M03-03 · GIBALJA VISUAL MANUAL

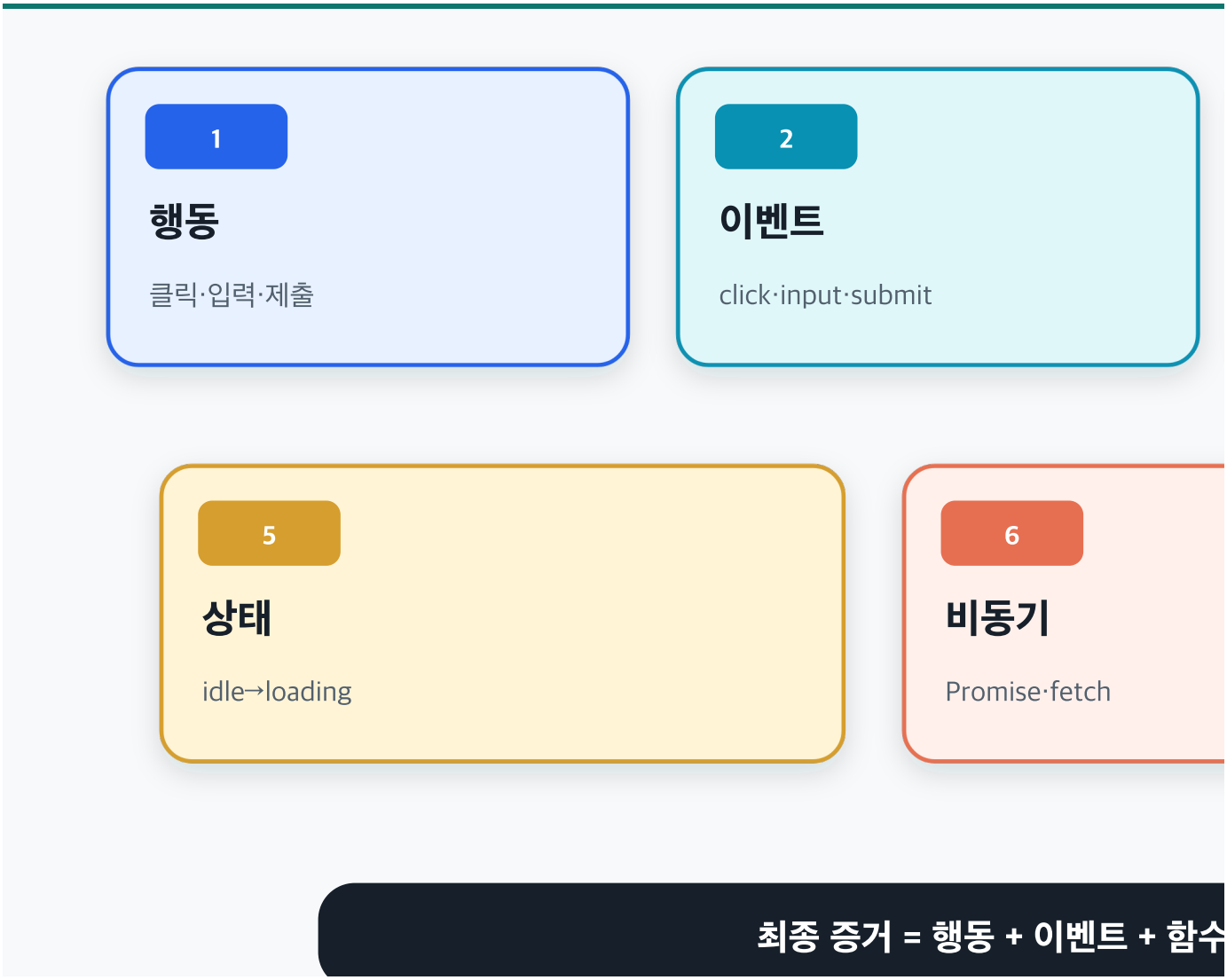
JavaScript 상태·이벤트·비동기 읽기

한 문장 목표: 사용자의 한 번의 행동을 이벤트 대상·전달 경로·실행 함수·상태 변화·비동기 요청·화면 결과의 증거로 연결하고, 성공뿐 아니라 빈 결과·오류·취소·늦은 응답까지 설명합니다.

난이도	개념	실습	셀프 테스트	최종 산출물
Level 1	60분	60분	15분	화면 행동 명세, 상태 전이도, 이벤트·비동기 추적표

"검색 버튼이 안 됩니다"는 현상입니다. "submit handler가 idle을 loading으로 바꾸고 GET 요청을 시작하지만, HTTP 500 Response를 성공 데이터처럼 해석해 success 화면을 그립니다"라고 쓰면 행동·코드·상태·요청·화면이 한 문장에 연결됩니다.

그림 1. 클릭 한 번은 행동에서 끝나지 않습니다. 이벤트 경로와 상태·비동기 결과를 거쳐 사용자가 보는 화면으로 이어집니다.


1
행동
클릭·입력·제출
2
이벤트
click·input·submit
5
상태
idle→loading
6
비동기
Promise·fetch
최종 증거 = 행동 + 이벤트 + 함수

3

경로

capture·target·bubble

4

핸들러

함수·기본 동작

7

화면

성공·빈 결과·오류

+ 전·후 상태 + 요청 + 화면 결과

1. 이 PDF를 공부하는 방법

1회차 · 일곱 단계를 소리 내어 읽기 · 15분

그림 1의 `행동 → 이벤트 → 경로 → 핸들러 → 상태 → 비동기 → 화면`을 외웁니다. 코드를 한 줄씩 해석하려 하지 말고, 각 단계의 증거가 어디에 있는지 먼저 말합니다.

2회차 · 상태와 비동기 층 분리하기 · 30분

`Promise fulfilled = 화면 success`라고 연결하지 않습니다. 전송·HTTP·본문·업무 데이터의 네 판정을 차례로 통과시키고, `success`, `empty`, `error`를 구분합니다.

3회차 · 실험실에서 여섯 결과 만들기 · 35분

이벤트·상태·비동기 실험실에서 정상 성공·빈 결과·HTTP 500·네트워크 실패·늦은 응답 경쟁·취소를 실행합니다. 오른쪽 시간선을 읽고 실습지에 옮깁니다.

4회차 · Chrome에서 실제 행동 추적하기 · 40분

Event listener breakpoint와 XHR/fetch breakpoint로 코드 실행을 멈춥니다. Call Stack·Scope·Network Initiator·Status·DOM 변화를 이어 붙입니다.

5회차 · 셀프 테스트 · 15분

정답을 가리고 10문제를 풉니다. 틀린 문제는 이벤트·상태·Promise·HTTP·업무 데이터·화면 중 섞어 생각한 층을 표시합니다.

학습 완료 기준

"두 번 검색하면 가끔 이전 결과가 보인다"를 "A 요청 1600ms 뒤 B 요청을 시작했고 B가 500ms에 먼저 success를 렌더했지만, A 응답이 나중에 도착해 최신 요청 판정 없이 같은 결과 영역을 덮습니다. 이전 fetch를 abort하고 응답 requestId가 최신 값일 때만 render하는 것을 완료 기준으로 합니다"처럼 설명할 수 있습니다.

2. 화면 행동은 일곱 단계의 증거 사슬입니다

JavaScript는 화면의 모든 일을 혼자 만드는 마법 상자가 아닙니다. 브라우저가 이벤트를 전달하고, 등록된 함수가 실행되며, 상태가 바뀌고, 필요하면 비동기 작업을 기다린 뒤 DOM이 새 상태를 표현합니다.

```
사용자: 검색 버튼 클릭
브라우저: click 전달 → form submit 기본 행동 준비
코드: submit listener → preventDefault → search()
상태: idle → loading
비동기: fetch → Response → JSON → 업무 데이터 판정
화면: success·empty·error·canceled 중 하나를 render
```

2.1. 일곱 질문

순서	질문	대표 증거
1 행동	사용자가 무엇을 했는가	클릭·입력·제출·취소·화면 이탈
2 이벤트	어떤 type과 target인가	Event 객체·Event Listeners 패널
3 경로	어느 listener를 어떤 순서로 거쳤는가	capture·target·bubble·currentTarget
4 핸들러	어떤 함수와 브라우저 기본 동작이 실행됐는가	breakpoint·Call Stack·`preventDefault()`
5 상태	전과 후의 화면 상태는 무엇인가	상태 변수·DOM·상태 메시지
6 비동기	무엇을 기다리고 어떻게 판정했는가	Promise·fetch·Status·Response·Timing
7 화면	사용자는 무엇을 보고 무엇을 할 수 있는가	결과·빈 화면·오류·취소·다음 행동

2.2. 관찰 문장의 기본 형식

```
[행동] 사용자가 검색 A를 누르면
[이벤트] button을 target으로 click이 전달되고 form submit이 발생한다.
[함수] submit handler는 기본 제출을 취소하고 search("A")를 호출한다.
[상태] UI는 idle에서 loading으로 바뀐다.
[비동기] GET 요청이 시작되고 200 Response의 JSON 3건을 해석한다.
[화면] UI는 success가 되어 결과 3건과 다음 행동을 표시한다.
```

이 형식을 지키면 “버튼”, “API”, “화면”을 서로 떨어진 문제로 보지 않게 됩니다.

BIG IDEA 01

좋은 화면 분석은 코드 줄을 많이 인용하는 일이 아니라, 사용자 행동과 최종 화면 사이의 원인·상태·증거를 빠짐없이 연결하는 일입니다.

30초 확인

Network에 200이 보였지만 화면은 빈 목록입니다. “API 성공”만 기록해도 될까요?

정답 보기
부족합니다. 전송과 HTTP는 성공했어도 본문이 기대 형식인지, 업무 데이터가 0건인지, render가 empty 분기를 가졌는지 확인해야 합니다.

3. 이벤트는 “무슨 일이 어디에서 시작됐는가”를 담습니다

DOM 표준에서 이벤트는 발생한 일을 나타내며, EventTarget은 listener를 등록할 수 있는 객체입니다. 요소에 `addEventListener(type, callback)`을 등록하면 해당 type의 이벤트가 전달될 때 callback이 실행됩니다.

```
const button = document.querySelector('#search');

button.addEventListener('click', event => {
  console.log(event.type);   // click
  console.log(event.target); // 실제 시작점
});
```

3.1. 자주 추적하는 이벤트

type	발생 계기	분석할 때 주의할 점
`click`	포인터·키보드 활성화	pointerdown만 보지 말고 접근 가능한 click 흐름 확인
`input`	입력값이 바뀜	매 키 입력 요청이면 debounce 필요 여부 확인
`change`	값 변경이 확정됨	컨트롤 종류마다 발생 시점이 다를 수 있음
`submit`	폼 제출	버튼 click 외 Enter 제출도 포함

type	발생 계기	분석할 때 주의할 점
`keydown`	키를 누름	키보드 단축과 기본 동작 충돌 확인
`focusin`	초점이 들어옴	bubble되어 위임하기 쉬움

폼은 click만 추적하면 놓치는 경로가 있습니다. 입력칸에서 Enter를 눌러도 submit이 발생할 수 있으므로 “제출”이 업무 행동이면 submit listener를 중심으로 읽습니다.

3.2. `isTrusted`는 출처 단서입니다

`event.isTrusted`는 사용자 에이전트가 만든 이벤트인지, 스크립트가 `dispatchEvent()` 등으로 만든 이벤트인지 구분하는 단서입니다. 이것만으로 보안 판정을 하지 않지만, 자동 테스트·합성 이벤트와 실제 입력 흐름을 구분할 때 유용합니다.

```
element.addEventListener('click', event => {
  console.log({ type: event.type, isTrusted: event.isTrusted });
});
```

Console에 이벤트 객체 전체나 Network 요청 전체를 복사하면 토큰·세션·개인정보가 포함될 수 있습니다. 공유 산출물에는 필요한 type·선택자·status·시간만 옮기고 민감한 값은 제거합니다.

