KB

It's Your Life

with

Callback!?


그만해.

Callback

Hell?

콜백 지옥? 그게 뭐죠?

callbackhell.js

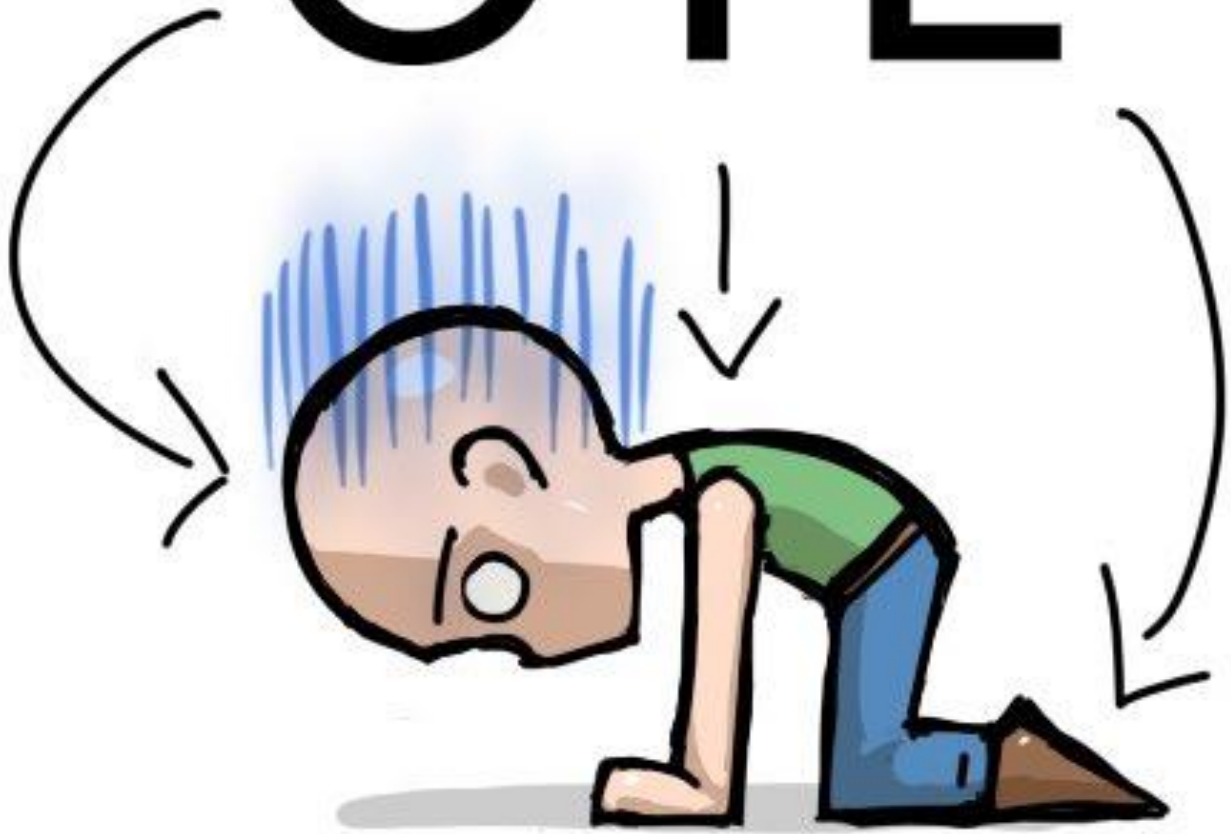
```
var floppy = require('floppy');

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function() {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', f
                floppy.load('disk3', function (data
                    floppy.prompt('Please insert di
                        floppy.load('disk4', functi
                            floppy.prompt('Please i
                                floppy.load('disk5'
                                    floppy.prompt('
                                        floppy.load
                                            //if no
                                        });
                                    });
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});
```

```
foo(() => {
    bar(() => {
        baz(() => {
            qux(() => {
                quux(() => {
                    quuz(() => {
                        corge(() => {
                            grault(() => {
                                run();
                            }).bind(this);
                        }).bind(this);
                    }).bind(this);
                }).bind(this);
            }).bind(this);
        }).bind(this);
    }).bind(this);
}).bind(this);
```

```
function callbackHellFunc(cb) {
  cb();
}

callbackHellFunc(function () {
  console.log(`1 번째 콜백 호출`);
  callbackHellFunc(function () {
    console.log(`2 번째 콜백 호출`);
    callbackHellFunc(function () {
      console.log(`3 번째 콜백 호출`);
      callbackHellFunc(() => {
        console.log(`4 번째 콜백 호출`);
        callbackHellFunc(() => {
          console.log(`5 번째 콜백 호출`);
          callbackHellFunc(() => {
            console.log(`6 번째 콜백 호출`);
            callbackHellFunc(() => {
              console.log('그만해.... 제발....');
            });
          });
        });
      });
    });
  });});
```

그 console.log
두 번째랑 네 번째를
좀 바꿔주겠니!?

