Home

2022-05-01

프론트엔드 테스팅 전략

> 원문: https://itnext.io/front-end-testing-strategy-5fddfd463feb

다양한 테스트 계층를 정의하고 현재 상황을 짚어보며 효과적인 전략에 대해 제안합니다.

앞서 이 글에서 중점으로 보는 부분은 아래와 같습니다.

- 프론트엔드 테스팅만을 중점적으로 다룹니다 (예: 리액트 어플리케션과 컴포넌트와의 상호작용).
- 많은 부분이 Kent C. Dodds의 테스팅 원칙을 기반으로 합니다. 그는 리액트 생태계에서 가장 유명한 개발자 중 한 명이며 여러 오픈소스 라이브러리를 만들었습니다. 뿐만 아니라 높은 퀄리티의 교육 과정들과 글을 배포하고 있습니다.

I) 소개

1. 우리는 왜 테스트를 작성해야하는가?

테스트는 작업하는 흐름의 속도를 높이고 코드 품질을 개선하는 데 도움이 될 수 있지만 테스트를 작성하는 가장 크고 중요한 이유는 **자신감**입니다. 앞으로 작성하게 될 코드가 앱에 영향을 최소화 할 것이라는 자신감을 원합니다. 따라서 어떤 작업을 하던지 여러 종류의 테스트가 가능한 한 큰 자신감을 북돋아 줄

수 있게 만들고 싶을 것 입니다. 그러기 위해서는 테스팅할 때 만들어지는 트레이드오프에 대해 인지하는 것이 필요합니다.

Resources:

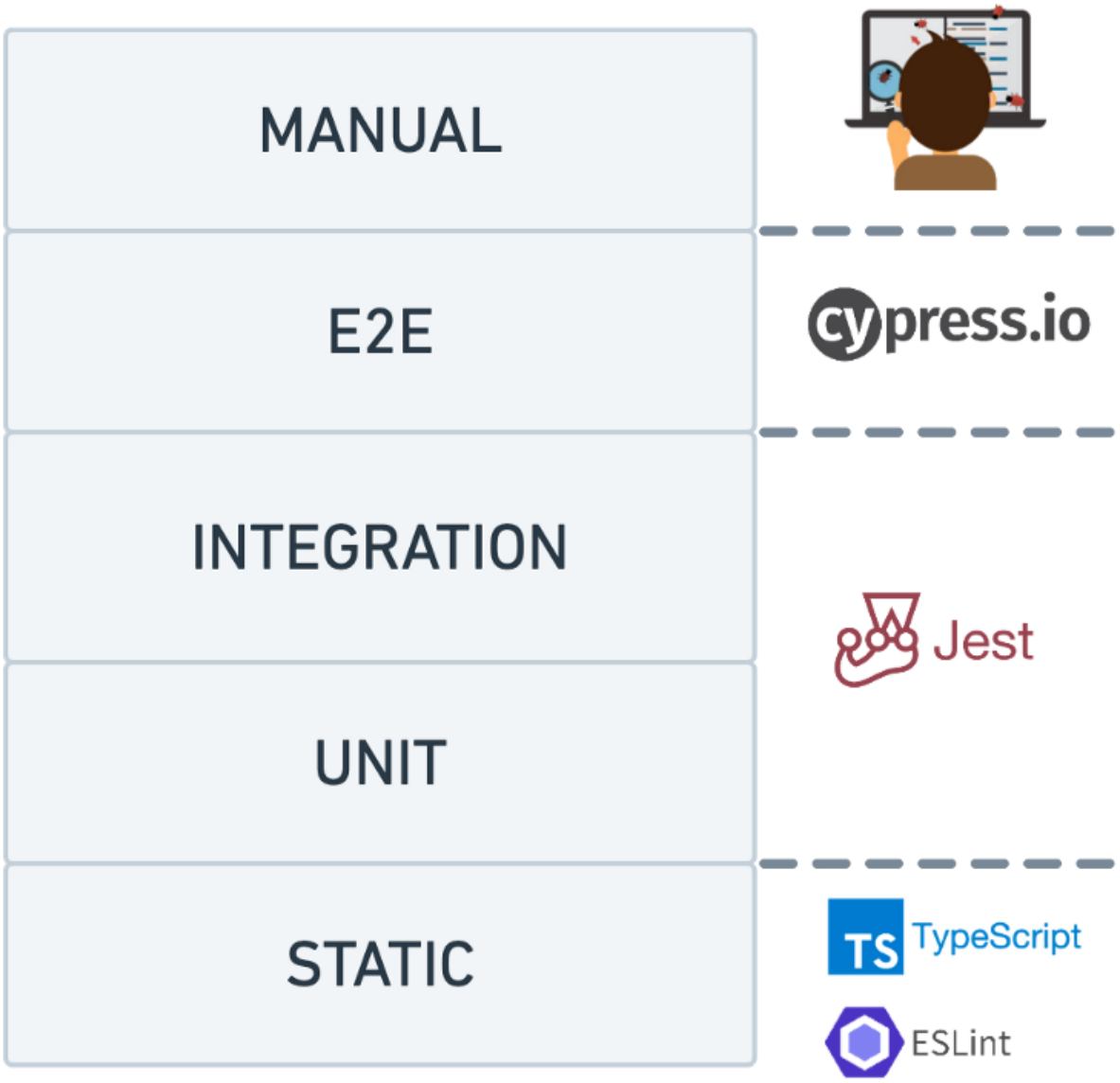
- Kent C. Dodds — Confidently Shipping Code
- Kent C. Dodds — How to know what to test?

