테스트

왜 테스트를 해야 하나요?

자동화된 테스트는 회귀를 방지하고 애플리케이션을 테스트 가능한 함수, 모듈, 클래스, 컴포넌트로 분리하도록 유도함으로써, 여러분과 팀이 복잡한 Vue 애플리케이션을 빠르고 자신 있게 구축할 수 있도록 도와줍니다. 모든 애플리케이션과 마찬가지로, 새 Vue 앱도 다양한 방식으로 문제가 발생할 수 있으며, 이러한 문제를 출시 전에 발견하고 수정할 수 있는 것이 중요합니다.

이 가이드에서는 기본 용어를 다루고, Vue 3 애플리케이션에 사용할 도구에 대한 권장 사항을 제공합니다.

Vue에 특화된 섹션으로 컴포저블에 대한 내용이 있습니다. 자세한 내용은 아래 컴포저블 테스트하기를 참고하세요.

언제 테스트해야 하나요?

테스트는 일찍 시작하세요! 가능한 한 빨리 테스트를 작성하기 시작하는 것을 권장합니다. 애플리케이션에 테스트를 추가하는 것을 미루면 미룰수록, 애플리케이션의 의존성이 많아지고 시작하기가 더 어려워집니다.

테스트 유형

Vue 애플리케이션의 테스트 전략을 설계할 때, 다음과 같은 테스트 유형을 활용해야 합니다:

- **단위(Unit)**: 주어진 함수, 클래스, 또는 컴포저블에 대한 입력이 예상한 출력이나 부수 효과를 내는지 확인합니다.

대로 동작하는지 확인합니다. 이 테스트는 단위 테스트보다 더 많은 코드를 가져오고, 더 복잡하며, 실행 시간이 더 오래 걸립니다.

엔드 투 엔드(End-to-end): 여러 페이지에 걸친 기능을 확인하고, 프로덕션 빌드된 Vue 애플리케이션에 대해 실제 네트워크 요청을 수행합니다. 이 테스트는 종종 데이터베이스나 기타 백엔드를 준비해야 합니다.

각 테스트 유형은 애플리케이션의 테스트 전략에서 역할을 하며, 각각 다른 유형의 문제로부터 여러분을 보호합니다.

개요

각 테스트 유형이 무엇인지, Vue 애플리케이션에서 어떻게 구현할 수 있는지 간단히 설명하고, 일반적인 권장 사항을 제공합니다.

단위 테스트

단위 테스트는 작고 독립적인 코드 단위가 예상대로 동작하는지 확인하기 위해 작성됩니다. 단위 테스트는 보통 하나의 함수, 클래스, 컴포저블, 또는 모듈을 다룹니다. 단위 테스트는 논리적 정확성에 집중하며, 애플리케이션 전체 기능 중 일부만을 다룹니다. 애플리케이션 환경의 많은 부분(예: 초기 상태, 복잡한 클래스, 서드파티 모듈, 네트워크 요청 등)을 모킹할 수 있습니다.

일반적으로, 단위 테스트는 함수의 비즈니스 로직과 논리적 정확성에 관한 문제를 잡아냅니다.

예를 들어, 다음과 같은 `increment` 함수가 있습니다:

helpers.js

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

이 함수는 매우 독립적이기 때문에, increment 함수를 호출하고 반환값이 예상대로 나오는지 쉽게 검증할 수 있습니다. 그래서 단위 테스트를 작성합니다.

이러한 검증 중 하나라도 실패한다면, 문제는 `increment` 함수 내부에 있다는 것이 명확합니다.

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

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

  test('최대값을 넘어서 현재 숫자를 증가시키지 않는다', () => {
    expect(increment(10, 10)).toBe(10)
  })

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

앞서 언급했듯이, 단위 테스트는 일반적으로 UI 렌더링, 네트워크 요청, 기타 환경적 요소와 관련 없는 독립적인 비즈니스 로직, 컴포넌트, 클래스, 모듈, 함수에 적용됩니다.

이들은 일반적으로 Vue와 관련 없는 순수 JavaScript / TypeScript 모듈입니다. Vue 애플리케이션에서 비즈니스 로직에 대한 단위 테스트를 작성하는 것은 다른 프레임워크를 사용하는 애플리케이션과 크게 다르지 않습니다.

