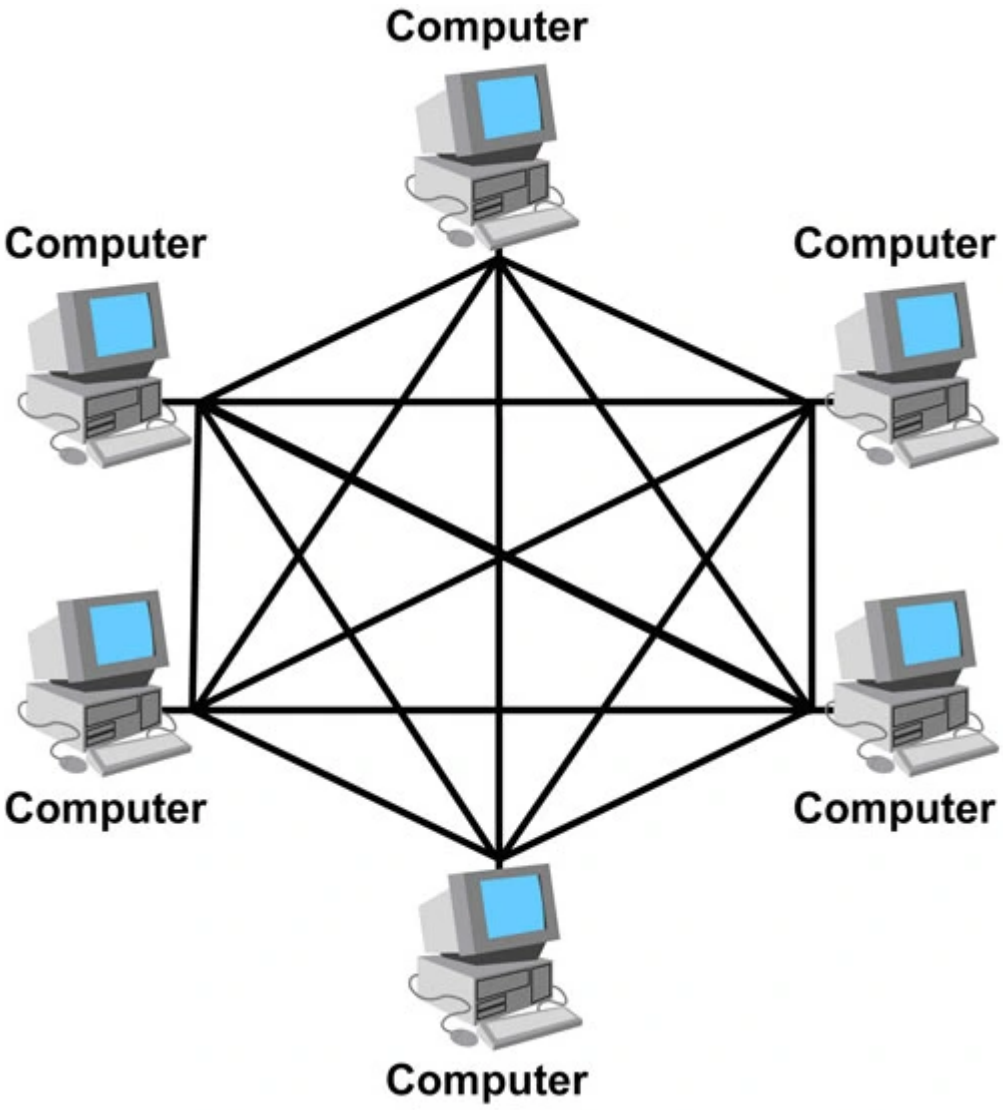
221221_2반_실습

Protocol



Protocol 개요

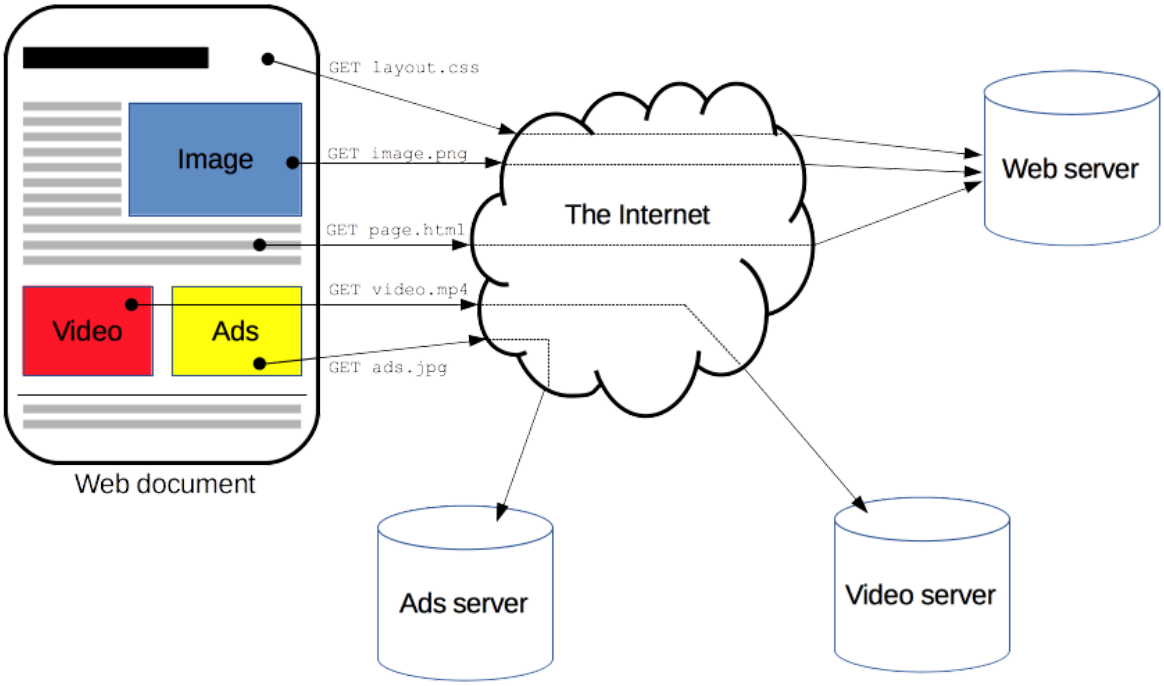
Protocol 이란?

- 컴퓨터들 간의 원활한 통신을 위해 지키기로 약속한 **규약**
