1. 노드 시작하기

1.1 핵심 개념 이해하기

1.1.1 서버

- 서버 : 네트워크를 통해 클라이언트에 정보나 서비스를 제공하는 컴퓨터 또는 프로그램

- 클라이언트 : 요청을 보내는 주체로 브라우저일 수도 있고, 데스크톱 프로그램일 수도 있고,   
모바일 앱일 수도 있고, 다른 서버에 요청을 보내는 서버일 수도 있음.

- 서버라고 해서 요청에 대한 응답만 하는 것은 아님. 다른 서버에 요청을 보낼 수도 있으며,   
이때는 요청을 보낸 서버가 클라이언트 역할을 함.

1.1.2 자바스크립트 런타임

- 노드는 자바스크립트 런타임이다. (자바스크립트 실행기)

- 런타임은 특정 언어로 만든 프로그램들을 실행할 수 있는 환경

- 노드는 V8과 더불어 libuv라는 라이브러리를 사용  
(libuv 라이브러리는 노드의 특성인 이벤트 기반, 논 블로킹 I/O 모델을 구현하고 있음)

1.1.3 이벤트 기반

- 이벤트 기반(event-driven)이란, 이벤트가 발생할 때 미리 저장해둔 작업을 수행하는 방식

- 이벤트 기반 시스템에서는 특정 이벤트가 발생할 때 무엇을 할지 미리 등록해두어야 함.  
이를 이벤트 리스너(event listener)에 콜백(callback) 함수를 등록한다고 표현

- 노드도 이벤트 기반 방식으로 동작하므로, 이벤트가 발생하면 이벤트 리스너에 등록해둔 콜백 함수를 호출. 발생한 이벤트가 없거나 발생했던 이벤트를 다 처리하면, 노드는 다음 이벤트가 발생할 때까지 대기

- 이벤트 루프 : 이벤트 발생 시 호출할 콜백 함수들을 관리, 호출된 콜백 함수의 실행 순서 결정

- 백그라운드 : setTimeout 같은 타이머나 이벤트 리스너들이 대기하는 곳

- 테스크 큐 (콜백 큐) : 이벤트 발생 후, 백그라운드에서는 테스크 큐로 타이머나 이벤트 리스너들의 콜백 함수들을 보냄

1.1.4 논 블로킹 I/O

- 논 블로킹 : 이전 작업이 완료될 때까지 대기하지 않고 다음 작업을 수행함  
(논 블로킹과 동시가 같은 의미가 아님)

블로킹 : 이전 작업이 끝나야만 다음 작업을 수행함

- 노드에서는 동기와 블로킹이 유사, 비동기와 논 블로킹이 유사

1.1.5 싱글 스레드

- 노드가 채택하고 있는 방식 : 싱글 스레드, 논 블로킹 모델

- 노드가 싱글 스레드로 동작하지 않는 경우 : 스레드풀, 워커 스레드

1.2 서버로서의 노드

- 노드의 장점 : 멀티 스레드 방식에 비해 적은 컴퓨터 자원 사용, I/O 작업이 많은 서버로 적합, 멀티 스레드 방식보다 쉬움, 웹 서버가 내장되어 있음, 자바스크립트를 사용함, JSON 형식과 쉽게 호환됨

- 노드의 단점 : 기본적으로 싱글 스레드라서 CPU 코어를 하나만 사용, CPU 작업이 많은 서버로는 부적합, 하나뿐인 스레드가 멈추지 않도록 관리가 필요함, 서버 규모가 커졌을 때 서버를 관리하기 어려움, 어중간한 성능

1.3 서버 외의 노드

- 노드 기반으로 돌아가는 대표적인 웹 프레임워크 : 앵귤러(Angular), 리액트(React), 뷰(Vue) 등

1.4 개발 환경 설정하기

- 노드 설치

- npm 버전 업데이트

- 비주얼 스튜디오 코드 설치

2. 알아두어야 할 자바스크립트

2.1 ES2015+

2.1.1 const, let

- 보통 자바스크립트를 배울 때는 var로 변수를 선언하는 방법부터 배움. 하지만 var은 이제 const와 let이 대체함.

- const와 let의 차이 : const는 한 번 값을 할당하면 다른 값을 할당할 수 없음, 초기화할 때 값을 할당하지 않으면 에러 발생 (상수)

2.1.2 템플릿 문자열

- 백틱( ` ) : 문자열 안에 변수를 넣을 수 있음

2.1.3 객체 리터럴

- ES5 : { name: name, age: age }

- ES2015 : { name, age }

2.1.4 화살표 함수

- 화살표 함수에서는 function 선언 대신 => 기호로 함수를 선언함. 또한 변수에 대입하면 나중에 재사용 가능

2.1.5 구조분해 할당

- 구조분해 할당을 사용하면 객체와 배열로부터 속성이나 요소를 쉽게 꺼낼 수 있음.

