3장

3장

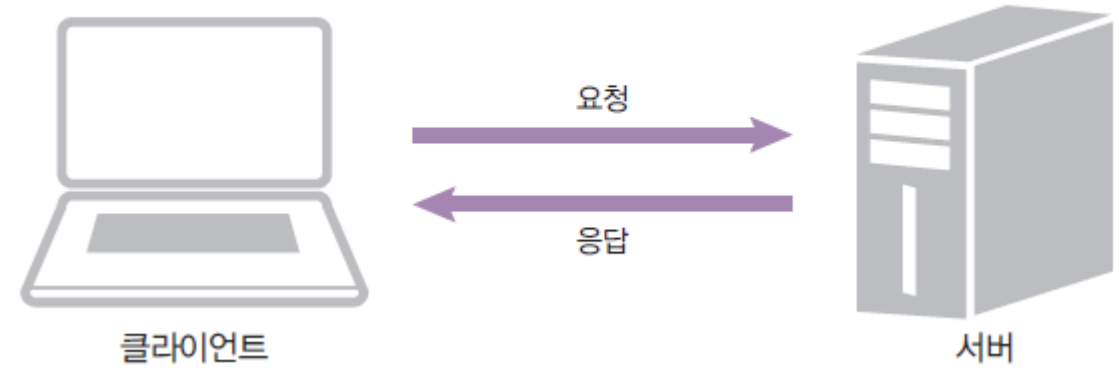
3.1 요청과 응답 이해하기

1. 서버와 클라이언트

서버와 클라이언트의 관계

- 클라이언트가 서버로 요청(request)을 보냄
- 서버는 요청을 처리
- 처리 후 클라이언트로 응답(response)을 보냄

그림 4-1 클라이언트와 서버의 관계



2. 노드로 http 서버 만들기

- http 요청에 응답하는 노드 서버
  - createServer로 요청 이벤트에 대기
  - req 객체는 요청에 관한 정보가, res 객체는 응답에 관한 정보가 담겨 있음

```
const http = require('http');

http.createServer((req, res) => {
    // 여기에 어떻게 응답할지 작성
})
```

3. 8080 포트에 연결하기

- res 메서드로 응답 보냄
  - write로 응답 내용을 적고
  - end로 응답 마무리(내용을 넣어도 됨)
- listen(포트) 메서드로 특정 포트에 연결

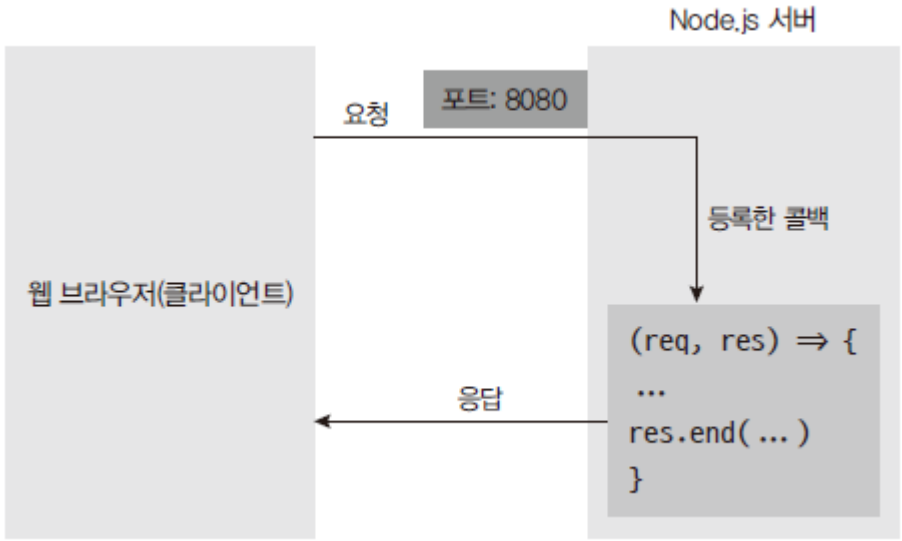
```
const http = require('http');

http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type' : 'text/html; charset=utf-8' });
    res.write('<h1>Hello Node!</h1>');
    res.end('<p>Hello Server!</p>')
})
    .listen(8080, () => {
        // 서버 연결
        console.log('8080 서버 연결');
    })
```

4. 8080 포트로 접속하기

- 스크립트를 실행하면 8080 포트에 연결됨

- http://localhost:8080 또는 http://127.0.0.1:8080에 접속

그림 4-2 서버 실행 화면

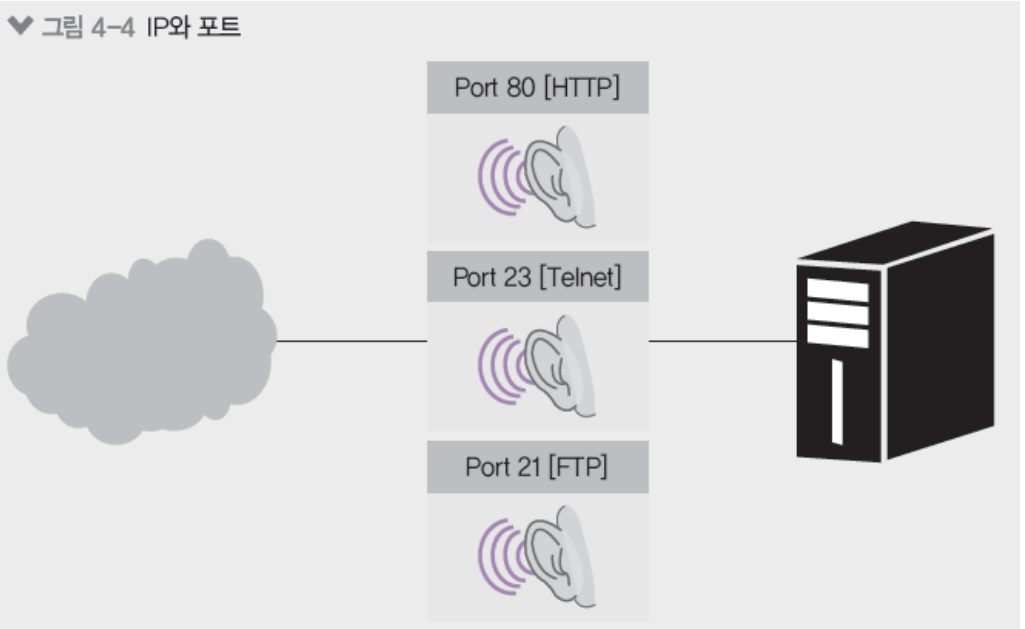


5. localhost와 포트

- localhost는 컴퓨터 내부 주소
  - 외부에서는 접근 불가능

- 포트는 서버 내에서 프로세스를 구분하는 번호
  - 기본적으로 http 서버는 80번 포트 사용(생략가능, https는 443)
  - 예) www.google.com:80 -> www.google.com
  - 다른 포트로 데이터베이스나 다른 서버 동시에 연결 가능

