

8월 24일 (화)

1. Codecademy - Async JavaScript and HTTP Requests - Learn JavaScript Syntax : Promises

Lesson 1. JavaScript Promises

(이전 TIL 참고)

Article 1. Graceful Asynchronous Programming with Promises (MDN)

출처 :

<https://developer.mozilla.org/ko/docs/Learn/JavaScript/Asynchronous/Promises>

- [What are promises?](#)
- [The trouble with callbacks](#)
- [Explaining basic promise syntax: A real example](#)
- [Promise terminology recap](#)
- [Running code in response to multiple promises fulfilling](#)
- [Running some final code after a promise fulfills/rejects](#)
- [Building your own custom promises](#)

- What are promises?

What are promises?

앞서서 [Promises](#) 를 미리 봤지만, 지금부터 좀더 깊이있게 들여다 볼 차례입니다..

Promise는 어떤 작업의 중간상태를 나타내는 오브젝트입니다. — 미래에 어떤 종류의 결과가 반환됨을 *promise* (약속) 해주는 오브젝트라고 보면 됩니다. Promise는 작업이 완료되어 결과를 반환해주는 정확한 시간을 보장해주지는 않지만, 사용할 수 있는 결과를 반환했을때 프로그래머의 의도대로 다음 코드를 진행 시키거나, 에러가 발생했을 때 그 에러를 우아하게/깔끔하게 처리할 수 있습니다.

일반적으로 우리는 비동기 작업이 결과를 반환하는데 얼마의 시간이 걸리는지 보다는(작업 시간이 매우 오래 걸리지 않는 한) 그 결과를 사용할 수 있는지 여부에 더 관심이 있습니다. 물론 나마지 코드 블럭을 막지 않는다는 것에 있어서 매우 좋습니다.

우리가 Promise로 가장 많이 할 작업중 하나는 Promise를 반환하는 웹API를 사용하는 것입니다. 가상의 비디오 채팅 애플리케이션이 있다고 해봅시다. 애플리케이션에는 친구 목록이 있고 각 친구 목록 옆의 버튼을 클릭하면 해당 친구와 비디오 채팅을 시작합니다.

그 버튼은 사용자 컴퓨터의 카메라와 마이크를 사용하기 위해 `getUserMedia()` 를 호출합니다.

`getUserMedia()` 는 사용자가 이러한 장치를 사용할 수 있는 권한을 가지고 있는지 확인해야 하고, 어떤 마이크와 카메라를 사용할 것인지 (혹은 음성 통화인지, 아니면 다른 옵션들이 있는지)를 체크해야하기 때문에 모든 결정이 내려질 때 까지 다음 작업을 차단할 수 있습니다. 또한 카메라와 마이크가 작동하기 전 까지 다음 작업을 차단할수도 있습니다.

`getUserMedia()` 는 브라우저의 main thread에서 실행되므로 `getUserMedia()` 결과가 반환되기 전 까지 후행 작업이 모두 차단됩니다. 이러한 blocking은 우리가 바라는게 아닙니다. Promise가 없으면 이러한 결정이 내려지기 전 까지 브라우저의 모든 것을 사용할 수 없게됩니다. 따라서 사용자가 선택한 장치를 활성화하고 소스에서 선택된 스트림에 대해 [MediaStream_\(en-US\)](#) 직접 반환하는 대신 `getUserMedia()` 는 모든 장치가 사용 가능한 상태가 되면 [MediaStream_\(en-US\)](#)이 포함된 `promise` 를 반환합니다.

비디오 채팅 애플리케이션의 코드는 아래처럼 작성할 수 있습니다. :

```
function handleCallButton(evt) {
  setStatusMessage("Calling...");
  navigator.mediaDevices.getUserMedia({video: true, audio: true})
    .then(chatStream => {
      selfViewElem.srcObject = chatStream;
      chatStream.getTracks().forEach(track => myPeerConnection.addTrack(track, chatStream));
      setStatusMessage("Connected");
    })
    .catch(err => {
      setStatusMessage("Failed to connect");
    });
}
```

이 기능은 상태 메시지에 "Calling..."을 출력하는 `setStatusMessage()` 함수로 시작하며 통화가 시도되고 있음을 나타냅니다. 그런 다음 `getUserMedia()`을 호출하여 비디오와 오디오 트랙이 모두 있는 스트림 요청을 합니다. 그리고 스트림을 획득하면 카메라에서 나오는 스트림을 "self view,"로 표시하기 위해 `video`엘리먼트를 설정합니다. 그리고 각 스트림의 트랙을 가져가 다른 사용자와의 연결을 나타내는 [WebRTC RTCPeerConnection](#)에 추가합니다. 그리고 마지막으로 상태 메시지를 "Connected"로 업데이트 합니다.

`getUserMedia()` 가 실패하면, `catch` 블럭이 실행되며, `setStatusMessage()` 를 사용하여 상태창에 오류 메시지를 표시합니다.

여기서 중요한건 `getUserMedia()` 는 카메라 스트림이 아직 확보되지 않았음에도 거의 즉시 반환을 해줬다는 것 입니다. `handleCallButton()` 함수가 자신을 호출한 코드로 결과를 이미 반환을 했더라도 `getUserMedia()` 의 작업이 종료되면 프로그래머가 작성한 다음 핸들러를 호출할 것 입니다. 앱이 스트리밍을 했다고 가정하지 않는 한 계속 실행 될 수 있습니다.

- [The trouble with callbacks](#)

Promise가 왜 좋은지 이해하기 위해 구식 callbacks를 살펴보고 어떤게 문제인지 파악 해보겠습니다.

