어떻게 프런트엔드 테스트를 작성할 것인가?

실무에 바로 적용하는 프런트엔드 테스트

단위 테스트

단일 함수의 결괏값 또는 단일 컴포넌트(클래스)의 상태(UI)나 행위를 검증한다.

무엇을 검증하는가

- 공통 컴포넌트
    - 다른 컴포넌트와의 상호 작용이 없다.
    - 해당 컴포넌트의 내부 비즈니스 로직을 기능 단위로 나눠 검증한다.

단위 테스트

단일 함수의 결괏값 또는 단일 컴포넌트(클래스)의 상태(UI)나 행위를 검증한다.

무엇을 검증하는가

- 공통 컴포넌트
    - 다른 컴포넌트와의 상호 작용이 없다.
    - 해당 컴포넌트의 내부 비즈니스 로직을 기능 단위로 나눠 검증한다.

단위 테스트 전략

- 별도 로직처리 없이 UI만 그리는 컴포넌트는 검증하지 않는다.
    - 해당 검증은 스토리북과 같은 도구를 통해 검증한다.
- 간단한 로직 처리만 하는 컴포넌트는 상위 컴포넌트의 통합 테스트에서 검증한다.

단위 테스트

단일 함수의 결괏값 또는 단일 컴포넌트(클래스)의 상태(UI)나 행위를 검증한다.

무엇을 검증하는가

- 공통 컴포넌트
    - 다른 컴포넌트와의 상호 작용이 없다.
    - 해당 컴포넌트의 내부 비즈니스 로직을 기능 단위로 나눠 검증한다.

단위 테스트 전략

- 별도 로직처리 없이 UI만 그리는 컴포넌트는 검증하지 않는다.
    - 해당 검증은 스토리북과 같은 도구를 통해 검증한다.
- 간단한 로직 처리만 하는 컴포넌트는 상위 컴포넌트의 통합 테스트에서 검증한다.

한계

- 여러 모듈이 조합 되었을 때 발생하는 이슈는 찾을 수 없다.
- 앱의 전반적인 기능이 비즈니스 요구사항에 맞게 동작 하는지 보장할 수 없다.

통합 테스트

두 개 이상의 모듈의 상호작용 또는 컴포넌트가 조합 되었을 때 비즈니스 로직이 올바른지 검증하는 테스트

무엇을 검증하는가

- **상태**나 **데이터**를 관리하는 컴포넌트
  - 특정 상태(state)를 기준으로 동작하는 컴포넌트 조합
  - API와 함께 상호작용하는 컴포넌트 조합
- 추가로, 단순 UI 렌더링 및 간단한 로직을 실행하는 컴포넌트까지 한 번에 효율적으로 검증 가능

통합 테스트

두 개 이상의 모듈의 상호작용 또는 컴포넌트가 조합 되었을 때 비즈니스 로직이 올바른지 검증하는 테스트

무엇을 검증하는가

- **상태**나 **데이터**를 관리하는 컴포넌트
  - 특정 상태(state)를 기준으로 동작하는 컴포넌트 조합
  - API와 함께 상호작용하는 컴포넌트 조합
- 추가로, 단순 UI 렌더링 및 간단한 로직을 실행하는 컴포넌트까지 한 번에 효율적으로 검증 가능

통합 테스트 전략

- 가능한 한 모킹을 하지 않고 실제와 유사하게 검증한다
- 비즈니스 로직을 처리하는 상태 관리나 API 호출은 상위 컴포넌트로 응집한다
- 변경 가능성을 고려해 여러 도메인의 기능이 조합된 비즈니스 로직은 나눠 작성하는 것이 좋다.

통합 테스트

두 개 이상의 모듈의 상호작용 또는 컴포넌트가 조합 되었을 때 비즈니스 로직이 올바른지 검증하는 테스트

무엇을 검증하는가

- **상태**나 **데이터**를 관리하는 컴포넌트
  - 특정 상태(state)를 기준으로 동작하는 컴포넌트 조합
  - API와 함께 상호작용하는 컴포넌트 조합
- 추가로, 단순 UI 렌더링 및 간단한 로직을 실행하는 컴포넌트까지 한 번에 효율적으로 검증 가능

통합 테스트 전략

- 가능한 한 모킹을 하지 않고 실제와 유사하게 검증한다
- 비즈니스 로직을 처리하는 상태 관리나 API 호출은 상위 컴포넌트로 응집한다
- 변경 가능성을 고려해 여러 도메인의 기능이 조합된 비즈니스 로직은 나눠 작성하는 것이 좋다.

