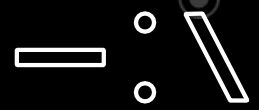
SHOUT OUR PASSION TOGETHER



PASSION

2nd Seminar

EXPERIENCE

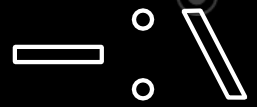
GROWTH

CHALLENGE

SHOUT OUR PASSION TOGETHER

SOPT

목차



01

Node js

NodeJS란?

1990년 Tim Berners lee가 **WEB**을 창시

Netscape라는 **웹 브라우저** 등장

Brendan Eich에 의해 **JavaScript** 등장, 동적인 체계 탑재

2004년 Gmail 등장, 이후 Gmap도 등장 (순수한 웹기술[HTML, CSS, JS]로 개발)

2008년 구글이 Chrome의 성능향상을 위해서 JavaScript Engine **V8** 개발, **오픈소스** 공개

2009년 Ryan Dal 이 자바스크립트 언어로 구현된 **Node.js** 선보임

Node js란?

WEB Browser, Javascript, NodeJS의 관계?

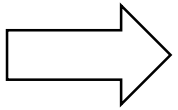
Node js란?

JavaScript는 프로그래밍 언어

Web Browser는 JavaScript을 해석하는 응용프로그램

JavaScript를 실행할 수 있는 런타임 환경

Node js란?



Web Browser에서만 동작할 수 있었던 JavaScript가
NodeJS 환경에서도 실행 가능해짐

그렇다면 Express란 무엇일까?

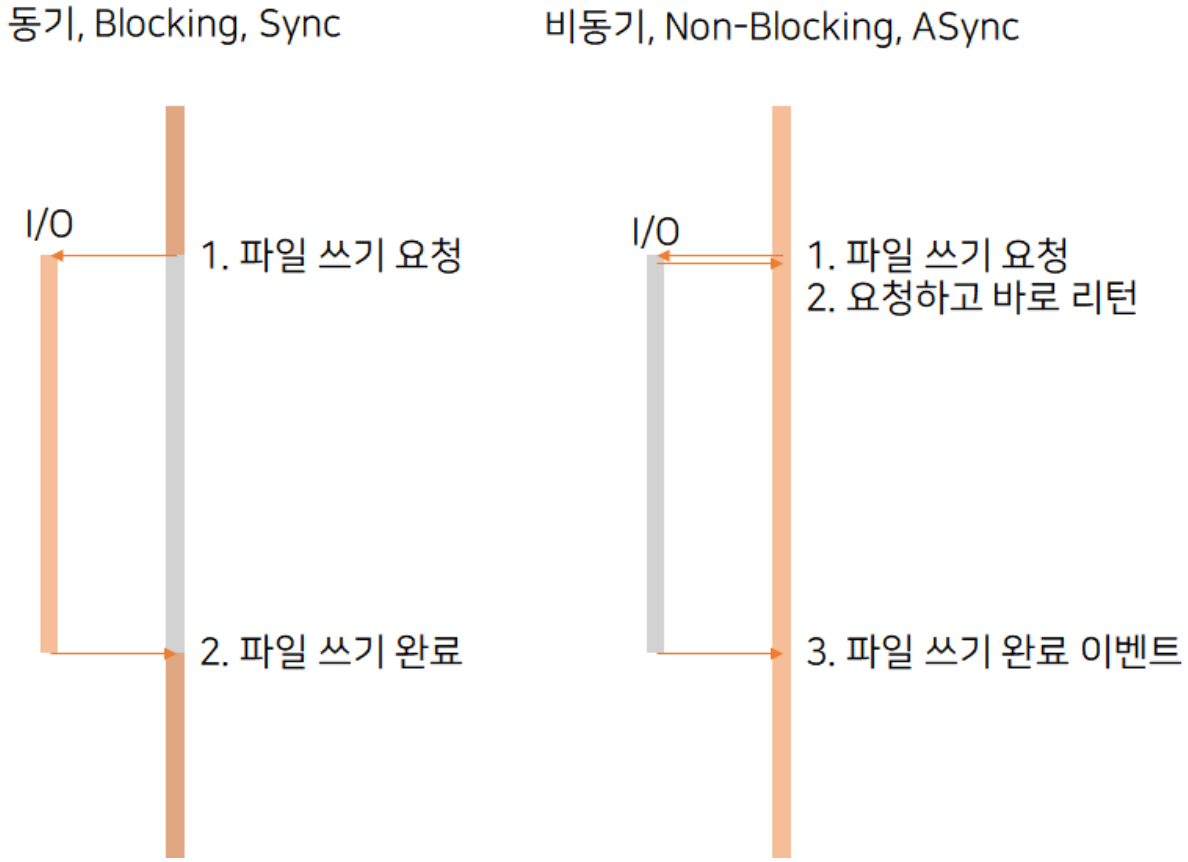
JavaScript 런타임 환경

NodeJs기반의 웹 어플리케이션 프레임워크

NodeJS의 특징

1. Non-Blocking I/O
2. Single Thread
3. 이벤트 기반

NodeJS의 특징: Non-Blocking I/O



동기(Blocking, Sync)

