

한 걸음 앞선 개발자가 지금 꼭 알아야 할 클로드 코드

한 걸음 앞선 개발자가

지금 꼭 알아야 할

MASTERING
CLAUDE
CODE

조훈, 정찬훈 지음

클로드

코드

CLAUDE
CODE

실무에서 검증된 개발 방식 그대로,

매일 1시간 4주

Claude Code 에이전트 실전 훈련!

```
# 설치(macOS, Windows)
# CLAUDE.md 설정
# MCP 연동/다양한 활용 전략
# 클로드 워크플로 전략
# 설계 → 부트스트래핑 →
  테스트 → 개선 → 명세
# 클로드 코드의 효율을 극대화
  하는 다양한 방법
```

갈벗

깃허브: https://github.com/sysnet4admin/_Book_Claude-Code

인공지능이 코드 생성과 수정의 주요 부분을 담당하게 되면서, 테스트는 선택할 수 있는 옵션이 아니라 코드의 기능 보장을 위한 필수적인 부분이 되었습니다. 이 문서에서는 웹 개발의 테스트 도구로 가장 많은 사용자와 레퍼런스를 가지는 [Playwright](#)(플레이라이트, 한국어로 쓰는 경우가 거의 없어서 우선 영어로 표기)를 사용하여 인공지능이 생성한 코드가 안전하게 동작하도록 테스트하는 방법에 대해서 알아보겠습니다.

시작하기에 앞서, 테스트가 왜 에이전트 기반 코딩에서 필수 요소인지 살펴보겠습니다. 테스트를 가장 쉽게 한단어로 표현하면, 도로 위의 안전 장치인 **가드레일(Guardrail)**이라고 할 수 있습니다.

[그림 1] 도로에 자욱한 안개 낀 상황에서도 가드레일이 있어 도로 밖으로 나가는 상황을 막을 수 있음



가드레일은 도로에서 가장 흔히 볼 수 있는 구조물로서 안전을 지키는 마지막 수문장 역할입니다. 이와 같이 가끔 어디로 갈지 모르는 인공지능 에이전트를 사용한 코딩에도 테스트를 통해 의도한 바에서 완전히 벗어나는 상황을 통제해야 합니다. 즉, 테스트 없이는 인공지능이 생성한 코드는 단순한 실험 결과물에 머물지만, 충분한 테스트 후에는 신뢰할 수 있는 산출물이 됩니다. 특히 웹 개발은 테스트의 중요성이 두드러지는 영역입니다. 사용자 인터페이스는 끊임없이 바뀌고, 브라우저와 디바이스 환경은 다양하며 네트워크 상태나 권한 설정 같은 외부 요인도 결과에 큰 영향을 줍니다. 이런 특성 때문에 단순한 유닛 테스트만으로는 실제 사용자 경험을 보장하기 어렵습니다.

그러한 복잡한 요구 사항을 충족하기 위해 **웹 테스트 자동화 도구**가 필요합니다. 웹 테스트 자동화 도구는 브라우저 단에서 실제 사용자가 겪는 시나리오를 그대로 재현해주고, 클릭, 입력, 네트워크 요청/응답까지 전체 흐름을 검증해 줍니다. 또한 크로스 브라우징, 모바일 뷰, 권한 에뮬레이션 같은 기능을 활용하면 다양한 환경에서 동일한 동작을 보장하여 보다 정확한 테스트를 진행할 수 있습니다. 따라서 이미 많은 레퍼런스를 가지고 있는 Playwright와 다른 도구들과 비교를 진행하여 왜 Playwright로 진행하려고 하는지

알아보겠습니다. 그리고 Playwright의 핵심 기능을 직접 실습해서 동작 방식을 이해하고 난 후에 클로드 코드에 MCP를 연결해 인공지능 에이전트를 통한 웹 테스트는 어느정도로 자동화되는지 직접 확인해 보도록 하겠습니다.

1. 웹 테스트 자동화 도구 비교

웹 테스트 자동화 도구는 시대에 따라 변화했다고 볼 수 있습니다.

