

# 비동기 데이터 로딩 및 에러 처리



## 수업내용

- UI 변경 감지 및 상태 검증을 자동화하여 React 프로젝트의 안정성을 테스트합니다.
- 동적 데이터가 정상적으로 로드되는지 검증합니다.
- Selenium을 활용하여 크롤링한 데이터를 JSON 및 CSV 형식으로 저장하고 불러올 수 있습니다.
- 테스트 과정에서 발생하는 오류를 분석하고, try-except 와 logging을 활용하여 예외를 처리합니다.
- 스크린샷 비교 및 로그 분석을 통해 테스트 결과를 시각적으로 검토하고, 실패 원인을 파악할 수 있습니다.

# 일정 CRU 자동화 - 일정 생성, 읽기, 수정



## CRUD

- Create(생성): 사용자가 일정을 추가하는 기능
- Read(조회): 생성된 일정을 확인하는 기능
- Update(수정): 기존 일정을 수정하는 기능
- Delete(삭제): 필요 없는 일정을 삭제하는 기능



- 일정 생성 → 생성된 일정 확인 (Create.tsx)
- 일정 읽기 → 특정 일자의 일정 가져오기 (Detail.tsx)
- 일정 수정 → 변경된 일정 확인 (Edit.tsx)
- 일정 삭제 → 해당 프로젝트에 없으므로 테스트 X



# 로그인 자동화에서 많이 쓰는 기본 시나리오

로그인

이메일: test33@example.com

비밀번호: .....

로그인 회원가입

- 로그인 자동화는 QA 자동화에서 가장 많이 사용되는 시나리오
- 페이지 이동 → 로그인 폼 대기 → 이메일·비번 입력 → 로그인 버튼 클릭으로 구성

```
driver.get("http://localhost:3000/login")
wait.until(EC.presence_of_element_located((By.NAME, "email"))).send_keys("test33@example.com")
```

- 페이지 로딩 속도 문제 대비 'WebDriverWait'으로 대기 처리 필수
- 로그인 성공 여부는 URL 변화 감지 또는 특정 요소 등장 여부로 확인

```
wait.until(lambda driver: "login" not in driver.current_url)
```

# 캘린더, 시간 입력 시, 강제 주입하는 방법

제목:

설명:

장소:

날짜:  연도-월-일

시작 시각:  -- --:--

종료 시각:  -- --:--

참가자:  사용자 검색

생성

```
driver.execute_script("")  
const input = arguments[0];  
input.setAttribute('value', arguments[1]);  
input.dispatchEvent(new Event('input', { bubbles: true }));  
input.dispatchEvent(new Event('change', { bubbles: true }));  
"", date_input, "2025-03-01")
```

- 캘린더, 시간 선택 필드는 'send\_keys()'로 입력이 막힌 경우 많음
- 이런 경우 JavaScript로 'value' 직접 주입 방식 활용
- 값만 주입하면 반영 안 될 수 있으므로 'input', 'change' 이벤트 강제 발생 처리
- Selenium 'execute\_script()' 활용해 강제 입력 방식으로 우회
- 캘린더 외에도 커스텀 슬라이더, 파일 업로드 등 다양한 UI에 응용 가능

# 일정 생성 성공 확인과 에러 원인 분석

localhost:3000/detail/67c33dc6bb221bcbd11cfe90

```
if "detail" in driver.current_url:  
    print("✓ 일정 생성 테스트 성공!")  
else:  
    print("✗ 일정 생성 실패 (400 Bad Request 가능)")
```

- 일정 생성 성공 여부는 URL 변화 감지 또는 성공 메시지 등장 확인 방식 활용
- 페이지 이동 기반 서비스는 URL 변화 감지가 가장 확실

제목:	일정 생성됐니?
설명:	잘 되겠지?
장소:	우리집
날짜:	Thu Mar 13 2025 02:03:00 GMT+0900 (한국 표준시)
시작 시각:	02:03
종료 시각:	02:06
참가자:	

[수정](#)

- 화면상 메시지 기반 서비스는 특정 요소 등장 여부로 확인 가능
- 실패 시 에러 원인별 로그 출력으로 디버깅 효율 개선
- 400 Bad Request, 500 에러, 필수 입력 누락 등 주요 원인 대비 필요



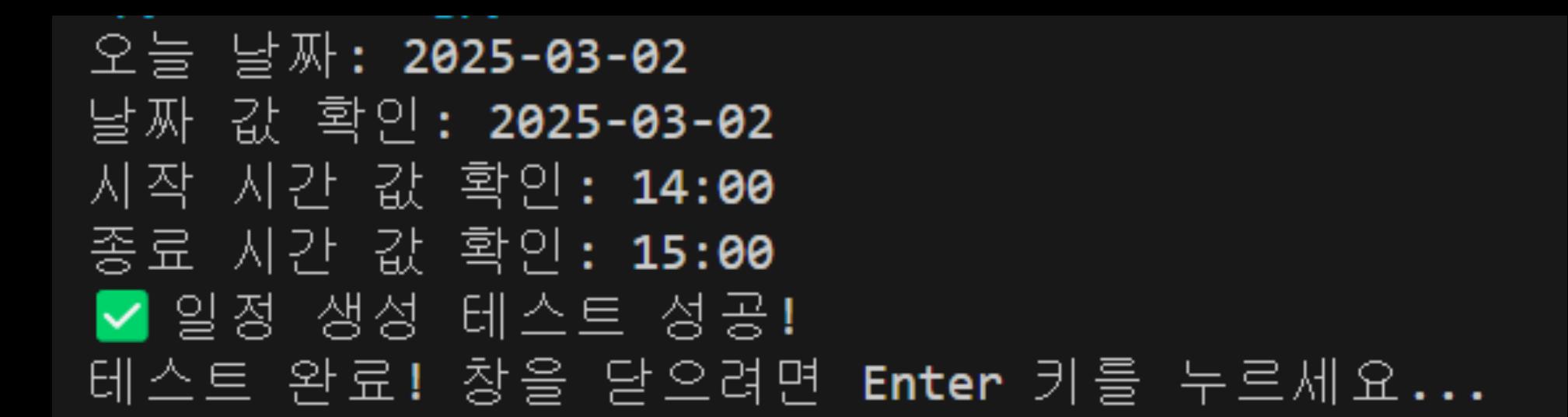
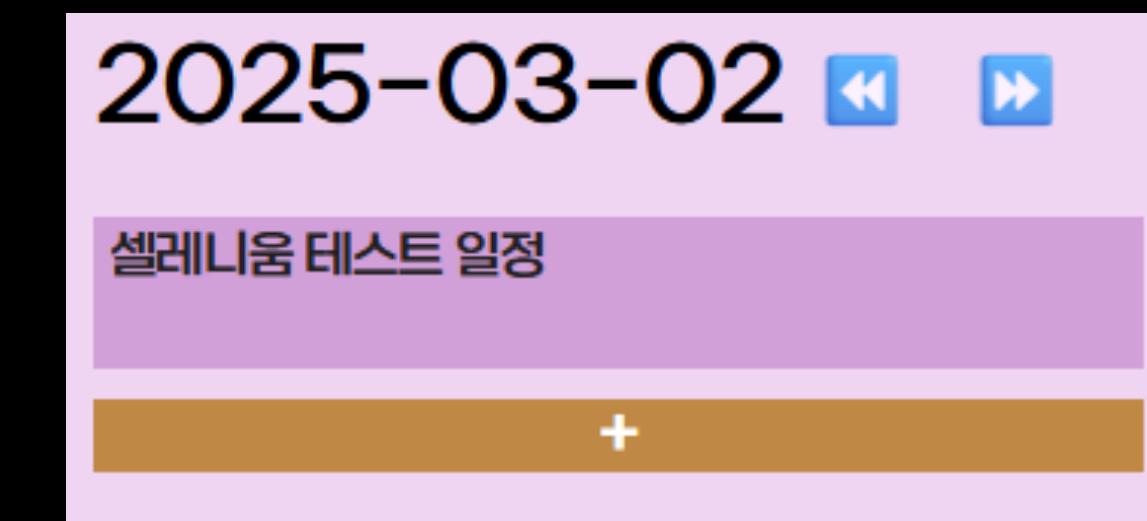
# 실습1 : 일정 생성 자동화 테스트