4. capture·target·bubble 경로를 읽습니다

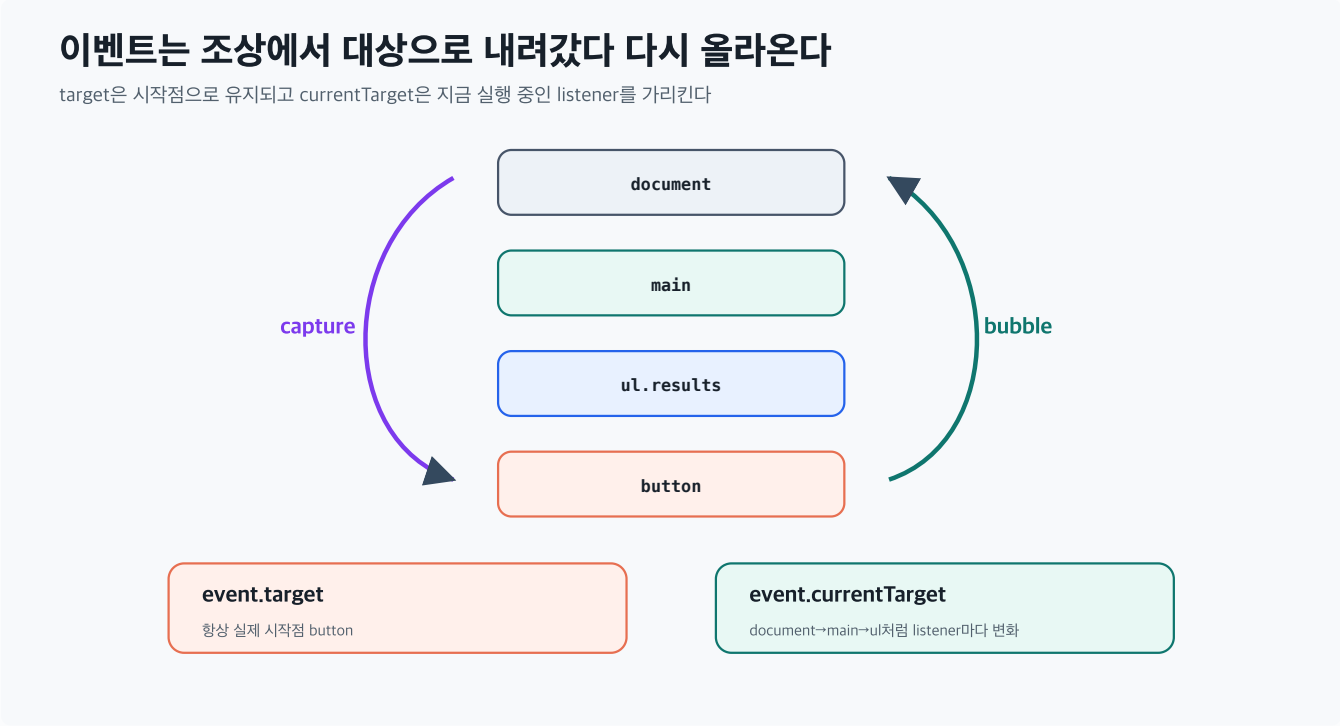
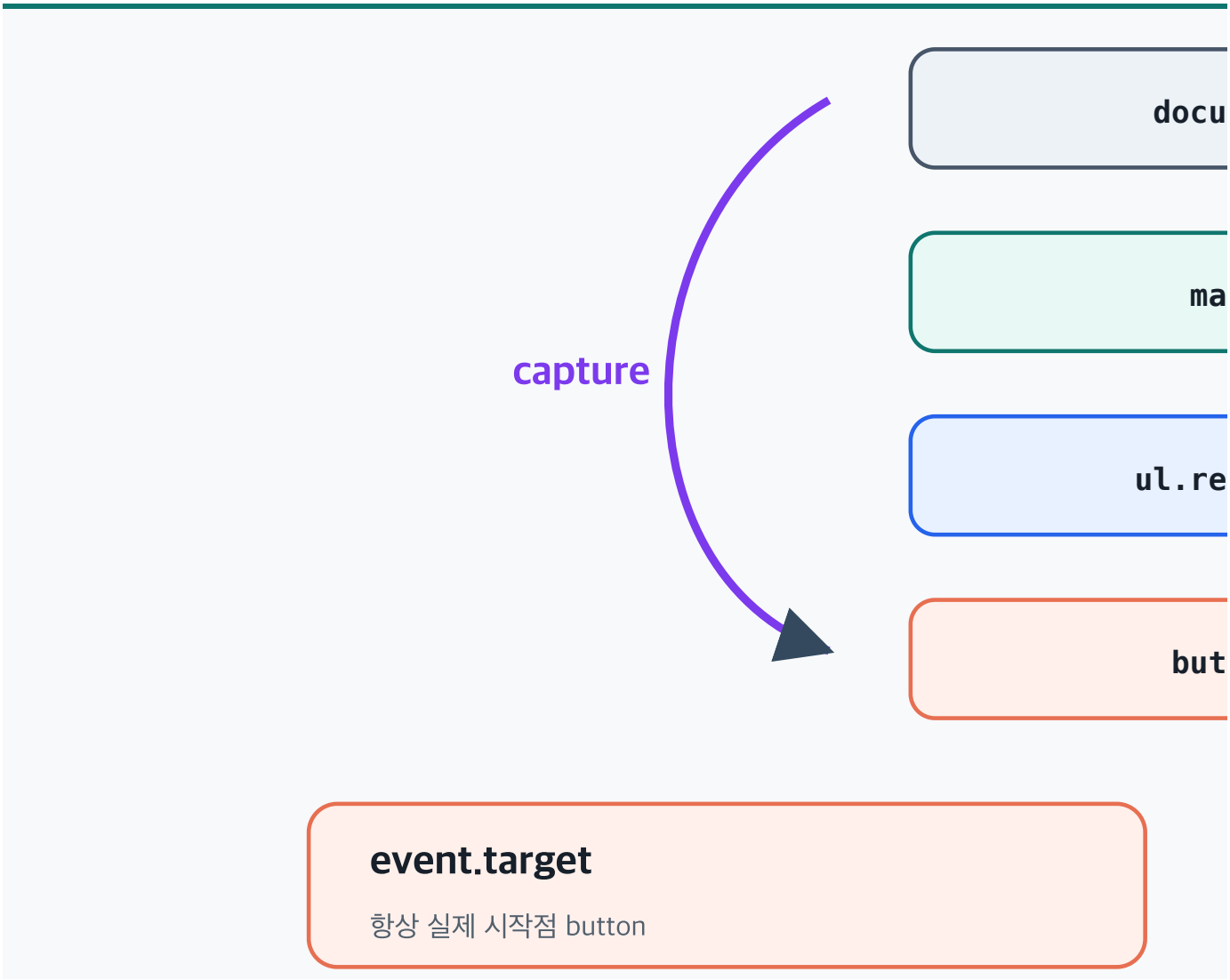
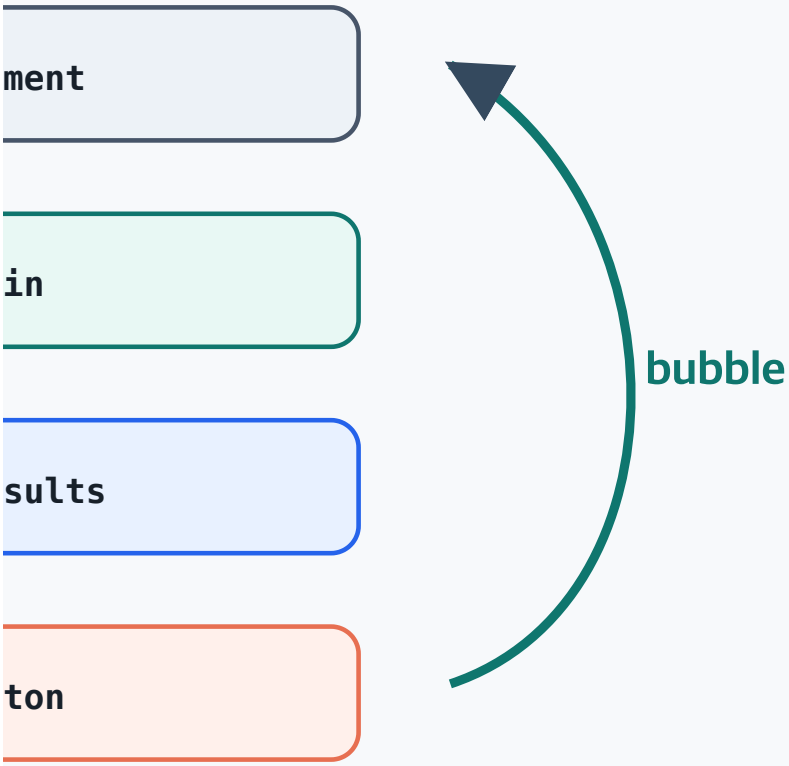


그림 2. target은 이벤트 시작점으로 유지되지만 currentTarget은 현재 listener가 등록된 요소로 바뀝니다.


docu
ma
capture
ul.re
but
event.target
항상 실제 시작점 button



event.currentTarget

document→main→ul처럼 listener마다 변화

4.1. 세 단계

1. **capture:** 조상에서 target 방향으로 내려갑니다.
2. **target:** 실제 이벤트 대상의 listener가 실행됩니다.
3. **bubble:** `bubbles`가 true인 이벤트가 target에서 조상 방향으로 올라갑니다.

```
const list = document.querySelector('.results');
const button = list.querySelector('button');

list.addEventListener('click', log, true);   // capture
button.addEventListener('click', log);       // target
list.addEventListener('click', log);         // bubble

function log(event) {
  console.log({
    target: event.target,
    currentTarget: event.currentTarget,
    phase: event.eventPhase
  });
}
```

4.2. `target`과 `currentTarget`

값	유지·변화	답하는 질문
`event.target`	경로 동안 시작점으로 유지	실제 어디에서 시작됐는가
`event.currentTarget`	listener마다 변화	지금 어느 요소에 등록된 함수가 실행 중인가

버튼 안의 아이콘을 누르면 target이 아이콘일 수 있습니다. 행동의 의미가 버튼에 있다면 `event.target.closest('button')`처럼 실제 행동 요소를 찾습니다.

4.3. 경로 전체 확인

`event.composedPath()`는 전달 경로를 배열로 돌려줍니다. Shadow DOM이 포함된 구성요소에서는 단순 `parentElement` 연쇄와 다를 수 있습니다. 초급 단계에서는 경로를 디버깅하는 도구로 사용하고, 내부 구현 전체를 외우지 않습니다.

```
const path = event.composedPath().map(node => node.nodeName ?? String(node));
```

30초 확인

부모 `ul` listener에서 자식 버튼을 눌렀습니다. `event.currentTarget`은 `ul`, `event.target`은 버튼입니다. 둘 중 어느 값으로 listener 등록 위치를 확인합니까?

정답 보기

`currentTarget`입니다. `target`은 이벤트가 시작된 실제 요소를 가리킵니다.

5. 기본 동작 취소와 전파 중지는 다릅니다

그림 3. `preventDefault()`는 링크 이동·폼 제출 같은 기본 동작을, `stopPropagation()`은 뒤 이벤트 경로 전달을 다룹니다.

브라우저 기본 동작

링크 이동 · 폼 제출 · 체크 상태 변경

```
event.preventDefault()
```

cancelable 이벤트일 때 기본 동작 취소 신호

전파는 계속될 수 있음

이벤트 경로 전달

capture · target · bubble listener 호출

```
event.stopPropagation()
```

현재 객체 뒤의 다른 경로 대상 전달을 중지

기본 동작은 남을 수 있음

5.1. `preventDefault()`

취소 가능한 이벤트에서 브라우저 기본 동작을 실행하지 않도록 신호합니다. 폼을 JavaScript로 처리할 때 페이지 이동을 막는 예가 대표적입니다.

```
form.addEventListener('submit', async event => {
  event.preventDefault();
  await search(new FormData(form));
});
```

`event.cancelable`이 false인 이벤트에는 취소할 기본 동작이 없습니다. `defaultPrevented`로 취소 요청 여부를 확인할 수 있습니다.

5.2. `stopPropagation()`

뒤 경로 대상에 이벤트가 전달되는 것을 막습니다. 이것은 브라우저 기본 동작을 자동으로 취소하지 않습니다. 모달·중첩 메뉴처럼 경계가 분명한 구성요소에서 쓸 수 있지만, 무분별하면 상위 분석·접근성·통합 listener가 행동을 받지 못합니다.

목적	메서드	남을 수 있는 것
폼 기본 제출·링크 이동 취소	`preventDefault()`	capture·bubble 전파
뒤 조상·대상 전달 중지	`stopPropagation()`	브라우저 기본 동작
같은 대상의 뒤 listener도 중지	`stopImmediatePropagation()`	기본 동작

코드 검토에서는 “왜 막는가”를 적습니다. `preventDefault()`와 `stopPropagation()`을 습관처럼 함께 호출하면 필요한 경로와 기본 기능까지 사라질 수 있습니다.

