KB금융그룹 | 국민의 평생 금융파트너

2024년 상반기 K-디지털 트레이닝

노드와 비동기 처리

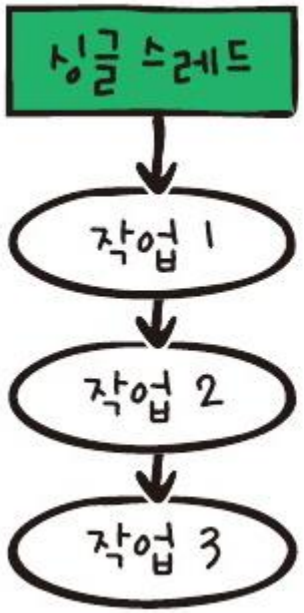
[KB] IT's Your Life

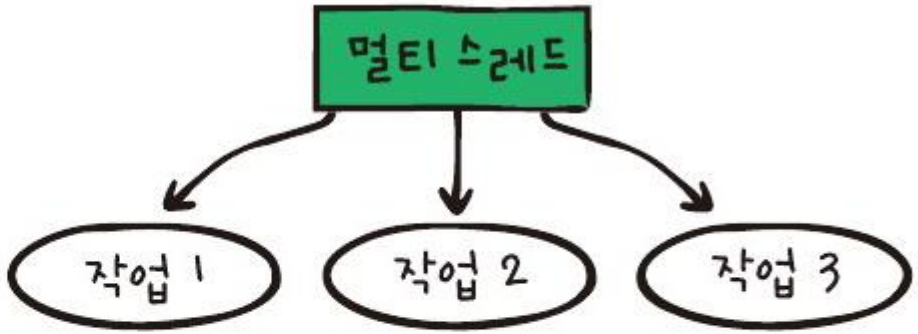
KB 국민은행

동기 처리란

- 스레드 thread
  - 작업을 처리하기 위해 자원을 사용하는 단위
  - 하나의 작업이 실행되는 최소 단위

- 싱글 스레드와 멀티 스레드





동기 처리란

- 자바스크립트는 싱글 스레드 언어

chapter05/sec01/sync.js

```
console.log('첫 번째 작업');
console.log('두 번째 작업');
console.log('세 번째 작업');
```

```
첫 번째 작업
두 번째 작업
세 번째 작업
```

비동기 처리

- 함수 호출을 하면 작업을 의뢰하고 바로 리턴
  - 호출한 함수가 끝나도 실제 작업은 시작하지 않았음
- 더 이상 실행할 코드가 없을 때 비동기 함수가 의뢰한 작업 실행
- 의뢰한 작업이 끝났음을 통지하기 위해 콜백 함수 호출
  - 작업을 의뢰할 때 콜백 함수를 같이 등록해 둠

chapter05/sec01/async-1.js

```
console.log('첫 번째 작업');
setTimeout(() => {
  console.log('두 번째 작업');
}, 3000);

console.log('세 번째 작업');
```

```
첫 번째 작업
세 번째 작업
두 번째 작업
```

chapter05/sec01/async-2.js

```
console.log('첫 번째 작업');
setTimeout(() => {
  console.log('두 번째 작업');
}, 0);

console.log('세 번째 작업');
```

```
첫 번째 작업
세 번째 작업
두 번째 작업
```

chapter05/sec01/async-3.js

```
const fs = require('node:fs');

fs.readdir('./', (err, files) => {
  if (err) {
    return console.error(err);
  }
  console.log(files);
});
console.log('Code is done.');
```

Code is done.
['sync.js', 'async-1.js', 'async-2.js', 'async-3.js']

블로킹 I/O, blocking I/O

- I/O를 수행할 때 우리의 코드 실행은 멈추됨
- 동기 함수는 블록킹 I/O로 실행
- 노드의 함수명이 ~Sync()로 끝남
- I/O의 결과 데이터가 리턴됨, 에러 발생시 예외가 발생

- 단점
  - I/O 작업은 시간이 많이 걸림
  - 그 시간동안 아무것도 할 수 없음

chapter05/sec02/blocking-1.js

```
const fs = require('fs');

const data = fs.readFileSync('example.txt'); // 블록킹 I/O
console.log(data); // 파일 읽기가 끝날 때까지 대기
console.log('코드 끝'); // 파일을 읽고 내용을 표시할 때까지 대기
```

chapter05/blocking-2.js

```
const http = require('http');

const server = http.createServer((req, res) => {
  if (req.url === '/home') {
    res.end('HOME');
  } else if (req.url === '/about') {
    res.end('ABOUT');
  } else {
    res.end('NOT FOUND');
  }
});

server.listen(3000, () => {
  console.log('http://localhost:3000 서버 실행 중');
});
```

chapter05/sec03/blocking-3.js

```
const http = require('http');

const server = http.createServer((req, res) => {
  if (req.url === '/home') {
    res.end('HOME');
  } else if (req.url === '/about') {
    for (let i = 0; i < 100; i++) {
      for (let j = 0; j < 100; j++) {
        console.log(`${i} ${j}`);
      }
    }
    res.end('ABOUT');
  } else {
    res.end('NOT FOUND');
  }
});

server.listen(3000, () => {
  console.log('http://localhost:3000 서버 실행 중');
});
```