## 실습 목표

- 일정 생성 페이지에서 각 입력 필드에 자동으로 값 입력
- 캘린더·시간 입력 필드는 JavaScript로 강제 값 주입 방식 실습
- 일정 생성 후 성공 여부 확인 및 로그 출력 방식 학습

## 실습 순서

1. 일정 생성 페이지 이동
2. 제목, 설명, 장소 입력 자동화
3. 날짜, 시작시간, 종료시간 JavaScript 강제 입력
4. 일정 생성 버튼 클릭
5. 성공 여부 확인 (URL 변화 감지)
6. 성공·실패 메시지 출력



# ☰ 오늘 날짜의 일정을 자동으로 찾는 방법

## 오늘 날짜 찾기 (기본 흐름)

- 일정 목록에서 오늘 날짜를 기준으로 동적 XPath 생성
- 오늘 날짜는 매일 변경되므로, 실행 시마다 자동으로 날짜 계산
- 날짜 요소를 찾은 후, 해당 날짜 아래에 있는 일정 목록 영역 탐색

```
today = datetime.date.today().strftime("%Y-%m-%d")
```

## 동적 XPath 활용

- '//div[text()='2025-03-01']}'처럼 날짜가 들어간 XPath 구성
- 오늘 날짜를 동적으로 계산해 XPath에 삽입
- 날짜가 바뀌어도 항상 최신 일정 탐색 가능

```
date_div = wait.until(EC.presence_of_element_located(  
    (By.XPATH, f"//div[text()='{today}']"))  
)
```

## 일정 목록 접근

- 날짜 요소의 바로 다음 형제 요소에서 일정 목록 컨테이너 탐색
- 첫 번째 일정 요소를 선택해 클릭 준비

```
schedule_container = date_div.find_element(By.XPATH, "following-sibling::div")  
first_schedule = schedule_container.find_element(By.XPATH, "./div")
```

# ☰ 오늘 일정 클릭 후 상세 페이지 이동 확인

## 일정 클릭 처리

- 오늘 날짜 아래에 있는 첫 번째 일정 요소 선택 및 클릭
- 클릭 동작 이후 페이지 이동 발생

```
first_schedule.click()
```

## 상세 페이지 URL 구조 이해

- 상세 페이지 URL: '/detail/일정ID' 형태로 구성
- URL에서 마지막 값이 일정 ID로 사용됨
- 추출한 일정 ID는 이후 수정 페이지로 이동할 때 활용

```
detail_url = driver.current_url  
print(f"🔗 상세 페이지 URL: {detail_url}")
```

## 상세 페이지 진입 확인

- 클릭 후 현재 URL 확인으로 상세 페이지 진입 여부 체크
- 단순 요소 확인보다 URL 변화 감지가 더 안정적

```
schedule_id = detail_url.split("/")[-1]  
print(f">ID 추출된 일정 ID: {schedule_id}")
```



## 일정 수정 페이지에서 제목, 설명 수정하기

```
edit_url = f"http://localhost:3000/edit/{schedule_id}"
driver.get(edit_url)
```

### 수정 페이지 진입

- 추출한 일정 ID를 활용해 수정 페이지 URL 동적 구성
- 구성된 URL로 이동해 수정 화면 진입

```
title_input = wait.until(EC.presence_of_element_located((By.NAME, "title")))
title_input.clear()
title_input.send_keys("업데이트된 회의 제목")
```

### 기존 데이터 삭제 및 새로운 내용 입력

- 제목과 설명 입력 필드는 기존 값이 남아 있으므로, clear()로 기존 값 삭제 후 새 값 입력
- 기존 값 위에 덮어쓰기하면 오타나 누락 발생 위험 있음
- 수정 가능한 모든 입력 필드에 공통 적용 가능

### 입력 완료 후 다음 단계로 준비

- 제목·설명 수정 후, 다음 단계인 "수정 완료 버튼 클릭" 대기 상태

# 수정 완료 버튼 클릭과 성공 여부 확인

```
update_button = driver.find_element(By.XPATH, "//button[text()='수정 완료'])")
update_button.click()
```

## 수정 완료 버튼 클릭

- 모든 수정 사항 입력 후, 수정 완료 버튼 클릭
- 클릭 동작으로 수정 요청 전송

```
time.sleep(3)
if "detail" in driver.current_url:
    print("✓ 일정 수정 자동화 테스트 성공!")
else:
    print("✗ 일정 수정 실패")
```

## 성공 여부 확인 기준

- 수정 성공 시, 상세 페이지로 자동 이동
- 이동한 페이지의 URL에 'detail'이 포함되는지 확인해 성공 여부 판단

## URL 변화 기반 성공 판단 패턴

- 화면 요소 등장 여부가 아닌, URL 변화 감지 방식으로 성공 여부 판단
- 페이지마다 로직이 다를 수 있어 가장 범용적으로 사용하는 방식

## 디버깅용 로그 출력

- 성공 여부에 따라 콘솔에 ✓ 또는 ✗ 메시지 출력
- 문제 발생 시 로그 기반 원인 분석 가능



## 실습 2 : 일정 수정 자동화 테스트

### 실습 목표

- Selenium을 활용하여 기존 일정의 제목과 설명을 자동으로 수정하는 방법을 학습
- 일정 수정 완료 후, 수정 성공 여부를 URL 변화로 확인하는 방법 익히기
- 실무에서 반복적으로 사용하는 동적 페이지 접근 및 데이터 수정 자동화 패턴 습득

### 실습 순서

1. 로그인 페이지 접속 및 로그인 완료
2. 특정 일정 상세 페이지로 이동
3. 수정 페이지로 이동
4. 제목과 설명 수정
5. 수정 완료 버튼 클릭 및 성공 여부 확인

<b>제목:</b>	셀레니움 테스트 일정업데이트된 회의 제목
<b>설명:</b>	새로운 회의 내용입니다.
<b>장소:</b>	회의실 A
<b>날짜:</b>	Sun Mar 02 2025 14:00:00 GMT+0900 (Korean Standard Time)
<b>시작 시각:</b>	14:00
<b>종료 시각:</b>	15:00
<b>참가자:</b>	

수정

오늘 날짜: 2025-03-02  
선택한 첫 번째 일정: 셀레니움 테스트 일정  
동적 추출된 상세 페이지 URL: <http://localhost:3000/detail/67c34137bb221bcbd11cff00>  
일정 수정 자동화 테스트 성공!  
테스트 완료 - 창을 닫으려면 Enter...

# 월 변경 자동화 개요 및 현재 월 정보 읽기

## 월 변경 테스트의 목적

- 일정 화면에서 이전 달 / 다음 달 로 이동하는 기능을 자동화로 검증
- 월이 변경될 때 올바른 월 정보가 표시되는지 확인
- 월별 일정이 정상적으로 표시되는지 함께 검증

## 현재 월 정보 확인 방법

- 화면에 표시된 월-연도 정보를 직접 읽어와서 현재 월을 확인
- 단순 텍스트 읽기 방식 대신, JavaScript 실행으로 텍스트 노드 직접 탐색
- 동적 렌더링 환경에서도 안정적으로 월 정보 추출 가능

일	월	화	수
2 업데이트된 회의 제목z	3	4	5
9	10	11	12

## 주요 포인트

- React 기반 화면에서는 월 정보가 DOM 구조 깊숙이 포함될 수 있어, 단순 'getText()' 대신 JavaScript로 접근하는 방식이 더 효과적
- 서비스 특성에 따라 월-연도 표시 위치가 다를 수 있으므로, 현재 서비스에 맞는 탐색 방식 적용 필요



