Home

2022-08-17

테스트 가능한 프런트엔드: 좋은 것, 나쁜 것, 깨지기 쉬운 것

> 원문: https://www.smashingmagazine.com/2022/07/testable-frontend-architecture/

짧은 요약

테스팅은 단지 도구와 프로세스에 대한 것이 아닙니다. 이것은 설계에 대한 것입니다. 이 글에서는 노암 로젠탈님이 프런트엔드와 서브시스템 그리고 다른 전략들 사이에서 테스팅을 조직하고 옳은 균형을 찾는 것에 대해 공유합니다.

단위, 통합 및 E2E 테스트 간에 테스트를 구성하는 방법과 UI 컴포넌트를 테스트하는 방법 등 반복적이고 정말로 어려운 딜레마에 직면한 프런트엔드 개발자, 관리자 및 팀을 자주 만나게 됩니다.

단위 테스트는 종종 사용자와 시스템에서 발생하는 "관심있는" 일들은 포착하지 못하는 것처럼 보이며, E2E 테스트는 일반적으로 실행하는 데 오랜 시간이 걸리거나 복잡한 구성이 필요합니다. 게가 테스팅 도구는 너무나도 많습니다(JEST, Cypress, Playwright 등). 그 모든 것을 어떻게 이해할 수 있을까요?

참고: 이 글은 예시와 시맨틱에 React를 사용하지만, 일부 가치들은 모든 UI 개발 패러다임에 적용됩니다.

왜 프런트엔드를 테스팅하는 것은 어려울까요?

우리는 프런트엔드를 하나의 시스템으로 만드는 것이 아니라 사용자 인터페이스 스토리를 구성하는 여러 컴포넌트와 기능으로 만드는 경향이 있습니다. HTML, JS 및 CSS를 구분하는 대신 자바스크립트 또는 JSX에 주로 사용되는 컴포넌트 코드를 사용하여 뷰 코드와 비즈니스 로직 코드를 혼합하는 것이 그 어느 때보다 매력적입니다. 제가 "우리"라고 말하는 것은 개발자 또는 컨설턴트로서 접한 거의 모든 웹 프로젝트를 의미합니다.

한 코드를 테스트하려고 할 때, 우리는 종종 React 컴포넌트를 렌더링하고 결과를 테스트하는 React Testing Library와 같은 것에서부터 시작하거나, 프로젝트와 잘 작동하도록 Cypress를 구성하느라 정신없이 하다가 잘못된 구성으로 끝나거나 포기하는 경우가 많습니다.

프런트엔드 테스팅 시스템을 구축하는 데 필요한 시간에 대해 관리자들과 이야기를 나눌 때, 관리자들과 우리는 그것이 무엇을 수반하는지, 그리고 우리의 노력이 결실을 맺는지, 그리고 우리가 무엇을 구축하든

최종 제품의 품질과 구축 속도에 얼마나 중요한지 정확히 알지 못합니다.

도구와 프로세스

팀에 일종의 **"의무적인 TDD"**(테스트 기반 개발) 프로세스가 있거나, 코드의 X%를 테스트로 처리해야 하는 코드 커버리지 게이트가 있으면 더 심각해집니다. 우리는 프런트엔드 개발자로서 하루를 마무리하고, 몇 개의 리액트 컴포넌트, 사용자 지정 후크 및 Redux 리듀서에 뿌려진 몇 개의 라인을 수정하여 버그를 수정한 다음, 우리가 한 일을 "커버링"하기 위한 "TDD" 테스트를 고안해야 합니다.

물론, 이것은 TDD가 아닙니다. TDD에서는 먼저 실패하는 테스트를 작성했을 것입니다. 하지만 제가 접한 대부분의 프런트엔드 시스템에서는 이와 같은 작업을 수행할 수 있는 인프라가 없으며, 중요한 버그를 수정하려고 할 때 먼저 실패하는 테스트를 작성하라는 요청은 비현실적인 경우가 많습니다.

커버리지 도구와 필수적인 단위 테스트는 우리 업계가 특정 도구와 프로세스에 집착하는 증상입니다. "당신의 테스트 전략은 무엇입니까?"는 종종 "우리는 TDD와 Cypress를 사용합니다" 또는 "우리는 MSW로 모킹합니다" 또는 "우리는 React Testing Library와 함께 Jest를 사용합니다"로 대답합니다.