피자를 주문한다고 생각해봅시다. 피자를 잘 주문하려면 몇 가지 단계를 진행해야 합니다. 토픽 위에 도우를 올리고 치즈를 뿌리는 등 각 단계가 뒤죽박죽 이거나 혹은 도우를 반죽하고 있는데 그 위에 토마토소스를 바르는 등 이전 작업이 끝나지 않고 다음 작업을 진행하는 것은 말이 안 됩니다. :

1. 먼저 원하는 토픽을 고릅니다. 결정 장애가 심할 경우 토픽을 고르는데 오래 걸릴 수 있습니다. 또한 마음을 바꿔 피자 대신 카레를 먹으려고 가게를 나올 수 있습니다.
2. 그다음 피자를 주문합니다. 식당이 바빠서 피자가 나오는데 오래 걸릴 수 있고, 마침 재료가 다 떨어졌으면 피자를 만들 수 없다고 할 것 입니다.
3. 마지막으로 피자를 받아서 먹습니다. 그런데! 만약 지갑을 놓고 와서 돈을 내지 못한다면 피자를 먹지 못할 수 있습니다.

구식 [callbacks](#)을 사용하면 아래와 같은 모습의 코드가 나타날것 입니다. :

```
chooseToppings(function(toppings) {  
    placeOrder(toppings, function(order) {  
        collectOrder(order, function(pizza) {  
            eatPizza(pizza);  
        }, failureCallback);  
    }, failureCallback);  
}, failureCallback);
```

이런 코드는 읽기도 힘들 뿐 아니라 (종종 "콜백 지옥" 이라 불림), `failureCallback()` 을 여러 번 작성해야 하며 또 한 다른 문제점도 가지고 있습니다.

Improvements with promises

위의 상황에서 Promise를 사용하면 읽기, 작성, 실행 모두 다 쉬워집니다. callback 대신 비동기 Promise를 사용하면 아래처럼 작성할 수 있습니다. :

```
chooseToppings()
  .then(function(toppings) {
    return placeOrder(toppings);
  })
  .then(function(order) {
    return collectOrder(order);
  })
  .then(function(pizza) {
    eatPizza(pizza);
  })
  .catch(failureCallback);
```

보기예 훨씬 더 좋군요! — 이렇게 작성하면 앞으로 어떤 일이 일어날지 쉽게 예측 가능합니다. 그리고 단 한개의 `.catch()`을 사용하여 모든 에러를 처리합니다. 그리고 main thread를 차단하지 않습니다. (그래서 피자를 주문하고 기다리는 동안 하던 게임을 마저 할 수 있습니다.), 또한 각 함수가 실행되기 전 이전 작업이 끝날때까지 기다립니다. 이런식으로 여러 개의 비동기 작업을 연쇄적으로 처리할 수 있습니다. 왜냐하면 각 `.then()` 블럭은 자신이 속한 블럭의 작업이 끝났을 때의 결과를 새로운 Promise 반환해주기 때문입니다. 어때요, 참 쉽죠?

화살표 함수를 사용하면 코드를 조금 더 간단하게 고칠 수 있습니다. :

```
chooseToppings()
  .then(toppings =>
    placeOrder(toppings)
  )
  .then(order =>
    collectOrder(order)
  )
  .then(pizza =>
    eatPizza(pizza)
  )
  .catch(failureCallback);
```

혹은 아래처럼 표현할 수 있습니다. :

```
chooseToppings()
  .then(toppings => placeOrder(toppings))
  .then(order => collectOrder(order))
  .then(pizza => eatPizza(pizza))
  .catch(failureCallback);
```

화살표 함수의 `() => x` 표현은 `() => { return x; }`의 약식 표현이므로 잘 작동합니다.

함수는 arguments를 직접 전달 하므로 함수처럼 표현하지 않고 아래와 같이 작성할 수도 있습니다. :

```
chooseToppings().then(placeOrder).then(collectOrder).then(eatPizza).catch(failureCallback);
```

그런데 이렇게 작성하면 읽기가 쉽지 않습니다. 사용자의 코드가 지금의 예제보다 더 복잡하다면 위의 방법은 사용하기 힘듭니다.

Note: 다음 장에서 배울 `async/await` 문법으로 좀 더 간결화 할 수 있습니다.

Promise는 이벤트 리스너와 유사하지만 몇 가지 다른점이 있습니다. :

- Promise는 한번에 성공/실패 중 하나의 결과값을 가집니다. 하나의 요청에 두 번 성공하고나 실패할 수 없습니다.
또한 이미 성공한 작업이 다시 실패로 돌아갈 수 없고 실패한 작업이 성공으로 돌아갈 수 없습니다.
- If a promise has succeeded or failed and you later add a success/failure callback, the correct callback will be called, even though the event took place earlier.

- [Explaining basic promise syntax: A real example](#)

Explaining basic promise syntax: A real example

모던 웹 API는 잠재적으로 긴 작업을 수행하는 함수에 Promise를 사용하므로 Promise가 무엇인지 이해하는 것은 매우 중요합니다. 현대적인 웹 기술을 사용하려면 Promise를 사용해야 합니다. 챕터의 후반부에서 직접 Promise를 만들어보겠지만, 지금은 일단 웹 API에서 접할 수 있는 몇 가지 예제를 살펴보겠습니다.

첫 번째로, 웹에서 이미지를 가져오기 위하여 [fetch\(\)](#) 메서드를 사용할 때입니다. [blob\(\) \(en-US\)](#) 메서드는 fetch가 응답한 원시 body 컨텐츠를 [Blob](#) 오브젝트로 변환시켜주고 [](#) 엘리먼트에 표현합니다. 이 예제는 [first article of the series](#) 유사합니다. 다만 Promise를 사용하기 위해 약간의 변경을 하겠습니다.

Note: The following example will not work if you just run it directly from the file (i.e. via a `file://` URL). You need to run it through a [local testing server](#), or use an online solution such as [Glitch](#) or [GitHub pages](#).

1. 먼저 [simple HTML template](#) 와 fetch 할 이미지인 [sample image file](#) 을 다운받습니다.
2. HTML `<body>` 하단에 `<script>` 엘리먼트를 삽입합니다.
3. `<script>` 엘리먼트 안에 아래와 같이 코드를 작성합니다. :

```
let promise = fetch('coffee.jpg');
```



`fetch()` 메서드를 호출하여, 네트워크에서 fetch 할 이미지의 URL을 매개변수로 전달합니다. 두 번째 매개변수를 사용할 수 있지만, 지금은 우선 간단하게 하나의 매개변수만 사용하겠습니다. 코드를 더 살펴보면 `promise` 변수에 `fetch()` 작업으로 반환된 Promise 오브젝트를 저장하고 있습니다. 이전에 말했듯이, 지금 오브젝트는 성공도 아니고 실패도 아닌 중간 상태를 저장하고 있습니다. 공식적으로는 `pending` 상태라고 부릅니다.