OTL

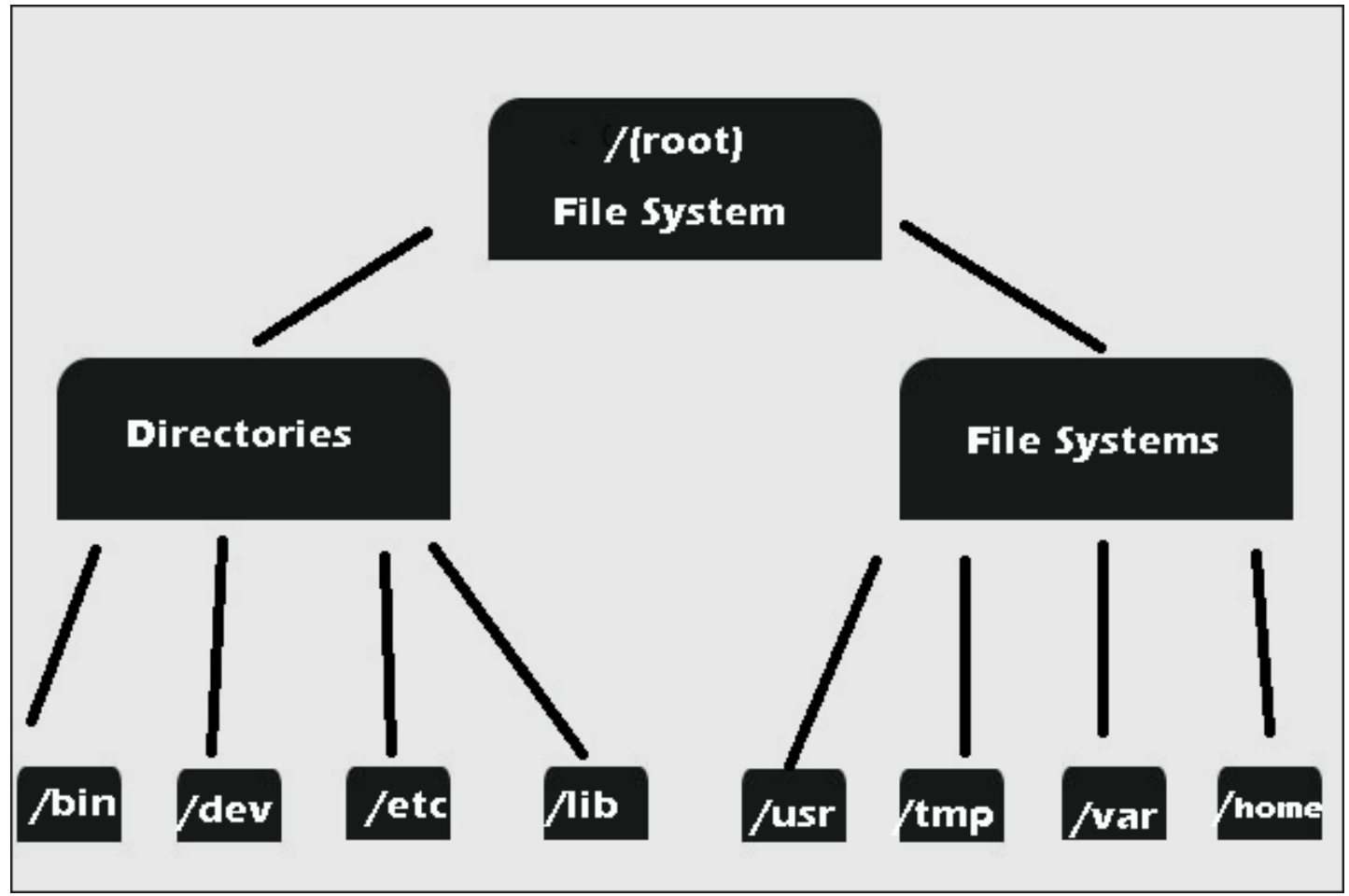



HELL!!!!!!!!

Callback Hell

체험 해보기!

File-System on JS


/(root)
File System
Directories
File Systems
/bin
/dev
/etc
/lib
/usr
/tmp
/var
/home

File-system 은 Node 의 기본 모듈 입니다!

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

- 파일을 읽는 것도 시간이 필요한 작업이므로 서버 통신과 비슷합니다 → 따라서, 비동기적 처리가 필요하며 기본 적으로 callback 을 지원합니다!
- fs.readFile('파일위치', '유니코드포맷', callback(err, data) {})
- err 은 파일 읽기가 잘 안되었을 때, Error 코드를 반환 합니다
- data 는 파일 읽기가 잘 되엇을 때, 읽은 data 를 반환합니다.