- **논블로킹 I/O, non-blocking I/O**
  - 작업을 비동기로 처리

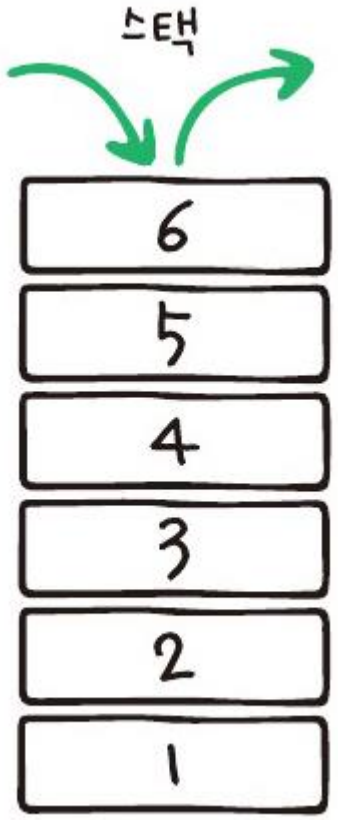
chapter05/sec02/non-blocking.js

```
const fs = require('fs');

fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    return console.log(err);
  }
  console.log(data);
});

console.log('코드 끝');
```

코드 끝
This File is Example

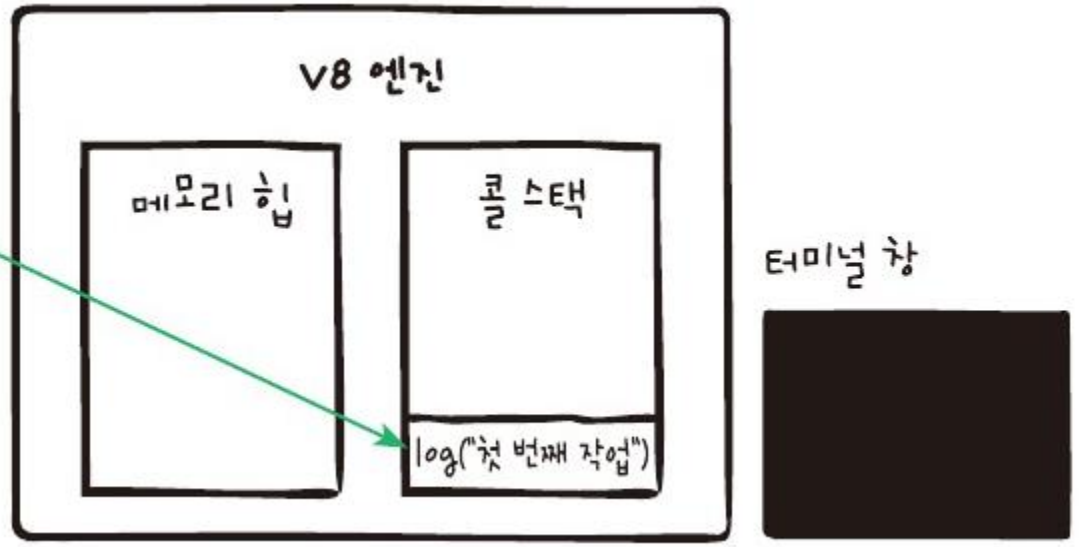
기본 처리 방법 살펴보기

- 콜 스택 call stack
  - 동기 함수 호출의 정보가 쌓이는 자료 구조