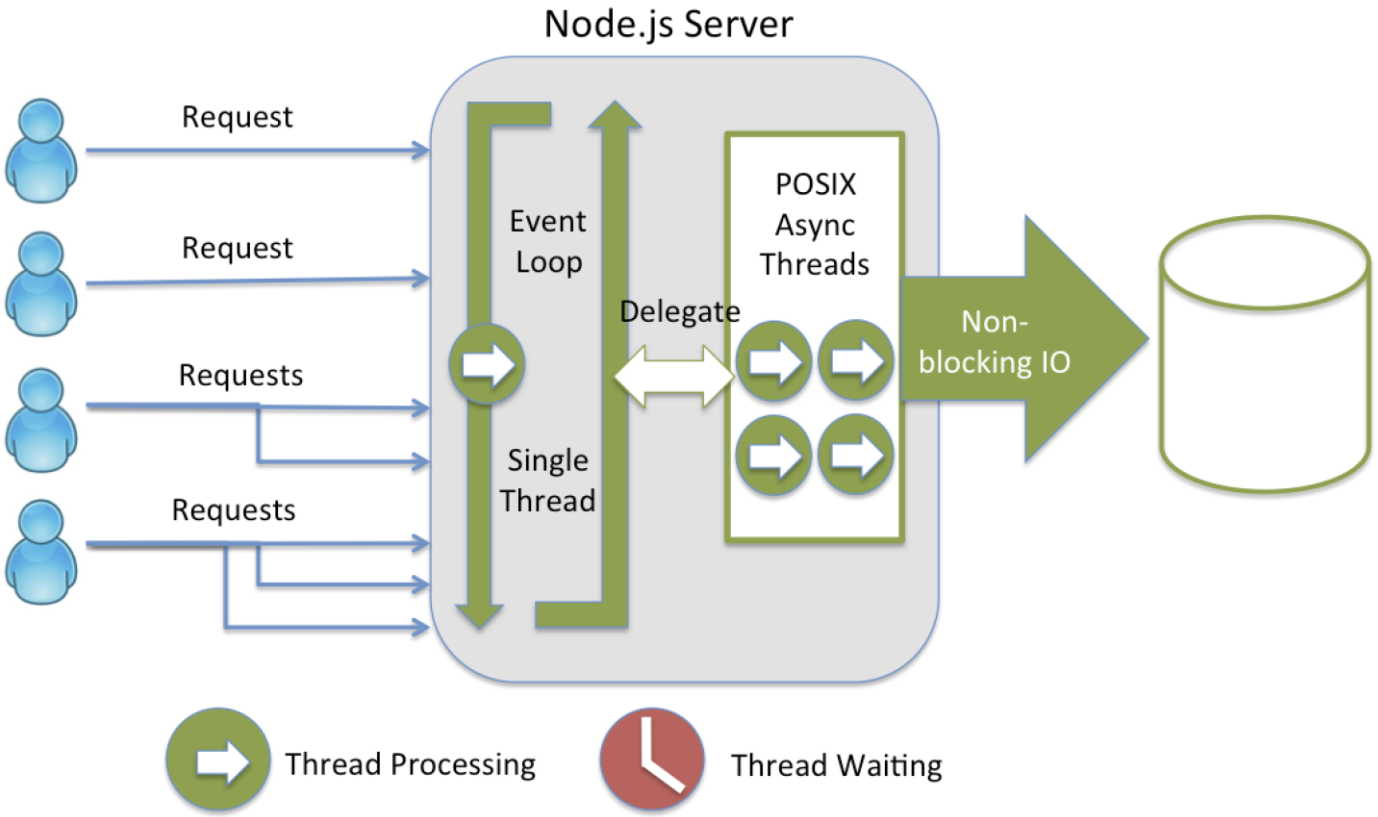
요청을 하고 완료를 할 때 까지 기다리는 방식

비동기(Non-Blocking, Async)

방식은 요청을 하고 바로 제어권을 돌려 받는 방식
즉 요청만 하고 다시 프로그램을 처리하다가
완료 이벤트가 발생하면 미리 지정한 처리를 진행한다.

Nodejs는 **비동기(Non-Blocking, Async)** 으로 동작한다.

NodeJS의 특징: Single Thread



Node.js는 하나의 Thread로만 동작한다.

NodeJS의 특징: Event-driven

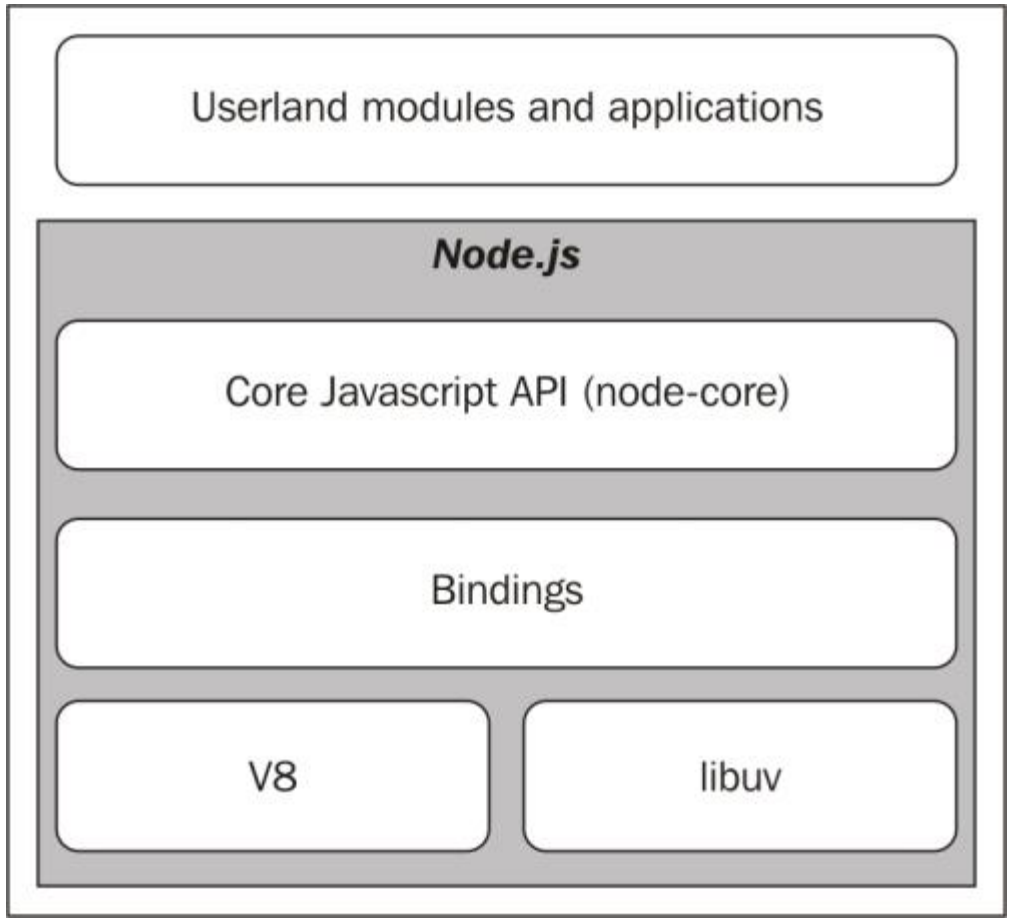
이벤트 기반(Event-Driven)

이벤트 기반은 이벤트가 발생 할 때 미리 지정해 놓은 작업을 수행하는 방식
노드는 이벤트에 Callback 함수를 지정해서 동작

동시성 vs 병렬성

동시성은 흐름을 실행시키는 것은 **하나**이지만 time-slicing, time-quantum등으로 작은 단위로 나누어서 흐름을 돌아가면서 동시에 일어나는 것처럼 만들어 주는 방식이며 ***병렬성***은 실제 흐름을 실행시키는 것이 **복수 개** 인 것을 의미합니다.