4. 작업이 성공적으로 진행될 때를 대응하기 위해 (이번 예제에선 [Response \(en-US\)](#) 가 반환될 때 입니다.), 우리는 Promise 오브젝트의 `.then()` 메서드를 호출합니다. `.then()` 블럭 안의 callback은 **executor**라고 부릅니다) Promise가 성공적으로 완료되고 [Response \(en-US\)](#) 오브젝트를 반환할 때만 실행합니다. — 이렇게 성공한 Promise의 상태를 **fulfilled**라고 부릅니다. 그리고 반환된 [Response \(en-US\)](#) 오브젝트를 매개변수로 전달합니다.

Note: The way that a `.then()` block works is similar to when you add an event listener to an object using `AddEventListener()`. It doesn't run until an event occurs (when the promise fulfills). The most notable difference is that a `.then()` will only run once for each time it is used, whereas an event listener could be invoked multiple times.

그리고 즉시 `blob()` 메서드를 실행하여 Response Body가 완전히 다운로드 됐는지 확인합니다. 그리고 Response Body가 이용 가능할 때 추가 작업을 할 수 있는 `Blob` 오브젝트로 변환시킵니다. 해당 코드는 아래와 같이 작성할 수 있습니다. :

```
response => response.blob()
```

위의 코드는 아래의 코드를 축약한 형태입니다.

```
function(response) {
  return response.blob();
}
```

이제 추가 설명은 충분하므로, JavaScript의 첫 번째 줄 아래에 다음과 같은 라인을 추가하세요.

```
let promise2 = promise.then(response => response.blob());
```

5. 각 `.then()` 을 호출하면 새로운 Promise를 만드는데, 이는 매우 유용합니다. 왜냐하면 `blob()` 메서드도 Promise를 반환하기 때문에, 두 번째 Promise의 `.then()` 메서드를 호출함으로써 이행시 반환되는 Blob 오브젝트를 처리할 수 있습니다. 한 가지 메서드를 실행하여 결과를 반환하는 것보다 Blob에 좀 더 복잡한 일을 추가하고 싶습니다. 이럴때는 중괄호{}로 묶습니다. (그렇지 않으면 에러가 발생합니다.).

이어서 아래와 같은 코드를 추가합니다.:

```
let promise3 = promise2.then(myBlob => {  
})
```

6. 이제 executor 함수를 아래와 같이 채워넣습니다. 중괄호 안에 작성하면 됩니다. :

```
let objectURL = URL.createObjectURL(myBlob);  
let image = document.createElement('img');  
image.src = objectURL;  
document.body.appendChild(image);
```

여기서 우리는 두 번째 Promise가 fulfills일 때 반환된 Blob을 매개변수로 전달받는 `URL.createObjectURL()` 메서드를 실행하고 있습니다. 이렇게 하면 오브젝트가 가지고 있는 URL이 반환됩니다. 그 다음 `` 엘리먼트를 만들고, 반환된 URL을 `src` 속성에 지정하여 DOM에 추가합니다. 이렇게 하면 페이지에 그림이 표시됩니다.

If you save the HTML file you've just created and load it in your browser, you'll see that the image is displayed in the page as expected. Good work!

Note: You will probably notice that these examples are somewhat contrived. You could just do away with the whole `fetch()` and `blob()` chain, and just create an `` element and set its `src` attribute value to the URL of the image file, `coffee.jpg`. We did, however, pick this example because it demonstrates promises in a nice simple fashion, rather than for its real-world appropriateness.

Responding to failure

현재 에러가 발생했을 때 어떻게 처리를 해야할지 작성된 코드가 없기 때문에 코드를 조금만 더 추가하여 좀 더 완벽하게 작성해봅시다. (Promise에서 에러가 발생한 상태를 **rejects**라 부릅니다). 이전에 봤던대로 `.catch()` 블록을 추가하여 오류를 핸들링 할 수 있습니다. 아래처럼 말이죠 :

```
let errorCase = promise3.catch(e => {
  console.log('There has been a problem with your fetch operation: ' + e.message);
});
```

에러 메시지를 확인하고 싶으면 잘못된 url을 지정해보세요, 개발자 도구 콘솔에서 에러를 확인할 수 있을것입니다.

물론 `.catch()` 블록 없이 코드를 작동시킬 수 있습니다. 하지만 좀 더 깊게 생각해보면 `.catch()` 블록이 없으면 어떤 에러가 발생했는지, 어떻게 해결해야 하는지 디버깅이 어렵습니다. 실제 앱에서 `.catch()` 을 사용하여 이미지 가져오기를 다시 실행하거나, 기본 이미지를 표시하는 등 작업을 지시할 수 있습니다.

Chaining the blocks together

위에서 사용한 코드는 작업이 어떻게 처리되는지 명확하게 보여주기 위해 매우 길게 코드를 작성했습니다. 이전 글에서 봤듯이, `.then()` 블록을 사용하여 연쇄 작업을 진행할 수 있습니다. (또한 `.catch()` 블록을 사용하여 에러 처리도 했지요). 앞선 예제의 코드는 아래와 같이 작성할 수도 있습니다. (see also [simple-fetch-chained.html](#) ↗ on GitHub):

```
fetch('coffee.jpg')
  .then(response => response.blob())
  .then(myBlob => {
    let objectURL = URL.createObjectURL(myBlob);
    let image = document.createElement('img');
    image.src = objectURL;
    document.body.appendChild(image);
  })
  .catch(e => {
    console.log('There has been a problem with your fetch operation: ' + e.message);
});
```

fulfilled promise 결과에 의해 반환된 값이 다음 `.then()` 블록의 executor 함수가 가진 파라미터로 전달 된다는 것을 꼭 기억하세요.

