겨울방학 스터디
3회차

College of Art & Technology
Chung-Ang University

Complex Intelligent Systems Laboratory

https://cislab.cau.ac.kr

테스트 없는 개발의 문제점

"코드 한 줄 수정했는데 서비스가 터졌다"

흔한 상황들

- 새 기능 추가 후 기존 로그인이 안 됨
- CSS 수정 후 다른 페이지 레이아웃 깨짐
- API 응답 구조 변경 후 프론트 전체 에러

수동 테스트만으로는 모든 케이스를 커버할 수 없음

우리가 하고 있는 테스트

일반적인 개발 흐름

- 코드 작성
- 브라우저 열기
- 버튼 클릭
- console.log 확인
- "오 된다!" 또는 "왜 안 되지..."

문제점

- Chrome에서 됐는데 Safari에서 안 됨
- 코드 수정할 때마다 전체 수동 테스트?

해결책

이 과정을 코드로 자동화하자!

→ E2E 테스트 도구: Playwright

테스트의 장점

01

검증 보장

리팩토링, 기능 추가 시
기존 기능을 깨뜨리지 않음을 보장

02

문서화

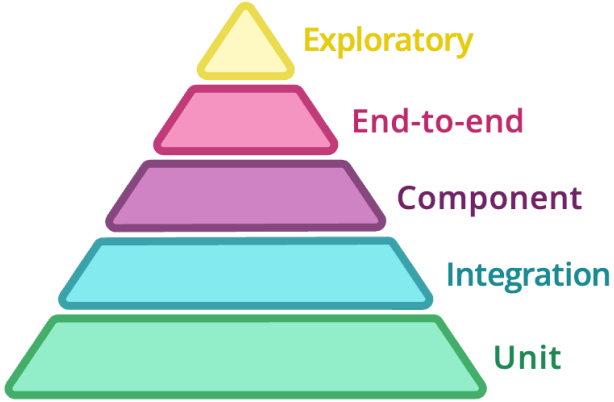
테스트 코드가 기능 명세서 역할

03

협업 효율

PR 리뷰 시 검증된 코드 제출

테스트 피라미드



위로 갈수록

신뢰도 높음 / 유지비용 높음 / 속도 느림

아래로 갈수록

비용 낮음 / 속도 빠름 / 작성 용이

효율적인 전략: 하위 단계에 집중, E2E는 핵심만

단위 테스트(Unit Test)

정의

함수나 클래스 같은 코드의 최소 단위를 독립적으로 검증

특징

- 빠른 실행 속도
- 낮은 작성 비용
- Mock/Stub으로 외부 의존성 격리

도구: Jest, Vitest, JUnit

// 코드 예시

```
const sum = (a, b) => a + b;

test('1 + 2 = 3', () => {
expect(sum(1, 2)).toBe(3);
});
```

통합 테스트 (Integration Test)

정의

여러 모듈이나 컴포넌트가 함께 잘 동작하는지 검증

특징

- 외부 의존성(DB, API) 포함
- 단위 테스트보다 느리지만 신뢰도 높음
- 시스템 결합부의 문제 조기 발견

도구: Jest, Mocha, Supertest

```
// 예시: API + DB 연동
test('유저 생성 API', async () => {
const res = await request(app)
.post('/users')
.send({ name: 'test' });
expect(res.status).toBe(201);
});
```

E2E 테스트 (End-to-End Test)

정의

실제 사용자 시나리오를 시뮬레이션하여 시스템 전체가 올바르게 동작하는지 검증

특징

- 실제 브라우저 환경에서 테스트
- 프론트 + 백엔드 + DB 모두 검증
- 가장 높은 신뢰도

도구: Playwright, Cypress, Selenium

// E2E 시나리오 예시

- 로그인 페이지 접속
- 이메일/비밀번호 입력
- 로그인 버튼 클릭
- 대시보드 페이지 이동 확인

→ 전체 흐름이 정상 동작하는지 검증

테스트 종류 비교

구분	목적	속도	비용	도구
정적 분석	코드 문법/타입 오류	매우 빠름	낮음	ESLint, Puff
단위	함수/모듈 단위 검증	빠름	낮음	Jest, Vitest, Pytest
통합	모듈 간 상호작용	보통	중간	Jest, Mocha, Pytest
E2E	사용자 시나리오 전체	느림	높음	Playwright

왜 E2E 테스트인가?

