Ken Bima Satria Gandasasmita 23/516183/PA/22062
Rakai Andaru Priandra 23/511442/PA/21796
Muhammad Razan Alamudi 23/511396/PA/21784
Muhammad Naufal Zahir 23/511471/PA/21804
Bambang Abhinawa Pinakasakti 23/511433/PA/21794

Software Testing Reports

Toggle Test

```
import { render, screen, fireEvent } from '@testing-library/react';
import { ThemeToggle } from '@/components/theme-toggle';
import { useTheme } from 'next-themes';

jest.mock('next-themes', () => ({
  useTheme: jest.fn(),
}));

describe('ThemeToggle', () => {
  const setThemeMock = jest.fn();

  beforeEach(() => {
    (useTheme as jest.Mock).mockReturnValue({
      setTheme: setThemeMock,
    });
  });

  afterEach(() => {
    jest.clearAllMocks();
  });

  it('renders toggle button', () => {
    render(<ThemeToggle />);
    expect(screen.getByRole('button', { name: /toggle theme/i
})).toBeInTheDocument();
  });

  it('opens menu and shows options', () => {
    render(<ThemeToggle />);
    fireEvent.click(screen.getByRole('button', { name: /toggle theme/i
}));
    expect(screen.getByText('Light')).toBeVisible();
    expect(screen.getByText('Dark')).toBeVisible();
```

```
    expect(screen.getByText('System')).toBeVisible();
  });

  it.each([
    ['Light', 'light'],
    ['Dark', 'dark'],
    ['System', 'system'],
  ])('sets theme to %s when clicked', (label, theme) => {
    render(<ThemeToggle />);
    fireEvent.click(screen.getByRole('button', { name: /toggle theme/i
}));
    fireEvent.click(screen.getByText(label));
    expect(setThemeMock).toHaveBeenCalledWith(theme);
  });
});
```

This test file verifies the functionality of a `ThemeToggle` component using **React Testing Library** and **Jest**. The component allows users to switch between **Light**, **Dark**, and **System** themes, and it relies on the `useTheme` hook from the `next-themes` library.

Mock Setup

TypeScript

```
jest.mock('next-themes', () => ({
  useTheme: jest.fn(),
}));
```

- Mocks the `useTheme` hook from `next-themes` so it can be controlled in the test environment.

TypeScript

```
const setThemeMock = jest.fn();
beforeEach(() => {
  (useTheme as jest.Mock).mockReturnValue({
    setTheme: setThemeMock,
  });
```

```
});
```

- Before each test, `useTheme` returns a mock object with a `setTheme` function, which you can monitor and assert against.

Test Cases Breakdown

1. Check if the toggle button renders

```TypeScript
it('renders toggle button', () => {
  render(<ThemeToggle />);
  expect(screen.getByRole('button', { name: /toggle theme/i
})).toBeInTheDocument();
});
```

- Verifies that the theme toggle button appears in the document with a label like "Toggle Theme".

2. Check if menu opens and shows options

```TypeScript
it('opens menu and shows options', () => {
  render(<ThemeToggle />);
  fireEvent.click(screen.getByRole('button', { name: /toggle
theme/i }));
  expect(screen.getByText('Light')).toBeVisible();
  expect(screen.getByText('Dark')).toBeVisible();
  expect(screen.getByText('System')).toBeVisible();
});
```

- Simulates a user clicking the toggle button.
- Checks that the three options **Light**, **Dark**, and **System** become visible.

3. Check if clicking an option sets the correct theme

TypeScript

```
it.each([
  ['Light', 'light'],
  ['Dark', 'dark'],
  ['System', 'system'],
])('sets theme to %s when clicked', (label, theme) => {
  render(<ThemeToggle />);
  fireEvent.click(screen.getByRole('button', { name: /toggle
theme/i }));
  fireEvent.click(screen.getByText(label));
  expect(setThemeMock).toHaveBeenCalledWith(theme);
});
```

- Uses **parameterized testing** with `it.each` to run the same test logic for each theme option.
- It clicks the toggle, selects one of the options, and verifies that `setThemeMock` was called with the correct theme string (`light`, `dark`, or `system`).

Button Testing

```
// components/ui/button.test.tsx
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { Button } from "../components/ui/button";

describe("Button component", () => {
  it("renders with default text", () => {
    render(<Button>Click me</Button>);
    expect(screen.getByRole("button", { name: /click me/i
})).toBeInTheDocument();
  });

  it("applies variant class correctly", () => {
    render(<Button variant="destructive">Delete</Button>);
    const btn = screen.getByRole("button", { name: /delete/i });
    expect(btn).toHaveClass("bg-destructive");
```

```
  });

  it("calls onClick handler when clicked", async () => {
    const user = userEvent.setup();
    const handleClick = jest.fn();
    render(<Button onClick={handleClick}>Click</Button>);
    await user.click(screen.getByRole("button", { name: /click/i }));
    expect(handleClick).toHaveBeenCalledTimes(1);
  });
});
```

1. Renders with default text

tsx
CopyEdit

```
it("renders with default text", () => {
  render(<Button>Click me</Button>);
  expect(screen.getByRole("button", { name: /click me/i
})).toBeInTheDocument();
});
```

- Ensure the button renders with the correct label/text.
- Confirms a button is in the document with the text "Click me".
- Verifies **rendering and accessibility** (by using `getByRole('button')` with the accessible name).