2.1.6 클래스

- 다른 언어처럼 클래스 기반으로 동작하는 것이 아니라, 자바스크립트는 여전히 프로토타입 기반으로 동작함.

2.1.7 프로미스

- 자바스크립트와 노드에서는 주로 비동기를 접함 (특히 이벤트 리스너를 사용할 때 콜백 함수를 자주 사용)

- 프로미스를 쉽게 설명하자면, 실행은 바로 하되 결과값은 나중에 받는 객체 (결과값은 실행이 완료된 후 then이나 catch 메서드를 통해 받는다)

- 프로미스를 return한 경우에는 프로미스가 수행된 후 다음 then이나 catch가 호출된다.

- 프로미스 여러 개를 한 번에 실행하는 방법 : Promise.all

2.1.8 async/await

- 노드 7.6 버전부터 지원되는 기능, ES2017에서 추가됨

- 함수 선언부를 일반 함수 대신 async function으로 교체한 후, 프로미스 앞에 await를 붙임

- for await of : 프로미스 배열을 순회할 수 있음

- async 함수의 반환값은 항상 Promise로 감싸짐

2.2 프런트엔드 자바스크립트

2.2.1 AJAX (Asynchronous Javascript And XML)

- 비동기적 웹 서비스를 개발할 때 사용하는 기법

- 이름에 XML이 들어 있지만 꼭 XML을 사용해야 하는 것은 아니며, 요즘에는 JSON을 많이 사용

- 쉽게 말해 페이지 이동 없이 서버에 요청을 보내고 응답을 받는 기술

- 웹 사이트 중에서 페이지 전환 없이 새로운 데이터를 불러오는 사이트는 대부분 AJAX 기술을 사용하고 있다고 보면 됨

- AJAX 요청 : jQuery나 axios 같은 라이브러리를 이용해서 보냄

2.2.2 FormData

- HTML form태그의 데이터를 동적으로 제어할 수 있는 기능 (주로 AJAX와 함께 사용)

- append 메서드 : 키-값 형식의 데이터를 저장할 수 있음, has 메서드 : 주어진 키에 해당하는 값이 있는지의 여부를 알림, get 메서드 : 주어진 키에 해당하는 값 하나를 가져옴, getAll : 해당하는 모든 값을 가져옴, delete 메서드 : 현재 키를 제거, set : 현재 키를 수정

2.2.3 encodeURIComponent, decodeURIComponent

- AJAX 요청을 보낼 때, 주소에 한글이 들어가는 경우

- 서버 종류에 따라 다르지만 서버가 한글 주소를 이해하지 못하는 경우가 있는데, 이럴 때는 window 객체의 메서드인 encodeURIComponent 메서드를 사용

- 받는 쪽에서는 decodeURIComponent를 사용

2.2.4 데이터 속성과 dataset

- 보안 : 프런트엔드에서 데이터를 내려보낼 때 첫 번째로 고려해야 할 점임. (클라이언트를 믿지 마라!)

- 데이터 속성(data attribute) : HTML5에서 HTML과 관련된 데이터를 저장하는 공식적인 방법

3. 노드 기능 알아보기

3.1 REPL 사용하기

- 자바스크립트는 스크립트 언어이므로 미리 컴파일을 하지 않아도 즉석에서 코드를 실행할 수 있음.

- REPL(Read Eval Print Loop) : 입력한 코드를 읽고(Read), 해석하고(Eval), 결과물을 반환하고(Print), 종료할 때까지 반복한다(Loop)

- 노드의 REPL을 직접 사용하기 : 윈도우에서는 명령 프롬프트, 맥이나 리눅스에서는 터미널을 열고 node를 입력, VS Code에서는 Ctrl + ` 을 누르면 터미널을 켤 수 있음

- REPL 종료 : Ctrl + C 를 두 번 누르거나, REPL 창에 .exit을 입력

- REPL은 한두 줄짜리 코드를 테스트해보는 용도로는 좋지만 여러 줄의 코드를 실행하기에는 불편함. 긴 코드인 경우에는 코드를 자바스크립트 파일로 만든 후 파일을 통째로 실행하는 것이 좋음.

3.2 JS 파일 실행하기

- 자바스크립트 파일을 생성

- 콘솔에서 node [자바스크립트 파일 경로]로 실행. 확장자(.js)는 생략해도 됨

3.3 모듈로 만들기

- 노드는 코드를 모듈로 만들 수 있다는 점에서 브라우저의 자바스크립트와 다르다.

- 모듈이란 특정한 기능을 하는 함수나 변수들의 집합, 모듈 자체로도 하나의 프로그램이면서 다른 프로그램의 부품으로도 사용할 수 있음.

- 모듈로 만들어두면 여러 프로그램에 해당 모듈을 재사용할 수 있음

3.4 노드 내장 객체 알아보기

3.4.1 global

- 브라우저의 window와 같은 전역 객체 (전역 객체 : 모든 파일에서 접근하 수 있음)

- 생략 가능 (global.require -> require, global.console -> console)

- 노드의 window, document 객체 : 노드에 DOM이나 BOM이 없으므로 window나 document 객체는 노드에서 사용할 수 없음.

