# Front-end 테스트 개요

## TL;DR

프론트엔드 코드는 사용자에 따라 다양한 환경(브라우저, 기기, 운영체제 등)에서 실행되기 때문에 테스트할 때 많은 변수들을 고려해야 합니다.

본 문서에서는 프론트엔드 자바스크립트 테스트에 사용되는 다양한 도구들과 사용법을 소개하고, 프로젝트 상황에 맞는 최적의 도구를 선택할 수 있도록 가이드를 제시합니다.



# Front-end 의 테스트는 언제 하는가?

예 :

- DB에 데이터를 입력하는 API를 개발 -> API 호출 -> DB값 검증
- 디자인 시안에 맞게 HTML/CSS를 작성 -> 브라우저에서 실제 렌더링된 결과를 확인
- 새로운 기능을 추가하기 위해 기존 모듈을 리팩토링 -> 영향을 받는 다른 모듈의 실행 결과를 확인
- 버그를 수정하기 위해 기존 함수를 수정 -> 버그가 수정 확인 & 영향을 받는 다른 모듈의 실행 결과를 확인
- 개발 환경에서 테스트된 어플리케이션을 리얼 환경에 배포 -> 배포 과정에서 발생한 문제가 없는지 확인

# Front-end 요소 별 테스트 전략

| 요소 | | 테스트 전략 |
|---|---|---|
| **시각적(visual) 표현** | ▪ 화면에 표시되는 비주얼 요소를 디자인 요구사항에 맞게 구현<br>▪ 레이아웃 / 색상 / 폰트 / 이미지 / 애니메이션 등<br>▪ 주로 HTML(DOM) / CSS 에 의해 결정 | • 실제 화면을 픽셀 단위로 테스트 -> 눈으로 직접 확인 or 자동 스크린샷 테스트<br>• HTML 구조를 테스트 -> HTML 구조를 직접 입력 or 스냅샷 테스트<br>• ~~특정 DOM 요소의 상태만 테스트 (버튼의 상태 / 텍스트) -> 시각적 테스트 아님~~ |

| | | |
|---|---|---|
| **사용자 입력 처리** | ▪ 사용자에 의한 마우스/키보드 입력 등을 요구사항에 맞게 처리<br>▪ 주로 DOM에 바인딩된 이벤트 핸들러에 의해 처리 | • 자바스크립트 API를 사용한 이벤트 시뮬레이션<br>• 라이브러리(jquery, React)를 이용한 이벤트 시뮬레이션<br>• E2E 도구를 이용한 이벤트 시뮬레이션 |
| **어플리케이션 상태 관리** | ▪ 사용자 입력 등에 의해 변경되는 어플리케이션의 상태를 관리<br>▪ Routing / 팝업 표시-숨김 / 읽기-편집 모드 변경 / 에러 메시지 표시<br>▪ 주로 순수 자바스크립트에 의해 처리 | • 어플리케이션의 상태를 관리하는 레이어만 분리해서 테스트 -> 단위 테스트<br>• 상태와 바인딩 된 DOM 요소의 상태를 테스트 -> 통합 테스트 |
| **서버와의 통신** | ▪ REST API / Socket 등으로 서버와 통신하며 어플리케이션 상태를 동기화<br>▪ 주로 브라우저 API 혹은 라이브러리를 사용해서 비동기 로직을 수행 | • 실제 API 서버를 이용 -> 통합/E2E 테스트<br>• Ajax 통신 모듈을 Mocking / 가상 API 서버를 구축 -> 단위/통합 테스트<br>• 서비스 레이어를 분리해서 Mocking - 단위/통합 테스트 |

# 테스트 종류

## 단위 테스트

설명

- **작은 단위(주로 모듈 단위)를 전체 애플리케이션에서 떼어 내어 분리된 환경에서 테스트하는 것을 말합니다.**

장점

- **세밀한 테스트와 빠른 실행**
  분리된 상태의 테스트이기 때문에 하나의 모듈이나 클래스에 대해 세밀한 부분까지 테스트할 수 있고 더 넓은 범위에서 테스트할 때보다 훨씬 빠르게 실행할 수 있습니다.

