21.9.02 3-5-5 crypto ~ 3-6-2

3.5.5 crypto

crypto는 다양한 방식의 암호화를 도와주는 모듈이다.

웹 서비스를 구현하면 비밀번호 등 민감한 정보를 안전하게 보호해야 하기 때문에 중요하다.

<단방향 암호화>

비밀번호는 보통 단방향 암호화 알고리즘을 사용해 암호화한다.

> 💡 **단방향 암호화 : 복호화할 수 없는 암호화. 즉, 원래 문자열을 찾을 수 없음.**

단방향 암호화 알고리즘은 주로 해시 기법을 사용한다.

> 💡 **해시 기법 : 어떠한 문자열을 고정된 길이의 다른 문자열을 바꿔버리는 방식**

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

console.log('base64:', crypto.createHash('sha512').update('비밀번호').digest('base64'));
console.log('hex: ',crypto.createHash('sha512').update('비밀번호').digest('hex'));
console.log('base64: ', crypto.createHash('sha512').update('다른 비밀번호').digest('base64'));
```

- **createHash(알고리즘)** : 사용할 해시 알고리즘을 넣어준다. md5, sha1, sha256, sha512 등이 가능하지만 md5와 sha1은 이미 취약점이 발견됐다.
- **update(문자열)** : 변환할 문자열을 넣어준다.
- **digest(인코딩)** : 인코딩할 알고리즘을 넣어준다. base64, hex, latin1이 주로 사용되는데 base64가 결과 문자열이 짧아 애용된다.

```
PS C:\Users\ggg05\OneDrive\바탕 화면\studyNodeJS\21-9-2> node hash
base64: dvfV6nyLRRt3NxKSlTHOkkEGgqW2HRtfu19Ou/psUXvwlebbXCboxIPmDYOFRIpqav2eUTBFuHaZri5x+usy1g==
hex:  76f7d5ea7c8b451b773712929531ce92410682a5b61d1b5fbb5f4ebbfa6c517bf095e6db5c26e8c483e60d8385448a
6a6afd9e513045b87699ae2e71faeb32d6
base64:  cx49cjC8ctKtMzwJGBY853itZeb6qxzXGvuUJkbWTGn5VXAFbAwXGEOxU2Qksoj+aM2GWPhc107mmkyohXMsQw==
```

현재는 주로 pdkdf2, bcypt, scrypt라는 알고리즘으로 비밀번호를 암호화한다. 이중에 pbkdf2는 기존 문자열에 salt라는 문자열을 붙인 후 해시 알고리즘을 반족해서 적용한다.

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

crypto.randomBytes(64, (err, buf)=>{
    const salt = buf.toString('base64');
    console.log('salt:', salt);

    //.pbkdf2(비밀번호, salt, 반복 횟수, 출력 바이트, 해시 알고리즘)
    crypto.pbkdf2('비밀번호', salt, 100000,64, 'sha512', (err,key)=>{
        console.log('password: ', key.toString('base64'));
    });
});
```

pbkdf2는 간단하지만 bcrypt나 scrypt보다 취약하다.

```
PS C:\Users\ggg05\OneDrive\바탕 화면\studyNodeJS\21-9-2> node pbkdf2
salt: wGAcduUBhm/4zDGf7J1+GOfQPNBLoYTnxM5KP87X7sPBpVo3ugWrU3wXKR6uwFgvZzDYNdSGBcGvtW2FXCxjNg==
password:  wCzvhu7JicSkKuEHqerS6ymLM53c943bKzt9o+fUkBsfStFosUK2FQEWA/ppVag3PXLnhATuGu7oZVhQZ03N1Q==
```

<양방향 암호화>

단방향 암호화와 달리 양방향 암호화는 복호화가 가능하다.

이때 **키**라는것이 사용되고, 복호화하려면 암호화할 때 사용한 키와 같은 키를 사용해야 한다.

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

const iv = new Buffer.alloc(16,'encodingkey','utf8');
//crypto.createCipher(알고리즘,키) : 사용 가능한 알고리즘 목록은 getCipher()
const key = crypto.scryptSync('열쇠','salt', 32);
const cipher = crypto.createCipheriv('aes-256-cbc', key,iv);

//cipher.update(문자열, 인코딩, 출력 인코딩) : 암호화할 대상과 대상의 인코딩,
//출력 결과물의 인코딩을 넣는다 보통 utf8인코딩, base64함호를 사용한다.
let result = cipher.update('암호화할 문장','utf8','base64');

//cipher.final(출력 인코딩) : 출력 결과물의 인코딩을 넣어주면 암호화가 완료.
result +=cipher.final('base64');
//console.log(cipher.final('base64'));

console.log('암호화:',result);