6. 이벤트 위임은 변하는 자식을 부모에서 처리합니다

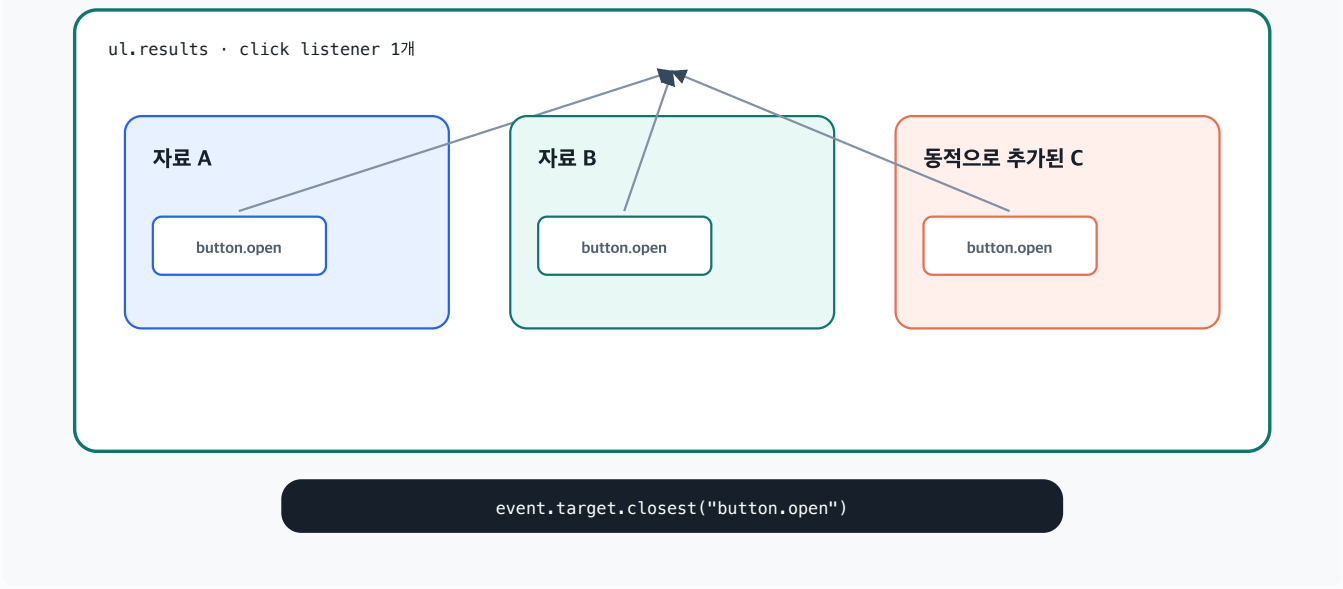
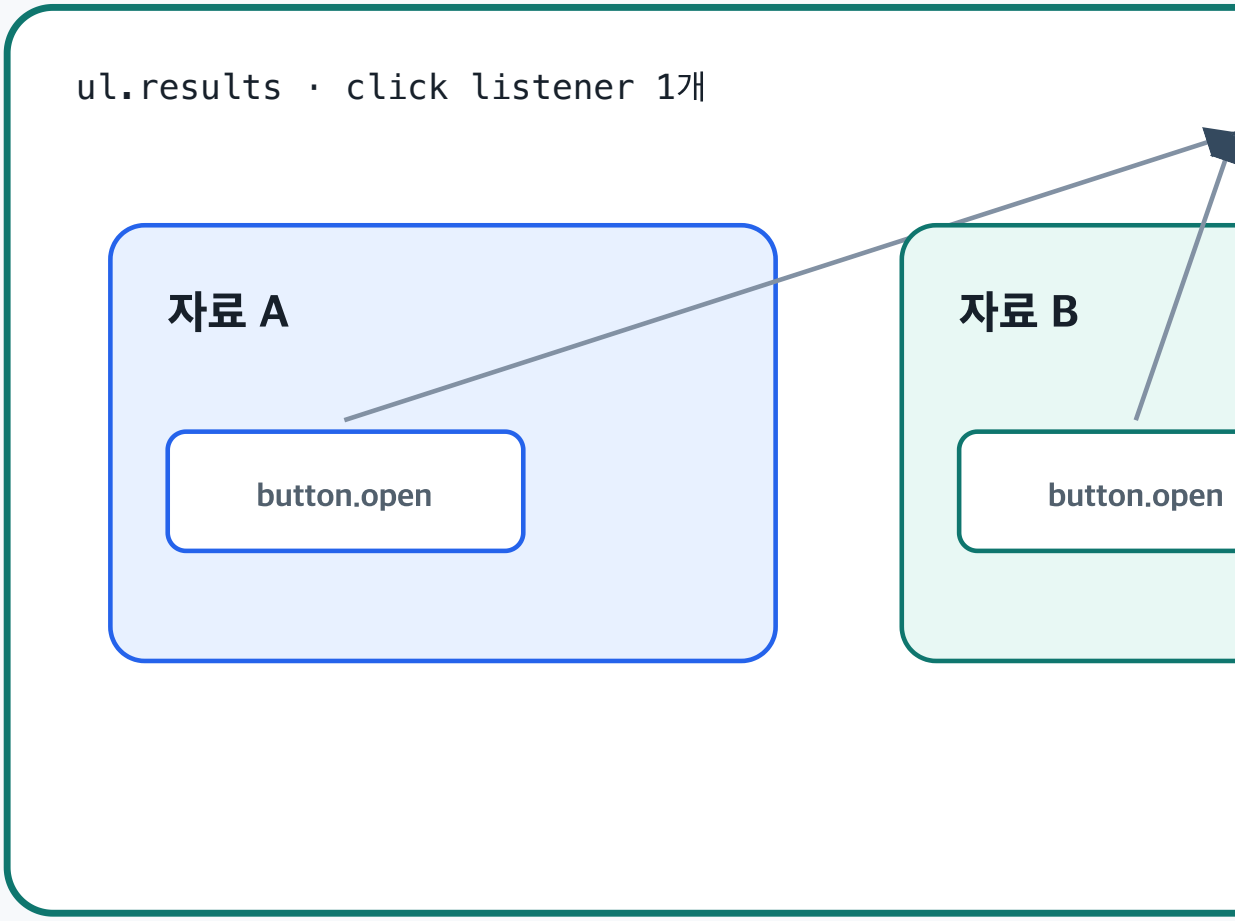
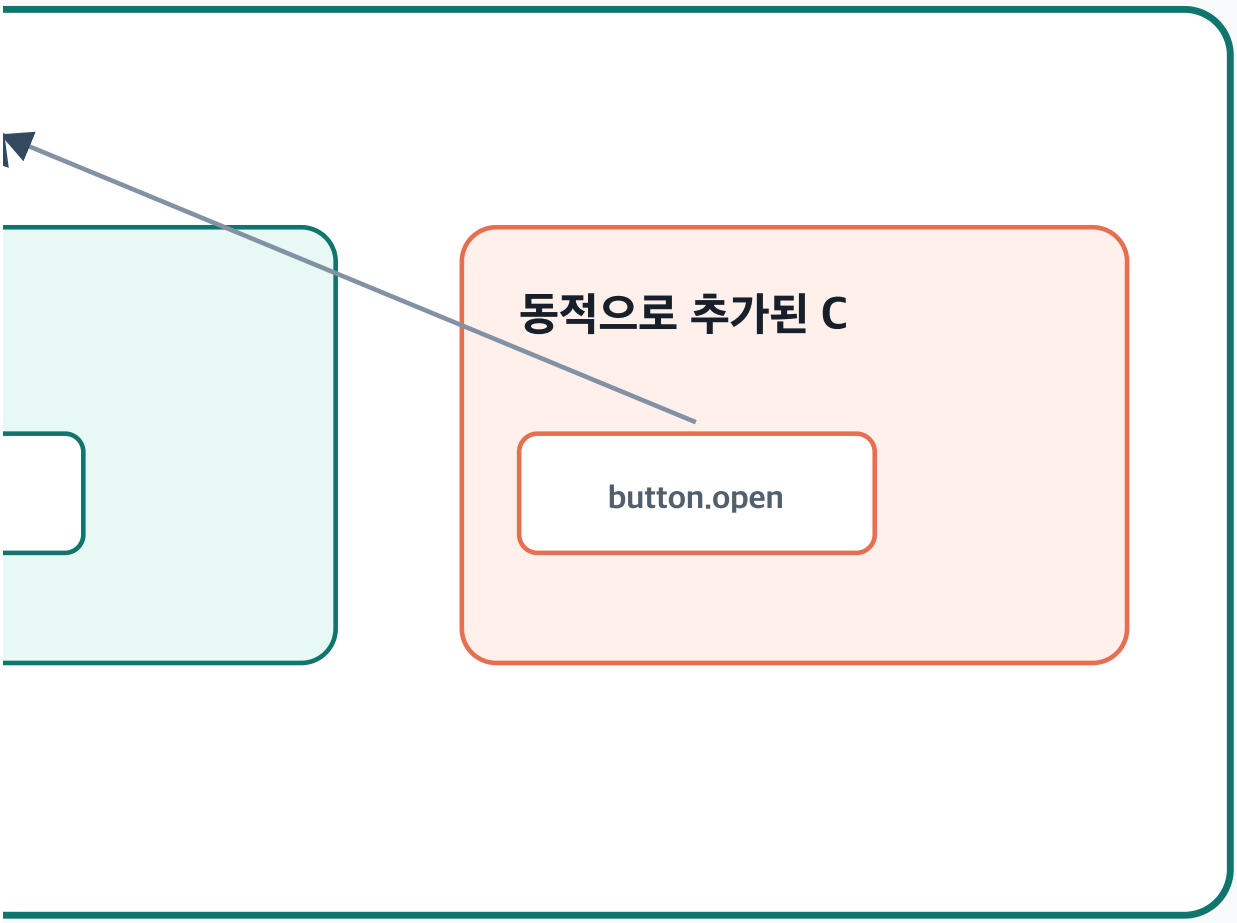


그림 4. 결과가 나중에 추가돼도 부모 listener는 bubble 경로에서 실제 행동 버튼을 찾아 처리할 수 있습니다.




```
event.target.close
```




```
est("button.open")
```

```
const results = document.querySelector('.results');

results.addEventListener('click', event => {
  const button = event.target.closest('button.open');
  if (!button || !results.contains(button)) return;

  openItem(button.dataset.id);
});
```

6.1. 위임 판독 질문

- listener는 어느 공통 부모에 등록됐습니까?
- 이벤트 type은 bubble됩니까?
- target이 버튼 내부 아이콘이어도 `closest()`로 버튼을 찾습니까?
- 선택한 버튼이 실제 위임 영역 안에 있는지 확인합니까?
- 자식이 추가·제거될 때 개별 listener 정리가 필요한 구조입니까?

6.2. 위임이 항상 정답은 아닙니다

독립 구성요소의 수명과 행동이 명확하면 각 구성요소가 listener를 소유하는 편이 읽기 쉽습니다. 위임은 자식이 많거나 동적으로 바뀌는 목록에서 특히 유용합니다. 패턴 이름보다 소유 경계와 정리 책임이 분명한지를 봅니다.

7. 화면 상태를 이름 있는 지도로 만듭니다

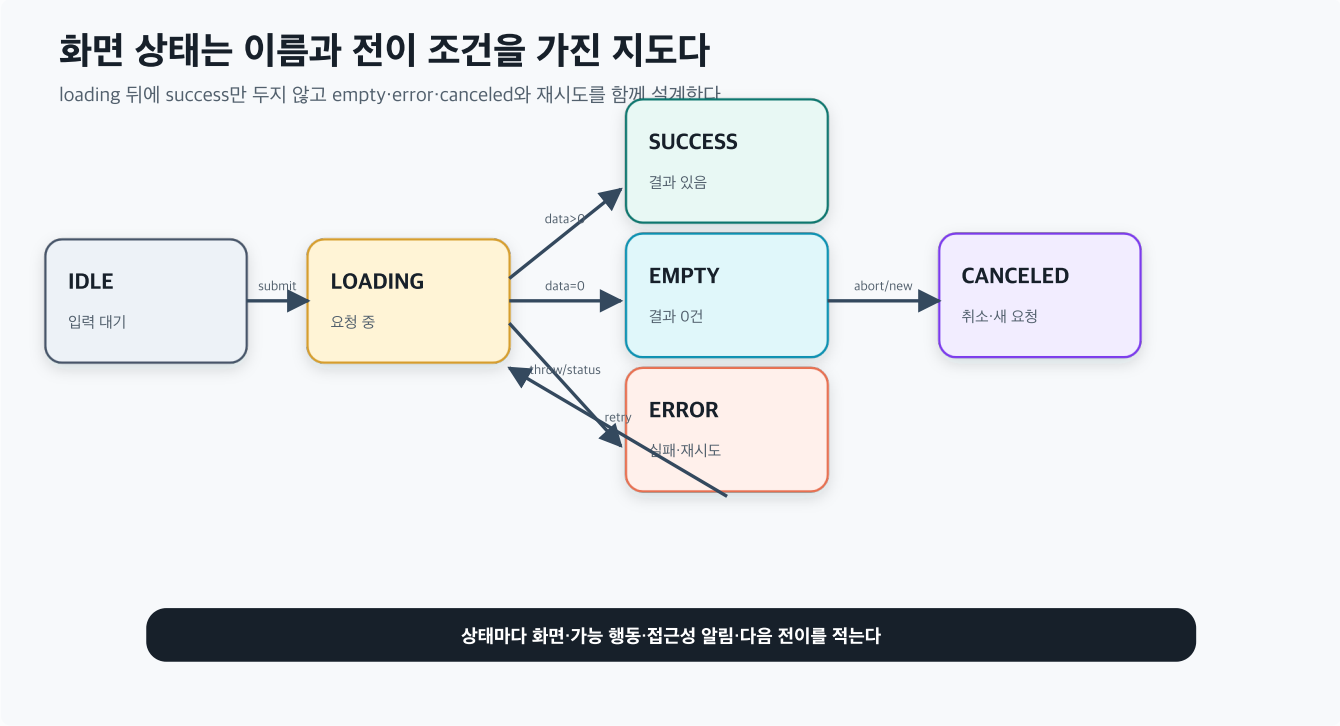
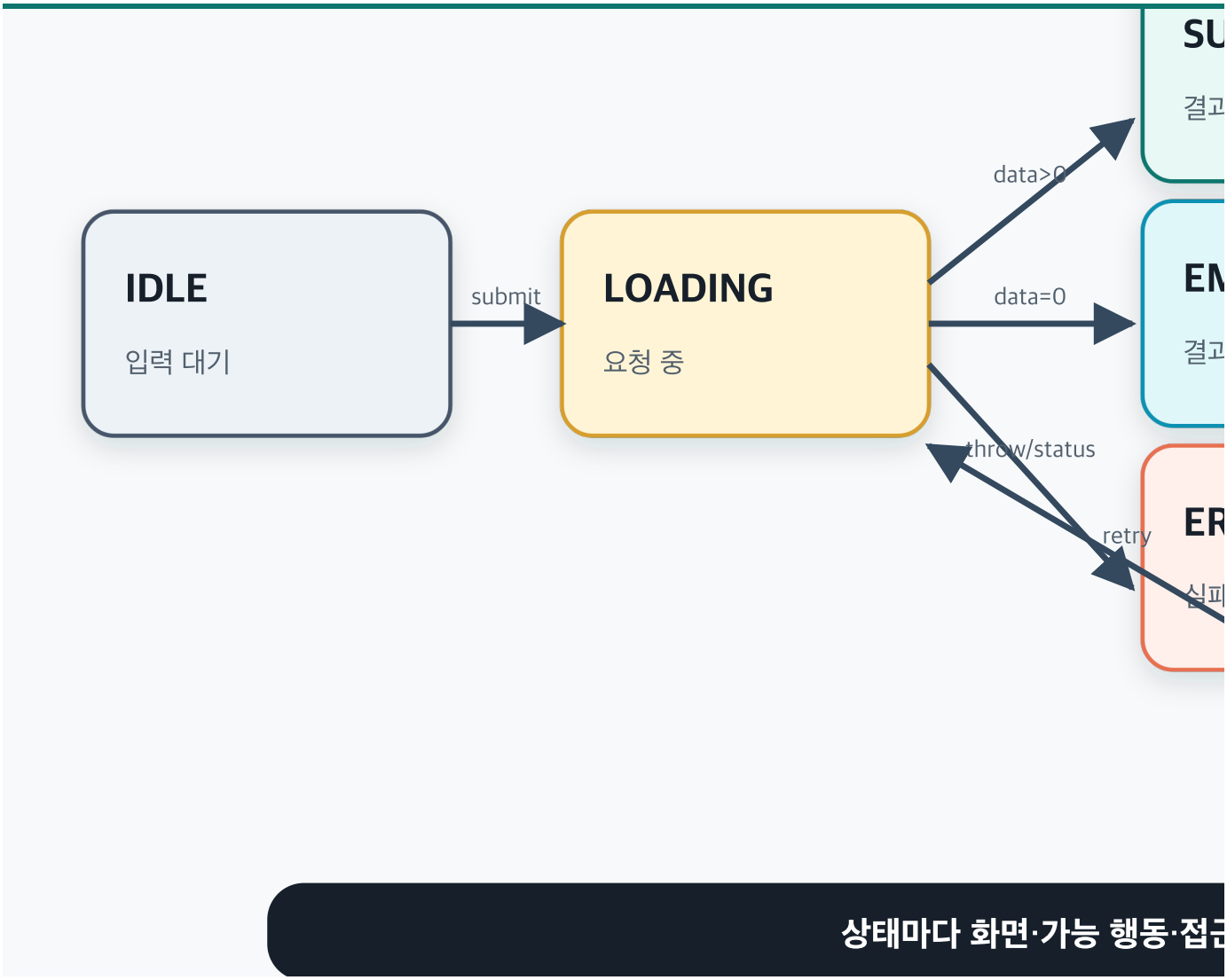
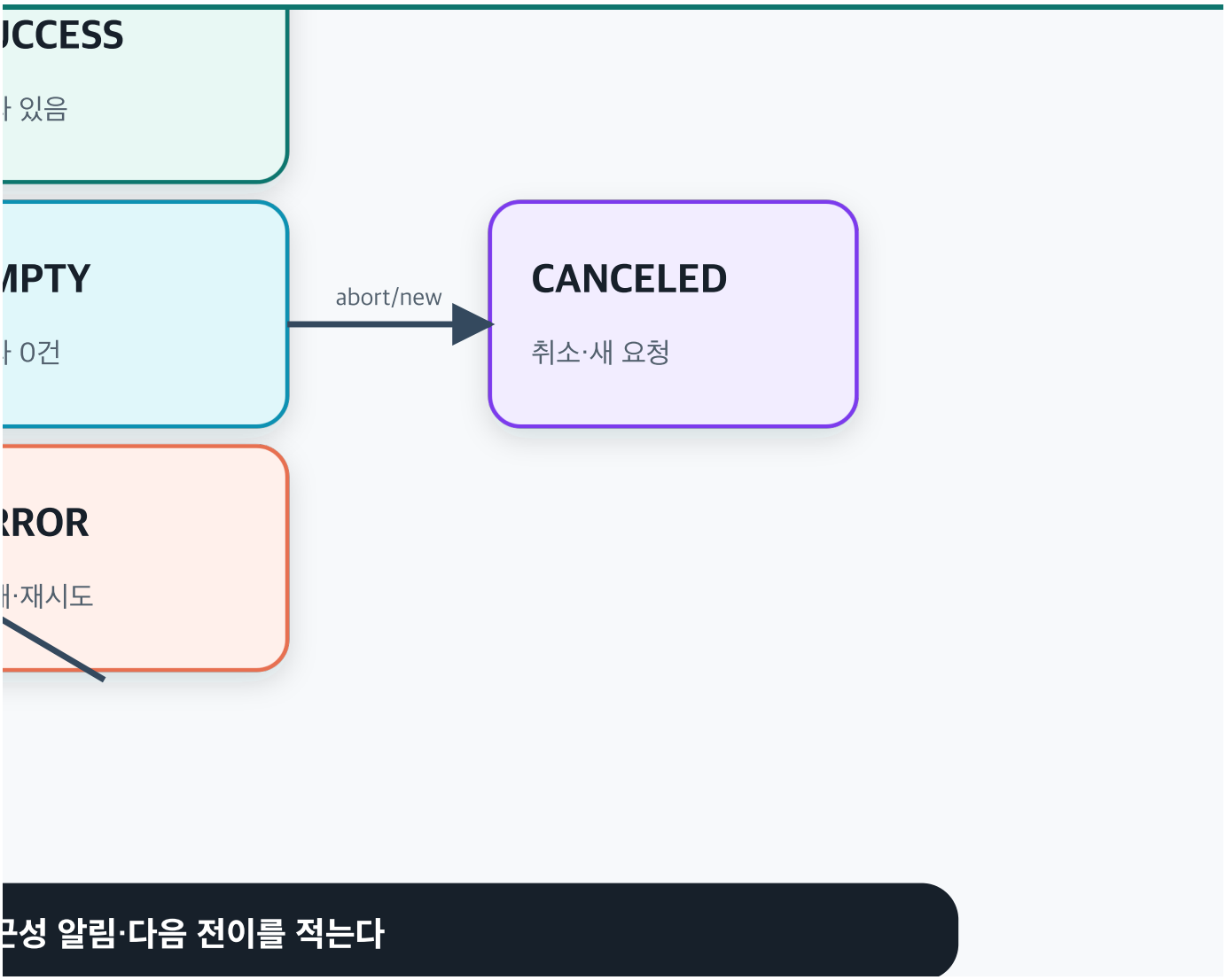