Vue에 특화된 기능을 단위 테스트해야 하는 경우는 두 가지가 있습니다:

1. 컴포저블
2. 컴포넌트

컴포저블

Vue 애플리케이션에 특화된 함수의 한 범주는 컴포저블로, 테스트 시 특별한 처리가 필요할 수 있습니다. 자세한 내용은 아래 컴포저블 테스트하기를 참고하세요.

컴포넌트 단위 테스트

컴포넌트는 두 가지 방식으로 테스트할 수 있습니다:

1. 화이트박스: 단위 테스트

   "화이트박스 테스트"는 컴포넌트의 구현 세부사항과 의존성을 인지합니다. 이 테스트는 **테스트 대상 컴포넌트의 격리**에 집중합니다. 보통 컴포넌트의 자식 중 일부 또는 전부를 모킹하고, 플러그인 상태와 의존성(예: Pinia)을 설정합니다.

2. 블랙박스: 컴포넌트 테스트

전체 시스템의 통합을 테스트하기 위해 가능한 한 적게 모킹합니다. 보통 모든 자식 컴포넌트를 렌더링하며, "통합 테스트"에 더 가깝다고 볼 수 있습니다. 아래 컴포넌트 테스트 권장 사항을 참고하세요.

권장 사항

Vitest

공식적으로 `create-vue` 로 생성된 프로젝트는 Vite를 기반으로 하므로, 동일한 설정과 변환 파이프라인을 직접 활용할 수 있는 단위 테스트 프레임워크를 사용하는 것이 좋습니다. Vitest는 바로 이 목적을 위해 Vue / Vite 팀 멤버들이 만든 단위 테스트 프레임워크입니다. Vite 기반 프로젝트에 최소한의 노력으로 통합할 수 있으며, 매우 빠릅니다.

기타 옵션

Jest는 인기 있는 단위 테스트 프레임워크입니다. 하지만 기존 Jest 테스트 스위트를 Vite 기반 프로젝트로 마이그레이션해야 하는 경우에만 Jest를 추천하며, Vitest가 더 원활한 통합과 더 나은 성능을 제공합니다.

컴포넌트 테스트

Vue 애플리케이션에서 컴포넌트는 UI의 주요 빌딩 블록입니다. 따라서 애플리케이션의 동작을 검증할 때 컴포넌트는 자연스러운 격리 단위가 됩니다. 세분화 관점에서 컴포넌트 테스트는 단위 테스트보다 상위에 위치하며, 일종의 통합 테스트로 볼 수 있습니다. Vue 애플리케이션의 많은 부분이 컴포넌트 테스트로 커버되어야 하며, 각 Vue 컴포넌트마다 자체적인 spec 파일을 두는 것을 권장합니다.

컴포넌트 테스트는 컴포넌트의 props, 이벤트, 제공하는 슬롯, 스타일, 클래스, 라이프사이클 훅 등과 관련된 문제를 잡아내야 합니다.

컴포넌트 테스트는 자식 컴포넌트를 모킹하지 않고, 사용자가 컴포넌트와 상호작용하듯이 컴포넌트와 자식 간의 상호작용을 테스트해야 합니다. 예를 들어, 컴포넌트 테스트는 사용자가 클릭하듯이 요소를 클릭해야 하며, 프로그램적으로 컴포넌트와 상호작용해서는 안 됩니다.

컴포넌트 테스트는 내부 구현 세부사항보다는 컴포넌트의 공개 인터페이스에 집중해야 합니다. 대부분의 컴포넌트에서 공개 인터페이스는 이벤트 발생, props, 슬롯에 한정됩니다. 테스트할 때는 **컴포넌트가 어떻게 동작하는지가 아니라, 무엇을 하는지 테스트**해야 함을 기억하세요.

해야 할 것

행동(Behavioral) 로직: 사용자 입력 이벤트에 대한 올바른 렌더링 업데이트나 이벤트 발생을 검증합니다.

아래 예제에서는 "increment"라는 DOM 요소가 있고 클릭할 수 있는 Stepper 컴포넌트를 보여줍니다. `max` 라는 prop을 전달하여 Stepper가 `2` 를 넘어서 증가하지 못하게 하므로, 버튼을 3번 클릭해도 UI에는 여전히 `2` 가 표시되어야 합니다.

Stepper의 구현에 대해 아무것도 알지 못하며, "입력"은 `max` prop이고 "출력"은 사용자가 보게 될 DOM의 상태임을 알 수 있습니다.

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 비공개 상태를 검증하거나, 컴포넌트의 비공개 메서드를 테스트하지 마세요. 구현 세부사항을 테스트하면 테스트가 취약해지며, 구현이 변경될 때 더 자주 깨지고 수정이 필요해집니다.