항목	 Selenium	 Puppeteer	 Playwright
 출시 연도	2004	2017	2019
 제공자	오픈 소스	Google	Microsoft
 주요 언어	Java, Python, C#, Ruby, JS 등	JavaScript / TypeScript	JS/TS, Python, Java, C#
 지원 브라우저	모든 주요 브라우저 (Chrome, Firefox, Safari, Edge, IE 등)	Chromium (Chrome/Edge), Firefox 일부 (실험적)	Chromium(Chrome/Edge), Firefox, WebKit(Safari 엔진)
 속도	 느림 (WebDriver 기반)	 빠름 (Chrome/Chromium)	  빠름 (자동 대기·병렬 실행·최신 구조)
 안정성	 매우 안정적 (20년 역사, 표준)	 크롤링/스크린샷에 안정적	 최신 구조로 안정적, QA/E2E 테스트에 강함
 생태계/커뮤니티	 매우 크고 성숙 (문서·플러그인 풍부)	 작음, 주로 크롤링 중심	 빠르게 성장 중, Selenium 대체 후보
 주요 활용	 대기업·대규모 테스트 자동화 표준 (CI/CD, QA 팀)	 크롤링, 데이터 수집, 스크린샷, PDF 생성	 차세대 E2E 테스트, QA 자동화, 멀티 브라우저 테스트
 특징	- 오래된 도구, 안정성·호환성 최고 - CI/CD 최적 - 속도 느림	- Headless Chrome 최적 - 최신 웹 기능 지원 - JS/TS 한정	- Puppeteer 팀 출신 제작 - Auto-wait, 네트워크 제어, 모바일 시뮬 - 설치 시 브라우저 번들 포함

2019년 그리고 2020년 정식으로 오픈 소스가 된 **Playwright**는 Selenium과 Puppeteer의 한계를 보완하는 형태로 진화하고 있으며, 이에 따라 현재 가장 널리 쓰이는 웹 브라우저 테스트를 위한 자동화 도구가 되었습니다. 특히 웹 개발에서 꼭 필요한 E2E(End-to-End) 테스트를 가장 효과적으로 제공하며, 크롤링과 같이 테스트 시나리오를 작성하기 위한 여러 부가 기능도 함께 제공하고 있습니다. 그외 Playwright에 대한 보다 자세한 내용은 [공식 홈페이지](#)의 내용을 참고해주시기 바랍니다.

2. Playwright

Playwright는 기본적으로 웹 어플리케이션을 위한 E2E 테스트 프레임워크입니다. Playwright 이름 하에 다양한 패키지들을 제공합니다. 기본이 되는 **playwright** 패키지와 이를 활용하는 테스트 프레임워크인 **@playwright/test** (일명 테스트 러너)를 중심으로 Playwright를 어떻게 활용되는지 살펴보고, **@playwright/mcp** 를 통해 자연어로 이를 제어하는 실습을 진행 하도록 하겠습니다.

<노트> E2E에서 가장 중요한 부분 중에 하나인 테스트 러너

테스트 러너(Test Runner)는 단순히 “테스트 코드를 실행하는 도구” 이상의 개념입니다. 프로그래밍 언어와 프레임워크 세계에서는 테스트 실행, 결과 수집, 보고를 위한 일련의 기능 집합을 묶어서 통상적으로 테스트 러너라고 부릅니다. 현업에서는 테스트 프레임워크, 테스트 도구, 테스트 환경과 같은 용어들과 뒤섞여 거의 같은 뜻으로 쓰이는 경우도 많습니다.

일반적인 테스트 러너는 다음과 같은 핵심 요소들을 포함합니다.

- 테스트 디스커버리: 프로젝트 안에서 작성된 테스트 케이스를 자동으로 탐색하고 수집하는 기능
- 실행 엔진: 테스트 코드를 순차/병렬적으로 실행하면서 환경을 초기화하고 종료하는 기능 제공
- 어설션(Assertion)과 매처(Matcher): 코드의 실제 동작 결과가 기대와 일치하는지를 판별하는 도구
- 보고: 콘솔 출력, HTML/JSON 리포트, CI 통합 리포트 등 다양한 형식으로 결과를 제공
- 훅/플러그인: 사전/사후 동작이나 추가 기능을 쉽게 확장할 수 있도록 함

그리고 프로그래밍 언어별 대표적인 테스트 러너는 다음과 같습니다.

- JavaScript/TypeScript: Jest, Mocha
- Python: pytest
- Java: JUnit
- Go: go test

이런 언어별이 아닌 목적별로 특화된 서비스가 존재하며, 그 중에 대표적인 것이 지금부터 다룰 Playwright에서 제공하는 테스트 러너 입니다. 해당 패키지 이름은 **@playwright/test** 입니다.