Note: `.then()` / `.catch()` blocks in promises are basically the async equivalent of a `try...catch` block in sync code. Bear in mind that synchronous `try...catch` won't work in async code.

- [Promise terminology recap](#)

Promise terminology recap

위의 섹션에서 다룬 내용은 정말 많습니다. 매우 중요한 내용을 다뤘으므로 개념을 명확히 이해하기 위해 몇번이고 다시 읽어보는게 좋습니다.

1. Promise가 생성되면 그 상태는 성공도 실패도 아닌 **pending**상태라고 부릅니다..
2. Promise결과가 반환되면 결과에 상관 없이 **resolved**상태라고 부릅니다..
 1. 성공적으로 처리된 Promise는 **fulfilled**상태이다. 이 상태가 되면 Promise 체인의 다음 `.then()` 블럭에서 사용할 수 있는 값을 반환합니다.. 그리고 `.then()` 블럭 내부의 executor 함수에 Promise에서 반환된 값이 파라미터로 전달됩니다..
 2. 실패한 Promise는 **rejected**상태이다. 이때 어떤 이유(**reason**) 때문에 Promise가 rejected 됐는지를 나타내는 에러 메시지를 포함한 결과가 반환됩니다. Promise 체이닝의 제일 마지막 `.catch()`에서 상세한 에러 메시지를 확인할 수 있습니다.

- [Running code in response to multiple promises fulfilling](#)

Running code in response to multiple promises fulfilling

위의 예제에서 Promise 사용의 기초를 확인했습니다. 이제 고급 기능들을 한번 보겠습니다. 제일 먼저 확인해볼 예제는 다음과 같습니다. 연쇄적으로 일어나는 작업은 좋습니다. 그런데 모든 Promise가 fulfilled일 경우 코드를 실행하고 싶은 경우가 있을 것입니다.

해당 기능을 `promise.all()` 이라는 스테틱 메서드를 사용하여 만들 수 있습니다. 이 메서드는 Promise의 배열을 매개변수로 삼고, 배열의 모든 Promise가 fulfil될 때만 새로운 fulfil Promise 오브젝트를 반환합니다. 아래처럼 말이죠 :

```
Promise.all([a, b, c]).then(values => {
  ...
});
```



배열의 모든 Promise가 fulfil 이면, `.then()` 블럭의 executor 함수로의 매개변수로 Promise 결과의 배열을 전달합니다. `Promise.all()` 의 Promise의 배열 중 하나라도 reject라면, 전체 결과가 reject가 됩니다.

이 방법은 매우 유용합니다. 웹 UI의 컨텐츠를 동적인 방법으로 채운다고 생각 해보겠습니다. 대부분 경우에 틈성등성 내용을 채우기보단, 원진한 내용을 채울것입니다.

다른 예제를 만들어서 실행해 보겠습니다.

1. 이미 만들어진 [page template](#) 을 다운받으세요 그리고 </body> 뒤에 <script> 엘리먼트를 만들어주세요.
2. 이미지 그리고 텍스트 파일([coffee.jpg](#), [tea.jpg](#), and [description.txt](#))을 다운받고 [page template](#) 와 같은 경로에 저장해주세요.
3. 먼저 Promise를 반환하는 몇 가지 함수를 만들어 `Promise.all()`로 결과를 반환합니다. 세 개의 `fetch()` 작업이 끝나고 다음 요청을 진행하고 싶다면 아래 코드처럼 `Promise.all()` 블럭을 작성합니다.

```
let a = fetch(url1);
let b = fetch(url2);
let c = fetch(url3);

Promise.all([a, b, c]).then(values => {
  ...
});
```

Promise가 fulfilled가 됐을 때, fulfillment handler 핸들러로 전달된 `values` 매개변수에는 각 `fetch()` 의 결과로 발생한 세 개의 `Response` 오브젝트가 들어있습니다.

하지만 우리는 단순히 결과만 넘겨주고 싶지 않습니다. 우리는 `fetch()` 언제 끝나는지 보다 불러온 데이터에 더 관심이 있습니다. 그말은 브라우저에 표현할 수 있는 Blob과 텍스트 문자열이 불러와 졌을 때 `Promise.all()` 블럭을 실행하고 싶다는 것입니다. <script> 엘리먼트에 아래와 같이 작성합니다. :

```
function fetchAndDecode(url, type) {
  return fetch(url).then(response => {
    if (type === 'blob') {
      return response.blob();
    } else if (type === 'text') {
      return response.text();
    }
  })
  .catch(e => {
    console.log('There has been a problem with your fetch operation: ' + e);
  });
}
```

살짝 복잡해 보이므로 하나하나 살펴봅시다. :

1. 먼저 `fetchAndDecode()` 함수를 정의했고 함수의 매개변수로 컨텐츠의 URL과 가져오는 리소스의 타입을 지정합니다.
2. 함수 내부에 첫 번째 예에서 본 것과 유사한 구조를 가진 코드가 있습니다. — `fetch()` 함수를 호출하여 전달받은 URL에서 리소스를 받아오도록 했습니다. 그리고 다음 Promise를 연쇄적으로 호출하여 디코딩된 (혹은 "읽은") Response Body를 반환하게 합니다. 이전 예에선 Blob만을 가져오기 때문에 `blob()` 메서드만 썼습니다.
3. 여기에선 이전과 다른 두 가지가 있습니다. :
 - 먼저 두 번째 Promise에서는 불러올 리소스의 `type` 이 무엇인지에 따라 반환받는 데이터가 달립니다. `executor` 함수 내부에, 간단한 `if ... else if` 구문을 사용하여 어떤 종류의 파일을 디코딩해야 하는지에 따라 다른 Promise를 반환하게 했습니다. (이 경우 `blob()`이나 `text()` 밖에 없지만, 이것을 잘 활용하여 다른 코드에 확장하여 적용할 수 있습니다.).
 - 두 번째로, `fetch()` 호출 앞에 `return` 키워드를 추가했습니다. 이렇게 하면 Promise 체이닝의 마지막 결과값을 함수의 결과로 반환해 줄 수 있습니다. (이 경우 `blob()` 혹은 `text()` 메서드에 의해 반환된 Promise입니다.) 사실상 `fetch()` 앞의 `return` 구문은 체이닝 결과를 다시 상단으로 전달하는 행위입니다.
4. 블럭의 마지막에는 `.catch()` 블럭을 추가하여 작업중 발생한 에러를 `.all()`의 배열로 전달합니다. 아무 Promise에서 `reject`가 발생하면, `catch` 블럭은 어떤 Promise에서 에러가 발생했는지 알려줄 것입니다. `.all()` (아래쪽에 있는) 블럭의 리소스에 문제가 있지 않는 이상 항상 `fulfill`일것입니다. `.all` 블럭의 마지막 체이닝에 `.catch()` 블럭을 추가하여 `reject`됐을때 확인을 할 수 있습니다.

