



# 21.8.26 2-1-6 프로미스 ~

<참조>

[ES6] Promises - then, catch, all, race, finally

이번 포스팅에서는 js 의 Promises 에 대해서 알아보도록 하겠습니다.  
사람은 한 번에 두 가지 일을 할 수가 없습니다. 흔히들 말하는 멀티태스킹도 실제로는 한 번에 두 가지 일을 동시에 하는 것이 아니라 빠르

☞ <https://ssungkang.tistory.com/entry/ES6-Promises-then-catch-all-race-finally>

**ES6**  
ECMAScript 2015

## 2.1.6 프로미스

자바스크립트와 노드에서는 주로 비동기 프로그래밍을 한다.

특히 이벤트 주도 방식 때문에 콜백 함수를 자주 사용한다.

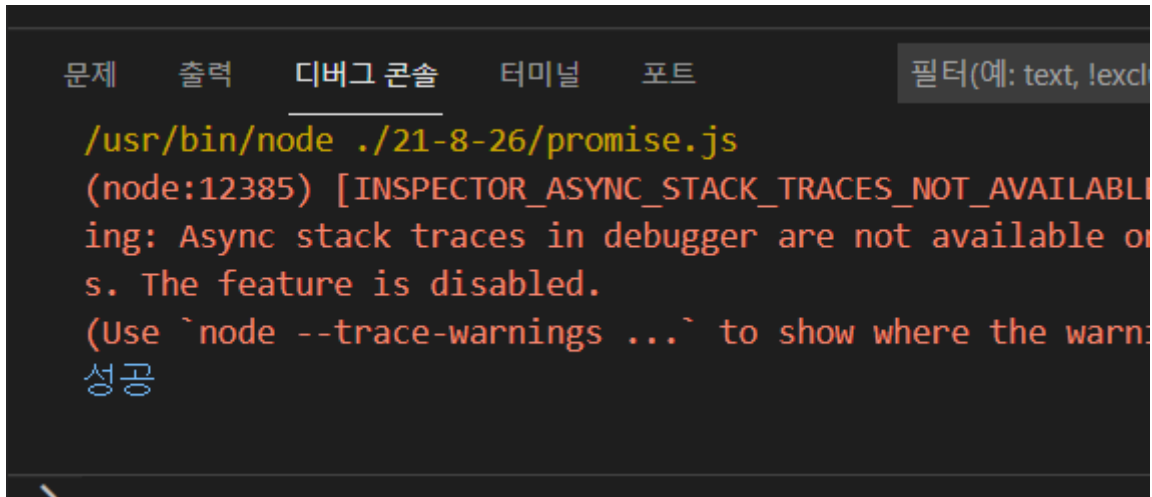
ES2015부터는 API들이 콜백 대신 프로미스 기반으로 재구성된다.

```
const condition = true;
const promise = new Promise((resolve, reject) => { //new로 promise객체 생성
  if (condition){
    resolve('성공');
  }
  else {
    reject('실패');
  }
});

promise
  .then((message) =>{
    console.log(message);
  })
  .catch((error)=> {
    console.error(error);
  });
```

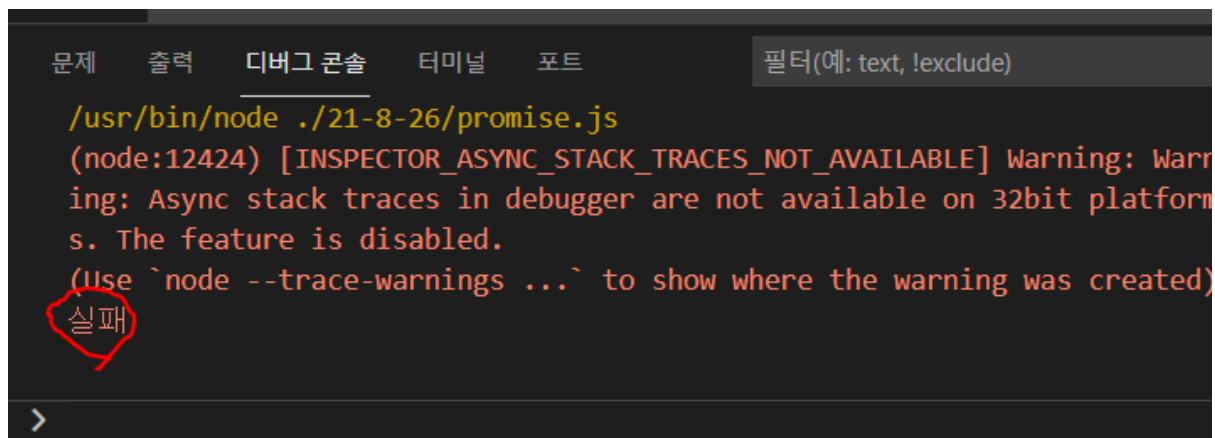
위 코드를 보면 condition값에 따라 resolve나 reject가 호출이 되고, **resolve가 호출이 되면 then이 reject가 호출되면 catch가 실행된다.**