- global 객체의 남용 : global 객체의 속성에 값을 대입하여 파일 간에 데이터를 공유할 수 있지만, 이를 남용해서는 안 됨. 프로그램의 규모가 커질수록 어떤 파일에서 global 객체에 값을 대입했는지 찾기 힘들어져 유지 보수에 어려움을 겪을 수 있음. 다른 파일의 값을 사용하고 싶다면, 모듈 형식으로 만들어서 명시적으로 값을 불러와 사용하는 것이 좋음.

3.4.2 console

- console도 노드에서는 window 대신 global 객체 안에 들어 있음

- console 객체는 보통 디버깅을 위해 사용, 대표적으로 console.log 메서드 (개발하면서 변수에 값이 제대로 들어 있는지 확인할 때, 에러 발생 시 에러 내용을 콘솔에 표시하기 위해, 코드 실행 시간을 알아보려고 할 때)

- 다른 로깅 함수들

\* console.time(레이블) : console.timeEnd(레이블)과 대응되어 같은 레이블을 가진 time과 timeEnd 사이의 시간을 측정

\* console.log(내용) : 평범한 로그를 콘솔에 표시함. console.log(내용, 내용, …)처럼 여러 내용을 동시에 표시할 수도 있음.

\* console.error(에러 내용) : 에러를 콘솔에 표시

\* console.table(배열) : 배열의 요소로 객체 리터럴을 넣으면, 객체의 속성들이 테이블 형식으로 표현됨.

\* console.dir(객체, 옵션) : 객체를 콘솔에 표시할 때 사용함. 옵션의 colors를 true로 하면 콘솔에 색이 추가되어 보기가 한결 편해짐, depth는 객체 안의 객체를 몇 단계까지 보여줄지를 결정함. (기본값 : 2)

\* console.trace(레이블) : 에러가 어디서 발생했는지 추적할 수 있게 함.

3.4.3 타이머

- 타이머 기능을 제공하는 함수인 setTimeout, setInterval, setImmediate는 노드에서 window 대신 global 객체 안에 들어 있음.

\* setTimeout(콜백 함수, 밀리초) : 주어진 밀리초(1,000분의 1초) 이후에 콜백 함수를 실행

\* setInterval(콜백 함수, 밀리초) : 주어진 밀리초마다 콜백 함수를 반복 실행

\* setImmediate(콜백 함수) : 콜백 함수를 즉시 실행

- 이 타이머 함수들은 모두 아이디를 반환함. 아이디를 사용하여 타이머를 취소할 수 있음.

\* clearTimeout(아이디) : setTimeout을 취소

\* clearInterval(아이디) : setInterval을 취소

\* clearImmediate(아이디) : setImmediate를 취소

- cf) setImmediate(콜백)과 setTimeout(콜백, 0)

\* 파일 시스템 접근, 네트워킹 같은 I/O 작업의 콜백 함수 안에서 타이머를 호출하는 경우는setImmediate가 setTimeout(콜백, 0)보다 먼저 실행됨.

\* 하지만, setImmediate가 항상 setTimeout(콜백, 0)보다 먼저 호출되지는 않음.

\* setTimeout(콜백, 0)은 사용하지 않는 것을 권장

3.4.4 \_\_filename, \_\_dirname

- 파일에 \_\_filename과 \_\_dirname을 넣어두면, 실행 시 현재 파일명과 현재 파일 경로로 바뀜.

3.4.5 module, exports, require

- module.exports : 한 번에 대입

- 각각의 변수를 exports 객체에 하나씩 넣는 방법

- 한 번 require한 파일은 require.cache에 저장되므로, 다음 번에 require할 때는 새로 불러오지 않고 require.cache에 있는 것이 재사용됨 / require.main : 노드 실행 시 첫 모듈을 가리킴

3.4.6 process

- process 객체는 현재 실행되고 있는 노드 프로세스에 대한 정보를 담고 있음

3.4.6.1 process.env

- 시스템의 환경 변수를 출력 (왼쪽이 환경 변수의 이름, 오른쪽이 값)

NODE\_OPTIONS=--max-old-space-size=8192

UV\_THREADPOOL\_SIZE=8

\* NODE\_OPTIONS : 노드를 실행할 때의 옵션들을 입력받는 환경 변수

\* --max-old-space-size=8192 : 노드의 메모리를 8GB까지 사용할 수 있게 함

\* UV\_THREADPOOL\_SIZE : 노드에서 기본적으로 사용하는 스레드풀의 스레드 개수를 조절할   
 수 있게 함

- process.env는 서비스의 중요한 키를 저장하는 공간으로도 사용됨

3.4.6.2 process.nextTick(콜백)