//crypto.ceateDecipheriv(알고리즘,키, iv) : 복호화할 때 사용하고
```

```
// 암호화할 때 사용했던 알고리즘과 키를 그대로 넣어야한다.
const decipher = crypto.createDecipheriv('aes-256-cbc', key, iv);

/*decipher.update(문자열, 인코딩, 출력 인코딩) : 암호화 된 문장
 그 문장의 인코딩, 복호화할 인코딩을 넣어준다. createCipher에 update에서
 utf8, base64 순서로 넣었다면 createDecipher에는 반대로 넣어준다.
*/
let result2 = decipher.update(result, 'base64', 'utf8');

//decipher.final(출력 인코딩) : 복호화 결과물의 인코딩을 넣어준다.
result2 += decipher.final('utf8');
console.log('복호화: ', result2);
```

이거 하느라 정말 머리 깨지는줄 알았다....

책에는 createCipher가 나왔지만 현재 노드 버전에서는 deprecated돼서

createCipheriv로 대체했어야 했다. 따라서 iv까지 추가해야한다.

iv는 숨길 필요도 없이 고정된 값이기만 하면 된다고 했다.

그리고 iv는 길이가 16, key는 길이가 32로 맞춰야 제대로 작동 된다는 것을 알았다.

😩😩😩😩😩😩😩😩😩😩😩😩😩😩😩😩😩

```
PS C:\Users\ggg05\OneDrive\바탕 화면\studyNodeJS\21-9-2> node cipher
암호화: TTxlJr2CENNnq70gVps158TQs+MguuvSKSh8aYWweqk=
복호화: 암호화할 문장
```

3.5.6 util

util은 각종 편의 기능을 모아둔 모듈이다.

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

//함수가 deprecated처리 되었음을 알려준다.
//첫 번째 인자 함수를 사용했을 때 두 번째 인자인 경고 메시지가 표시된다.
const dontUseMe = util.deprecate((x,y)=>{
    console.log(x+y);
},'dontUseMe 함수는 deprecated되었으니 더 이상 사용하지 마세요!');
dontUseMe(1,2);

//콜백 패턴을 프로미스 패턴으로 바꿔준다. 바꿀 함수를 인자로 사용하면 된다.
const randomBytesPromise  = util.promisify(crypto.randomBytes);
```

```
randomBytesPromise(64)
    .then((buf) =>{
        console.log(buf.toString('base64'));
    })
    .catch((error)=>{
        console.error(error);
    });
```

```
PS C:\Users\ggg05\OneDrive\바탕 화면\studyNodeJS\21-9-2> node util
3
(node:7752) DeprecationWarning: dontUseMe 함수는 deprecated되었으니 더 이상 사용하지 마세요!
(Use `node --trace-deprecation ...` to show where the warning was created)
xka3IxEs6v/sl0qTDURMEMi0GnBLTccbwYCoAmuuFLuX1HoOdXIBueLP35TYwKFV0FHYtLuZ2dDkylw8QyrZDQ==
PS C:\Users\ggg05\OneDrive\바탕 화면\studyNodeJS\21-9-2>
```

3.6 파일 시스템 접근하기

fs모듈은 파일 시스템에 접근하는 모듈이다.

```
저를 읽어주세요.
```

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

fs.readFile('./readme.txt', (err,data)=>{
    if (err){
        throw err;
    }
    console.log(data);
    console.log(data.toString());
});
```

readFile의 결과물은 **버퍼라는 형식으로 제공**된다.

버퍼는 사람이 읽을 수 있는 형식이 아니므로 toString()을 사용해 문자열로 변환한다.

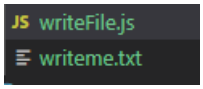
```
PS C:\Users\ggg05\OneDrive\바탕 화면\studyNodeJS\21-9-2> node readFile
<Buffer ec a0 80 eb a5 bc 20 ec 9d bd ec 96 b4 ec a3 bc ec 84 b8 ec 9a 94 2e>
저를 읽어주세요.
```

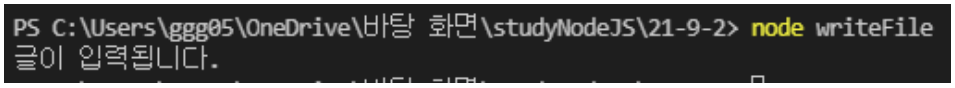
이제는 파일을 만들어본다.

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

fs.writeFile('./writeme.txt', '글이 입력됩니다.', (err)=>{
    if (err){
        throw err;
    }
    fs.readFile('./writeme.txt', (err,data)=>{
        if(err){
            throw err;
        }
        console.log(data.toString());
    });
});
```

첫 번째 인자로 파일 경로와 이름을 넣고 두 번째 인자에 내용을 입력한다.