별도의 QA/테스트 조직을 보유한 일부 회사는 테스트 계획처럼 보이는 것을 만들려고 합니다. 그러나 이러한 문제는 개발과 함께 테스트를 작성하기가 어려운 다른 문제에 도달하는 경우가 많습니다.

Jest, Cypress 및 Playwright와 같은 툴은 훌륭하며, 코드 커버리지는 의미가 있고, TDD는 코드 품질을 유지하기 위한 결정적인 작업 방식입니다. 그러나 이러한 솔루션은 종종 아키텍처를 대체합니다. 즉, 인터페이스의 좋은 계획, 단위 간의 좋은 기능 특징, 시스템의 명확한 API, 제품의 명확한 UI 정의 등 오래된 우려 사항들과의 분리입니다. 하지만 **프로세스는 아키텍처가 아닙니다.**

나쁜 것

필수 테스트 규칙이나 CI의 코드 커버리지 게이트와 같은 조직의 프로세스를 존중하기 위해 Jest 또는 다른 테스팅 도구를 사용하여 변경한 코드베이스의 모든 부분을 모킹하고, 이제 "올바른" 결과를 제공하는지 확인하는 하나 이상의 "단위" 테스트를 추가합니다.

이 테스트의 문제는 쓰기 어려운 것 외에도, 이제 사실상의 **규약**을 만들었다는 것입니다. 우리는 함수가 예상된 결과를 제공한다는 것을 확인할 뿐만 아니라, 이 기능이 테스트가 예상하는 시그니처를 가지고 있으며, 시뮬레이션된 실험과 같은 방식으로 환경을 사용한다는 것도 검증하고 있습니다. 만약 우리가 그 기능 시그니처가 그 환경에서 어떻게 사용하는지를 리팩토링 하고 싶다면, 그 테스트는 우리가 의도하지 않은 규약으로 인한 무거운 짐이 될 것입니다. 기능이 작동하더라도 실패할 수 있으며, 내부 변경 사항으로 인해 시뮬레이션된 환경이 더 이상 실제 환경과 일치하지 않기 때문에 성공할 수 있습니다.

> (역주) 시그니처: 함수의 입력과 출력을 정의합니다. 규약: 제공자와 사용자와의 규약을 의미합니다. 예를 들어, 어떠한 REST API가 있다고 가정했을 때 API 스펙을 바탕으로 사용자와 규약이 만들어집니다.

만약 여러분이 이렇게 테스트를 작성하고 있다면 **부디 그만두세요.** 시간을 낭비하고 제품의 품질과 속도를 악화시키고 있습니다.

> "명시되지 않은 시뮬레이션 환경의 환상 속의 세계를 만들고 내부 기능 시그니처와 내부 환경 상태에 의존하는 테스트보다는 자동화 테스트를 아예 하지 않는 것이 좋습니다."

규약

테스트가 좋은지 나쁜지를 이해하는 좋은 방법은 쉬운 영어(또는 모국어)로 규약을 작성하는 것입니다. 규약은 테스트 뿐만 아니라 **환경에 대한 가정**도 나타내야 합니다. 예를 들어, "사용자 이름 U와 암호 Y를 지정하면 이 로그인 함수는 OK를 반환합니다."규약은 보통 상태와 기댓값입니다. 위의 내용은 좋은 규약입니다. 상태와 기댓값은 명확합니다. 투명한 테스트 관행을 가진 회사들에게 이것은 새로운 소식이 아닙니다.

"현재 useState훅 값이 14이고 Redux 스토어가 userCache라는 배열을 3명의 사용자로 보유하고 있는 환경에서 로그인 기능은 ...해야 합니다."라는 구현 세부 정보로 인해 더 복잡해집니다.

이 규약은 구현 방법에 따라 특정되기 때문에 매우 취약합니다. 규약을 안정적으로 유지하고, 비즈니스 요구사항이 있을 때 규약을 변경하며, 구현을 유연하게 해야합니다. 환경에서 의존하고 있는 것들이 튼튼하게 잘 정의되어 있는지 확인해보세요.

깨지기 쉬운 것

관심사 분리가 누락되고, 시스템 간에 명확한 API가 없으며, 기능과 함께 명확한 시그니처와 기댓값이 부족하면 기능 또는 회귀 분석을 테스트할 수 있는 유일한 방법이 E2E로 끝납니다. 이는 E2E 테스트가 전체 시스템을 실행하고 사용자에게 가까운 특정 스토리가 예상대로 작동하도록 하기 때문에 나쁘지 않습니다.

