테스트

왜 테스트 해야 할까요?

자동화된 테스트는 여러 의미에서 퇴보될 수 있는 코드를 사전에 방지하고, 앱을 테스트 가능한 함수, 모듈, 클래스 및 컴포넌트로 분할하도록 권장하여, 귀하와 귀하의 팀이 복잡한 Vue 앱을 빠르고 자신 있게 빌드하는 데 도움이 됩니다. 다른 앱과 마찬가지로 새 Vue 앱은 여러 가지 방법으로 중단될 수 있으며, 릴리스 전에 이러한 문제를 파악하고 수정할 수 있는 것이 중요합니다.

이 가이드에서는 기본 용어를 다루고 Vue 3 앱에 어떤 도구를 선택해야 하는지에 대한 권장 사항을 제공합니다.

컴포저블을 다루는 Vue 특정 섹션이 있습니다. 자세한 내용은 아래의 컴포저블 테스트를 참조하세요.

테스트 시기

어서 테스트을 시작하세요!! 가능한 한 빨리 테스트 작성을 시작하는 것이 좋습니다. 앱에 테스트 추가를 미룰수록 앱에 더 많은 의존성이 생기고 시작하기가 더 어려워집니다.

테스트 유형

Vue 앱의 테스트 전략을 설계할 때는 다음 테스트 유형을 활용해야 합니다:

단위: 주어진 함수, 클래스 또는 컴포저블에 제공된 입력 정보가 의도하는 출력 또는 사이드 이펙트를 생성하는지 확인합니다.
컴포넌트: 컴포넌트가 마운트, 렌더링, 상호 작용이 의도대로 작동하는지 확인합니다. 이러한 테스트는 단위 테스트보다 더 많은 코드를 가져오고 더 복잡하며 실행하는 데 더 많은 시간이 필요합니다.
End-to-end: 여러 페이지에 걸쳐 있는 기능을 확인하고, 프로덕션으로 빌드되는 Vue 앱처럼 실제 네트워크 요청을 합니다. 이러한 테스트에는 종종 데이터베이스 또는 기타 백엔드를 구축하는 작업이 포함됩니다.

각 테스트 유형은 앱의 테스트 전략에서 역할을 수행하며, 각각 다른 유형의 문제로부터 사용자를 보호해 줍니다.

개요

각 기능에 대해 간략한 설명과 Vue 앱에 이러한 기능을 구현하는 방법 및 몇 가지 일반적인 권장 사항에 대해 알아보겠습니다.

> 용어 설명
>
> 이 가이드 문서에서는 수차례 "모의"(영문에서: **mock**)라는 단어가 나옵니다. 사전적 의미와 유사지만 원활한 맥락 파악을 위해서
>
> > "테스트 대상과 상호작용 하는 환경 또는 구성품들이, 이미 의도대로 구성되어 있다고 가정"
>
> 하는 것이라고 인지하고 있는 것이 좋습니다.

단위 테스트

단위 테스트는 작고 독립된 코드 단위가 예상대로 작동하는지 확인하기 위해 작성됩니다. 단위 테스트는 일반적으로 단일 함수, 클래스, 컴포저블 또는 모듈을 다룹니다. 단위 테스트는 논리적 정확성에 초점을 맞추고 앱의 전체 기능 중 작은 부분에만 관심을 둡니다. 앱 환경의 많은 부분(예: 초기 상태, 복잡한 클래스, 타사 모듈 및 네트워크 요청)을 모의할 수 있습니다.

일반적으로 단위 테스트는 함수가 해야 할 일에 대한 논리 및 논리적 정확성 문제를 포착합니다.

예를 들어 다음 `increment` 함수를 살펴봅시다:

```js
// helpers.js
export function increment (current, max = 10) {
  if (current < max) {
    return current + 1
  }
  return current
}
```

매우 독립적이기 때문에 이 함수를 호출하고 의도하는 결과 값을 반환한다는 검증이 쉬워보이므로, 단위 테스트를 작성해 보겠습니다.