함수의 `body` 안에 있는 코드는 비동기적이고 Promise 기반이므로, 전체 함수는 Promise로 작동합니다. — 편리하죠?

4. 다음으로 `fetchAndDecode()` 함수를 세 번 호출하여 이미지와 텍스트를 가져오고 디코딩 하는 과정을 시작합니다.

그리고 반환된 `Promise`를 각각의 변수에 저장합니다. 이전 코드에 이어서 아래 코드를 추가하세요. :

```
let coffee = fetchAndDecode('coffee.jpg', 'blob');
let tea = fetchAndDecode('tea.jpg', 'blob');
let description = fetchAndDecode('description.txt', 'text');
```

5. 다음으로 위의 세 가지 코드가 모두 `fulfilled`가 됐을 때 원하는 코드를 실행하기 위해 `Promise.all()` 블럭을 만듭니다. 우선, `.then()` call 안에 비어있는 `executor`를 추가하세요 :

```
Promise.all([coffee, tea, description]).then(values => {
}) ;
```

위에서 `Promise`를 포함하는 배열을 매개 변수로 사용하는 것을 확인할 수 있습니다. `executor`는 세 가지 `Promise`가 `resolve`될 때만 실행될 것 입니다. 그리고 `executor`가 실행될 때 개별적인 `Promise`의 결과를 포함하는 `[coffee-results, tea-results, description-results]` 배열을 매개 변수로 전달받을 것 입니다. (여기선 디코딩된 Response Body 입니다.).

6. 마지막으로 executor 함수를 작성합니다. 예제에선 반환된 결과를 별도의 변수로 저장하기 위해 간단한 동기화 코드를 사용합니다. (Blob에서 오브젝트 URLs 생성), 그리고 페이지에 텍스트와 이미지를 표시합니다.

```
console.log(values);
// Store each value returned from the promises in separate variables; c1
let objectURL1 = URL.createObjectURL(values[0]);
let objectURL2 = URL.createObjectURL(values[1]);
let descText = values[2];

// Display the images in <img> elements
let image1 = document.createElement('img');
let image2 = document.createElement('img');
image1.src = objectURL1;
image2.src = objectURL2;
document.body.appendChild(image1);
document.body.appendChild(image2);

// Display the text in a paragraph
let para = document.createElement('p');
para.textContent = descText;
document.body.appendChild(para);
```

7. 코드를 저장하고 창을 새로고치면 보기엔 좋지 않지만, UI 구성 요소가 모두 표시된 것을 볼 수 있습니다.

여기서 제공한 코드는 매우 기초적이지만, 내용을 전달하기에는 아주 좋습니다..

- [Running some final code after a promise fulfills/rejects](#)

Running some final code after a promise fulfills/rejects

Promise의 결과가 fulfilled인지 rejected인지 관계 없이 Promise가 완료된 후 최종 코드 블럭을 실행하려는 경우가 있을 것입니다. 이전에는 아래 예시처럼 .then() 블럭과 .catch() 블럭의 callbacks에 아래와 같이 runFinalCode()를 넣었습니다. :

```
myPromise
  .then(response => {
    doSomething(response);
    runFinalCode();
  })
  .catch(e => {
    returnError(e);
    runFinalCode();
});
});
```

보다 최근의 현대 브라우저에서는 [.finally\(\)](#) 메서드를 사용할 수 있습니다. 이 메서드를 Promise 체이닝의 끝에 배치하여 코드 반복을 줄이고 좀 더 우아하게 일을 처리할 수 있습니다. 아래와 같이 마지막 블럭에 적용할 수 있습니다. :

```
myPromise
  .then(response => {
    doSomething(response);
  })
  .catch(e => {
    returnError(e);
  })
  .finally(() => {
    runFinalCode();
});
});
```

실제 예시는 [promise-finally.html demo](#) 에 나와있습니다. (see the [source code](#) also). 이 예시는 위에서 만들 어본 Promise.all() 데모와 똑같이 작동합니다. 다만 이번에는 fetchAndDecode() 함수에 다음 연쇄 작업으로 finally() 를 호출합니다.:

```
function fetchAndDecode(url, type) {
  return fetch(url).then(response => {
    if(type === 'blob') {
      return response.blob();
    } else if(type === 'text') {
      return response.text();
    }
  })
  .catch(e => {
    console.log(`There has been a problem with your fetch operation for resource "${url}": ` + e.message);
  })
  .finally(() => {
    console.log(`fetch attempt for "${url}" finished.`);
  });
}
```

이 로그는 각 fetch시도가 완료되면 콘솔에 메시지를 출력하여 사용자에게 알려줍니다.

Note: then()/catch()/finally() is the async equivalent to try/catch/finally in sync code.

- [Building your own custom promises](#)

Building your own custom promises

여기까지 오느라 수고하셨습니다. 여기까지 오면서 우리는 Promise를 직접 만들어봤습니다. 여러 개의 Promise를 `.then()` 을 사용하여 체이닝 하거나 사용자 정의함수를 조합하여, 비동기 Promise기반 함수를 만들었습니다. 이전에 만든 `fetchAndDecode()` 함수가 이를 잘 보여주고있죠.