NodeJS 구조



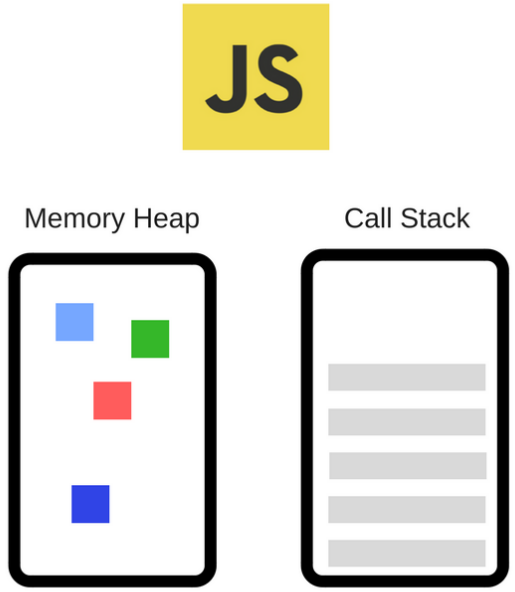
V8
구글에서 Chrome 브라우저 용으로 개발한 자바스크립트 엔진
V8은 혁신적인 설계와 속도, 그리고 효율적인 메모리 관리로 높은 평가

Libuv
C언어로 만들어져서 낮은 수준의 기능들을 javascript에 매핑하고 사용하도록 해주는 바인딩 세트

Core Javascript: Nodejs API을 구현

[참고용] JavScript 심화

Javascript의 대부분은 Single Thread를 쓰며 Call back Queue를 사용

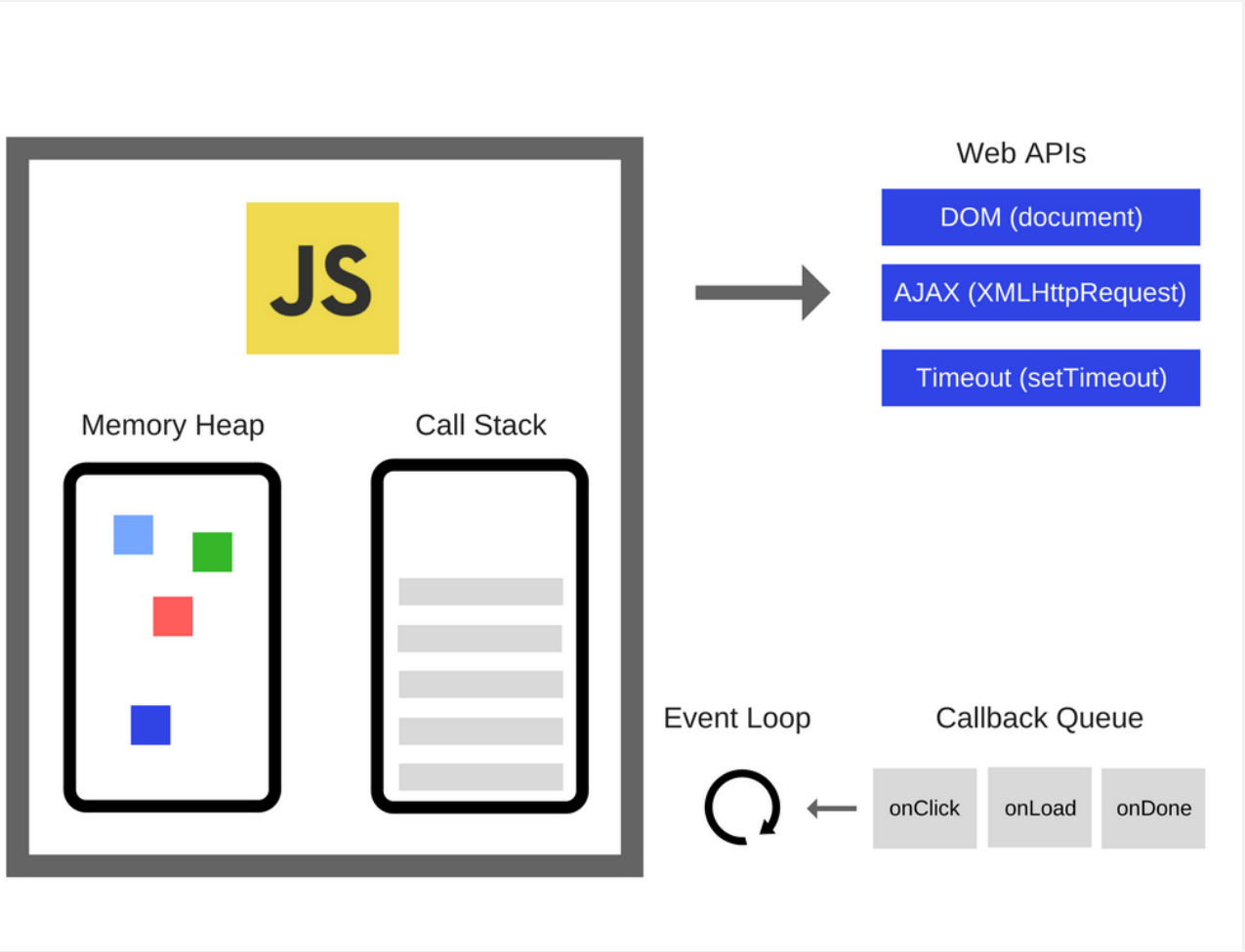


JavaScript는 크게 Memory Heap과 Call Stack으로 구성

Memory Heap:
메모리 할당되는 곳

Call Stack:
호출 스택이 쌓이는 곳

[참고용] JavScript 심화



Web Browser, node등의 다양한 런타임이 존재

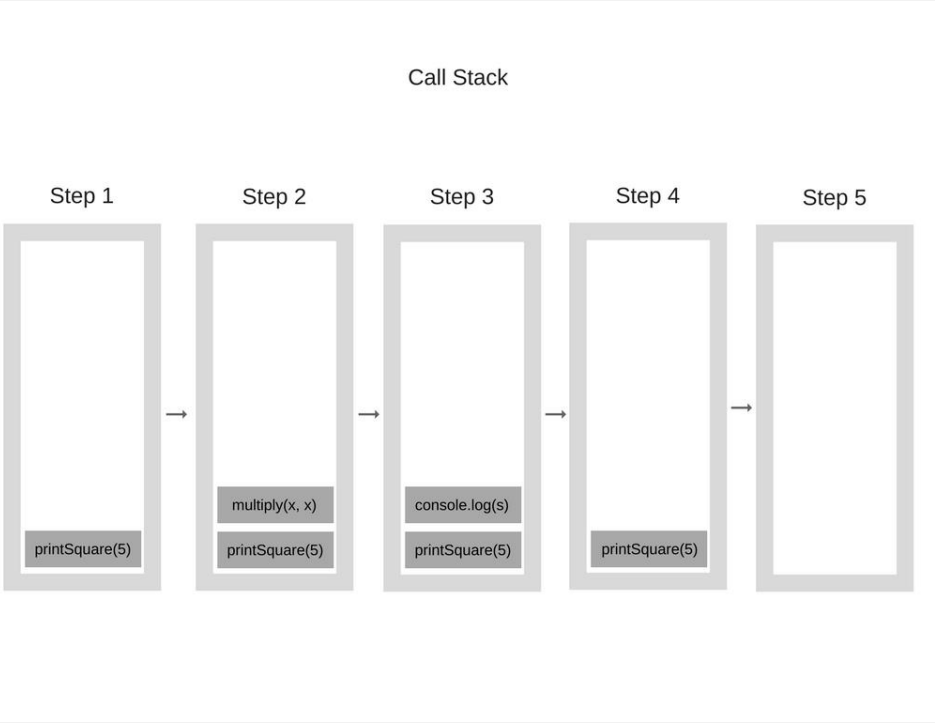
자바스크립트 엔진 외에도 자바스크립트에 관여하는 다양한 요소들이 존재

- Web API
DOM, AJAX, Timeout과 같이 브라우저에서 제공하는 API
- event loop
- Callback Queue

[참고용] JavScript 심화

Callback Stack: Call Stack을 이용해서 현재 프로그램의 위치를 추적

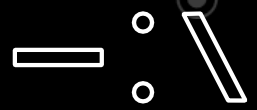
```
function multiply(x, y) {
    return x * y;
}
function printSquare(x) {
    var s = multiply(x, x);
    console.log(s);
}
printSquare(5);
```



[참고용] JavScript 심화

더 자세히 공부하기

출처: https://blog.sessionstack.com/how-does-javascript-actually-work-part-1-b0bacc073cf
한글 번역: https://joshua1988.github.io/web-development/translation/javascript/how-js-works-inside-engine/



02

동기 / 비동기

비동기 실습

```
function task1(){
    setTimeout(function(){
        console.log('task1');
    }, 0);
}
function task2(){
    console.log('task2');
}
function task3(){
    console.log('task3');
}
task1();
task2();
task3();
```

소스코드 링크:
https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-async.js

Quiz

Task1: 동기 vs 비동기
Task2: 동기 vs 비동기
Task3: 동기 vs 비동기

Fs 모듈: 동기 파일 쓰기

실습: practice-sync-file-write.js

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

const numArr = [1, 2, 3, 4, 5];

const fileCommonName = 'syncText';

numArr.forEach((num) => {
    const fileName = fileCommonName+num;
    const data = `reserved message for the '${fileName}'`;
    fs.writeFileSync(`${fileName}.txt`, data);
    console.log(`file[${fileName}] write complete`);
})
```

소스코드 링크:
https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-sync-file-write.js

출력 결과

```
file[syncText1] write complete
file[syncText2] write complete
file[syncText3] write complete
file[syncText4] write complete
file[syncText5] write complete
```

Fs 모듈: 비동기 파일 쓰기