- 이벤트 루프가 다른 콜백 함수들보다 nextTick의 콜백 함수를 우선으로 처리하도록 만듦

- process.nextTick은 setImmediate나 setTimeout보다 먼저 실행됨

- 코드 맨 밑에 Promise를 넣은 것은 resolve된 Promise도 nextTick처럼 다른 콜백들보다 우선시되기 때문. 그래서 process.nextTick과 Promise를 마이크로태스크(microtask)라고 따로 구분지어 부름

3.4.6.3 process.exit(코드)

- 실행 중인 노드 프로세스를 종료

- 서버 환경에서 이 함수를 사용하면 서버가 멈추므로, 특수한 경우를 제외하고는 서버에서 잘 사용하지 않음. 하지만, 서버 외의 독립적인 프로그램에서는 수동으로 노드를 멈추기 위해 사용

- process.exit 메서드는 인수로 코드 번호를 줄 수 있음. 인수를 주지 않거나 0을 주면 정상 종료를 뜻하고, 1을 주면 비정상 종료를 뜻함 (만약 에러가 발생해서 종료하는 경우에는 1을 넣으면 됨)

3.5 노드 내장 모듈 사용하기

3.5.1 os

- 웹 브라우저에 사용되는 자바스크립트는 운영체제의 정보를 가져올 수 없지만, 노드는 os 모듈에 정보가 담겨 있어 정보를 가져올 수 있음

\* os.arch() : process.arch()와 동일함

\* os.platform() : process.platform()과 동일함

\* os.type() : 운영체제의 종류를 보여줌

\* os.uptime() : 운영체제의 부팅 이후 흐른 시간(초)을 보여줌 / process.uptime() : 노드의 실행 시간

\* os.hostname() : 컴퓨터의 이름을 보여줌

\* os.release() : 운영체제의 버전을 보여줌

\* os.homedir() : 홈 디렉터리 경로를 보여줌

\* os.tmpdir() : 임시 파일 저장 경로를 보여줌

\* os.cpus() : 컴퓨터의 코어 정보를 보여줌

\* os.freemem() : 사용 가능한 메모리(RAM)을 보여줌

\* os.totalmem() : 전체 메모리 용량을 보여줌

\* os.constants() : 각종 에러와 신호에 대한 정보가 담겨 있음

- os 모듈은 주로 컴퓨터 내부 자원에 빈번하게 접근하는 경우 사용됨. 즉, 일반적인 웹 서비스를 사용할 때는 사용 빈도가 높지 않음.

3.5.2 path

- 폴더와 파일의 경로를 쉽게 조작하도록 도와주는 모듈

- path 모듈이 필요한 이유 중 하나는 운영체제별로 경로 구분자가 다르기 때문

윈도 타입(\로 구분)과 POSIX 타입(/로 구분, 유닉스 기반의 운영체제 - 맥과 리눅스)

\* path.sep : 경로의 구분자 (윈도 : \, POSIX : /)

\* path.delimiter : 환경 변수의 구분자 (윈도 : 세미콜론(;), POSIX : 콜론(:))

\* path.dirname(경로) : 파일이 위치한 폴더 경로를 보여줌

\* path.extname(경로) : 파일의 확장자를 보여줌

\* path.basename(경로, 확장자) : 파일의 이름(확장자 포함)을 표시  
(파일의 이름만 표시하고 싶다면 basename의 두 번째 인수로 파일의 확장자를 넣으면 됨)

\* path.parse(경로) : 파일 경로를 root, dir, base, ext, name으로 분리

\* path.format(객체) : path.parse()한 객체를 파일 경로로 합침

\* path.normalize(경로) : \나 /를 실수로 여러 번 사용했거나 혼용했을 때 정상적인 경로로 변환

\* path.isAbsolute(경로) : 파일의 경로가 절대경로인지 상대경로인지를 true나 false로 알림

\* path.relative(기준경로, 비교경로) : 경로를 두 개 넣으면 첫 번째 경로에서 두 번째 경로로 가는 방법을 알려줌

\* path.join(경로, …) : 여러 인수를 넣으면 하나의 경로로 합침  
(상대경로인 ..(부모 디렉터리)과 .(현 위치)도 알아서 처리함)

\* path.resolve(경로, …) : /를 만나면 path.resolve는 절대경로로 인식해서 앞의 경로를 무시  
(path.join은 상대경로로 처리)

3.5.3 url

- 인터넷 주소를 쉽게 조작하도록 도와주는 모듈

- url 처리에는 크게 두 가지 방식(노드 버전 7에서 추가된 WHATWG(웹 표준을 정하는 단체의 이름) 방식의 url, 예전부터 노드에서 사용하던 방식의 url)이 있음

- 기존 노드 방식에서는 두 메서드를 주로 사용함

\* url.parse(주소) : 주소를 분해함. WHATWG 방식과 비교하면 username과 password 대신 auth 속성이 있고, searchParams 대신 query가 있다.

\* url.format(객체) : WHATWG 방식 url과 기존 노드의 url을 모두 사용할 수 있음. 분해되었던 url 객체를 다시 원래 상태로 조립함.