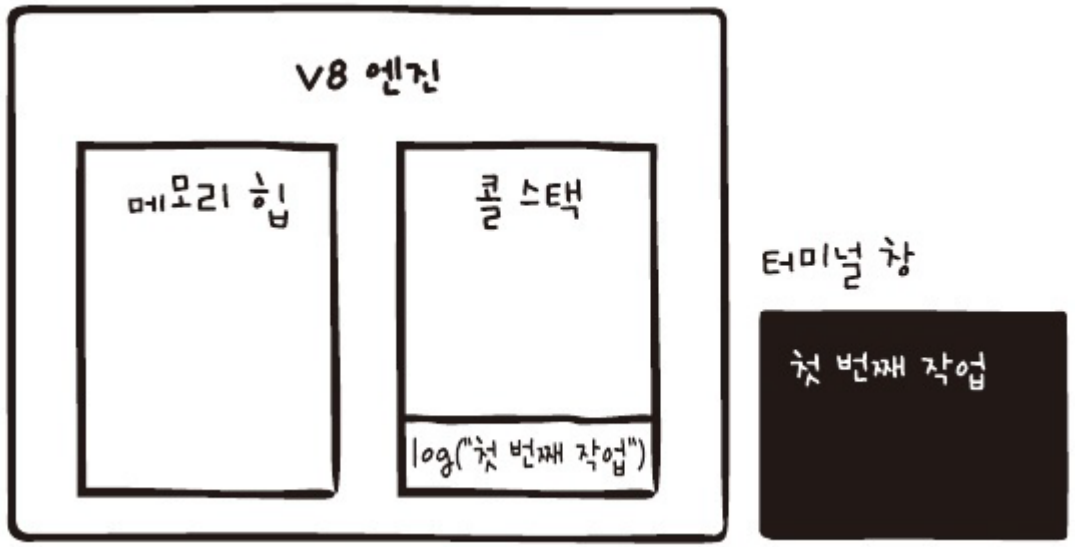


스택의 자료구조

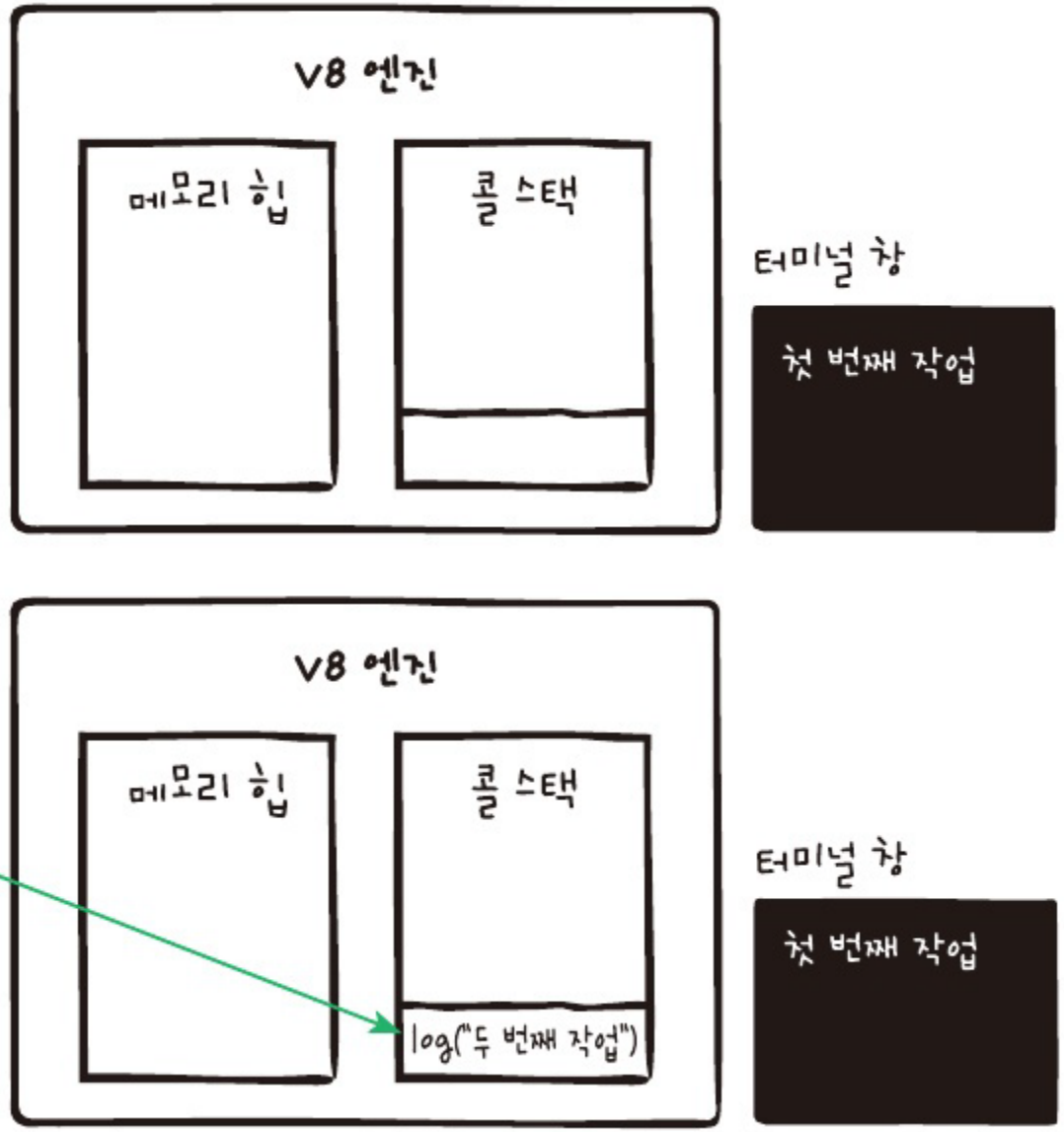
```
1   console.log("첫 번째 작업");
2   console.log("두 번째 작업");
3   console.log("세 번째 작업");
```



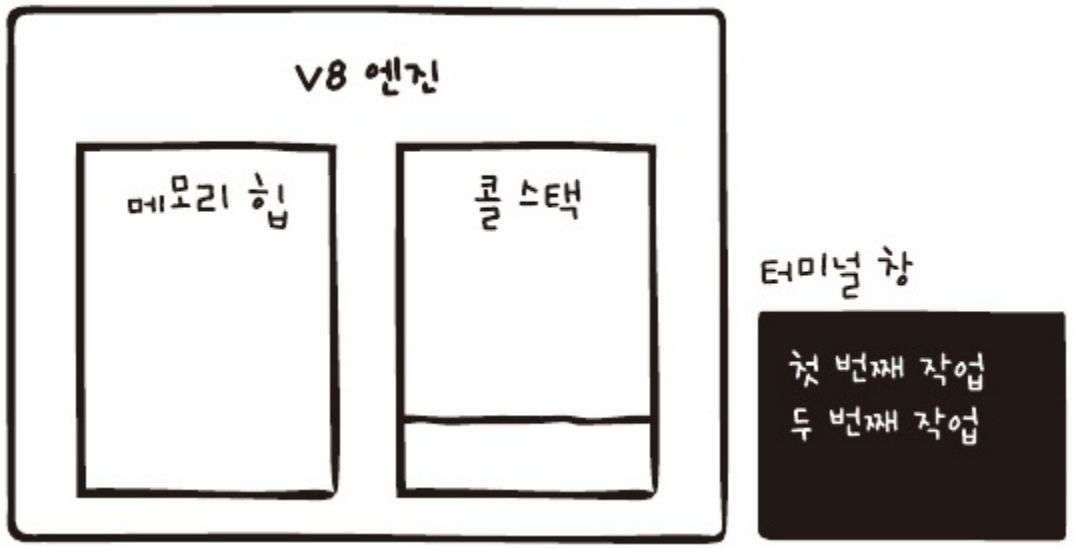

```
1   console.log("첫 번째 작업");
2   console.log("두 번째 작업");
3   console.log("세 번째 작업");
```



```
1   console.log("첫 번째 작업");
2   console.log("두 번째 작업");
3   console.log("세 번째 작업");
```