해당 디렉터리에 파일이 생성되고 파일에 내용이 로깅된다.





3.6.1 동기 메서드와 비동기 메서드

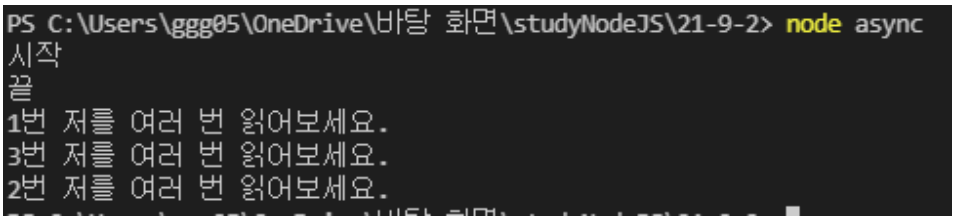
노드에 대부분의 메서드는 비동식 방식으로 처리하지만, 몇몇 메서드는 동기 방식으로도 사용할 수 있다.

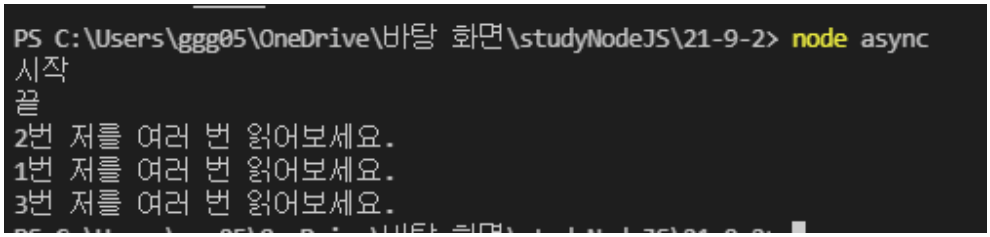
```
저를 여러 번 읽어보세요.
```

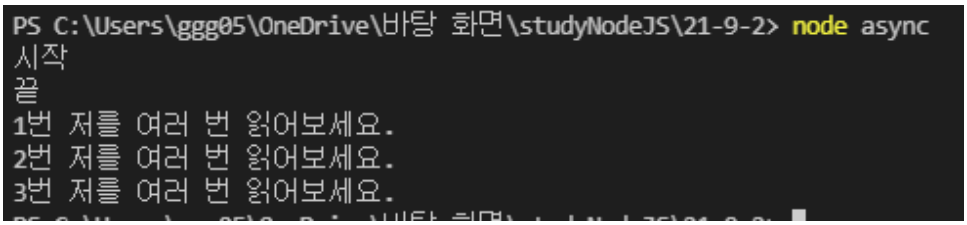
```
const fs=require('fs');

console.log('시작');
fs.readFile('./readme2.txt', (err, data)=>{
    if(err){
        throw err;
    }
    console.log('1번', data.toString());
});
fs.readFile('./readme2.txt', (err, data)=>{
    if(err){
        throw err;
    }
    console.log('2번', data.toString());
});
fs.readFile('./readme2.txt', (err, data)=>{
```

```
    if(err){
        throw err;
    }
    console.log('3번', data.toString());
});
console.log('끝');
```

해당 코드를 실행해보면







이렇게 할 때마다 다른 결과가 나오게 된다.

비동기 메서드들은 백그라운드에 해당 파일을 읽으라고만 요청하고 다음 작업으로 넘어간다.

따라서 파일 읽기 요청만 세 번 보내고 console.log('끝')를 실행한다.

나중에 읽기가 완료되면 백그라운드가 다시 메인 스레드에 알림을 주고 메인 스레드는 그때 등록된 콜백 함수를 실행한다.

💡 **동기와 비동기** : 함수가 바로 return되는지 여부
블로킹과 논블로킹 : 백그라운드 작업 완료 여부
동기-블로킹 : 백그라운드 작업 완료 여부를 계속 확인하며, 호출한 함수가 바로 return되지않고 백그라운드 작업이 끝나야 return된다.
비동기-논블로킹 : 호출한 함수가 바로 return되어 다음 작업으로 넘어가고, 백그라운드 작업 완료 여부는 신경 쓰지 않고 나중에 백그라운드가 알림을 줄 때 처리한다.

readFileSync 메서드를 활용하면 **동기 메서드이기 때문에 백그라운드가 작업하는 동안 메인 스레드는 아무것도 못하기 대기**하고 있어야한다. 따라서 **굉장히 비효율적**이며 사용해야 하는 경우가 드물다.