- host 부분 없이 pathname 부분만 오는 주소인 경우에는 WHATWG 방식이 처리할 수 없음.

- WHATWG 방식은 search 부분을 searchParams라는 특수한 객체로 반환하므로 유용함. search 부분은 보통 주소를 통해 데이터를 전달할 때 사용함. search는 물음표(?)로 시작하고, 그 뒤에 **키=값** 형식으로 데이터를 전달함. 여러 키가 있을 경우에는 **&**로 구분함.

- searchParams 객체는 search 부분을 조작하는 다양한 메서드를 지원함. (formData 객체 메서드와 비슷함)

\* getAll(키) : 키에 해당하는 모든 값을 가져옴. category 키에는 nodejs와 javascript라는 두 가지 값이 들어 있음.

\* get(키) : 키에 해당하는 첫 번째 값만 가져옴.

\* has(키) : 해당 키가 있는지 없는지를 검사함.

\* keys() : searchParams의 모든 키를 반복기(iterator) 객체로 가져옴

\* values() : searchParams의 모든 값을 반복기 객체로 가져옴.

\* append(키, 값) : 해당 키를 추가함. 같은 키의 값이 있다면 유지하고 하나 더 추가함.

\* set(키, 값) : append와 비슷하지만, 같은 키의 값들을 모두 지우고 새로 추가함.

\* delete(키) : 해당 키를 제거함.

\* toString() : 조작한 searchParams 객체를 다시 문자열로 만듦. 이 문자열을 search에 대입하면 주소 객체에 반영됨.

- query 같은 문자열보다 searchParams가 유용한 이유는 query의 경우 querystring 모듈을 한 번 더 사용해야 하기 때문

3.5.4 querystring

- WHATWG 방식의 url 대신 기존 노드의 url을 사용할 때, search 부분을 사용하기 쉽게 객체로 만드는 모듈

- 실제 프로젝트에서 모듈 여러 개를 파일 하나에 불러올 수 있음.

\* querystring.parse(쿼리) : url의 query 부분을 자바스크립트 객체로 분해함.

\* querystring.stringify(객체) : 분해된 query 객체를 문자열로 다시 조립함.

3.5.5 crypto

- 다양한 방식의 암호화를 도와주는 모듈

3.5.5.1 단방향 암호화

- 단방향 암호화란 복호화할 수 없는 암호화 방식을 뜻함.

- 복호화 : 암호화된 문자열을 원래 문자열로 되돌려놓는 것을 의미

- 즉, 단방향 암호화는 한 번 암호화하면 원래 문자열을 찾을 수 없다. 복호화할 수 없으므로 암호화라고 표현하는 대신 해시 함수라고 부르기도 한다.

- 단방향 암호화 알고리즘은 주로 해시 기법을 사용함. 해시 기법이란 어떠한 문자열을 고정된 길이의 다른 문자열로 바꿔버리는 방식 (입력 문자열의 길이는 다르지만, 출력 문자열의 길이는 고정)

- 노드에서 해시 함수는 다음과 같이 사용됨.

\* createHash(알고리즘) : 사용할 해시 알고리즘을 넣음. md5, sha1, sha256, sha512 등이 가능(md5, sha1은 이미 취약점이 발견됨)

\* update(문자열) : 반환할 문자열을 넣음.

\* digest(인코딩) : 인코딩할 알고리즘을 넣음. (base64, hex, latin1이 주로 사용되는데, 그중 base64가 결과 문자열이 가장 짧아 애용됨). 결과물로 반환된 문자열을 반환함.

- 현재는 주로 pbkdf2나 bcrypt, scrypt라는 알고리즘으로 비밀번호를 암호화하고 있다.

- pbkdf2는 기존 문자열에 salt라고 불리는 문자열을 붙인 후, 해시 알고리즘을 반복 적용하는 것

- pbkdf2는 간단하지만 bcrypt나 scrypt보다 취약하므로, 나중에 더 나은 보안이 필요하면 bcrypt나 scrypt 방식을 사용하면 됨.

3.5.5.2 양방향 암호화

- 암호화된 문자열을 복호화할 수 있으며, 키(열쇠)라는 것이 사용됨.

- 대칭형 암호화에서 암호를 복호화하려면 암호화할 때 사용한 키와 같은 키를 사용해야 함.

\* crypto.createCipheriv(알고리즘, 키 iv) : 암호화 알고리즘과 키, iv를 넣음. 암호화 알고리즘은 aes-256-cbc를 사용했으며, 다른 알고리즘을 사용해도 됨(aes-256-cbc 알고리즘의 경우 키는 32바이트여야 하고, iv는 16바이트여야 함). iv는 암호화할 때 사용하는 초기화 벡터를 의미함. 사용 가능한 알고리즘 목록은 crypto.getCiphers()를 호출하면 볼 수 있음.