다양한 Promise 기반 API를 결합하여 사용자 정의 함수를 만드는 것은, Promise와 함께 원하는 기능을 만드는 가장 일반적인 방법이며, 대부분 모던 API는 이와 같은 원리를 기반으로 만들어지고 있습니다. 그리고 또 다른 방법이 있습니다.

Using the `Promise()` constructor

`Promise()` constructor를 사용하여 사용자 정의 Promise를 만들 수 있습니다. 주로 Promise기반이 아닌 구식 비동기 API코드를 Promise기반 코드로 만들고 싶을 경우 사용합니다. 이 방법은 구식 프로젝트 코드, 라이브러리, 혹은 프레임워크를 지금의 Promise 코드와 함께 사용할 때 유용합니다.

간단한 예를 들어 살펴보겠습니다. — 여기 Promise와 함께 사용되는 `setTimeout()` 호출이 있습니다. — 이 함수는 2초 후에 "Success!"라는 문자열과 함께 resolve됩니다. (통과된 `resolve()` 호출에 의해);

```
let timeoutPromise = new Promise((resolve, reject) => {
  setTimeout(function() {
    resolve('Success!');
  }, 2000);
});
```

`resolve()` 와 `reject()` 는 Promise의 `fulfill` / `reject`일때의 일을 수행하기 위해 호출한 함수입니다. 이번의 경우 Promise는 "Success!"문자와 함께 `fulfill` 됐습니다.

따라서 이 Promise를 호출할 때, 그 끝에 `.then()` 블럭을 사용하면 "Success!" 문자열이 전달될 것입니다. 아래 코드는 간단한 alert메시지를 출력하는 방법입니다. :

```
timeoutPromise
  .then((message) => {
    alert(message);
  })
```

혹은 아래처럼 쓸 수 있죠

```
timeoutPromise.then(alert);
```

Try [running this live](#)  to see the result (also see the [source code](#) ).

위의 예시는 유연하게 적용된 예시가 아닙니다. — Promise는 항산 하나의 문자열로만 fulfil됩니다. 그리고 `reject()` 조건도 정의되어있지 않습니다. (사실, `setTimeout()` 은 실패 조건이 필요없습니다, 그러니 이 예제에서는 없어도 됩니다.).

Note: Why `resolve()`, and not `fulfill()`? The answer we'll give you, for now, is *it's complicated*.

Rejecting a custom promise

`reject()` 메서드를 사용하여 Promise가 reject상태일 때 전달할 값을 지정할 수 있습니다. — `resolve()` 와 똑같습니다. 여기엔 하나의 값만 들어갈 수 있습니다. Promise가 reject 되면 에러는 `.catch()` 블럭으로 전달됩니다.

이전 예시를 좀 더 확장하여 `reject()` 을 추가하고, Promise가 fulfil일 때 다른 메시지도 전달할 수 있게 만들어봅시다.

이전 예시 [previous example](#) ↗를 복사한 후 이미 있는 `timeoutPromise()` 함수를 아래처럼 정의해주세요. :

```
function timeoutPromise(message, interval) {
  return new Promise((resolve, reject) => {
    if (message === '' || typeof message !== 'string') {
      reject('Message is empty or not a string');
    } else if (interval < 0 || typeof interval !== 'number') {
      reject('Interval is negative or not a number');
    } else {
      setTimeout(function(){
        resolve(message);
      }, interval);
    }
  });
}
```

함수를 살펴보면 두 가지 매개변수가 있습니다. — 출력할 메시지와(`message`) 메시지를 출력할 때 까지 기다릴 시간(`interval`)입니다. 맨 위에 `Promise` 오브젝트를 반환하도록 되어있습니다. 따라서 함수를 실행하면 우리가 사용하고 싶은 `Promise`가 반환될 것입니다..

Promise constructor 안에는 몇가지 사항을 확인하기 위해 `if ... else` 구문이 있습니다. :

1. 첫번째로 메시지의 유효성을 검사합니다. 메시지가 비어있거나 문자가 아닌 경우, 에러 메시지와 함께 Promise를 `reject`합니다.
2. 그 다음으로 `interval`의 유효성을 검사합니다. 숫자가 아니거나 음수일 경우, 에러 메시지와 함께 Promise를 `reject`합니다.
3. 마지막은 항목은, 두 매개변수를 확인하여 유효할 경우 `setTimeout()` 함수에 지정된 `interval`에 맞춰 Promise를 `resolve`합니다.

`timeoutPromise()` 함수는 `Promise`를 반환하므로, `.then()`, `.catch()`, 기타등등을 사용해 Promise 체이닝을 만들 수 있습니다. 아래와 같이 작성해봅시다. — 기존에 있는 `timeoutPromise` 를 삭제하고 아래처럼 바꿔주세요. :

```
timeoutPromise('Hello there!', 1000)
.then(message => {
  alert(message);
})
.catch(e => {
  console.log('Error: ' + e);
});
```

이 코드를 저장하고 브라우저를 새로 고침하면 1초 후에 'Hello there!' `alert`가 출력될 것입니다. 이제 메시지 내용을 비우거나 `interval`을 음수로 지정해보세요 그렇게 하면 Promise가 `reject`되며 에러 메시지를 콘솔에 출력해 줄 것입니다. 또한 `resolved` 메시지를 다르게 만들어 줄 수도 있습니다.

Note: You can find our version of this example on GitHub as [custom-promise2.html](#) (see also the [source code](#)).

A more real-world example

위의 예제는 개념을 이해하기 쉽게 단순하게 만들었지만, 실제로 그다지 비동기적이지는 않습니다. 억지로 비동기적 작업을 구현하기 위해 `setTimeout()`을 사용하여 함수를 만들었지만 사용자 정의 Promise를 만들고 에러를 다루기엔 충분한 예제였습니다.