컴포넌트의 궁극적인 역할은 올바른 DOM 출력을 렌더링하는 것이므로, DOM 출력에 집중한 테스트가 동일한 수준(혹은 그 이상)의 정확성 보장을 제공하면서도 더 견고하고 변화에 강합니다.

스냅샷 테스트에만 의존하지 마세요. HTML 문자열을 검증하는 것은 정확성을 설명하지 않습니다. 의도를 가지고 테스트를 작성하세요.

어떤 메서드를 철저히 테스트해야 한다면, 별도의 유틸리티 함수로 추출하여 전용 단위 테스트를 작성하는 것을 고려하세요. 깔끔하게 추출할 수 없다면, 해당 메서드를 포함하는 컴포넌트, 통합, 또는 엔드 투 엔드 테스트의 일부로 테스트할 수 있습니다.

권장 사항

적합합니다. 컴포넌트와 DOM은 `@vue/test-utils` 를 사용해 테스트할 수 있습니다.

Cypress Component Testing은 스타일이 제대로 렌더링되거나 네이티브 DOM 이벤트 트리거에 따라 기대 동작이 달라지는 컴포넌트에 적합합니다. @testing-library/cypress를 통해 Testing Library와 함께 사용할 수 있습니다.

Vitest와 브라우저 기반 러너의 주요 차이점은 속도와 실행 컨텍스트입니다. 간단히 말해, Cypress와 같은 브라우저 기반 러너는 Vitest와 같은 Node 기반 러너가 잡지 못하는 문제(예: 스타일 문제, 실제 네이티브 DOM 이벤트, 쿠키, 로컬 스토리지, 네트워크 실패 등)를 잡을 수 있지만, 브라우저를 열고 스타일시트를 컴파일하는 등으로 인해 *Vitest*보다 훨씬 느립니다. Cypress는 컴포넌트 테스트를 지원하는 브라우저 기반 러너입니다. Vitest와 Cypress의 최신 비교 정보는 Vitest의 비교 페이지를 참고하세요.

마운트 라이브러리

컴포넌트 테스트는 보통 테스트 대상 컴포넌트를 독립적으로 마운트하고, 사용자 입력 이벤트를 시뮬레이션하며, 렌더링된 DOM 출력을 검증하는 과정을 포함합니다. 이러한 작업을 더 쉽게 해주는 전용 유틸리티 라이브러리가 있습니다.

`@vue/test-utils` 는 Vue 전용 API에 접근할 수 있도록 작성된 공식 저수준 컴포넌트 테스트 라이브러리입니다. `@testing-library/vue` 도 이 라이브러리 위에 구축되어 있습니다.

`@testing-library/vue` 는 구현 세부사항에 의존하지 않고 컴포넌트를 테스트하는 데 중점을 둔 Vue 테스트 라이브러리입니다. "테스트가 소프트웨어 사용 방식과 비슷할수록 더 많은 신뢰를 줄 수 있다"는 원칙을 따릅니다.

애플리케이션에서 컴포넌트 테스트에는 `@vue/test-utils` 사용을 권장합니다. `@testing-library/vue` 는 Suspense가 있는 비동기 컴포넌트 테스트에 문제가 있으므로 주의해서 사용해야 합니다.

기타 옵션

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

WebdriverIO는 표준화된 자동화 기반 네이티브 사용자 상호작용을 활용한 크로스 브라우저 컴포넌트 테스트에 적합합니다. Testing Library와 함께 사용할 수도 있습니다.

E2E 테스트

는 프로덕션에 배포된 애플리케이션의 전체적인 커버리지를 제공하는 데 한계가 있습니다. 그 결과, 엔드 투 엔드(E2E) 테스트는 애플리케이션에서 아마도 가장 중요한 측면, 즉 실제 사용자가 애플리케이션을 사용할 때 발생하는 일을 커버합니다.

엔드 투 엔드 테스트는 프로덕션 빌드된 Vue 애플리케이션에 대해 네트워크 요청을 수행하는 다중 페이지 애플리케이션 동작에 집중합니다. 종종 데이터베이스나 기타 백엔드를 준비해야 하며, 실제 스테이징 환경에서 실행될 수도 있습니다.

엔드 투 엔드 테스트는 라우터, 상태 관리 라이브러리, 최상위 컴포넌트(예: App 또는 Layout), 공개 자산, 요청 처리 등과 관련된 문제를 자주 잡아냅니다. 앞서 언급했듯이, 단위 테스트나 컴포넌트 테스트로는 잡기 힘든 중요한 문제를 잡아냅니다.