\* cipher.update(문자열, 인코딩, 출력 인코딩) : 암호화할 대상과 대상의 인코딩, 출력 결과물의 인코딩을 넣음. 보통 문자열은 utf8 인코딩을, 암호는 base64를 많이 사용함.

\* cipher.final(출력 인코딩) : 출력 결과물의 인코딩을 넣으면 암호화가 완료됨.

\* crypto.createDecipheriv(알고리즘, 키, iv) : 복호화할 때 사용. 암호화할 때 사용했던 알고리즘과 키, iv를 그대로 넣어야 함.

\* decipher.update(문자열, 인코딩, 출력 인코딩) : 암호화된 문장, 그 문장의 인코딩, 복호화할 인코딩을 넣음. createCipheriv의 update()에서 urf8, base64순으로 넣었다면 createDecipheriv의 update()에서는 base64, utf8순으로 넣으면 됨.

\* decipher.final(출력 인코딩) : 복호화할 결과물의 인코딩을 넣음.

- 좀 더 간단하게 암호화를 하고 싶다면 npm 패키지인 crypto-js를 사용할 것

3.5.6 util

- util이라는 이름처럼 각종 편의 기능을 모아둔 모듈

- 계속해서 API가 추가되고 있으며, 가끔 deprecated되어 사라지는 경우도 있음

- util에서 자주 사용되는 두 메서드

\* util.deprecate : 함수가 deprecated 처리되었음을 알림. 첫 번째 인수로 넣은 함수를 사용했을 때 경고 메시지가 출력됨. 두 번째 인수로 경고 메시지를 넣으면 됨. 함수가 조만간 사라지거나 변경될 때 알려줄 수 있어 유용함.

\* util.promisify : 콜백 패턴을 프로미스 패턴으로 바꿈. 바꿀 함수를 인수로 제공하면 됨. 이렇게 바꿔두면 async/await 패턴까지 사용할 수 있음.

3.5.7 worker\_threads

- 노드에서 멀티 스레드 방식으로 작업하는 방법을 소개. worker\_threads 모듈로 가능함.

3.5.8 child\_process

- 노드에서 다른 프로그램을 실행하고 싶거나 명령어를 수행하고 싶을 때 사용하는 모듈

- 이 모듈을 통해 다른 언어의 코드(파이썬 등)를 실행하고 결과값을 받을 수 있음.

- exec vs spawn : exec은 셸을 실행해서 명령어를 수행하고, spawn은 새로운 프로세스를 띄우면서 명령어를 실행함. spawn에서도 세 번째 인수로 { shell: true }를 제공하면 exec처럼 셸을 실행해서 명령어를 수행함.

3.5.9 기타 모듈들

- assert : 값을 비교하여 프로그램이 제대로 동작하는지 테스트하는 데 사용

- dns : 도메인 이름에 대한 IP 주소를 얻어내는 데 사용

- net : HTTP보다 로우 레벨인 TCP나 IPC 통신을 할 때 사용

- string\_decoder : 버퍼 데이터를 문자열로 바꾸는 데 사용

- tls : TLS와 SSL에 관련된 작업을 할 때 사용

- tty : 터미널과 관련된 작업을 할 때 사용

- dgram : UDP와 관련된 작업을 할 때 사용

- v8 : V8 엔진에 직접 접근할 때 사용

- vm : 가상 머신에 직접 접근할 때 사용

3.6 파일 시스템 접근하기

- fs 모듈은 파일 시스템에 접근하는 모듈

- 즉, 파일을 생성하거나 삭제하고, 읽거나 쓸 수 있음. 폴더도 만들거나 지울 수 있음.

- fs는 기본적으로 콜백 형식의 모듈이므로 실무에서 사용하기가 불편함. 따라서 fs 모듈을 프로미스 형식으로 바꿔 사용

3.6.1 동기 메서드와 비동기 메서드

- setTimeout 같은 타이머와 process.nextTick 외에도, 노드는 대부분의 메서드를 비동기 방식으로 처리하지만 몇몇 메서드는 동기 방식으로도 사용할 수 있음. 특히 fs 모듈이 그러한 메서드를 많이 가지고 있음.

- 비동기 메서드들은 백그라운드에 해당 파일을 읽으라고만 요청하고 다음 작업으로 넘어감. 이 방식은 수백 개의 I/O 요청이 들어와도 메인 스레드는 백그라운드에 요청 처리를 위임함. 그 후로도 얼마든지 요청을 더 받을 수 있음. 나중에 백그라운드가 각각의 요청 처리가 완료되었다고 알리면 그때 콜백 함수를 처리하면 됨.