실습: practice-async-file-write.js

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

const numArr = [1, 2, 3, 4, 5];
const fileCommonName = 'asyncText';

numArr.forEach((num) => {
    const fileName = fileCommonName+num;
    const data = `reserved message for the '${fileName}'`;
    fs.writeFile(`${fileName}.txt`, data, ()=>{
        console.log(`file[${fileName}] write complete`);
    });
});
```

소스코드 링크:
https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-async-file-write.js

출력 결과

```
file[asyncText3] write complete
file[asyncText4] write complete
file[asyncText2] write complete
file[asyncText1] write complete
file[asyncText5] write complete
```

Fs 모듈: 동기 파일 읽기

실습: practice-sync-file-read.js

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

const numArr = [1, 2, 3, 4, 5];
const fileCommonName = 'syncText';

numArr.forEach((num) => {
    const fileName = fileCommonName+num;
    const data = fs.readFileSync(`${fileName}.txt`);
    console.log(`file[${fileName}] with ${data}`);
})
```

소스코드 링크:
https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-sync-file-read.js

출력 결과

```
file[syncText1] with reserved message for the 'syncText1'
file[syncText2] with reserved message for the 'syncText2'
file[syncText3] with reserved message for the 'syncText3'
file[syncText4] with reserved message for the 'syncText4'
file[syncText5] with reserved message for the 'syncText5'
```

Fs 모듈: 비동기 파일 읽기

실습: practice-async-file-read.js

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

const numArr = [1, 2, 3, 4, 5];
const fileCommonName = 'asyncText';

numArr.forEach((num) => {
    const fileName = fileCommonName+num;
    fs.readFile(`${fileName}.txt`, (err, data) => {
        console.log(`file[${fileName}] with ${data}`);
    });
});
```

소스코드 링크:
https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-async-file-read.js

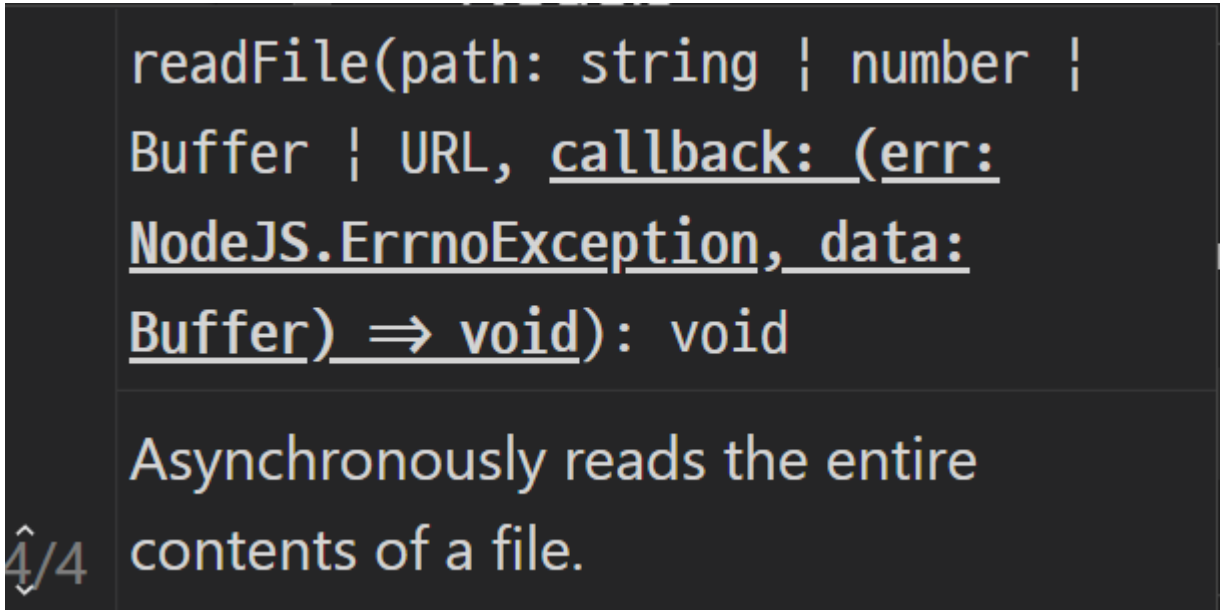
출력 결과

```
file[syncText1] with reserved message for the 'syncText1'
file[syncText2] with reserved message for the 'syncText2'
file[syncText3] with reserved message for the 'syncText3'
file[syncText4] with reserved message for the 'syncText4'
file[syncText5] with reserved message for the 'syncText5'
```