단점

- **모의객체 사용 ↑, 각 모듈간의 상호작용 테스트 불가**
  의존성이 있는 모듈을 제어하기 위해 필연적으로 모의 객체(Mocking)을 사용할 수밖에 없으며,  경우 각 모듈이 실제로 잘 연결되어 상호 작용하는지에 대해서는 검증하지 못합니다.
- **작은 리팩토링에도 영향을 받음**
  각 모듈의 사소한 API 변경에도 영향을 받기 때문에 작은 단위의 리팩토링에도 쉽게 깨지는 문제가 있습니다.

## 통합 테스트

설명

- **통합 테스트는 단위 테스트보다 좀 더 넓은 범위의 테스트를 말하며 보통 두 개 이상의 모듈이 실제로 연결된 상태를 테스트합니다.**

장점

- **모의 객체 사용 ↓, 모듈간 상호 작용 테스트 가능**
  여러 개의 모듈이 동시에 상호 작용하는 것을 테스트하기 때문에 단위 테스트에 비해 모의 객체의 사용이 적으며, 모듈 간의 연결에서 발생하는 에러를 검증할 수 있습니다.
- **비교적 작은 리팩토링엔 영향을 받지 않음**
  비교적 넓은 범위에서의 API 변경에만 영향을 받기 때문에 단위 테스트와 비교해 리팩토링을 할 때 쉽게 깨지지 않는 장점이 있습니다.

단점

- **복잡한 코드의 테스트가 번거롭고, 중복 테스트 발생 가능**
  단일 모듈이 복잡한 알고리즘이나 분기문을 갖고 있을 때 단위 테스트에 비해 테스트가 번거롭고, 테스트 중복이 발생할 확률이 높다는 단점이 있습니다.

## E2E 테스트

설명

- **단위 테스트나 통합 테스트는 모두 내부 구조를 알고 있는 개발자의 관점에서 제품 일부분만을 선별해서 테스트하는 방식입니다.**
- **E2E 테스트는 이와 다르게 실제 사용자의 관점에서 테스트를 진행하며, 그런 의미에서 기능(Functional) 테스트 혹은 UI(User Interface) 테스트라고 불리기도 합니다.**

장점

- **사용자와 유사한 환경에서 테스트, 실제 상황에서 발생가능한 에러 확인 가능**
  E2E 테스트는 사용자의 실행 환경과 거의 동일한 환경에서 테스트를 진행하기 때문에 실제 상황에서 발생할 수 있는 에러를 사전에 발견할 수 있다는 장점이 있습니다.
- **브라우저 직접 조작과 관련된 테스트 가능**
  특히 브라우저를 외부에서 직접 제어할 수 있어 자바스크립트의 API만으로는 제어할 수 없는 행위(브라우저 크기 변경, 실제 키보드 입력 등)를 테스트할 수도 있습니다.
- **리팩토링에 영향받지 않음**
  테스트 코드가 실제 코드 내부 구조에 영향을 받지 않기 때문에 큰 범위의 리팩토링에도 깨지지 않으며, 이를 통해 개발자들이 좀 더 자신감 있게 코드를 개선할 수 있도록 도와줍니다.

단점

- **상대적으로 테스트 속도가 느리며, 테스트 작성 시 고려해야 할 부분이 많음**
  단위 테스트나 통합 테스트에 비해 테스트의 실행 속도가 느리기 때문에 개발 단계에서 빠른 피드백을 받기가 어려우며, 세부 모듈들이 갖는 다양한 상황들의 조합을 고려해야 하기 때문에 테스트를 작성하기가 쉽지 않다는 단점이 있습니다.
- **작은 단위로 테스트를 나눌 수 없어, 테스트 코드 사이에 중복 발생 가능**
  또한 큰 단위의 기능을 작은 기능으로 나누어 테스트할 수가 없기 때문에 필연적으로 테스트 사이에 중복이 발생할 수밖에 없습니다.
- **통제되지 않는 환경에서의 테스트로 테스트결과를 신뢰하기 어려움**
  통제된 샌드박스 환경에서의 테스트가 아니기 때문에 테스트 실행 환경의 예상하지 못한 문제들(네트워크 오류, 프로세스 대기로 인한 타임아웃 등)로 인해 테스트가 가끔 실패하는 일이 발생하며, 이 때문에 테스트를 100% 신뢰할 수 없는 문제가 발생하기도 합니다.

