7/27 ~ 7/31 4주차 - 4

[JS] 비동기 (axios , promise, async ,,)

동기

- 동기는 말 그디로 동시에 일어난다는 뜻
- 요청과 그 결과가 동시에 일어난다는 약속
- 바로 요청을 하면 시간이 얼마가 걸리던지 요청한 자리에서 결과가 주어져야 한다.

  → 요청과 결과가 한 자리에서 동시에 일어남

  → A노드와 B노드 사이의 작업 처리 단위(transaction)를 동시에 맞춘다.

비동기

- 비동기는 동시에 일어나지 않는다.
- 요청과 결과가 동시에 일어나지 않을 거라는 약속

  → 요청한 그 자리에서 결과가 주어지지 않음

  → 노드 사이의 작업 처리 단위를 동시에 맞추지 않아도 된다.

자바스크립트는 싱글 스레드 프로그래밍언어기 때문에 비동기 처리가 필수적이다.

비동기 처리는 요청의 결과가 언제 반환될지 알 수 없기 때문에 동기식으로 처리하는 기법들이 사용되어야한다.

대표적으로 setTimeout이 있고 callback과 promise가 있습니다.

세 가지 모두 비동기 코드를 동기식으로 작성하는데 훌륭한 기법들이지만, 모두 약간의 문제점을 가지고 있다.

async와 await는 이런 문제들을. 해결함과 동시에 그 사용법에 있어서도 훨씬 단순해졌다.

비동기적 Javascript

- C, Java, Python 을 사용하면 상식적으로 별도의 스레드나 프로세스를 사용하지 않는 이상, 먼저 작성된 순서대로 즉, 동기적으로 코드가 실행된다.
- 3번째 줄에 있는 코드의 작업이 5번째 줄에 있는 코드보다 늦게 끝나는 비상식적인 일은 발생하지 않는다는 뜻이다.
- 하지만 자바스크립트는 먼저 실행된 코드의 작업이 끝나기 전에 더 나중에 실행된 코드의 작업이 끝날 수 있다.

아주 간단한 예

```
function first() {
  setTimeout(() => {
      console.log("The First function has been called.")
    }, 1000)
}


function second() {
  setTimeout(() => {
      console.log("The Second function has been called.")
    }, 500)
}

first() second()
```

- first 함수가 호출되면, setTimeout 을 통해 **1000ms** 가 지나고서야 문장이 출력되지만, second 함수는 문자열 출력에 고작 **500ms** 밖에 걸리지 않는다.

- 코드에서는 first 함수를 먼저 호출했지만, 결과는 다음과 같다.

```
The Second function has been called.
The First function has been called.
```

- 이것이 Javascript 의 비동기성이다.
- 하지만, **자바스크립트는 하나의 스레드 (Single Thread) 기반의 언어**이다.
- 즉, 자바스크립트는 **한번에 하나의 작업밖에 수행하지 못한다**는 의미이다.
- 그런데 이상하다. 자바스크립트는 위의 간단한 예제는 물론이고, Ajax로 데이터를 불러오면서 Mouseover 이벤트를 처리하면서 애니메이션을 동작시킨다.

어떻게 이런 동시성 (Concurrency) 이 가능한 것일까? Javascript Engine 의 구조를 살펴보면 좋은데 나중에 하기.

setTimeout에 대해서

- setTimeout 은 특정 시간 동안 기다렸다가 이후 첫번째 파라미터의 함수를 실행하는 방식을 사용합니다.

```
let first =10;

function add(x, y) {
  return x + y;
}

setTimeout(() => {
  const result = add(first, 20);
  console.log(result);
}, 1000);

// 결과 30
```

```
let first =10;

function add(x, y) {
  return x + y;
}

setTimeout(() => {
  const result = add(first, 20);
  console.log(result);
}, 1000);
first =30;

// 결과 50
```

이런 식이면 문제가 생길 수 있다.

- 자바스크립트는 각각의 task를 큐에 적재해두고 순서대로 처리한다.
- 이 때 어떤 코드가 새로운 태스크로 적재되지에 대한 이해가 부족하면 위와 같은 실수를 저지를 수 있다.
- 최초의 task 는 스크립트 파일 자체이다.
- 이 첫번째 task 내에 setTimeout은 별도의 task를 생성하고 첫번째 task 가 종료되길 기다린다.
- 첫번째 task 인 스크립트의 실행이 끝나면 비로소 setTimeout 의 함수를 실행할 준비를 한다.
- 즉, first 의 값은 초기에 10 이였지만 첫번째 스크립트가 종료되면 20 이 되기때문에 결과적으로 result 는 40 이 된다.