- **프로토콜**은 컴퓨터 내부에서, 또는 컴퓨터 사이에서 데이터의 교환 방식을 정의하는 규칙 체계

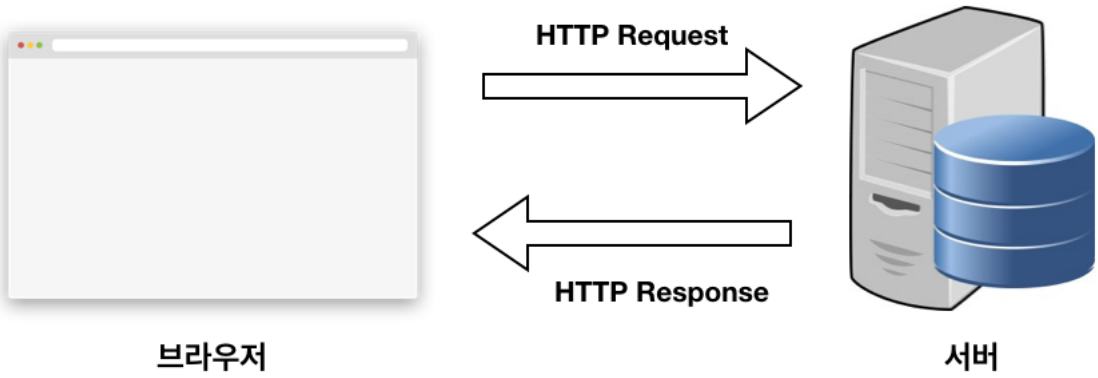
Protocols

- IP
- TCP
- UDP
- HTTP
- SNMP
- FTP
- SMTP
- SSH
- Telnet

HTTP(HyperText Transfer Protocol)