그림 4-4 IP와 포트



6. 이벤트 리스너 붙이기

- listening과 error 이벤트를 붙일 수 있음.

```
const http = require('http');

const server = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type' : 'text/html; charset=utf-8' });
    res.write('<h1>Hello Node!</h1>');
    res.end('<p>Hello Server!</p>')
})

server.listen(8080)

server.on('listening', () => {
    // 서버 연결
    console.log('8080 서버 연결');
})

server.on('error', (err) => {
    console.error(err);
})
```

7. 한 번에 여러 개의 서버 실행하기

- createServer를 여러 번 호출하면 됨.
  - 단, 두 서버의 포트를 다르게 지정해야 함.
  - 같게 지정하면 EADDRINUSE 에러 발생

```
const http = require('http');

const server1 = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type' : 'text/html; charset=utf-8' });
    res.write('<h1>Hello Node!</h1>');
    res.end('<p>Hello Server!</p>')
})

server1.listen(8080, () => {
    console.log('8080 서버 연결');
})

const server2 = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
    res.write('<h1>Hello Node!</h1>');
    res.end('<p>Hello Server!</p>')
})

server2.listen(8081, () => {
    console.log('8081 서버 연결');
})
```

8. html 읽어서 전송하기

- write와 end에 문자열을 넣는 것은 비효율적
  - fs 모듈로 html을 읽어서 전송하자
  - write가 버퍼도 전송 가능

```
const http = require('http');
const fs = require('fs').promises;

http.createServer(async (req, res) => {
    try {
        const data = await fs.readFile('./server.html')
        res.writeHead(200, { 'Content-Type' : 'text/html; charset=utf-8' });
        res.end(data)
    } catch (err) {
        console.error(err);
        res.writeHead(500, { 'Content-Type': 'text/plain; charset=utf-8' })
        res.end(err.message)
    }
})
.listen(8080, () => {
    console.log('8080 서버 연결');
})
```

3.2 REST API와 라우팅

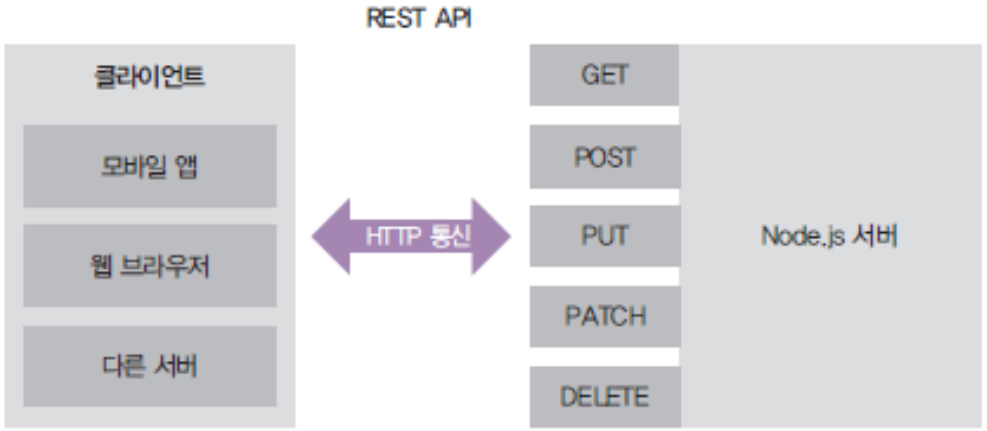
1. REST API

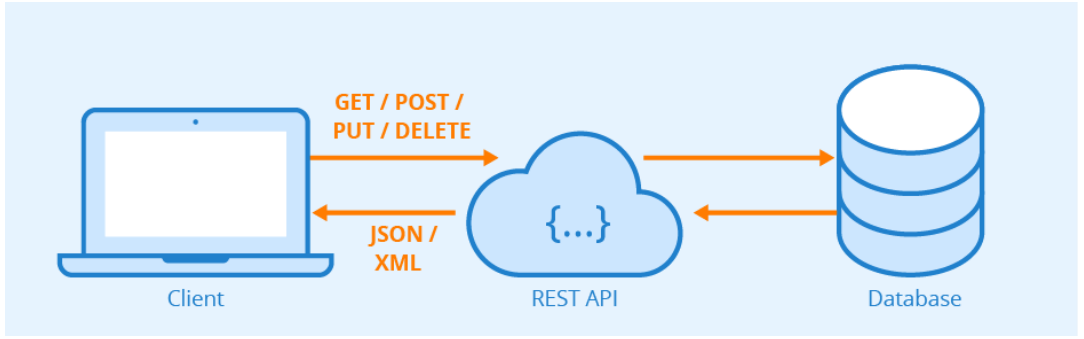
- 서버에 요청을 보낼 때는 주소를 통해 요청의 내용을 표현
  - /index.html이면 index.html을 보내달라는 뜻
  - 항상 html을 요구할 필요는 없음
  - 서버가 이해하기 쉬운 주소가 좋음
- REST API(Representational State Transfer)
  - 서버의 자원을 정의하고 자원에 대한 주소를 지정하는 방법
  - /user이면 사용자 정보에 관한 정보를 요청하는 것
  - /post면 게시글에 관련된 자원을 요청하는 것
- HTTP 요청 메서드
  - GET: 서버 자원을 가져오려고 할 때 사용
  - POST: 서버에 자원을 새로 등록하고자 할 때 사용(또는 뭘 써야할 지 애매할 때)
  - PUT: 서버의 자원을 요청에 들어있는 자원으로 치환하고자 할 때 사용
  - PATCH: 서버 자원의 일부만 수정하고자 할 때 사용
  - DELETE: 서버의 자원을 삭제하고자 할 때 사용

2. HTTP 프로토콜

- 클라이언트가 누구든 서버와 HTTP 프로토콜로 소통 가능
  - iOS, 안드로이드, 웹이 모두 같은 주소로 요청 보낼 수 있음
  - 서버와 클라이언트의 분리

