1. Testing Components
2. Mocking APIs.
3. Testing Forms.
4. Testing State Management
5. Testing Authentication
6. Testing Routing.

Testing Routing using vitest.

Testing routing in a **Next.js** project using **Vitest** involves simulating navigation between pages, verifying that components load properly based on the route, and ensuring that routing logic (such as redirects or route guards) works as expected.

To get started, let's walk through how routing works in **Next.js** and how to simulate and test it using Vitest and testing libraries like **React Testing Library**.

Concepts for Testing Routing:

1. **Next.js Routing**: In Next.js, routing is file-based. For example, pages/index.tsx is the homepage, and pages/about.tsx is the /about route.
2. **React Router Mocks**: When testing, we need to simulate routing behavior since the tests don't run in a real browser.
3. **useRouter Hook**: The useRouter hook provided by Next.js helps you access the router object. You will often need to **mock** this in your tests.

4. **Steps to Test Routing:**
5. **1. Install Required Packages**
6. You need **Vitest**, **@testing-library/react**, and **next-router-mock** (a mock library for Next.js routing) to simulate routing in your tests.
7. Run the following commands:

```
npm install --save-dev vitest @testing-library/react @testing-library/jest-dom next-router-mock
```

2. Mocking Next.js useRouter

Since routing relies on Next.js' useRouter, you'll need to **mock** it in your test environment.

Here’s how you can mock the router using next-router-mock:

```
// Import the necessary testing tools
import { render, screen } from '@testing-library/react';
import { describe, it, expect } from 'vitest';
import { useRouter } from 'next/router'; // Import the Next.js useRouter hook
import nextRouterMock from 'next-router-mock';
import Home from './pages/index'; // Replace with your component path

// Mock the Next.js router globally
vi.mock('next/router', () => require('next-router-mock'));

describe('Home Page', () => {
  it('should render the homepage content based on routing', () => {
    // Use the router mock to set up a route
    nextRouterMock.push('/');

    // Render the component (e.g., Home page)
    render(<Home />);

    // Check if the content is rendered correctly
    expect(screen.getByText(/Welcome to the homepage/i)).toBeInTheDocument();
  });
});
```

3. Test Navigation Between Pages

You can simulate navigation between pages and check if the correct component renders. Let's say you have two pages: Home and About.

Here's how you can write tests to verify routing between them:

Example Test for Routing:

```
import { render, screen } from '@testing-library/react';
import { describe, it, expect } from 'vitest';
import { useRouter } from 'next/router';
import nextRouterMock from 'next-router-mock';
import About from './pages/about'; // Adjust with your about page path

vi.mock('next/router', () => require('next-router-mock'));

describe('About Page', () => {
  it('should navigate to the About page and display correct content', () => {
    // Mock the router to navigate to /about
    nextRouterMock.push('/about');

    // Render the about page
    render(<About />);

    // Check if the about page content is displayed
    expect(screen.getByText(/About Us/i)).toBeInTheDocument();
  });
});
```

4. Testing Route Guards or Redirects

If you have **protected routes** or **redirects**, you can simulate the routing logic and verify it.

For example, assume that if a user is not authenticated, they should be redirected from /dashboard to /login.

Code:

```
import { render } from '@testing-library/react';
import { describe, it, expect } from 'vitest';
import nextRouterMock from 'next-router-mock';
import Dashboard from './pages/dashboard'; // Replace with actual path

vi.mock('next/router', () => require('next-router-mock'));

describe('Dashboard Page - Protected Route', () => {
  it('should redirect to /login if the user is not authenticated', () => {
    // Mock the router to start at /dashboard
    nextRouterMock.push('/dashboard');

    // Mock an unauthenticated user (you may have your own auth logic to mock)
    const isAuthenticated = false;

    // Render the dashboard page, assuming it has redirect logic
    render(<Dashboard isAuthenticated={isAuthenticated} />);

    // Expect the router to push to the login page
    expect(nextRouterMock.pathname).toBe('/login');
  });
});
```

```
import { render } from '@testing-library/react';
import { describe, it, expect } from 'vitest';
import nextRouterMock from 'next-router-mock';
import Dashboard from './pages/dashboard'; // Replace with actual path

vi.mock('next/router', () => require('next-router-mock'));

describe('Dashboard Page - Protected Route', () => {
  it('should redirect to /login if the user is not authenticated', () => {
    // Mock the router to start at /dashboard
    nextRouterMock.push('/dashboard');

    // Mock an unauthenticated user (you may have your own auth logic to mock)
    const isAuthenticated = false;

    // Render the dashboard page, assuming it has redirect logic
    render(<Dashboard isAuthenticated={isAuthenticated} />);

    // Expect the router to push to the login page
    expect(nextRouterMock.pathname).toBe('/login');
  });
});
```