# 테스트 환경 종류

## 브라우저

- 실제 브라우저 환경에서 테스트 코드를 실행 (Karma + Jasmine)
- 실제 브라우저를 실행해야 하기 때문에 번거로움 (Headless 브라우저 사용)
- 테스트파일 별로 별도의 브라우저에서 테스트 하기가 어려움 (속도 문제)
- 빈 웹페이지를 만들고 모든 스크립트 파일 및 CSS 등을 include 해서 테스트 (번들 과정 필요)
- 브라우저의 모든 API를 사용해서 테스트 가능
- 브라우저 호환성 테스트 가능
  - 개발시 : 빠른 Feedback을 위해 Headless 브라우저를 사용
  - 빌드시 : CI 서버 및 Webdriver와 연동하여 여러개의 브라우저에서 테스트

## Node.js

- Node.js 환경에서 테스트 코드를 실행 (Mocha, Jest 등)
- 브라우저에 비해서 가볍기 때문에 실행속도가 빠름
- 개별 테스트 파일을 별도의 프로세스에서 실행할 수 없음 (병렬 실행 가능)
- 브라우저 API 대신 JSDom 을 이용해서 테스트
- 실제 렌더링을 해 주지 않으므로, 렌더링 관련 테스트 불가능
- 브라우저 호환성 테스트 불가능

# 테스트 도구 종류

## Testing Frameworks(정의 된 규약을 가지고 테스트 코드를 작성할 수 있게 도움)

- 사용자가 테스트 코드를 작성할 수 있는 기반을 제공해주는 자바스크립트 도구입니다.
- 프레임워크가 제공하는 함수들을 사용해서 테스트 코드를 작성하면, 프레임워크가 테스트 코드를 자동으로 실행한 후 성공 및 실패에 대한 결과를 반환해줍니다.

예 :

Mocha, Jasime(Jest), AVA

## Test Runners(작성한 테스트 코드가 실제로 동작 시킴)

- 파일을 읽어들여 작성한 코드를 실행하고, 그 결과를 특정한 형식으로 출력해줍니다.
- 테스트의 수행 결과는 리포터(Reporter)를 지정해서 원하는 형태로 출력할 수 있습니다.
- 부가적으로 테스트 코드나 소스 코드가 변경된 경우 영향을 받는 테스트를 자동으로 재실행해주는 왓쳐(Watcher) 등의 기능도 제공합니다.
- 이 중 Node.js 기반의 테스트 러너들은 굳이 러너의 실행 환경과 코드의 실행 환경을 구분할 필요가 없기 때문에 대부분 테스트 프레임워크와 통합된 형태로 제공됩니다.

예 :

Karma(브라우저 환경 테스트), Jest, Mocha, AVA(Node.js 환경 테스트)

## Assertion(단언) Libraries(테스트 코드 작성 시, 성공/실패 조건을 정의하기 위한 함수를 제공)

- 테스트 코드는 주로 테스트를 위한 초기화와 단언으로 이루어지며, 단언은 개별 테스트가 통과하기 위한 조건을 명확하게 기술하기 위해 사용됩니다.
- 보통은 테스트 프레임워크에서 다양한 방식의 단언 API를 기본 제공하고 있으며, Mocha의 경우에만 Chai와 같은 별도의 단언 라이브러리를 사용하도록 권장하고 있습니다.
- 초기의 단언 라이브러리들은 JUnit과 유사한 방식의 API를 많이 따랐지만, 최근에 가장 많이 사용되는 Chai, Jasmine 등에서는 좀 더 자연어에 가까운 BDD(Behavior-driven development 방식의 API가 사용됩니다.
- 대부분의 단언 라이브러리들은 사용자들이 필요에 따라 자신만의 단언을 추가해서 사용할 수 있는 플러그인 확장 기능을 제공합니다.

예 :

대부분 테스트 프레임워크에 포함된 형태로 사용 (Jasmine, Jest, AVA)

Mocha의 경우 별도의 라이브러리인 Chai를 사용

## 테스트 더블(test double) 라이브러리(테스트에 필요한 추가적인 요소를 제공 (mocks, stubs, fake servers, etc...))

- 실제 객체 대신 테스트를 위해 동작하는 객체를 말하며, 주로 분리된(isloated) 단위 테스트를 위해 외부 의존성을 임의로 주입하기 위해서 사용합니다.
- 스파이(spy), 스텁(stub), 목(mock) 등의 다양한 테스트 더블을 사용할 수 있으며, 이들을 쉽게 만들 수 있도록 도와주는 라이브러리를 테스트 더블 라이브러리라고 합니다.
- 단언과 마찬가지로 테스트 더블을 위한 함수들도 테스트 프레임워크에서 기본 제공되는 경우가 대부분이며, Mocha의 경우에만 Sinon.JS 등의 별도 라이브러리를 사용하도록 권장하고 있습니다.

