into act(...):

```
act(() => {
  /* fire events that update state */
});
/* assert on the output */
```

This ensures that you**'re testing the behavior the user would see in the browser. Learn more at https://reactjs.org/link/wrap-tests-with-act**

*ERROR: TestingLibraryElementError: Unable to find an element by: [data-testid="todoItem"]*

# Await'ing for a condition to be satisfied

```
beforeEach(() => {
  renderedComponent = render(<TodoApp />);
  newItemTextField = renderedComponent
                     .getByPlaceholderText("Put TODO description here");
  newItemButton = renderedComponent
                  .getByRole("button", { name: "Add TODO item" }) ;
});
it("Adds the specified todo item to the list", () => {
  userEvent.type(newItemTextField, "Write a better test input");
  userEvent.click(newItemButton);
  await waitFor( () =>
       expect(renderedComponent.getByTestId("todoItem")).toHaveTextContent(
       "Write a better test input"
       ));
});
```

waitFor will repeatedly execute its callback until it passes, for up to 1 second

# Testing Library Cheat Sheet

| | No Match | 1 Match | 1+ Match | Await? |
|---|---|---|---|---|
| **getBy** | throw | return | throw | No |
| **findBy** | throw | return | throw | Yes |
| **queryBy** | null | return | throw | No |
| **getAllBy** | throw | array | array | No |
| **findAllBy** | throw | array | array | Yes |
| **queryAllBy** | [] | array | array | No |

- Get and query have different behavior when there are different numbers of matches
- Find is *async* and will return a promise to wait for all rendering to complete

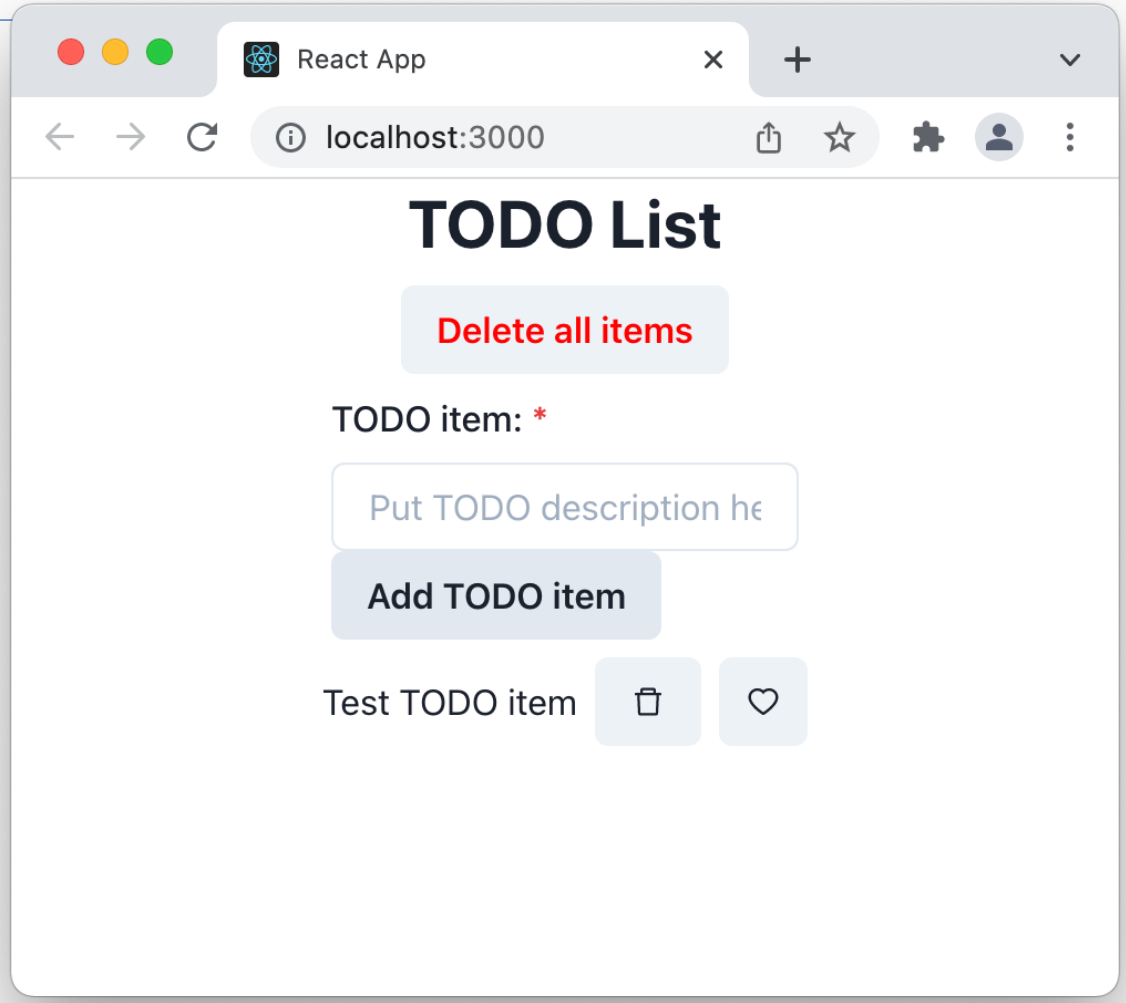
https://testing-library.com/docs/react-testing-library/cheatsheet

# Testing Todo App's add todo item

```
beforeEach(() => {
  renderedComponent = render(<TodoApp />);
  newItemTextField = renderedComponent
                     .getByPlaceholderText("Put TODO description here");
  newItemButton = renderedComponent
                  .getByRole("button", { name: "Add TODO item" }) ;
});
it("Adds the specified todo item to the list", () => {
  userEvent.type(newItemTextField, "Write a better test input");
  userEvent.click(newItemButton);
  expect(renderedComponent.findByTestId("todoItem")).toHaveTextContent(
    "Write a better test input"
  );
});
```

# Activity: Testing React

- Extend the test suite that we discussed in this lesson to also:
  - Test like/unlike on the TodoItem
  - Test the “delete all items” button




Download the activity handout: Linked on course web page for week 7, or at: https://bit.ly/3JV08Lw

Testing Library cheatsheet: https://testing-library.com/docs/react-testing-library/cheatsheet

# Review: Learning Objectives for this Lesson

- you now should be able to:
  - Understand the tradeoff between designing UIs for testability designing tests for UIs
  - Be able to map the three core steps of a test (construct, act, check) to UI component testing
  - Be able to write component-level test for React using Jest