파일 읽기!

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

fs.readFile('readme.txt', 'utf-8', function (err, data) {
  if (err) {
    console.log(err);
  } else {
    console.log(data);
  }
});
```

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

fs.readFile('readme.txt', 'utf-8', (err, data) => {
  if (err) {
    console.log(err);
  } else {
    console.log(data);
  }
});
```

File-system 으로 비동기 프로그래밍 체험하기

File-system 과 비동기 프로그래밍

- 파일 시스템을 이용해서 비동기 프로그래밍 코드를 짜봅시다!
- JS의 특성으로 인해 각각 readFile 메소드를 동시에 비동기적으로 실행 시켜 봅시다!
- 동시에 readme.txt 파일을 읽어서 console.log 로 출력하는 코드를 작성해 봅시다!

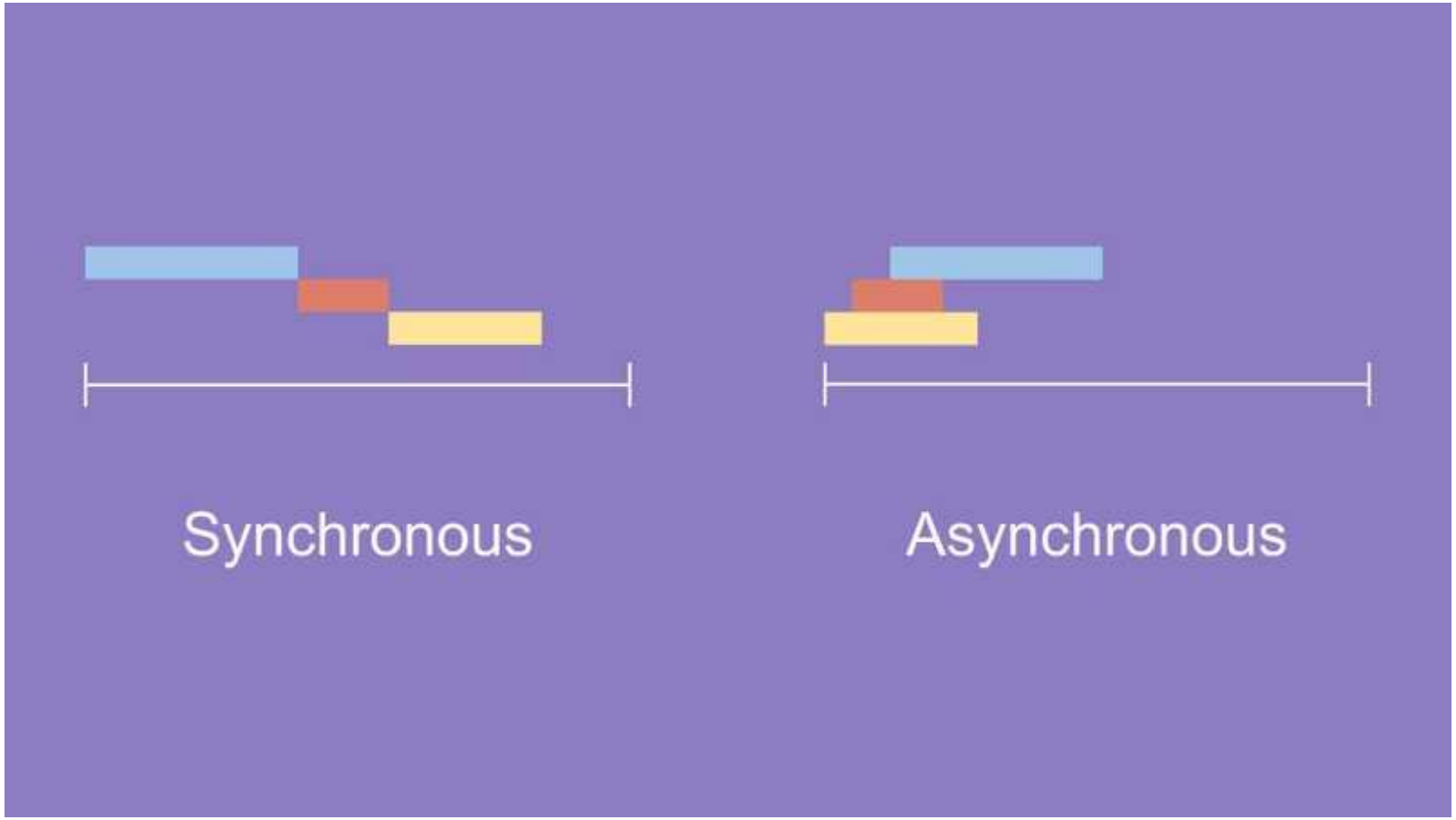
```
const fs = require('fs');

fs.readFile('./readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('1번', data.toString());
});
fs.readFile('readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('2번', data.toString());
});
fs.readFile('readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('3번', data.toString());
});
fs.readFile('readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('4번', data.toString());
});
```

