**Node.js**

**<<01. 개요 및 설치>>**

\* **Node.js:** Chrom V8 JavaScript 엔진으로 빌드된 JavaScript 런타임(프로그래밍 언어가 동작하는 환경)

**\* Node.js가 실행되는 환경:** Node.js가 설치된 컴퓨터, Node.js가 실행되는 브라우저

**\* Node.js LTS:** LTS(Long Term Suppoerted)는 장기적으로 안정되고 신뢰도가 높은 지원이 보장되는 버전이므로, 유지/보수와 보안(서버 운영 등)에 초점을 맞춰 대부분 사용자에게 추천되는 버전

**\* Node Version Manager(nvm):**

- Node.js 버전 맞춰주는 관리자

- https://github.com/coreybutler/nvm-windows/releases

- VS터미널로 nvm –version 으로 설치 되었는지 확인

**<<02. NVM 사용법>>**

**\* nvm으로 node.js 다운 방법**

- VS 터미널에서 nvm ls 버전

**\* nvm으로 node.js 사용 방법**

- VS 터미널에서 nvm use 버전

**\* node.js 여러 버전을 설치하는 이유**

- 여러 개발을 할 수 있으니

**\* nvm 명령어 확인**

- VS 터미널에서 nvm --help

**<<03. NPM 개요(1)>>**

**\* NPM(Node Package Manager):** 전세계의 개발자들이 만든 다양한 기능(패키지, 모듈)들을 관리

**<<04. NPM 개요(2)>>**

**\* 터미널에서 npm init -y**

**\* 터미널에서 npm install parcel-bundler -D**

- 하나의 패키지로 취급되어 다 다운 받아짐

- node\_modules > package.json > parcel-bundler > package.json에 들어가면 dependencies 에 사용한 패키지를 확인 가능

- 내가 만든 package.json에 devDependencies에 내가 설치한 패키지가 들어감

**\* npm을 이용**하면 실제 패키지가 설치되고 devDependencies혹은 dependencies라는 옵션에 명시가 됨

**\* npm 장점:**

- 만약 패키지가 들어간node\_moduels를 지워버렸음 -> 나중에 다시 터미널에서 npm install 혹은 npm i를 하면 설치됨

- 한번 설치한 패키지는 내역이 남기 때문에 프로젝트에 해당하는 패키지를 다운 받을 수 있음

**\* package-lock vs package**

- package는 직접으로 관리하는 것

- package-lock은 내부 패키지에 대한 것을 자동으로 관리

- node\_modules는 삭제해도 되지만 프로젝트를 관리하면서 삭제하면 안 됨.

**\* devDependencies vs dependencies**

- devDependencies: 커맨드에서 ‘-D’ 라는 것을 붙이면 여기에 설치됨

- dependencies: 그렇지 않은 것은 여기에 설치됨

\* 패키지 A만 설치했는데 패키지A가 사용하는 또 다른 패키지가 설치됨

- 내부에서 어떤 패키지를 사용하는지 다 알 수는 없으니까 package-lock이 관리해줌

**\* 개발용 의존성 패키지 설치(-D, --save-dev)**

$ npm install -D 패키지명

- 개발할 때만 필요하고 웹브라우저에서 동작할 때는 필요하지 않음(웹브라우저에서는 직접 동작 X)

**\* 일반 의존성 설치**

$ npm install 패키지명

- 웹브라우저에서도 동작할 수 있음

**<<05. 개발 서버 실행과 빌드>>**

**\* Open with Live Sever을 이용하여 src 연결:** 구식 방법

**\* 터미널을 이용:**

- parcel index.html

- 내 로컬 컴퓨터에서 index.html을 중심으로 개발용으로 서버를 열겠다

- package.json에서 scripts에다 스크립트 이름 지정하고 index.html 작성함

- 터미널에서 npm run dev //실행시킬 스크립트 이름이 dev

- parcel-bundler를 이용해 로컬 컴퓨터로 서버를 엶

**\* lodash**

- main.js 추가, import \_ from ‘lodash’

ㄴ node\_modules에 있는 lodash.js를 가져옴

- console.log(\_.camelCase(‘hello world’)); //hello world가 camelCase에 맞게 출력됨

**\* parcel index.html:** 로컬 환경에서 개발용으로 서버를 열어 브라우저에서 확인

**\* parcel build index.html:** 사용자들이 보는 용도의 결과물이 출력됨(코드 난독화 후)

**\* 코드 난독화:** 작성된 코드를 읽기 어렵게 만드는 작업, 빌드된 결과(제품)는 브라우저에서 해석되는 용도로, 용량을 축소하고 읽기 어렵게 만드는 등의 최적화를 거치는 것이 좋음

**\* 번들(Bundle):** 프로젝트 개발에 사용한 여러 모듈(패키지)을 하나로 묶어내는 작업

**<<06. 유의적 버전(SemVer)>>**

**\* 시맨틱 버전, 유의적 버전(Semantic Versioning, SemVer):**

- ^Major.Minor.Patch

- E.g, 12.14.1

- Major: 기존 버전과 호환되지 않는 새로운 버전

- Minor: 기존 버전과 호환되는 새로운 기능이 추가된 버전

- Patch: 기존 버전과 호환되는 버그 및 오타 등이 수정된 버전

- ^: Major 버전 안에서 가장 최신 버전으로 업데이트 가능

- npm info lodash //latest: 최신 버전 알려줌

- 프로젝트에 명시한 버전(내가 개발한 환경의 package.json)과 실제로 설치된 버전(node\_modules의 package.json)이 같은지 확인

- npm install lodash@4.17.20 //@버전 => 해당하는 버전으로 덮어씌움

ㄴ 명시한 버전과 실제 설치된 버전 변경됨

- npm update lodash //최신 버전으로 업데이트

ㄴ 그 이유는 명시한 패키지 버전의 앞에 ^ 기호를 넣어줬기 때문에 가장 최신의 상태로 업데이트 함

ㄴ 만약 ^ 없으면 기존 버전 그대로 유지

- 상황에따라 언제든지 최신 버전으로 업데이트 될 수 있어 편리할 수 있지만, 원하지 않는 버전으로 업데이트 될 수 있음 => 적절히 사용하는 것이 좋음

**<<07. NPM 프로젝트의 버전 관리(.gitgnore)>>**

\* 후작업으로 만들 수 있는 폴더/파일은 github에 따로 push 안 해도 됨(= 버전 관리 안 해도 됨)

\* npm i 를 통해 명시된 패키지 다운

\* npm run build 를 통해 서버 엶

\* 버전 관리를 하는 것이 효율이 떨어질 수 있음. 그래서 .gitignore을 통하여 .cache, dist, node\_modules, 이외 파일 등는 버전 관리 안 하겠다고 선언하는 게 좋음.

\* .gitignore: git에서 무시할 내용을 명시

\* git status

\* git add .

\* git commit -m ‘프로젝트 생성’

\* git log

\* git remote add origin github //원격의 저장소 추가

\* git push origin master