E2E 테스트의 문제는 범위가 매우 넓다는 것입니다. 전체 사용자 여정을 테스트하면 일반적으로 새 기능이 라이브되고 있거나 회귀하는 적절한 지점을 찾고 테스트 사례를 실행하는 전체 프로세스를 인증하고 처음부터 환경을 설정해야 합니다.

E2E의 특성상, 이러한 각 단계는 CI 실행 시 중단되거나 지연될 수 있는 많은 시스템에 의존하기 때문에 **예측 불가능한 지연**이 발생할 수 있으며, "선택기"(사용자가 수행하는 작업을 프로그래밍 방식으로 모방하는 방법)를 신중하게 만들어야 합니다. 일부 규모있는 팀은 이를 위해 근본 원인 분석을 위한 시스템을 갖추고 있으며, 이를 위한 testim.io과 같은 솔루션도 있습니다. 하지만, 이것은 해결하기 쉬운 문제가 아닙니다.

대다수의 경우 버그는 기능이나 시스템에 존재하는데, 여기까지 도달하기 위해 전체 제품을 실행 하는 것은 너무 많은 테스트입니다. 새 코드 변경으로 인해 관련 없는 사용자 이동 경로에 회귀 분석이 표시될 수 있습니다.

E2E 테스트는 전반적인 테스트 조합에서 확실히 자리를 잡았으며 하위 시스템에 국한되지 않는 문제를 찾는 데 유용합니다. 그러나 이러한 시스템에 너무 많이 의존하는 것은 서로 다른 시스템 간의 우려와 API 장벽의 분리가 충분히 정의되지 않았음을 나타냅니다.

좋은 것

단위 테스트는 제한적이거나 모킹을 많이 하는 환경에 의존하고 E2E 테스트는 비용이 많이 들고 깨지기 쉬운 경향이 있기 때문에 **통합 테스트**는 종종 좋은 중간 기반을 제공합니다. UI 통합 테스트를 통해 전체 시스템이 다른 시스템과 격리되어 실행되므로 모킹될 수 있지만 시스템 자체는 수정 없이 실행됩니다.

프런트엔드를 테스트하는 경우, 전체 프런트엔드를 하나의 시스템으로 실행하고 다른 시스템/"백엔드"를 시뮬레이션하여 시스템과 무관한 깨짐 및 다운타임을 방지합니다.

프런트엔드 시스템이 너무 복잡해지면 일부 로직을 하위 시스템으로 포팅하고 이러한 하위 시스템에 대한 명확한 API를 정의합니다.

균형을 맞추세요

코드를 하위 시스템으로 분리하는 것이 **항상** 올바른 선택은 아닙니다. 모든 변경 사항에 대해 하위 시스템과 프런트엔드를 모두 업데이트하는 경우 분리가 도움이 되지 않는 오버헤드가 될 수 있습니다.

하위 시스템 간의 계약이 어느 정도 자율적으로 만들 수 있는 경우 하위 시스템에 UI 로직을 분리합니다. 또한 마이크로 프런트엔드는 때로는 올바른 접근 방식이지만 특정 문제를 이해하는 것보다는 솔루션에 중점을 두기 때문에 이 부분에서 주의해야 합니다.

UI 컴포넌트를 분할 정복하여 테스트합니다.

UI 컴포넌트 테스트의 어려움은 테스팅의 일반적인 어려움 중 특별한 경우입니다. UI 컴포넌트의 주요 문제는 해당 API와 환경이 제대로 정의되지 않는 경우가 많다는 것입니다. 리액트 세계에서 컴포넌트는 일부 종속성을 가집니다. 일부는 "prop"이고 일부는 hook(예: 컨텍스트 또는 Redux)입니다. 리액트 세계 외부의 컴포넌트는 종종 다른 버전의 동일한 전역 변수들에 의존합니다. 일반적인 React 컴포넌트 코드를 볼 때, 테스트하는 방법에 대한 전략이 혼란스러울 수 있습니다.

UI 테스트가 어렵지만 이 중 일부는 피할 수 없습니다. 하지만 다음과 같은 방법으로 문제를 나누면, 우리는 문제를 상당히 줄일 수 있습니다.

로직과 UI를 분리합니다.