이러한 검증 중 하나라도 실패하면 문제가 `increment` 함수에 있음이 분명합니다.

```js
// helpers.spec.js
import { increment } from './helpers'

describe('increment', () => {
  test('현재 숫자를 1씩 증가', () => {
    expect(increment(0, 10)).toBe(1)
  })

  test('현재 숫자를 최대값 이상으로 증가시키지 않습니다.', () => {
    expect(increment(10, 10)).toBe(10)
  })

  test('기본 최대값은 10입니다.', () => {
    expect(increment(10)).toBe(10)
  })
})
```

앞서 언급했듯이 단위 테스트는 일반적으로 UI 렌더링, 네트워크 요청 또는 기타 환경 문제를 포함하지 않는 자체적으로 해야 할 일에 대한 논리, 컴포넌트, 클래스, 모듈 또는 함수에 적용됩니다.

이들은 일반적으로 Vue와 관련이 없는 일반 JavaScript/TypeScript 모듈입니다. 일반적으로 Vue 앱에서 비즈니스 로직에 대한 단위 테스트를 작성하는 것은 다른 프레임워크를 사용하는 앱과 크게 다르지 않습니다.

Vue 관련 기능을 단위 테스트하는 두 가지 경우가 있습니다.

컴포저블
컴포넌트

컴포저블

Vue 앱에 특별한 함수의 한 범주인 컴포저블은 테스트 중에 특별한 처리가 필요할 수 있습니다. 자세한 내용은 아래의 컴포저블 테스트를 참조하세요.

컴포넌트 단위 테스트

컴포넌트는 두 가지 방법으로 테스트할 수 있습니다:

Whitebox: 단위 테스트

"화이트박스 테스트"는 컴포넌트의 구현 세부 정보 및 의존성을 인식합니다. 테스트 중인 컴포넌트를 **격리**하는 데 중점을 둡니다. 이러한 테스트에는 일반적으로 컴포넌트의 모든 자식은 아닐지라도 일부를 모의하고 플러그인 상태 및 의존성(예: Vuex)을 설정하는 작업이 포함됩니다.

Blackbox: 컴포넌트 테스트

"블랙박스 테스트"는 컴포넌트의 구현 세부 정보를 인식하지 못합니다. 이러한 테스트는 컴포넌트와 전체 시스템의 통합을 테스트하기 위해 가능한 한 적게 모의합니다. 일반적으로 모든 자식 컴포넌트를 렌더링하며 "통합 테스트"에 가깝습니다. 아래의 컴포넌트 테스트 권장 사항을 참조하세요.

추천

Vitest

create-vue 로 생성된 공식 설정은 **Vite**를 기반으로 하므로 동일한 구성을 활용하고 Vite에서 직접 파이프라인을 변환할 수 있는 단위 테스트 프레임워크를 사용하는 것이 좋습니다. **Vitest**는 이러한 목적을 위해 특별히 설계된 단위 테스트 프레임워크로 Vue/Vite 팀이 만들고 유지 관리합니다. 최소한의 노력으로 Vite 기반 프로젝트와 통합되며, 매우 빠릅니다.

다른 선택지

Jest는 인기 있는 단위 테스트 프레임워크입니다. 그러나 Vite 기반 프로젝트로 이전해야 하는 기존의 Jest 테스트 스위트가 있는 경우에만 Jest를 권장합니다. 왜냐하면 Vitest는 더 원활한 통합과 더 나은 성능을 제공하기 때문입니다.

컴포넌트 테스트

Vue 앱에서 컴포넌트는 UI 구성의 주요 요소입니다. 따라서 컴포넌트는 앱의 동작을 검증할 때 자연스럽게 독립된 단위입니다. 컴포넌트 테스트는 단위 테스트의 상위 개념이며, 통합 테스트의 한 형태로 간주될 수 있습니다. Vue 앱의 대부분은 컴포넌트 테스트로 다루어야 하며, 각 Vue 컴포넌트에는 자체 스팩 파일이 있는 것이 좋습니다.

컴포넌트 테스트는 컴포넌트의 props, 이벤트, 제공하는 슬롯, 스타일, 클래스, 생명 주기 훅 등과 관련된 문제를 포착해야 합니다.

컴포넌트 테스트는 자식 컴포넌트를 모의해서는 안 되며, 사용자가 하는 것처럼 컴포넌트와 상호 작용하여 컴포넌트와 해당 자식 간의 상호 작용을 테스트해야 합니다. 예를 들어, 컴포넌트 테스트는 프로그래밍 방식으로 컴포넌트와 상호 작용하는 대신 사용자가 엘리먼트를 클릭하는 것과 같아야 합니다.

