[asyncio & Playwright] Client행동모사 HTTP req/res 서버 테스팅 (예시와 함께)

▼ Await 문법

> `await` 키워드는 **반드시** `async` 함수 안에서만 사용해야 합니다.

- **비동기 함수**는 `async`로 정의되며, 이 함수 내부에서 시간이 오래 걸리는 작업(예: 파일 읽기, 네트워크 요청 등)을 `await` 키워드를 통해 비동기로 처리합니다.
- **await**는 해당 작업이 끝날 때까지 다른 작업(코루틴)들이 실행될 수 있도록 **이벤트 루프에 제어권**을 넘깁니다.
- 작업이 끝나면 제어권을 다시 받아서 나머지 코드를 계속 실행합니다.
- **await가 중요한 이유**
  - **네트워크 요청**이나 **파일 입출력** 같은 작업은 시간이 오래 걸리므로, 이러한 작업을 비동기로 처리하지 않으면 프로그램 전체가 멈추는 문제가 발생할 수 있습니다.
  - `await`는 이러한 작업이 끝날 때까지 **비동기적으로 대기**하여 **다른 작업을 동시에 수행**할 수 있게 해줍니다.
- Async란 ?
  - Python에서 **`async`**는 **비동기 프로그래밍**을 위한 **코루틴(coroutine)**을 정의하는 데 사용되는 키워드
- Await가 순차적으로 쌓여있을 때

```
await press_enter()
await select_radio_button('F')
await click_button('jspsych-survey-multi-choice-next')
await press_enter()
await press_enter()
```

위와 같이 Await가 순차적으로 쌓여있는 경우, 각 비동기함수끼린ㄴ 순차적으로 실행된다. 즉 Await플래그가 있다면 앞 Await플래그 구문이 끝나기 전까지 대기한다.

▼ Finally 란?

Python에서 **`finally`**는 **`try`-`except`-`finally` 구문**의 일부로, **예외 처리와 상관없이 항상 실행되는 코드 블록**을 정의합니다.

- **`finally`의 목적**
  1. **리소스 정리**: 파일, 네트워크, 데이터베이스 연결 등을 **닫는 작업**.
  2. **코드 안정성 보장**: 예외 발생 여부와 상관없이 **중요한 작업**을 수행해야 할 때.
  3. **프로그램 종료 전 작업**: 로그 저장, 세션 종료 등 프로그램의 **마무리 작업**을 수행할 때.

▼ Asyncio.gather이란

여러 비동기 작업을 동시에 실행할 때 사용하는 방법 즉, 병렬실행을 위한 방법

- Await구문은 별다른 처리가 없으면 이전 Await구문이 끝날때 까지 대기한다. 즉 비동기 작업들이 순차적으로 실행되는데
- 비동기 작업들 끼리 분리해서 병렬로 실행시키고 싶다면, 비동기 작업들을 정의하고 asyncio.gather(task1() , task2()) 와 같이 실행하면 된다.
-

▼ Asyncio로 이벤트 루프를 통한 비동기 작업 핸들링

`asyncio.get_event_loop()`로 현재 이벤트 루프를 가져와 `run_until_complete()`로 비동기 함수가 완료될 때까지 실행합니다.

```
loop = asyncio.get_event_loop()  # 이벤트 루프 생성 또는 가져오기
```

```
loop.run_until_complete(main())  # main() 함수가 끝날 때까지 이벤트 루프 실행
```

1. loop = asyncio.get_event_loop()

   현재 쓰레드에 설정된 이벤트 루프를 가져오는 함수이다. 현재 쓰레드에 설정되어 있는 **이벤트 루프가 없다면 이벤트 루프를 새로 생성하여 이를 현재 쓰레드에 설정한 뒤 해당 이벤트 루프를 반환한다.** 즉 이 함수의 호출은 코루틴의 실행을 위해 이벤트 루프를 준비하는 과정으로 볼 수 있다. async 식별자로 시작하는 함수들이 이제 이 이벤트 루프에서 실행된다.