그림 5. 상태는 단순 변수 목록이 아니라 진입 조건·화면·가능 행동·다음 전이를 가진 지도입니다.


IDLE
입력 대기
submit
LOADING
요청 중
data>0
data=0
throw/status
retry
상태마다 화면·가능 행동·접근


CCESS
있음
MPTY
0건
abort/new
CANCELED
취소·새 요청
RROR
·재시도
성 알림·다음 전이를 적는다

여기서 상태 기계는 학습과 기획을 위한 모델입니다. 반드시 특정 프레임워크나 상태 관리 라이브러리를 도입하라는 뜻이 아닙니다. 작은 화면은 문자열 하나와 render 함수로도 같은 원칙을 구현할 수 있습니다.

7.1. 여섯 상태

상태	의미	화면에 필요한 것	다음 행동
`idle`	아직 시작 전	입력·실행 버튼·기본 안내	제출
`loading`	처리 중	진행 안내·필요 시 취소	기다림·취소
`success`	결과 있음	결과·개수·후속 행동	열기·수정·다시 검색
`empty`	정상 처리됐지만 0건	빈 이유·조건 변경 제안	검색어 수정
`error`	완료하지 못함	오류 범주·재시도·문의 단서	재시도·돌아가기
`canceled`	의도적으로 중단	취소 확인·복구 가능성	다시 실행

7.2. 상태와 boolean 여러 개

```
// 조합 충돌이 생기기 쉬운 예
let isLoading = false;
let hasError = false;
let isEmpty = false;

// 한 시점의 주요 상태를 명시하는 예
let viewState = 'idle';
```

boolean 세 개는 `isLoading=true`이면서 `hasError=true`인 모순 조합을 만들 수 있습니다. 한 번에 하나여야 하는 주요 상태는 상태값 하나로 표현하고, 데이터·필터·선택값 같은 별도 차원은 따로 둡니다.

7.3. 상태마다 네 가지를 적습니다

```
이름: loading
진입: submit 후 요청 시작
표시: “자료를 찾는 중입니다” + 취소 버튼
종료: data>0→success, data=0→empty, 오류→error, abort→canceled
```

BIG IDEA 02

상태 이름은 개발자 내부 변수만이 아니라 사용자가 지금 무엇을 기다리고 보고 다시 할 수 있는지를 약속하는 화면 계약입니다.

30초 확인

요청은 200이고 JSON도 정상인데 `items`가 빈 배열입니다. `success`와 `empty` 중 무엇이 학습자와 사용자에게 더 분명합니까?

정답 보기

화면 주요 상태는 `empty`가 더 분명합니다. 전송·HTTP·본문 해석은 성공했지만 표시할 업무 결과가 없다는 의미를 따로 전달합니다.

8. Promise 상태와 UI 상태를 섞지 않습니다

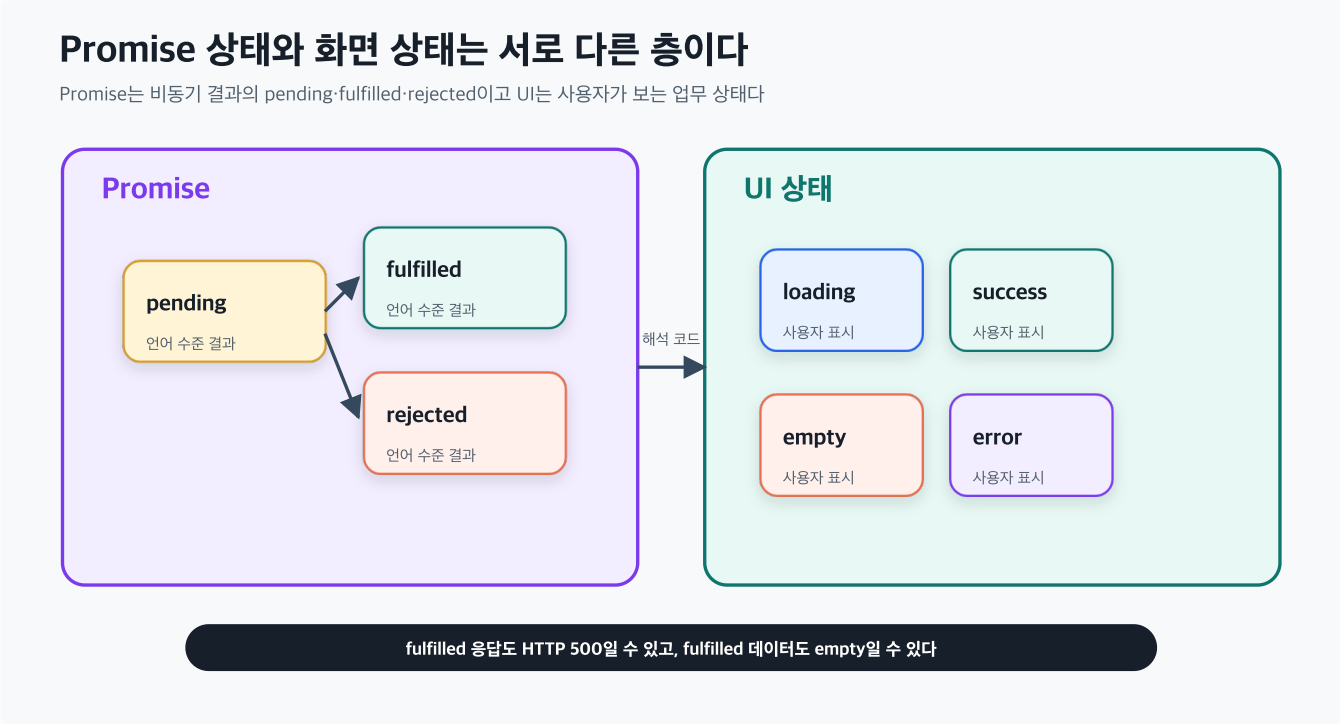
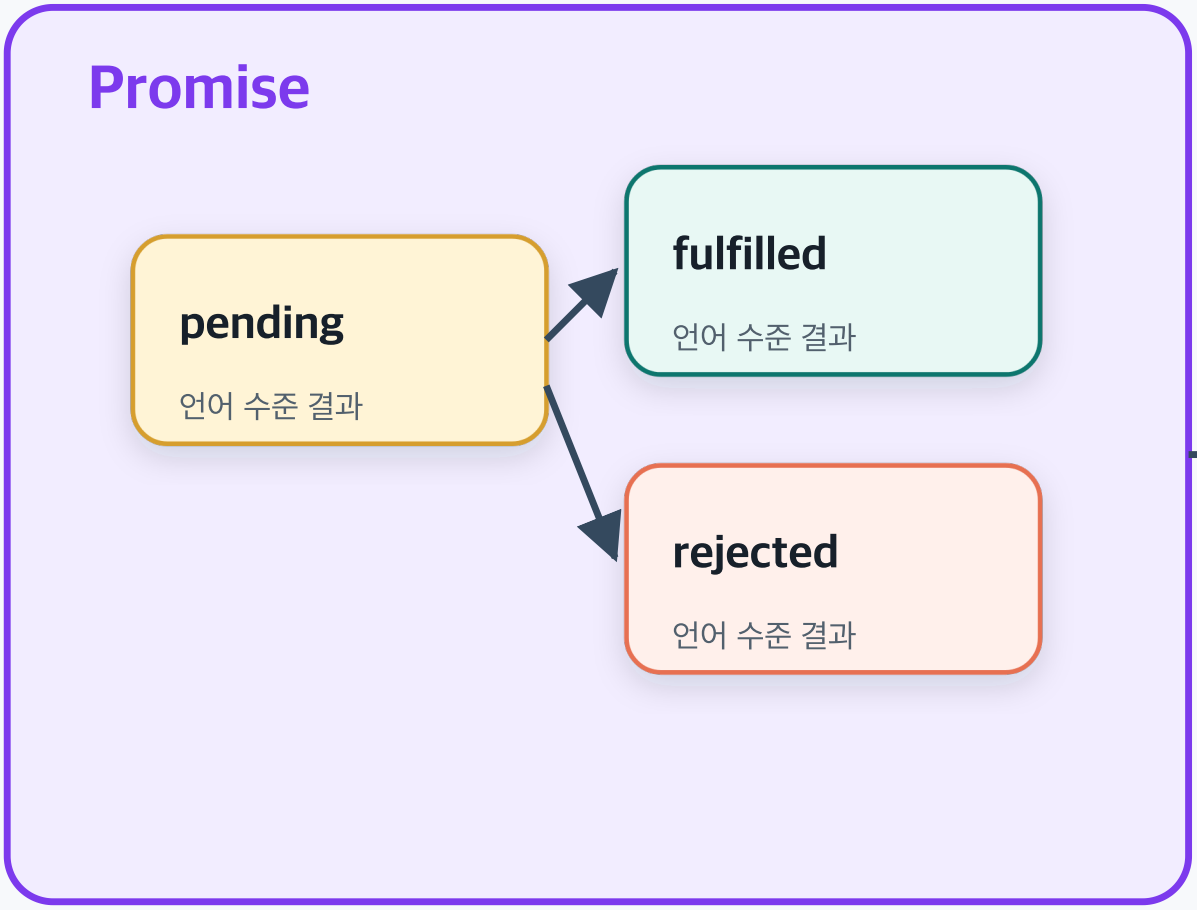
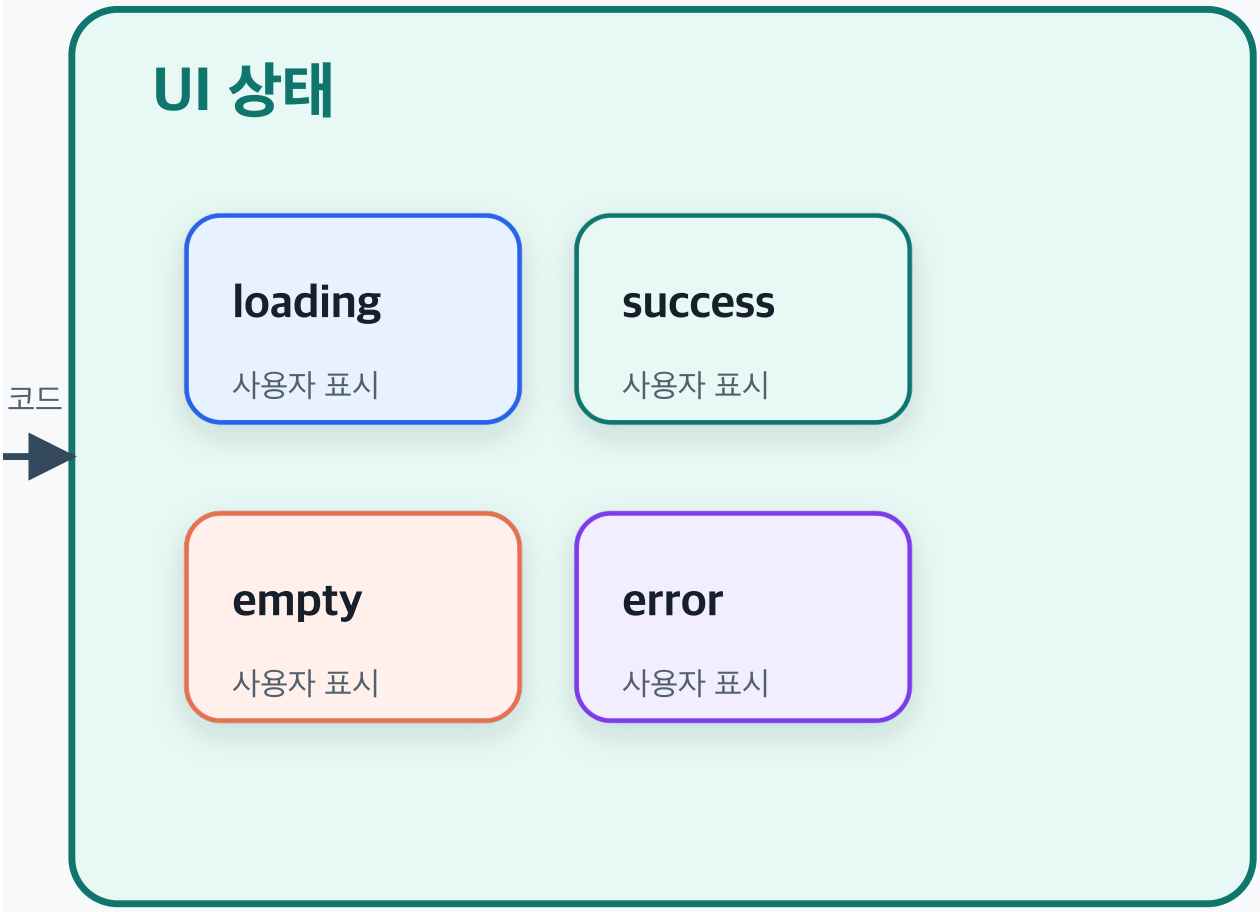