한계

- 전체 워크플로우를 검증하기 어려우며, 만약 검증 하더라도 지나치게 모킹(또는 스텁)에 의존한다.

스냅샷 테스트

UI 컴포넌트의 렌더링 결과나 함수의 결과를 직렬화해 기록하여 이전의 스냅샷과 비교하는 테스트

무엇을 검증하는가

- 스냅샷을 비교해 기존과 다른 변경점을 알아차리고 의도하지 않은 변경을 빠르게 찾는다.

스냅샷 테스트

UI 컴포넌트의 렌더링 결과나 함수의 결과를 직렬화해 기록하여 이전의 스냅샷과 비교하는 테스트

무엇을 검증하는가

- 스냅샷을 비교해 기존과 다른 변경점을 알아차리고 의도하지 않은 변경을 빠르게 찾는다.

한계

- 의도하지 않은 스냅샷 업데이트 가능성 존재한다.
- 실제로 렌더링 하는 것이 아니기 때문에 스타일이나 레이아웃이 제대로 반영되는지 알 수 없고, CSS가 수정되었을 때 변경 사항을 감지할 수 없다.
- 스냅샷 업데이트 과정이 필요하기 때문에 TDD 사이클과는 다르게 작성될 수 있다.

시각적 회귀 테스트

무엇을 검증하는가

- 스냅샷 테스트와 다르게 실제 렌더링된 UI 결과 이미지를 스냅샷으로 저장해 비교하는 테스트
  - 이를 통해 스타일이나 레이아웃 변경 사항까지 모두 감지해 검증한다.
- 스토리북과 같은 컴포넌트 UI 개발 도구를 연동해 좀 더 편리하게 실행할 수 있다.
- Github Actions를 사용해 워크플로우를 자동화하면 UI 리뷰까지 빠르게 피드백 받을 수 있다.

시각적 회귀 테스트

무엇을 검증하는가

- 스냅샷 테스트와 다르게 실제 렌더링된 UI 결과 이미지를 스냅샷으로 저장해 비교하는 테스트
    - 이를 통해 스타일이나 레이아웃 변경 사항까지 모두 감지해 검증한다.
- 스토리북과 같은 컴포넌트 UI 개발 도구를 연동해 좀 더 편리하게 실행할 수 있다.
- Github Actions를 사용해 워크플로우를 자동화하면 UI 리뷰까지 빠르게 피드백 받을 수 있다.

시각적 회귀 테스트 전략

- 실제 UI만 렌더링하는 컴포넌트를 대상으로 실행하는 것이 좋다.
- 크로스 브라우징 또는 스타일 변경으로 UI가 자주 틀어지는 컴포넌트를 검증하자.

시각적 회귀 테스트

무엇을 검증하는가

- 스냅샷 테스트와 다르게 실제 렌더링된 UI 결과 이미지를 스냅샷으로 저장해 비교하는 테스트
  - 이를 통해 스타일이나 레이아웃 변경 사항까지 모두 감지해 검증한다.
- 스토리북과 같은 컴포넌트 UI 개발 도구를 연동해 좀 더 편리하게 실행할 수 있다.
- Github Actions를 사용해 워크플로우를 자동화하면 UI 리뷰까지 빠르게 피드백 받을 수 있다.

시각적 회귀 테스트 전략

- 실제 UI만 렌더링하는 컴포넌트를 대상으로 실행하는 것이 좋다.
- 크로스 브라우징 또는 스타일 변경으로 UI가 자주 틀어지는 컴포넌트를 검증하자.

한계

- 대부분 유료 도구가 많아 비용 부담이 존재하며, 직접 구축할 경우 관리 부담이 존재한다.
- 어떤 이유로 변경 사항이 발생했는지 추론에 시간이 소요된다.
- 실행 시간이 매우 오래 걸리기 때문에 빠른 피드백을 받을 수 없다.
  - CI 연동은 필수이며, 개발 단계에서 도입할 수 없어 TDD 사이클은 불가능하다.