# 이전 달 이동과 월 변경 확인 자동화

## 이전 달 이동 처리 방법

- 일정 화면에서 이전 달 버튼(◀)을 클릭해 한 달 이전으로 이동
- 월이 변경되면, 상단에 표시되는 월 정보를 다시 읽어 변경 여부 확인

## 월 변경 후 현재 월 정보 재확인

- 월 변경 버튼 클릭 후, 일정 데이터가 로드될 때까지 잠시 대기
- 'WebDriverWait'을 활용해 페이지 업데이트 완료까지 기다림
- 월 정보를 다시 읽어 원하는 월로 제대로 변경되었는지 확인

```
prev_month_button = wait.until(EC.element_to_be_clickable(  
    (By.XPATH, "//button[contains(text(), '◀')]"))  
prev_month_button.click()  
  
# 월 변경 후 업데이트 시간 고려해 약간 대기  
time.sleep(2)  
  
# 다시 현재 월 정보 읽기  
current_month_text = driver.execute_script(  
    "const divs = Array.from(document.querySelectorAll('div'));  
    for (const div of divs) {  
        if (div.innerText.includes('2025')) {  
            for (const node of div.childNodes) {  
                if (node.nodeType === Node.TEXT_NODE) {  
                    return node.nodeValue.trim();  
                }  
            }  
        }  
    }  
    return null;  
")
```

# 월 변경 후 일정 데이터 확인 및 검증

```
date_div = wait.until(EC.presence_of_element_located(  
    (By.XPATH, f"//div[text()='2025-03-01']"))  
)  
  
# 해당 날짜 아래에 있는 일정 컨테이너 찾기  
schedule_container = date_div.find_element(By.XPATH, "following-sibling::div")  
  
# 일정 목록 탐색  
schedules = schedule_container.find_elements(By.CLASS_NAME, "preview")  
print(f"📅 2025-03-01 일정 개수: {len(schedules)}")
```

## 월 변경 후 일정 데이터 확인 방법

- 월 변경 후 해당 월의 일정이 정상적으로 표시되는지 확인
- 일정은 날짜별로 표시되며, 각 날짜 아래에 일정 리스트가 존재하는 구조
- 특정 날짜의 일정 데이터가 몇 개인지 확인하고 출력

## 일정이 없는 경우 처리 예시

```
if len(schedules) == 0:  
    print("🔍 해당 날짜는 일정이 없습니다.")
```

# ☰ 이전/다음 달 이동 후 UI 및 일정 로드

## 월 변경 후 데이터 확인 코드 분석

- 월 변경 후 'current\_month\_text'를 다시 읽어서 최종 확인 로그를 출력
- 이 단계에서 "원하는 월로 정상 변경되었는지" 검증

```
print(f"📅 최종 확인 - 현재 월: {current_month_text}")
```

- 해당 월의 일정 데이터가 몇 개인지 **개수 출력**
- 이 값이 예상과 다르면, 일정 조회 실패로 판단 가능

```
if len(schedules) == 0:  
    print("🔍 해당 월은 일정이 없습니다.")
```

- 일정 데이터가 없는 경우, **예외 상황 처리**
- 일정이 없는 것도 정상적인 상황이므로, 별도 안내 메시지를 출력
- 일정 데이터가 하나라도 있으면, 정상적으로 데이터가 표시된 것으로 판단

```
else:  
    print("✅ 일정 데이터 정상 확인")
```



## 실습3 : 월 변경 자동화 테스트

### 실습 목표

- Selenium을 이용해 월 변경 버튼(◀, ▶)을 클릭
- 월 정보가 정상적으로 변경되는지 확인
- 월별 일정 데이터가 제대로 표시되는지 자동화로 검증

### 실습 순서

1. 일정 페이지로 이동
2. 현재 월 정보 확인
3. 이전 달 버튼 클릭 및 월 변경 확인

The screenshot shows a calendar interface. At the top left, there are two buttons: '일정 열람' (View Calendar) and '일정 생성' (Create Event). Below these buttons, the date '2025-02-28' is displayed, followed by left and right navigation arrows. A purple rectangular box highlights the text '셀레니움 테스트 일정' (Selenium Test Schedule). To the right of the date, there is a table:

일	월
2	3

Below the table, a dark gray status bar displays the following information:

- 오늘 날짜: 2025-03-02
- 현재 월 표시: 2025-03-02
- 변경 후 월 표시: 2025-03-01
- 변경 후 월 표시: 2025-02-28
- ✓ 2025년 2월로 이동 완료!
- ✓ 2월 28일 클릭 완료!

At the bottom of the status bar, it says '확인 후 종료하려면 Enter를 누르세요...' (Press Enter to exit).



## 퀴즈 타임



QUIZ 1번부터 8번을 풀어봐요!

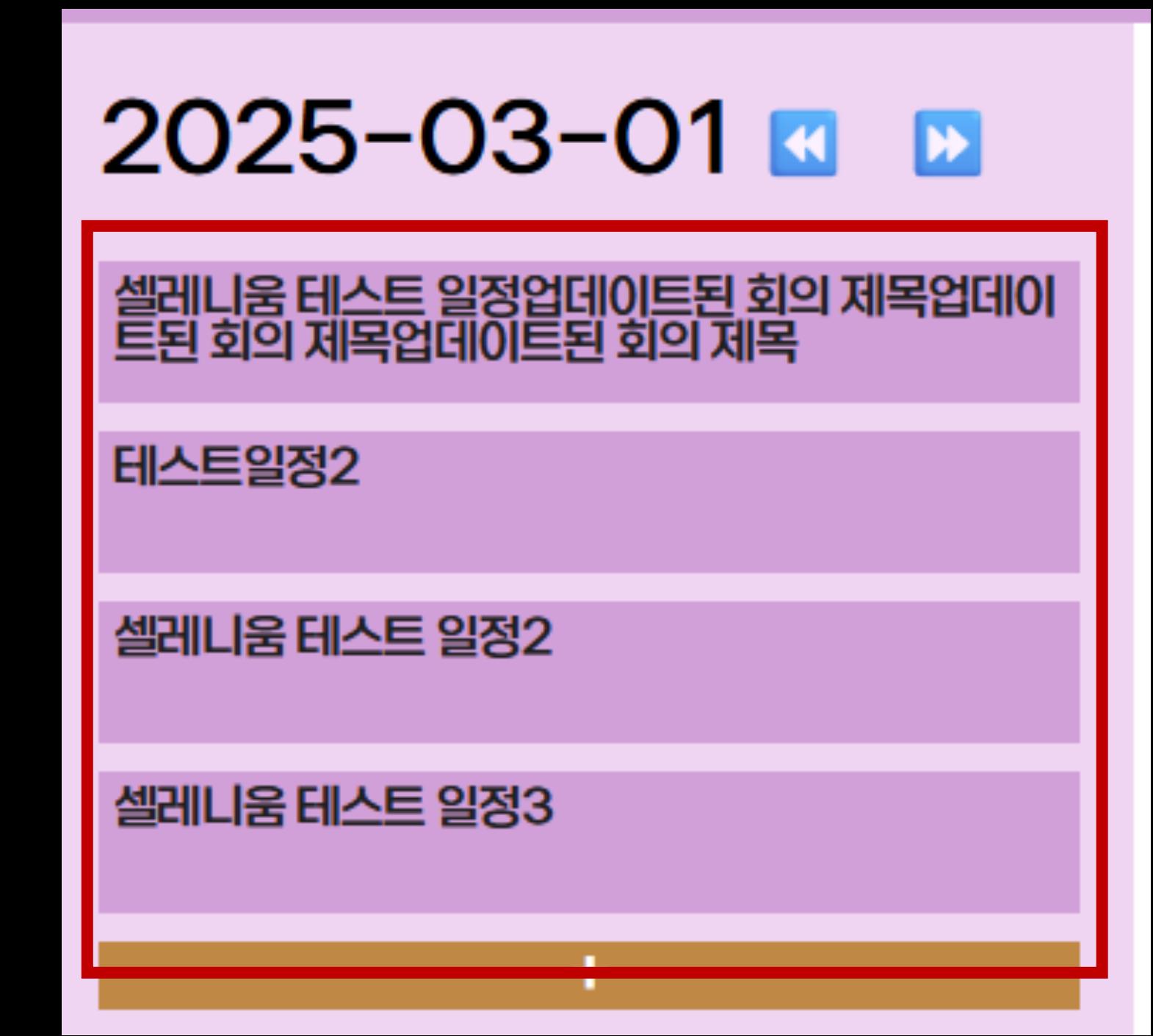
# 동적 일정 데이터 크롤링 흐름 이해하기

## 일정 크롤링 흐름 개요

- Selenium을 이용한 일정 크롤링은 단순 페이지 수집이 아닌, 동적 데이터 처리가 필요
- 날짜별로 일정이 다르고, 일정이 있는 날과 없는 날의 UI 구조도 다를 수 있음
- 일정 제목과 상세 링크를 수집해 JSON으로 저장하는 것이 목표