File-system 과 비동기 프로그래밍

- 과연 결과는 어떻게 될까요!?

```
[Running] node "d:\git\KB_
1번 readme 텍스트 입니다!
2번 readme 텍스트 입니다!
3번 readme 텍스트 입니다!
4번 readme 텍스트 입니다!
```

```
[Running] node "d:\git\K
2번 readme 텍스트 입니다!
1번 readme 텍스트 입니다!
3번 readme 텍스트 입니다!
4번 readme 텍스트 입니다!
```


Synchronous
Asynchronous

File-system 과 비동기 프로그래밍

- File 을 읽는 것은 JS가 각각의 Thread 에 실어서 처리하기 때문에 각각의 Thread 상황에 따라 file 읽는 속도가 다르게 됩니다.
- 따라서, 꼭 1, 2, 3, 4 로 실행 된다는 보장이 없죠!
- 그럼 1, 2, 3, 4 순서로 실행 시키려면 어찌하면 될까요?

Callback

Hell?!

실습, 콜백 지옥 코드로 구현

- Fs 를 이용하여 파일 읽기 결과가 반드시 1번, 2번, 3번, 4번으로 나올 수 있게 코드를 구현하여 봅시다!
- 콜백 지옥을 사용하여 구현해야 합니다!

```
[Running] node "d:\git\K
1번 readme 텍스트 입니다!
2번 readme 텍스트 입니다!
3번 readme 텍스트 입니다!
4번 readme 텍스트 입니다!
```

Callback 지옥으로 구현하기!

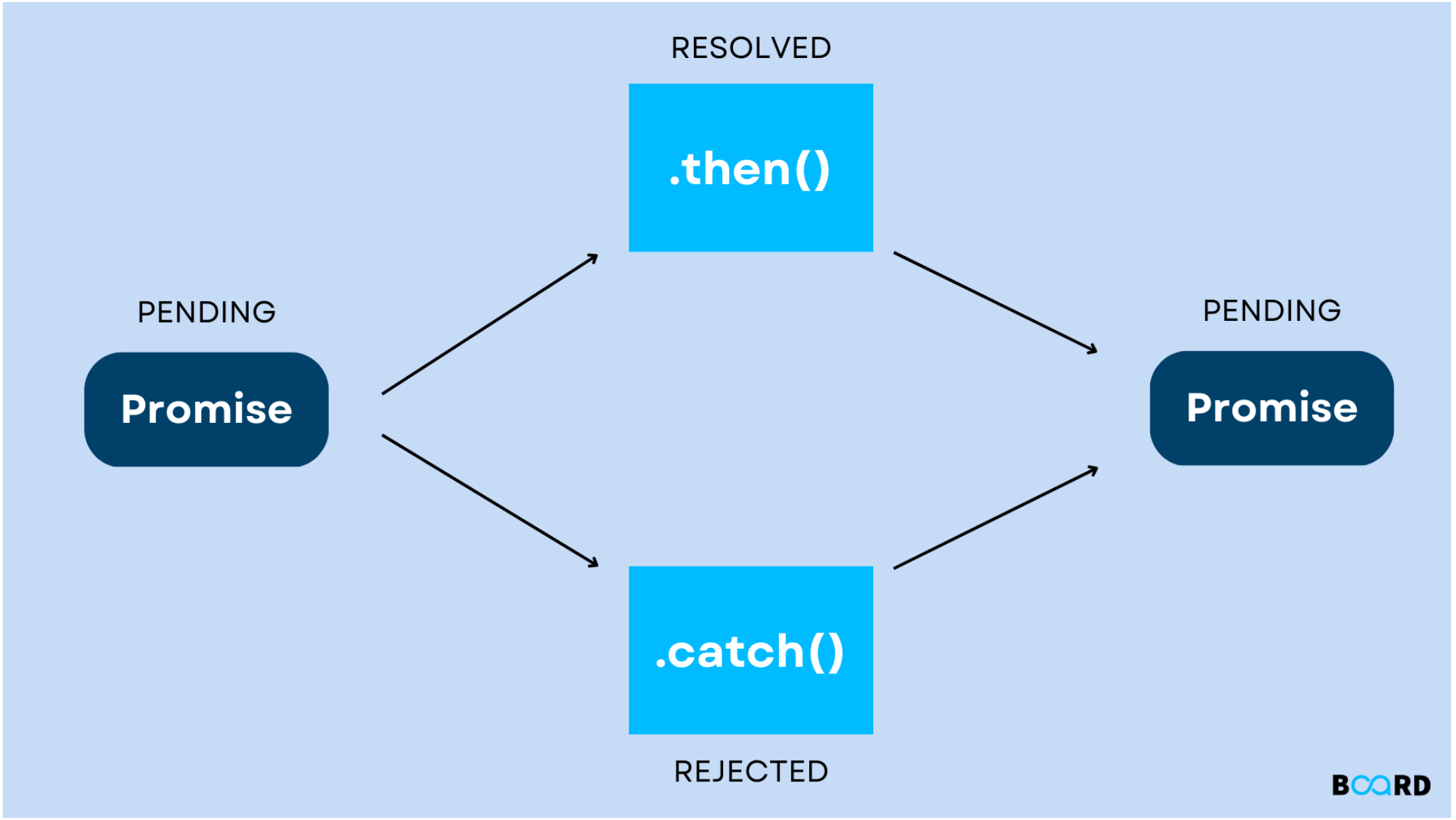
- 콜백으로 구현을 해도 의도했던 것 처럼 1, 2, 3, 4 가 순서대로 실행이 됩니다!
- 다만 구조 자체가 아까 말씀 드렸던, 가독성과 수정에 좋지 않은 콜백지옥의 형태를 가지게 됩니다
- 그럼, 이 콜백 지옥을 어찌 탈출 할 수 있을까요!?