각 제품과 언어별 구현은 다르지만, 공통적으로 코드를 검증하고 품질을 담보한다는 목적을 가집니다. 그래서 통칭하여 “테스트 러너”라는 명칭을 사용합니다. 특히 대규모 시스템, 클라우드 환경, 인공지능 코드 생성 시대에서는 테스트 러너가 단순한 편의 도구가 아니라 가장 강력한 가드레일 중 하나가 됩니다. AI가 생성한 코드가 언제나 불완전할 수 있는 상황에서, 테스트 러너는 자동으로 실행·검증·리포트하는 과정을 통해 결과물의 신뢰성을 확보하므로 점진적으로 이에 대한 중요성은 높아질 것 입니다. </노트>

2.1 Playwright의 4가지 대표 기능

실습에 앞서 Playwright가 제공하는 4가지 대표적인 기능을 먼저 간략하게 소개드립니다. 해당 기능들을 통해 Playwright는 웹 기반 E2E 테스트를 작성에 필요한 다양한 편의를 제공합니다.

자동 대기

명령을 실행하는 데 그치지 않고, 유관된 요소가 실제로 준비될 때까지 자동으로 기다려 줍니다.

예를 들어 버튼을 클릭하기 전에 DOM에 나타날 때까지, 입력 필드를 채우기 전에 활성화될 때까지 알아서 기다립니다. 이러한 부분이 자동적으로 처리되지 않으면 테스트 코드 레벨에서 하나하나 직접 `sleep(1000)` 같은 임시 지연 코드를 일일이 넣어야하고 실제 테스트하고자하는 동작과는 별개의 지저분한 코드들이 많이 작성되게 됩니다. 또한 적절한 처리 로직의 부재로 인해 발생할 수 있는, 불안정하게 실패하는 Flaky 테스트(상황에 따라 다른 결과가 나오는 불안정한 테스트)를 크게 줄일 수 있습니다.

expect API

웹 페이지의 요소들이 어떤 상태여야하는지를 테스트하는건 쉽지 않습니다. `expect` API를 활용하면 보다 직관적으로 웹 요소들의 상태를 검증할 수 있습니다. `expect` 구문을 이용하면 DOM 요소의 텍스트, URL, 네트워크 응답까지 직관적으로 검증할 수 있습니다. `expect(page).toHaveURL(...)`이나 `expect(locator).toHaveText(...)`처럼 사람이 평소 쓰는 문장과 비슷한 방식으로 작성할 수 있어 가독성이 좋습니다. 테스트 코드가 곧 시나리오 문서처럼 읽히는 효과가 있습니다.

병렬 실행

테스트가 많아질수록 실행 시간이 길어지는데, Playwright는 기본적으로 테스트 케이스를 병렬로 실행해 속도를 높여 줍니다. CI/CD 환경에서는 테스트 수십 개, 수백 개를 동시에 돌려도 안정적으로 관리할 수 있어, 대규모 프로젝트에서도 빠른 피드백 루프를 유지할 수 있습니다.

리포팅/디버깅

실패한 테스트를 분석할 때는 단순 로그만으로 부족할 수 있습니다. Playwright는 HTML 리포트와 함께 스크린샷, 실행 영상, 네트워크 트레이스, 콘솔 로그까지 자동으로 수집할 수 있습니다. 덕분에 “테스트가 왜 실패했는지”를 시각적으로 바로 확인할 수 있고, Trace Viewer로 실행 과정을 단계별로 재생해 보며 원인을 빠르게 추적하는데 도움을 줍니다.

2.2 Playwright를 이용한 E2E 테스트

간단한 테스트를 통해 Playwright의 기능을 확인하겠습니다. 먼저 기본 패키지인 **playwright**와 **playwright**의 테스트 러너 프레임워크인 **@playwright/test**를 설치합니다. **playwright** 공통 컴포넌트 및 제어도구들을 포함하며, **@playwright/test**는 **playwright**를 활용, E2E 테스트를 수행하는데 필요한 테스트 러너, 리포트 등을 제공합니다.

<Terminal박스>

```
$ npm install playwright
```

```
$ npm install @playwright/test
```

</Terminal박스>

네이버 검색 기능을 검증하는 테스트를 통해 Playwright를 좀 더 이해해보겠습니다.

테스트 시나리오는 성공/실패 두가지로 각각 다음과 같습니다.

성공 케이스	실패 케이스
<ul style="list-style-type: none">- www.naver.com 진입- 날씨 키워드를 검색- search.naver.com으로 페이지 이동- <code>#main_pack</code> 요소가 나올때까지 대기	<ol style="list-style-type: none">1. 존재하지 않는 요소 찾기<ul style="list-style-type: none">- www.naver.com 진입- <code>#non-existent-element</code> 요소가 나올때까지 대기2. 네트워크 에러<ul style="list-style-type: none">- 의도적인 네트워크 에러 주입3. 자바스크립트 에러<ul style="list-style-type: none">- 의도적인 스크립트 에러 주입

실패 케이스는 테스트가 실패할때 당시의 여러 컨텍스트 정보를 수집하기위해 의도적으로 실패하도록 만든 테스트 입니다. Playwright가 여러 방식으로 에러를 주입할 수 있음을 보여드리고 싶어 다양한 실패케이스를 적어봤습니다.