HTTP 개요

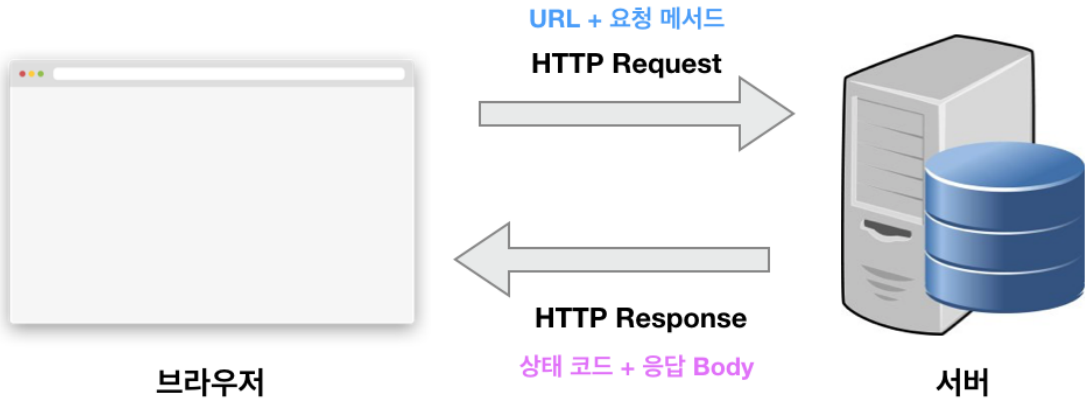


- `HTTP`는 `HTML` **문서와 같은 리소스들을 가져올 수 있도록 해주는** 프로토콜
- HTTP는 웹에서 이루어지는 모든 데이터 교환의 기초이며, **클라이언트-서버 프로토콜**이기도 합니다.

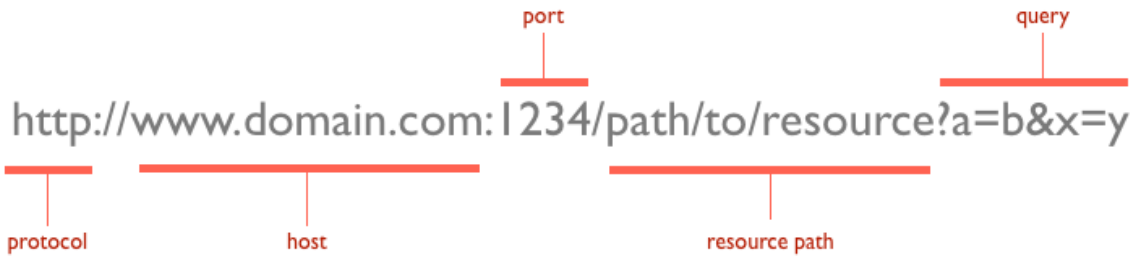
> 💡 클라이언트-서버 프로토콜이란 (보통 웹브라우저인) 수신자 측에 의해 요청이 초기화되는 프로토콜을 의미

- 하나의 완전한 문서는 텍스트, 레이아웃 설명, 이미지, 비디오, 스크립트 등 불러온 (fetched) 하위 문서들로 재구성됩니다.
- 애플리케이션 레벨의 프로토콜로 TCP/IP 위에서 작동
- Port : 80

HTTP Request & Response



1. 요청과 응답
    - 클라이언트와 서버들은 (데이터 스트림과 대조적으로) 개별적인 메시지 교환에 의해 통신합니다.
    - 보통 브라우저인 클라이언트에 의해 전송되는 메시지를 요청(`requests`)이라고 부르며, 그에 대해 서버에서 응답으로 전송되는 메시지를 응답(`responses`)이라고 부릅니다.
2. `URL`(Uniform Resource Locators)