예 : Sinon, Jasmine, Jest

# 테스트 도구

## 테스팅 도구 만족도 변화 (2016 ~ 2019)

https://2019.stateofjs.com/testing/

## 대중적으로 사용하는 조합

- 브라우저 : `Karma + Jasmine`
- Node.js : `Jest` or `Mocha + Chai + Sinon.js`

## Jest

설명

- 페이스북에서 만든 React를 위해 만든 오픈소스 테스트 프레임워크
- 최근 프론트엔드 개발에서 가장 활발하게 사용
- 페이스북에서 만든 자바스크립트 테스팅 라이브러리. 오픈소스(MIT)
- 현재 페이스북 내의 모든 자바스크립트 테스트에 사용됨
- 테스트 러너 / 구조화 / 단언 / 테스트 더블 등의 기능을 모두 포함
- Node 환경에서 JSDom을 이용해 테스트 (브라우저 테스트 불가)
- 테스트를 병렬로 수행해서 속도를 높임

장점

- 쉬운 설치 및 실행
- 쉬운 커버리지 측정
- jsdom 내장
- 스냅샷 테스트
- 테스트 파일 필터링
- 샌드박스 병렬 테스트
- Jasmine 과 호환되는 단언 API 형식 (처음엔 Jasmine 사용 -> 현재 자체 구현)
- Zero Configuration : 설정 없이 간단하게 실행할 수 있음
- E2E 테스트 도구와 함께 사용하면 단점 보완 가능

## Jasmine

- BDD 스타일의 단언 API를 사용하는 통합 테스크 프레임워크

장점

- 비동기 코드 테스트 지원
- 모든 환경에서 사용가능 (Node.js & 브라우저)
- 별도 라이브러리 설치 필요 X
- 테스트 명세를 그룹화 할 수 있다 (목적에 맞게 묶어 관리 가능)
- 비교적 빠른 테스트

## Mocha

- 다른 모의 라이브러리를 함께 사용할 수 있음
    - 즉, 단언 라이브러리를 따로 설정해주어야 함.

장점

- 유연하다 (어느 라이브러리와도 함께 사용될 수 있다.)
- 간단하고 명료한 API
- 비동기 테스트 코드 지원
- 테스트 실행이 빠름
- 보다 높은 사용성

단점

- 자동모의나 스냅샷 테스트 미지원
- 설치(환경 구성이)가 까다롭다.
- 비교적 비동기 테스트 지원이 적다

## Karma

- 실제 브라우저에서 테스트를 실행할 수 있도록 도와줌

실행방식

- 커맨드 라인에서 Karma 실행시,  -> `HTML`  ->
- 브라우저 직접 실행 후 접속하면 로드된 코드가 실행되어 실행 결과가 브라우저 콘솔에 출력됨
- Karma는 해당 정보를 지정된 리포터를 통해 정리한 후, 커맨드 라인에 결과 보여줌

- 테스트 커버리지 `lstanbul` lib로 설치해 측정 가능
    - `coverage` 폴더 안 브라우저 런처 별 폴더 생성 `index.html`을 열어 측정 결과 확인
- 크로스 브라우징 테스트 가능
    - 로컬 PC에서도 다양한 브라우저에 대한 테스트 동시 실행 가능

장점

- 오래된 브라우저나 브라우저 간 호환성을 지원해야할 때 좋음
- 웹 드라이버 등으로 원격 테스트 가능
- 여러 브라우저 동시 테스트 가능 (`Selenium`등 도구 설치 및 간단한 코드 설정 필요)
- 모든 브라우저 지원