위 내용은 아래와 같은 자바스크립트(js)로 작성할 수 있습니다:

<Code박스> **index.test.js**

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

```

test.describe('Example Tests with Rich Reporting', () => {

  test('성공 테스트 - 네이버 검색', async ({ page }) => {
    // Console 로그 캡처
    page.on('console', msg => {
      console.log(`Console ${msg.type()}: ${msg.text()}`);
    });

    // Network 요청 모니터링
    page.on('request', request => {
      console.log(`Request: ${request.method()} ${request.url()}`);
    });

    await page.goto('https://www.naver.com');

    // 스크린샷 수동 캡처
    await page.screenshot({ path: 'test-results/naver-main.png' });

    await page.fill("input[name='query']", "날씨");
    await page.press("input[name='query']", "Enter");

    await page.waitForURL("**/search.naver.com/**");
    await page.waitForSelector("#main_pack", { timeout: 10000 });

    // 검색 결과 페이지 스크린샷
    await page.screenshot({ path: 'test-results/search-results.png' });

    const title = await page.title();
    expect(title).toContain("playwright");
  });

  test('실패 테스트 - 존재하지 않는 요소', async ({ page }) => {
    // Console 로그 수집
    const consoleLogs = [];
    page.on('console', msg => {
      consoleLogs.push(`${msg.type()}: ${msg.text()}`);
    });

    await page.goto('https://www.naver.com');

    // 의도적으로 실패시켜서 스크린샷과 trace 생성
    await expect(page.locator('#non-existent-element')).toBeVisible({ timeout:
3000 });
  });
}

```



```

test('네트워크 에러 시뮬레이션', async ({ page }) => {
  // 네트워크 요청 차단
  await page.route('**/*', route => {
    if (route.request().url().includes('search.naver.com')) {
      route.abort();
    } else {
      route.continue();
    }
  });

  await page.goto('https://www.naver.com');
  await page.fill("input[name='query']", "날씨");
  await page.press("input[name='query']", "Enter");

  // 이 테스트는 실패할 것임 (네트워크 요청이 차단됨)
  await page.waitForURL("**/search.naver.com/**", { timeout: 5000 });
});

test('JavaScript 에러가 있는 페이지', async ({ page }) => {
  const errors = [];
  const consoleMessages = [];

  // JavaScript 에러 캡처
  page.on('pageerror', error => {
    errors.push(error.message);
    console.log('Page error:', error.message);
  });

  // Console 메시지 캡처
  page.on('console', msg => {
    consoleMessages.push(`${msg.type()}: ${msg.text()}`);
  });

  await page.goto('https://www.naver.com');

  // JavaScript 에러 발생시키기
  await page.evaluate(() => {
    console.log('This is a test log message');
    console.warn('This is a warning message');
    console.error('This is an error message');

    // 의도적으로 에러 발생
    throw new Error('Intentional test error for demonstration');
  });
});

```



```
// 페이지에 임의의 검증 추가 (통과할 것임)
await expect(page.locator('input[name="query"]')).toBeVisible();
});
});
```

</Code박스>

이대로도 괜찮지만 좀 더 풍부한 데이터 캡처를 위해 기존의 자바스크립트에 추가 설정을 더합니다.

<Code박스> **playwright.config.js**

```
import { defineConfig } from '@playwright/test';

export default defineConfig({
  testDir: '.',
  timeout: 30 * 1000,
  expect: {
    timeout: 5000
  },
  fullyParallel: true,
  forbidOnly: !!process.env.CI,
  reporter: [
    ['html', { open: 'never' }],
    ['list']
  ],
  use: {
    // Capture screenshot on failure
    screenshot: 'only-on-failure',
    // Record video on failure
    video: 'retain-on-failure',
    // Capture network trace
    trace: 'retain-on-failure',
    // Set viewport
    viewport: { width: 1280, height: 720 },
    // Ignore HTTPS errors
    ignoreHTTPSErrors: true,
  },
});
```

</Code박스>

위 설정에 따라 테스트가 실패하면 스크린샷, 녹화 영상, 네트워크 트레이스 캡처가 이뤄집니다.

여기까지 현재 경로에서 아래와 같은 구성이 보여야 합니다.