    - 서버에 자원을 요청하기 위해 입력하는 영문 주소
3. HTTP Method
    a. `GET` : 존재하는 자원에 대한 요청 - **조회(`Read`)**
    b. `POST` : 새로운 자원을 **생성(`Create`)**
    c. `PUT` : 존재하는 자원에 대한 **변경(`Update`)**
    d. `DELETE` : 존재하는 자원에 대한 **삭제(`Delete`)**
    - 이와 같이 데이터에 대한 조회, 생성, 변경, 삭제 동작을 HTTP 요청 메서드로 정의할 수 있습니다. 참고로 때에 따라서는 POST 메서드로 PUT, DELETE의 동작도

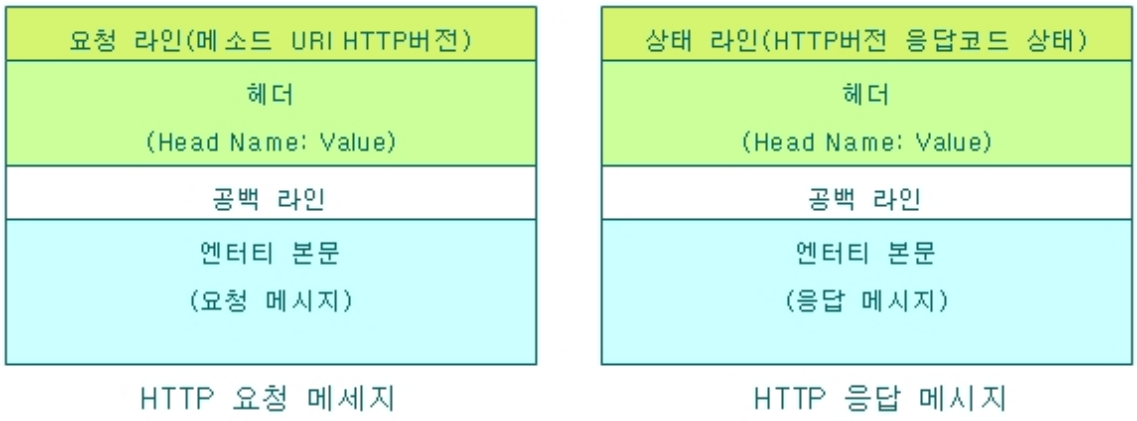
수행할 수 있습니다

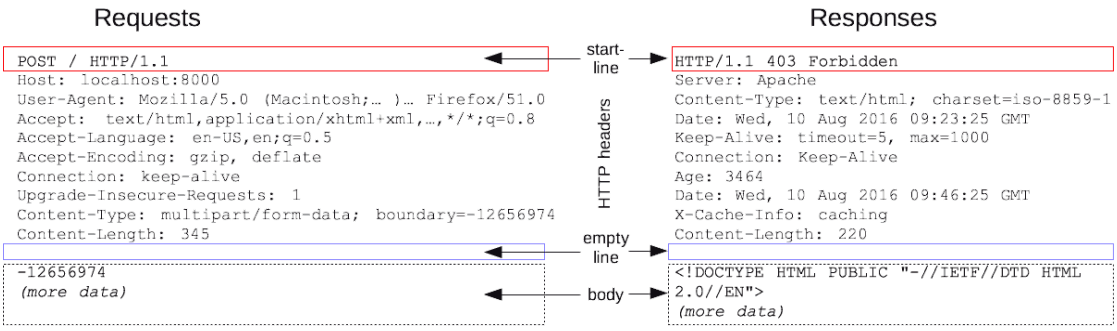
4. HTTP 상태코드
    - 2xx: 성공
        - **200** : GET 요청에 대한 성공
        - 201: 정상적으로 생성이 되었다는걸 서버에서 알려줌 (회원가입 등의 기능에서 사용)
        - 204 : No Content. 성공했으나 응답 본문에 데이터가 없음
        - 205 : Reset Content. 성공했으나 클라이언트의 화면을 새로 고침하도록 권고
        - 206 : Partial Conent. 성공했으나 일부 범위의 데이터만 반환
    - 3xx: 리다이렉션
        - 300번대의 상태 코드는 대부분 클라이언트가 이전 주소로 데이터를 요청하여 서버에서 새 URL로 리다이렉트를 유도하는 경우이다.
        - 301 : Moved Permanently, 요청한 자원이 새 URL에 존재
        - 303 : See Other, 요청한 자원이 임시 주소에 존재
        - 304 : Not Modified, 요청한 자원이 변경되지 않았으므로 클라이언트에서 캐싱된 자원을 사용하도록 권고. ETag와 같은 정보를 활용하여 변경 여부를 확인
    - 4xx: 클라이언트 에러
        - 400번대 상태 코드는 대부분 클라이언트의 코드가 잘못된 경우이다. 유효하지 않은 자원을 요청했거나 요청이나 권한이 잘못된 경우 발생하는데, 가장 익숙한 상태 코드는 404 코드이다. 요청한 자원이 서버에 없다는 의미를 말한다.
        - **400** : Bad Request, 잘못된 요청
        - 401 : Unauthorized, 권한 없이 요청. Authorization 헤더가 잘못된 경우
        - **403** : Forbidden, 서버에서 해당 자원에 대해 접근 금지
        - **404** : Not Found, 요청한 자원이 서버에 존재하지 않음. 없는 url 혹은 존재하지 않는 api를 가지고 요청했을때
        - 405 : Method Not Allowed, 허용되지 않은 요청 메서드
        - 409 : Conflict, 최신 자원이 아닌데 업데이트하는 경우. ex) 파일 업로드 시 버전 충돌
    - 5xx: 서버 에러