현재 condition은 true이고 resolve를 호출할 것이다. 그리고 then이 실행될 것이고 '성공'이라는 메시지가 뜰 것이다.



The screenshot shows a Node.js debugger interface with tabs for '문제' (Issue), '출력' (Output), '디버그 콘솔' (Debug Console), '터미널' (Terminal), and '포트' (Port). The '디버그 콘솔' tab is active. The console output shows the command `/usr/bin/node ./21-8-26/promise.js` and a warning message: `(node:12385) [INSPECTOR_ASYNC_STACK_TRACES_NOT_AVAILABLE] Warning: Async stack traces in debugger are not available on 32bit platforms. The feature is disabled. (Use `node --trace-warnings ...` to show where the warning was created)`. Below the warning, the word '성공' (Success) is written in blue text.

반대로 condition을 false로 바꾸면 '실패'로 바뀔 것이다.



The screenshot shows the same Node.js debugger interface as the previous one. The console output is identical, including the command `/usr/bin/node ./21-8-26/promise.js` and the warning message. However, below the warning, the word '실패' (Failure) is written in red text and circled with a red hand-drawn line.

then이나 catch에 또 다른 then과 catch를 붙일 수 있다.

**이전 then의 return값을 다음 then의 매개변수로 넘긴다.**

프로미스를 return한 경우 프로미스가 수행된 후 다음 then이나 catch가 호출된다.

```

const condition = true;
const promise = new Promise((resolve, reject) => { //new로 promise객체 생성
  if (condition){
    resolve('성공');
  }
  else {
    reject('실패');
  }
});

promise
  //////////1//////////
  .then((message)=>{
    return new Promise((resolve, reject)=> {
      resolve(message);
    });
  })
  //////////2//////////
  .then((message2)=>{
    console.log(message2);
    return new Promise((resolve, reject)=>{
      resolve(message2);
    });
  })
  //////////3//////////
  .then((message3)=>{
    console.log(message3);
  })
  //////////4//////////
  .catch((error)=>{
    console.error(error);
  });

```

1. condition이 true이므로 조건문을 통해 resolve가 호출된다.
2. 1번 부분의 then의 매개변수로 resolve의 인자인 '성공'이 들어간다.
3. 1번의 Promise가 return되며 resolve가 호출되고 안에 인자인 message('성공')은 2번 부분의 매개변수로 들어간다.
4. 2번 부분 처음에 **message2('성공')**이 로깅되고 다시 Promise를 생성해 return하며 3번의 매개변수로 전달한다.
5. 3번의 매개변수인 **message3('성공')**이 로깅되며 실행을 마친다.

```
문제   출력   디버그 콘솔   터미널   포트   필터(예: text, !exclud

/usr/bin/node ./21-8-26/promise.js
(node:12546) [INSPECTOR_ASYNC_STACK_TRACES_NOT_AVAILABLE]
Warning: Async stack traces in debugger are not available on this platform. The feature is disabled.
(Use `node --trace-warnings ...` to show where the warning was thrown)
성공
성공
```

```
function findAndSaveUser(Users){ // 함수 선언(매개변수 : Users)
  Users.findOne({}, (err,user)=>{ //첫 번째 콜백
    if(err){ //오류처리1
      return console.log(err);
    }
    user.name = 'zero';
    user.save((err)=> { //두 번째 콜백
      if(err){ //오류처리2
        return console.error(err);
      }
      Users.findOne({gender: 'm'},(err,user)=>{ //세 번째 콜백
        //생략
      });
    });
  });
}
```

만약 이렇게 세 번의 콜백 함수가 중첩돼있는 경우 콜백 함수가 나올 때마다 코드의 깊이가 깊어짐.

+각 콜백 함수마다 에러처리도 처리해야함.