실습: practice-async-file-read.js

Fs 모듈: 비동기 파일 읽기

```
fs.readFile(`${fileName}.txt`, (err, data) => {
        console.log(`file[${fileName}] with ${data}`);
    });
```



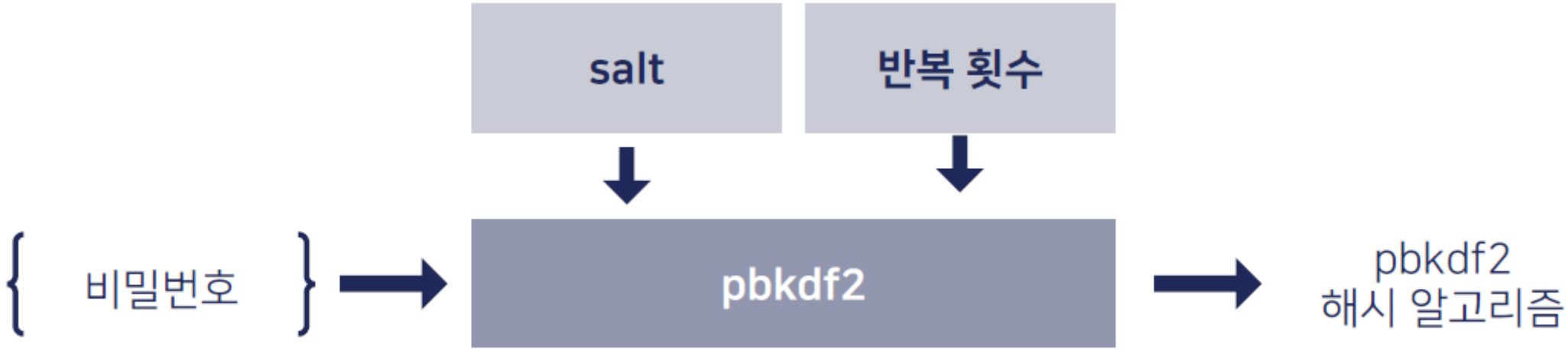
여기서 readFile Method의 2번째 파라미터를 보면
(err, data) => {} 으로 function으로 지정되어 있다.

이는 callback함수를 지정한 것이며 오른쪽을 보면
Callback함수의 첫번째 인자는 error 정보,
두번째 인자는 data인 것을 알 수 있다.

즉, readFile이 수행 완료되었다면 미리 지정해 놓은 callback 함수가 호출이 되며
이때 error정보가 있으면 err에, 성공했으면 읽어온 값을 data로 지정된다.

Crypto pbkdf2

PBKDF2(Password-Based Key Derivation Function)
가장 많이 사용되는 key derivation function
해시 함수의 컨테이너인 PBKDF2는 솔트를 적용한 후 해시 함수의 반복 횟수를 임의로 선택
PBKDF2는 아주 가볍고 구현하기 쉬우며, SHA와 같이 검증된 해시 함수만을 사용한다.
※ bcrypt나 **scrypt**가 보안이 더 뛰어남.



Crypto pbkdf2

Hash:
해시 알고리즘은 문자열을 특정 규칙을 이용해 다른 문자열로 치환하는 방식
해시 알고리즘으로는 sha256, **sha512** 등

Salt:
해시 알고리즘으로 암호화 하기 전에 우선 평문 암호에 salt라고 불리는 임의의 문자열을 붙인 후 암호화
이렇게 하여 원본 암호를 더 찾기 어렵게 만든다.

Key stretching:
salt와 hash만으로도 부족하기에 실사용에서는 키 스트레칭이라는 기법을 함께 사용함
이는 평문 암호에 salt를 치고 해시 암호화를 하는 작업을 **수만~수십만번 반복해서 암호화 하는 방법**

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-module-pbkdf2.js

Pbkdf2 example

DIGEST = PBKDF2(PRF, Password, Salt, c, DLen)

PRF: 난수(예: HMAC)
Password: 패스워드
Salt: 암호학 솔트
c: 원하는 iteration 반복 수
DLen: 원하는 다이제스트 길이

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

const password = 'password';
crypto.randomBytes(32, (err, salt) => {
    if(err) throw err;
    crypto.pbkdf2(password, salt, 1, 32, 'sha512', (err, derivedKey) => {
        if(err) throw err;
        fs.writeFile('password.txt', derivedKey.toString('hex'), (err) => {
            if(err) throw err;
            console.log('complete write password');
        })
    })
})
```

Callback hell

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

const password = 'password';
crypto.randomBytes(32, (err, salt) => {
    if(err) throw err;
    crypto.pbkdf2(password, salt, 1, 32, 'sha512', (err, derivedKey) => {
        if(err) throw err;
        fs.writeFile('password.txt', derivedKey.toString('hex'), (err) => {
            if(err) throw err;
            console.log('complete write password');
        })
    })
})
```

callback함수 안에
  callback함수 안에
    callback 함수안에
      결과를 출력하는 로직

이처럼 여러 콜백함수가 중첩되어 코드를 읽기 어려워지는 것을 **콜백헬**이라고 합니다.

비동기 처리에는 다양한 장점이 있지만 이 Callback hell이 발생하는 문제점이 있습니다.