이상
콜백지옥 탈출 방법에
대해 알아 봤습니다!

Promise!

Promise!

- Promise 는 Callback 을 대체하기 위해서 나온 개념입니다
- 말 그대로 '약속', 특정 작업을 수행한 다음 작업이 완료 되면 해당 결과 값을 돌려 주겠다는 약속! 입니다
- 대신, 비동기적 처리를 위해서 특정 약속이 수행 중일 때 JS 가 해당 결과를 기다려주는 기능이 추가가 되었습니다!


RESOLVED
.then()
PENDING
Promise
PENDING
Promise
.catch()
REJECTED
BOARD

Promise!

- Promise 는 생성자 입니다! 따라서 new 로 사용하죠!

```
const promise = new Promise(function(resolve, reject) {});
```

- resolve, reject 라는 2개의 콜백 함수를 받아서 사용합니다.
- promise 가 할당 되면 이 promise 는 resolve 또는 reject 함수가 callback 될 때 까지 무한 대기 합니다.
- Resolve 는 promise 가 정상적으로 이행 되었을 경우 사용하며, reject 는 반대의 경우에 사용합니다.

Promise!

- resolve 는 추후에 then 으로 받으며, reject 는 catch 로 받습니다!
- resolve, reject 콜백 함수의 경우는 데이터를 매개변수로 보낼 수 있습니다.
- Resolve, reject 가 사용되지 않으면 promise 는 해당 콜백이 나올 때 까지 pending 상태가 되어 기다립니다!

Promise

올때까지 여기서 기다릴거야

```
const promise = new Promise(function (resolve, reject) {
  console.log('프로미스 시작!');
  setTimeout(() => {
    console.log('setTimeout 끝!');
    resolve('프로미스로 비동기 구현 성공!');
  }, 2000);
});

console.log(promise);

promise.then(function (data) {
  console.log(data);
});
```

```
프로미스 시작!
Promise { <pending> }
setTimeout 끝!
프로미스로 비동기 구현 성공!
```

setTimeout 으로 2초 대기!
& resolve 가 나올 때 까지
Pending 상태로 대기

콜백 지옥을 Promise 로 변경 하기

- fs.promises 를 사용해서 콜백 지옥을 promise 코드로 변경해 봅시다
- fs.promises 해당 메소드의 판단 여부를 스스로 판단 후 →
- 파일 읽기가 성공 하면 resolve
- 실패하면 reject 를 알아서 알아서 반환 합니다! → 고로 편리합니다!

```js
const fs = require('fs').promises;
fs.readFile('./readme.txt')
  .then((data) => {
    console.log('1번', data.toString());
    return fs.readFile('./readme.txt');
  })
  .then((data) => {
    console.log('2번', data.toString());
    return fs.readFile('./readme.txt');
  })
  .then((data) => {
    console.log('3번', data.toString());
    return fs.readFile('./readme.txt');
  })
  .then((data) => {
    console.log('4번', data.toString());
  })
  .catch((err) => {
    throw err;
  });
```

```
[Running] node "c:\Users\s
1번 readme 텍스트 입니다!
2번 readme 텍스트 입니다!
3번 readme 텍스트 입니다!
4번 readme 텍스트 입니다!

[Done] exited with code=0

[Running] node "c:\Users\s
1번 readme 텍스트 입니다!
2번 readme 텍스트 입니다!
3번 readme 텍스트 입니다!
4번 readme 텍스트 입니다!

[Done] exited with code=0

[Running] node "c:\Users\s
1번 readme 텍스트 입니다!
2번 readme 텍스트 입니다!
3번 readme 텍스트 입니다!
4번 readme 텍스트 입니다!
```