그림 6. Promise는 언어 수준의 비동기 결과이고 UI 상태는 사용자가 보는 업무 결과입니다. 둘은 해석 코드로 연결됩니다.



fulfilled 응답도 HTTP 500일 수 있고



, fulfilled 데이터도 empty일 수 있다

8.1. Promise의 세 상태

- `pending` : 아직 이행도 거부도 되지 않음
- `fulfilled` : 값과 함께 이행됨
- `rejected` : 이유와 함께 거부됨

fulfilled와 rejected를 합쳐 settled라고 합니다. 한 번 settled된 Promise는 다시 다른 상태로 바뀌지 않습니다.

```
const promise = fetch('/api/items');

promise
  .then(response => response.json())
  .then(data => render(data))
  .catch(error => showError(error))
  .finally(() => hideProgress());
```

8.2. `async` · `await`

`async function`은 Promise를 반환합니다. `await`는 해당 async 함수의 실행을 Promise 정착까지 잠시 멈추고, 브라우저 전체와 사용자 입력을 동기적으로 얼리는 명령이 아닙니다.

```
async function loadItems() {
  try {
    setState('loading');
    const response = await fetch('/api/items');
    const data = await response.json();
    setState(data.items.length ? 'success' : 'empty');
  } catch (error) {
    setState(error.name === 'AbortError' ? 'canceled' : 'error');
  }
}
```

8.3. `finally()`는 성공 화면을 뜻하지 않습니다

`finally()`는 이행·거부 모두에서 정리 작업을 실행합니다. loading 표시 제거·버튼 복구·timer 정리에 적합하지만, success를 설정하면 오류까지 성공으로 덮을 수 있습니다.

Promise 결과	가능한 UI 결과
pending	loading
fulfilled Response 200	success·empty·업무 오류

Promise 결과	가능한 UI 결과
fulfilled Response 500	HTTP 판정 뒤 error
rejected NetworkError	error
rejected AbortError	canceled

“Promise 성공”이라는 표현은 무엇이 fulfilled됐는지 불분명합니다. `fetch Promise가 Response로 fulfilled`, `HTTP 200`, `본문 3건`, `UI success`처럼 층을 붙여 씁니다.

9. fetch 결과는 네 개의 판정문을 통과합니다

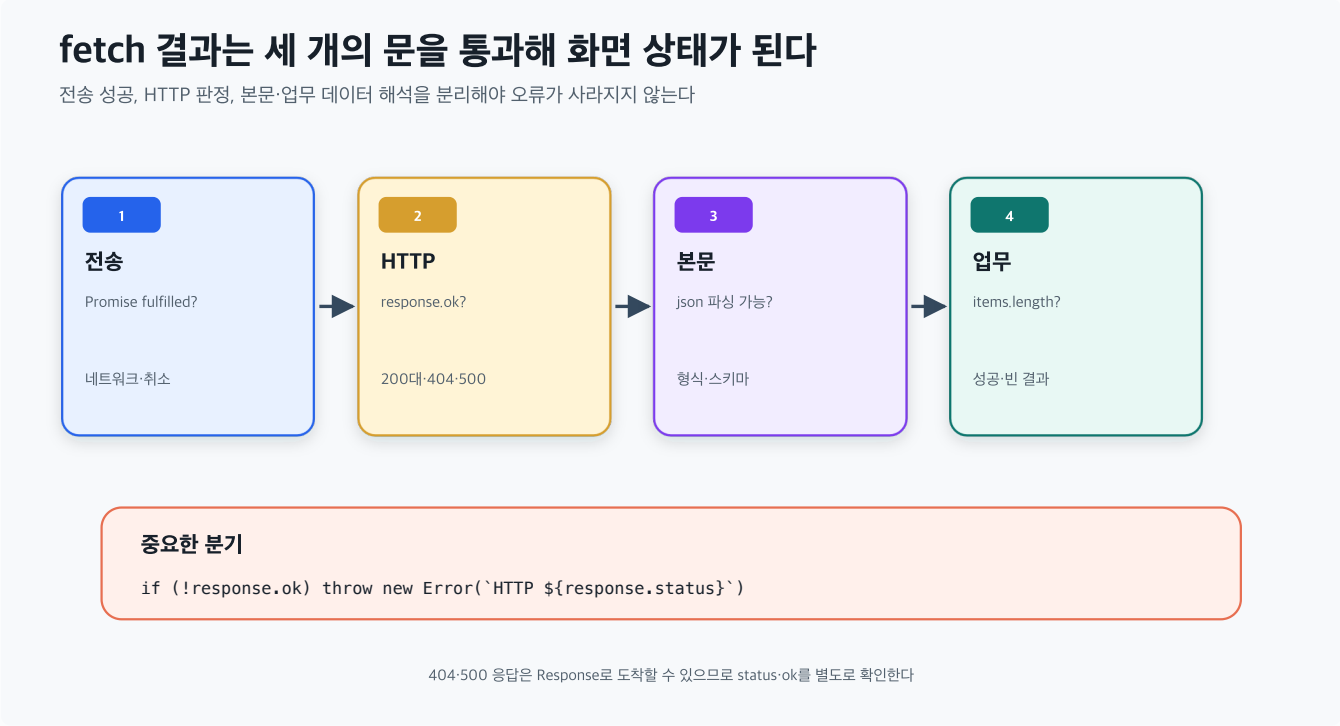
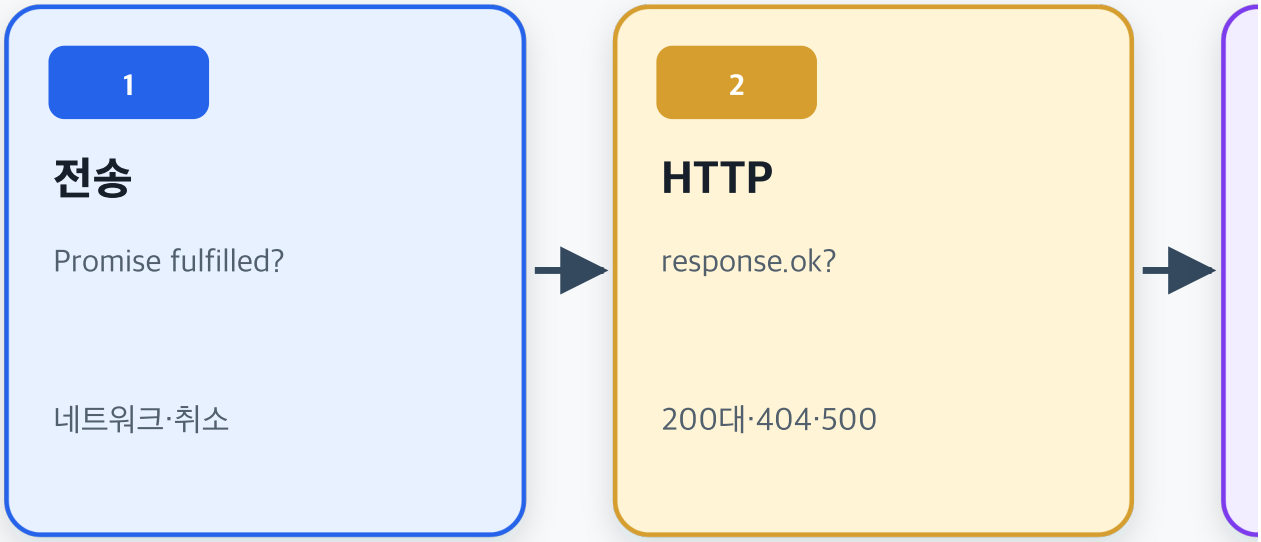


그림 7. 전송·HTTP·본문·업무 데이터의 네 문을 분리하면 “응답은 왔는데 왜 오류인가”를 설명할 수 있습니다.



중요한 분기

```
if (!response.ok) throw new Error(`HTTP ${response.st
```


```
tatus}`)
```

9.1. 1문 · 전송

`fetch()`는 Promise를 반환합니다. 네트워크 오류·중단·요청 구성 실패 등에서는 rejected가 될 수 있습니다. 반면 HTTP 404·500은 서버에서 HTTP Response를 받은 것이므로 일반적으로 fetch Promise가 Response로 fulfilled됩니다.

9.2. 2문 · HTTP

```
const response = await fetch('/api/items');

if (!response.ok) {
  throw new Error(`HTTP ${response.status}`);
}
```

`response.ok`는 status가 200~299 범위인지 나타냅니다. 500 Response가 도착했다고 fetch 자체를 네트워크 실패로 부르지 않습니다. 애플리케이션이 `ok`나 `status`를 보고 오류 흐름으로 바꿉니다.

9.3. 3문 · 본문

HTTP 200이어도 빈 본문·잘못된 JSON·예상과 다른 스키마일 수 있습니다.

```
const data = await response.json();

if (!Array.isArray(data.items)) {
  throw new Error('Expected items array');
}
```

9.4. 4문 · 업무 데이터

```
if (data.items.length === 0) {
  setState('empty');
} else {
  setState('success');
}
```

결제 거절·권한 부족·처리 대기처럼 HTTP 200 본문 안에 업무 실패가 담기는 API도 있습니다. API 계약을 확인하고 업무 코드·필드를 판정해야 합니다.

9.5. 네 문장 기록법

층	기록 예
전송	fetch Promise는 Response로 fulfilled
HTTP	status 500, `ok=false`
본문	오류 JSON을 정상 해석
업무·UI	서버 오류로 분류해 error·재시도 표시

30초 확인

`fetch('/api').catch(showError)`만 있고 500 Response에서 catch가 실행되지 않습니다. 첫 개선은 무엇입니까?

정답 보기

fulfilled된 Response의 `ok` 또는 `status`를 확인하고, 비정상 HTTP를 명시적으로 오류 흐름으로 전환합니다. 그 뒤 본문과 업무 상태를 별도 판정합니다.