그림 4-15 REST API





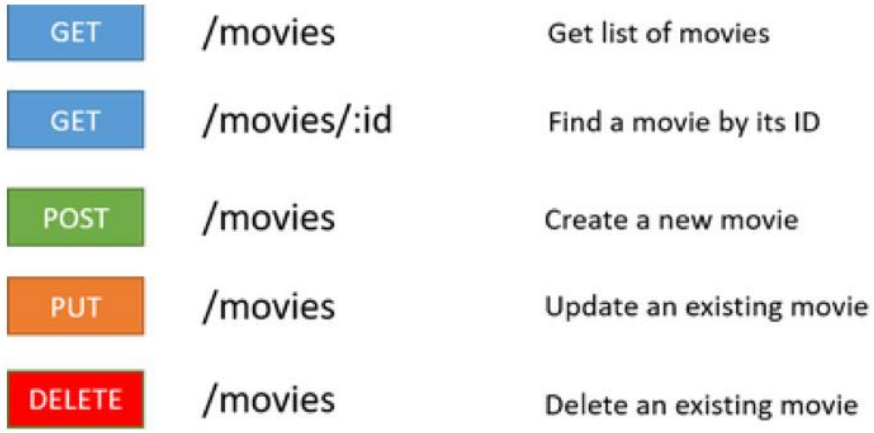
2. HTTP 프로토콜

RESTful

- REST API를 사용한 주소 체계를 이용하는 서버
- GET /user는 사용자를 조회하는 요청, POST /user는 사용자를 등록하는 요청

표 4-1 서버 주소 구조

HTTP 메서드	주소	역할
GET	/	restFront.html 파일 제공
GET	/about	about.html 파일 제공
GET	/users	사용자 목록 제공
GET	기타	기타 정적 파일 제공
POST	/users	사용자 등록
PUT	/users/사용자id	해당 id의 사용자 수정
DELETE	/users/사용자id	해당 id의 사용자 제거



3. REST 서버

restServer.js

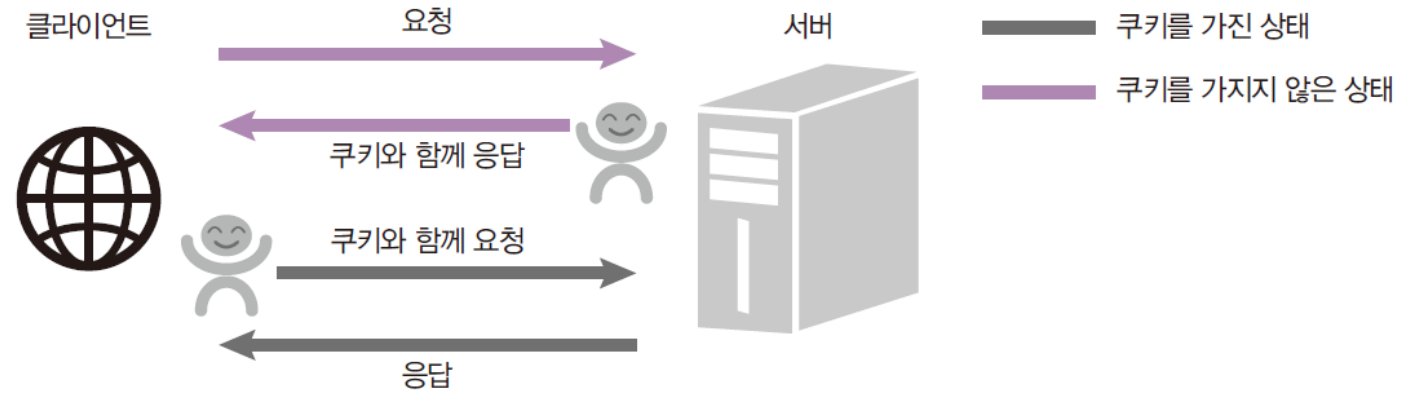
- GET 메서드
  - /, /about 요청 주소는 페이지를 요청하는 것이므로 HTML 파일을 읽어서 전송
  - /users 요청 주소는 AJAX 요청이므로 users 데이터를 전송 (JSON 형식으로 보내기 위해 JSON.stringify)
  - 그 외의 GET 요청은 CSS나 JS 파일을 요청하는 것이므로 찾아서 보내주고, 없다면 404 NOT FOUND 에러를 응답합니다.
- POST와 PUT 메서드 (클라이언트로부터 데이터를 받으므로 특별한 처리가 필요)
  - req.on('data', 콜백), req.on('end', 콜백) 부분 : readStream으로 요청과 같이 들어오는 요청 본문을 받음
    단, 문자열이므로 JSON으로 만드는 JSON.parse 과정이 한 번 필요
- DELETE 메서드
  - 요청이 오면 주소에 들어 있는 키에 해당하는 사용자를 제거

3.3 쿠키와 세션 이해하기

1. 쿠키의 필요성

- 요청에는 한 가지 단점이 있음
  - 누가 요청을 보냈는지 모름(IP 주소와 브라우저 정보 정도만 알고 있음)
  - 로그인을 구현하면 됨
  - 쿠키와 세션이 필요
- 쿠키: 키=값의 쌍
  - name=choi
  - 매 요청마다 서버에 동봉해서 보냄
  - 서버는 쿠키를 읽어 누구인지 파악 가능

그림 4-13 쿠키



2. 쿠키 서버 만들기

- 쿠키 넣는 것을 직접 구현
  - writeHead: 요청 헤더에 입력하는 메서드
  - Set-Cookie: 브라우저에게 쿠키를 설정하라고 명령
- 쿠키: 키=값의 쌍
  - 'name=value'
  - 매 요청마다 서버에 동봉해서 보냄

```
const http = require('http');

http.createServer((req, res) => {
   console.log(req.url, req.headers.cookie);
   res.writeHead(200, {
      'content-type': 'text/html; charset=utf-8',
      'set-cookie' : 'mycookie=test'
   });
   res.end('Hello Cookie')
}).listen(8080, () => {
      // 서버 연결
      console.log('8080 서버 연결');
})
```