실습, 콜백 코드를 promise 로 구현하기

- 아래의 콜백 코드를 promise 로 구현하세요

```js
function shouldIBuyLotto(callback) {
  console.log('나 로또 사도 될까!?');
  setTimeout(() => {
    const rand = parseInt(Math.random() * 10);
    console.log(`나온 숫자는 ${rand}`);

    // rand 가 5 이상이면 로또 사자!
    if (rand >= 5) {
      callback('아싸! 로또 사자!');
    } else {
      callback('아... 망했어요...');
    }
  }, 3000);
}

function showResult(msg) {
  console.log(msg);
}

shouldIBuyLotto(showResult);
```

```
[Running] node "c:\Users
나 로또 사도 될까!?
나온 숫자는 9
아싸! 로또 사자!

[Done] exited with code=

[Running] node "c:\Users
나 로또 사도 될까!?
나온 숫자는 3
아... 망했어요...
```

Async / Await

최신 기술인 Async, Await 도 적용!

- Function 앞에 async 를 붙이면 해당 함수는 항상 Promise 를 반환
- 즉, async 가 붙은 함수에서 return 을 쓰면 아래와 동일한 역할을 합니다.

```
async function f1() {
  return 1;
}

async function f2() {
  return Promise.resolve(1);
}
```

- Async 가 붙은 함수 내부에는 Await 키워드 사용이 가능!
- Await 은 promise 가 결과(resolve, reject)를 가져다 줄 때 까지 기다립니다.

Async, Await 도 적용!

- 단, async 는 함수를 정의하는 상황에서 쓰이므로 함수 정의 후, 해당 함수를 외부에서 한번 사용해 줘야합니다!

Await

```
const promise = new Promise(function (resolve, reject) {
  console.log('프로미스 시작!');
  setTimeout(() => {
    console.log('setTimeout 끝!');
    resolve('프로미스로 비동기 구현 성공!');
  }, 2000);
});

promise.then(function (data) {
  console.log(data);
});
```

이 부분을 더 보기 편하게 바꿔볼 예정입니다!

그때 async/await 를 씁니다!

```
const promise = new Promise(function (resolve, reject) {
  console.log('프로미스 시작!');
  setTimeout(() => {
    console.log('setTimeout 끝!');
    resolve('프로미스로 비동기 구현 성공!');
  }, 2000);
});

async function asynFunc() {
  const result = await promise;
  console.log(result);
}

asynFunc();
```

Await 에 의해 JS 가 저 위치에서 동작을 멈춥니다!

멈춘 동안 위에서 선언한 Promise 에서 시간이 흐르고
Resolve 가 반환 되는 순간!

Await 는 resolve 와 함께 사라져 버리고
Resolve 가 가져온 데이터만 리턴(=남게) 됩니다

```
프로미스 시작!
setTimeout 끝!
프로미스로 비동기 구현 성공!
```

Syntactic Sugar

- 같은 기능을 하지만 문법 상으로 더 편리하게 바꿔 주는 것을 Syntactic Sugar 라 부릅니다!
- Async, Await 는 promise 의 Syntactic Sugar 입니다!
- Promise 는 기존에 JS 코드 스타일과 다르기 때문에 promise 를 기존 코드 스타일로 사용할 수 있도록 만들어 준 것이 Async, Await 입니다!

```
promise.then(function (data) {
  console.log(data);
});
```

```
async function asynFunc() {
  const result = await promise;
  console.log(result);
}

asynFunc();
```

그런데 이거 왜 쓰나요!?

그런데 이거 왜 쓰나요!?

```
const fs = require('fs').promises;
fs.readFile('./readme.txt')
  .then((data) => {
    console.log('1번', data.toString());
    return fs.readFile('./readme.txt');
  })
  .then((data) => {
    console.log('2번', data.toString());
    return fs.readFile('./readme.txt');
  })
  .then((data) => {
    console.log('3번', data.toString());
    return fs.readFile('./readme.txt');
  })
  .then((data) => {
    console.log('4번', data.toString());
  })
  .catch((err) => {
    throw err;
  });
```