E2E 테스트

실제 앱을 구동해 전체 소프트웨어 시스템 전반의 흐름을 검증한다.

- 개발이 완료되는 시점에 전반적인 기능 테스트가 가능
- 유관 부서의 협력이 필요하며, 도입을 위한 일정 확보 필요

무엇을 검증하는가

- 사용자가 앱을 사용하는 다양한 시나리오를 검증 → FE부터 BE 전반의 상태를 확인

E2E 테스트 전략

- 가능한 한 모킹을 하지 않고 테스트를 실행한다.
    - API 호출이 어렵거나 서드파티 라이브러리 또는 외부 앱을 사용하는 특정 상황에서만 스터빙

E2E 테스트

실제 앱을 구동해 전체 소프트웨어 시스템 전반의 흐름을 검증한다.

- 개발이 완료되는 시점에 전반적인 기능 테스트가 가능
- 유관 부서의 협력이 필요하며, 도입을 위한 일정 확보 필요

무엇을 검증하는가

- 사용자가 앱을 사용하는 다양한 시나리오를 검증 → FE부터 BE 전반의 상태를 확인

E2E 테스트 전략

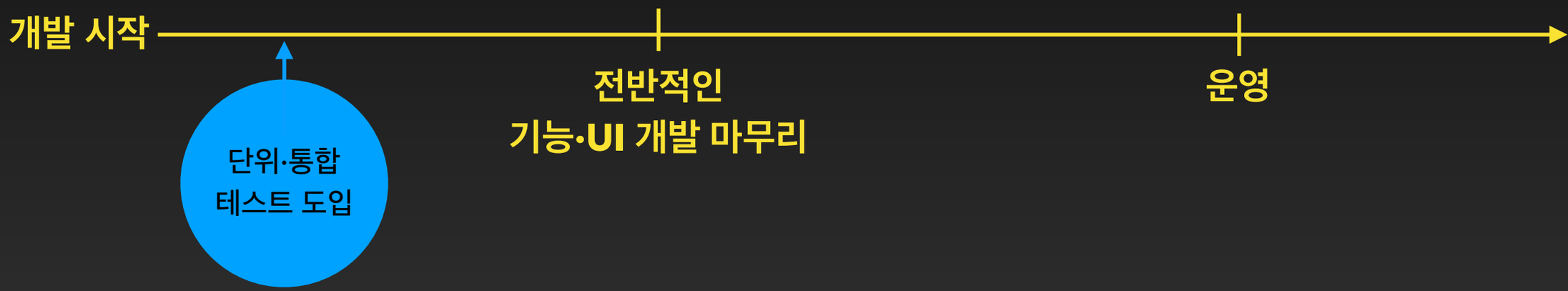
- 가능한 한 모킹을 하지 않고 테스트를 실행한다.
    - API 호출이 어렵거나 서드파티 라이브러리 또는 외부 앱을 사용하는 특정 상황에서만 스터빙