컴포넌트 테스트는 내부 구현 세부 사항보다는 컴포넌트의 공개 인터페이스에 중점을 두어야 합니다. 대부분의 컴포넌트에서 공개 인터페이스는 이벤트 발생, props 및 슬롯정도 입니다. "**컴포넌트가 어떻게 작동하는지가 아니라, 어떤 작동을 하는지**"를 테스트해야 합니다.

해야 할 것

시각적 로직의 경우: 입력된 props 및 슬롯을 기반으로 올바른 렌더링 출력을 검증합니다.

작동 로직의 경우: 사용자 입력 이벤트에 대한 응답으로 올바른 렌더링 업데이트 또는 발송(emit)된 이벤트를 검증합니다.

아래 예에서는 "increment"라는 레이블이 지정된 DOM 엘리먼트가 있고, 클릭할 수 있는 스테퍼 컴포넌트를 보여줍니다. 스테퍼가 2 를 초과하여 계속 증가하지 않도록 하는 max 라는 prop을 전달하므로, 버튼을 3번 클릭해도 UI는 여전히 2 를 표시해야 합니다.

우리는 스테퍼의 구현에 대해 아는 것이 없고, 단지 "입력"이 max 라는 prop이고, "출력"이 사용자가 보게 될 DOM의 상태라는 것 뿐입니다.

Vue Test Utils Cypress Testing Library

js

```
const valueSelector = '[data-testid=stepper-value]'
const buttonSelector = '[data-testid=increment]'

const wrapper = mount(Stepper, {
  props: {
    max: 1
  }
})

expect(wrapper.find(valueSelector).text()).toContain('0')

await wrapper.find(buttonSelector).trigger('click')
```

```
expect(wrapper.find(valueSelector).text()).toContain('1')
```

하지 말아야 할 것

컴포넌트 인스턴스의 개인 상태를 검증하거나 컴포넌트의 개인 메서드를 테스트하지 마십시오. 구현 세부 정보를 테스트하면 테스트가 중단될 가능성이 높고 구현이 변경될 때 업데이트가 필요하기 때문에 테스트가 취약해집니다.

컴포넌트의 궁극적인 역할은 올바른 DOM 출력을 렌더링하는 것이므로, DOM 출력에 초점을 맞춘 테스트는 변경사항에도 강력하고 탄력적이면서 동일한 수준의 정확성 보장을 제공합니다.

스냅샷 테스트에만 의존하지 마십시오. HTML 문자열을 검증하는 것이 정확성을 의미하지 않으므로 명확한 목적이 있는 테스트를 작성하십시오.

메서드를 철저히 테스트해야 하는 경우, 독립 실행형 유틸리티 기능으로 추출하는 것을 고려하고, 전용 단위 테스트를 작성하십시오. 깔끔하게 추출할 수 없는 경우, 컴포넌트나 통합 또는 이를 포괄하는 E2E 테스트의 일부로 테스트할 수 있습니다.

추천

Vitest는 헤드리스로 렌더링되는 컴포넌트나 컴포저블을 위해 사용됩니다(예: VueUse의 **useFavicon** 함수). 컴포넌트와 DOM은 **@vue/test-utils** 를 사용하여 테스트할 수 있습니다.

Cypress 컴포넌트 테스팅은 스타일의 적절한 렌더링이나 네이티브 DOM 이벤트의 트리거링에 따라 예상되는 동작을 하는 컴포넌트를 위해 사용됩니다. 이는 **@testing-library/cypress**를 통해 Testing Library와 함께 사용될 수 있습니다.

Vitest와 브라우저 기반 러너의 주요 차이점은 속도와 실행 컨텍스트입니다. 간단히 말해서, Cypress와 같은 브라우저 기반 러너는 Vitest와 같은 노드 기반 러너가 포착할 수 없는 문제(예: 스타일 문제, 실제 네이티브 DOM 이벤트, 쿠키, 로컬 스토리지 및 네트워크 에러)를 포착할 수 있지만, 브라우저 기반 러너는 브라우저를 열고 스타일시트를 컴파일하는 등의 작업을 수행하기 때문에 *Vitest보다 훨씬 느립니다*. Cypress는 컴포넌트 테스트를 지원하는 브라우저 기반 러너입니다. Vitest와 Cypress를 비교한 최신 정보는 비교 페이지를 참조하십시오.