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-module-pbkdf2-sync.js

Callback hell: 해결 시도, Sync

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

const password = 'password';
const salt = crypto.randomBytes(32);
const derivedKey = crypto.pbkdf2Sync(password, salt, 1, 32, 'sha512');
fs.writeFileSync('password.txt', derivedKey.toString('hex'));
console.log('complete write password');
```

Crypto 모듈은 fs와 마찬가지로 Async 모드와 Sync 모드 모두 지원합니다.
따라서 Sync 함수를 이용해서 위와 같이 콜백헬을 제거할 수 있습니다.
하지만 이는 비동기가 아닌 **동기 프로그램**이므로 때문에 해결 책이 될 수 없습니다.

Callback hell: 해결 방법

Callback Hell을 최소화 하는 3가지 방법

1. Keep your code shallow
2. Modularize
3. Handle every single error

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-module-pbkdf2-fix1.js

Callback hell: 1. Keep your code shallow

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

const password = 'password';
crypto.randomBytes(32, (err, salt) => {
   if(err) throw err;
   crypto.pbkdf2(password, salt, 1, 32, 'sha512', (err, derivedKey) => {
      if(err) throw err;
      fs.writeFile('password.txt', derivedKey.toString('hex'), (err) => {
         if(err) throw err;
         console.log('complete write password');
      })
   })
})
```

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

const password = 'password1234';
crypto.randomBytes(32, madeSaltFunc);
function madeSaltFunc(err, salt) {
   if(err) throw err;
   crypto.pbkdf2(password, salt, 1, 32, 'sha512', madeKeyFunc);
}
function madeKeyFunc(err, derivedKey) {
   if(err) throw err;
   fs.writeFile('password.txt', derivedKey.toString('hex'), wroteFileFunc);
}
function wroteFileFunc(err) {
   if(err) throw err;
   console.log('complete write password');
}
```

콜백함수를 명시적으로 정의하여서 연결해주면 Callback hell을 최소화 할 수 있습니다.

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-module-pbkdf2-fix2.js

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/encryption.js

Callback hell: 2. Modularize

작은 모듈을 만들고 이를 조립하여 큰 모듈을 만들어서 콜백 헬을 감소시키는 방법

실습 코드는 암호화 하는 로직과 파일을 쓰는 로직 두가지 책임 존재
암호 로직과 파일 로직을 분리할 수 있다.
여기서는 암호 로직을 encryption.js로 분리.

```
const crypto = require('crypto');
const pbkdf2 = require('pbkdf2');
function encryptPBKDF2(password, next){
    crypto.randomBytes(32, madeSaltFunc);
    function madeSaltFunc(err, salt) {
        if(err) throw err;
        pbkdf2.pbkdf2(password, salt, 1, 32, 'sha512', madeKeyFunc);
    }
    function madeKeyFunc(err, derivedKey) {
        if(err) throw err;
        next(err, derivedKey.toString('hex'));
    }
}
module.exports = encryptPBKDF2;
```

encryption.js

```
const fs = require('fs');
const encryption = require('./encryption');

const password = 'password1234';
encryption(password, (error, derivedKey) => {
    fs.writeFile('password2.txt', derivedKey, wroteFileFunc);
    function wroteFileFunc(err) {
        if(err) throw err;
        console.log('complete write password');
    }
})
```

Practice-module-pbkdf2-fix2.js

Callback hell: 3. Handle every single error

- 1번(Keep your code shallow),2번 (modularize)은 readable에 관련된 rule이였다면
- 3번(Handle every single error)은 stable에 관련된 규칙입니다.

비동기 패턴을 사용하면 중간에 에러가 발생하는 경우 특별한 처리를 하지 않는다면
그 에러에 대해서 절 때 알 수 없는 경우가 발생합니다.

따라서 가장 유명한 방법은 **콜백 함수의 첫 번째 인자를 error 관련 값으로 지정**하는 것입니다.
지금까지 실습 코드를 확인해보면 콜백함수에 (err, ...) => {}으로 시작하는 것을 확인 할 수 있습니다.
이렇게 하는 이유는 만약 err가 첫번째 인자가 아니라면 다음과 같은 문제가 발생합니다.

```
function handleFile (file) {
    // ...
}
```

이와 같이 코드를 작성하면 에러에 대한 처리를 생략하고 코드를 작성하게 됩니다.
따라서 명시적으로 지정하여서 프로그래머가 error를 처리하도록 만들어야 합니다.

Callback hell: 3. Handle every single error

※ Javascript Standard Style에도 이와 같은 에러 관련 규칙이 명시 되어있습니다.

eslint: handle-callback-err

- **Always handle the** `err` function parameter.

  eslint: handle-callback-err