   이벤트 루프에 대해서 더 자세하게 설명하면, 이벤트 루프는 비동기 작업들을 예약하고 실행하는 시스템이다. 이는 이벤트 루프는 `콜백 함수가 미래의 어떤 시점에서 호출되도록 예약하는 메커니즘`을 제공한다.

   asyncio라는 라이브러리는 여러 Task가 있을 때 총괄 책임자의 역할을 하는 기능을 가지고 있어, 작업들에 대한 이벤트 작업들을 핸들링 및 스케줄링한다. 즉 컨트롤 타워이다. 여기서 이벤트루프라는 말은 무한 루프를 돌며 매 루프마다 작업들을 하나씩 실행시키는 로직이기 때문이다.

   여러 작업을 동시에 실행하게 하며, 하나의 작업이 완료될 때까지 다른 작업들이 차례로 실행된다.

2. loop.run_until_complete(main())

   run_untill_complete(coroutine)은 코루틴 (비동기 함수)가 완료될 때까지 이벤트 루푸를 실행한다. 즉 main 코루틴이 완료될 때까지 이벤트 루프가 계속 실행된다. 그리고 다 끝나면 이벤트 루프가 멈춘다. 반대로 말하면 모든 비동기 작업이 끝날 때까지 이벤트 루프가 멈추지 않는다.

> **추가 참고: `asyncio.run()` 대체**
>
> Python 3.7부터는 **`asyncio.run()`**을 사용하는 것이 권장됩니다.
>
> 이것은 **이벤트 루프를 자동으로 생성 및 관리**해주기 때문에 코드가 더 간단해집니다.
>
> ```
> import asyncio
>
> async def main():
>     print("Hello")
>     await asyncio.sleep(1)
>     print("World")
>
> if __name__ == "__main__":
>     asyncio.run(main())  # 간결하게 이벤트 루프 실행
> ```

 Selenium & Playwright의 공통/차이점

필요한 라이브러리 불러오기.

```
import asyncio
import json
import random
import time
from playwright.async_api import async_playwright
import nest_asyncio
```

- `asyncio`: 비동기 함수 실행을 지원.
- `json`: 네트워크 요청/응답 데이터를 JSON으로 저장하기 위해 사용.
- `random`: 지연 시간을 임의로 설정할 때 사용.
- `time`: 시간 지연이나 타이머 설정에 필요.
- `playwright.async_api`: 웹 브라우저 제어를 위한 Playwright의 비동기 API.
- `nest_asyncio`: Jupyter 환경에서 이벤트 루프 충돌을 방지하기 위한 모듈.

```
nest_asyncio.apply()
```

- **Jupyter 환경에서의 충돌 방지**를 위해 이벤트 루프를 수정합니다.

```
SEM_MAX = 40
MAX_USERS = 40
URL = "https://dydtkddl.github.io/WebResearch/2"
URL = "http://webresearch.pythonanywhere.com/"
semaphore = asyncio.Semaphore(SEM_MAX)
```

- `SEM_MAX`: **동시에 실행할 최대 사용자 수**를 제한 (최대 40명).
- `MAX_USERS`: **총 사용자 수**를 설정합니다 (40명 시뮬레이션).
- `URL`: 접속할 웹사이트 주소를 설정합니다.
  - **세마포어**(`Semaphore`): 특정 동작의 동시 실행 수를 제한하는 동기화 도구입니다. 여기서는 최대 40명의 사용자만 동시에 시뮬레이션 됩니다.