마운팅 라이브러리

컴포넌트 테스트에는 테스트 중인 컴포넌트를 격리하여 탑재하고, 시뮬레이션된 사용자 입력 이벤트를 트리거하고, 렌더링된 DOM 출력에 대한 검증이 포함되는 경우가 많습니다. 이러한 작업을 더 간단하게 만드는 전용 유틸리티 라이브러리가 있습니다.

@vue/test-utils 는 사용자에게 Vue 특정 API에 대한 접근을 제공하기 위해 작성된 공식 저수준 컴포넌트 테스트 라이브러리입니다. 또한 저수준 라이브러리 @testing-library/vue 가 그 위에 구축됩니다.

@testing-library/vue 는 구현 세부 사항에 의존하지 않고 구성 요소 테스트에 중점을 둔 Vue 테스트 라이브러리입니다. 기본 원칙은 테스트가 소프트웨어 사용 방식과 유사할수록 더 많은 신뢰를 제공할 수 있다는 것입니다.

앱의 컴포넌트를 테스트하려면 @vue/test-utils 를 사용하는 것이 좋습니다. @testing-library/vue 는 Suspense로 비동기 컴포넌트를 테스트하는 데 문제가 있으므로 주의해서 사용해야 합니다.

다른 선택지

Nightwatch는 Vue 컴포넌트 테스트를 지원하는 E2E 테스트 러너입니다. (예제 프로젝트)

WebdriverIO는 표준화된 자동화를 기반으로 하는 네이티브 사용자 상호작용에 의존하는 크로스 브라우저 컴포넌트 테스팅을 위해 사용됩니다. 이는 Testing Library와 함께 사용될 수도 있습니다.

E2E 테스트

단위 테스트는 개발자에게 어느 정도의 신뢰를 제공하지만, 단위 및 컴포넌트 테스트는 프로덕션에 배포된 애플리케이션의 전체적인 커버리지를 제공하는 데에 한계가 있습니다. 따라서, 엔드투엔드(E2E) 테스트는 애플리케이션의 가장 중요한 측면, 즉 사용자가 실제로 애플리케이션을 사용할 때 발생하는 일에 대한 커버리지를 제공합니다.

엔드투엔드 테스트는 프로덕션 빌드된 Vue 애플리케이션에 대한 네트워크 요청을 수행하는 멀티 페이지 애플리케이션의 동작에 중점을 둡니다. 이들은 종종 데이터베이스나 기타 백엔드를 구축하는 것을 포함하며, 라이브 스테이징 환경에서 실행될 수도 있습니다.

엔드투엔드 테스트는 라우터, 상태 관리 라이브러리, 최상위 컴포넌트(예: App 또는 Layout), 공개 자산 또는 요청 처리와 같은 문제를 종종 포착합니다. 위에서 언급했듯이, 이들은 단위 테스트나 컴포넌트 테스트로는 발견하기 어려운 중요한 문제들을 포착합니다.

엔드투엔드 테스트는 Vue 애플리케이션 코드를 가져오지 않고, 실제 브라우저에서 전체 페이지를 통해 애플리케이션을 탐색하여 테스트하는 데 완전히 의존합니다.

엔드투엔드 테스트는 애플리케이션의 여러 레이어를 검증합니다. 이들은 로컬에서 빌드된 애플리케이션을 대상으로 하거나 심지어 라이브 스테이징 환경을 대상으로 할 수 있습니다. 스테이징 환경에서의 테스트는 프론트엔드 코드와 정적 서버뿐만 아니라 모든 관련 백엔드 서비스 및 인프라를 포함합니다.

> 소프트웨어가 사용되는 방식과 테스트가 유사할수록, 테스트는 더 많은 신뢰를 줄 수 있습니다. - **Kent C. Dodds** - Testing Library 저자

사용자 작업이 앱에 미치는 영향을 테스트함으로써, E2E 테스트는 종종 앱이 제대로 작동하는지 여부에 대한 신뢰도를 높입니다.

