Apply recognised principles to the building of high-quality software components, analysing the difference between safety and quality, and show how mechanisms that improve quality may be used to partially underpin a safety argument.

I applied principles to React components, such as modularity, ensuring that components have well-defined function and clear responsibility. I also handled API errors using try/catch and managed independent state like loading by react useState hook. I used async/await approaches for asynchronous data handling. I created Jest test, which reduce the probability of software faults. However, I noticed I could refactor it, creating reusable hook and improve data validation to prevents runtime errors. I also plan to add automated testing.  
Analysing the difference between safety and quality, I realised that quality can exist without safety. By focusing on user experience and developing requirements I was able to provide good quality, but it didn’t ensure safety code. It could still be vulnerable to crashes or unexpected behaviours.   
Applying quality principles supports the safety by reducing the risk of systematic failures, increasing confidence in software behaviour, and enabling more robust threat mitigation.  
  
  
**make test:**  
  
import React from 'react';

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

import { SemanticSearchBar } from './SemanticSearchBar';

import '@testing-library/jest-dom/extend-expect';

// Mock the fetch function

global.fetch = jest.fn();

describe('SemanticSearchBar Component', () => {

beforeEach(() => {

fetch.mockClear();

});

test('renders the form inputs and button', () => {

render(<SemanticSearchBar />);

expect(screen.getByLabelText(/Search by Genre/i)).toBeInTheDocument();

expect(screen.getByLabelText(/Search by Plot\/Description/i)).toBeInTheDocument();

expect(screen.getByRole('button', { name: /Search/i })).toBeInTheDocument();

});

test('shows error message when both inputs are empty and form is submitted', async () => {

render(<SemanticSearchBar />);

fireEvent.click(screen.getByRole('button', { name: /Search/i }));

expect(await screen.findByText(/Please provide a genre or query/i)).toBeInTheDocument();

});

test('calls fetch with correct parameters when form is submitted', async () => {

const mockMovies = { movies: ['Movie 1, Movie 2'] };

fetch.mockResolvedValueOnce({

ok: true,

json: async () => mockMovies,

});

render(<SemanticSearchBar />);

fireEvent.change(screen.getByLabelText(/Search by Genre/i), { target: { value: 'Action' } });

fireEvent.change(screen.getByLabelText(/Search by Plot\/Description/i), { target: { value: 'Hero' } });

fireEvent.click(screen.getByRole('button', { name: /Search/i }));

await waitFor(() => expect(fetch).toHaveBeenCalledTimes(1));

expect(fetch).toHaveBeenCalledWith('https://127.0.0.1:443/api/search', expect.objectContaining({

method: 'POST',

credentials: 'include',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ genre: 'Action', search: 'Hero' }),

}));

});

test('displays movie titles after successful fetch', async () => {

const mockMovies = { movies: ['Movie 1, Movie 2'] };

fetch.mockResolvedValueOnce({

ok: true,

json: async () => mockMovies,

});

render(<SemanticSearchBar />);

fireEvent.change(screen.getByLabelText(/Search by Genre/i), { target: { value: 'Action' } });

fireEvent.click(screen.getByRole('button', { name: /Search/i }));

expect(await screen.findByText('Movie 1')).toBeInTheDocument();

expect(screen.getByText('Movie 2')).toBeInTheDocument();

});

test('displays error message on fetch failure', async () => {

fetch.mockResolvedValueOnce({

ok: false,

json: async () => ({ message: 'Fetch failed' }),

});

render(<SemanticSearchBar />);

fireEvent.change(screen.getByLabelText(/Search by Genre/i), { target: { value: 'Action' } });

fireEvent.click(screen.getByRole('button', { name: /Search/i }));

expect(await screen.findByText(/Fetch failed/i)).toBeInTheDocument();

});

});