## 동적 크롤링의 핵심 포인트

- 일정은 화면 렌더링 후 동적으로 생성되는 경우가 많아, 요소 대기 처리가 필수
- 날짜와 일정 목록이 HTML 구조적으로 연결되어 있어 동적 관계를 정확히 파악해야 함
- 일정 클릭 시 페이지 이동 후, 뒤로 가기(back) 동작을 활용해 반복 크롤링 진행





# 날짜 동적 탐색 및 일정 컨테이너 접근 방법

## 날짜 탐색의 중요성

- 크롤링 시 오늘 날짜를 기준으로 데이터 수집하는 경우가 많음
- 날짜는 매일 바뀌기 때문에, 날짜 탐색은 동적 탐색 방식이 필요

## 날짜 요소 동적 탐색 방법

- Python에서 현재 날짜를 구해 문자열로 저장
- XPath에 날짜를 직접 삽입해 오늘 날짜에 해당하는 요소 찾기

```
today = datetime.today().strftime('%Y-%m-%d')
```

```
date_div = wait.until(EC.presence_of_element_located(  
    (By.XPATH, f"//div[text()='{today}']"))  
)
```



# 날짜별 일정 목록 동적 탐색

- 일정 크롤링은 날짜 아래에 있는 일정 목록 컨테이너를 기준으로 탐색
- 날짜는 고정 위치가 아니므로, 오늘 날짜를 기준으로 요소 위치를 찾는 방식 필요
- 오늘 날짜 요소를 찾은 뒤, 바로 아래 형제 요소에서 일정 목록 탐색

## 날짜 요소 찾기

```
date_div = wait.until(EC.presence_of_element_located(  
    (By.XPATH, f"//div[text()='{today}']"))  
)
```

## 날짜 요소 아래 일정 컨테이너 탐색

```
schedule_container = date_div.find_element(By.XPATH, "following-sibling::div")
```



## 일정 클릭과 상세 페이지 이동 처리

- 일정 크롤링은 일정 목록에서 제목만 가져오는 게 아니라, 상세 정보 확인을 위해 클릭이 필요
- 각 일정은 클릭 시 상세 페이지로 이동하는 구조
- 상세 페이지 URL을 가져와서, 일정 데이터와 함께 저장하는 방식으로 구성

### 첫 번째 일정 클릭

```
first_schedule = schedule_container.find_elements(By.XPATH, "./div")[0]
first_schedule.click()
```

### 상세 페이지 URL 수집

```
detail_url = driver.current_url
```

# 상세 페이지 URL 수집과 뒤로가기 처리

- 일정 클릭 후 이동한 상세 페이지 URL은 일정 데이터와 함께 저장
- URL 수집은 'driver.current\_url'로 바로 가능

## **stale element 오류 방지**

- 한 번 찾은 요소는 페이지 이동 후 다시 사용할 수 없는 경우가 많음
- 일정 클릭 → 상세 페이지 이동 → 뒤로가기 후
- 목록 전체를 새로 탐색하는 방식으로 오류 예방

## 상세 페이지 URL 저장

```
detail_url = driver.current_url
schedules.append({
    "title": schedule_title,
    "link": detail_url
})
```

## 상세 페이지 URL 수집

```
driver.back()
```



# 수집한 데이터를 JSON 파일로 저장하기

## 일정 데이터 저장 방식

- 크롤링한 일정 데이터는 리스트 형태로 관리
- 각 일정은 다음 정보 포함
  - title: 일정 제목
  - link: 상세 페이지 URL

```
[  
    {"title": "팀 회의", "link": "http://localhost:3000/detail/abc123"},  
    {"title": "프로젝트 리뷰", "link": "http://localhost:3000/detail/def456"}  
]
```

## Python JSON 저장 방법

- Python의 json 모듈 활용
- ensure\_ascii=False: 한글 깨짐 방지
- indent=4: 보기 좋은 들여쓰기 적용

```
with open("schedules.json", "w", encoding="utf-8") as f:  
    json.dump(schedules, f, ensure_ascii=False, indent=4)
```

# 저장된 데이터(JSON) 불러오기와 재활용

## 저장된 일정 데이터 불러오기

- 크롤링한 일정은 JSON 파일로 저장됨
- 저장된 파일은 다음과 같이 읽어올 수 있음

```
with open("schedules.json", "r", encoding="utf-8") as f:  
    schedules = json.load(f)
```

## 저장 데이터 재활용 방법

다음과 같은 작업에서 유용하게 활용 가능

- 일정 존재 여부 확인
- 특정 일정 바로 열기 (URL 바로 접근)
- 기존 데이터에 신규 일정 추가 저장

```
for schedule in schedules:  
    print(f"{schedule['title']} - {schedule['link']})")
```

## 일정 데이터 저장의 중요성

- 크롤링한 데이터는 단순 저장이 아닌, 이후 관리와 재활용까지 고려한 구조가 필요
- 일정 데이터는 날짜별·주제별로 구분해 저장하면 관리 효율 극대화
- 저장 포맷과 저장 방식도 목적에 맞게 선택

### 날짜별 파일 저장 방식

저장 방식	장점	단점
날짜별 파일 저장	날짜별로 관리 쉬움, 필요한 날짜만 불러오기 편리	파일 수가 많아질 수 있음
한 파일에 계속 추가	파일 관리 쉬움	데이터 양이 많아질 경우 성능 저하 가능

### 데이터 중복 관리 방법

- 일정 제목과 URL 기준으로 중복 검사 후 저장

```
if any(s['link'] == new_schedule['link'] for s in schedules):
    print("이미 저장된 일정입니다.")
else:
    schedules.append(new_schedule)
```

# ⚠ JSON 저장 시 오류 처리와 로그 관리

## 크롤링 데이터 저장 시 발생 가능한 문제

- 데이터 누락
  - : 크롤링 도중 브라우저 종료, 네트워크 문제로 일부 일정 누락 가능
- JSON 저장 실패
  - : 권한 문제, 디스크 용량 부족, 인코딩 문제 등으로 저장 실패 가능
- 잘못된 데이터 저장
  - : 비정상 크롤링 데이터가 저장되어 JSON 구조 깨짐 가능

## 일정 제목·링크 값이 모두 존재하는지 확인

```
if not schedule['title'] or not schedule['link']:
    print("⚠ 유효하지 않은 데이터는 저장하지 않습니다.")
    continue
```

## 데이터 중복 여부 확인 (URL 기준 중복 체크)

```
if any(s['link'] == schedule['link'] for s in schedules):
    print(f"⚠ 이미 저장된 일정: {schedule['title']}")
    continue
```



# 데이터 저장을 위한 JSON 메서드

## Python에서 제공하는 기본 메서드

메서드	설명	예시
<b>json.dump()</b>	파이썬 객체 → JSON 파일 저장	<code>json.dump(data, file)</code>
<b>json.dumps()</b>	파이썬 객체 → JSON 문자열 변환	<code>json_str = json.dumps(data)</code>
<b>json.load()</b>	JSON 파일 → 파이썬 객체 불러오기	<code>data = json.load(file)</code>
<b>json.loads()</b>	JSON 문자열 → 파이썬 객체 변환	<code>data = json.loads(json_str)</code>