2. 테스트의 종류

> 벽에 페인트를 던져 많은 부분의 벽을 칠할 수 있겠지만 브러시를 들고 벽에 오르지 않으면 모서리는 절대 칠하지 못할 것입니다. 🖌️

이 은유는 테스팅에 관한 내용에도 딱 맞게 적용됩니다. 왜냐하면 기본적으로 여러분이 세울 테스팅 전략을 올바르게 선택하는 것이 벽을 칠할 브러시를 선택할 때와 동일선상에 있기 때문입니다. 다양한 경우들에 대한 다양한 유형의 테스트가 (A/B 테스트, 성능 테스트, 연기 테스트, 회귀 테스트, ...)가 있으며 각각은 트레이드오프를 수반합니다.

개발자가 직접 테스트하는 수동 테스팅은 테스트 계층 중의 하나입니다. 자동화 테스팅에 대해 이야기할 때는 단위 테스트, 통합 테스트, 종단 간 테스트(End to End, E2E)와 같은 가장 일반적인 것들에 대해 알아봅니다. 그리고 자바와 같은 다른 언어와 달리 자바스크립트에서는 기본적으로 타입이 없으므로 일반적인 정적 도구 툴을 사용한 테스트도 알아봅시다.(Typescript, Flow, ESLint, ...).



정적 테스트

정적 테스트는 코드 수행 없이 실행될 수 있습니다. 설정이 쉽고 빠르며 애플리케이션을 개발하는 동안 오타 및 타입 에러를 지속적으로 포착할 수 있습니다.

`TypeScript` (타입 프로그래밍 언어) 및 `ESLint` (린터)는 이러한 유형의 테스트를 수행하는 일반적인 도구입니다.

단위 테스트

단위 테스트는 소프트웨어의 작고 독립적인 부분들(또는 원자 단위)이 예상대로 동작하는지 확인합니다. 일반적으로 종속성(공동 작업자)이 없거나 테스트를 위해 모킹된 것들에 대해 테스트합니다. `Jest` 는 해당 유형의 테스트를 수행하는 공통 도구입니다.

통합 테스트

통합 테스트는 여러 단위(기능, 구성 요소, 클래스 등)의 조합으로 함께 의도한 대로 동작하는지 확인합니다. 행동을 전체적으로 테스트하고 가능한 한 적게 모킹하려고 합니다. 통합 테스트는 다른 것들과 독립적인 하나의 시스템(예: 프론트엔드)을 다루고 있습니다.

`Jest` 는 이 유형을 테스트 하는데 가장 일반적으로 사용되는 도구입니다.

> "단위 란 무엇인가? vs "통합은 무엇입니까?" 코드를 보는 관점에 따라 다릅니다.
>
> 하나의 드롭다운이 있는 컴포넌트 라이브러리를 사용하는 애플리케이션 예로 들어봅시다.
>
> - 컴포넌트 라이브러리 관점: Dropdown의 내부의 함수는 단위이며 Dropwodn 전체는 통합입니다.
> - 애플리케이션 관점: Dropdown은 단위가 됩니다. 페이지의 Form은 통합입니다.

종단 간 테스트 (E2E)

E2E 테스트(어떤 때는 "함수형 테스팅"이라고 불리기도 하는) 모든 시스템(Front-end, Back-end, ...)을 포함하여 테스트합니다. 브라우저를 자동화하고 애플리케이션를 통틀어 정형화한 사용자 흐름을 재현하려고 합니다. (애플리케이션을 로드하고, 로그인을 한 뒤, 페이지와 상호작용하기 등등...). E2E는 여러분의 소프트웨어의 큰 부분을 보장할 수 있도록 도와줍니다.

`Cypress` 는 이 유형을 테스트하는데 가장 일반적으로 사용되는 도구입니다.