엔드 투 엔드 테스트는 Vue 애플리케이션의 코드를 가져오지 않고, 실제 브라우저에서 전체 페이지를 탐색하여 애플리케이션을 테스트하는 데 전적으로 의존합니다.

엔드 투 엔드 테스트는 애플리케이션의 여러 계층을 검증합니다. 로컬로 빌드된 애플리케이션이나 실제 스테이징 환경을 대상으로 할 수 있습니다. 스테이징 환경을 대상으로 테스트하면 프론트엔드 코드와 정적 서버뿐만 아니라 모든 백엔드 서비스와 인프라까지 포함됩니다.

> 테스트가 소프트웨어 사용 방식과 비슷할수록 더 많은 신뢰를 줄 수 있습니다. - **Kent C. Dodds** - Testing Library 저자

사용자 행동이 애플리케이션에 미치는 영향을 테스트함으로써, E2E 테스트는 애플리케이션이 제대로 동작하는지에 대한 신뢰도를 높이는 핵심이 되는 경우가 많습니다.

E2E 테스트 솔루션 선택하기

웹에서의 엔드 투 엔드(E2E) 테스트는 신뢰할 수 없는(불안정한) 테스트와 개발 프로세스 지연으로 인해 부정적인 평판을 얻었지만, 최신 E2E 도구는 더 신뢰할 수 있고, 상호작용적이며, 유용한 테스트를 만들기 위해 발전해 왔습니다. E2E 테스트 프레임워크를 선택할 때, 다음 섹션에서는 애플리케이션에 적합한 테스트 프레임워크를 선택할 때 고려해야 할 사항을 안내합니다.

크로스 브라우저 테스트

엔드 투 엔드(E2E) 테스트의 주요 이점 중 하나는 여러 브라우저에서 애플리케이션을 테스트할 수 있다는 점입니다. 100% 크로스 브라우저 커버리지가 바람직해 보일 수 있지만, 크로스 브라우저 테스트는 일관되게 실행하는 데 추가 시간과 컴퓨터 자원이 필요하므로 팀의 자원에 대한 수익이 점점 줄어듭니다. 따라서 애플리케이션에 필요한 크로스 브라우저 테스트의 양을 선택할 때 이 트레이드오프를 염두에 두는 것이 중요합니다.

더 빠른 피드백 루프

엔드 투 엔드(E2E) 테스트와 개발의 주요 문제 중 하나는 전체 테스트 스위트를 실행하는 데 시간이 오래 걸린다는 점입니다. 일반적으로 이는 CI/CD 파이프라인에서만 수행됩니다. 최신 E2E

게 실행될 수 있도록 도왔습니다. 또한, 로컬 개발 시 작업 중인 페이지에 대한 단일 테스트만 선택적으로 실행하고, 테스트의 핫 리로딩을 제공하는 기능은 개발자의 워크플로우와 생산성을 높이는 데 도움이 됩니다.

일류 디버깅 경험

개발자들은 전통적으로 터미널 창에서 로그를 스캔하여 테스트에서 무엇이 잘못되었는지 파악했지만, 최신 엔드 투 엔드(E2E) 테스트 프레임워크는 개발자가 이미 익숙한 도구(예: 브라우저 개발자 도구)를 활용할 수 있도록 해줍니다.

헤드리스 모드에서의 가시성

엔드 투 엔드(E2E) 테스트는 CI/CD 파이프라인에서 종종 헤드리스 브라우저(즉, 사용자가 볼 수 있는 브라우저가 열리지 않음)에서 실행됩니다. 최신 E2E 테스트 프레임워크의 중요한 기능은 테스트 중 애플리케이션의 스냅샷 및/또는 비디오를 볼 수 있는 기능으로, 오류가 발생하는 이유에 대한 통찰을 제공합니다. 과거에는 이러한 통합을 유지하는 것이 번거로웠습니다.

권장 사항

Playwright는 Chromium, WebKit, Firefox를 지원하는 훌륭한 E2E 테스트 솔루션입니다. Windows, Linux, macOS에서, 로컬 또는 CI에서, 헤드리스 또는 헤디드로, Google Chrome for Android와 Mobile Safari의 네이티브 모바일 에뮬레이션까지 지원합니다. 정보가 풍부한 UI, 뛰어난 디버깅, 내장된 검증, 병렬화, 트레이스, 불안정한 테스트 제거를 위한 설계가 특징입니다. 컴포넌트 테스트도 지원하지만, 실험적입니다. Playwright는 오픈 소스이며 Microsoft에서 유지 관리합니다.