### JSON 데이터 활용 포인트

- 크롤링 데이터 저장 및 재활용 필수 포맷
- 웹 서비스 연동 시에도 JSON 포맷 그대로 활용 가능
- 일정 데이터 외에도, 크롤링 로그·통계 저장에도 사용



# 실습 4 : 일정 데이터 크롤링 후 JSON 저장

## 실습 목표

- Selenium을 이용해 오늘 날짜의 일정 목록을 크롤링
- 각 일정의 제목과 상세 페이지 링크를 수집
- JSON 파일로 저장하는 자동화 코드를 완성

## 실습 순서

1. 일정 목록 탐색
2. 일정별 상세 페이지 이동 및 정보 수집
3. 수집한 데이터 JSON 저장
4. 오류 및 중복 일정 처리
5. 저장 완료 후 데이터 확인

```
오늘 날짜: 2025-03-02
일정 발견: 업데이트된 회의 제목z
상세 페이지 URL: http://localhost:3000/detail/67c34137bb221bcbd11cff00
일정 크롤링 완료 및 schedules.json 저장 완료!
[
  {
    "title": "업데이트된 회의 제목z",
    "link": "http://localhost:3000/detail/67c34137bb221bcbd11cff00"
  }
]
```

```
{ schedules.json > ...
1 [ 
2 { 
3   "title": "업데이트된 회의 제목z",
4   "link": "http://localhost:3000/detail/67c34137bb221bcbd11cff00"
5 } 
6 ] }
```



# 자동화 테스트에서 에러 처리 기초

## 자동화 테스트에서 에러 처리 왜 필요할까?

- 웹 자동화 환경은 예측 불가능한 요소가 많음
  - 페이지 로드 지연
  - 예상치 못한 팝업
  - 요소 로드 실패
  - 네트워크 문제
- 에러 발생 시 테스트 중단 대신 정확한 원인 파악과 로그 기록 필수
- 특히 반복 테스트 환경에서는 안정성 확보를 위해 에러 처리 및 복구 로직이 반드시 필요

## 주요 에러 유형

### 1. NoSuchElementException

- 요소를 찾지 못할 때 발생
- XPath, CSS 선택자 오류
- 페이지 구조 변경 등

### 2. TimeoutException

- 요소가 기한 내 나타나지 않을 때 발생
- 네트워크 지연, 서버 응답 지연 원인

### 3. WebDriverException

- 드라이버 문제 (브라우저 종료 등)

```
Traceback (most recent call last):
  File "d:\필요\python\241121\test.py", line 15, in <module>
    wait.until(EC.presence_of_element_located((By.NAME, "emailf"))).send_keys("test33@example.com")
    ~~~~~~
  File "d:\필요\python\241121\.venv\Lib\site-packages\selenium\webdriver\support\wait.py", line 146, in until
    raise TimeoutException(message, screen, stacktrace)
selenium.common.exceptions.TimeoutException: Message:
Stacktrace:
RemoteError@chrome://remote/content/shared/RemoteError.sys.mjs:8:8
WebDriverError@chrome://remote/content/shared/webdriver/Errors.sys.mjs:197:5
NoSuchElementException@chrome://remote/content/shared/webdriver/Errors.sys.mjs:527:5
dom.find/</<@chrome://remote/content/shared/DOM.sys.mjs:136:16
```

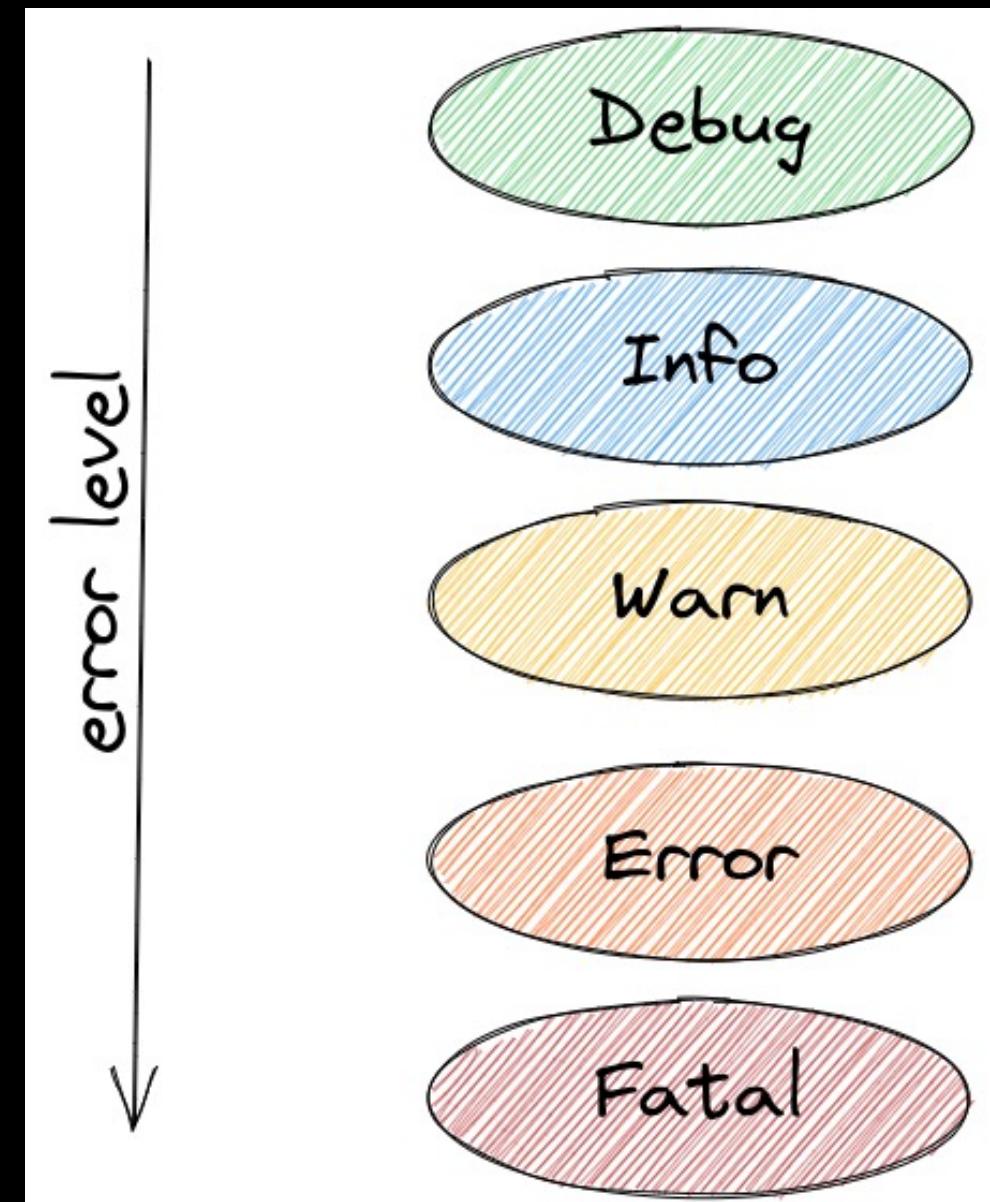


## 왜 로그가 중요할까?

- 자동화 테스트는 무인 실행되는 경우가 많음
- 오류 발생 시 원인 파악을 위한 기록(로그) 필수
- 개발 환경, 테스트 환경, 서버 환경 등 환경에 따라 발생하는 에러 원인도 달라질 수 있음

## Python logging 기본 구성

로그 레벨	의미
DEBUG	상세한 디버그 정보
INFO	일반적인 실행 흐름 정보
WARNING	경고 메시지 (심각하지 않지만 주의 필요)
ERROR	에러 발생 상황 기록
CRITICAL	치명적인 에러 상황 기록