좀 더 공부해볼 추가내용을 소개해주고 싶습니다. 바로 [Jake Archibald's idb library](#)입니다 이 라이브러리는 `Promise()` constructor의 비동기작업 응용을 보여주는 유용한 라이브러리입니다. 클라이언트측에서 데이터를 저장하고 검색하기 위한 구식 callback 기반 API로 Promise와 함께 사용하는 [IndexedDB API](#)입니다.
[main library file](#)을 살펴보면 우리가 지금까지 다뤄본것과 같은 종류의 테크닉을 볼 수 있습니다. 아래 코드 블록은 basic request model이 Promise를 사용하게끔 변환해 주는 IndexedDB 메서드입니다. :

```
function promisifyRequest(request) {
    return new Promise(function(resolve, reject) {
        request.onsuccess = function() {
            resolve(request.result);
        };

        request.onerror = function() {
            reject(request.error);
        };
    });
}
```

우리가 했던것 처럼 적절한 타이밍에 Promise를 fulfil하고 reject하는 이벤트 핸들러를 두 개 추가했습니다. :

- `request` 의 `success event`가 실행될 때, `onsuccess` 핸들러에 의해 fulfill된 Promise의 `request result`를 반환한다.
- 반면 `request`'s `error event`가 실행되면 `onerror` 핸들러에 의해 reject된 Promise의 `request error`를 반환한다.

Article 2. JavaScript Promises for Dummies (모형을 위한 프로미스)

출처 : <https://www.digitalocean.com/community/tutorials/understanding-javascript-promises>

- Understanding Promises
 - Creating a Promise
 - Consuming Promises
 - Chaining Promises
 - Promises are Asynchronous
 - Promises in ES5, ES6(2015), ES7(Next)
 - Promises and When to Use Them
 - Observables
 - Conclusion
-
- Understanding Promises

프로미스를 요약하자면:

"너가 아이라고 상상하고 너의 엄마는 다음주에 너에게 핸드폰을 사준다고 약속했다라는 가정을 해보자."

다음주에 핸드폰을 가질 수 있을지 너는 모른다. 너의 엄마는 핸드폰을 사줄 수도 있고 안 사줄 수도 있다.

이것이 프로미스이다. 프로미스는 다음 3가지 상태를 가진다.

1. Pending: You *don't know* if you will get that phone
2. Fulfilled: Mom is *happy*, she buys you a brand new phone
3. Rejected: Mom is *unhappy*, she doesn't buy you a phone

- **Creating a Promise**

```
// ES5: Part 1

var isMomHappy = false;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // fulfilled
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }
    }
);
```

위 상황을 코드로 재현해보았다.

```
// promise syntax look like this
new Promise(function (resolve, reject) { ... } );
```

- **Consuming Promises**

이제 프로미스를 소비해보자!!

```
// ES5: Part 2

var willIGetNewPhone = ... // continue from part 1
```

```

// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
            // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {
            // oops, mom didn't buy it
            console.log(error.message);
            // output: 'mom is not happy'
        });
};

askMom();

```

- **Chaining Promises**

프로미스는 체이닝이 가능하다.

예를 들어, 아이가 핸드폰을 사게되면 친구에게 자랑을 하기로 했다고 가정해보자. 그에 대한 메서드인 showOff 프로미스를 추가할 수 있다.

```

// ES5

// 2nd promise
var showOff = function (phone) {
    return new Promise(
        function (resolve, reject) {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};

// 두개 모두 같은 함수이다.

// shorten it

// 2nd promise
var showOff = function (phone) {
    var message = 'Hey friend, I have a new ' +
        phone.color + ' ' + phone.brand + ' phone';

```

```
        return Promise.resolve(message);
};
```

```
// call our promise
var askMom = function () {
    willIGetNewPhone
    .then(showOff) // chain it here
    .then(function (fulfilled) {
        console.log(fulfilled);
        // output: 'Hey friend, I have a new black Samsung phone.'
    })
    .catch(function (error) {
        // oops, mom don't buy it
        console.log(error.message);
        // output: 'mom is not happy'
    });
};
```

- **Promises are Asynchronous**

프로미스는 비동기적이다.

```
// call our promise
var askMom = function () {
    console.log('before asking Mom'); // log before
    willIGetNewPhone
    .then(showOff)
    .then(function (fulfilled) {
        console.log(fulfilled);
    })
    .catch(function (error) {
        console.log(error.message);
    });
    console.log('after asking mom'); // log after
}
```

위 예시를 실행시키면 우리의 기대와는 다르게 다음과 같은 결과가 나온다.

1. before asking Mom
2. after asking mom
3. Hey friend, I have a new black Samsung phone.

우리는 프로미스 블록이 실행되는 동안 기다리지 않는다. 이것을 비동기적 연산이라고 한다.
코드는 blocking, waiting 없이 실행될 것이다.

```
// ES5: Full example

var isMomHappy = true;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // fulfilled
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }
    }
);

// 2nd promise
var showOff = function (phone) {
    var message = 'Hey friend, I have a new ' +
        phone.color + ' ' + phone.brand + ' phone';

    return Promise.resolve(message);
};

// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(showOff) // chain it here
        .then(function (fulfilled) {
            console.log(fulfilled);
        })
};
```

```

        // output: 'Hey friend, I have a new black Samsung phone.'
    })
    .catch(function (error) {
        // oops, mom don't buy it
        console.log(error.message);
        // output: 'mom is not happy'
    });
};

askMom();

```

- **Promises in ES5, ES6(2015), ES7(Next)**

```

//_ ES6: Full example_

const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => { // fat arrow
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }
    }
);

// 2nd promise
const showOff = function (phone) {
    const message = 'Hey friend, I have a new ' +
        phone.color + ' ' + phone.brand + ' phone';
    return Promise.resolve(message);
};

// call our promise
const askMom = function () {
    willIGetNewPhone
        .then(showOff)
        .then(fulfilled => console.log(fulfilled)) // fat arrow
        .catch(error => console.log(error.message)); // fat arrow
}