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

async function main() {
  let data = await fs.readFile('./readme.txt');
  console.log('1번', data.toString());

  data = await fs.readFile('./readme.txt');
  console.log('2번', data.toString());

  data = await fs.readFile('./readme.txt');
  console.log('3번', data.toString());

  data = await fs.readFile('./readme.txt');
  console.log('4번', data.toString());
}

main();
```

Async / Await 로 콜백지옥 탈출!

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

async function main() {
  let data = await fs.readFile('./readme.txt');
  console.log('1번', data.toString());

  data = await fs.readFile('./readme.txt');
  console.log('2번', data.toString());

  data = await fs.readFile('./readme.txt');
  console.log('3번', data.toString());

  data = await fs.readFile('./readme.txt');
  console.log('4번', data.toString());
}

main();
```

```
[Running] node "c:\Users\st
1번 readme 텍스트 입니다!
2번 readme 텍스트 입니다!
3번 readme 텍스트 입니다!
4번 readme 텍스트 입니다!

[Done] exited with code=0 i

[Running] node "c:\Users\st
1번 readme 텍스트 입니다!
2번 readme 텍스트 입니다!
3번 readme 텍스트 입니다!
4번 readme 텍스트 입니다!

[Done] exited with code=0 i

[Running] node "c:\Users\st
1번 readme 텍스트 입니다!
2번 readme 텍스트 입니다!
3번 readme 텍스트 입니다!
4번 readme 텍스트 입니다!
```

실습, promise 코드를 async/await 로 변경

- 이전 실습에서 구현했던 promise 코드를 async / await 로 변경해 주세요 ~! ☺

실제 서버 통신 체험 해보기!

Express

http://expressjs.com/

Back-End

서버의 기본!

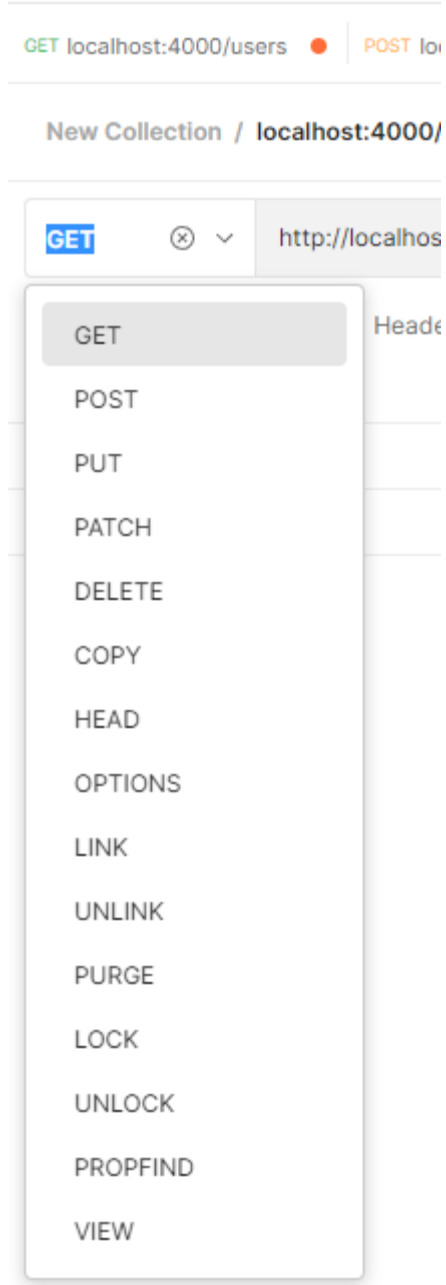
요청 메소드

Create ➡ POST

Read ➡ GET

Update ➡ PUT

Delete ➡ DELETE


GET localhost:4000/users
POST lo
New Collection / localhost:4000/
GET
http://localhos
Heade
GET
POST
PUT
PATCH
DELETE
COPY
HEAD
OPTIONS
LINK
UNLINK
PURGE
LOCK
UNLOCK
PROPFIND
VIEW

HTTP Status Codes

1XX
INFORMATIONAL

2XX
SUCCESS

3XX
REDIRECTION

4XX
CLIENT ERROR

5XX
SERVER ERROR

HTTP Status

- 100 번대 : 정보 / 리퀘스트를 받고 처리 중
- 200 번대 : 성공 / 리퀘스트를 정상 처리
- 300 번대 : 리디렉션 / 처리 완료를 위해서는 추가 동작 필요
- 400 번대 : 클라이언트 에러 / 클라이언트에서 요청을 잘못 보냄
- 500 번대 : 서버 에러 / 리퀘스트는 잘 들어 갔지만 서버에서 처리를 못함

HTTP Status Codes

Level 200 (Success)