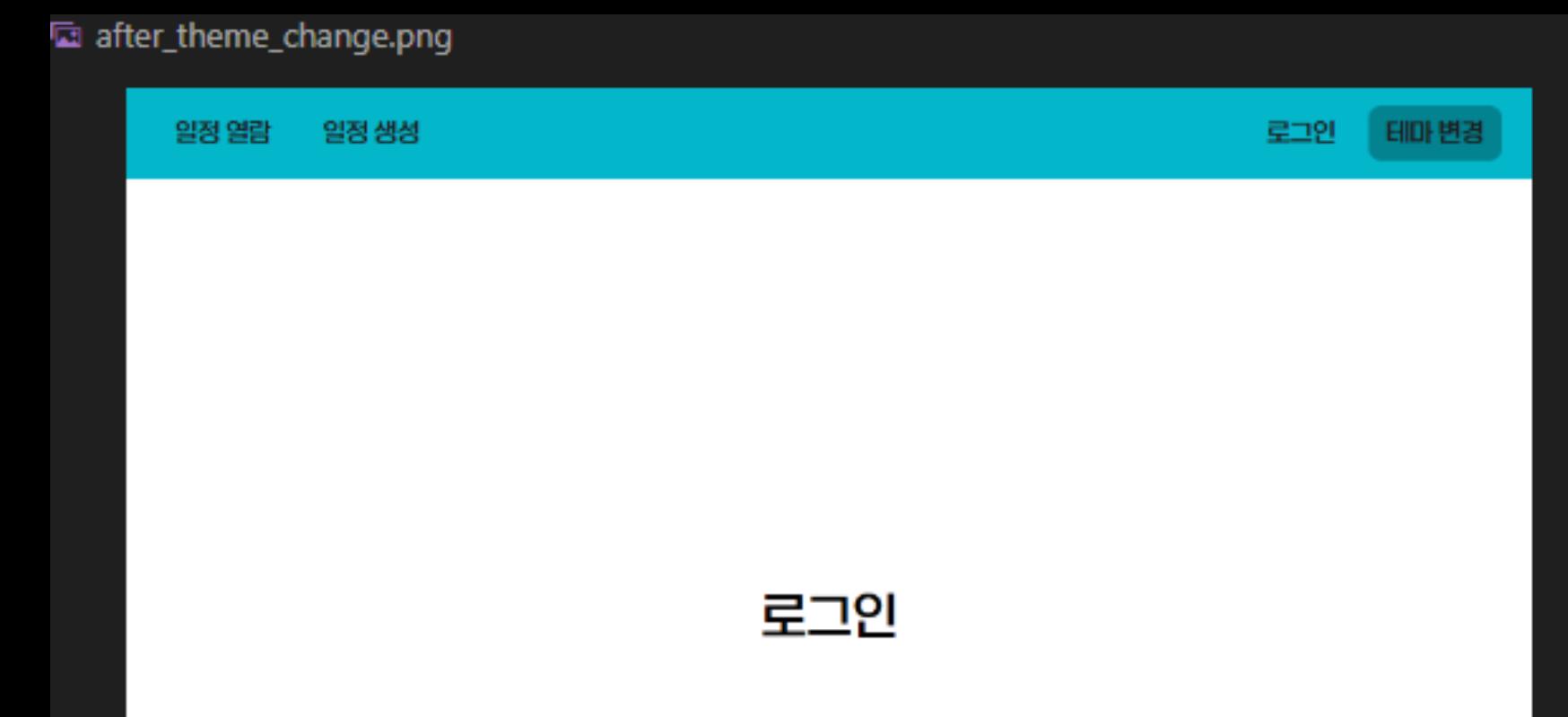
# Selenium 에러 발생 시 스크린샷 저장 방법

## 에러 발생 시 스크린샷 저장의 필요성

- 원인 분석에 중요한 자료 제공
- 실시간 디버깅이 어려운 환경에서 유용
- 예상과 실제 화면이 어떻게 다른지 즉시 확인 가능
- 로그 텍스트만으로는 파악 어려운 화면 구성 문제 분석에 효과적

## save\_screenshot() 활용 방법

```
driver.save_screenshot("error_screenshot.png")
```



# CSV 데이터 vs 웹 데이터 – 일치 여부 검증

## ✓ 왜 CSV 데이터 검증이 필요한가?

- 크롤링한 데이터를 저장한 CSV 파일이 웹에 표시된 데이터와 정확히 일치하는지 확인해야 함
- 데이터 불일치 발생 시 원인 분석 및 수정 가능
- `find\_element()`, `assert` 등을 활용하여 데이터 검증 가능

```
# ✓ CSV 파일에서 일정 데이터 불러오기
with open("cleaned_schedules.csv", "r", encoding="utf-8") as f:
    reader = csv.reader(f)
    csv_data = [row for row in reader]

# ✓ 웹에서 일정 제목 가져오기
wait = WebDriverWait(driver, 10)
web_titles = [el.text for el in wait.until(
    EC.presence_of_all_elements_located((By.CLASS_NAME, "schedule-title")))] # 일정 제목
# 클래스명 확인 필요

# ✓ 데이터 검증
for row in csv_data:
    assert row[0] in web_titles, f"✗ 일정 {row[0]}이(가) 웹에서 발견되지 않음"

print("✓ CSV 데이터와 웹 데이터가 일치합니다!")
```

# ⚠ Selenium 예외 종류와 대응 방법

## Selenium 주요 예외와 원인

예외명	발생 원인	주요 대응 방법
NoSuchElementException	요소 탐색 실패	선택자 확인 / 대기시간 조정
TimeoutException	지정한 시간 내 요소 미출현	대기시간 증가 / 조건 변경
WebDriverException	드라이버 종료, 세션 문제	드라이버 재시작, 예외처리 강화
ElementClickInterceptedException	다른 요소가 가림	스크롤 이동 / 가림 요소 제거

## 예외별 기본 처리 패턴

```
try:  
    element = wait.until(EC.presence_of_element_located((By.NAME, "title")))  
    element.click()  
except NoSuchElementException:  
    save_error_screenshot(driver, "no_element.png")  
    logging.error("요소를 찾을 수 없음")  
except TimeoutException:  
    save_error_screenshot(driver, "timeout.png")  
    logging.error("지정한 시간 내 요소 미출현")  
except WebDriverException:  
    save_error_screenshot(driver, "webdriver_issue.png")  
    logging.critical("WebDriver 문제 발생")
```



# WebDriverWait과 명시적 대기 활용법

```
wait = WebDriverWait(driver, 10)
wait.until(EC.presence_of_element_located((By.NAME, "email"))).send_keys("test33@example.com")
```

- 로그인 페이지에서 이메일 입력칸이 나타날 때까지 최대 10초 대기
- 나타나면 바로 이메일 입력
- 기다려도 안 나타나면 TimeoutException 발생

## 명시적 대기 패턴 분석

단계	코드 예시	설명
대기 설정	wait = WebDriverWait(driver, 10)	최대 10초 대기 설정
조건 적용	EC.presence_of_element_located	요소 존재 여부 확인
요소 탐색	wait.until()	조건 충족될 때까지 반복 시도
예외 처리	TimeoutException	시간 초과 시 예외 발생

# 자동화 테스트에서 에러 처리와 로깅 설정

## 에러 처리와 로깅의 중요성

- 자동화 테스트에서는 요소 미발견, 응답 지연, DOM 변경 등 다양한 오류 발생 가능
- 단순 실패 로그만 남기는 것이 아니라, 에러 상황 분석 및 원인 파악을 위해 상세 로그와 스크린샷 저장 필요
- Python의 logging 모듈을 활용해 로그 파일 생성 및 출력 가능

## 기본 로깅 설정 구성법

```
import logging

logging.basicConfig(
    level=logging.INFO, # 로그 레벨 설정
    format="%(asctime)s [%(levelname)s] %(message)s", # 출력 포맷 설정
    handlers=[
        logging.FileHandler("schedule_test.log", encoding="utf-8"), # 로그 파일 저장
        logging.StreamHandler() # 콘솔 출력
    ]
)
```