한계

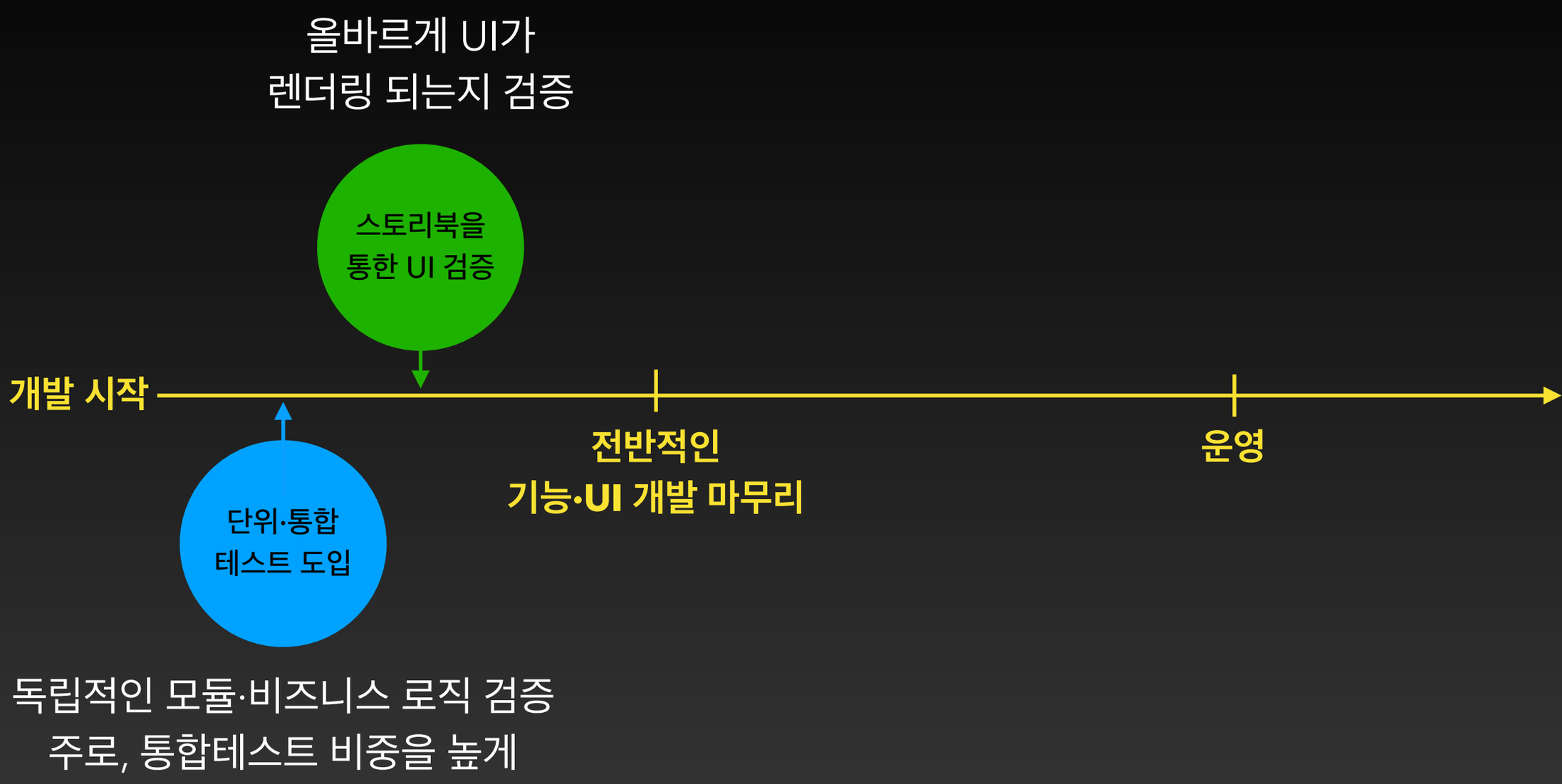
- 실행 시간이 오래 걸려 개발 생산성이 저하될 수 있다.
- 외부 환경 요소로 인해 테스트가 깨질 수 있어 테스트 실행을 위한 관리 비용이 많이 들어간다.
- 디버깅 시간이 오래 걸린다.
    - 테스트에 영향을 미칠 수 있는 범위가 크고, 이슈의 원인을 찾아 수정하는데 많은 시간이 소요됨