수동 테스트

수동 테스트는 자동화된 도구를 사용하지 않고 수행되는 테스트를 제안합니다.

사람이 사용자가 어떤 것을 하는지 정확히 재현합니다.

예시)

1. 컴퓨터에 앞에 앉는다.
2. 브라우저로 애플리케이션을 연다.
3. 로그인을 한다.
4. 여기저기 클릭해본다.

목표는 개발자가 예상하지 못하거나 자동화된 도구에서 감지하지 못한 버그를 잡는 것입니다.

Resources:

- Kent C. Dodds — What We Can Learn About Testing From The Wheel
- Kent C. Dodds — The Testing Trophy and Testing Classifications
- Kent C. Dodds — Static vs Unit vs Integration vs E2E Testing for Frontend Apps

3. 어떤 테스트가 가장 큰 자신감을 주는가?

> 테스트가 소프트웨어 사용 방식과 유사할수록 더 많은 자신감을 줄 수 있습니다.

테스트를 통해 여러분은 소프트웨어가 릴리스 하였을 때 의도한 대로 동작하고 있는지 확인하려 합니다. 이렇게 하려면 최선의 방법은 소프트웨어에 대한 최종 사용자의 관점을 염두에 두고 테스트를 작성하는 것입니다.

간단한 원칙에 따라 수동 테스트는 최종 사용자가 애플리케이션과 상호작용 하는 것과 가장 유사하기 때문에 최선의 선택지처럼 보입니다. 그러나 소프트웨어를 테스트할 때 규모에 유연한 해결책이 아닙니다.

여러분은 소프트웨어에서 만든 어떠한 변경사항도, 개발환경에서 프로덕션에 배포하기 전에는 아무것도 깨지지 않길 바라겠지만 만약 이것들을 수동 테스트를 한다면 굉장히 오래 걸릴 겁니다. 덧붙여서 수동 테스트는 많은 휴먼 에러를 만들게 합니다.

그렇기 때문에 대안이 더 빠르고, 확장 가능하며, 휴먼 에러로부터 해방되는 **자동화 테스트**를 선호하는 것입니다. (잘 작성된 경우)

> [역주] human error: 사람의 실수로 만들어지는 오류 (위키백과)

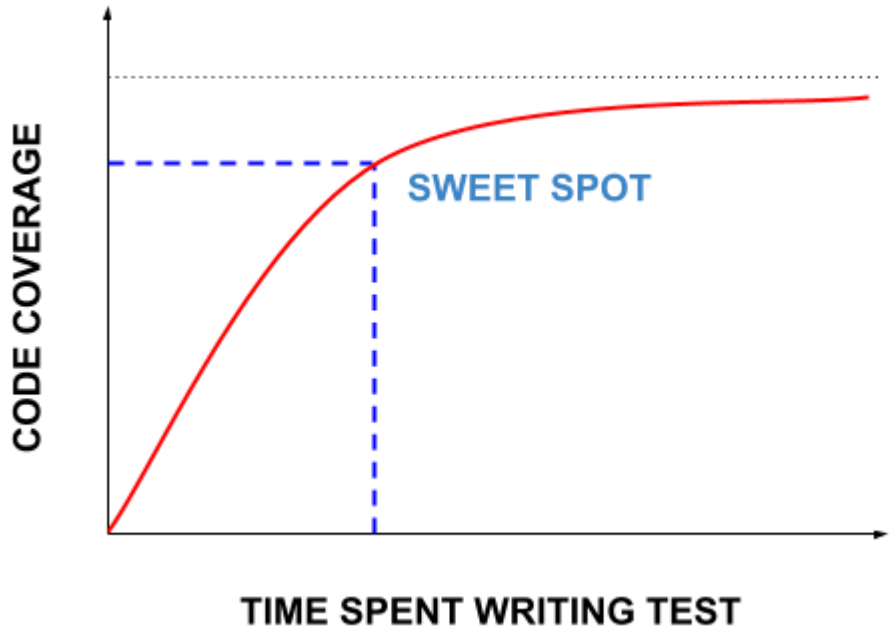
Resources:

- Testing Library — Guiding Principles

II) 트레이드오프에 대해서 이야기해봅시다

> 트레이드오프: 트레이드오프(trade-off, tradeoff) 또는 상충 관계는 다른 측면에서 이득을 얻으면서 (중략) 잃어버리는 일이 수반되는 상황적 결정이다. 즉, 하나가 증가하면 다른 하나는 무조건 감소한다는 것을 뜻한다. (위키백과)

1. 코드 커버리지



애플리케이션에 대한 100% 코드 커버리지를 의무화하는 것은 좋은 생각이 아닙니다. 코드 커버리지가 특정 숫자를 넘어 증가할 때 테스트로부터 받는 이득이 적어지게 됩니다 (70%로 가정해봅시다). 만약 100%를 달성하기 위해 노력한다면, 테스트가 필요하지 않은 것을 위해 많은 시간을 보내고 있는 자신을 발견하게 될 것입니다.