- 로그 레벨 INFO 설정 : 정보성 메시지부터 에러까지 기록
- 파일과 콘솔 동시 출력 : 테스트 과정 실시간 확인 + 기록 보존
- UTF-8 인코딩 : 한글 로그 깨짐 방지

# 자동화 테스트에서 try-except 활용법

## try-except의 역할

- 자동화 테스트 과정에서 예상치 못한 오류 발생 시, 프로그램 강제 종료 방지 및 원인 기록을 위해 필수
- 특히, Selenium 테스트에서는 요소 탐색 실패, 타임아웃, 연결 오류 등 다양한 예외 발생 가능

## 예외 처리 기본 구조

```
try:  
    element = wait.until(EC.presence_of_element_located((By.NAME, "title")))  
    element.send_keys("자동화 일정")  
except NoSuchElementException:  
    logging.error("✖ 요소를 찾을 수 없습니다.")  
    save_error_screenshot(driver, "element_not_found.png")  
except TimeoutException:  
    logging.error("⌚ 요소 대기 시간 초과")  
    save_error_screenshot(driver, "timeout_error.png")  
except WebDriverException as e:  
    logging.error(f"⚠ 드라이버 오류 발생: {e}")  
    save_error_screenshot(driver, "webdriver_exception.png")
```

# 에러 발생 시 스크린샷 저장 및 상세 디버깅

## 에러 상황 재현을 위한 스크린샷 저장

- Selenium은 save\_screenshot() 메서드로 현재 화면 그대로 이미지 저장 가능
- 단순 로그보다 시각적 증거 확보에 유리
- 화면 상태가 실제 어떤지 확인 가능해 UI 깨짐, 데이터 누락 등 문제 원인 파악에 유용

## 저장 방식 설계 포인트

- 파일명에 타임스탬프 추가로 문제 발생 시점 추적
- 에러 유형별로 파일명에 구분 키워드 포함 (예: login\_error.png)
- 프로젝트 경로 내 logs/screenshots 폴더 분리 저장으로 관리 편의성 확보

```
def save_error_screenshot(driver, filename="error_screenshot.png"):
    now = datetime.now().strftime("%Y%m%d_%H%M%S")
    full_path = f"./logs/screenshots/{now}_{filename}"
    driver.save_screenshot(full_path)
    logging.error(f"🚨 스크린샷 저장 완료: {full_path}")
```

# 로그 파일 관리 전략과 로그 레벨 활용법

## 왜 로그 파일 관리가 중요한가?

- 자동화 테스트는 하루에도 수십 번 실행 → 로그 파일이 누적되면 관리 어려움
- 파일 크기 관리 / 보존 기간 설정 등 로그 파일 관리 전략 필요
- 디버깅뿐만 아니라, 장기적인 품질 추적 데이터로도 활용 가능

## 로그 파일 저장 전략

전략명	설명
날짜별 로그 분리	날짜별 파일로 분리해 일자별 이력 관리
실행 시점 로그 생성	매 테스트마다 고유 파일 생성해 테스트 단위로 관리
보존 기간 설정	일정 기간 지난 로그는 자동 삭제 또는 보관
로그 레벨 필터링	필요 시점에 맞춰 디버그/에러 등 레벨별 로그만 남기기

```
schedule_test.log
1 2025-03-01 16:49:31,578 [INFO] 로그인 페이지 접속
2 2025-03-01 16:49:32,014 [INFO] 로그인 시도
3 2025-03-01 16:49:32,520 [INFO] 로그인 성공 및 페이지 이동 확인
4 2025-03-01 16:49:32,713 [INFO] 일정 생성 페이지 접속
5 2025-03-01 16:49:32,896 [INFO] 제목 입력 완료
6 2025-03-01 16:49:33,050 [INFO] 설명 입력 완료
7 2025-03-01 16:49:33,120 [INFO] 장소 입력 완료
8 2025-03-01 16:49:33,122 [INFO] 오늘 날짜: 2025-03-01
9 2025-03-01 16:49:33,188 [INFO] 날짜 값 확인: 2025-03-01
10 2025-03-01 16:49:33,275 [INFO] 시작 시간 값 확인: 14:00
11 2025-03-01 16:49:33,323 [INFO] 종료 시간 값 확인: 15:00
12 2025-03-01 16:49:33,605 [INFO] 생성 버튼 클릭 완료
13 2025-03-01 16:49:36,615 [INFO] ✅ 일정 생성 테스트 성공!
14 2025-03-01 16:51:31,953 [INFO] 로그인 페이지 접속
15 2025-03-01 16:51:32,489 [INFO] 로그인 시도
16 2025-03-01 16:51:32,995 [INFO] 로그인 성공 및 페이지 이동 확인
```



# 에러 흐름 추적과 심화 로깅 기법

## 에러 흐름 추적이 중요한 이유

- 테스트 실패 원인 파악 속도를 크게 단축
- 단순히 '어떤 에러가 발생했다'는 정보만으로는 실제 원인 파악이 어려움
- 발생 위치, 당시 상태, 재시도 여부 등 맥락(Context)까지 함께 남겨야 원인 분석 가능

## 심화 로깅 기법 (콘텍스트 로깅)

기법	설명
단계별 로그	단계별로 INFO 로그 남겨, 실패 지점 직전 단계 파악
주요 데이터 로그	테스트 중 다루는 핵심 데이터(날짜, 제목 등)를 별도 로그
예외 스택 트레이스	try-except에서 exc_info=True로 예외 정보 자동 출력
추가 캡처 데이터	스크린샷 외에도 현재 URL, 페이지 제목 등 함께 기록

```
try:  
    title = driver.find_element(By.NAME, "title").get_attribute("value")  
    logging.info("현재 입력된 제목: {title}")  
except NoSuchElementException:  
    logging.error("제목 입력란을 찾을 수 없습니다.", exc_info=True)
```



# 재시도(리트라이) 로직 설계와 자동 복구 전략

## 왜 재시도 로직이 필요할까?

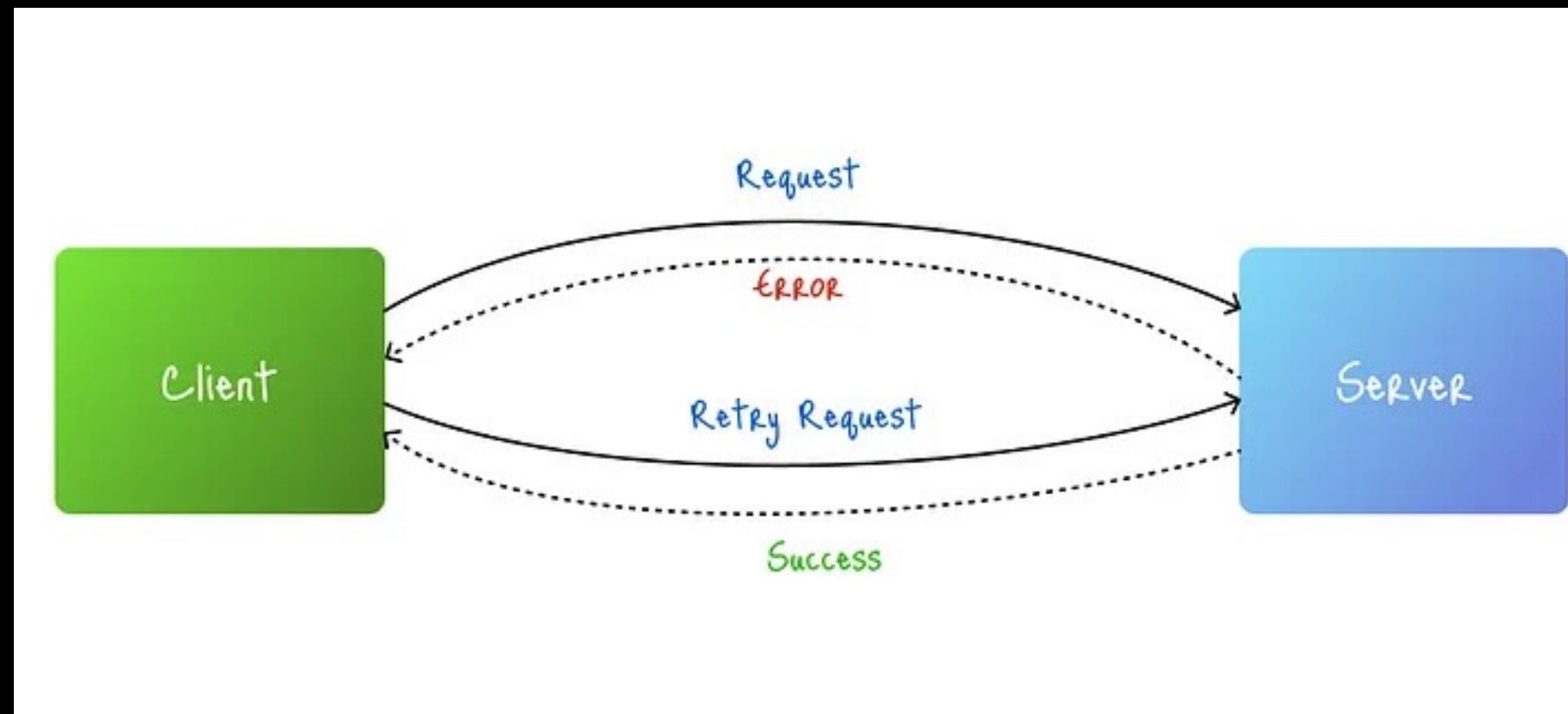
- 자동화 테스트는 네트워크 지연, 비동기 데이터 로드 문제로 인해 예상치 못한 요소 탐색 실패나 타임아웃 발생 가능
- 일시적 오류로 테스트 전체가 실패하지 않도록 재시도(리트라이) 구조를 추가하면 안정성 대폭 향상