01 **실제 사용자 경험 검증**
사용자 관점에서 전체 흐름 테스트

02 **크로스 브라우징 이슈 방지**
Chrome, Safari, Firefox 등 다양한 환경

03 **배포 전 최종 테스트**
CI/CD 파이프라인에서 자동 검증

핵심 전략

핵심 시나리오만 최소화해서 접근

- 로그인/회원가입
- 주요 기능
- 핵심 사용자 플로우

Playwright 소개

Microsoft Open Source

브라우저 자동화 프레임워크

웹 테스트 및 자동화를 위한 오픈소스 라이브러리

Chromium	Firefox	WebKit

Windows, Linux, macOS + 모바일 환경 지원

Playwright – 작동 원리

브라우저 컨텍스트란?

브라우저의 **시크릿 모드(Incognito)**와 유사한 격리된 환경

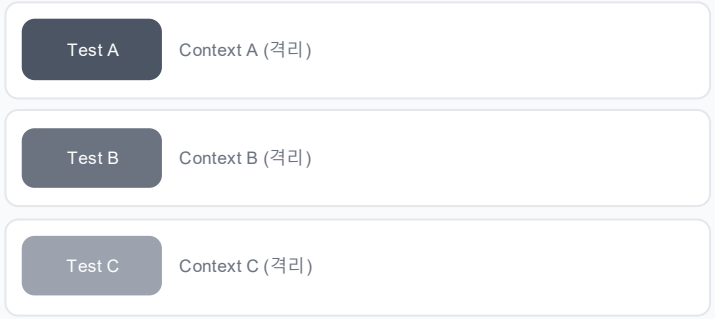
격리되는 것들

- 쿠키 (Cookies)
- 로컬 스토리지 (localStorage)
- 세션 스토리지 (sessionStorage)
- 캐시 (Cache)

생성 속도

새 컨텍스트 생성 = **밀리초 단위** (거의 오버헤드 없음)

병렬 실행 시에도 안전



테스트 A에서 로그인해도 테스트 B에는 영향 없음!

Playwright 프로젝트 구조

폴더 구조

```
my-project/
├── tests/
│   └── example.spec.ts
├── playwright.config.ts
├── package.json
└── test-results/
```

tests/
테스트 파일들이 위치. *.spec.ts 패턴

playwright.config.ts
브라우저, 타임아웃, 리포터 등 설정

test-results/
테스트 결과, 스크린샷, 비디오 저장

Playwright - Spec, Fixture

Spec (Specification)

"무엇을 테스트할 것인가?"

- 테스트 시나리오와 검증 로직
- .spec.ts 파일에 작성
- test() 함수로 케이스 정의

Fixture

"어떤 환경에서 실행할 것인가?"

- 테스트 실행 환경 제공
- 자동 설정(Setup) 및 정리(Teardown)
- 테스트 간 완전 격리 보장

```
// example.spec.ts

import { test, expect } from '@playwright/test';

test('basic test', async ({ page }) => {
// { page }가 바로 Fixture!
await page.goto('https://example.com');
await expect(page).toHaveTitle(/Example/);
});
```

핵심 포인트

Playwright가 {page}를 자동으로 생성하고 주입해줍니다. 테스트 끝나면 자동 정리!

Playwright - Page Object Model (POM)

POM이란?

웹 페이지를 **클래스(Class)**로 정의하여 테스트 코드와 UI 조작을 분리하는 디자인 패턴

역할 분리

Spec (테스트)	↔	Page Object
"무엇을 검증"		"어떻게 조작"

POM의 3가지 장점

1. 유지보수 용이
UI 변경 시 Page 클래스만 수정

2. 코드 재사용
로그인 등 공통 동작을 여러 테스트에서 사용

3. 가독성 향상
loginPage.clickLogin() vs page.locator(...)