```
// ✓ ok
run(function (err) {
  if (err) throw err
  window.alert('done')
})
```

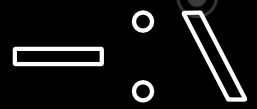
```
// X avoid
run(function (err) {
  window.alert('done')
})
```

Callback hell: 3. Handle every single error

참고 블로그

영어: http://callbackhell.com/

한글 번역: https://medium.com/@pitzcarraldo/callback-hell-%EA%B3%BC-promise-pattern-471976ffd139



03

Promise

Promise

➢ Javascript에서는 Callback Hell을 해결하기 위한 방법으로 Promise 라는 객체가 존재합니다.

프로미스
비동기 처리에 사용되는 객체
프로미스 객체가 생성되는 순간에는 알 수 없는 값을 처리하기 위해서 만드는 대리자
값을 바로 반환하는 대신에 Promise 객체를 반환해서 **비동기 메소드를 동기 메소드처럼** 만들어 줌

Promise에는 3가지 상태가 존재합니다.
pending: 최초 생성된 시점의 상태
fulfilled: 작업이 성공적으로 완료 된 상태
rejected: 작업이 실패한 상태

Promise객체는 아래와 같이 만들 수 있습니다.

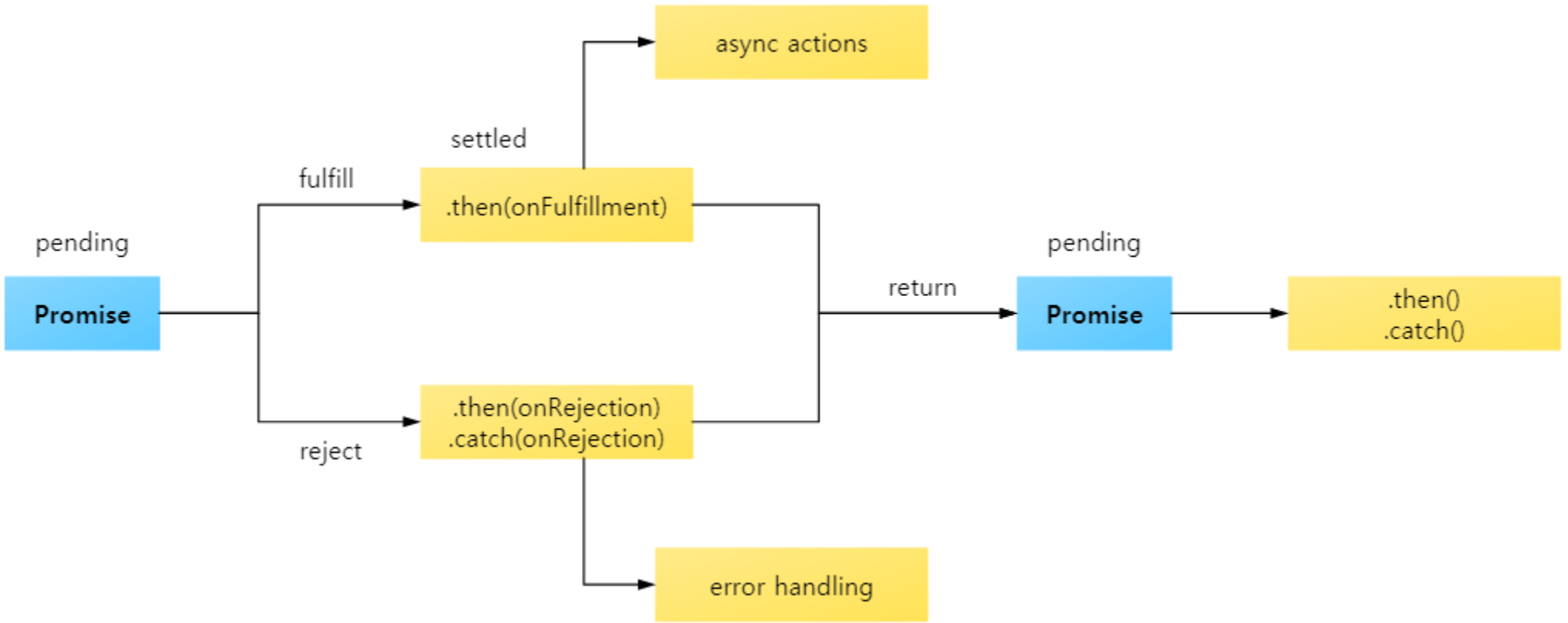
```
function readFile(filename, enc){
  return new Promise(function (fulfill, reject){ ... });
}
```

Promise Syntax

```
new Promise(executor);
```

executor:
resolve와 reject를 파라미터로 갖는 함수
resolve는 작업이 성공했을 때
reject는 실패했을 때 실행되는 callback함수입니다.

Promise의 흐름



resolve 함수를 호출하면 fulfilled 상태과 되며 reject 함수를 호출하면 rejected 상태가 됩니다. fulfilled 상태는 then을 통해서 전달되며 rejected는 catch를 통해서 전달 됩니다.

Promise의 흐름

fulfilled case

```
function getData() {
  return new Promise(function (resolve, reject) {
    var data = 25;
    resolve(data);
  });
}
getData().then(function (resolvedData) {
  console.log(resolvedData);
});
```

⇒ 25

rejected case

```
function getData() {
  return new Promise(function (resolve, reject) {
    reject(new Error("Request is failed"));
  });
}

getData().then().catch(function (err) {
  console.log(err);
});
```

⇒ Error: Request is failed

Promise Chaining

Promise Chaining은 여러 개의 프로미스를 연결하여 사용하는 것을 의미

Js Copy

```
function getData() {
  return new Promise({
    // ...
  });
}

// then() 으로 여러 개의 프로미스를 연결한 형식
getData()
  .then(function (data) {
    // ...
  })
  .then(function () {
    // ...
  })
  .then(function () {
    // ...
  });
```

Promise 참고 자료

참고 블로그

영어: https://joshua1988.github.io/web-development/javascript/promise-for-beginners/

Promise Chaining

실습 1

1.score배열 생성
2.배열의 합 구하기
3.점수를 등급으로 변환하기
이 3가지 작업을 promise chaining으로 구현

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-promise-1.js

Promise Chaining

1.score배열 생성

```
function getScoreArray(size) {
    return new Promise(function (resolve, reject) {
        if(size <= 0){
            reject(new Error("size must be positive"));
            return;
        }
        const arr = [...Array(size)].map(idx =>
parseInt(Math.random()*11));
        console.log(`array is ${arr}`);
        resolve(arr);
    });
}
```

첫번째 함수(getScoreArray)는 size개만큼 임의의 숫자를 생성하는 함수입니다.
만약 size가 0과 같거나 작으면 reject로 에러는 전달합니다.

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-promise-1.js

Promise Chaining

2. 배열의 합 구하기

```
function getSum(arr) {
    return new Promise(function (resolve, reject) {
        const sum = arr.reduce((prev, current) =>  prev + current);
        if(sum <= 0){
            reject(new Error("sum must be larger than 0"));
            return;
        }
        console.log(`sum: ${sum}`);
        resolve(sum);
    });
}
```

두번째 함수(getSum)는 배열을 인자로 받아서 요소의 합을 구합니다. 이때 reduce 함수를 이용했습니다.
이때 합의 결과가 0보다 작은 경우는 에러를 발생시킵니다.
이때 주의해야 할 점이 있는데, reject함수를 호출한 이후에 return을 하지 않는다면
reject와 resolve 모두 호출되어 원치 않는 결과를 얻을 수 있습니다.

Promise Chaining

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-promise-1.js

3. 점수를 등급으로 변환하기

3번째 함수는 getGrade입니다.
점수를 입력 받으면 등급을 반환해주는 함수입니다.
이때 50점 미만인 경우에는
에러를 발생시키도록 구현했습니다.