## 재시도 로직 구성 요소

구성 요소	설명
최대 재시도 횟수	동일 동작을 몇 번까지 반복할지 설정 (ex: 3회)
재시도 간격	각 시도 사이의 대기 시간 (ex: 2초)
예외 종류	어떤 예외가 발생했을 때만 재시도할지 지정
최종 실패 처리	재시도 실패 시 최종 에러 로그 및 스크린샷 저장

```
def find_element_with_retry(driver, by, value, retries=3, delay=2):
    for attempt in range(retries):
        try:
            element = WebDriverWait(driver, 5).until(EC.presence_of_element_located((by,
value)))
            logging.info(f"✓ 요소 찾기 성공: {value} (시도 {attempt+1}/{retries})")
            return element
        except (NoSuchElementException, TimeoutException) as e:
            logging.warning(f"⚠ 요소 찾기 실패 - 재시도 중... (시도 {attempt+1}/{retries})")
            time.sleep(delay)
            logging.error(f"✗ 최종 실패 - 요소 찾지 못함: {value}")
            save_error_screenshot(driver, "retry_fail.png")
            raise e
```

# 재시도 로직 활용 흐름과 실패 대응 전략



## 재시도 로직 전체 흐름도

1. 요소 탐색 시도
2. 성공 시 바로 반환
3. 실패 시 로그 출력 및 대기 (ex: 2초)
4. 최대 재시도 횟수 도달 여부 체크

## 재시도 로직 적용 시 주의점

- 무한 루프 방지를 위해 반드시 최대 시도 횟수 설정
- 매 시도 간 충분한 대기 시간 설정해 서버 부담 최소화
- 로그에 시도 횟수 및 실패 원인 남겨야 원인 분석 가능

# 실습 5: 테스트 자동화에서 에러 처리

## 실습 목표

- Selenium 자동화 테스트 수행 중 발생할 수 있는 오류 상황을 체계적으로 처리하는 방법 학습
- try-except 문을 활용한 예외 처리 기법 습득
- logging 모듈을 활용해 테스트 과정 및 에러 상황을 로그 파일로 기록하는 방법 학습
- 오류 발생 시 화면을 캡처해 문제 원인을 시각적으로 확인하는 방법 익히기

## 실습 순서

1. 테스트 자동화 환경 설정 및 로깅 구성
2. 로그인 단계에서 에러 처리 연습
3. 일정 생성 단계에서 입력 오류 처리
4. 일정 생성 성공 여부 확인 및 실패 시 디버깅

```
schedule_test.log
1 2025-03-01 16:49:31,578 [INFO] 로그인 페이지 접속
2 2025-03-01 16:49:32,014 [INFO] 로그인 시도
3 2025-03-01 16:49:32,520 [INFO] 로그인 성공 및 페이지 이동 확인
4 2025-03-01 16:49:32,713 [INFO] 일정 생성 페이지 접속
5 2025-03-01 16:49:32,896 [INFO] 제목 입력 완료
6 2025-03-01 16:49:33,050 [INFO] 설명 입력 완료
7 2025-03-01 16:49:33,120 [INFO] 장소 입력 완료
8 2025-03-01 16:49:33,122 [INFO] 오늘 날짜: 2025-03-01
9 2025-03-01 16:49:33,188 [INFO] 날짜 값 확인: 2025-03-01
10 2025-03-01 16:49:33,275 [INFO] 시작 시간 값 확인: 14:00
11 2025-03-01 16:49:33,323 [INFO] 종료 시간 값 확인: 15:00
12 2025-03-01 16:49:33,605 [INFO] 생성 버튼 클릭 완료
13 2025-03-01 16:49:36,615 [INFO] ✅ 일정 생성 테스트 성공! 2025-03-02 23:35:54,568 [INFO] 로그인 페이지 접속
14 2025-03-01 16:51:31,953 [INFO] 로그인 페이지 접속 2025-03-02 23:35:55,187 [INFO] 로그인 시도
15 2025-03-01 16:51:32,489 [INFO] 로그인 시도 2025-03-02 23:35:55,201 [INFO] 로그인 성공 및 페이지 이동 확인
16 2025-03-01 16:51:32,995 [INFO] 로그인 성공 및 페이지 이동 2025-03-02 23:35:55,466 [INFO] 일정 생성 페이지 접속
17 2025-03-01 16:51:33,286 [INFO] 일정 생성 페이지 접속
18 2025-03-01 16:51:33,501 [INFO] 제목 입력 완료 2025-03-02 23:35:55,647 [INFO] 제목 입력 완료
19 2025-03-01 16:51:33,644 [INFO] 설명 입력 완료 2025-03-02 23:35:55,769 [INFO] 설명 입력 완료
20 2025-03-01 16:51:33,730 [INFO] 장소 입력 완료 2025-03-02 23:35:55,848 [INFO] 장소 입력 완료
21 2025-03-01 16:51:33,730 [INFO] 오늘 날짜: 2025-03-01 2025-03-02 23:35:55,848 [INFO] 오늘 날짜: 2025-03-02
22 2025-03-01 16:51:33,781 [INFO] 날짜 값 확인: 2025-03-01 2025-03-02 23:35:55,899 [INFO] 날짜 값 확인: 2025-03-02
23 2025-03-01 16:51:33,822 [INFO] 시작 시간 값 확인: 14:00 2025-03-02 23:35:55,932 [INFO] 시작 시간 값 확인: 14:00
24 2025-03-01 16:51:33,869 [INFO] 종료 시간 값 확인: 15:00 2025-03-02 23:35:55,966 [INFO] 종료 시간 값 확인: 15:00
25 2025-03-01 16:51:34,107 [INFO] 생성 버튼 클릭 완료 2025-03-02 23:35:56,191 [INFO] 생성 버튼 클릭 완료
26 2025-03-01 16:51:37,112 [INFO] ✅ 일정 생성 테스트 성공! 2025-03-02 23:35:59,199 [INFO] ✅ 일정 생성 테스트 성공!
```



## 퀴즈 타임



QUIZ 9번부터 16번을 풀어봐요!