이런 코드를 위에서 배운 then과 catch를 통해 바꿀 수 있다.

```
function findAndSaveUser(Users){
  Users.findOne({})
    .then((user)=>{
      user.name = 'zero';
      return user.save();
    })
    .then((users)=> {
      return Users.findOne({gender: 'm'});
    });
}
```

```

    })
    .then((user)=>{
        //생략
    })
    .catch(err=>{
        console.error(err);
    });
}

```

## <정리>

**Promise**개체는 비동기 작업이 맞이할 미래의 완료 또는 실패와 그 결과 값을 나타냄

- 매개변수

**Promise**는 매개변수로 executor를 받는다. (executor란 **resolve**와 **reject** 인수를 전달할 실행함수)

resolve와 reject가 함수로서 호출되면 promise를 이행하거나 거부

- **Promise**는 대기상태 or 연산이 성공한 형태 or 연산에 실패한 상태 총 3가지 형태를 가진다.

Promise에도 종류가 있다.

- Promise all

all()은 모든 promise를 실행한 뒤 전부 실행한 후, 하나의 Promise로 반환한다.

3개의 promise가 모두 끝날 때 까지 기다리고 결과를 array로 반환한다.

```

const p1 = new Promise(resolve => {
    setTimeout(resolve, 2000, "First");
})

const p2 = new Promise(resolve => {
    setTimeout(resolve, 1000, "Second");
})

const p3 = new Promise(resolve => {
    setTimeout(resolve, 3000, "Third");
})

const allPromise = Promise.all([p1,p2,p3]);
allPromise.then(values => console.log(values));
// [ 'First', 'Second', 'Third' ]

```

예외 처리를 해줄 때는 allPromise에 대해서만 해주면 된다.

```

const p1 = new Promise((resolve, reject) => {
  setTimeout(reject, 2000, "First reject");
})

const p2 = new Promise(resolve => {
  setTimeout(resolve, 1000, "Second");
})

const p3 = new Promise(resolve => {
  setTimeout(resolve, 3000, "Third");
})

const allPromise = Promise.all([p1,p2,p3]);
allPromise
.then(values => console.log(values))
.catch(error => console.log(error));
// First reject

```

- Promise race

race()는 주어진 Promise 중 가장 먼저 완료된 것의 결과값을 이행하거나 거부한다.

```

const p1 = new Promise((resolve, reject) => {
  setTimeout(reject, 2000, "First reject");
})

const p2 = new Promise(resolve => {
  setTimeout(resolve, 1000, "Second"); //1초만에 끝나기 때문에
})                                     //Second가 실행된다.

const p3 = new Promise(resolve => {
  setTimeout(resolve, 3000, "Third");
})

const allPromise = Promise.race([p1,p2,p3]);
allPromise
.then(values => console.log(values))
.catch(error => console.log(error));
// Second

```

- Promise finally

finally()는 Promise가 resolve 되던 reject 되던 상관없이 지정된 함수를 실행한다.

promise의 결과에 상관없이 동작 해야할 때 유용하다.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, "First");
});

p1
  .then(result => console.log(result))
  .finally(()=>console.log("end")); //결과에 상관없이 실행
// First
// end
```

## 2.1.7 async/await

**async/await** 문법은 promise를 사용한 코드를 한 번 더 깔끔하게 줄여준다!

일단 위에서 한 예제를 하나 가져온다.

```
function findAndSaveUser(Users){
  Users.findOne({})
    .then((user)=>{
      user.name = 'zero';
      return user.save();
    })
    .then((users)=> {
      return Users.findOne({gender: 'm'});
    })
    .then((user)=>{
      //생략
    })
    .catch(err=>{
      console.error(err);
    });
}
```

이것을 **async/await** 문법을 사용하면 다음과 같이 바뀐다.

```
async function findAndSaveUser(User){
  let user = await URLSearchParams.findOne({});
  user.name = 'zero'
  user = await user.save();
  user = await URLSearchParams.findOne({gender: 'm'})
  //생략
}
```

**function 앞에는 async**가 각 **promise앞에는 await**가 붙어진 형태로 간단하게 바꿨다.

예를 들면 `await Users.findOne({})`이 resolve될 때 까지 기다린 뒤, `user` 변수를 초기화한다.

대신 아래와 같이 `try/catch`를 사용해서 예외처리를 해줘야한다.

```
async function findAndSaveUser(User){
  try{
    let user = await URLSearchParams.findOne({});
    user.name = 'zero'
    user = await user.save();
    user = await URLSearchParams.findOne({gender:'m'})
    //생략
  }
  catch(error){
    console.error(error);
  }
}
```