<Terminal박스>

```
$ tree -L 1 .  
.  
├── index.test.js  
├── node_modules  
├── package-lock.json  
├── package.json  
└── playwright.config.js
```

</Terminal박스>

이제 **playwright**의 테스트를 실행합니다. HTML로 결과를 리포트 받을 수 있도록 **--report=html** 옵션을 넣습니다.

<Terminal박스>

```
$ npx playwright test --reporter=html  
  
Running 4 tests using 4 workers  
  
index.test.js:5:7 › Example Tests with Rich Reporting › 성공 테스트 - 네이버 검색  
  
Request: GET https://www.naver.com/  
  
[종락]  
  
1 passed (6.9s)  
  
Serving HTML report at http://localhost:9323. Press Ctrl+C to quit.
```

</Terminal박스>

테스트가 끝나면 브라우저에 보고서 페이지가 열립니다.

이후에도 이 생성된 HTML 형태의 보고서를 보려면 **npx playwright show-report**를 실행하면 됩니다.

<Terminal박스>

```
$ npx playwright show-report
```

</Terminal박스>

작업 후 테스트에 성공한 내용과 실패한 내용을 가진 **보고서**는 다음과 출력됩니다..

All 4

✓ Passed 1

✗ Failed 3

Flaky 0

Skipped 0

9/21/2025, 8:21:26 PM Total time: 6.9s

▼ index.test.js

✗ Example Tests with Rich Reporting > 실패 테스트 - 존재하지 않는 요소 3.9s

index.test.js:34 ▶ ▮▮ View Trace

✗ Example Tests with Rich Reporting > 네트워크 에러 시뮬레이션 6.3s

index.test.js:47 ▶ ▮▮ View Trace

✗ Example Tests with Rich Reporting > JavaScript 에러가 있는 페이지 2.1s

index.test.js:65 ▶ ▮▮ View Trace

✓ Example Tests with Rich Reporting > 성공 테스트 - 네이버 검색 3.8s

index.test.js:5

실패한 테스트 케이스는 상세 보고서와 함께 스크린샷, 녹화 영상, 네트워크 트레이스 등이 제공됩니다.

Example Tests with Rich Reporting

실패 테스트 - 존재하지 않는 요소

index.test.js:34

View Trace | 3.9s

Run

Errors

Error: expect(locator).toBeVisible() failed

Locator: locator('#non-existent-element')

Expected: visible

Received: <element(s) not found>

Timeout: 3000ms

Call log:

- Expect "toBeVisible" with timeout 3000ms
- waiting for locator('#non-existent-element')

```
42 |
43 | // 의도적으로 실패시켜서 스크린샷과 trace 생성
> 44 | await expect(page.locator('#non-existent-element')).toBeVisible({ timeout: 3000 });
    |                                     ^
45 | });
46 |
47 | test('네트워크 에러 시뮬레이션', async ({ page }) => {
    | at /Users/gavin.jeong/src/playwright/index.test.js:44:57
```

Test Steps

Before Hooks

186ms

Navigate to "/" — index.test.js:41

393ms

Expect "toBeVisible" — index.test.js:44

3.0s

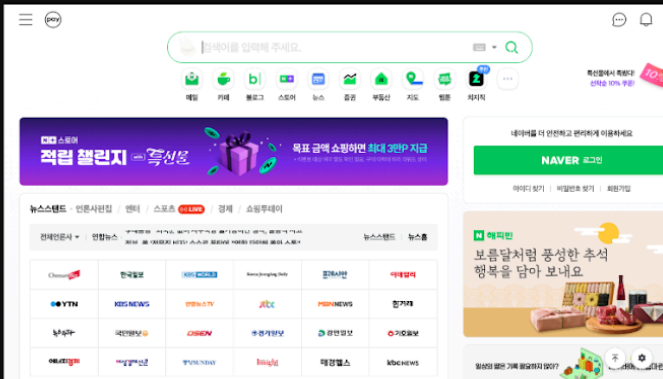
After Hooks

402ms

Worker Cleanup

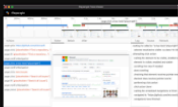
17ms

Screenshots



screenshot

Traces



trace

이렇게 강력한 기능을 갖고 있지만 이런 테스트를 일일이 작성하는건 그렇게 쉽지 않습니다. Playwright와 관련된 내용을 클로드 코드와 같은 인공지능 에이전트에게 맡기면 훨씬 수월하게 그리고 빠르게 진행할 수 있습니다. 이를 바로 이어서 확인해 보겠습니다.