테스트가 필요하지 않은 것들은 다음과 같습니다.

- **로직이 없는 것들**: 어떤 버그도 ESLint나 Typescript에 의해 발견될 수 있습니다.
- **구현 세부사항**: 구현 세부사항은 여러분의 소프트웨어가 잘 동작하고 있을 때 큰 자신감을 주지는 않습니다. 그리고 여러분의 리팩터링을 느리게 만듭니다. 동작이 동일하게 유지되므로 코드를 리팩터링 할 때 매우 드문 경우로 테스트를 변경해야할 것입니다. 이것은 아래 사항으로 도달하게 합니다.
  - 거짓 음성: 애플리케이션 코드를 리팩터링 할 때 깨질 수 있습니다.
  - 거짓 양성: 애플리케이션 코드가 깨져도 실패하지 않을 수 있습니다.

> **구현 세부 사항을 어떻게 결정하나요?**
>
> 여러분의 테스트가 실제 코드가 사용되지 않는 경우에 대해 무엇인가를 한다면 그것은 세부 구현 사항을 테스팅 하는 것 입니다(예를 들면 공개된 private 함수).
>
> - [역주] 거짓 음성: 스팸 메일이 아닌데 스팸 메일이라고 하는 것
> - [역주] 거짓 양성: 스팸 메일인데 스팸 메일이 아니라고 하는 것

코드 커버리지에 대한 일반적인 오해

코드 커버리지가 의미하는 것은 무엇일까요?

- 이 테스트가 실행될 때 이 코드 라인이 실행되었습니다.

코드 커버리지가 의미하지 않는 것은 무엇일까요?

- 코드의 이 부분은 비즈니스 요구 사항에 따라 동작합니다.
- 코드의 이 부분은 애플리케이션의 다른 모든 코드들과 잘 동작합니다.
- 코드의 이 부분은 다른 부분보다 테스트하는 것이 더 중요합니다.

코드 커버리지 < 유스 케이스(Use Case) 커버리지

테스트 중인 코드 자체보다는 유스 케이스에 대해서는 더 신경 써야 합니다. 그 이유는 아래와 같습니다.

> 유스 케이스: 시스템의 동작을 사용자의 입장에서 표현한 시나리오(= 사용자의 행위 또는 시나리오) (위키백과)

- 코드의 변경 사항이 유스 케이스보다 더 많습니다.
- 유스 케이스가 깨졌다고 하더라도 코드는 "동작" 할 수 있습니다.

Resources:

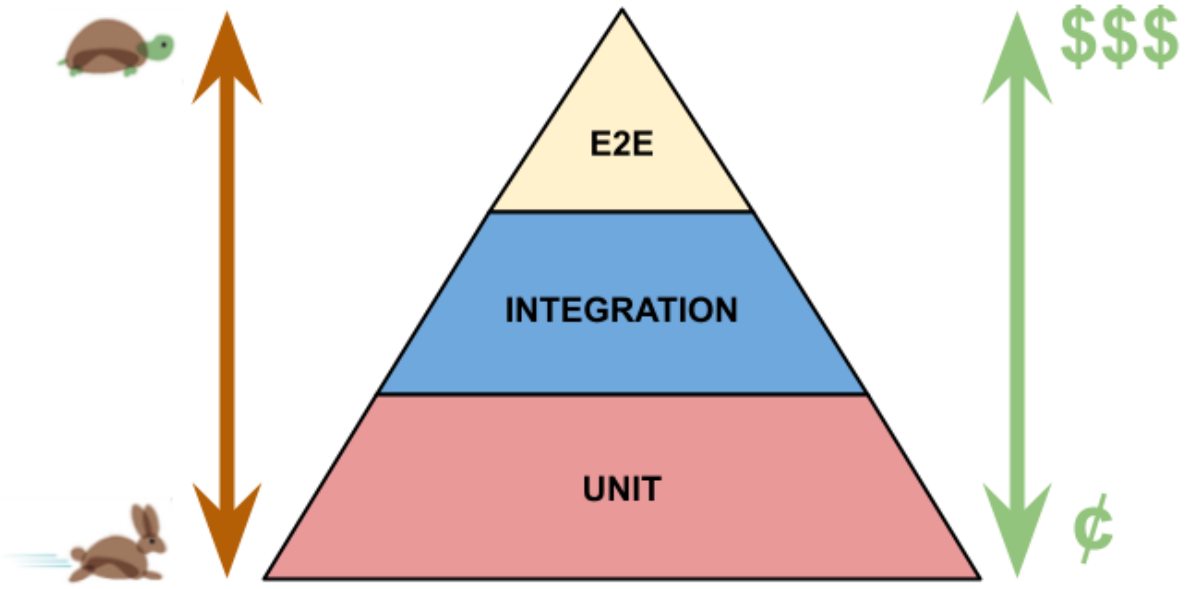
- Kent C. Dodds — Common Testing Mistakes
- Aaron Abramov — Establishing testing patterns with software design principles
- Kent C. Dodds — Testing Implementation Details
- Kent C. Dodds — Avoid the Test User
- Kent C. Dodds — How to know what to test?