- 501 : Not Implemented, 요청한 동작에 대해 서버가 수행할 수 없는 경우
- 503 : Service Unavailable, 서버가 과부하 또는 유지 보수로 내려간 경우

HTTP Message

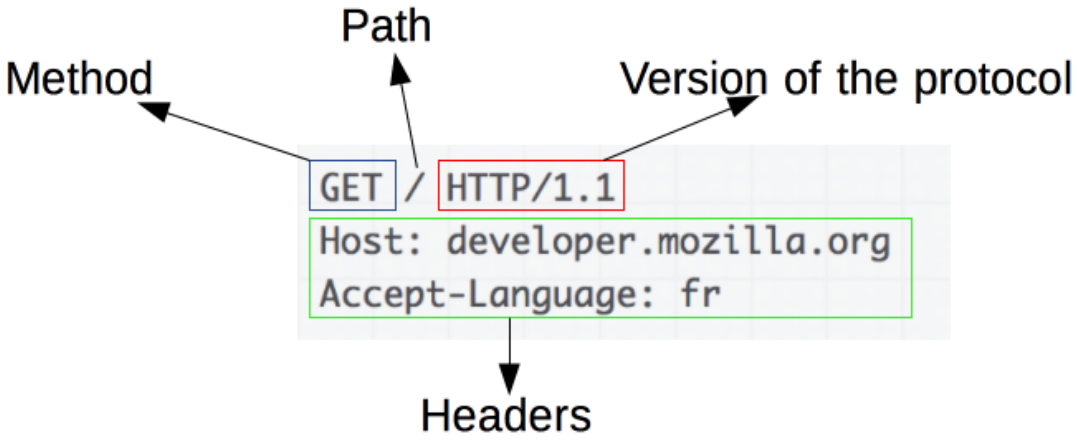
- HTTP Request & Response 구조





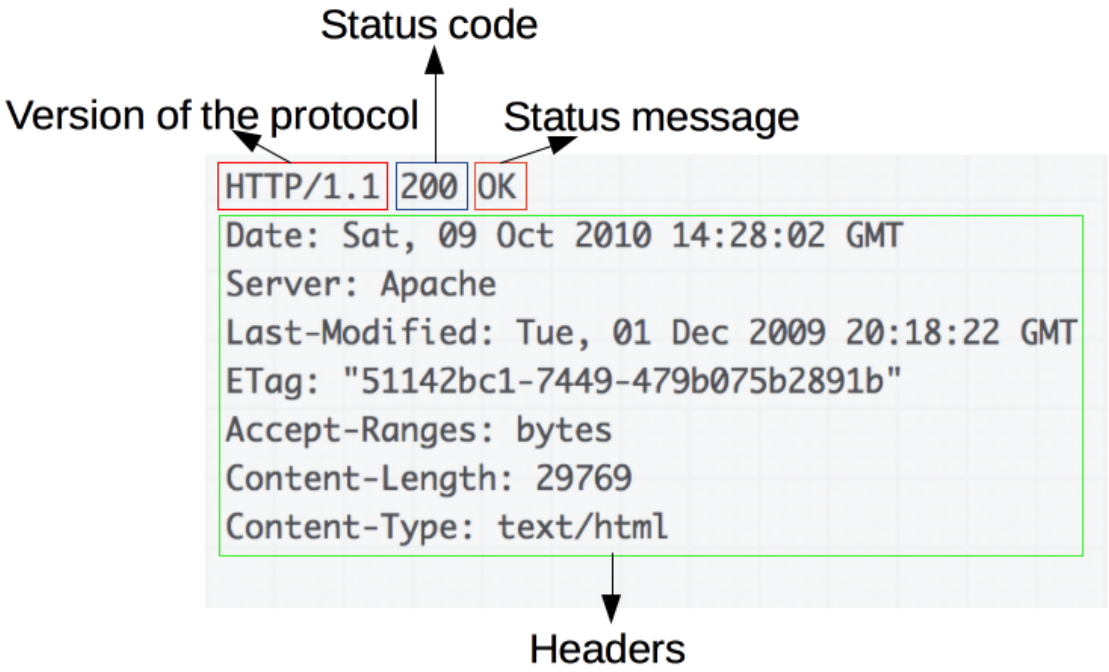
1. 시작 줄(`start-line`)에는 실행되어야 할 요청, 또은 요청 수행에 대한 성공 또는 실패가 기록되어 있습니다. 이 줄은 항상 한 줄로 끝납니다.
2. 옵션으로 HTTP 헤더(`header`) 세트가 들어갑니다. 여기에는 요청에 대한 설명, 혹은 메시지 본문에 대한 설명이 들어갑니다.
3. 요청에 대한 모든 메타 정보가 전송되었음을 알리는 빈 줄(blank line)이 삽입됩니다.
4. 요청과 관련된 내용(HTML 폼 콘텐츠 등)이 옵션으로 들어가거나, 응답과 관련된 문서(document)가 들어갑니다. 본문의 존재 유무 및 크기는 첫 줄과 HTTP 헤더에 명시됩니다.

- HTTP Request



- Head
  - start-line
    1. HTTP Method
    2. URL, Protocol, Port, Domain Path
    3. HTTP Version
  - header : 서버에 대한 추가 정보를 전달하는 선택적 헤더
- Body
  - 본문은 요청의 마지막 부분에 들어갑니다.
  - 모든 요청에 본문이 들어가지는 않습니다.
  - `GET`, `HEAD`, `DELETE`, `OPTIONS`처럼 리소스를 가져오는 요청은 보통 본문이 필요가 없습니다.
  - single-resource bodies
  - multiple-resource bodies

- HTTP Response



- Head
  - status-line