10. event loop는 task 뒤 microtask를 비우고 렌더 기회를 봅니다

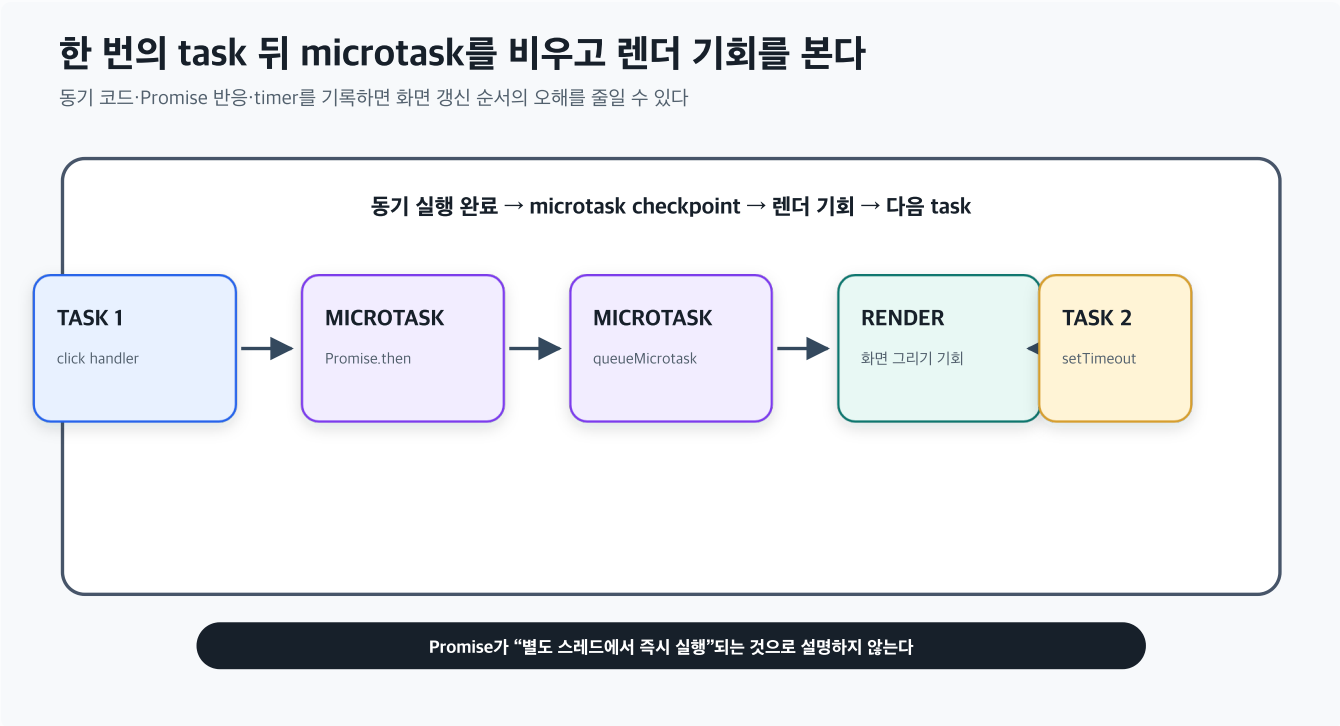
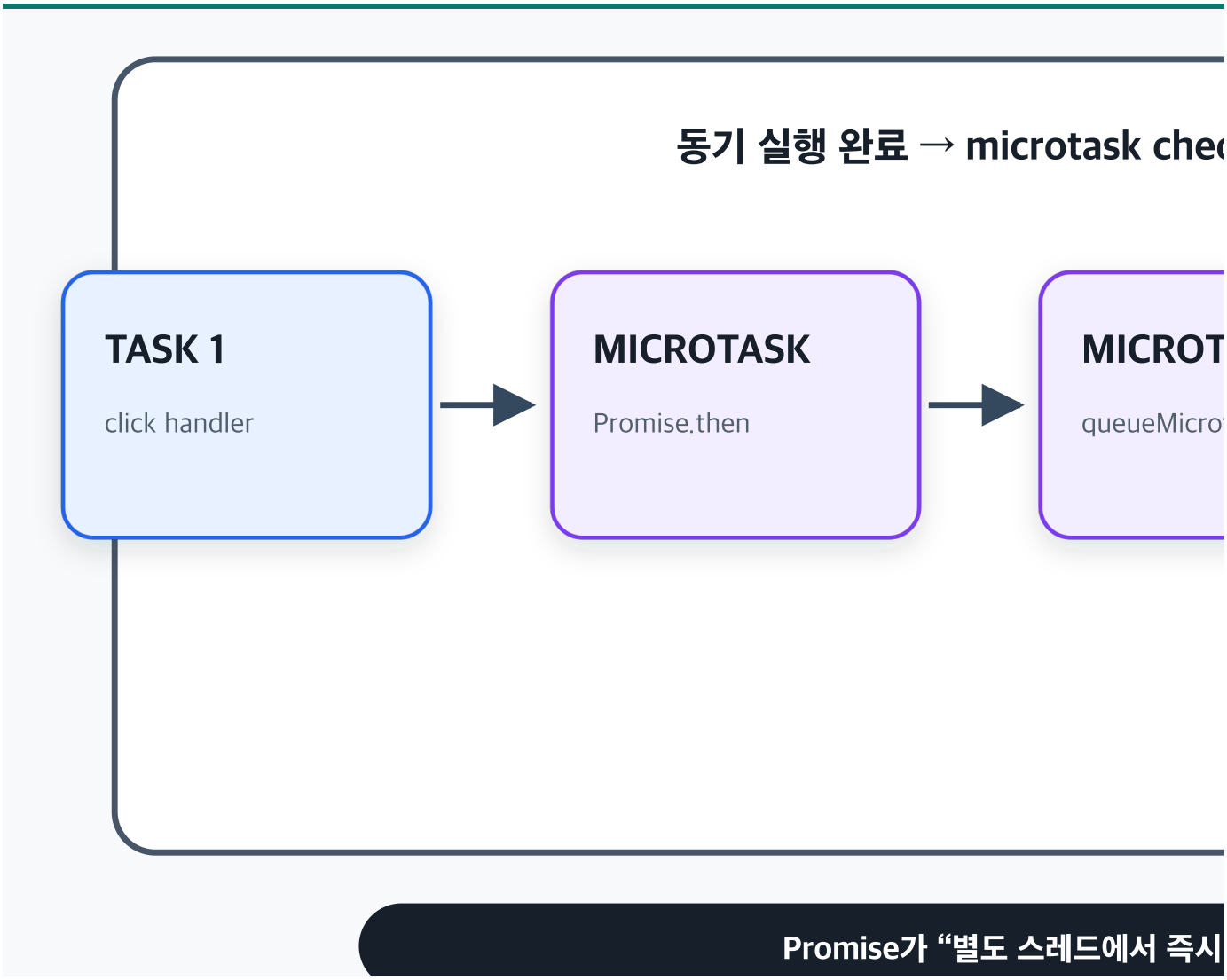
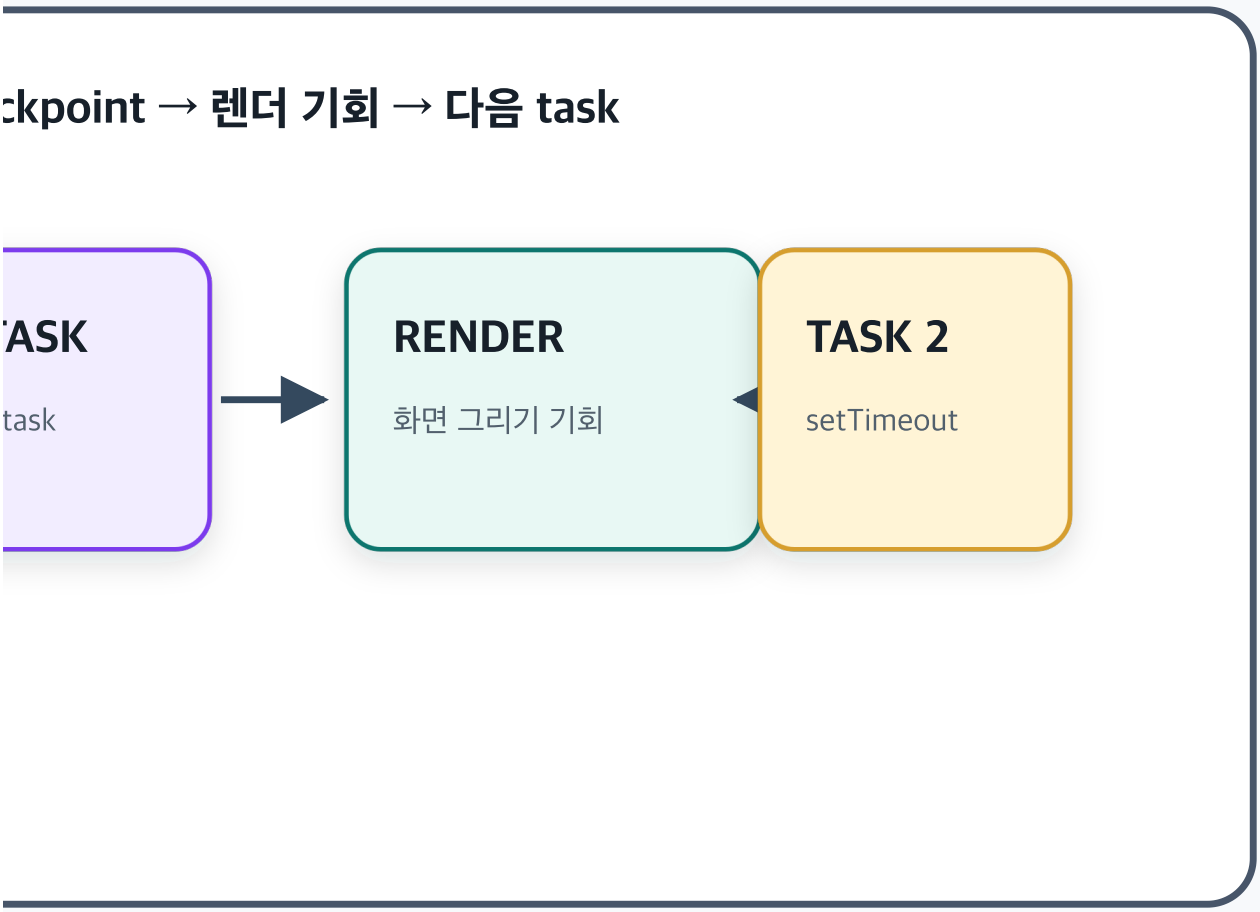


그림 8. 한 task는 끝까지 실행되고, 이어진 microtask checkpoint 뒤 브라우저가 렌더할 기회를 볼 수 있습니다.


동기 실행 완료 → microtask chec
TASK 1
click handler
MICROTASK
Promise.then
MICROT
queueMicro
Promise가 “별도 스레드에서 즉시



실행"되는 것으로 설명하지 않는다

10.1. 초급 실행 순서

```
1. click task의 handler를 끝까지 실행
2. Promise 반응 등 대기 microtask를 처리
3. 브라우저가 렌더 기회를 판단
4. timer 같은 다음 task 실행
```

브라우저의 실제 스케줄링에는 여러 task source와 렌더 조건이 있으므로 “항상 이 한 줄만 그대로 실행된다”는 뜻은 아닙니다. 초급 분석에서는 한 task가 중간에 다른 click task와 섞이지 않고 끝난다는 점, Promise 반응이 다음 timer task보다 먼저 처리될 수 있다는 점을 잡습니다.

10.2. 실행 순서 예제

```
console.log('A');

setTimeout(() => console.log('B · task'), 0);

Promise.resolve().then(() => console.log('C · microtask'));

console.log('D');
```

일반적인 출력은 `A → D → C → B` 입니다. 현재 script task가 끝난 뒤 Promise microtask가 실행되고, 그 뒤 timer task가 실행됩니다.

10.3. 로딩 표시가 안 보이는 이유

```
setState('loading');
doVeryHeavySynchronousWork();
setState('success');
```

긴 동기 작업이 같은 task에서 메인 스레드를 점유하면 브라우저가 중간 loading 상태를 그릴 렌더 기회를 얻지 못할 수 있습니다. 상태 변수는 바뀌었지만 픽셀은 loading을 건너뛰고 success만 보일 수 있습니다.

10.4. microtask도 너무 길 수 있습니다

Promise callback이 계속 새 microtask를 추가하면 다음 task와 렌더가 늦어질 수 있습니다. “비동기니까 화면이 무조건 부드럽다”가 아니라, 각 콜백의 실행 시간과 큐 누적을 확인합니다.

BIG IDEA 03

상태 값이 바뀐 시점과 사용자가 새 픽셀을 본 시점은 같다고 단정할 수 없습니다. 긴 task와 microtask 누적은 입력과 렌더를 늦춥니다.

11. 늦은 이전 응답이 최신 화면을 덮을 수 있습니다

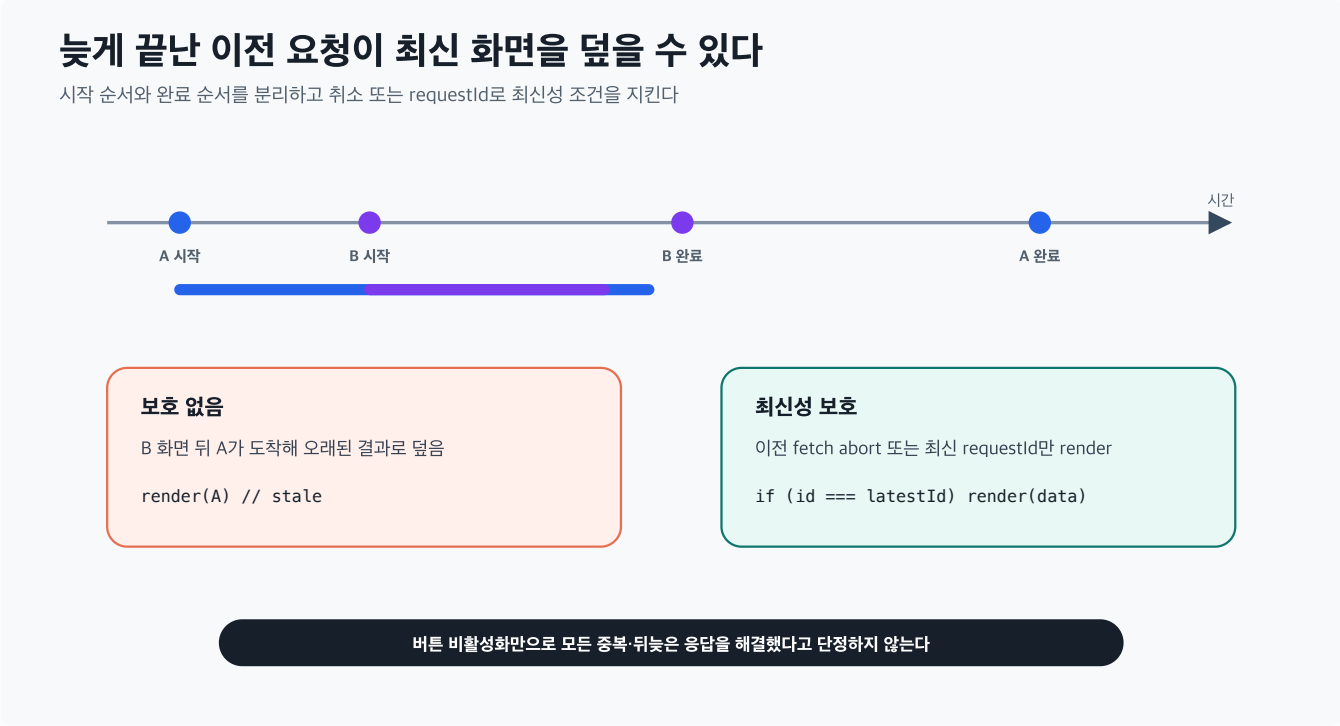
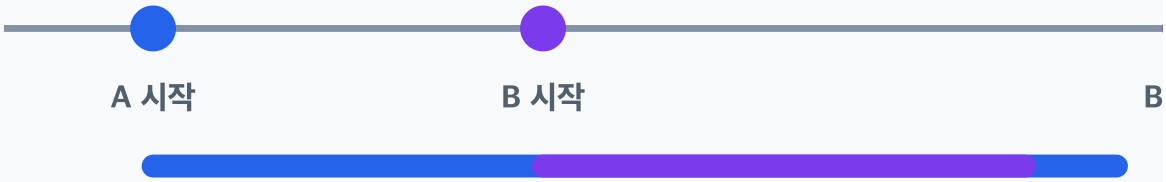


그림 9. 요청 시작 순서와 완료 순서는 다를 수 있으므로 render 직전에 최신 의도인지 확인합니다.