- 노드에서는 동기-블로킹 방식과 비동기-논 블로킹 방식이 대부분

동기-블로킹 방식 : 백그라운드 작업 완료 여부를 계속 확인하며, 호출한 함수가 바로 return 되지 않고 백그라운드 작업이 끝나야 return됨

비동기-논 블로킹 방식 : 호출한 함수가 바로 return되어 다음 작업으로 넘어가며, 백그라운드 작업 완료 여부는 신경 쓰지 않고 나중에 백그라운드가 알림을 줄 때 비로소 처리

3.6.2 버퍼와 스트림 이해하기

- 노드는 파일을 읽을 때 메모리에 파일 크기만큼 공간을 마련해두며 파일 데이터를 메모리에 저장한 뒤 사용자가 조작할 수 있도록 함. 이때 메모리에 저장된 데이터가 바로 버퍼임.

- Buffer : 버퍼를 직접 다룰 수 있는 클래스

- Buffer 객체는 여러 가지 메서드를 제공함.

\* form(문자열) : 문자열을 버퍼로 바꿀 수 있음. length 속성은 버퍼의 크기를 알림(바이트 단위)

\* toString(버퍼) : 버퍼를 다시 문자열로 바꿀 수 있음. 이때 base64나 hex를 인수로 넣으면 해당 인코딩으로 변환 가능

\* concat(배열) : 배열 안에 든 버퍼들을 하나로 합침

\* alloc(바이트) : 빈 버퍼를 생성함. 바이트를 인수로 넣으면 해당 크기의 버퍼가 생성됨.

- 모든 내용을 버퍼에 다 쓴 후에야 다음 동작으로 넘어가므로 파일 읽기, 압축, 파일 쓰기 등의 조작을 연달아 할 때 매번 전체 용량을 버퍼로 처리해야 다음 단계로 넘어갈 수 있음.

- 그래서 버퍼의 크기를 작게 만든 후 여러 번으로 나눠 보내는 방식이 등장함. 이를 편리하게 만든 것이 스트림

3.6.3 기타 fs 메서드 알아보기

- fs는 파일 시스템을 조작하는 다양한 메서드를 제공함. 다음 네 가지의 메서드는 모두 비동기 메서드이므로 한 메서드의 콜백에서 다른 메서드를 호출함.

\* fs.access(경로, 옵션, 콜백) : 폴더나 파일에 접근할 수 있는지를 체크. 두 번째 인수로 상수들을 넣음(F\_OK는 파일 존재 여부, R\_OK는 읽기 권한 여부, W\_OK는 쓰기 권한 여부를 체크함). 파일/폴더나 권한이 없다면 에러가 발생하는데 파일/폴더가 없을 때의 에러 코드는 ENOENT

\* fs.mkdir(경로, 콜백) : 폴더를 만드는 메서드. 이미 폴더가 있다면 에러가 발생하므로 먼저 access 메서드를 호출해서 확인하는 것이 중요함.

\* fs.open(경로, 옵션, 콜백) : 파일의 아이디(fd 변수)를 가져오는 메서드. 파일이 없다면 파일을 생성한 뒤 그 아이디를 가져옴. 두 번째 인수로 어떤 동작을 할 것인지를 설정할 수 있음(쓰려면 w, 읽으려면 r, 기존 파일에 추가하려면 a).

\* fs.rename(기존 경로, 새 경로, 콜백) : 파일의 이름을 바꾸는 메서드. 기존 파일 위치와 새로운 파일 위치를 적으면 됨(꼭 같은 폴더를 지정할 필요는 없으므로 잘라내기 같은 기능을 할 수도 있음).

- 폴더 내용 확인 및 삭제와 관련된 메서드

\* fs.readdir(경로, 콜백) : 폴더 안의 내용물을 확인할 수 있음. 배열 안에 내부 파일과 폴더명이 나옴.

\* fs.unlink(경로, 콜백) : 파일을 지울 수 있음. 파일이 없다면 에러가 발생하므로 먼저 파일이 있는지를 꼭 확인해야 함.

\* fs.rmdir(경로, 콜백) : 폴더를 지울 수 있음. 폴더 안에 파일들이 있다면 에러가 발생하므로 먼저 내부 파일을 모두 지우고 호출해야 함.

3.6.4 스레드풀 알아보기

- 비동기 메서드들은 백그라운드에서 실행되고, 실행된 후에는 다시 메인 스레드의 콜백 함수나 프로미스의 then 부분이 실행된다. 이때 fs 메서드를 여러 번 실행해도 백그라운드에서 동시에 처리되는데, 바로 스레드풀이 있기 때문.

- fs 외에도 내부적으로 스레드풀을 사용하는 모듈로는 crypto, zlib, dns.lookup 등이 있음.

- 우리는 스레드풀을 직접 컨트롤할 수는 없지만 개수를 조절할 수는 있음. 스레드의 숫자를 크게 할 때는 자신의 컴퓨터 코어 개수와 같거나 많게 두어야 뚜렷한 효과가 발생함.

3.7 이벤트 이해하기

- 스트림을 배울 때 on(‘data’, 콜백) 또는 on(‘end’, 콜백)을 사용했음. 바로 data라는 이벤트와 end라는 이벤트가 발생할 때 콜백 함수를 호출하도록 이벤트를 등록한 것임.