    1. Protocol Version
    2. Status Code
    3. Status Message
  - header : 요청 헤더와 비슷한, HTTP 헤더들
- Body

JavaScript

Async/Await

async/await 란?

- JavaScript의 비동기 처리 방법
- `async` 함수는 항상 promise를 반환, 만약 `async` 함수의 반환값이 명시적으로 promise가 아니라면 암묵적으로 promise로 감싸집니다.

- 자바스크립트는 `await` 키워드를 만나면 프라미스가 처리될 때까지 기다립니다.
- promise, async/await 비교1
  - promise

```
function foo() {
        return Promise.resolve(1)
}
```

```
function foo() {
        return Promise.resolve(1).then(() => undefined)
}
```

  - async/await

```
async function foo() {
        return 1
}
```

- `promise` VS `async/await`
  - promise

```
function p() {
return new Promise((resolve, reject) => {
  resolve('hello');
// or reject(new Error('error');
  });
}

p().then((n) => console.log(n));
```

  - async/await

```
async function p2(){ // async을 지정해주면 Promise를 리턴하는 함수로 만들어줍니다.
  return 'hello';
}

p2().then((n) => console.log(n));
```

  - 이처럼 `async`를 사용하면 `promise` 코드를 훨씬 **직관적**으로 나타낼 수 있습니다.