200 : OK

201 : Created

203 : Non-Authoritative Information

204 : No Content

Level 400

400 : Bad Request

401 : Unauthorized

403 : Forbidden

404 : Not Found

409 : Conflict

Level 500

500 : Internal Server Error

503 : Service Unavailable

501 : Not Implemented

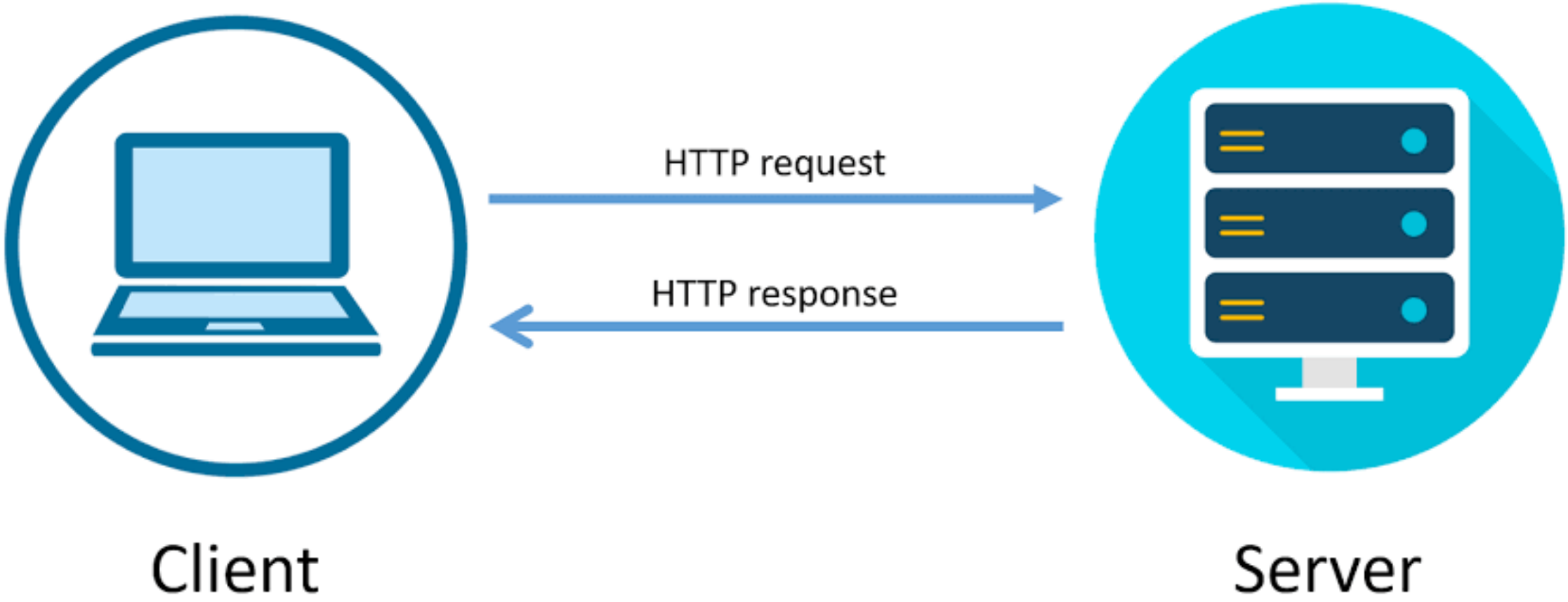
504 : Gateway Timeout

599 : Network timeout

502 : Bad Gateway

간단한

통신 경험하기!


HTTP request
HTTP response
Client
Server

Express 서버 구축

```
const express = require('express');
const cors = require('cors');

const PORT = 4000;

const app = express();
app.use(cors());

app.get('/request', (req, res) => {
  res.status(200).json('안녕하세요. 여기는 백엔드 입니다!');
});

app.get('/error', (req, res) => {
  res.status(500).json('비상!!! 에러 발생!!!!!');
});

app.listen(PORT, () => {
  console.log(`데이터 통신 서버가 ${PORT}에서 작동 중입니다!`);
});
```

```html
<!DOCTYPE html>
<html lang="en">
  <body>
    <h1 class="header">Hello, Protocol</h1>
    <button onclick="fetchData()">백엔드 통신 경험하기</button>
    <button onclick="errorHandling()">백엔드 에러 경험하기</button>
  </body>

  <script>
    const headerEl = document.querySelector('.header');
    const fetchData = () => {
      fetch('http://localhost:4000/request')
        .then((res) => {
          return res.json();
        })
        .then((data) => {
          console.log(data);
          headerEl.innerHTML = data;
        });
    };
```

```
async function errorHandling() {
      try {
        const res = await fetch('http://localhost:4000/error');
        const data = await res.json();
        console.log(data);
        headerEl.innerHTML = data;
      } catch (err) {
        console.log(err);
      }
    }
  </script>
</html>
```

Hello, Protocol

백엔드 통신 경험하기 백엔드 에러 경험하기

안녕하세요. 여기는 백엔드 입니다!

백엔드 통신 경험하기 백엔드 에러 경험하기

비상!!! 에러 발생!!!!!

백엔드 통신 경험하기 백엔드 에러 경험하기