보호 없음

B 화면 뒤 A가 도착해 오래된 결과로 덮음

```
render(A) // stale
```

버튼 비활성화만으로 모든 중복·뒤늦은

최신성 보호

이전 fetch abort 또는 최신 requestId만 render

```
if (id === latestId) render(data)
```

은 응답을 해결했다고 단정하지 않는다

11.1. 경쟁 상태 재현

```
0ms   A 검색 시작 · 느림
120ms B 검색 시작 · 빠름
620ms B 완료 → 최신 결과 B render
1600ms A 완료 → 보호 없으면 이전 결과 A가 화면을 덮음
```

오류가 항상 발생하는 것이 아니므로 “가끔 이전 결과가 보인다”처럼 보고됩니다. 지연을 인위적으로 다르게 만들거나 Network throttling을 사용해 완료 순서를 뒤집습니다.

11.2. 최신 request ID 확인

```
let latestRequestId = 0;

async function search(query) {
  const requestId = ++latestRequestId;
  setState('loading');

  const response = await fetch(`/api/items?q=${encodeURIComponent(query)}`);
  if (!response.ok) throw new Error(`HTTP ${response.status}`);
  const data = await response.json();

  if (requestId !== latestRequestId) return;
  render(data);
}
```

이 방어는 오래된 응답의 화면 반영을 막지만 이전 요청 자체를 중단하지는 않습니다.

11.3. `AbortController`

```
let activeController;

async function search(query) {
  activeController?.abort();
  activeController = new AbortController();

  try {
    const response = await fetch(`/api/items?q=${encodeURIComponent(query)}`, {
      signal: activeController.signal
    });
    // HTTP·본문·최신성 판정
  } catch (error) {
    if (error.name === 'AbortError') {
      setState('canceled');
      return;
    }
    setState('error');
  }
}
```

abort는 클라이언트가 더 이상 결과를 원하지 않는다는 신호입니다. 서버가 이미 시작한 저장·결제·생성 작업까지 되돌리는 보장은 아닙니다. 중요한 쓰기 작업은 서버의 멱등성·중복 처리 규칙과 함께 설계합니다.

11.4. 중복 행동 검수표

현상	클라이언트 대책	서버·업무 대책
검색 연속 입력	debounce·이전 abort·최신 ID	캐시·요청 제한
저장 버튼 연속 클릭	진행 중 잠금·결과 복구	멱등성 키·중복 판정
화면 이탈 뒤 응답	cleanup·abort·render guard	불필요 작업 취소 가능성
재시도 중 이전 응답	세대·request ID	요청 상태 조회·업무 ID

30초 확인

버튼을 disabled로 바꾸면 모든 중복 제출이 해결됩니까?

정답 보기
아닙니다. 키보드·다른 탭·재전송·네트워크 재시도·서버 중복 처리까지 모두 막는 보장은 없습니다. 화면 잠금은 사용자 피드백이고, 중요한 업무는 서버 멱등성과 상태 판정이 함께 필요합니다.

12. 상태 변화는 보이고 들려야 합니다

W3C WCAG 2.2의 상태 메시지 이해 자료는 초점을 옮기지 않고도 작업 결과·진행·오류 같은 상태를 프로그램이 판별할 수 있도록 제공하는 것을 설명합니다. 검색 중·결과 개수·빈 결과·오류는 대표적인 상태 메시지입니다.

```
<div role="status" aria-live="polite">
  자료를 찾는 중입니다.
</div>
```

12.1. 상태별 문구

상태	피해야 할 문구	더 분명한 문구
loading	처리 중	자료를 찾는 중입니다
success	완료	18개의 결과를 찾았습니다
empty	없음	검색 결과가 없습니다. 검색어를 바꾸어 보세요
error	오류	서버 응답을 받지 못했습니다. 다시 시도하세요
canceled	실패	요청을 취소했습니다. 다시 검색할 수 있습니다

12.2. 초점을 함부로 옮기지 않습니다

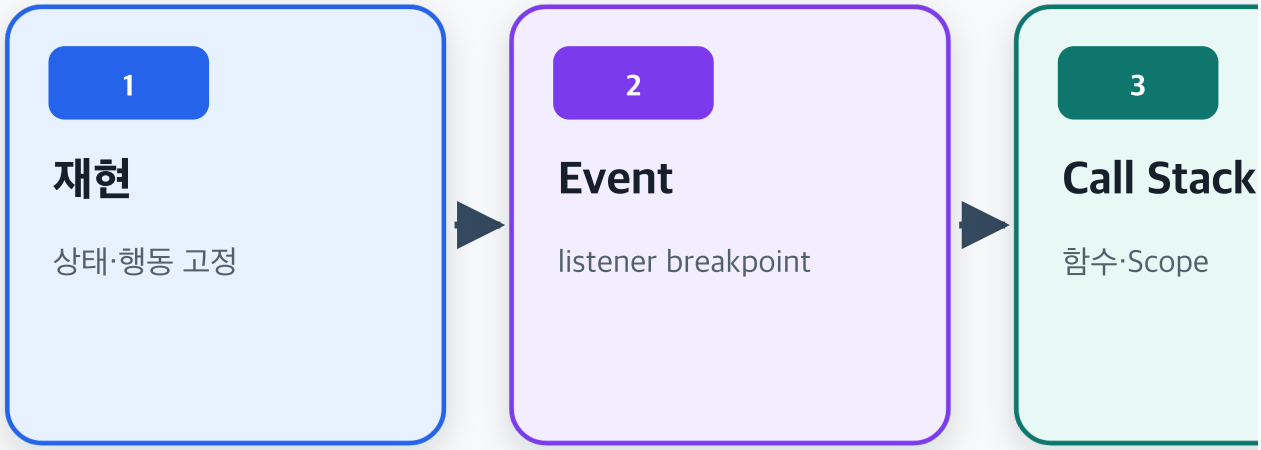
상태 갱신만으로 결과 제목에 초점을 강제로 이동하면 키보드·스크린리더 사용자가 현재 위치를 잃을 수 있습니다. 상태 메시지를 알리고 현재 조작 위치를 유지하는 것이 기본입니다. 전체 화면 전환·대화상자 같은 명확한 문맥 변화에서는 별도의 초점 관리가 필요합니다.

12.3. 색만으로 구분하지 않습니다

빨강·초록 배경만 바꾸지 않고 텍스트·아이콘·제목을 함께 제공합니다. loading indicator에는 무엇을 기다리는지, error에는 가능한 다음 행동이 있어야 합니다.

13. Chrome DevTools로 행동에서 DOM까지 추적합니다

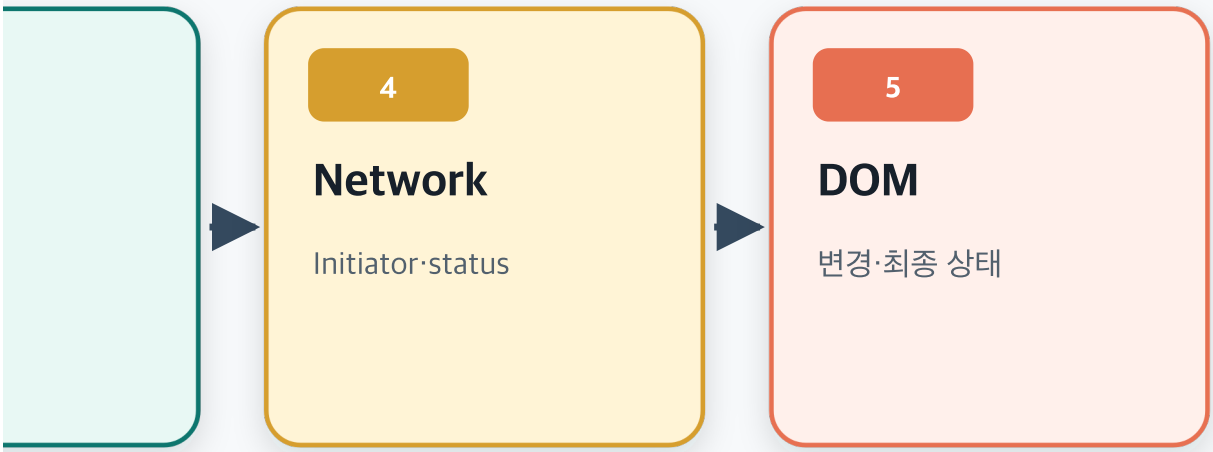
그림 10. Console 로그를 무작정 늘리지 않고 breakpoint·Call Stack·Network Initiator·DOM을 시간순 증거로 연결합니다.



한 건의 추적 기록

click → onSearch() → state=loading → GET /api/items → 500 → state

Console 로그만 모으지 않고 breakpoint·s



=error → “다시 시도” 표시

stack·request·DOM을 같은 시각선에 둔다

13.1. 재현 조건 고정

```
화면·URL 패턴: _____________________________
로그인 역할·데이터: _____________________
시작 상태: idle·success·기타 _____________
행동: _____________________________________
예상 결과: _________________________________
실제 결과: _________________________________
```

13.2. Event listener breakpoint

1. DevTools > Sources를 엽니다.
2. Event Listener Breakpoints에서 Mouse > click 또는 Control > submit을 켭니다.
3. 행동을 실행합니다.
4. 멈춘 코드에서 Call Stack을 따라 앱 함수로 이동합니다.
5. Scope에서 event·상태·입력 값을 확인합니다.

브라우저·라이브러리 내부 코드에서 먼저 멈추더라도 Call Stack의 앱 파일 프레임을 찾습니다. 필요하면 ignore list를 활용합니다.

13.3. XHR/fetch breakpoint

Sources > XHR/fetch Breakpoints에 URL의 안정적인 일부를 추가하면 요청 생성 직전에 멈출 수 있습니다. 요청이 “어디서 시작됐는가”를 찾을 때 유용합니다.

13.4. Network 증거

탭·열	보는 값	답하는 질문
Headers	Method·URL·Status	어떤 요청과 HTTP 결과인가
Payload	query·body 구조	어떤 입력이 전송됐는가
Preview·Response	본문 형태	기대 스키마와 데이터인가
Initiator	코드·호출 연쇄	누가 요청을 만들었는가
Timing	queue·wait·download	어느 구간이 느린가

Network의 민감한 헤더·쿠키·본문은 공유하지 않습니다. URL도 내부 경로와 식별자를 마스킹하고 Method·Status·패턴만 기록할 수 있습니다.

13.5. DOM breakpoint와 예외 중단

화면이 바뀌지만 코드를 찾기 어렵다면 Elements에서 대상 요소에 DOM breakpoint를 걸어 하위 트리·속성 변화에서 멈춥니다. 오류가 catch에서 사라진다면 Sources의 exception breakpoint를 켜 throw 지점을 확인합니다.

13.6. 한 건의 추적 기록

```
click → onSearchClick() → setState('loading')
→ GET /api/items → HTTP 500 → checkResponse() throw
→ catch → setState('error') → “다시 시도” 표시
```

스크린샷 한 장보다 “행동 → 멈춘 함수 → 상태 전·후 → 요청 → 응답 → 화면” 한 줄이 재현과 수정에 강합니다. 각 화살표에 DevTools에서 확인한 증거를 하나씩 붙입니다.

14. 실습 · 여섯 시나리오를 직접 실행합니다

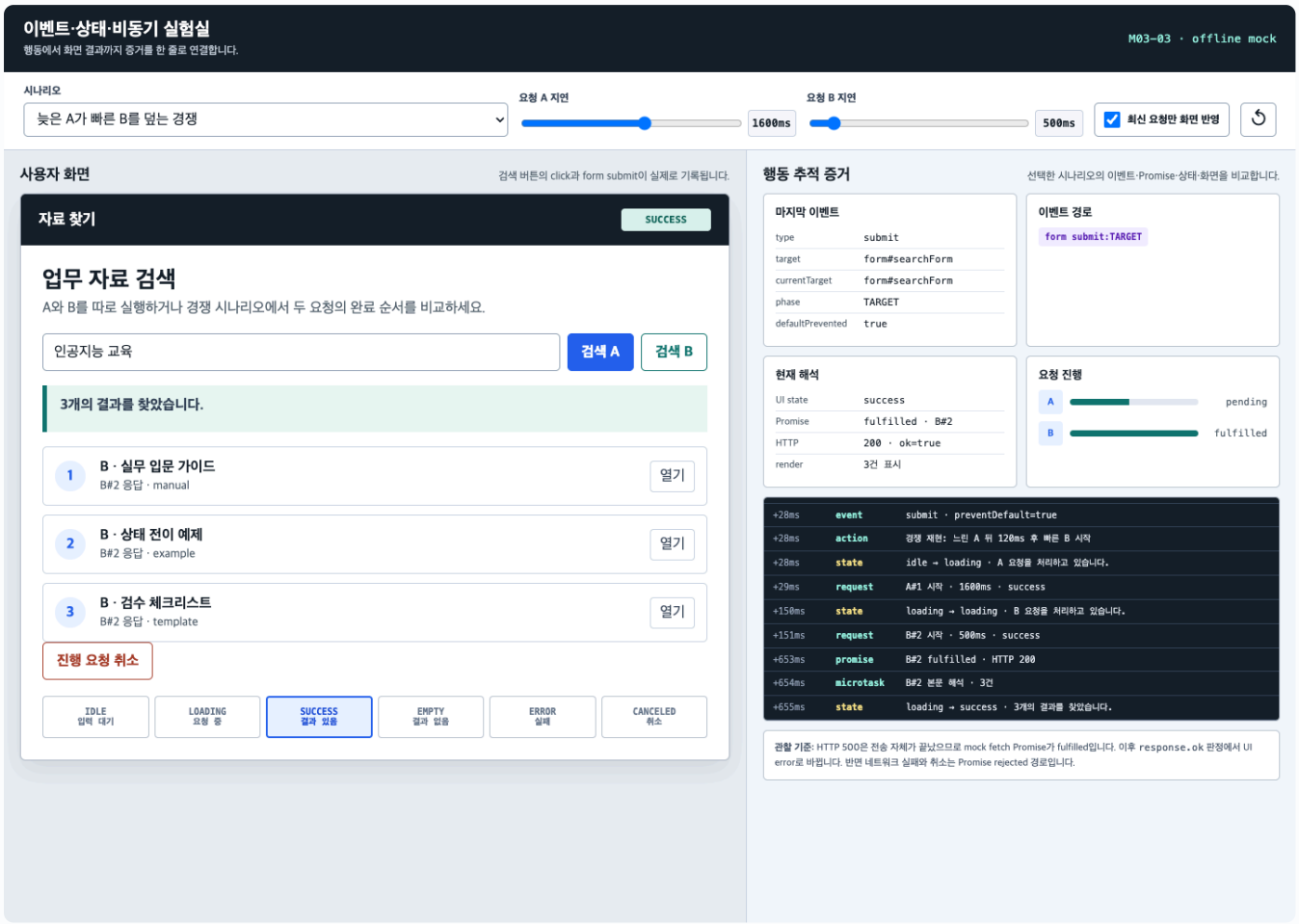


그림 11. 실험실은 실제 click·submit 경로, 화면 상태, mock Promise·HTTP 판정, 요청 A·B의 완료 순서를 한 화면에 보여 줍니다.