2. 무엇을 테스트해야 하는지 어떻게 알 수 있을까요?

모든 것을 테스트할 수 없으므로 여러분의 노력을 집중할 곳을 결정해야 합니다.

테스팅 피라미드

여러분은 다양한 테스트 계층에 대해 이야기하는 인기 있는 방법인 테스트 피라미드에 대해 이미 알고 있을 수도 있습니다. 다음은 마틴 파울러의 블로그와 Google Testing blog의 내용을 같이 이야기 합니다.



피라미드의 각각의 형태가 가지는 크기는 여러분이 애플리케이션을 테스팅할 때 얼마만큼 집중해야 하는가와 관련이 있습니다. 두 화살표는 피라미드를 통과할 때 생겨나는 트레이드오프 관계를 나타냅니다. 피라미드를 올라갈 때 테스트는 쓰기/실행이 느리게 진행되고 (시간과 자원 측면에서) 실행/유지하는 데 더 비쌉니다. 올라갈수록 여러분의 테스트는 더 까다로워고 실패하게 되는 점들이 많아지는 경향이 있습니다.

이 두 가지 관점을 따르면, 저렴하고 빠른 단위 테스트에 더 많은 시간을 보내야 한다는 것을 보여줍니다.

하지만 몇 가지 문제가 있습니다.

1 - 이 피라미드는 2012년에 만들어졌으며 이 가정을 바탕으로 합니다. (Martin Fowler의 블로그의 하단 메모에서 제공)

> **2:** The pyramid is based on the assumption that broad-stack tests are expensive, slow, and brittle compared to more focused tests, such as unit tests. While this is usually true, there are exceptions. If my high level tests are fast, reliable, and cheap to modify - then lower-level tests aren't needed.

> 피라미드는 단위 테스트와 같은 집중 테스트에 비해 범위가 넓은 테스트가 비싸고 느리며 부서지기 쉽다는 가정을 기반으로 합니다. 이것은 일반적으로 사실이지만 예외가 있습니다. 높은 수준의 테스트가 빠르고 안정적이며 수정 비용이 저렴하다면 낮은 수준의 테스트가 필요하지 않습니다.

지금은 조금 다릅니다. 테스트 도구는 이전과 비교할 때 훨씬 우수해졌습니다. (성능 측면에서는 더 나은 것뿐만 아니라 테스트의 어떤 부분이 어떤 문제로 실패를 일으켰는지 알 수 있게 되었습니다).

2 - 피라미드는 프론트엔드 테스트 생태계가 지금만큼 크지 않은 시기에 만들어졌습니다. 10년 전, 우리의 도구는 프론트엔드에서 사용자를 재현할 수 없었습니다.