컴포넌트 코드를 테스트하기 쉽게 하는 주요 요소는 코드를 적게 사용하는 것입니다. 여러분의 컴포넌트 코드를 보고 물어보세요, 이 부분이 실제로 어떤 방식으로든 document에 연결될 필요가 있습니까? 아니면 분리하여 테스트할 수 있는 별도의 장치/시스템입니까?

프레임워크에 구애받지 않고 UI에서 사용되는지 모르는 단순한 JavaScript "논리" 코드가 많을수록 혼란스럽거나 깨지거나 비용이 많이 드는 방법으로 테스트할 필요가 줄어듭니다. 또한 이 코드는 이동성이 뛰어나 작업자 또는 서버로 이동할 수 있으며 UI 코드는 더 적기 때문에 프레임워크 전체에서 이동성이 뛰어납니다.

UI 컴포넌트를 앱 위젯에서 분리합니다.

UI 코드를 테스트하기 어려운 또 다른 점은 컴포넌트가 서로 매우 다르다는 것입니다. 예를 들어, 앱의 대시보드에 대한 모든 세부 정보를 포함하는 "AppDashboard" 컴포넌트와 앱의 여러 위치에 나타나는 범

용 재사용 위젯인 "DatePicker" 컴포넌트가 있을 수 있습니다.

DatePicker는 **UI 빌딩 블록**으로, 여러 상황에서 UI로 구성할 수 있지만 환경에서 많은 것을 요구하지 않습니다. 사용자 자신의 앱 데이터에 국한된 것은 아닙니다.

반면 AppDashboard는 **앱 위젯**입니다. 응용 프로그램 전체에서 재사용되는 경우가 많지 않을 수 있으며, 한 번만 나타날 수도 있습니다. 따라서 많은 매개 변수가 필요하지 않지만, 앱의 목적과 관련된 데이터와 같은 환경으로부터 많은 정보가 필요합니다.

UI 빌딩 블록 테스팅

UI 빌딩 블록은 가능한 파라매트릭(또는 React에서의 "prop-based") 형태이어야 합니다. 그것들은 너무 많은 컨텍스를 끌어나면 안되며(전역, Redux, useContext), 컴포넌트마다 환경 설정에 너무 많은 것 조건들을 요구해서도 안됩니다.

> [역주] 파라매트릭(parametric) : 하나 또는 여러 개의 매개 변수를 입력으로 출력을 나타내는 형태

파라메트릭 UI 구성 요소를 테스트하는 현명한 방법은 환경을 한 번만 설정하고(예: 브라우저 및 필요한 모든 환경) 환경을 재설정하지 않고 여러 테스트를 실행있게 하는 것입니다.

이 내용의 좋은 예로는 브라우저 공급업체가 상호 운용성을 테스트하기 위해 사용하는 포괄적인 테스트 세트인 웹 플랫폼 테스트가 있습니다. 대부분의 경우 브라우저와 테스트 서버는 한 번 설정되며 테스트마다 새 환경을 설정하지 않고 다시 사용할 수 있습니다.