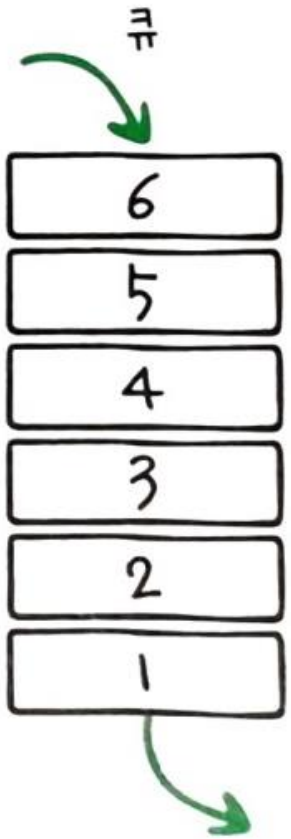


```
1   console.log("첫 번째 작업");
2   console.log("두 번째 작업");
3   console.log("세 번째 작업");
```



이벤트 루프로 비동기 처리하기

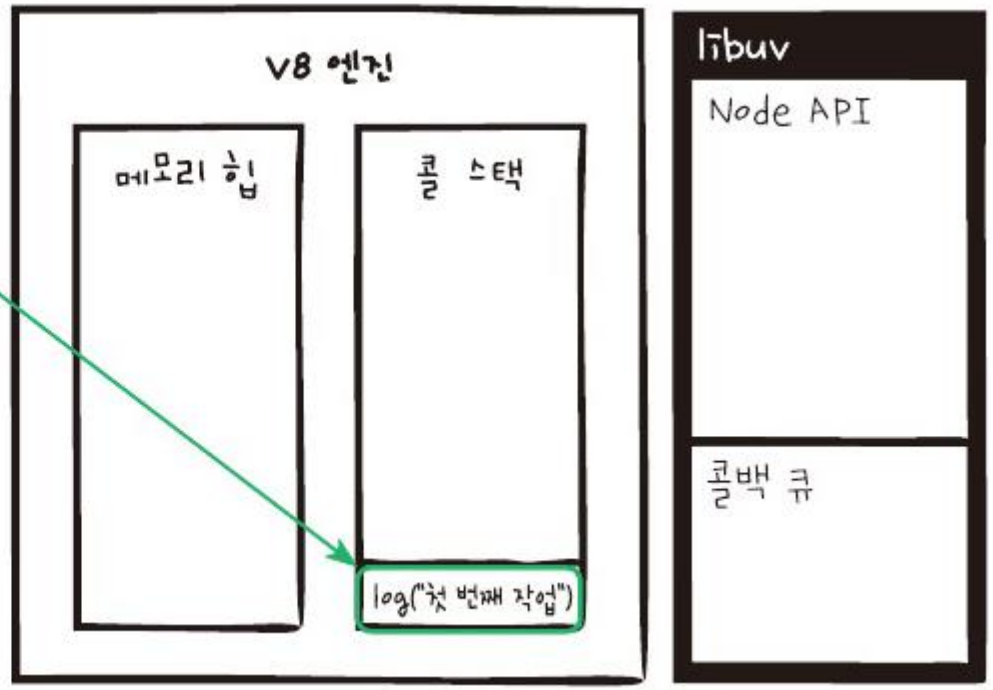
- I/O 완료 후 실행해야 할 콜백은 libuv의 큐를 이용해 관리 → 콜백 큐



큐의 자료구조

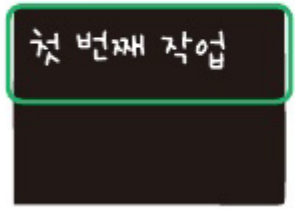
```
1  console.log("첫 번째 작업");
2  setTimeout(() => {
3    console.log("두 번째 작업");
4  }, 3000);
5  console.log("세 번째 작업");
```