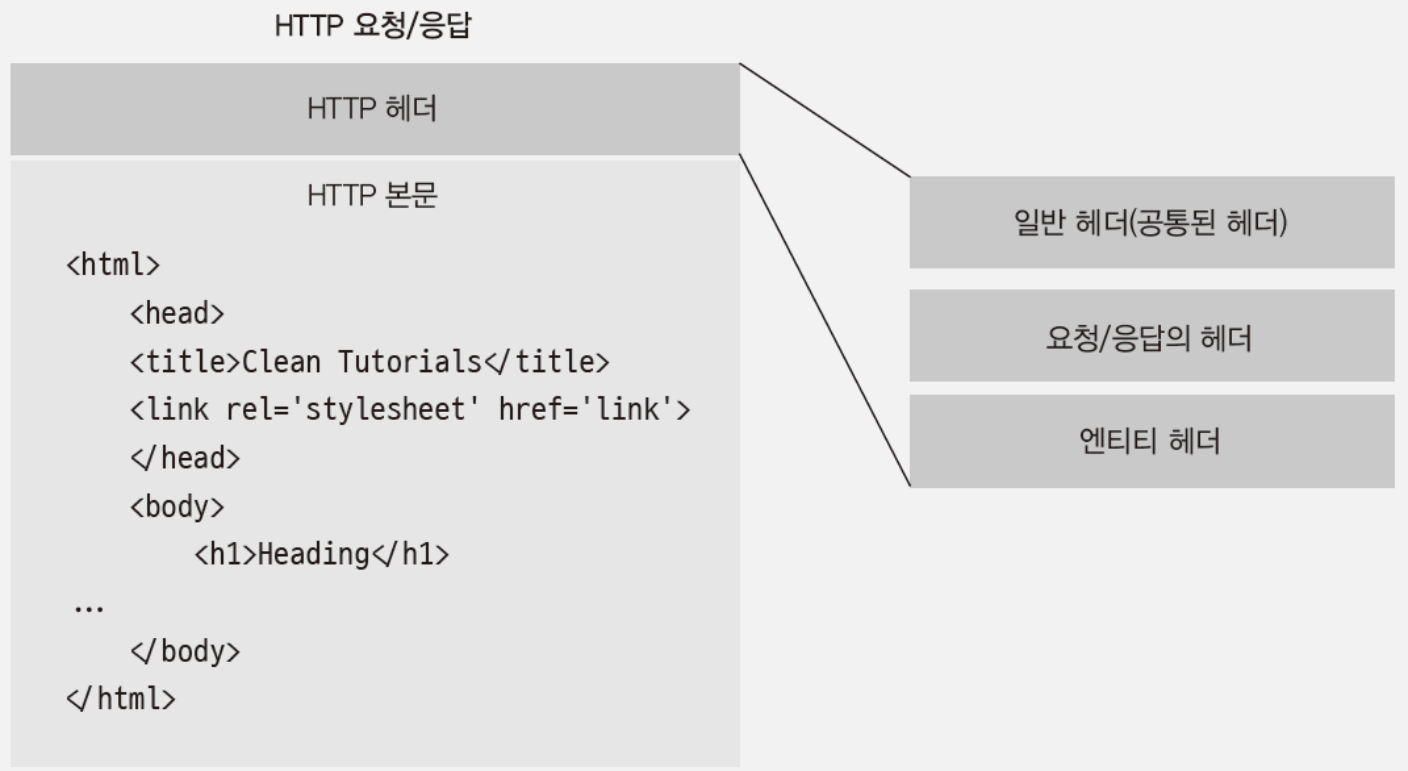
3. 쿠키 서버 실행하기

- req.url: 요청 주소
- req.headers.cookie: 쿠키가 문자열로 담겨있음

- localhost:8080에 접속
  - 첫 요청이 전송되고 응답이 왔을 때 쿠키가 설정됨
  - favicon.ico는 브라우저가 자동으로 보내는 요청
  - 두 번째 요청인 favicon.ico에 쿠키가 넣어져서 요청됨

4. 헤더와 본문

- http 요청과 응답은 헤더와 본문을 가짐
  - Headers(헤더) : 요청 또는 응답에 대한 부가적인 정보를 가짐
  - Body(본문) : 주고받는 실제 데이터 (Payload / Preview, Response)
  - 쿠키는 부가적인 정보이므로 헤더에 저장



5. http 상태 코드

- writeHead 메서드에 첫 번째 인수로 넣은 값
  - 요청이 성공했는지 실패했는지를 알려줌
  - 2XX: 성공을 알리는 상태 코드 [200(성공), 201(작성됨)]
  - 3XX: 리다이렉션(다른 페이지로 이동)을 알리는 상태 코드 [301(영구 이동), 302(임시 이동)]
  - 4XX: 요청 자체에 오류 [401(권한 없음), 403(금지됨), 404(찾을 수 없음)]
  - 5XX: 요청은 제대로 왔지만 서버에 예기치 못한 에러 발생
    [500(내부 서버 오류), 502(불량 게이트웨이), 503(서비스를 사용할 수 없음)]

6. 쿠키로 나를 식별하기

- 쿠키에 내 정보를 입력
  - parseCookies: 쿠키 문자열을 객체로 변환
  - 주소가 /login인 경우와 /인 경우로 나뉨 (/login인 경우 쿼리스트링으로 온 이름을 쿠키로 저장)
  - 그 외의 경우 쿠키가 있는지 없는지 판단
    - 있으면 환영 인사
    - 없으면 로그인 페이지로 리다이렉트