2. Applies variant class correctly

tsx
CopyEdit

```
it("applies variant class correctly", () => {
  render(<Button variant="destructive">Delete</Button>);
  const btn = screen.getByRole("button", { name: /delete/i });
  expect(btn).toHaveClass("bg-destructive");
});
```

- Ensure styling changes based on the `variant` prop.
- Verifies that the rendered button with the `destructive` variant has the class `bg-destructive` (assumed to be part of your styling system like Tailwind CSS or custom classnames).
- Confirms **visual styling behavior**.

3. Calls `onClick` when clicked

tsx
CopyEdit

```
it("calls onClick handler when clicked", async () => {
  const user = userEvent.setup();
  const handleClick = jest.fn();
  render(<Button onClick={handleClick}>Click</Button>);
  await user.click(screen.getByRole("button", { name: /click/i }));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

- Ensure clicking the button triggers the `onClick` function.
- Uses `jest.fn()` to mock a click handler and verifies it's called once after simulating a user click.
- Tests **interactivity** and **functionality**.

Input Test

```
import React from 'react';

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

import { Input } from '@/components/ui/input'; // adjust path accordingly



describe('Input component', () => {
```

```
it('renders an input element', () => {
  render(<Input data-testid="input-test" />);
  const input = screen.getByTestId('input-test');
  expect(input).toBeInTheDocument();
  expect(input.tagName).toBe('INPUT');
});

it('applies the given type attribute', () => {
  render(<Input type="email" data-testid="input-type" />);
  const input = screen.getByTestId('input-type');
  expect(input).toHaveAttribute('type', 'email');
});

it('includes the custom className along with default classes', () => {
  const customClass = 'custom-class';
  render(<Input className={customClass} data-testid="input-class" />);
  const input = screen.getByTestId('input-class');
  expect(input).toHaveClass('border-input');
  expect(input).toHaveClass(customClass);
});

it('forwards other props to the input element', () => {
  render(<Input placeholder="Enter text" data-testid="input-props" />);
```

```
    const input = screen.getByTestId('input-props');
    expect(input).toHaveAttribute('placeholder', 'Enter text');
  });

  it('includes data-slot="input" attribute', () => {
    render(<Input data-testid="input-slot" />);
    const input = screen.getByTestId('input-slot');
    expect(input).toHaveAttribute('data-slot', 'input');
  });
});
```

1. Renders an input element

tsx

CopyEdit

```
it('renders an input element', () => {
  render(<Input data-testid="input-test" />);
  const input = screen.getByTestId('input-test');
  expect(input).toBeInTheDocument();
  expect(input.tagName).toBe('INPUT');
});
```

- Ensures the component renders and is of the correct element type (`<input>`).

2. Applies the given `type` attribute

tsx

CopyEdit

```
it('applies the given type attribute', () => {
  render(<Input type="email" data-testid="input-type" />);
  const input = screen.getByTestId('input-type');
  expect(input).toHaveAttribute('type', 'email');
});
```

- Verifies that the `type` prop (e.g., `"email"`) is correctly forwarded to the DOM element.

3. Includes the custom `className` along with default classes

tsx

CopyEdit

```
it('includes the custom className along with default classes', () => {
  const customClass = 'custom-class';
  render(<Input className={customClass} data-testid="input-class" />);
  const input = screen.getByTestId('input-class');
  expect(input).toHaveClass('border-input'); // assumed default class
  expect(input).toHaveClass(customClass);    // custom class
});
```

- Tests if both:

    - A **default class** (like `border-input`) is applied.
    - A **custom `className`** passed as a prop is appended correctly.
- Useful for ensuring styling flexibility.

4. Forwards other props (like `placeholder`)

tsx

CopyEdit

```
it('forwards other props to the input element', () => {
  render(<Input placeholder="Enter text" data-testid="input-props"
/>);
  const input = screen.getByTestId('input-props');
  expect(input).toHaveAttribute('placeholder', 'Enter text');
});
```

- Ensures **spread props** (e.g., `placeholder`, `aria-*`, `disabled`) are forwarded to the native `<input>` element.

5. Includes `data-slot="input"` attribute

tsx

CopyEdit

```
it('includes data-slot="input" attribute', () => {
  render(<Input data-testid="input-slot" />);
  const input = screen.getByTestId('input-slot');
  expect(input).toHaveAttribute('data-slot', 'input');
});
```

Confirms the presence of a structural or styling attribute (`data-slot="input"`), likely used for styling systems like **Radix UI**, **Tailwind Variants**, or **component libraries**.