E2E 테스트 솔루션 선택

웹에서의 E2E 테스트는 신뢰할 수 없는 테스트 및 개발 프로세스 속도 저하로 부정적인 평판을 얻었지만, 최신 E2E 도구는 보다 안정적이고 대화식이며 유용한 테스트를 만들기 위해 발전했습니다. 다음 섹션에서는 E2E 테스트 프레임워크를 선택할 때, 앱에 대한 테스트 프레임워크를 선택할 때 염두에 두어야 할 사항에 대한 몇 가지 지침을 제공합니다.

크로스 브라우저 테스팅

엔드투엔드(E2E) 테스팅의 주요 장점 중 하나는 다양한 브라우저에서 애플리케이션을 테스트할 수 있는 능력입니다. 100% 크로스 브라우저 커버리지를 가지는 것이 바람직해 보일 수 있지만, 일관되게 실행하기 위해 필요한 추가 시간과 기계 파워로 인해 팀 자원에 대한 효용이 감소한다는 점을 유의해야 합니다. 따라서 애플리케이션이 필요로 하는 크로스 브라우저 테스팅의 양을 선택할 때 이러한 트레이드오프를 고려하는 것이 중요합니다.

더 빠른 피드백 루프

엔드투엔드(E2E) 테스트와 개발의 주요 문제 중 하나는 전체 테스트 스위트를 실행하는 데 오랜 시간이 걸린다는 것입니다. 일반적으로 이는 지속적 통합 및 배포(CI/CD) 파이프라인에서만 수행됩니다. 현대의 E2E 테스트 프레임워크는 병렬 처리와 같은 기능을 추가하여 CI/CD 파이프라인이 이전보다 훨씬 빨리 실행될 수 있도록 도와줍니다. 또한, 로컬에서 개발할 때, 작업 중인 페이지에 대해 단일 테스트를 선택적으로 실행하는 능력과 테스트의 핫 리로딩을 제공함으로써 개발자의 워크플로우와 생산성을 향상시킬 수 있습니다.

일급 디버깅 경험

개발자들은 전통적으로 테스트에서 무엇이 잘못되었는지 파악하기 위해 터미널 창의 로그를 스캔하는 데 의존해왔습니다. 그러나 현대의 엔드투엔드(E2E) 테스트 프레임워크는 개발자가 이미 익숙한 도구, 예를 들어 브라우저 개발자 도구를 활용할 수 있도록 합니다.

헤드리스 모드에서의 가시성

엔드투엔드(E2E) 테스트가 지속적 통합/배포 파이프라인에서 실행될 때, 종종 헤드리스 브라우저(즉, 사용자가 볼 수 있는 브라우저가 열리지 않음)에서 실행됩니다. 현대 E2E 테스트 프레임워크의 중요한 기능은 테스트 중 애플리케이션의 스냅샷 및/또는 비디오를 볼 수 있는 능력으로, 오류가 발생하는 이유에 대한 일부 통찰력을 제공합니다. 이러한 통합을 유지하는 것은 역사적으로 번거로웠습니다.

추천

Cypress

전반적으로, Cypress는 정보 제공이 뛰어난 그래픽 인터페이스, 우수한 디버그 기능, 내장된 어설션, 스텁, 플레이크 저항성, 병렬 처리, 스냅샷과 같은 기능을 갖춘 가장 완벽한 E2E 솔루션을 제공한다고 믿습니다. 위에서 언급한 것처럼, 컴포넌트 테스트에 대한 지원도 제공합니다. Chromium 기반 브라우저, Firefox, Electron을 지원합니다. WebKit 지원도 가능하지만 실험적 상태로 표시되어 있습니다.

다른 선택지

Playwright 역시 모든 최신 렌더링 엔진을 지원하는 훌륭한 E2E 테스트 솔루션입니다. Chromium, WebKit, Firefox를 포함한 모든 최신 렌더링 엔진을 지원합니다. Windows, Linux, macOS에서 로컬 또는 CI에서, 헤드리스 또는 헤드 모드에서, Android용 Google Chrome과 모바일 Safari의 네이티브 모바일 에뮬레이션으로 테스트할 수 있습니다.

Nightwatch는 **Selenium WebDriver**를 기반으로 한 E2E 테스트 솔루션입니다. 이는 네이티브 모바일 테스트를 포함한 가장 넓은 브라우저 지원 범위를 제공합니다. Selenium 기반 솔루션은 Playwright나 Cypress보다 느립니다.