```

```
};  
askMom();
```

ES7 (async await) → .then과 .catch 없이 더 쉽게 이해하도록 하는 문법이다.

```
// ES7: Full example  
const isMomHappy = true;  
  
// Promise  
const willIGetNewPhone = new Promise(  
    (resolve, reject) => {  
        if (isMomHappy) {  
            const phone = {  
                brand: 'Samsung',  
                color: 'black'  
            };  
            resolve(phone);  
        } else {  
            const reason = new Error('mom is not happy');  
            reject(reason);  
        }  
    }  
);  
  
// 2nd promise  
async function showOff(phone) {  
    return new Promise(  
        (resolve, reject) => {  
            var message = 'Hey friend, I have a new ' +  
                phone.color + ' ' + phone.brand + ' phone';  
  
            resolve(message);  
        }  
    );  
};  
  
// call our promise in ES7 async await style  
async function askMom() {  
    try {  
        console.log('before asking Mom');  
  
        let phone = await willIGetNewPhone;  
        let message = await showOff(phone);  
  
        console.log(message);  
        console.log('after asking mom');  
    }  
    catch (error) {  
        console.log(error.message);  
    }  
}
```

```
        }
    }

// async await it here too
(async () => {
    await askMom();
})();
```

- **Promises and When to Use Them**

우리는 왜 프로미스를 필요로 할까? 프로미스가 없었을 땐 어떻게 했을까?

Normal Function VS Async Function

Let's take a look at these two examples. Both examples perform the addition of two numbers: one adds using normal functions, and the other adds remotely.

Normal Function to Add Two Numbers

```
// add two numbers normally

function add (num1, num2) {
    return num1 + num2;
}

const result = add(1, 2); // you get result = 3 immediately
```

Async Function to Add Two numbers

```
// add two numbers remotely

// get the result by calling an API
const result = getAddResultFromServer('http://www.example.com?num1=1&num2=2');
// you get result = "undefined"
```

If you add the numbers with the normal function, you get the result immediately. However, when you issue a remote call to get the result, you need to wait, and you can't get the result immediately.

You don't know if you will get the result because the server might be down, slow in response, etc. You don't want your entire process to be blocked while waiting for the result.

Calling APIs, downloading files, and reading files are among some of the usual async operations that you'll perform.

You do not need to use promises for an asynchronous call. Prior to promises, we used callbacks. Callbacks are a function you call when you get the return result. Let's modify the previous example to accept a callback.

```
// add two numbers remotely
// get the result by calling an API

function addAsync (num1, num2, callback) {
    // use the famous jQuery getJSON callback API
    return $.getJSON('http://www.example.com', {
        num1: num1,
        num2: num2
    }, callback);
}

addAsync(1, 2, success => {
    // callback
    const result = success; // you get result = 3 here
});
```

Subsequent Async Action (순차적인 Async Action)

숫자를 하나씩 더하는 것 대신에, 우리는 3번 더하는 것을 원한다. 일반적인 함수에서는 아래와 같이 할 수 있다.

```
// add two numbers normally

let resultA, resultB, resultC;

function add (num1, num2) {
    return num1 + num2;
}
```

```
resultA = add(1, 2); // you get resultA = 3 immediately
resultB = add(resultA, 3); // you get resultB = 6 immediately
resultC = add(resultB, 4); // you get resultC = 10 immediately

console.log('total' + resultC);
console.log(resultA, resultB, resultC);
```

이를 콜백으로 바꾸면 아래와 같다.

```
// add two numbers remotely
// get the result by calling an API

let resultA, resultB, resultC;

function addAsync (num1, num2, callback) {
    // use the famous jQuery getJSON callback API
    // https://api.jquery.com/jQuerygetJSON/
    return $.getJSON('http://www.example.com', {
        num1: num1,
        num2: num2
    }, callback);
}

addAsync(1, 2, success => {
    // callback 1
    resultA = success; // you get result = 3 here

    addAsync(resultA, 3, success => {
        // callback 2
        resultB = success; // you get result = 6 here

        addAsync(resultB, 4, success => {
            // callback 3
            resultC = success; // you get result = 10 here

            console.log('total' + resultC);
            console.log(resultA, resultB, resultC);
        });
    });
});
```

바로 콜백 지옥이 되는 것이다.

Avoiding Deeply Nested Callbacks

Promise 를 사용하여 콜백 지옥을 벗어날 수 있다. ES7 을 통해 더 간단히 바꿀 수 있다.

```
// add two numbers remotely using observable

let resultA, resultB, resultC;

function addAsync(num1, num2) {
    // use ES6 fetch API, which return a promise
    // What is .json()? https://developer.mozilla.org/en-US/docs/Web/API/Body/json
    return fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
        .then(x => x.json());
}

addAsync(1, 2)
    .then(success => {
        resultA = success;
        return resultA;
    })
    .then(success => addAsync(success, 3))
    .then(success => {
        resultB = success;
        return resultB;
    })
    .then(success => addAsync(success, 4))
    .then(success => {
        resultC = success;
        return resultC;
    })
    .then(success => {
        console.log('total: ' + success)
        console.log(resultA, resultB, resultC)
    });
}
```

- **Observables**

Observables

Before you settle down with promises, there is something that has come about to help you deal with async data called `Observables`.

Let's look at the same demo written with Observables. In this example, we will use `RxJS` for the observables.

```
let Observable = Rx.Observable;
let resultA, resultB, resultC;

function addAsync(num1, num2) {
    // use ES6 fetch API, which return a promise
    const promise = fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
        .then(x => x.json());

    return Observable.fromPromise(promise);
}

addAsync(1,2)
    .do(x => resultA = x)
    .flatMap(x => addAsync(x, 3))
    .do(x => resultB = x)
    .flatMap(x => addAsync(x, 4))
    .do(x => resultC = x)
    .subscribe(x => {
        console.log('total: ' + x)
        console.log(resultA, resultB, resultC)
    });
}
```

Observables can do more interesting things. For example, `delay` add function by 3 seconds with just one line of code or retry so you can retry a call a certain number of times.

```
...
addAsync(1,2)
    .delay(3000) // delay 3 seconds
    .do(x => resultA = x)
    ...
}
```

You may read one of my RxJs posts [here](#).

- Conclusion

너 스스로 콜백과 프로미스에 익숙해지는 것이 중요하다. 그것들을 이해하고 사용해봐라. Observables 에 대해 아직 걱정하지 마라. 이 3가지가 너의 개발에 있어서 거의 대부분을 차지할 것이다.