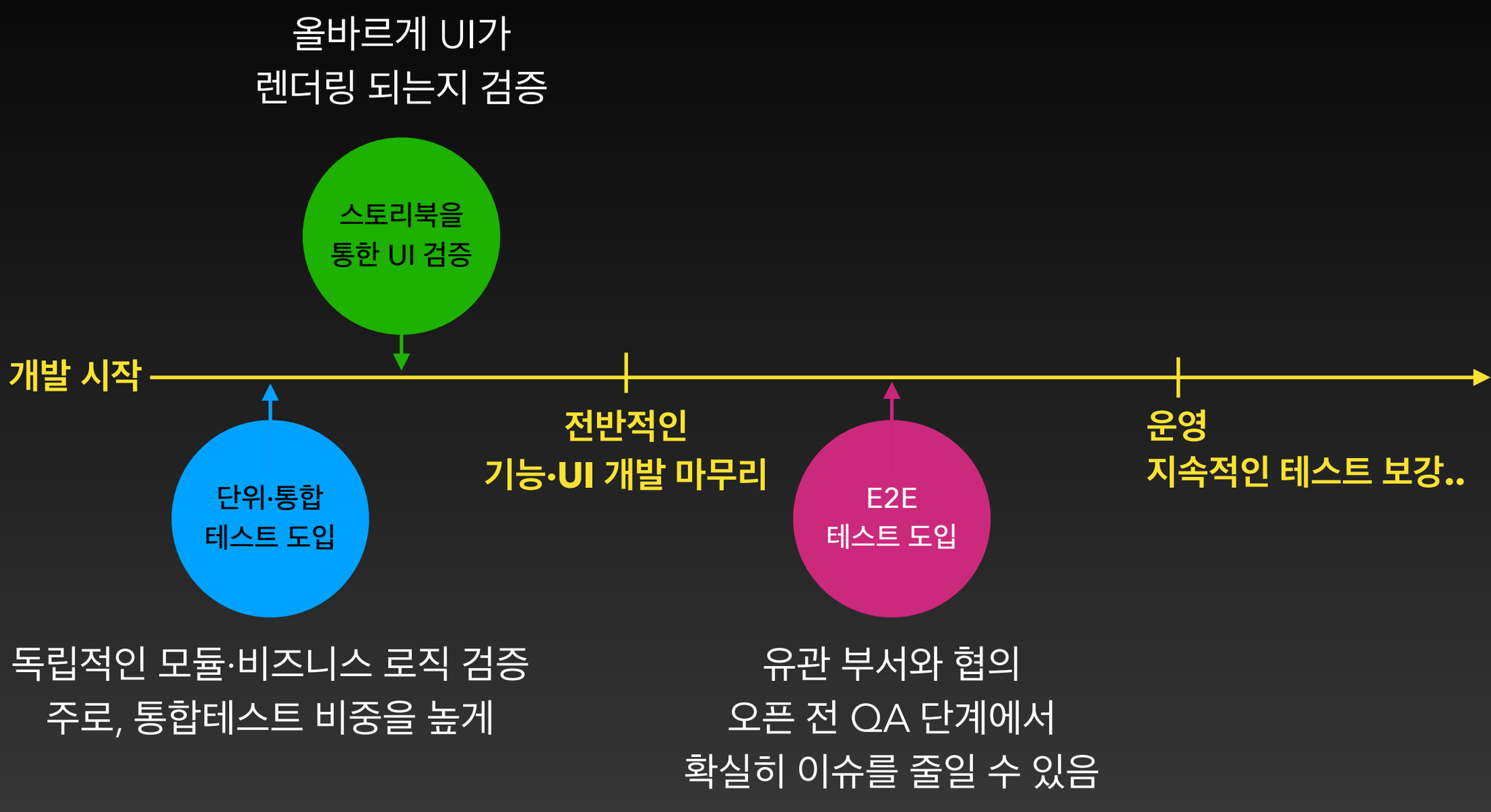
테스트 총 정리

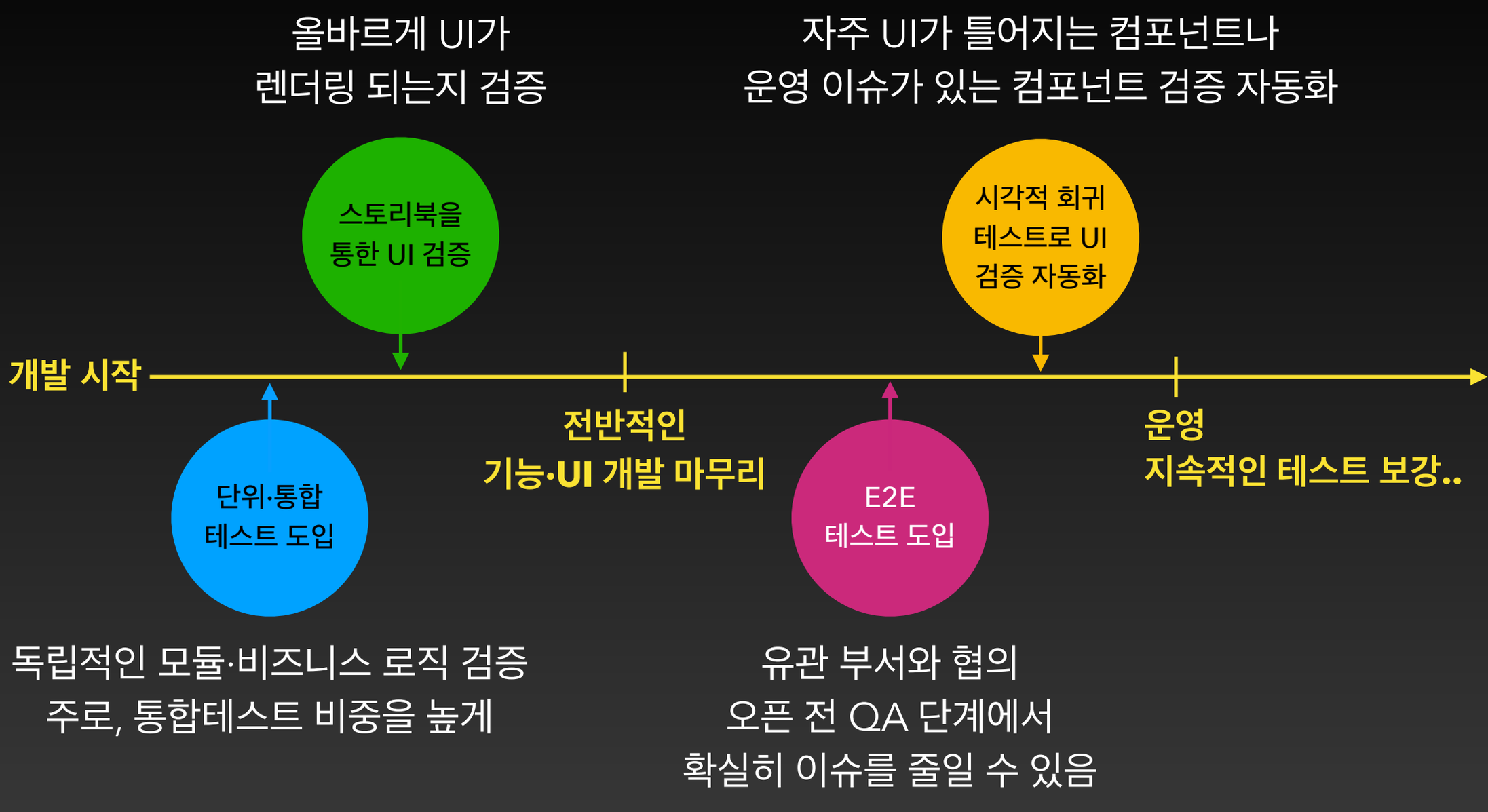
	검증 범위	도입 시점	실행 시간	TDD 적용
단위 테스트	독립적인 모듈 단위	개발 단계	매우 짧음	가능
통합 테스트	일부 모듈이 조합 되었을 때의 비즈니스 로직	개발 단계	짧음/보통	가능
시각적 회귀 테스트	컴포넌트 UI	개발 및 기능 검증이 완료된 상태	매우 오래 걸림	불가능
E2E 테스트	앱의 전반적인 워크플로우	개발 완료 상태 (QA 직전)	매우 오래 걸림	불가능