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>쿠키&세션 이해하기</title>
</head>
<body>
    <form action="/login">
        <input id="name" name="name" type="text" placeholder="이름을 입력하세요.">
        <button id="login">로그인</button>
    </form>
</body>
</html>
```

7. 쿠키 옵션

- Set-Cookie 시 다양한 옵션이 있음
  - 쿠키명=쿠키값 : 기본적인 쿠키의 값
  - Expires=날짜 : 특정 날짜 이후 쿠키 제거 (기본값 : 클라이언트가 종료될 때까지)
  - Max-age=초 : 해당 초가 경과 시 쿠키 제거 (Expires보다 우선)
  - Domain=도메인명 : 쿠키가 전송될 도메인을 특정 (기본값 : 현재 도메인)
  - Path=URL : 쿠키가 전송될 URL을 특정 (기본값 : '/' [모든 URL에서 쿠키를 전송 가능])
  - Secure : HTTPS일 경우에만 쿠키가 전송
  - HttpOnly : 자바스크립트에서 쿠키에 접근 불가 (쿠키 조작을 방지)

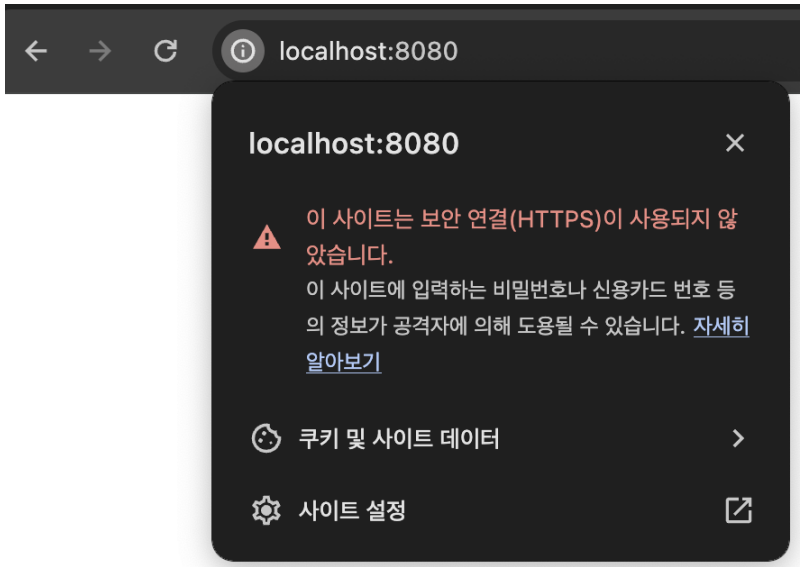
8. 세션 사용하기

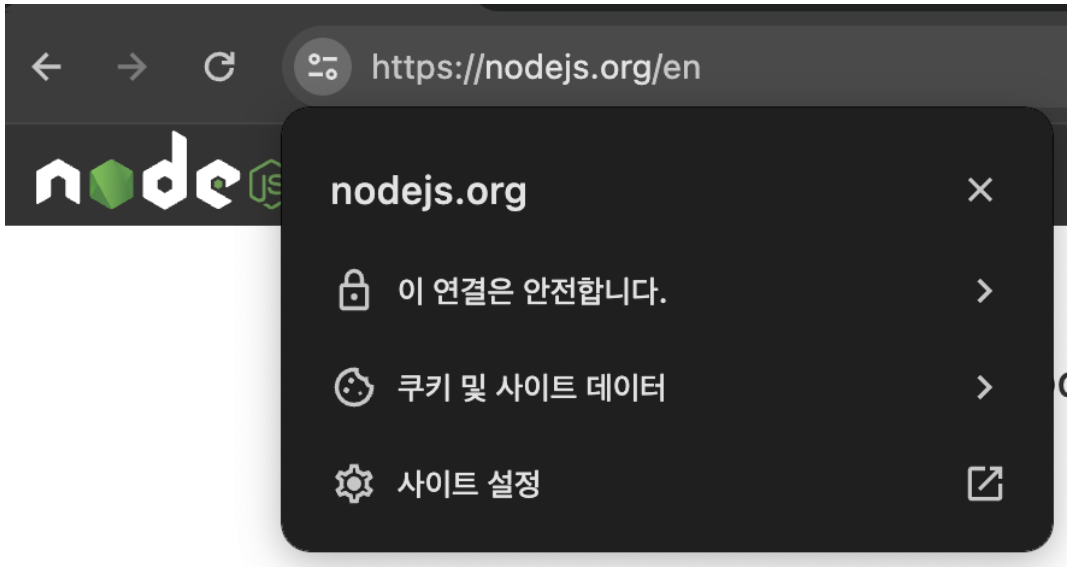
- 쿠키의 정보는 노출되고 수정되는 위험이 있음
  - 중요한 정보는 서버에서 관리하고 클라이언트에는 세션 키만 제공
  - 서버에 세션 객체(session) 생성 후, uniqueInt(키)를 만들어 속성명으로 사용
  - 속성 값에 정보 저장하고 uniqueInt를 클라이언트에 보냄

3.4 https와 http2

1. https

- 웹 서버에 SSL 암호화를 추가하는 모듈
  - 오고 가는 데이터를 암호화해서 중간에 다른 사람이 요청을 가로채더라도 내용을 확인할 수 없음
  - 요즘에는 https 적용이 필수(개인 정보가 있는 곳은 특히)





2. https 서버

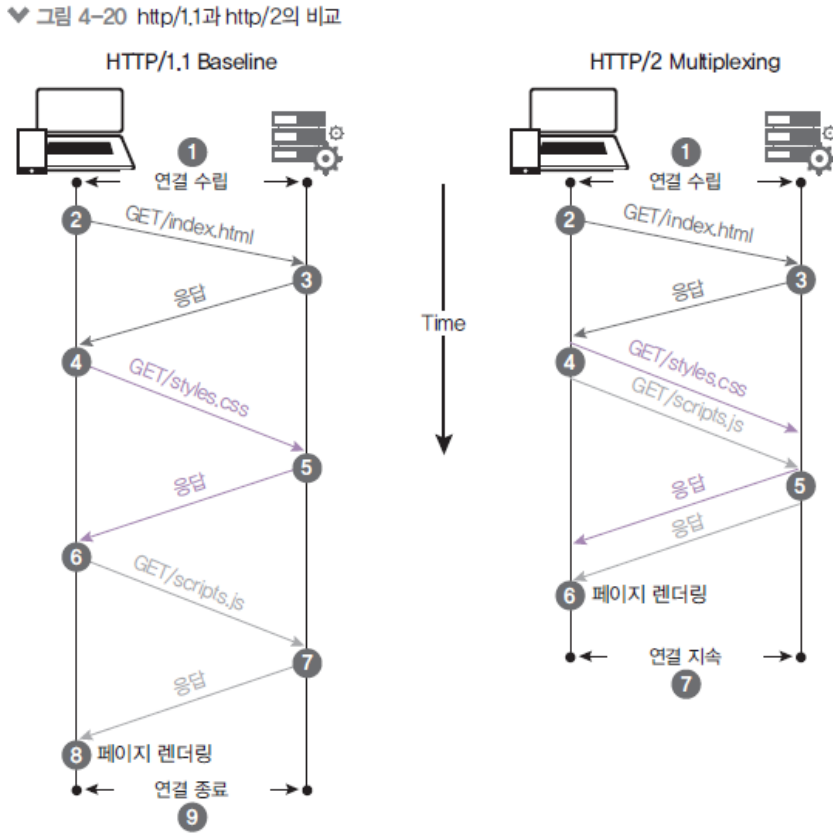
- http 서버를 https 서버로
  - 암호화를 위해 인증서가 필요한데 발급받아야 함

- createServer가 인자를 두 개 받음
  - 첫 번째 인자는 인증서와 관련된 옵션 객체
  - pem, crt, key 등 인증서를 구입할 때 얻을 수 있는 파일 넣기
  - 두 번째 인자는 서버 로직

```
const https = require('https');
const fs = require('fs');

https.createServer({
    cert: fs.readFileSync('도메인 인증서'),
    key: fs.readFileSync('도메인 비밀키'),
    ca: [
        fs.readFileSync('상위 인증서'),
        fs.readFileSync('중간 인증서')
    ],
}, (req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
    res.write('<h1>Hello Node!</h1>');
    res.end('<p>Hello Server!</p>');
})
    .listen(443, () => {
        console.log('443번 포트에서 서버 대기 중입니다!');
});
```

3. http2

- SSL 암호화와 더불어 최신 HTTP 프로토콜인 http/2를 사용하는 모듈
  - 요청 및 응답 방식이 기존 http/1.1보다 개선됨
  - 웹의 속도도 개선됨



4. http2 적용 서버

- https 모듈을 http2로, createServer 메서드를 createSecureServer 메서드로