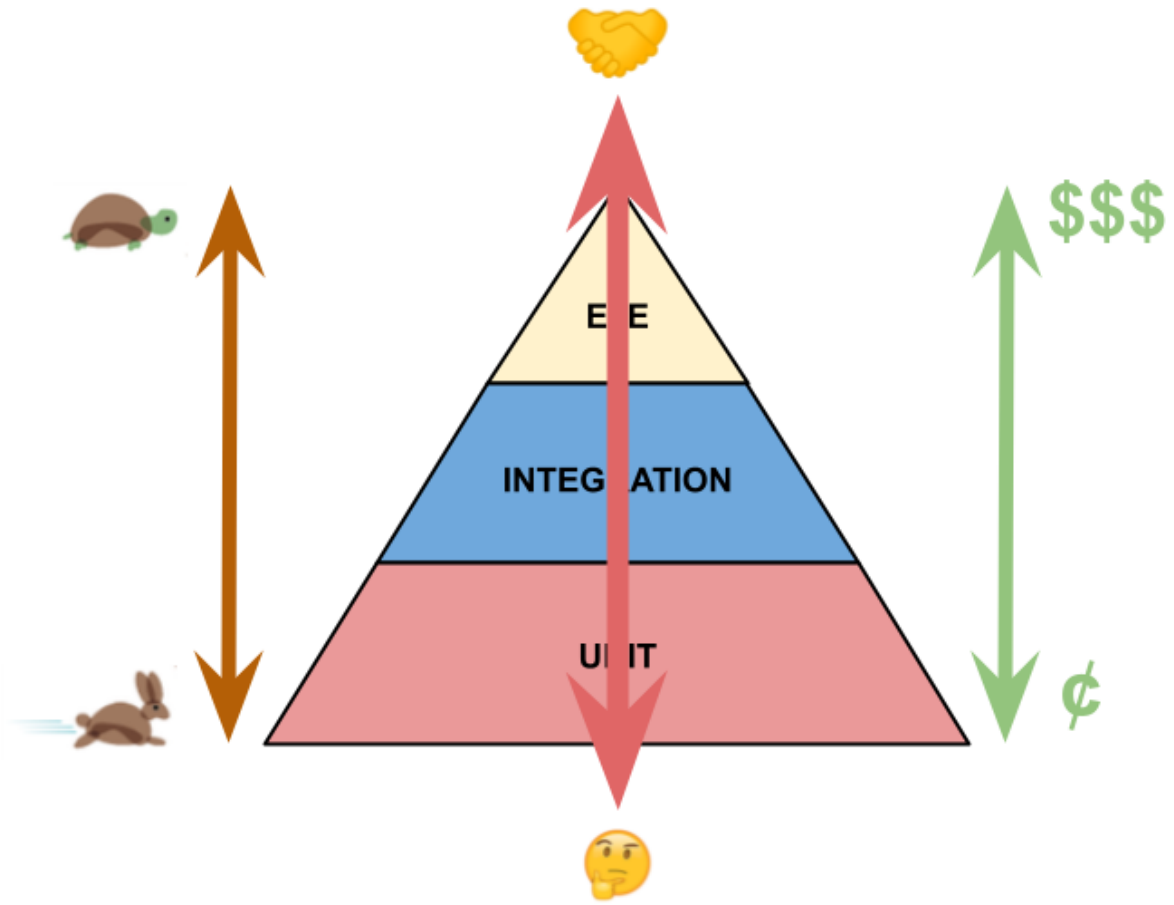
대부분의 테스트, 도구 및 피라미드는 백엔드에 중점을 두었습니다. 이유는 백엔드가 프론트엔드 테스트와 비교하여 테스트하기가 더 쉽기 때문입니다. 백엔드는 다른 소프트웨어(사용자로 간주될 수 있음)와 직

접 대화하는 소프트웨어를 구축합니다. 그래서 마치 그들이 테스트를 하기 위하여 원하는 대로 코드를 작성하여 테스트를 작성하는 것처럼 보이는 것은 자연스러운 일입니다.

프론트엔드의 경우, 사용자는 손가락과 손이 있으며 클릭을 하고 키보드에 무언가를 입력할 수 있습니다. 또한, 잠재적으로 일부 스크린 리더를 사용하는 사용자가 될 수 있습니다. 동작을 재현하는 것이 훨씬 더 복잡합니다. 다행히도 최근 몇 년 동안 많은 것들이 바뀌 었으며 이제는 사용자와 같은 방식으로 소프트웨어를 테스트할 수 있는 더 나은 도구 (예: `Testing Library`)를 보유하고 있습니다.

3 - 피라미드는 정적 테스트를 다루지 않습니다. 정적 유형 언어를 사용하는 개발자를 위해 정의 되었기 때문입니다 (예: `Java` , `C`). 그러나 JavaScript를 사용하고 그러한 피드백을 얻으려면 외부 도구 (`TypeScript` , `Eslint`)를 추가해야하므로 실제로는 테스트 계층 중 하나로 볼 수 있습니다.

4 - 이 피라미드에는 남은 한 가지 측면이 있습니다. 피라미드를 올라갈 때 각 형태의 테스트의 신뢰 계수가 증가합니다.



신뢰 계수는 테스트 피라미드가 높을수록 소프트웨어를 사용하는 방식과 비슷한 테스트에 가까워지고 있습니다. 그리고 여러분이 그로부터 얻는 자신감도 올라갑니다.

> 예를 들어, 결제 기능이 있는 경우 실제 사용자와 상호 작용하는 것과 동일한 방식으로 전체 프로세스를 테스트하면 그 과정에 사용된 모든 하위 요소를 함께 보장할 수 있습니다.