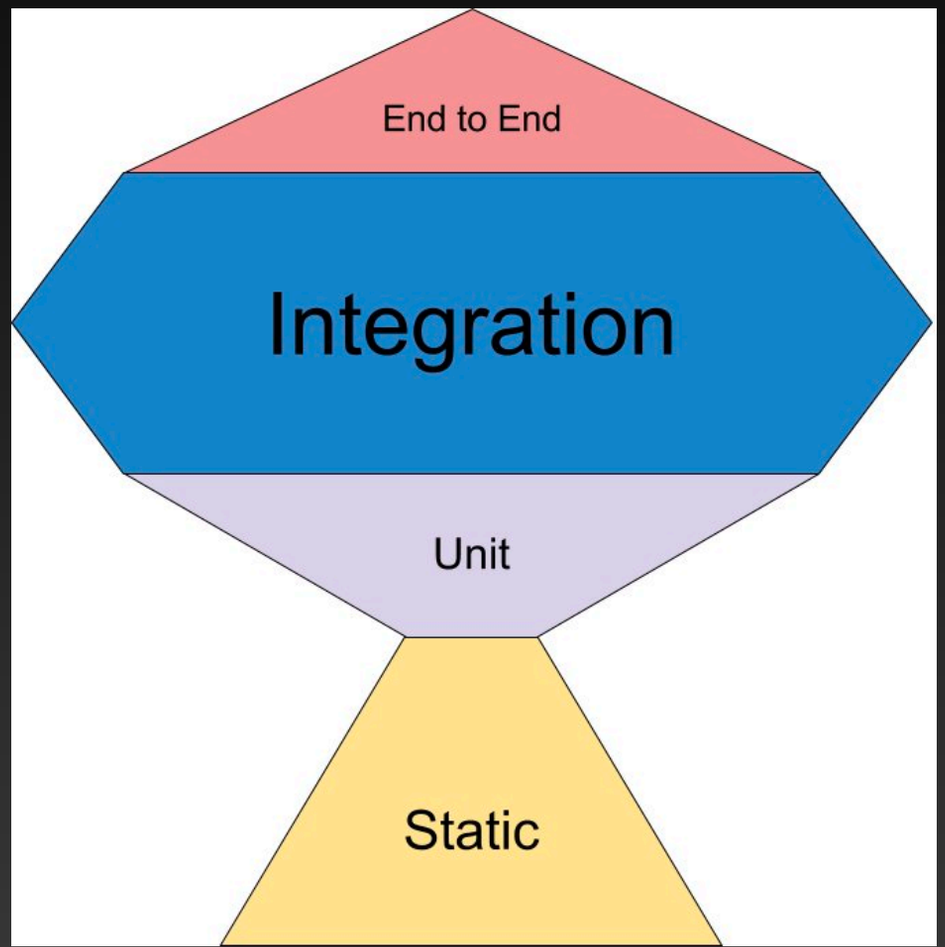
독립적인 모듈·비즈니스 로직 검증
주로, 통합테스트 비중을 높게


올바르게 UI가
렌더링 되는지 검증
스토리북을
통한 UI 검증
개발 시작
전반적인
기능·UI 개발 마무리
운영
단위·통합
테스트 도입
독립적인 모듈·비즈니스 로직 검증
주로, 통합테스트 비중을 높게


올바르게 UI가
렌더링 되는지 검증
스토리북을
통한 UI 검증
개발 시작
단위·통합
테스트 도입
전반적인
기능·UI 개발 마무리
E2E
테스트 도입
운영
지속적인 테스트 보강..
독립적인 모듈·비즈니스 로직 검증
주로, 통합테스트 비중을 높게
유관 부서와 협의
오픈 전 QA 단계에서
확실히 이슈를 줄일 수 있음