Playwright - POM + Fixture

```
// pages/LoginPage.ts (Page Object)

export class LoginPage {
constructor(private page: Page) {}

async login(email: string, pw: string) {
await this.page
.getByPlaceholder('이메일')
.fill(email);
await this.page
.getByPlaceholder('비밀번호')
.fill(pw);
await this.page
.getByRole('button').click();
}
}
```

```
// 테스트에서 사용

test('로그인', async ({ page }) => {
const loginPage =
new LoginPage(page);
await loginPage.login(
'test@test.com', '1234');
});
```

Fixture 결합 (심화)

test.extend()로 loginPage를 Fixture로 등록하면 매번 새로할 필요 없이 자동 주입 가능

테스트 구조

```
// login.spec.ts

import { test, expect } from '@playwright/test';

test.describe('로그인 기능', () => {
test.beforeEach(async ({ page }) => {
await page.goto('/login');
});

test('성공 케이스', async ({ page }) => {
// 테스트 로직
});
});
```

test.describe()
관련 테스트를 그룹화

test.beforeEach()
각 테스트 전 실행되는 설정

test()
개별 테스트 케이스 정의

{ page }
Playwright가 제공하는 Fixture

Playwright 기초 문법

01

Locator

요소 찾기

→

02

Action

동작 실행

→

03

Assertion

결과 검증

요소를 찾고 → 동작을 실행하고 → 결과를 검증한다

Locator - 요소 찾기 (1)

```
// page.locator() - CSS 선택자
await page.locator('button').click();
await page.locator('.btn').click();
```

```
// page.getByRole() - ARIA role
await page.getByRole('button',
{ name: '로그인' }).click();
```

```
// page.getByTestId() - data-testid
await page.getByTestId('submit')
.click();
```

Locator 선택 우선순위

- getByRole - 접근성 기반 (권장)
- getByTestId - 테스트 전용 ID
- getByText - 텍스트 기반
- getByPlaceholder - placeholder
- locator - CSS (최후 수단)

CSS 클래스보다 역할(role)이나 텍스트 기반이 더 안정적입니다.

Locator - 요소 찾기 (2)

```
// page.getByText() - 텍스트 기반
await page.getByText('환영합니다')
.isVisible();
// exact: true 로 정확 매칭
```

```
// page.getByLabel() - label 기반
await page.getByLabel('비밀번호')
.fill('mypassword');
```

```
// page.getByPlaceholder()
await page
.getByPlaceholder('이메일 입력')
.fill('test@test.com');
```

```
// page.getByAltText() - alt 속성
await page
.getByAltText('프로필 이미지')
.click();
```

Locator - 필터링과 체이닝

```
// locator.filter() - 조건부 필터링
await page.locator('li')
.filter({ hasText: '완료됨' })
.click();
```

```
// 인덱스로 선택
await page.locator('li').first();
await page.locator('li').last();
await page.locator('li').nth(2);
```

체이닝 패턴

Locator 메서드들은 체이닝이 가능합니다.

```
// 체이닝 예시
await page
.getByRole('list')
.filter({ hasText: '할 일' })
.getByRole('button')
.click();
```

Playwright 기초 문법

01

Locator

요소 찾기

→

02

Action

동작 실행

→

03

Assertion

결과 검증

요소를 찾고 → 동작을 실행하고 → 결과를 검증한다

Action - 동작 실행

```
// click() - 클릭
await page.getByRole('button').click();
```

```
// fill() - 입력 (기존 값 삭제 후)
await page.getByPlaceholder('이메일')
.fill('test@test.com');
```

```
// type() - 타이핑 (한 글자씩)
await page.locator('input')
.type('Hello', { delay: 100 });
```

```
// press() - 키보드 입력
await page.keyboard.press('Enter');
await page.keyboard.press('Tab');
```

```
// hover() - 마우스 호버
await page.locator('.menu').hover();
```

```
// check() / uncheck() - 체크박스
await page.getByLabel('동의').check();
await page.getByLabel('동의').uncheck();
```

Auto-Wait 기능

Auto-Wait이란?

Playwright는 액션 수행 전 요소가 준비될 때까지 자동으로 대기합니다.