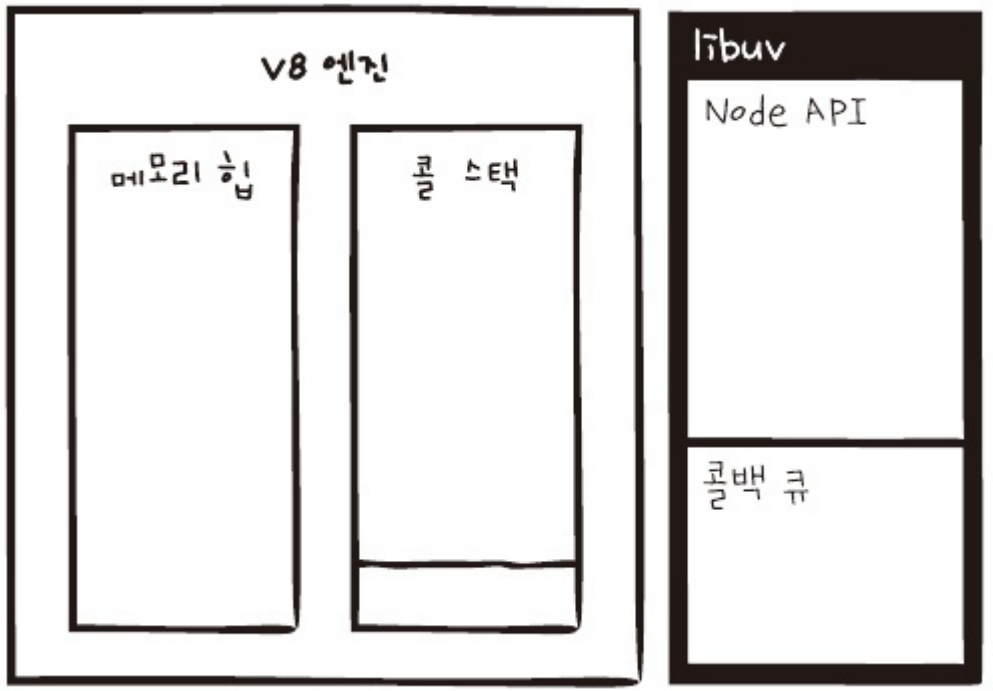
터미널 창



```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

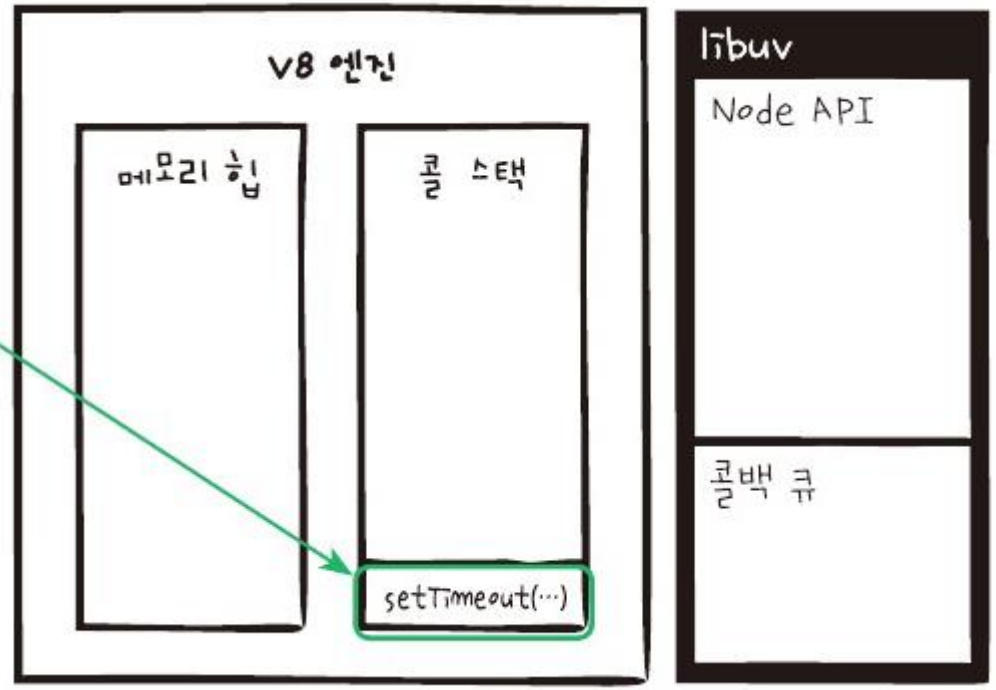




```
1  console.log("첫 번째 작업");
2  setTimeout(() => {
3    console.log("두 번째 작업");
4  }, 3000);
5  console.log("세 번째 작업");
```

터미널 창

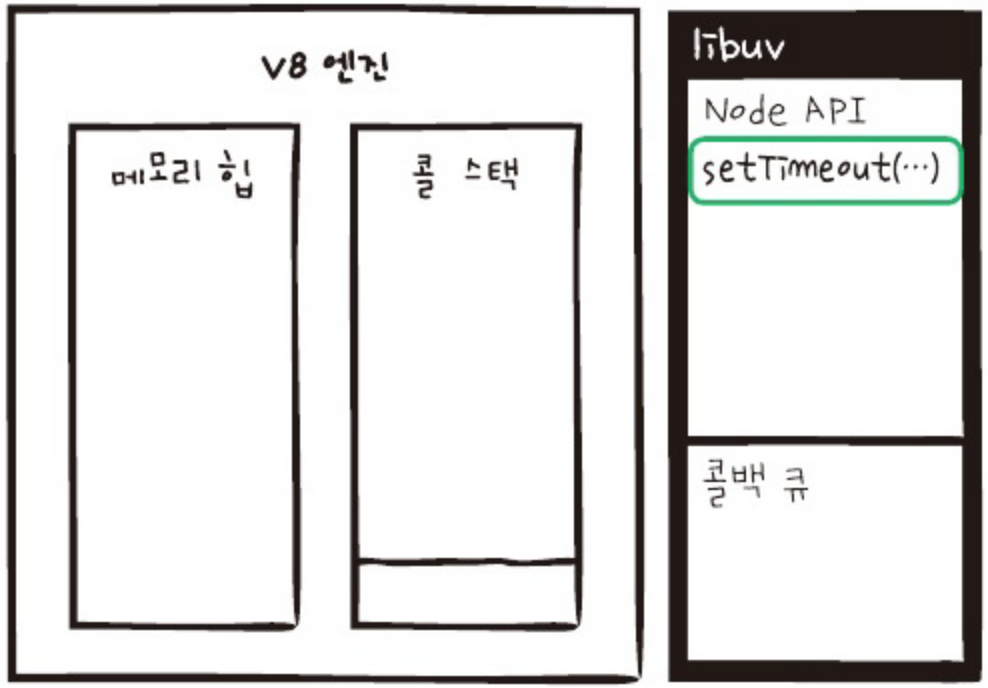
```
첫 번째 작업
```