```
const http2 = require('http2');
const fs = require('fs');

http2.createSecureServer({
    cert: fs.readFileSync('도메인 인증서'),
    key: fs.readFileSync('도메인 비밀키'),
    ca: [
        fs.readFileSync('상위 인증서'),
        fs.readFileSync('중간 인증서')
    ],
}, (req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
    res.write('<h1>Hello Node!</h1>');
    res.end('<p>Hello Server!</p>');
})
    .listen(443, () => {
        console.log('443번 포트에서 서버 대기 중입니다!');
    });
```

3.5 cluster

1. cluster

- 기본적으로 싱글 스레드인 노드가 CPU 코어를 모두 사용할 수 있게 해주는 모듈
  - 포트를 공유하는 노드 프로세스를 여러 개 둘 수 있음
  - 요청이 많이 들어왔을 때 병렬로 실행된 서버의 개수만큼 요청이 분산되어 서버에 무리가 덜 감
  - 보통은 코어 하나만 활용하나 코어를 늘려 cluster로 코어 하나당 노드 프로세스 하나가 돌아가도록 배정 가능
  - 코어가 8개라고 해서, 성능이 8배가 되는 것은 아니지만 개선됨
  - 컴퓨터 자원(메모리, 세션 등)을 공유하지 못한다는 단점이 있지만, 별도 데이터베이스 서버를 도입해 해결 가능

2. 서버 클러스터링

마스터 프로세스와 워커 프로세스

- 마스터 프로세스는 CPU 개수만큼 워커 프로세스를 만듦(worker_threads랑 구조 비슷)
- 요청이 들어오면 워커 프로세스에 고르게 분배

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if(cluster.isMaster) {
    console.log('마스터 프로세스 아이디', process.pid);
    for(let i = 0; i < numCPUs; i += 1) {
        cluster.fork();
    }

    cluster.on('exit', (worker, code, signal) => {
        console.log('${worker.process.pid}번 워커가 종료되었습니다.');
        console.log('code', code, 'signal', signal);
    });
} else {
    http.createServer((req, res) => {
        res.write('<h1>Hello Node! ${cluster.worker.process.pid} </h1>');
        res.end('<p>Hello Cluster!</p>');
    }).listen(8080, () => console.log('${process.pid}번 워커 실행'))
}
```

3. 워커 프로세스 개수 확인하기

- 요청이 들어올 때마다 서버 종료되도록 설정
  - 실행한 컴퓨터의 코어가 8개이면 8번 요청을 받고 종료됨

```
http.createServer((req, res) => {
        res.write('<h1>Hello Node! ${cluster.worker.process.pid} </h1>');
        res.end('<p>Hello Cluster!</p>');
        setTimeout(() => { // 요청이 들어올 때마다 1초 후에 서버가 종료되도록 설정
            process.exit(1);
        }, 1000);
    }).listen(8080, () => console.log('${process.pid}번 워커 실행'))
```

4. 워커 프로세스 다시 살리기

- 워커가 죽을 때마다 새로운 워커를 생성
  - 이 방식은 좋지 않음
  - 오류 자체를 해결하지 않는 한 계속 문제가 발생
  - 하지만 서버가 종료되는 현상을 막을 수 있어 참고할 만함.

```
if(cluster.isMaster) {
    console.log('마스터 프로세스 아이디', process.pid);
    for(let i = 0; i < numCPUs; i += 1) {
        cluster.fork();
    }

    cluster.on('exit', (worker, code, signal) => {
        console.log('${worker.process.pid}번 워커가 종료되었습니다.');
        console.log('code', code, 'signal', signal);
        cluster.fork();
    });
```

3.6 npm 알아보기

1. npm이란

- Node Package Manager
  - 노드의 패키지 매니저
  - 다른 사람들이 만든 소스 코드들을 모아둔 저장소
  - 남의 코드를 사용하여 프로그래밍 가능
  - 이미 있는 기능을 다시 구현할 필요가 없어 효율적
  - 오픈 소스 생태계를 구성중

  - 패키지: npm에 업로드된 노드 모듈
  - 모듈이 다른 모듈을 사용할 수 있듯 패키지도 다른 패키지를 사용할 수 있음
  - 의존 관계라고 부름

3.7 package.json으로 패키지 관리하기

1. package.json

- 현재 프로젝트에 대한 정보와 사용 중인 패키지에 대한 정보를 담은 파일
  - 같은 패키지라도 버전별로 기능이 다를 수 있으므로 버전을 기록해두어야 함
  - 동일한 버전을 설치하지 않으면 문제가 생길 수 있음
  - 노드 프로젝트 시작 전 package.json부터 만들고 시작함 [npm init]

```
package name: (~~~) my_first_npm
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author: choi
license: (ISC)
About to write to ~~~/~~~/~~~/~~~/package.json:

{
  "name": "my_first_npm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "choi",
  "license": "ISC"
}


Is this OK? (yes)
```

2. package.json 속성들

- package name : 패키지의 이름 (name 속성)
- version : 패키지의 버전 (npm의 버전은 다소 엄격하게 관리)
- entry point : 자바스크립트 실행 파일 진입점 (main 속성)
- test command : 코드를 테스트할 때 입력할 명령어 (scripts 속성 내 test 속성)
- git repository : 코드를 저장해둔 Git 저장소 주소 (repository 속성)
- keywords : npm 공식 홈페이지에서 패키지를 쉽게 탐색 (keywords 속성에 저장)
- license : 해당 패키지의 라이선스를 넣어주면 됩니다.