사용자 시뮬레이션 함수 정의

```
async def user_simulation(user_id):
    """사용자 시뮬레이션 함수"""
    ### 01 : 세마포어로 동시 실행 수 제한
      async with semaphore:
        await asyncio.sleep(random.uniform(2, 5)) ## 시뮬레이션 시작 전
        ## 2~5초의 지연을 무작위로 추가
        print(f"사용자 {user_id} 시뮬레이션 시작")


    ### 02 : Playwright 브라우저를 비동기 방식으로 실행
        async with async_playwright() as p:
            browser = await p.chromium.launch(headless=False)
            context = await browser.new_context(
                bypass_csp=True,
                no_viewport=True,
                ignore_https_errors=True,
                storage_state=None
            )
            page = await context.new_page()
                network_log = []

        def log_request(request):
            """HTTP 요청 로그"""
            network_log.append({
                "type": "request",
                "url": request.url,
                "method": request.method,
                "headers": dict(request.headers),
            })
        async def log_response(response):
            """HTTP 응답 로그"""
            try:
                body = await response.body()
                network_log.append({
                    "type": "response",
                    "url": response.url,
                    "status": response.status,
```

```
                    "headers": dict(response.headers),
                    "body": body.decode("utf-8", errors="ignore")
                })
            except Exception as e:
                print(f"사용자 {user_id}: 응답 로깅 중 오류 - {e}")
    ### 03 : 페이지 요청/응답에 대한 이벤트를 연결
        page.on("request", log_request)
        page.on("response", log_response)

        try:
            await page.goto(URL)
            ## 엔터키 함수
            async def press_enter():
                await page.keyboard.press("Enter")
                print(f"사용자 {user_id}: 엔터 입력 완료")
                await asyncio.sleep(random.uniform(0.5, 1.5))
            ## 다음 버튼 함수
            async def click_button(button_id):
                await page.click(f'#{button_id}')
                print(f"사용자 {user_id}: 버튼 {button_id} 클릭 완료")
                await asyncio.sleep(random.uniform(1, 2))
            ## 라디오 버튼 선택 함수
            async def select_radio_button(value):
                await page.click(f'input[name="jspsych-survey-multi-choice-response-0"][v
alue="{value}"]')
                print(f"사용자 {user_id}: 라디오 버튼 {value} 선택 완료")
                await asyncio.sleep(random.uniform(2, 5))

            await press_enter()
            await click_button('jspsych-instructions-next')
            await click_button('jspsych-instructions-next')
            await click_button('jspsych-instructions-next')
            await press_enter()

            await select_radio_button('C')
            await click_button('jspsych-survey-multi-choice-next')
            await press_enter()

            await press_enter()
            await select_radio_button('F')
            await click_button('jspsych-survey-multi-choice-next')
            await press_enter()
            await press_enter()

            for _ in range(8):
                await select_radio_button('F')
                await click_button('jspsych-survey-multi-choice-next')
        ##
        finally:
            with open(f"user_{user_id}_network_log.json", "w") as f:
                json.dump(network_log, f, indent=4)

            await browser.close()
            print(f"사용자 {user_id} 시뮬레이션 완료")

## 모든 사용자 시뮬레이션을 비동기적으로 실행합니다.
# 이때 main함수가 코루틴이기때에 async식별자가 앞에 붙는다.
async def main():
```

```
    tasks = [user_simulation(user_id) for user_id in range(MAX_USERS)]
    await asyncio.gather(*tasks)

## 메인 프로그램을 실행합니다. 이벤트 루프를 사용해 main()을 실행합니다.
if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

- 사용자별 **시뮬레이션 함수**.
- `async with semaphore` : 세마포어를 사용하여 최대 `SEM_MAX`만큼 동시 실행이 가능합니다.
- **Playwright 브라우저를 비동기 방식**으로 실행
  - `chromium.launch()` : Chromium 브라우저를 실행합니다.
  - `headless=False` : 브라우저가 **화면에 표시**되도록 설정합니다.
  - `bypass_csp=True` : 콘텐츠 보안정책(CSP) 우회를 활성화.
  - `ignore_https_errors=True` : HTTPS 오류를 무시합니다.
- **페이지 요청/응답에 대한 이벤트**를 연결합니다. 각 요청과 응답이 발생할 때마다 로그 함수가 호출됩니다.