```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

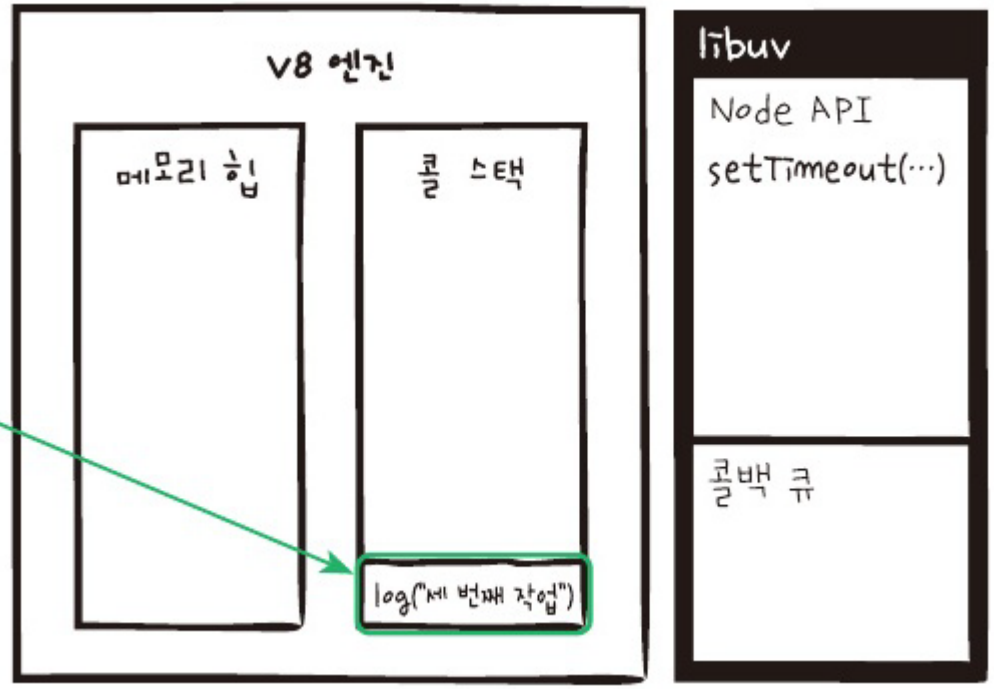
```
첫 번째 작업
```



```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

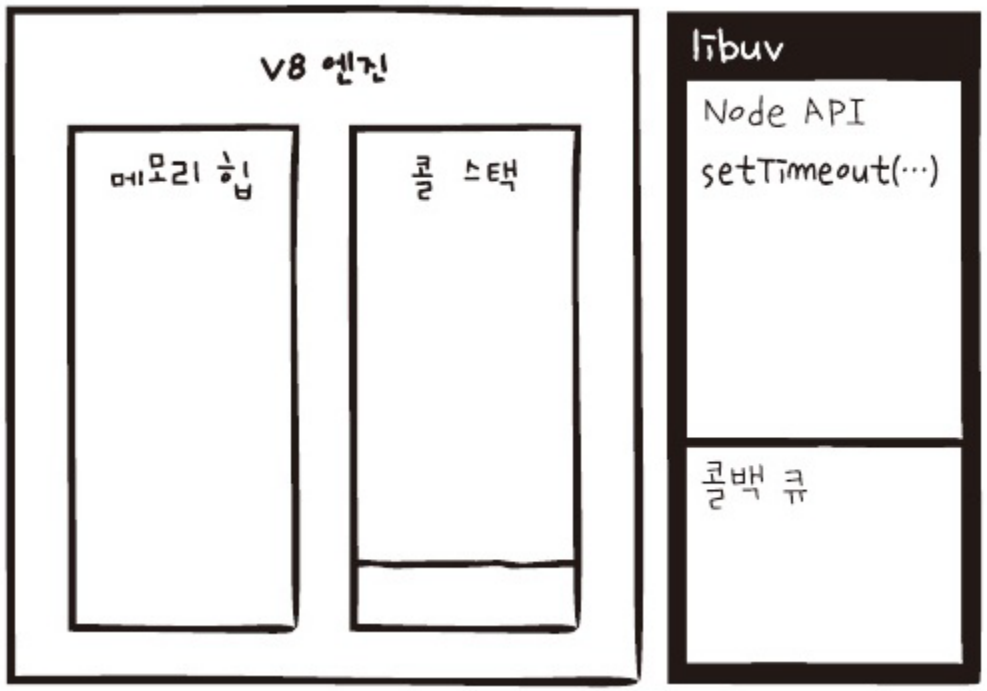
첫 번째 작업