- 함수에 `async`만 붙이면 자동으로 `promise` 객체로 인식되고, `return`값은 `resolve()`값과 동일합니다.

async/await 사용

1. async/await 문법

```
async function 함수명() {
  await 비동기_처리_메서드_명();
}
```

- 저 함수의 앞에 `async` 라는 예약어를 붙입니다. 그러고 나서 함수의 내부 로직 중 HTTP 통신을 하는 비동기 처리 코드 앞에 `await`를 붙입니다. 여기서 주의하셔야 할 점은 비동기 처리 메서드가 꼭 프로미스 객체를 반환해야 `await`가 의도한 대로 동작합니다.

2. async

- `async`는 function 앞에 위치

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

f().then(alert); // 1
```

3. await

- 프라미스가 처리될 때까지 함수 실행을 대기
- `await`는 `promise.then`보다 좀 더 세련되게 프라미스의 `result` 값을 얻을 수 있도록 해주는 문법

> ⚠️ **일반 함수엔 `await`을 사용할 수 없습니다.**

- `async` 함수가 아닌데 `await`을 사용하면 문법 에러가 발생

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // Syntax error
}
```

async/await 예제

- 예제1

```
async function showAvatar() {

  // JSON 읽기
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // github 사용자 정보 읽기
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = await githubResponse.json();

  // 아바타 보여주기
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // 3초 대기
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

- 예제2

```
async function fetchAuthorName(postId) {
  const postResponse = await fetch(
    `https://jsonplaceholder.typicode.com/posts/${postId}`
  );
  const post = await postResponse.json();
  const userId = post.userId;

  try {
    const userResponse = await fetch(
      `https://jsonplaceholder.typicode.com/users/${userId}`
    );
    const user = await userResponse.json();
    return user.name;
  } catch (err) {
    console.log("Faile to fetch user:", err);
    return "Unknown";
  }
}

fetchAuthorName(1).then((name) => console.log("name:", name));
```

Fetch

fetch() 란?

- 원격 API를 간편하게 호출할 수 있도록 브라우저에서 제공하는 함수
- 자바스크립트를 사용하면 필요할 때 **서버에 네트워크 요청을 보내고 새로운 정보를 받아오는 일**을 할 수 있다.
- HTTP 파이프라인을 구성하는 요청과 응답 등의 요소를 JavaScript에서 접근하고 조작할 수 있는 인터페이스를 제공

fetch() 사용

- 사용법

```
fetch(url, options)
  .then((response) => console.log("response:", response))
  .catch((error) => console.log("error:", error));
```

  - `url` : 접근하고자 하는 URL
  - `ptions` : 선택 매개변수, method나 header 등을 지정할 수 있음
  - `options`에 아무것도 넘기지 않으면 요청은 `GET` 메서드로 진행
    - 데이터 추가할 때는 `POST`
  - `fetch()`를 호출하면 브라우저는 네트워크 요청을 보내고 `promise`가 반환된다.

fetch() 예제

- 예제1

```
let result = fetch(serverURL);

result
  .fetch(response => {
    if(response.ok)  {
      //요청 성공
    }
  })
  .catch(error => {
    // 요청 실패
  })
```

  - 네트워크 요청 성공 시 Promise는 Response 객체를 `resolve` 한다.
  - 네트워크 요청 실패 시 Promise는 Response 객체를 `reject` 한다.

- 예제2 - async/await

```
let url = 'https://api.github.com/repos/javascript-tutorial/ko.javascript.info/com
mits';
let response = await fetch(url);

let commits = await response.json(); // 응답 본문을 읽고 JSON 형태로 파싱함

alert(commits[0].author.login);
```

  - `response.text()` : 응답을 읽고 텍스트를 반환
  - `response.json()` : 응답을 JSON 형태로 파싱
  - `response.formData()` : 응답을 `FormData` 객체 형태로 반환

- 예제2 - promise

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commit
s')
  .then(response => response.json())
  .then(commits => alert(commits[0].author.login));
```