Explanation of the Example Test:

1. **Mocking useRouter**:
    - vi.mock('next/router', ...) ensures that the useRouter hook is mocked throughout your tests. This is essential because the real useRouter interacts with the browser, which is not available in a test environment.
2. **Using nextRouterMock.push()**:
    - This simulates navigating to a specific route (like /about or /dashboard) in your test. It mimics what happens when a user manually navigates to a route in their browser.
3. **Rendering the Component**:
    - After simulating the route with nextRouterMock.push(), the component is rendered with render(<Component />).
4. **Assertions**:
    - We check if the component behaves as expected based on the route. For example, in the dashboard test, if the user isn't authenticated, we expect a redirect to /login.

Final Tips:

- **next-router-mock** is a lightweight and easy-to-use mock for Next.js routing. It helps simulate real-world routing behavior in your tests.
- You can also test more complex routing scenarios, like nested routes, dynamic parameters, and query strings using similar approaches by adjusting the router mock accordingly.

By mastering these concepts, you'll be able to write robust tests for routing logic in your Next.js applications.

Testing State Management in vitest

Testing **state management** in your application using **Vitest** is crucial to ensure that your application behaves as expected when the state changes. State management can involve tools like **React's `useState`**, **useReducer**, or external state management libraries such as **Redux**, **Zustand**, or **Recoil**. Regardless of the tool, the goal is to verify that the state is properly managed and that the application reacts correctly to state changes.

Here's a breakdown of how to test state management in Vitest:

1. Testing React's `useState`

When testing a component that uses `useState`, you're checking if the state is initialized correctly and if the component reacts properly to state changes. Let's start with an example.

Example Component Using `useState`:

```
import React, { useState } from 'react';


function Counter() {
  const [count, setCount] = useState(0);


  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}


export default Counter;
```

Testing useState with Vitest:

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { describe, it, expect } from 'vitest';
import Counter from './Counter'; // Replace with the actual path to the component

describe('Counter component', () => {
  it('should initialize with count 0', () => {
    render(<Counter />);

    // Check if the count is initially 0
    expect(screen.getByText(/Count: 0/i)).toBeInTheDocument();
  });

  it('should increment the count when the button is clicked', async () => {
    render(<Counter />);
    const user = userEvent.setup();

    // Click the button to increment the count
    const button = screen.getByText(/Increment/i);
    await user.click(button);

    // Check if the count has incremented to 1
    expect(screen.getByText(/Count: 1/i)).toBeInTheDocument();
  });
});
```

2. Testing React's useReducer

When using useReducer, you're often managing more complex state. The testing approach is similar to useState, but you're testing if actions correctly update the state based on your reducer logic.

Example Component Using useReducer:

```
import React, { useReducer } from 'react';


const initialState = { count: 0 };


function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}


function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);


  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
    </div>
```

```
  );
}

export default Counter;
```

Testing useReducer with Vitest:

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { describe, it, expect } from 'vitest';
import Counter from './Counter'; // Replace with the actual path to the component

describe('Counter component with useReducer', () => {
  it('should initialize with count 0', () => {
    render(<Counter />);
    expect(screen.getByText(/Count: 0/i)).toBeInTheDocument();
  });

  it('should increment the count when increment button is clicked', async () => {
    render(<Counter />);
    const user = userEvent.setup();
    await user.click(screen.getByText(/Increment/i));

    expect(screen.getByText(/Count: 1/i)).toBeInTheDocument();
  });

  it('should decrement the count when decrement button is clicked', async () => {
```

```
  render(<Counter />);
  const user = userEvent.setup();
  await user.click(screen.getByText(/Decrement/i));

  expect(screen.getByText(/Count: -1/i)).toBeInTheDocument();
 });
});
```

3. Testing with Redux (or Other State Management Libraries)

When testing **Redux** (or libraries like **Zustand** or **Recoil**), you are testing the interaction between the state and components. You mock the Redux store or the global state management, dispatch actions, and ensure the components are updated according to the state.