단점

- 브라우저에 너무 특화되어있음

# 좋은 테스트 코드 작성을 위한 전략

## 1. 실행 속도가 빨라야 한다.

- 빠른 피드백 -> 개발 속도를 빠르게 해 줌
- 너무 느리면 테스트를 자주 실행하지 않게 됨

## 2. 내부 구현 변경 시 실패하지 않아야 한다.

- 리팩토링할 때 테스트가 깨진다면? -> 오히려 코드 개선을 방해
- 구현 종속적인 테스트를 작성하지 않는다
    - 내부 구현을 모른채 테스트를 작성(BlackBox 테스팅)
    - 인터페이스를 기준으로 테스트를 작성한다.
- 자주 변하는 로직과 변하지 않는 로직을 구분 (ex: 모델과 뷰를 분리)

## 3. 버그를 검출할 수 있어야 한다.

- 소스 코드에 버그가 있어도 검출하지 못한다면 잘못된 테스트
- 테스트가 기대하는 결과를 구체적으로 명시하지 않으면 버그를 검출할 수 없음
- 테스트 더블의 사용을 최소화한다. -> 과하게 사용하면 연결 과정에서의 버그를 검출할 수 없음

## 4. 테스트의 결과가 안정적이어야 한다.

- 특정 환경에서만 실패하거나, 간헐적으로 결과가 달라지는 테스트는 신뢰할 수가 없음
- 외부 환경의 영향을 최소화해서 동일한 결과를 최대한 보장할 수 있어야 함
- 현재 시간, 네트워크 상태, 외부 프로세스 등은 모의 객체나 별도의 도구를 사용해서 직접 조작할 수 있어야 함.

## 5. 의도가 명확히 드러나야 한다.

- 가독성 : ~~"기계가 읽기 좋은 코드"~~ -> "사람이 읽기 좋은 코드"
- 테스트 코드도 실제 코드와 동일한 기준으로 품질 관리를 해야 함
- 테스트 코드를 보고 한 눈에 어떤 내용을 테스트하는지를 파악할 수 있어야 함.
- 공통 로직, Fixture, Mock 등은 분리해서 관리

# 시각적 회귀 테스트(Visual Regression Test)

| 시각적 요소 테스트 | 기능적 요소 테스트 |
| --- | --- |

| | |
|---|---|
| • 변경사항을 반영하고도 똑같은 UI가 유지되는지를 검사합니다.<br>• PR의 변경사항에서 전후 비교를 한눈에 하기 위함<br>• 의도치 않은 UI 변동을 방지하기 위함<br><br>도구: Cypress 등 | • 단위 테스트는 제어된 입력이 주어졌을 때 UI 코드가 올바른 출력을 반환하는지 확인합니다. |

## Why?

functional vs design 관점

- 많은 사람들이 기능적 회귀 테스트를 할때 이러한 기능의 테스트가 시각적 요소의 테스트까지 보장한다고 착각한다

Visual Changelog 관점

- 여러가지 컴포넌트 라이브러리들이나 디자인시스템에서 체인지 로그들을 보관하거나 비교함으로써 시각적 변화들을 기록할수 있다는 점이 있다. 또 버그의 증거를 찾을수 있다.

e2e 테스트와 비교 관점

- 클래스가 존재하는지 유무를 판단하는것은 해당 레이아웃이 제대로 보여지는지 혹은 깨졌는지 등등을 파악하기가 어려움

## 테스트 도구

CI상에서 연동 가능한 도구들이 있음

- Storybook
- Chromatic
- Cypress

추후 참고 하면 좋을 자료

- Keeping a React Design System consistent
- Visual Regression Testing
- 실용적인 프론트엔드 테스트 전략 (2)
- Visual Regression Testing
- Guide To Visual Regression Testing With Visual Testing Tools

출처

https://ui.toast.com/fe-guide/ko_TEST

https://www.merixstudio.com/blog/mocha-vs-jest/

https://blog.rhostem.com/posts/2020-10-14-beginners-guide-to-testing-react-1#overview_of_testing_react_apps

https://nhnent.dooray.com/share/posts/9jRYF1fxRwiCvwi6VM0VgA

https://storybook.js.org/tutorials/design-systems-for-developers/react/ko/test/

https://ideveloper2.dev/blog/2021-01-24--%EC%8B%9C%EA%B0%81%EC%A0%81-%ED%9A%8C%EA%B7%80-%ED%85%8C%EC%8A%A4%ED%8A%B8-visual-regression-test/