Cypress는 정보가 풍부한 그래픽 인터페이스, 뛰어난 디버깅, 내장된 검증, 스텁, 불안정성 저항, 스냅샷을 제공합니다. 위에서 언급했듯이, 컴포넌트 테스트에 대해 안정적으로 지원합니다. Cypress는 Chromium 기반 브라우저, Firefox, Electron을 지원합니다. WebKit 지원도 있지만, 실험적입니다. Cypress는 MIT 라이선스이지만, 병렬화와 같은 일부 기능은 Cypress Cloud 구독이 필요합니다.

기타 옵션

Nightwatch는 Selenium WebDriver를 기반으로 한 E2E 테스트 솔루션입니다. 이로 인해 가장 넓은 브라우저 지원 범위(네이티브 모바일 테스트 포함)를 가집니다. Selenium 기반 솔루션은 Playwright나 Cypress보다 느릴 수 있습니다.

입니다.

레시피

프로젝트에 Vitest 추가하기

Vite 기반 Vue 프로젝트에서 다음을 실행하세요:

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

다음으로, Vite 설정에 `test` 옵션 블록을 추가하세요:

vite.config.js

```js
import { defineConfig } from 'vite'

export default defineConfig({
  // ...
  test: {
    // jest와 유사한 전역 테스트 API 활성화
    globals: true,
    // happy-dom으로 DOM 시뮬레이션
    // (happy-dom을 peer dependency로 설치해야 함)
    environment: 'happy-dom'
  }
})
```

> **TIP**
>
> TypeScript를 사용하는 경우, `tsconfig.json` 의 `types` 필드에 `vitest/globals` 를 추가하세요.
>
> tsconfig.json
>
> ```json
> {
>   "compilerOptions": {
>     "types": ["vitest/globals"]
>   }
> }
> ```

트의 test 디렉터리나 소스 파일 옆의 test 디렉터리에 둘 수 있습니다. Vitest는 네이밍 규칙에 따라 자동으로 검색합니다.

MyComponent.test.js

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

test('정상 동작해야 한다', () => {
  const { getByText } = render(MyComponent, {
    props: {
      /* ... */
    }
  })

  // 출력 검증
  getByText('...')
})
```

마지막으로, `package.json` 에 테스트 스크립트를 추가하고 실행하세요:

package.json

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

```sh
> npm test
```

컴포저블 테스트하기

> 이 섹션은 컴포저블 섹션을 읽었다고 가정합니다.

컴포저블을 테스트할 때, 호스트 컴포넌트 인스턴스에 의존하지 않는 컴포저블과 의존하는 컴포저블로 나눌 수 있습니다.

컴포저블이 다음 API를 사용할 때 호스트 컴포넌트 인스턴스에 의존합니다:

라이프사이클 훅
Provide / Inject

수 있습니다:

counter.js

```js
import { ref } from 'vue'

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

  return {
    count,
    increment
  }
}
```

counter.test.js

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

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

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

라이프사이클 훅이나 Provide / Inject에 의존하는 컴포저블은 테스트를 위해 호스트 컴포넌트로 감싸야 합니다. 다음과 같은 헬퍼를 만들 수 있습니다:

test-utils.js

```js
import { createApp } from 'vue'

export function withSetup(composable) {
  let result
  const app = createApp({
    setup() {
      result = composable()
      // 템플릿 누락 경고 억제
      return () => {}
    }
  })
  app.mount(document.createElement('div'))
  // 결과와 app 인스턴스를 반환
  // provide/unmount 테스트용
```

```
}
```

foo.test.js

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

test('useFoo', () => {
  const [result, app] = withSetup(() => useFoo(123))
  // 주입 테스트를 위한 provide 모킹
  app.provide(...)
  // 검증 실행
  expect(result.foo.value).toBe(1)
  // 필요하다면 onUnmounted 훅 트리거
  app.unmount()
})
```

더 복잡한 컴포저블의 경우, 컴포넌트 테스트 기법을 사용해 래퍼 컴포넌트에 대한 테스트를 작성하는 것이 더 쉬울 수도 있습니다.

GitHub에서 이 페이지 편집

Previous
상태 관리

Next
서버 사이드 렌더링 (SSR)