Example Component Using Redux:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment } from './counterSlice'; // Your Redux slice

function Counter() {
 const count = useSelector((state) => state.counter.value);
 const dispatch = useDispatch();

 return (
  <div>
```

```
      <p>Count: {count}</p>
      <button onClick={() => dispatch(increment())}>Increment</button>
    </div>
  );
}

export default Counter;
```

Testing Redux with Vitest:

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { describe, it, expect } from 'vitest';
import { Provider } from 'react-redux';
import { store } from './store'; // Your Redux store
import Counter from './Counter'; // Replace with the actual path

describe('Counter component with Redux', () => {
  it('should render the initial count from the Redux store', () => {
    render(
      <Provider store={store}>
        <Counter />
      </Provider>
    );

    // Check if the count is initially 0 from Redux
```

```
    expect(screen.getByText(/Count: 0/i)).toBeInTheDocument();
  });

  it('should increment the count when the button is clicked', async () => {
    render(
      <Provider store={store}>
        <Counter />
      </Provider>
    );
    const user = userEvent.setup();

    // Click the increment button
    await user.click(screen.getByText(/Increment/i));

    // Check if the count has incremented
    expect(screen.getByText(/Count: 1/i)).toBeInTheDocument();
  });
});
```

4. Testing Asynchronous State Management

If your state management involves asynchronous actions (e.g., fetching data from an API), you can use **mocking** tools like vi.mock() to mock API calls or actions, and then test how your component handles the state after the asynchronous action.

Example of Testing Async Redux Thunks:

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

```
import { store } from './store'; // Your Redux store
import { fetchUserData } from './userSlice'; // Async action (thunk)
import UserComponent from './UserComponent'; // Replace with actual component

describe('UserComponent with async actions', () => {
  it('should fetch and display user data', async () => {
    // Dispatch the async thunk to fetch user data
    store.dispatch(fetchUserData());

    render(
      <Provider store={store}>
        <UserComponent />
      </Provider>
    );

    // Wait for the user data to appear on the screen
    await waitFor(() => expect(screen.getByText(/User Name/i)).toBeInTheDocument());
  });
});
```

General Principles for Testing State Management:

1. **Render the Component**: Always render the component inside the necessary provider (e.g., Redux Provider, Zustand context) if using external state management.
2. **Interact with the State**: Simulate user actions like clicking buttons or changing inputs to trigger state changes.
3. **Assert State Changes**: Use assertions (expect()) to verify the state's initial values and how they change after actions are dispatched.

4. **Mock Async Actions**: When testing async actions (like API calls), mock the APIs to avoid hitting real endpoints and control the returned data.

By following these steps, you can effectively test state management in your React components using Vitest, ensuring your application state behaves as expected in different scenarios.

Testing Forms in Vitest

Testing forms in Vitest involves checking that the form elements (input fields, buttons, etc.) behave as expected, that the form's submission works properly, and that validation logic functions correctly. This includes simulating user input, triggering events like submitting the form, and verifying that the form handles input and state changes as designed.

Key Aspects to Test in Forms:

1. **Rendering**: Ensure the form renders with the correct initial state (e.g., empty fields).
2. **User Interaction**: Simulate user input in form fields and check if the state updates accordingly.
3. **Form Validation**: Test client-side validation (e.g., required fields, email formats).
4. **Form Submission**: Test that the form submits correctly, handles data properly, and triggers necessary side effects (e.g., API calls).

Let’s walk through how to test forms using **Vitest** with **React Testing Library** and **userEvent** to simulate user interactions.

1. Testing Form Rendering and User Input

Start with a simple form component where the user can input their name and email.

Example Form Component:

```
import { useState } from 'react';


function SimpleForm() {
  const [name, setName] = useState('');
```

```
const [email, setEmail] = useState('');

const handleSubmit = (e) => {
  e.preventDefault();
  alert(`Name: ${name}, Email: ${email}`);
};

return (
  <form onSubmit={handleSubmit}>
    <label htmlFor="name">Name:</label>
    <input
      id="name"
      type="text"
      value={name}
      onChange={(e) => setName(e.target.value)}
    />

    <label htmlFor="email">Email:</label>
    <input
      id="email"
      type="email"
      value={email}
      onChange={(e) => setEmail(e.target.value)}
    />

    <button type="submit">Submit</button>
  </form>
);
```

```
}

export default SimpleForm;
```

Test for Rendering and User Input:

This test checks if the form renders correctly, simulates typing into the input fields, and checks that the state is updated.

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { describe, it, expect } from 'vitest';
import SimpleForm from './SimpleForm'; // Adjust the path to your form component

describe('SimpleForm Component', () => {
  it('should render the form with empty fields', () => {
    render(<SimpleForm />);

    // Check if the name and email fields are initially empty
    expect(screen.getByLabelText(/name/i)).toHaveValue('');
    expect(screen.getByLabelText(/email/i)).toHaveValue('');
  });

  it('should allow the user to type into the input fields', async () => {
    render(<SimpleForm />);
    const user = userEvent.setup();

    // Simulate typing in the name input
    const nameInput = screen.getByLabelText(/name/i);
```

```
    await user.type(nameInput, 'John Doe');
    expect(nameInput).toHaveValue('John Doe');