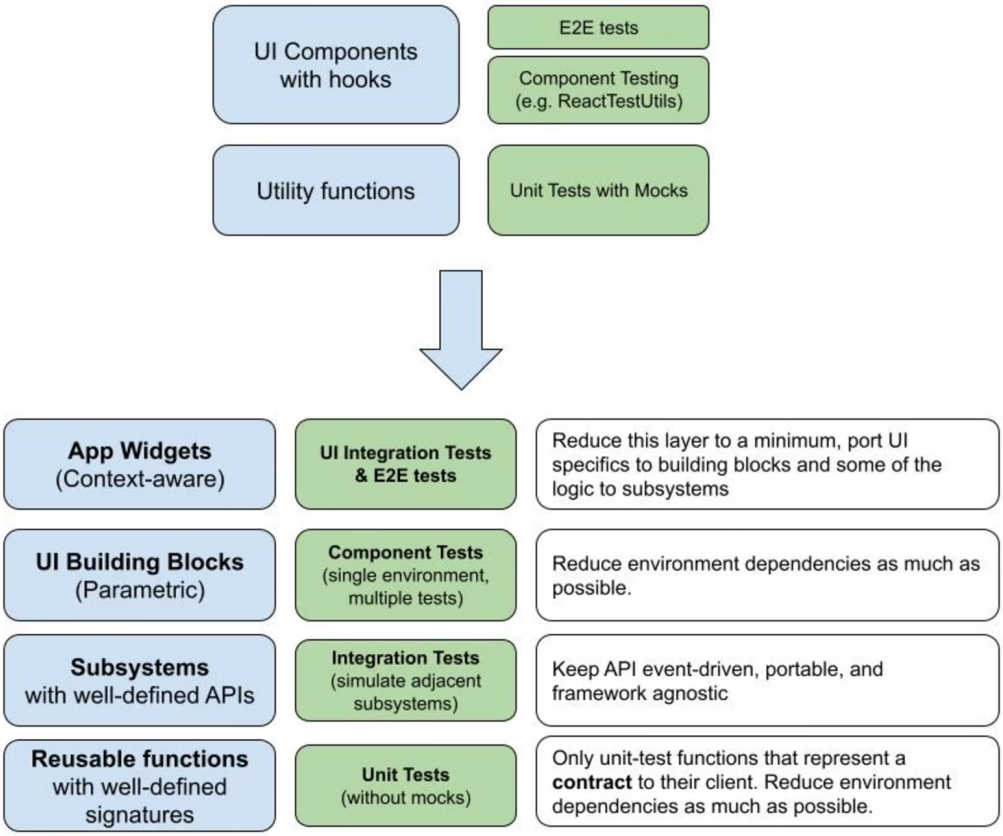
앱 위젯 테스팅

앱 위젯은 **매개 변수**가 아니라 **상황**에 따라 다릅니다. 일반적으로 환경에 많은 것이 필요하고 여러 시나리오에서 운영해야 하지만, 이러한 시나리오를 다르게 만드는 것은 대개 데이터나 사용자 상호 작용에 있습니다.

UI 빌딩 블록을 테스트하는 것과 같은 방식으로 앱 위젯을 테스트하는 것은 매력적입니다. 즉, 모든 다른 후크를 만족시키는 가짜 환경을 만들고 그들이 무엇을 만드는지 볼 수 있습니다. 그러나 이러한 환경은 취약하고 앱이 진화함에 따라 지속적으로 변경되는 경향이 있으며 이러한 테스트는 오래되어 위젯의 기능을 부정확하게 볼 수 있습니다.

상황에 맞는 컴포넌트를 테스트하는 가장 신뢰할 수 있는 방법은 사용자가 보는 앱이라는 실제 맥락 내에 있습니다. 이러한 앱 위젯을 UI 통합 테스트 또는 e2e 테스트로 테스트하되, UI 또는 유틸리티의 다른 부분을 모킹하여 단위 테스트를 수행할 필요는 없습니다.

테스트 가능한 UI 컨닝 페이퍼



(Large Preview)

요약

프런트엔드 테스트는 복잡한데, 이는 종종 UI 코드가 관심사 분리 측면에서 부족하기 때문입니다. 비즈니스 로직 상태머신은 프레임워크별 뷰 코드에 얽히고 설켜 있으며 맥락을-따르는 앱 위젯은 분리된 매개 변수의 UI 빌딩 블록에 얽혀 있습니다. 모든 것이 얽혀 있을 때 신뢰할 수 있는 유일한 테스트 방법은 깨지고 비용이 많이 드는 e2e 테스트에서 "모든 것"을 테스트하는 것입니다.

이 문제를 관리하려면 특정 프로세스와 툴이 아닌 아키텍처에 의존해야 합니다.

- 일부 비즈니스 로직 흐름을 뷰에 구애받지 않는 코드(예: 상태머신)로 변환합니다.
- **앱 위젯에서 빌딩 블록을 분리**하여 다르게 테스트합니다.
- 프런트엔드의 다른 부분이 아니라 **백엔드와 서브시스템을 모킹** 하십시오.
- 시스템 시그니처 및 규약에 대해 생각하고 또 다시 생각해 보십시오.
- **테스트 코드를 경외심을 담아 대하세요.** 그것은 여러분의 코드의 중요한 부분이지, 나중에 생각하는 것이 아닙니다.

프런트엔드 및 서브시스템 간, 그리고 서로 다른 전략 간에 적절한 균형을 이루는 것이 소프트웨어 아키텍처의 기술입니다. 그것을 올바르게 하는 것은 어렵고 경험이 필요합니다. 이런 경험을 얻는 가장 좋은 방법은 노력하고 배우는 것입니다. 이 글이 학습에 조금이나마 도움이 되었으면 좋겠습니다.

> 기술적인 측면에서 검토해주신 벤자민 그린바움과 예호나탄 다니브에게 감사드립니다.