3. npm 스크립트

- npm init이 완료되면 폴더에 package.json이 생성됨

```
{
  "name": "my_first_npm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "choi",
  "license": "ISC"
}
```

- npm run [스크립트명]으로 스크립트 실행

```
npm run test
```

4. 패키지 설치하기

- express 설치하기

```
npm install express
```

- package.json에 기록됨(dependencies에 express 이름과 버전 추가됨)

package.json

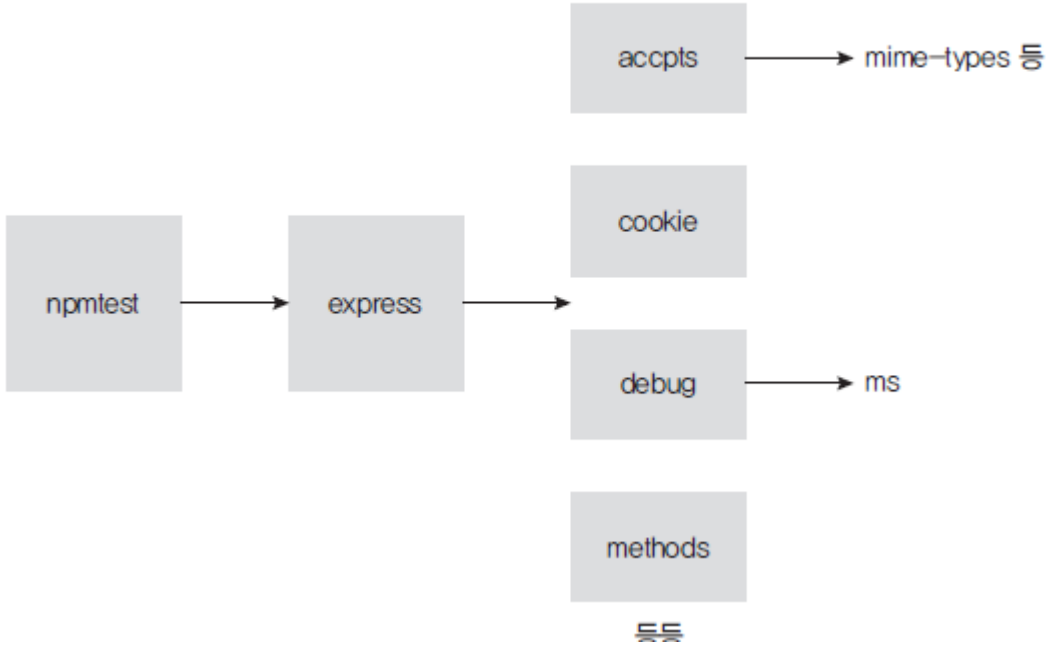
```
{
  "name": "npmtest",
  ...
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1",
```

5. node_modules

- npm install 시 node_modules 폴더 생성
  - 내부에 설치한 패키지들이 들어 있음
  - express 외에도 express와 의존 관계가 있는 패키지들이 모두 설치됨
- package-lock.json도 생성되어 패키지 간 의존 관계를 명확하게 표시함

그림 5-3 node_modules 폴더 안의 패키지

그림 5-4 의존 관계



6. 여러 패키지 동시 설치하기

- npm install 패키지1 패키지2 패키지3 …

콘솔

```
$ npm install morgan cookie-parser express-session
npm WARN npmtest@0.0.1 No repository field.

+ morgan@1.10.0
+ express-session@1.17.1
+ cookie-parser@1.4.5
added 8 packages from 5 contributors in 3.115s
```

```
"dependencies": {
"body-parser": "^1.20.2",
"cookie-parser": "^1.4.6",
"express": "^4.18.3",
"express-session": "^1.18.0",
"morgan": "^1.10.0"
}
```

7. 개발용 패키지

- npm install --save-dev 패키지명 또는 npm i -D 패키지명
  - devDependencies에 추가됨

콘솔

```
$ npm install --save-dev nodemon
+ nodemon@1.17.3
added 227 packages from 134 contributors in 24.735s
```

```
"devDependencies": {
"nodemon": "^3.1.0"
}
```

8. Peer Dependencies

- A 라이브러리의 peerDependencies가 다음과 같은 경우
  - A 라이브러리는 jQuery@3이 설치되었다고 간주하고 코딩한 것

package.json

```
{
  ...
  "peerDependencies": {
    "jQuery": "^3.0.0"
  }
}
```

- ERESOLVE unable to resolve dependency tree
  - npm i --force로 모든 버전 설치하기
  - npm i --legacy-peer-deps로 peerDependencies 무시하기

9. npm ci

- npm i를 할 때마다 package.json과 package-lock.json이 변할 수 있음
  - 배포 시에는 npm ci로 배포하기
- node_modules는 git 같은 버전 관리 시스템에 커밋할 필요가 없음
  - npm i나 npm ci를 하면 동일하게 복구됨

10. 글로벌(전역) 패키지

- npm install --global 패키지명 또는 npm i -g 패키지명
  - 모든 프로젝트와 콘솔에서 패키지를 사용할 수 있음
  - 예제는 rm -rf(리눅스의 삭제 명령)를 흉내내는 rimraf 패키지의 글로벌 설치
  - npx로 글로벌 설치 없이 글로벌 명령어 사용 가능

콘솔

```
$ npm install --global rimraf
C:\Users\zerocho\AppData\Roaming\npm\rimraf -> C:\Users\zerocho\AppData\Roaming\npm\
node_modules\rimraf\bin.js

+ rimraf@3.0.2
added 12 packages from 4 contributors in 1.015s
```

콘솔

```
$ rimraf node_modules
```

콘솔

```
$ npm install --save-dev rimraf
$ npx rimraf node_modules
```

3.8 패키지 버전 이해하기

1. SemVer 버저닝

- 노드 패키지의 버전은 SemVer(유의적 버저닝) 방식을 따름
  - Major(주 버전), Minor(부 버전), Patch(수 버전)

  - 노드에서는 배포를 할 때 항상 버전을 올려야 함
  - Major는 하위 버전과 호환되지 않은 수정 사항이 생겼을 때 올림
  - Minor는 하위 버전과 호환되는 수정 사항이 생겼을 때 올림
  - Patch는 기능에 버그를 해결했을 때 올림

그림 5-5 SemVer

2. 버전 기호 사용하기

- 버전 앞에 기호를 붙여 의미를 더함
  - ^1.1.1: 패키지 업데이트 시 minor 버전까지만 업데이트 됨(2.0.0버전은 안 됨)
  - ~1.1.1: 패키지 업데이트 시 patch버전까지만 업데이트 됨(1.2.0버전은 안 됨)
  - >=, <=, >, <는 이상, 이하, 초과, 미만.
  - @latest는 최신 버전을 설치하라는 의미
  - 실험적인 버전이 존재한다면 @next로 실험적인 버전 설치 가능(불안정함)
  - 각 버전마다 부가적으로 알파/베타/RC 버전이 존재할 수도 있음(1.1.1-alpha.0, 2.0.0-beta.1, 2.0.0-rc.0)

3.9 기타 npm 명령어

1. 기타 명령어

- **npm outdated**: 어떤 패키지에 기능 변화가 생겼는지 알 수 있음

그림 5-6 npm outdated

Package	Current	Wanted	Latest	Location
nodemon	1.19.4	1.19.4	2.0.1	npmtest
rimraf	2.6.3	2.7.1	3.0.0	npmtest