- createReadStream 같은 경우는 내부적으로 알아서 data와 end 이벤트를 호출하지만, 우리가 직접 이벤트를 만들 수도 있는데 이는 events 모듈을 사용하면 됨.

\* on(이벤트명, 콜백) : 이벤트 이름과 이벤트 발생 시의 콜백을 연결함 (이렇게 연결하는 동작을 이벤트 리스닝이라고 부름). 이벤트 하나에 이벤트 여러 개를 달아줄 수도 있음.

\* addListener(이벤트명, 콜백) : on과 기능이 같음.

\* emit(이벤트명) : 이벤트를 호출하는 메서드. 이벤트 이름을 인수로 넣으면 미리 등록해뒀던 이벤트 콜백이 실행됨.

\* once(이벤트명, 콜백) : 한 번만 실행되는 이벤트.

\* removeAllListeners(이벤트명) : 이벤트에 연결된 모든 이벤트 리스너를 제거

\* removeListener(이벤트명, 리스너) : 이벤트에 연결된 리스너를 하나씩 제거(리스너를 넣어야 한다는 것을 잊으면 안됨)

\* off(이벤트명, 콜백) : 노드 10 버전에서 추가된 메서드로, removeListener와 기능이 같음.

\* listenerCount(이벤트명) : 현재 리스너가 몇 개 연결되어 있는지 확인

3.8 예외 처리하기

- 예외란 보통 처리하지 못한 에러를 가리킴.

- 멀티 스레드 프로그램에서는 스레드 하나가 멈추면 그 일을 다른 스레드가 대신하지만, 노드의 메인 스레드는 하나뿐이므로 그 하나를 소중히 보호해야 함.

- 메인 스레드가 에러로 인해 멈춘다는 것은 스레드를 갖고 있는 프로세스가 멈춘다는 뜻이고, 전체 서버도 멈춘다는 뜻과 같다.

3.8.1 자주 발생하는 에러들

- **node: command not found** : 노드를 설치했지만 이 에러가 발생하는 경우는 환경 변수가 제대로 설정되지 않은 것. 환경 변수에는 노드가 설치된 경로가 포함되어야 함. node 외의 다른 명령어도 마찬가지이다. 그 명령어를 수행할 수 있는 파일이 환경 변수에 들어 있어야 명령어를 콘솔에서 사용할 수 있다.

- **ReferenceError: 모듈 is not defined** : 모듈을 require했는지 확인

- **Error: Cannot find module 모듈명** : 해당 모듈을 require했지만 설치하지 않았음.

- **Error: Can’t set headers after they are sent** : 요청에 대한 응답을 보낼 때 응답을 두 번 이상 보냄. (요청에 대한 응답은 한 번만 보내야 한다. 응답을 보내는 메서드를 두 번 이상 사용하지 않았는지 체크해야 함.)

- **FATAL ERROR: CALL\_AND\_RETRY\_LAST Allocation failed - JavaScript heap out of memory** : 코드를 실행할 때 메모리가 부족하여 스크립트가 정상 작동하지 않는 경우 (코드가 잘못되었을 확률이 높으므로 코드를 점검해보기. 만약 코드는 정상이지만 노드가 활용할 수 있는 메모리가 부족한 경우라면 노드의 메모리를 늘릴 수 있음.)

- **UnhandledPromiseRejectionWarning: Unhandled promise rejection** : 프로미스 사용 시 catch 메서드를 붙이지 않으면 발생

- **EADDRINUSE 포트번호** : 해당 포트번호에 이미 다른 프로세스가 연결되어 있음. 그 프로세스는 노드 프로세스일 수도 있고 다른 프로그램일 수도 있음. 그 프로세스를 종료하거나 다른 포트번호를 사용해야 함.

- **EACCES 또는 EPERM** : 노드가 작업을 수행하는 데 권한이 충분하지 않음. 파일/폴더 수정, 삭제, 생성 권한을 확인해보는 것이 좋음. 맥이나 리눅스 운영체제라면 명령어 앞에 sudo를 붙이는 것도 방법

- **EJSOMPARSE** : package.json 등의 JSON 파일에 문법 오류가 있을 때 발생. 자바스크립트 객체와는 형식이 조금 다르니 쉼표 같은 게 빠지거나 추가되지는 않았는지 확인하기

- **ECONNREFUSED** : 요청을 보냈으나 연결이 성립되지 않을 때 발생. 요청을 받는 서버의 주소가 올바른지, 꺼져 있지는 않는지 확인하기

- **ETARGET** : package.json에 기록한 패키지 버전이 존재하지 않을 때 발생. 해당 버전이 존재하는지 확인하기

- **ETIMEOUT** : 요청을 보냈으나 응답이 일정 시간 내에 오지 않을 때 발생. 역시 요청을 받는 서버의 상태를 점검해보기

- **ENOENT: no such file or directory** : 지정한 폴더나 파일이 존재하지 않는 경우. 맥이나 리눅스 운영체제에서는 대소문자도 구별함.