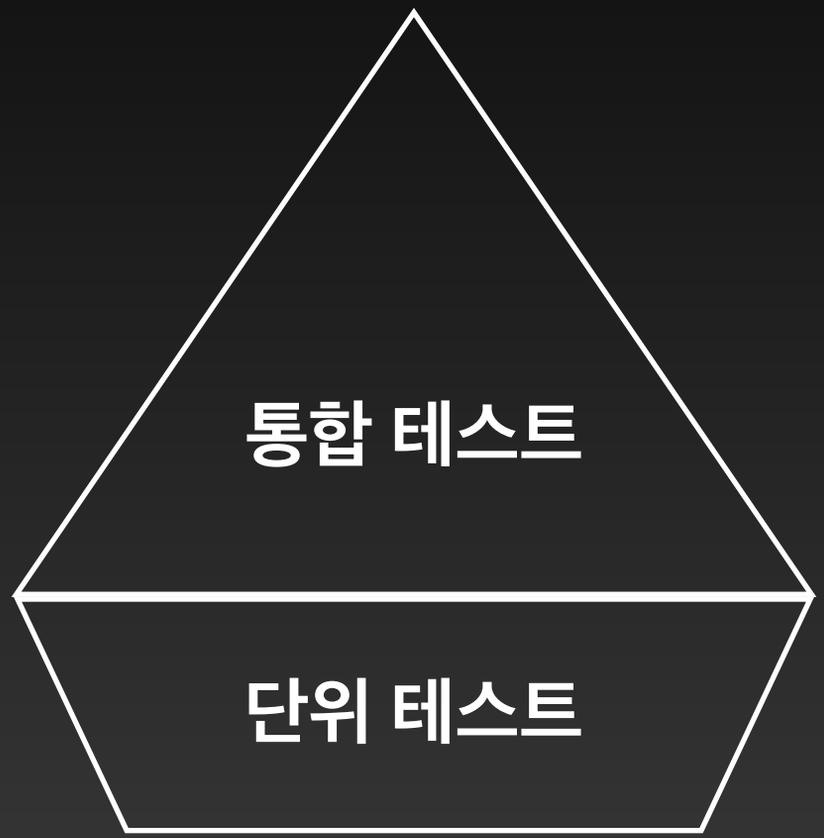

올바르게 UI가
렌더링 되는지 검증
자주 UI가 틀어지는 컴포넌트나
운영 이슈가 있는 컴포넌트 검증 자동화
스토리북을
통한 UI 검증
시각적 회귀
테스트로 UI
검증 자동화
개발 시작
전반적인
기능·UI 개발 마무리
운영
지속적인 테스트 보강..
단위·통합
테스트 도입
E2E
테스트 도입
독립적인 모듈·비즈니스 로직 검증
주로, 통합테스트 비중을 높게
유관 부서와 협의
오픈 전 QA 단계에서
확실히 이슈를 줄일 수 있음

테스트 트로피



출처: https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications

테스트 전략 - 1



소규모 앱에서 데이터를 조회하는 정도라면,
✔ 단위·통합 테스트로 기능 검증
✔ 스토리북으로 UI 검증

테스트 전략 - 2

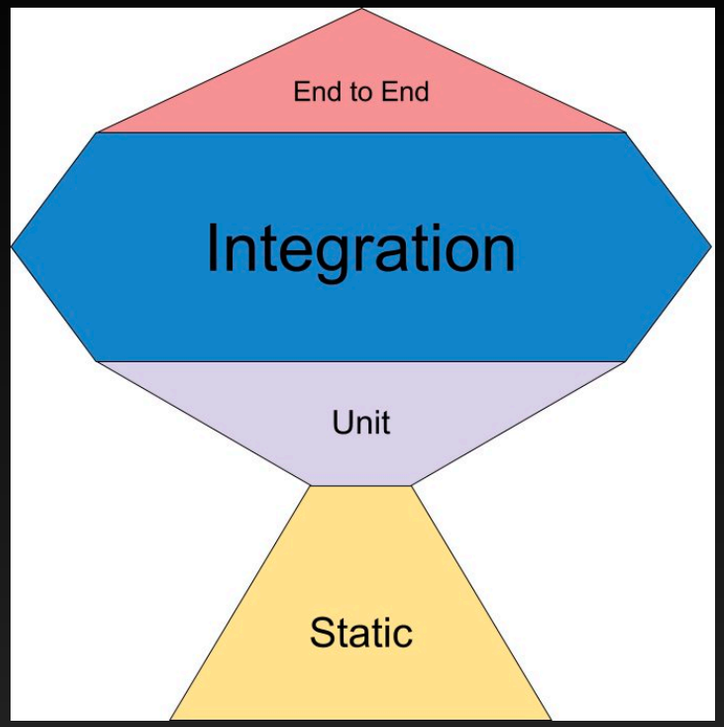
페이지 단위로 단순한 기능만 있다면
✔ 전체 페이지를 기준으로 단위 테스트와 일부 통합 테스트만 진행하면 됨

테스트 전략 - 3

거대한 레거시 앱에 테스트를 도입한다면

✔ 모듈 단위 단위 테스트부터 도입

✔ 중요 워크플로우에 대해 E2E 테스트를 도입해 안정성 향상


End to End
Integration
Unit
Static


통합 테스트
단위 테스트


통합 테스트
단위 테스트


E2E 테스트
단위 테스트

결국 테스트를 도입하는 가장 중요한 목표는

사용자에게 안정적인 서비스를 제공하는 것

지금까지 강의 들어주셔서 감사합니다! 👋