    // Simulate typing in the email input
    const emailInput = screen.getByLabelText(/email/i);
    await user.type(emailInput, 'john@example.com');
    expect(emailInput).toHaveValue('john@example.com');
  });
});
```

2. Testing Form Validation

If your form includes client-side validation (e.g., required fields, valid email format), you should test the validation logic.

Example Component with Validation:

```
function FormWithValidation() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [errors, setErrors] = useState({ name: '', email: '' });

  const handleSubmit = (e) => {
    e.preventDefault();
    let hasErrors = false;

    if (!name) {
      setErrors((prev) => ({ ...prev, name: 'Name is required' }));
```

```
    hasErrors = true;
  }

  if (!email || !/\S+@\S+\.\S+/.test(email)) {
    setErrors((prev) => ({ ...prev, email: 'Invalid email address' }));
    hasErrors = true;
  }

  if (!hasErrors) {
    alert(`Form submitted: Name - ${name}, Email - ${email}`);
  }
};

return (
  <form onSubmit={handleSubmit}>
    <label htmlFor="name">Name:</label>
    <input
      id="name"
      type="text"
      value={name}
      onChange={(e) => {
        setName(e.target.value);
        setErrors((prev) => ({ ...prev, name: '' }));
      }}
    />
    {errors.name && <p role="alert">{errors.name}</p>}

    <label htmlFor="email">Email:</label>
```

```
      <input
        id="email"
        type="email"
        value={email}
        onChange={(e) => {
          setEmail(e.target.value);
          setErrors((prev) => ({ ...prev, email: '' }));
        }}
      />
      {errors.email && <p role="alert">{errors.email}</p>}

      <button type="submit">Submit</button>
    </form>
  );
}

export default FormWithValidation;
```

Test for Validation:

This test checks whether the validation error messages appear when invalid data is entered or the form is submitted without data.

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { describe, it, expect } from 'vitest';
import FormWithValidation from './FormWithValidation'; // Adjust the path

describe('FormWithValidation Component', () => {
```

```
  it('should show validation errors when fields are empty', async () => {
    render(<FormWithValidation />);
    const user = userEvent.setup();

    // Click the submit button without typing anything
    await user.click(screen.getByText(/submit/i));

    // Expect validation errors to be displayed
    expect(screen.getByRole('alert', { name: /name is required/i })).toBeInTheDocument();
    expect(screen.getByRole('alert', { name: /invalid email address/i })).toBeInTheDocument();
  });

  it('should not show validation errors when form is valid', async () => {
    render(<FormWithValidation />);
    const user = userEvent.setup();

    // Simulate typing valid inputs
    await user.type(screen.getByLabelText(/name/i), 'John Doe');
    await user.type(screen.getByLabelText(/email/i), 'john@example.com');

    // Click the submit button
    await user.click(screen.getByText(/submit/i));

    // Validation errors should not be present
    expect(screen.queryByRole('alert')).toBeNull();
  });
});
```

3. Testing Form Submission

Once validation is complete, you can test that the form submission behaves as expected. For example, you might check if an API is called when the form is submitted.

Example with API Call:

```
function FormWithApi() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');

  const handleSubmit = async (e) => {
    e.preventDefault();
    // Simulate API call
    await fetch('/api/submit', {
      method: 'POST',
      body: JSON.stringify({ name, email }),
    });
  };

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="name">Name:</label>
      <input
        id="name"
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />

      <label htmlFor="email">Email:</label>
```

```
      <input
        id="email"
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />

      <button type="submit">Submit</button>
    </form>
  );
}

export default FormWithApi;
```

Test Form Submission with API Mock:

In this case, you can mock the API call using vi.fn().

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { describe, it, expect, vi } from 'vitest';
import FormWithApi from './FormWithApi';

describe('FormWithApi Component', () => {
  it('should call the API when the form is submitted', async () => {
    const mockFetch = vi.fn(() =>
      Promise.resolve({
        json: () => Promise.resolve({ success: true }),
      })
```

```
    );
    global.fetch = mockFetch;

    render(<FormWithApi />);
    const user = userEvent.setup();

    // Simulate typing and submitting the form
    await user.type(screen.getByLabelText(/name/i), 'John Doe');
    await user.type(screen.getByLabelText(/email/i), 'john@example.com');
    await user.click(screen.getByText(/submit/i));

    // Expect the fetch function to have been called with correct data
    expect(mockFetch).toHaveBeenCalledWith('/api/submit', expect.anything());
  });
});
```

Summary:

- **Render the Form**: Use render() to mount the form component.
- **Simulate User Input**: Use userEvent to simulate typing in fields and submitting the form.
- **Check for Validation**: Test that the form properly validates inputs (e.g., required