- **npm update**: package.json에 따라 패키지 업데이트
- **npm uninstall 패키지명**: 패키지 삭제(npm rm 패키지명으로도 가능)
- **npm search 검색어**: npm 패키지를 검색할 수 있음(npmjs.com에서도 가능)
- **npm info 패키지명**: 패키지의 세부 정보 파악 가능
- **npm ls** : 현재 패키지 리스트 확인
- **npm login**: npm에 로그인을 하기 위한 명령어(npmjs.com에서 회원가입 필요)
- **npm whoami**: 현재 사용자가 누구인지 알려줌
- **npm logout**: 로그인한 계정을 로그아웃

2. 기타 명령어

- **npm version 버전**: package.json의 버전을 올림(Git에 커밋도 함)

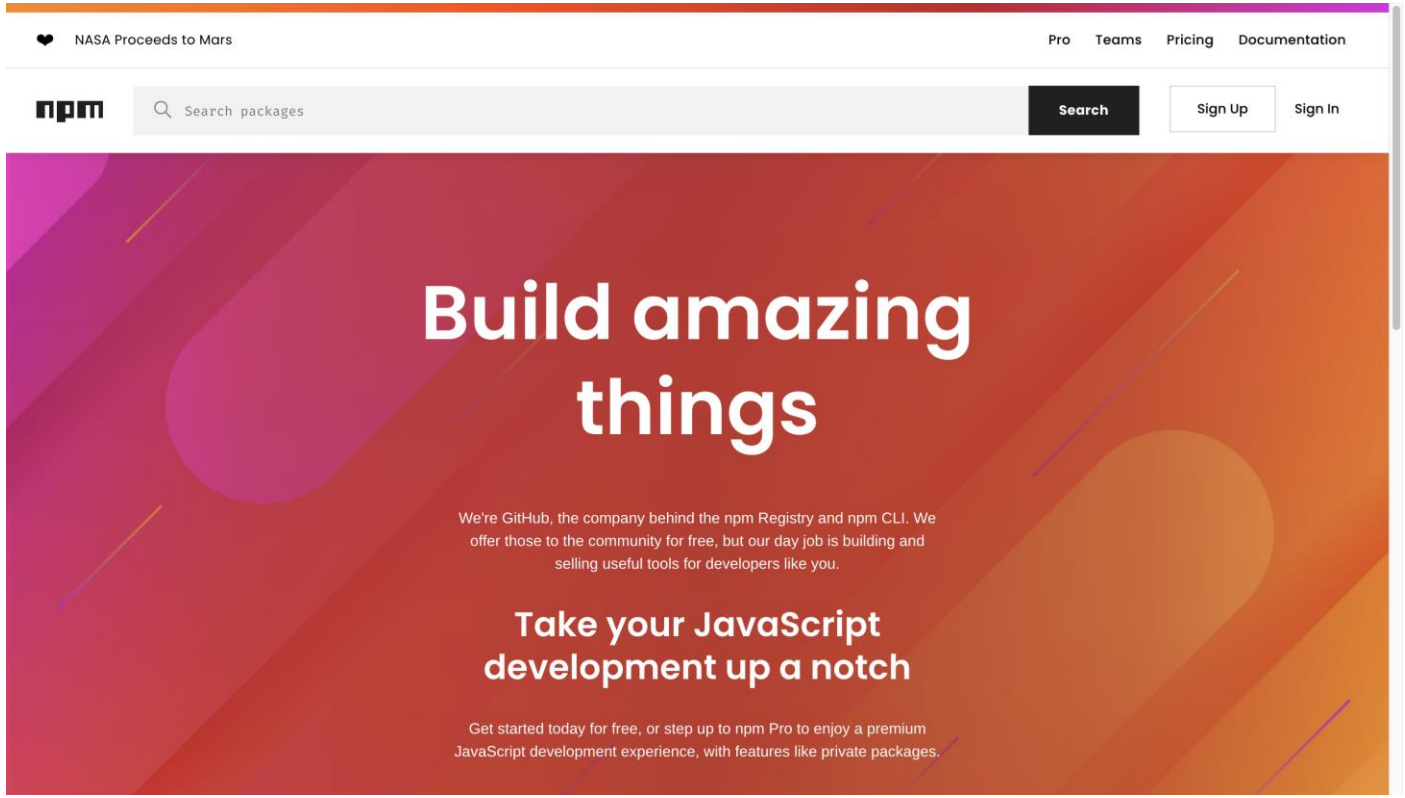
```
npm version 5.3.2, npm version minor
```

- **npm deprecate [패키지명][버전] [메시지]**: 패키지를 설치할 때 경고 메시지를 띄우게 함(오류가 있는 패키지에 적용)
- **npm publish**: 자신이 만든 패키지를 배포
- **npm unpublish --force**: 자신이 만든 패키지를 배포 중단 (배포 후 72시간 내에만 가능)
  - 다른 사람이 내 패키지를 사용하고 있는데 배포가 중단되면 문제가 생기기 때문
- 기타 명령어는 https://docs.npmjs.com의 CLI Commands에서 확인

3.10 패키지 배포하기

1. npm 회원가입

- npmjs.com에 접속해서 회원가입



2. 배포할 패키지 작성

- package.json과 main 부분과 배포할 파일 경로명이 일치해야 함
  - “main”: “index.js”

index.js

```
module.exports = () => {
  return 'hello package';
};
```

3. 배포 시도하기

npm publish 입력

콘솔

```
$ npm publish
npm notice
// notice 생략
npm ERR! code E403
npm ERR! 403 403 Forbidden - PUT https://registry.npmjs.org/npmtest - You do not have
permission to publish "npmtest". Are you logged in as the correct user?
npm ERR! 403 In most cases, you or one of your dependencies are requesting
npm ERR! 403 a package version that is forbidden by your security policy.

npm ERR! A complete log of this run can be found in:
npm ERR!     C:\Users\speak\AppData\Roaming\npm-cache\_logs\2020-04-
24T05_52_25_852Z-debug.log
```

npmtest란 이름을 누가 사용중

- 이름이 겹치면 안 되므로 다른 것으로 바꿔서 배포

4. 배포 시도하기

- 이름을 변경한 후 npm publish 입력

```
npm publish
```

```
npm info 이름
```

5. 버전 올리기

- 버전을 올리지 않으면 E403 에러
- npm version 명령어로 버전 올리기

npm version patch

npm publish

6. 배포 취소하기

- 24시간 내에 npm unpublish 패키지명 입력

```
npm unpublish 이름 --force
```