CallBack

- callback 함수란 호출하는 함수(calling function)가 호출되는 함수(called 함수)로 전달하는 함수를 말하며 이때 callback 함수의 제어권은 호출되는 함수에게 있다.
- 쉽게 이야기 하면 비동기 처리가 끝나고 행동하고 싶은 일을 넣어 두는 것이다. setTimeout 에서 first=20을 넣었던 것과 같이
- callback 함수는 setTimeout 함수와 같은 비동기 코드를 동기식으로 처리하기 위해 사용한다.

- production 에 사용되는 코드에서는 보통 네트워크 요청 등의 비동기 코드에 많이 사용된다.

```
function test(n, callback) {
  setTimeout(() => {
    const number = n + 1;
    console.log(number);
    if (callback) {
      callback(number);
    }
  }, 1000);
}

test(0, n => {
  test(n, n => {
    test(n, n => {
      console.log("작업 끝");
    });
  });
});
```

- 하지만 callBack 함수가 많아지면 굉장히 코드가 난잡해진다
- callback 을 연달아 사용하게 되면 에러가 발생할 가능성이 높고, 코드의 가독성도 크게 떨어지게 된다.
- callback 은 비동기 코드를 동기적 만드는데 확실한 방법이긴 하지만 남발하게 되면 가독성이 크게 떨어지고 코드의 복잡성도 크게 증가하게 된다.
- 또한 callback 의 호출에 대한 제어권이 다른 함수들에게 넘어가 버리기 때문에 각 콜백 함수가 언제 어떻게 몇번 실행되는지 확신을 할 수 없다.
- 이렇게 코드를 작성하면(물론 잘 하면 상관없지만), 특히 여러명이서 코드를 공유하는 경우라면 결과를 예측하기 어려울 뿐 아니라 코드내에서 에러가 발생할 확률도 높아진다.

Promise

- promise 어떤 작업이 성공했을 때(resolve), promise 객체의 then() 함수에 넘겨진 파라미터(함수)를 단 한번만 호출하겠다는 약속이다.
- callback 의 경우 제어권이 호출되는 함수로 넘어가 버리기 때문에 신뢰성이 다소 떨어지지만 promise 는 함수 실행이 성공했을때 then() 함수의 파라미터(함수)가 단 한번만 호출되기 때문에 함수를 호출하는 입장에서 확신을 가지고 코드를 작성할 수 있다.
- 또한 실패했을 경우(reject)에도 catch()함수를 통해서 실패 이후의 작업을 처리할 수 있습니다.

실패와 성공 모두를 잡을 수 있다. 콜백 함수에서 발생 할 수 있는 콜백지옥을 없애기 위해서 라이브러리로 만들어졌지만 사용 빈도도 높고 해서 JS 에 공식적으로 등록 되었다.

```
const myPromise = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    resolve('result');
  },1000)
});

myPromise.then((res)=>{
  console.log(res);
}).catch((erro)=>{
  console.log(erro);
})
```

- 결과를 then 과 catch 로 받을 수 있다. 성공하게 되면 then 으로 받고 실패 하게 되면 catch 로 받게 된다.

Error 를 던졌을 때

```
const myPromise = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    reject(new Error('erro'))
  },1000)
});

myPromise.then((res)=>{
  console.log(res);
}).catch((erro)=>{
  console.error(erro);
})
```

// error 받을 때는 (console.error 라고 사용한다.)

async / await

```
function goTobad(ms){
  return new Promise( resolve => setTimeout(resolve,ms));

}

async function process(){
  console.log("hi");
  await goTobad(1000);
  console.log("good afternoon");
}

process();
```

```
hi
// 1초 뒤에
good afternoon
```

await 를 promise 앞에 써주면 된다.

[javascript] async, await를 사용하여 비동기 javascript를 동기식으로 만들자

async, await 는 ES8(ECMAScript2017)의 공식 스펙(링크)으로 비교적 최근에 정의된 문법입니다. async, await를 사용하면 비동기 코드를 작성할 때 비교적 쉽고 명확하게 코드를 작성할 수 있습니다. 자바스크립트는 싱글 스레드 프로그래밍언어기 때문에 비동기처리가 필수적입니다. 비동기 처리는 그 결과가 언제 반환될지 알수 없기 때문에 동기식으로 처

https://blueshw.github.io/2018/02/27/async-await/