```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

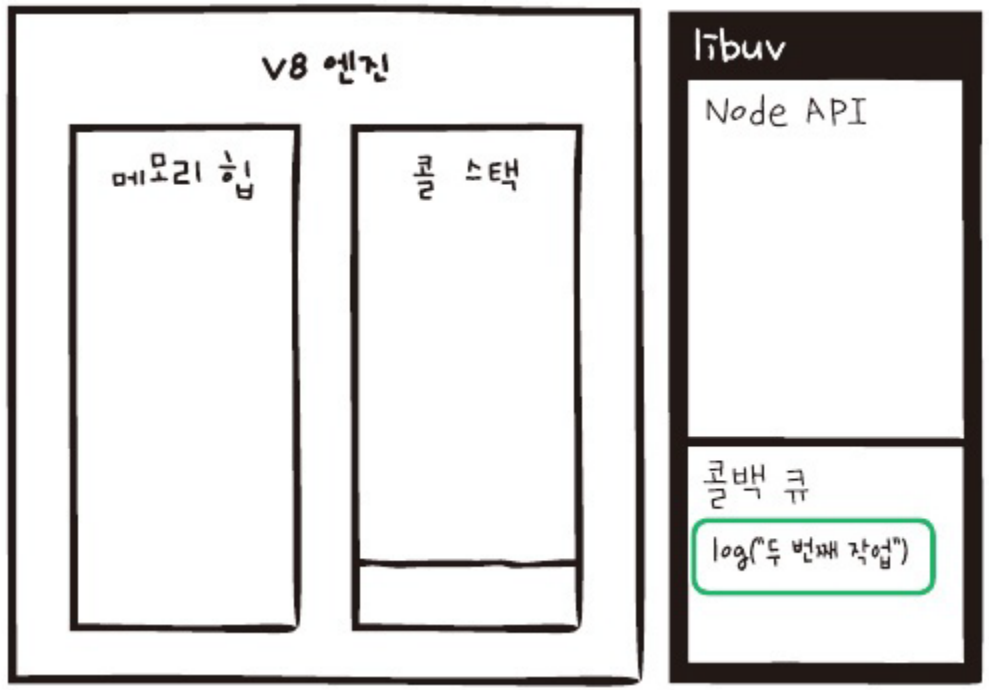
첫 번째 작업
세 번째 작업



```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

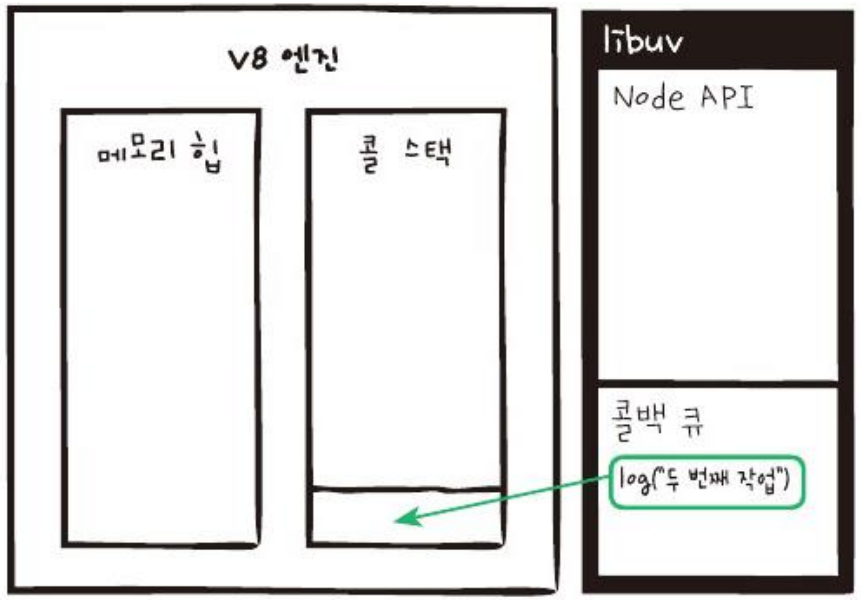
```
첫 번째 작업
세 번째 작업
```