자동 대기 조건

- 요소가 DOM에 존재
- 요소가 화면에 보임 (visible)
- 요소가 안정적 (애니메이션 완료)
- 요소가 활성화 (enabled)

장점

- sleep() 없이 안정적 테스트
- Flaky 테스트 감소
- 비동기 로딩 자동 처리

```
// 명시적 대기가 필요할 때
await page.waitForURL('/dashboard');
await locator.waitFor();
```

Playwright 기초 문법

01

Locator

요소 찾기

→

02

Action

동작 실행

→

03

Assertion

결과 검증

요소를 찾고 → 동작을 실행하고 → 결과를 검증한다

Assertion - 결과 검증

```
// 요소 가시성 검증
await expect(locator).toBeVisible();
await expect(locator).toBeHidden();
```

```
// 텍스트 검증
await expect(locator)
.toHaveText('환영합니다');
await expect(locator)
.toContainText('환영');
```

```
// 개수 검증
await expect(locator).toHaveCount(5);
```

```
// URL 검증
await expect(page)
.toHaveURL('/dashboard');
```

```
// 속성/값 검증
await expect(locator)
.toHaveValue('입력된 값');
await expect(locator)
.toHaveAttribute('href', '/link');
```

Web-first Assertions

권장: Web-first Assertion

```
await expect(locator).toBeVisible();
```

- 조건 충족까지 자동 재시도 (기본 5초)
- 비동기 웹 환경에 최적화
- Flaky 테스트 방지

지양: 수동 Assertion

```
expect(await page.isVisible()).toBe(true);
```

- 기다리지 않고 즉시 반환
- 요소 로딩 중이면 실패
- Flaky 테스트 원인

자동 재시도 메커니즘

요소가 0.5초 뒤에 나타나면? → Web-first는 기다렸다가 통과 / 수동은 즉시 실패

테스트 실행 방법

```
# 모든 테스트 실행
npx playwright test
```

```
# 특정 파일만 실행
npx playwright test login.spec.ts
```

```
# UI 모드로 실행
npx playwright test --ui
```

```
# 테스트 리포트 확인
npx playwright show-report
```

VS Code Extension

- 사이드바에서 테스트 목록 확인
- 개별 테스트 클릭으로 실행
- 디버깅 모드 지원
- Pick Locator 기능

추천 워크플로우

개발 중: VS Code Extension / CI/CD: CLI 명령어

테스트 실행 옵션

플래그	설명	예시
--ui	UI 모드 실행	`npx playwright test --ui`
--headed	브라우저 창 표시	`npx playwright test --headed`
--debug	디버거 모드	`npx playwright test --debug`
--trace on	Trace 수집	`npx playwright test --trace on`
--project	특정 브라우저 지정	`--project=chromium`
-g	테스트명 필터	`-g "로그인"`

테스트 실행 방법

1 **테스트 실행** `npx playwright test`

→

2 **결과 확인** 터미널에서 ✓/✗

→

3 **리포트 확인** `show-report`

→

4 **Trace 분석** `show-trace`

```
# 기본 실행
npx playwright test login.spec.js
```

```
# 브라우저 보면서
npx playwright test --headed
```

```
# 리포트 보기
npx playwright show-report
```

테스트 - UI 모드

```
# UI 모드 시작
npx playwright test --ui
```

주요 기능

1. **Watch 모드** - 파일 변경 시 자동 재실행
2. **Time Travel** - 각 액션 시점 화면 확인
3. **Pick Locator** - 요소 클릭으로 선택자 생성

테스트 실행– Trace Viewer

```
# Trace 수집
npx playwright test --trace on
--trace on-first-retry (권장)
```

```
# Trace 보기
npx playwright show-trace
```

Trace Viewer 탭

Call - 액션 상세 정보
Console - 브라우저 콘솔 로그
Network - API 요청/응답
Source - 소스 코드 위치

로그인 테스트 코드 예시

```
// tests/auth/login.spec.ts

import { test, expect } from '@playwright/test';

test('로그인 성공', async ({ page }) => {
// 1. 페이지 접속
await page.goto('http://localhost:3000/login');

// 2. 입력
await page.getByPlaceholder('이메일')
.fill('test@kmap.com');
await page.getByPlaceholder('비밀번호')
.fill('password123');

// 3. 클릭
await page.getByRole('button',
{ name: '로그인' }).click();

// 4. 검증
await expect(page).toHaveURL('/dashboard');
});
```