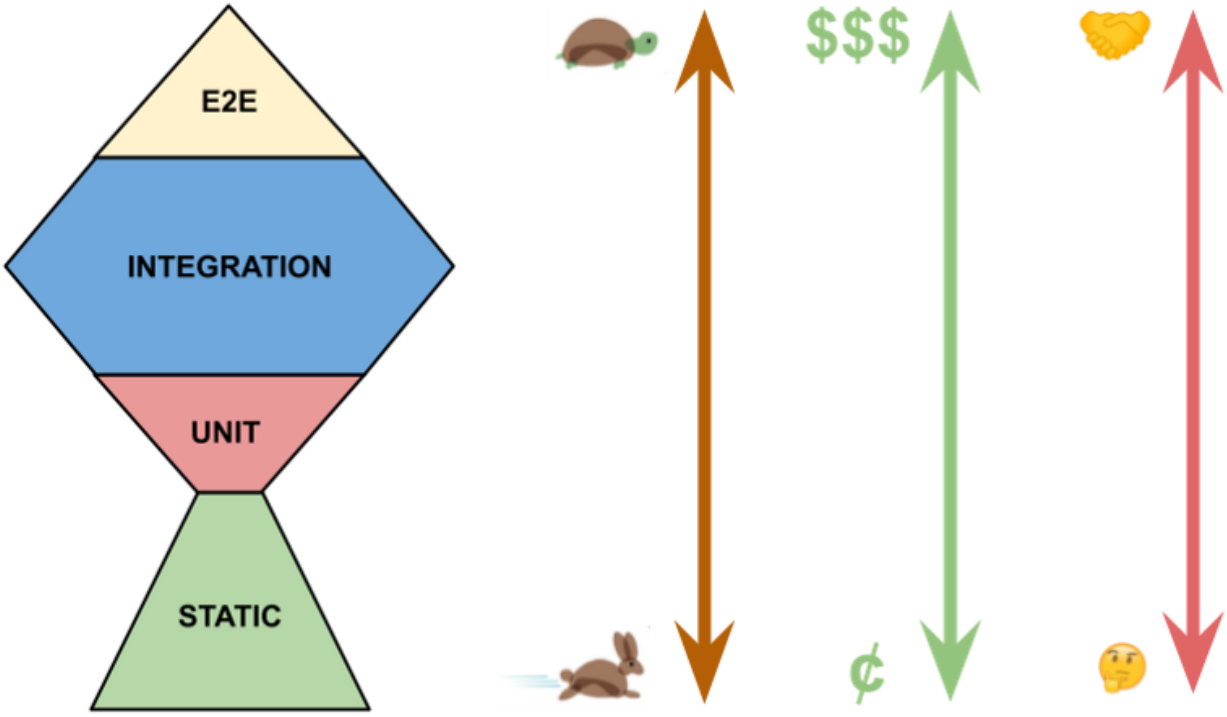
따라서 E2E 테스트는 단위 테스트보다 느리고 비싸지만 애플리케이션이 의도한 대로 동작한다는 확신을 가져옵니다. 반대로, 단위 테스트는 더 빠르고 저렴하지만, 아래 내용들을 여러분 자신에게서 발견할 수

있습니다.

- 여러 부분으로 나누어 테스트합니다 (이들이 함께일 때도 제대로 동작한다는 확신을 갖지 못함)
- 구현 세부사항을 테스트
- 매우 많은 모의 객체를 만들어 냄

이것들은 "사용자와 상호 작용하는 것과 동일한 방식으로 응용 프로그램을 테스트하는 것"의 원칙에서 벗어나는 많은 요인들입니다.

테스팅 트로피



테스트 트로피를 사용하면 **e2e**, **통합** 및 **단위**의 동일한 테스트 스택을 유지합니다. 하지만 각각의 비율을 변경했습니다. 또한 JavaScript의 현실을 나타내기 위해 **정적** 계층를 추가합니다.

E2E tests 소프트웨어가 의도한 대로 잘 동작한다는 확신을 갖는 자동 테스트 유형입니다. 아마도 미래에 우리의 E2E 도구들은 우리가 100% 집중할 수 있도록 훨씬 나아질 것이지만, 현재도 여전히 작성하고 실행하는 것이 비쌉니다. 따라서 여러 시스템 (프론트엔드, 백엔드 등)의 상호 작용이 필요하고 프로덕션 데이터를 나타내는 일부 높은 레벨의 중요한 테스트에만 집중할 것입니다. 따라서 E2E에서는 일반적으로 성공하는 시나리오만 가정하고 그 이상을 넘어가는 테스트는 피할 것입니다.