```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

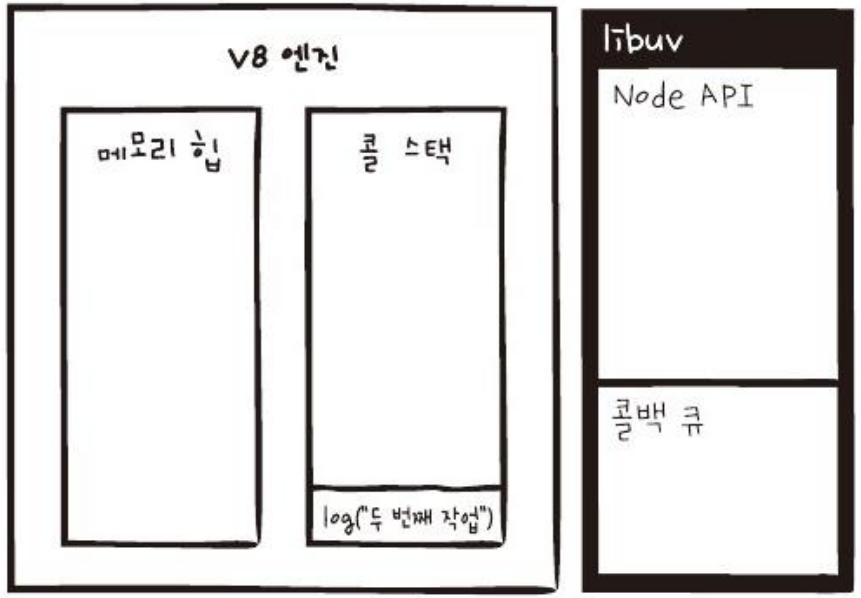
```
첫 번째 작업
세 번째 작업
```




```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

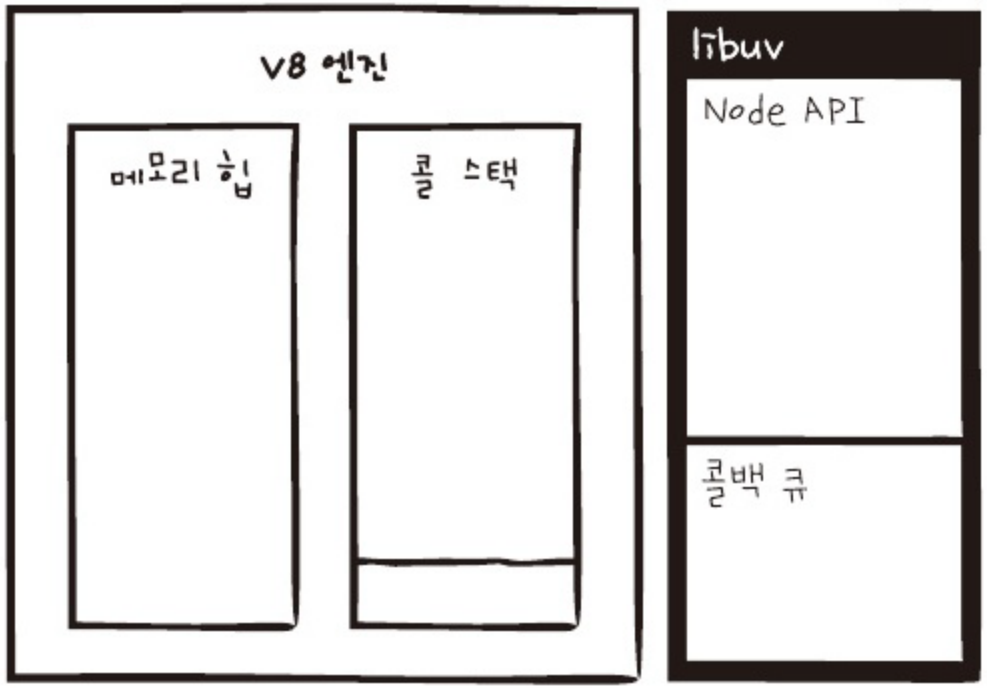
```
첫 번째 작업
세 번째 작업
```



```
1   console.log("첫 번째 작업");
2   setTimeout(() => {
3     console.log("두 번째 작업");
4   }, 3000);
5   console.log("세 번째 작업");
```

터미널 창

```
첫 번째 작업
세 번째 작업
두 번째 작업
```



콜백 함수

- 문제점
  - 콜백 함수에서 또다른 비동기 함수 호출을 하게됨
  - 콜백 함수가 또 지정됨, 그 콜백 함수에서 또다른 비동기 함수 호출을 하게됨
  - 이런 패턴이 반복되면 → 콜백 지옥!!

프라미스 promise

- 비동기 함수에서 프라미스 객체를 리턴
- 비동기 작업이 성공적으로 끝나면 then 함수를 실행
- 오류가 발생하여 실패했을 때는 catch 함수를 실행

- promise를 리턴하는 모듈 함수 가져오기

```
const fs = rquire('fs').promises;

fs.비동기함수()
  .then((result)=> { /* 성공시 실행할 코드 */})
  .catch((err)=>{ /* 실패시 실행할 코드 */})
```

chapter05/sec04/promise.js

```
const fs = require('fs').promises;

fs.readdir('./')
  .then((result) => console.log(result))
  .catch((err) => console.error(err));
```

async/await

- ECMA 2017(ES8)부터 도입된 비동기 처리 방법
- 비동기 처리를 하는 함수 앞에 async 키워드 설정
- 비동기 함수 호출시 await를 앞에 붙임
  - 비동기 함수는 반드시 Promise 객체를 리턴해야 함
- 예외 처리는 try-catch 블록으로 처리

→ 비동기 처리지만 동기 처럼 코드를 작성할 수 있음

chapter05/sec04/await.js

```
const fs = require('fs').promises;

async function readDirAsyn() {
  try {
    const files = await fs.readdir('./');  // Promise 객체를 리턴하는 비동기 함수
    console.log(files);
  } catch (err) {
    console.error(err);
  }
}

readDirAsyn();
```