2.3 Playwright와 MCP를 연동

이제 클로드 코드와 연동하기 위한 Playwright MCP를 구성하고 진행해 보겠습니다.

기존에 `@playwright/test`와 동일하게 `@playwright/mcp`를 통해서 설치할 수도 있습니다. 하지만 MCP는 상대적으로 최근에 만들어진 패턴이고 아직까지 보안적으로 타당한 구성이 맞는지는 논란의 소지가 있습니다. 하지만 이런 관점에서 최소한의 샌드박싱을 통해 MCP서버를 실행하는게 안전할 가능성이 높은 것은 사실이며, 편의적인 측면에서도 MCP를 일반 라이브러리와 분리해서 관리하는 편이 더 일반적입니다.

따라서 `npx`를 사용해 MCP를 연동하여 진행하겠습니다. `npx/uvx`에 관련한 설치 구성 그리고 유의할 점 등의 내용은 다음 [문서](#)를 참고하시기 바랍니다.

<Terminal박스>

```
$ claude mcp add playwright -- npx @playwright/mcp@latest
```

</Terminal박스>

이제 클로드 코드에게 자연어로 질의할 수 있습니다!!!

앞서 봤던 성공 케이스 테스트와 유사하게 날씨를 검색하도록 해봅시다. 빠르게 클로드 코드로 확인하기 위해 `YOLO(--dangerously-skip-permissions)` 모드로 진행합니다.

<Claude Code박스>

```
> Playwright로 네이버에서 오늘 서울 날씨를 검색해 결과를 알려줘

● 네이버에서 오늘 서울 날씨를 검색해드리겠습니다.

● playwright - Navigate to a URL (MCP)(url: "https://www.naver.com")
  └─ ⚠ Large MCP response (~10.4k tokens), this can fill up context quickly
  └─ ### Ran Playwright code

    ```js
 await page.goto('https://www.naver.com');

 ... +724 lines (ctrl+r to expand)
```

- 이제 검색창에 "서울 날씨"를 입력하겠습니다.

- playwright - Type text (MCP)(element: "검색어 입력창", ref: "e37", text: "서울 날씨", submit: true)

- └ Error: MCP tool "browser\_type" response (61502 tokens) exceeds maximum allowed tokens (25000). Please use pagination, filtering, or limit parameters to reduce the response size.

- 검색창에 입력하고 검색 버튼을 클릭하겠습니다.

[중략]

현재 온도: 21.2°C체감온도: 어제보다 0.3° 높음, 구름많음최저기온: 21.9°습도: 62%북동풍: 2.9m/s

하루 일정:

- 미세먼지: 좋음
- 초미세먼지: 좋음
- 자외선: 보통
- 일몰: 18:29

시간대별 온도 예보:

- 11시: 22°
- 12시: 24°
- 13시: 25°
- 14-16시: 26°
- 17시: 25°
- 18시: 24°
- 19-23시: 23-22°

현재 구름이 많은 날씨이며, 하루 종일 비교적 따뜻한 기온을 유지할 예정입니다. 미세먼지와 초미세먼지 모두 좋음 수준이라 외출하기에 좋은 날씨입니다.

</Claude Code박스>

이전에 살펴본 진행한 테스트 시나리오처럼 에이전트가 Playwright을 실행하여 네이버에서 “날씨”를 검색하는 동작을 수행함을 확인할 수 있습니다. 이번엔 기능 검증 동작을 넘어, 앞서 우리가 직접 작성해야했던 E2E 테스트를 만들어보도록 하겠습니다.

MCP로 연결된 클로드 코드에 E2E 테스트 작성을 자연어로 지시합니다.

<Claude Code박스>

> 네이버에서 서울 날씨를 검색하는 Playwright Test를 작성해줘

네이버에서 서울 날씨를 검색하는 Playwright 테스트를 작성해드리겠습니다.

- Write(index-test.js)

- └ Wrote 121 lines to index.test.js

```
const { test, expect } = require('@playwright/test');

test.describe('네이버 날씨 검색 테스트', () => {

 test('서울 날씨를 검색하고 결과를 확인한다', async ({ page }) => {

 // 1. 네이버 메인 페이지로 이동

 await page.goto('https://www.naver.com');

 // 2. 페이지 로딩 대기

 await page.waitForLoadState('networkidle');

 ... +111 lines (ctrl+r to expand)
```

[종락]

- 네이버에서 서울 날씨를 검색하는 Playwright 테스트를 완성했습니다!