통합 테스트는 대부분의 노력을 쏟아야 하기 때문에 트로피의 가장 넓은 계층입니다. 통합 테스트는 자신감과 속도/비용 사이의 트레이드오프에 좋은 균형을 이루어 테스트를 작성하고 실행합니다. 또한 통합 테스트는 몇가지 성공 시나리오와 불행한 시나리오를 테스트합니다. 통합 테스트에서는 소프트웨어의 여러 부분이 의도한 대로 잘 동작한다는 것을 보장하는 테스팅을 함으로써 최종 사용자와 상호 작용하는 것과 같은 방식으로 많은 사용 사례를 다룰 수 있습니다. 그럼으로써 대체로 작은 독립적인 내용들을 테스트하는 것을 신경 쓰지 않아도 된다는 것을 알게 됩니다.

단위 테스트는 트로피의 가장 작은 부분입니다. 그것은 작은 엣지 케이스를 테스트하는 데 집중하는 것 또는 일반적으로 복잡한 논리가 많은 순수한 기능에 초점을 맞추는 것입니다. 여기서는 비즈니스 사례를 테스트하지 않습니다 (그 사례들을 최종 사용자가 가질 경우). 단위 테스트의 사용자는 일반적으로 개발자의 관점에서 그 기능을 호출하는 다른 개발자입니다.

정적 테스트는 애플리케이션을 실행하지 않고도 많은 정보를 제공하기 때문에 모든 곳에 적용해야 합니다.

> 각각의 비율은 100% 정확하지 않으며 대부분 시간을 단위 테스트보다는 E2E를 더 많이 가져가는 것을 권합니다.

Resources:

- Kent C. Dodds — How to know what to test?
- Kent C. Dodds — Why you should double down on integration tests?

III) 결론

> 테스트를 작성하세요. 너무 많지 않게. 통합을 중점적으로.

Guillermo Rauch (Vercel의 CEO 및 창립자)의 이 트윗은 테스트 철학의 완벽한 요약입니다.

- **테스트를 작성하세요**: 소프트웨어가 의도한 대로 작동한다는 확신을 갖습니다.
- **너무 많지 않게**: 100% 코드 범위를 의무화하지 마세요. 테스트 중인 코드 자체보다는 그 코드가 기반이 되는 유스 케이스에 대해서는 더 신경 써야 합니다.
- **통합을 중점적으로**: 테스트를 작성하고 실행하기 위한 신뢰와 속도/비용 사이의 트레이드오프에서 가장 적합한 균형을 가지고 있습니다.

수동 테스트는 테스트 전략 중에서 수십 년 동안 여전히 자리를 차지할 것이라는 점도 언급하는 것이 중요합니다. 개발자가 예상하고 테스트하지 않은 일부 사용 사례가 항상 있을 것이며, 실제 사람의 눈은 그 부분에 도움이 될 수 있습니다. 그러나 자동화된 테스트를 작성하면 신뢰도가 높아지고 수동 테스트의 필요성이 줄어듭니다. 수동 테스트에는 많은 이점이 있지만 필요한 시간 및 비용과 같은 많은 문제가 있습니다.

다시 한번 트레이드오프가 무엇인지 짚어본다면, 실제 사용자에게 중요한 것이 무엇인지 그래서 여러분이 어디에 노력을 쏟아야 할지에 관한 것이라고 할 수 있습니다. 여러분은 변경 사항을 배포할 때 확신을 가지고 코드가 비즈니스 요구 사항을 충족시키고 사용자에게 멋진 경험을 제공하고 싶을 것입니다. 그러기 위해서는 다양한 테스트 전략을 함께 사용해야 합니다.

읽어주셔서 감사합니다.

Resources:

- Kent C. Dodds — Write tests. Not too many. Mostly integration.
- Kent C. Dodds — Confident React
- Adrià Fontcuberta — The Pragmatic Front End Tester

1 Comment - *powered by utteranc.es*

doong-jo commented on 2022년 5월 1일

혹시, 테스트를 작성하는 것이 나의 상황과 맞지 않고, 함께 할 동료들이 없어서 "알긴 아는데 :

켄트 벡(Kent Beck)의 익스트림 프로그래밍 2판 서문에 감동적인 글귀가 있습니다.

> 상황이 어떻건 간에 당신은 언제나 더 나아질 수 있습니다. (No matter the circumstance you can always i
> 당신은 언제나 자기 자신부터 개선을 시작할 수 있습니다. (You can always start improving with yourself)
> 당신은 언제나 오늘부터 개선을 시작할 수 있습니다. (You can always start improving today)

사실 우리가 할 수 있는 것은 정말 많습니다. 그리고 그것은 바로 지금, 바로 여기에서 시작할 수 있습니다. 지금 여기(

Write Preview

Sign in to comment

Styling with Markdown is supported