포인트

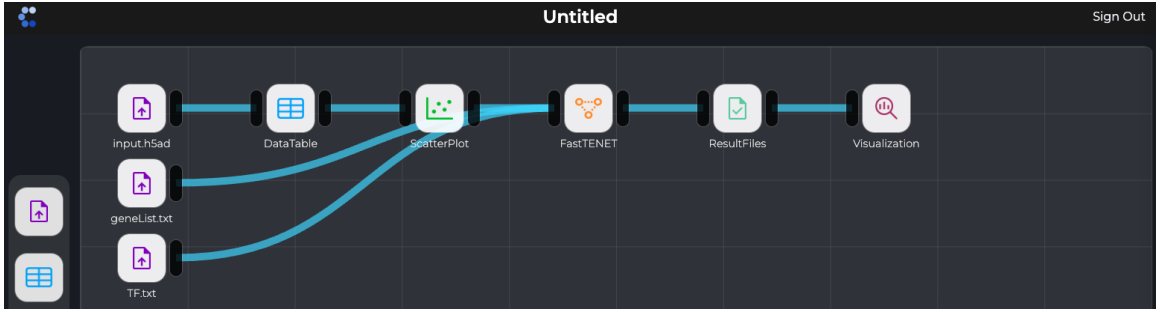
getByPlaceholder, getByRole 사용

사용자 관점의 Locator 선택

실행 명령어

```
npx playwright test login.spec.ts
```

CellCraft 테스트 예시



01 파일 할당 → 02 데이터 표시 → 03 시각화 → 04 알고리즘 → 05 실행 → 06 결과

01 파일 할당
InputFile 노드에 h5ad 파일 할당, Vuex 스토어 영속성 검증

02 데이터 표시
DataTable 렌더링, 백엔드 API 응답 및 컬럼/행 검증

03 산점도
Plotly UMAP 시각화, X/Y축 드롭다운 인터랙션

04 알고리즘 설정
TENET 파라미터 구성, 다양한 입력 타입 수정 검증

05 워크플로우 실행
상태 모니터링 (PENDING→RUNNING→REVOKED), 로그 및 DAG 시각화

06 결과 시각화
GRNViz 플러그인 실행, Plotly 차트 렌더링 및 다운로드 검증

CellCraft 테스트 예시

활용 목적

- 라이브 서비스 모니터링
- 일정 주기마다 자동 실행
- 핵심 기능 정상 작동 검증
- 배포 전 회귀 테스트

테스트 구성

6개 핵심 시나리오로 사용자 워크플로우 전체 흐름 커버

테스트 폴더 구조

```
frontend/tests/e2e/workflows/
├── 01-file-assignment.spec.js
├── 02-data-display.spec.js
├── 03-scatter-plot.spec.js
├── 04-algorithm-config.spec.js
├── 05-workflow-execution.spec.js
└── 06-result-visualization.spec.js
```

```
# 전체 테스트 실행
npx playwright test tests/e2e/workflows/
```

MCP + Playwright

MCP (Model Context Protocol)

AI 모델이 외부 도구와 상호작용할 수 있게 해주는 프로토콜

Playwright MCP Server

Claude가 Playwright를 직접 제어하여 브라우저 자동화를 수행

가능한 것들

- 자연어로 테스트 시나리오 작성 요청
- AI가 직접 브라우저 조작
- 스크린샷 캡처 및 분석
- 테스트 코드 자동 생성

활용 예시

"로그인 페이지에서 잘못된 비밀번호를 입력했을 때 에러 메시지 테스트해줘"

MCP 활용 시연

시연 내용

- 자연어로 테스트 요청
- AI가 브라우저 자동 조작
- 결과 확인 및 코드 생성

예시 프롬프트

"KMAP 사이트에 접속해서 로그인 폼이 제대로 표시되는지 확인해줘"

Q&A

다음 회차 안내

📅 1/27 (화) – PR 및 코드 리뷰 중간 점검

📝 과제:
- PR, 코드 리뷰 각자 1개 이상 하기 (다음주 화요일까지)
- 테스트 관련 블로그 글 작성하기 (다음주 목요일까지)