```
function getGrade(result) {
    return new Promise(function (resolve, reject) {
        let grade;
        switch(parseInt(result / 10)){
            case 9:
            case 8:
                grade = 'A'; break;
            case 7:
                grade = 'B'; break;
            case 6:
                grade = 'C'; break;
            case 5:
                grade = 'D'; break;
            default:
                reject(new Error(`too low score(${result})`));
                return;
        }
        resolve(grade);
    });
}
```

Promise Chaining

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-promise-1.js

4. 함수 Chaining 하기

```
getScoreArray(10)
.then(getSum)
.then(getGrade)
.then((result) => console.log(`grade is ${result}`))
.catch(err => {
    console.log(`Error: ${err}`);
})
```

각각의 Promise가 then을 통해서 결과가 전달되며
중간에 에러가 발생하면 이후의 then으로 전달되지 않으며
catch으로 이동하게 됩니다.

Promise Chaining

실습 2

json2csv모듈과 csvtojson 모듈을 이용해서
Promise 패턴으로 csvManager를 구현해보겠습니다.

Promise Chaining

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-json2csv.js

json2csv

json2csv 모듈은 JSON 형태의 Javascript 객체를 CSV형태의 String으로 변환해줍니다. json2csv

Csv란? Excel과 DB와 호환되는 텍스트 형식입니다.
첫번째 줄에는 필드 이름이 들어가며 2번째 줄부터 각 ROW에 해당하는 값이 들어 있습니다.

Practice-json2csv.js

```
const json2csv = require('json2csv');

const jsonArray = [{id:'admin', pw: 'admin', name:'관리자'},
                   {id:'heesung', pw: '1q2w3e4r!', name:'윤희성'},
                   ,{id:'starbucks', pw: 'JamongBlackHoneyTea', name:'스타벅스'}];

const resultCsv = json2csv.parse(jsonArray)
console.log(resultCsv)
```

출력값

```
"id","pw","name"
"admin","admin","관리자"
"heesung","1q2w3e4r!","윤희성"
"starbucks","JamongBlackHoneyTea","스타벅스"
```

https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-SEMINAR/blob/develop/seminar2/sourcecode/practice-csvtojson.js

Promise Chaining

Csv2json

csvtojson 모듈은 csv 포맷에서 JSON으로 가져오는 모듈입니다. csvtojson

csv2json.csv

```
"id","pw","name"
"admin","admin","관리자"
"heesung","1q2w3e4r!","윤희성"
"starbucks","JamongBlackHoneyTea","스타벅스"
```

Practice-csvtojson.js

```
const csv = require('csvtojson');

csv().fromFile('./csvtojson.csv').then((jsonArr) => {
    if (!jsonArr) {
        console.log(`file read err: ${err}`);
        return;
    }
    console.log(jsonArr);
}, (err) => {
    console.log(`err with readCSV: ${err}`);
})
```

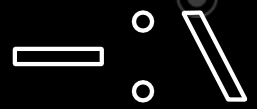
출력값

```
[ { id: 'admin', pw: 'admin', name: '관리자' },
  { id: 'heesung', pw: '1q2w3e4r!', name: '윤희성' },
  { id: 'starbucks', pw: 'JamongBlackHoneyTea', name: '스타벅스' } ]
```

실습 2

1. json data를 csv파일로 저장
2. csv파일을 읽어서 json data로 반환

csvManager1.js csvManager1Test.js csvManager2.js csvManager2Test.js



04

Async Await

async/await

practice-async-await.js

ES6 이후에 나온 자바스크립트 비동기 패턴입니다.
기존의 비동기 처리 방식인 콜백함수와 Promise의 단점을 보완
하여 읽기 좋은 코드로 만들어 줍니다.

다음과 같이 비동기 함수가 있을 때 아래와 같이 적용 될 수 있습니다.

```
function fetchItems() {
    return new Promise(function (resolve, reject) {
        var items = [1, 2, 3];
        resolve(items);
    });
}
```

일반 Promise 버전

```
function promiseVer() {
    fetchItems().then(resultItems => {
        console.log(resultItems); // [1,2,3]
    });
}
```

Async/Await 버전

```
async function asyncVer() {
    const resultItems = await fetchItems();
    console.log(resultItems); // [1,2,3]
}
```

async/await

practice-async-await.js

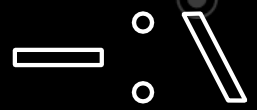
function 앞에 async 를 붙이고 처리할 비동기 메소드 앞에 await만 붙이면 된다.

```
async function 함수명() {
  await 비동기_처리_메서드_명();
}
```

Arrow function의 경우 async/await

```
const asyncVer2 = async () => {
    const resultItems = await fetchItems();
    console.log(resultItems); // [1,2,3]
}
```

⇒ Promise에 비해 **문법이 간단하고 가독성**이 좋기 때문에 많이 사용하는 방법이다.



05

과제안내

과제 안내

1. Node JS에 대해서 알아보자

NodeJS를 공부하고 정리 문서를 만드는 것이 첫 번째 과제입니다.
정리 문서에는 아래와 같은 내용을 넣으시면 됩니다.
- NodeJS란
- NodeJS의 특징
- NodeJS의 장단점
- 그 이외 NodeJS관련된 내용이면 뭐든 괜찮습니다!

또한 참고한 블로그가 있다면 링크로 출처를 표시해주세요!

SHARED-LEARNNG 프로젝트에 NodeJS라는 폴더를 만들 예정입니다.
폴더 내에 [조 이름]으로 파일을 만들어서 한명이 push + pull Request를 날려주세요!
※ 이때 대제목을 미리 작성해두면 충돌이 일어나는 것을 막을 수 있습니다!(자세한건 갠톡으로)
이후 Merge되면 나머지 팀원들도 pull을 받은 이후에 해당 파일에 내용을 추가하면 됩니다.

참고링크: https://asfirstalways.tistory.com/43
https://epdl-studio.tistory.com/76
https://geonlee.tistory.com/92

과제 안내

2. 과제 https://github.com/WITH-SOPT-SERVER/SOPT-SERVER-ASSIGNMENT

세미나에서 앉게 되는 조를 등록/조회/변경할 수 있는 서버를 만드는 것이 과제입니다.
난이도별로 도전 과제가 주어집니다. Level 1까지만 구현해도 과제로 인정됩니다!

Level 1

•member.csv파일에 사용자 정보(이름, 그룹 번호)가 저장 되어있다.
•[/api/group]: 전체 그룹 구성을 조회한다.
•[/api/group/:groupIdx]: 그룹 구성원을 조회한다.
※ QueryParameter 방식 localhost:3000/api/group/:groupIdx 으로 라우팅을 설정하고 localhost:3000/api/group/4이 호출된다면
req.params.groupId에 4라는 값이 들어오게 됩니다!

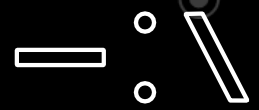
Level 2

•Group.csv에 Group 정보(그룹 번호, 그룹 이름)가 저장 되어있다.
•조회할 때 groupIdx대신에 그룹 이름으로 보여준다.

Level 3

•구성원들의 조원을 섞어주는 모듈(groupMixer)을 만든다.[방식 자유]

SHOUT OUR PASSION TOGETHER



PASSION

Thank You :)

EXPERIENCE

GROWTH

CHALLENGE

SHOUT OUR PASSION TOGETHER

SOPT