</Claude Code박스>

앞서 확인한 예제와 유사한 구성의 **js 파일**이 바로 생성되는 걸 확인할 수 있습니다.  
(재미있는 사실! 현재 사용하는 모든 예제 코드는 클로드 코드로 생성했습니다. )

여기까지 PlayWright MCP에 대한 기본 활용을 확인해봤습니다.

실제 웹서비스 개발에서는 어떻게 활용할수 있을까요? 다음과 같이 요청할 수 있습니다.

<Claude Code박스>

> 간단한 CRUD 예제를 Typescript로 작성해줘, 기본적인 동작 검증을 할수 있는 e2e 테스트를 playwright test runner로 작성하고 검증해줘

- I'll create a simple CRUD example in TypeScript with Playwright e2e tests. Let me set up the project structure and implement the functionality.



- Let me first create the project structure and package.json: // 1. 코드 작성

- Write(package.json)

└─ Wrote 26 lines to package.json

```
{
 "name": "typescript-crud-example",
 "version": "1.0.0",
 "description": "Simple CRUD example with TypeScript and Playwright tests",
 "main": "dist/server.js",
 "scripts": {
 "build": "tsc",
 "start": "node dist/server.js",
 "dev": "ts-node src/server.ts",
 "test": "playwright test",
 ... +16 lines (ctrl+r to expand)
```

[종락]

- Now let me install dependencies and run the tests to verify everything works:

- Bash(npm install)

└─ added 105 packages, and audited 106 packages in 3s

```
14 packages are looking for funding
... +3 lines (ctrl+r to expand)
```

- Bash(npm run install-browsers)

```
└─ > typescript-crud-example@1.0.0 install-browsers
 > playwright install
```

- Bash(npm run build)

```
└─ > typescript-crud-example@1.0.0 build
 > tsc
```

- Bash(npm test) // 2. 테스트 및 실패

```
└─ Error: Command timed out after 2m 0.0s
```

```
> typescript-crud-example@1.0.0 test
```

```
> playwright test
```

```
Running 9 tests using 7 workers
```

```
[2/9] [chromium] › tests/crud.spec.ts:31:7 › User CRUD Operations › should create a new user
... +212 lines (ctrl+r to see all)
```

- I see the issue. The problem is with the dialog handling in the beforeEach hook. Let me fix the test by simplifying the beforeEach setup: // 3. 스스로 테스트 결과를 보며 코드 개선 시도

[종락]

- Perfect! The project is complete and working. Let me summarize what we've accomplished: // 4. 완성

Summary

- ✅ CRUD Application Created

- TypeScript backend with Express.js
- RESTful API with full CRUD operations for users
- Simple HTML frontend with JavaScript
- In-memory data storage with proper TypeScript interfaces

### ✅ Comprehensive E2E Testing

- 5 Playwright tests covering all CRUD operations
- Tests for user creation, reading, updating, and deletion
- Form validation testing
- UI element verification
- Sequential test execution to avoid conflicts

### ✅ All Tests Passing

5 passed (1.5s)

- ✅ Should create and display a user
- ✅ Should update a user
- ✅ Should delete a user
- ✅ Should validate form inputs
- ✅ Should display interface correctly

### ✅ Key Features

- Create: Form-based user creation with validation
- Read: User list display with all details
- Update: Inline editing with save/cancel options
- Delete: User removal with confirmation dialogs
- Frontend: Responsive interface with real-time updates
- Testing: Automated browser testing covering all functionality

The application is production-ready with proper TypeScript compilation, comprehensive testing, and clean architecture. You can run `npm run dev` to start the development server and `npm test` to verify all functionality works correctly.

</Cluade Code박스>

처리 과정을 모두 표현하기에는 다소 많아 중요한 일부 발췌해서 넣었습니다. 사실 프롬프트 자체에 크게 특별한건 없습니다. 다만 검증을 Playwright를 통해 검증하도록 했을뿐입니다. 하지만 이것만으로도 단순히 개발을 지시하는것에 비해 에이전트의 동작은 크게 달라집니다.

위에서 주석이 달린 부분을 보시면 에이전트가 스스로 **1. 코드를 작성하고, 2. 테스트 후 이슈를 발견, 3. 개선 시도를 거쳐 4. 완성**하는 과정을 스스로 밟고 있는걸 볼 수 있습니다. 결과적으로 어느 정도 차이가 있을 수 있지만 아래와 같은 형태로 산출물이 만들어집니다.