14.1. 준비 파일

- L03-03 이벤트·상태·비동기 실험실
- L03-03 실습 안내
- T03-03 화면 행동·상태 추적표
- JavaScript 이벤트·상태·비동기 용어집

14.2. 여섯 번 실행

순서	시나리오	반드시 볼 증거
1	정상 성공	submit defaultPrevented·loading→success·3건
2	정상 빈 결과	Promise fulfilled·HTTP 200·empty

순서	시나리오	반드시 볼 증거
3	HTTP 500	fetch fulfilled·`ok=false`·error
4	네트워크 실패	Promise rejected·HTTP 없음·error
5	늦은 응답 경쟁	B 먼저 표시·A stale 또는 덮어쓰기
6	사용자 취소	AbortError·canceled·다시 실행

14.3. 실습 완료 산출물

1. 일곱 단계 행동 추적 3건
2. 상태 여섯 개와 전이 조건
3. Promise·HTTP·본문·업무 판정 비교표
4. 보호 전·후 늦은 응답 결과
5. 오류·경쟁 개선 요청 각 1건

실제 서비스 실습은 공개 가능한 화면에서만 진행합니다. Authorization·Cookie·개인정보·내부 호스트·결제 정보가 보이는 Network 캡처는 공유 PDF와 yeoncore.ai 게시물에 포함하지 않습니다.

15. 인쇄용 화면 행동 명세 워크시트

15.1. 한 행동의 일곱 단계

단계	기록
행동	
이벤트 type·target	
capture·target·bubble 경로	
handler·기본 동작	
UI 상태 전→후	
Promise·HTTP·본문·업무 데이터	
화면 메시지·다음 행동	

15.2. 상태 전이 표

이전 상태	사건·조건	다음 상태	화면 표시	다음 행동
idle		loading		
loading	결과 있음	success		
loading	0건	empty		
loading	오류	error		
loading	취소	canceled		
error·empty·canceled	재시도	loading		

15.3. 비동기 판정

```
요청 ID: _____  시작 시각: _____
Method·URL 패턴: _____
Promise: pending → fulfilled·rejected _____
HTTP status·ok: _____
본문 형식·스키마: _____
업무 데이터 판정: success·empty·domain error _____
render 전 최신성 검사: _____
최종 UI 상태·메시지: _____
```

15.4. 개선 요청

```
[재현] _____
[증거] event _____ → handler _____ → state _____→_____
       → request _____ → Promise _____ → HTTP _____ → render _____
[영향] _____
[개선] _____
[완료 기준] _____
```

16. 셀프 테스트 · 정답을 가리고 풉니다

문제 1

부모 목록 listener에서 자식 버튼을 눌렀을 때 `event.target`과 `event.currentTarget`은 각각 무엇을 가리킬 수 있습니까?

문제 2

`preventDefault()`와 `stopPropagation()`의 목적 차이를 한 문장씩 쓰십시오.

문제 3

HTTP 500 Response에서 `fetch()` Promise가 fulfilled될 수 있는 이유는 무엇입니까?

문제 4

Promise `fulfilled`와 UI `success`가 같은 말이 아닌 사례를 두 개 쓰십시오.

문제 5

`idle → loading → success`만 정의한 검색 화면에서 빠진 주요 상태 세 개는 무엇입니까?

문제 6

다음 코드의 일반적인 출력 순서를 쓰십시오.

```
console.log('A');
setTimeout(() => console.log('B'), 0);
Promise.resolve().then(() => console.log('C'));
console.log('D');
```

문제 7

느린 A 요청 뒤 빠른 B 요청을 실행했습니다. B가 먼저 표시된 뒤 A가 화면을 덮는 현상의 이름과 방어책 두 가지는 무엇입니까?

문제 8

`AbortController.abort()`가 결제·저장 같은 서버 작업까지 반드시 되돌린다고 볼 수 없는 이유는 무엇입니까?

문제 9

Chrome에서 click 처리 함수와 요청 생성 위치를 찾는 breakpoint 두 종류는 무엇입니까?

문제 10

검색 결과가 0건으로 바뀌었습니다. 초점을 결과 영역으로 강제 이동하지 않고도 보조 기술에 상태를 전달하는 방법 한 가지를 쓰십시오.

17. 셀프 테스트 정답과 해설

정답 1

`target`은 실제 시작점인 자식 버튼 또는 버튼 내부 요소를 가리킬 수 있고, `currentTarget`은 현재 listener가 등록된 부모 목록을 가리킵니다.

정답 2

`preventDefault()`는 취소 가능한 브라우저 기본 동작을 막고, `stopPropagation()`은 뒤 이벤트 경로 대상으로 전달되는 것을 막습니다. 하나가 다른 하나를 자동으로 대신하지 않습니다.

정답 3

HTTP 500은 서버로부터 HTTP Response를 받은 결과이므로 전송 Promise는 Response로 fulfilled될 수 있습니다. 애플리케이션이 `response.ok` 또는 `status`를 판정해 error 흐름으로 바꿉니다.

정답 4

fulfilled된 Response가 HTTP 500일 수 있고, fulfilled된 200 응답의 데이터가 0건이라 UI가 empty일 수 있습니다. 본문 파싱·업무 규칙에서 오류가 날 수도 있습니다.

정답 5

`empty`, `error`, `canceled`입니다. 화면 목적에 따라 권한 부족·부분 성공·처리 대기 같은 별도 상태가 더 필요할 수 있습니다.

정답 6

일반적으로 `A → D → C → B`입니다. 현재 task의 동기 코드가 끝난 뒤 Promise microtask가 실행되고, 다음 timer task가 이어집니다.

정답 7

비동기 경쟁 상태 또는 stale response 문제입니다. 이전 요청을 `AbortController`로 중단하고, render 직전에 request ID가 최신인지 확인할 수 있습니다.

정답 8

abort는 클라이언트의 요청·응답 처리를 중단하는 신호이며 서버가 이미 업무 처리를 시작했을 수 있습니다. 중요한 쓰기는 서버 멱등성·중복 처리·상태 조회와 함께 설계해야 합니다.

정답 9

Event Listener Breakpoint와 XHR/fetch Breakpoint입니다. Call Stack·Network Initiator를 함께 보면 행동 함수와 요청 시작점을 연결할 수 있습니다.

정답 10

예를 들어 `role="status"` 또는 적절한 `aria-live="polite"` 영역의 텍스트를 “검색 결과가 없습니다”로 갱신합니다. 단순 상태 갱신 때문에 초점을 불필요하게 옮기지 않습니다.

오답 진단표

틀린 문제	다시 볼 층	다시 볼 그림
1·2	이벤트 경로·기본 동작	그림 2·3·4
3·4	Promise·HTTP·업무 상태	그림 6·7
5	화면 상태 전이	그림 5
6	task·microtask·렌더	그림 8
7·8	경쟁·취소·서버 보장	그림 9
9	DevTools 증거	그림 10
10	상태 메시지	12장

18. 한 장 요약과 다음 단계

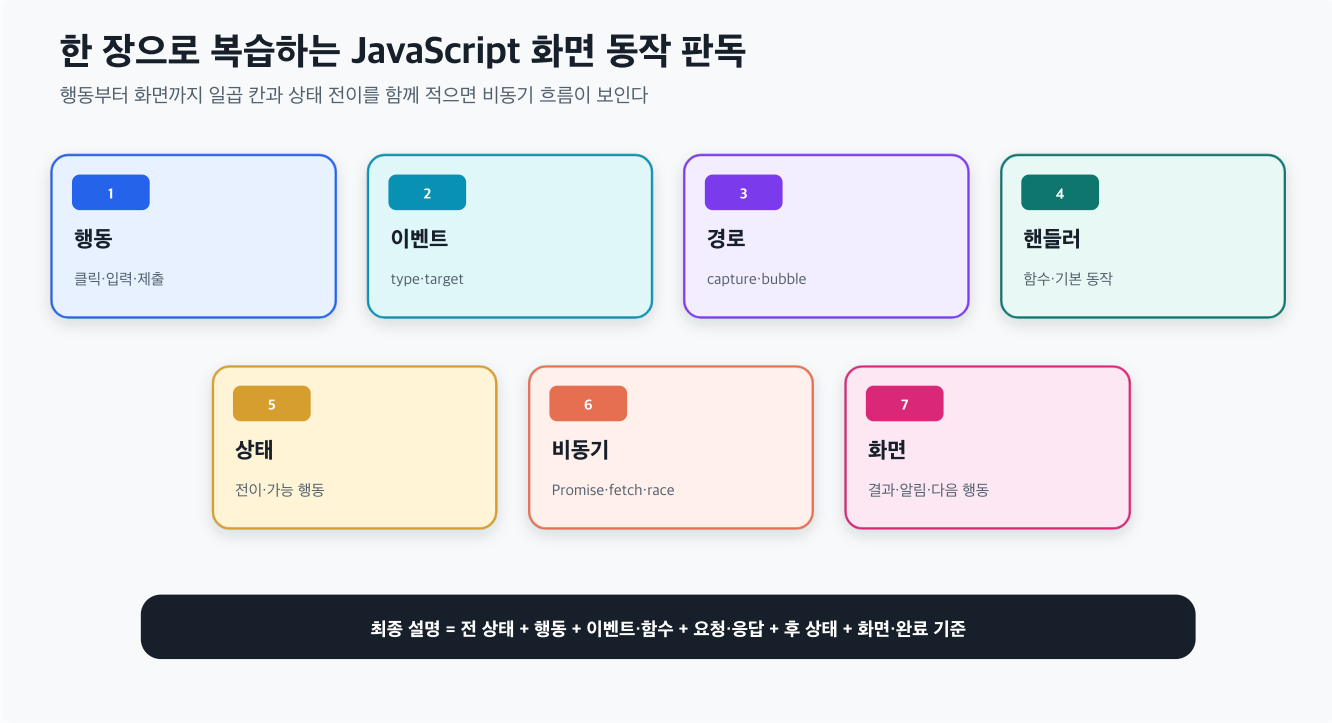


그림 12. 행동·이벤트·경로·핸들러·상태·비동기·화면의 일곱 칸을 채우면 화면 동작을 재현 가능한 증거로 바꿀 수 있습니다.


1
행동
클릭·입력·제출
2
이벤트
type·target
5
상태
전이·가능 행동
6
비동기
Promise·fetch·rac
최종 설명 = 전 상태 + 행동 + 이벤트·함수

3

경로

capture·bubble

4

핸들러

함수·기본 동작

e

7

화면

결과·알림·다음 행동

+ 요청·응답 + 후 상태 + 화면·완료 기준

18.1. 60초 판독 순서

```
1. 행동과 시작 화면 상태를 고정한다.
2. event type·target·currentTarget·phase를 찾는다.
3. handler와 브라우저 기본 동작을 구분한다.
4. UI 상태의 전·후와 화면 메시지를 기록한다.
5. Promise·HTTP·본문·업무 데이터 네 층을 판정한다.
6. 늦은 응답·중복·취소 순서를 시험한다.
7. 사용자가 보는 결과와 다음 행동을 완료 기준으로 쓴다.
```

18.2. 산출물 품질 체크

- [ ] “안 됨” 대신 행동·상태·요청 조건을 썼습니다.
- [ ] target과 currentTarget을 구분했습니다.
- [ ] 기본 동작 취소와 전파 중지를 구분했습니다.
- [ ] success·empty·error·canceled를 서로 다른 화면 상태로 정의했습니다.
- [ ] fetch fulfilled와 HTTP 성공을 구분했습니다.
- [ ] 늦은 이전 응답과 빠른 연속 행동을 시험했습니다.
- [ ] 상태 메시지가 시각·보조 기술 모두에 전달됩니다.
- [ ] 공유 증거에서 토큰·개인정보·내부 주소를 제거했습니다.

18.3. 다음 매뉴얼

다음은 **M03-04 사용자 흐름과 화면 상태 설계하기**입니다. 이번 매뉴얼에서 익힌 행동·사건·상태 전이를 여러 화면의 진입·분기·완료·이탈·복구 경로로 확장하고, 화면별 정상·빈 결과·오류·권한 상태를 정의합니다.

공식 참고 자료

- WHATWG DOM · EventTarget과 이벤트
- WHATWG DOM · Dispatching events
- WHATWG HTML · Web application APIs와 event loop
- WHATWG Fetch Standard · fetch method
- WHATWG Fetch Standard · AbortController 연동
- ECMAScript 2025 · Promise Objects
- Chrome DevTools · JavaScript breakpoints
- Chrome DevTools · Network 패널 참고
- W3C WCAG 2.2 · Understanding Status Messages

규격과 브라우저 도구는 바뀔 수 있습니다. 이 파일은 2026-07-15에 위 공식 자료와 Chrome 150 환경으로 검토했습니다.