WebdriverIO는 WebDriver 프로토콜을 기반으로 한 웹 및 모바일 테스트 자동화 프레임워크입니다.

레시피

프로젝트에 Vitest 추가

Vite 기반 Vue 프로젝트에서 다음을 실행:

```sh
> npm install -D vitest happy-dom @testing-library/vue
```

다음으로 `test` 옵션 블록을 추가하도록 Vite 구성을 업데이트:

```js
// vite.config.js
import { defineConfig } from 'vite'

export default defineConfig({
  // ...
  test: {
    // jest와 같은 전역 테스트 API 사용
    globals: true,
    // happy-dom으로 DOM 시뮬레이션
    // (피어 의존성으로 happy-dom을 설치해야 함)
    environment: 'happy-dom'
  }
})
```

> **TIP**
>
> TypeScript를 사용하는 경우, `tsconfig.json` 파일의 `types` 필드에 `vitest/globals` 를 추가하십시오.
>
> ```json
> // tsconfig.json
>
> {
>   "compileroptions": {
>     "types": ["vitest/globals"]
>   }
> }
> ```

그 다음, 프로젝트 내에 `*.test.js` 로 끝나는 파일을 생성합니다. 모든 테스트 파일을 프로젝트 루트의 테스트 디렉토리에 두거나 소스 파일 옆의 테스트 디렉토리에 배치할 수 있습니다. Vitest는 명명 규칙을 사용하여 자동으로 이들을 검색할 것입니다.

```js
// MyComponent.test.js
import { render } from '@testing-library/vue'
import MyComponent from './MyComponent.vue'

test('it should work', () => {
  const { getByText } = render(MyComponent, {
    props: {
      /* ... */
    }
  })

  // 검증 통과를 위한 출력 값
  getByText('...')
})
```

마지막으로 `package.json` 을 업데이트하여 테스트 스크립트를 추가하고 실행:

```json
{
  // ...
  "scripts": {
    "test": "vitest"
  }
}
```

```sh
> npm test
```

컴포저블 테스트

이 섹션은 컴포저블 섹션을 읽었다고 가정합니다.

컴포저블을 테스트할 때, 호스트 컴포넌트 인스턴스에 의존하는 경우와 그렇지 않은 경우 두 가지 범주로 나눌 수 있습니다.

컴포저블은 아래 API를 사용할 때 호스트 컴포넌트 인스턴스에 따라 달라집니다.

생명 주기 훅
Provide(제공) / Inject(주입)

컴포저블이 반응형 API만 사용하는 경우 직접 호출하고, 반환된 상태/메서드를 검증하여 테스트할 수 있습니다:

```js
// counter.js
import { ref } from 'vue'

export function useCounter() {
  const count = ref(0)
  const increment = () => count.value++

  return {
    count,
    increment
  }
}
```

```js
// counter.test.js
import { useCounter } from './counter.js'

test('useCounter', () => {
  const { count, increment } = useCounter()
  expect(count.value).toBe(0)

  increment()
  expect(count.value).toBe(1)
})
```

생명 주기 훅 또는 Provide/Inject에 의존하는 컴포저블은 테스트할 호스트 컴포넌트에 래핑되어야 합니다. 다음과 같은 헬퍼를 만들 수 있습니다:

```js
// test-utils.js
import { createApp } from 'vue'

export function withSetup(composable) {
  let result
  const app = createApp({
    setup() {
      result = composable()
      // 누락된 템플릿 경고 억제
      return () => {}
    }
  })
```

```
  app.mount(document.createElement('div'))
  // 결과를 반환하고
  // provide/unmount 테스트를 위한 앱 인스턴스
  return [result, app]
}
```

```
import { withSetup } from './test-utils'
import { useFoo } from './foo'

test('useFoo', () => {
  const [result, app] = withSetup(() => useFoo(123))
  // 테스트 inject(주입)를 위한 모의 제공
  app.provide(...)
  // 검증 실행
  expect(result.foo.value).toBe(1)
  // 필요한 경우 onUnmounted 훅 실행
  app.unmount()
})
```

js

더 복잡한 컴포저블의 경우, 컴포넌트 테스트 기술을 사용하여 래퍼 컴포넌트에 대한 테스트를 작성하여 테스트하는 것이 더 쉬울 수도 있습니다.