<Terminal박스>

```
$ tree -L 1 .
.
├── dist
├── node_modules
├── package-lock.json
├── package.json
├── playwright-report
├── playwright.config.ts
├── public
├── README.md
├── src
├── test-results
├── tests
└── tsconfig.json

8 directories, 5 files
```

</Terminal박스>

아직까지 LLM을 기반으로하는 AI는 뭐가 튀어나올지 확신할 수 없는 블랙박스입니다. 최근 빠른 발전으로 상당히 높은 확률로 기대하는 결과물을 뽑기시작했지만 그 자체론 아직 결과물을 충분히 담보하기 어렵습니다.

이때 이렇게 **인공지능 에이전트** 스스로 생성한 코드를 검증할 수 있게되면 매우 효과적입니다. 이 과정을 기존의 전통적인 개발방식이나 ChatGPT와 같은 LLM기반 인공지능을 보조로하는 코딩과 다음과 같이 비교해볼 수 있습니다.

- **기존 방식:** 사람이 테스트 코드 작성 → 실행 → 결과 확인 → 수정
- **인공지능 보조 코딩:** 인공지능이 코드 생성 → 사람이 실행 → 결과 검증도 사람이 진행
- **인공지능 에이전트 코딩:** 클로드 코드가 테스트 생성 → 실행 → 결과 해석 → 코드 개선 → 반복

이렇게 인공지능 에이전트가 작성한 코드를 테스트를 통해 검증까지 할 수 있게되면, 테스트가 실패할때 스스로 생성한 코드를 다시 수정하고 테스트 하는 과정으로 반복을 유도할 수 있습니다. 좀 더 짧은 표현으로는 피드백 루프를 만든다고도 표현합니다.

#### <노트> 피드백 루프

Feedback Loop, Self Feedback Loop, Self-Improving Loop, Self Refine Loop 등 다양한 형태로 표현되고 있습니다. 모두 AI 에이전트가 스스로의 출력물을 검증하여 이를 개선하는 사이클을 뜻합니다. 기존의 바이트 코딩과 에이전트 기반 코딩을 구별하는 가장 큰 특징이기도 합니다.

#### </노트>

Playwright는 에이전트에게 단순한 단위 테스트 수준을 넘어서 사용자 레벨의 실행 환경을 제공하고, 에이전트는 생성하는 주체이자 그것을 이해하고 수행하는 주체이기도 하여 테스트 개선을 위한 피드백 루프가 인공지능 에이전트를 통해 자동으로 진행됩니다. 예시는 단순하지만 좀 더 이를 확장하면 구현해야할 각각의 기능에 대한 명세를 두고 각 내용들을 Playwright로 검증, 좀 더 안정적으로 개발 산출물들을 만들 수 있도록 제어할 수 있게됩니다. 아래는 명세를 기반으로 이를 명령하는 입력 예시 입니다.

#### <Claude Code박스>

> 다음 명세를 구현하고 Playwright 테스트를 통해 검증해줘

[로그인]

- 사용자가 /login 페이지에 접속한다
- 아이디(testuser)와 비밀번호(password123)를 입력한다
- 로그인 버튼을 클릭한다
- "로그인 성공"이라는 문구가 페이지에 표시되어야 한다

[회원리스트]

- 회원 목록이 테이블 형태로 표시된다

- 각 회원의 정보(ID, 이름, 이메일, 가입일, 상태)가 보여진다
- 최소 5명 이상의 회원 데이터가 표시되어야 한다
- "회원 추가" 버튼이 페이지 상단에 위치한다
- 검색 기능으로 회원명으로 필터링할 수 있다

#### [중략]

#### [로그아웃]

- 회원 목록 페이지 우상단에 "로그아웃" 버튼이 표시된다
- 로그아웃 버튼을 클릭한다
- 로그인 세션이 종료되고 /login 페이지로 리다이렉트된다
- 로그아웃 후 /members 페이지에 직접 접속 시도 시 자동으로 /login 페이지로 이동한다
- 브라우저 웹로가기로 회원 목록에 접근할 수 없어야 한다

#### </Claude Code박스>

테스트는 단순한 품질 보증 절차가 아니라, 에이전트 코딩을 가능하게 하는 핵심 안전장치입니다.

Playwright는 넓은 브라우저 지원과 강력한 테스트 프레임워크 기능을 바탕으로, 사람 손으로는 관리하기 어려운 브라우저 동작을 안정적으로 자동화합니다. 여기에 MCP를 결합하면, 자연어를 통해 피드백 루프를 지시, 브라우저 테스트 기반의 개발 프로세스를 손쉽게 구현할 수 있습니다. 이를 통해 개발자가 보다 안정적이고 신뢰할 수 있는 산출물을 얻을 수 있는 토대를 만들 수 있습니다.