Node.js 교과서 개정2판

길벗

4장

4.1 요청과 응답 이해하기

4.2 REST API와 라우팅

4.3 쿠키와 세션 이해하기

4.4 https와 http2

4.5 cluster

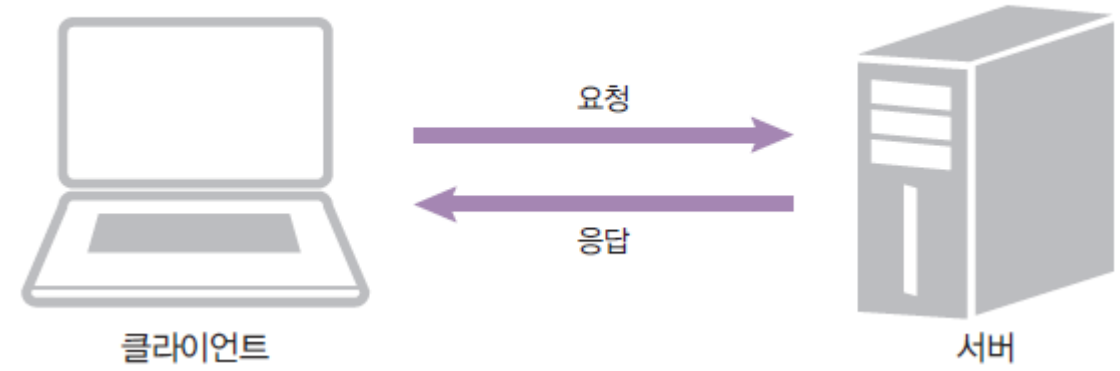
4.1 요청과 응답 이해하기

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

createServer.js

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

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

3. 8080 포트에 연결하기

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

server1.js

```
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('8080번 포트에서 서버 대기 중입니다!');
  });
```

4. 8080 포트로 접속하기

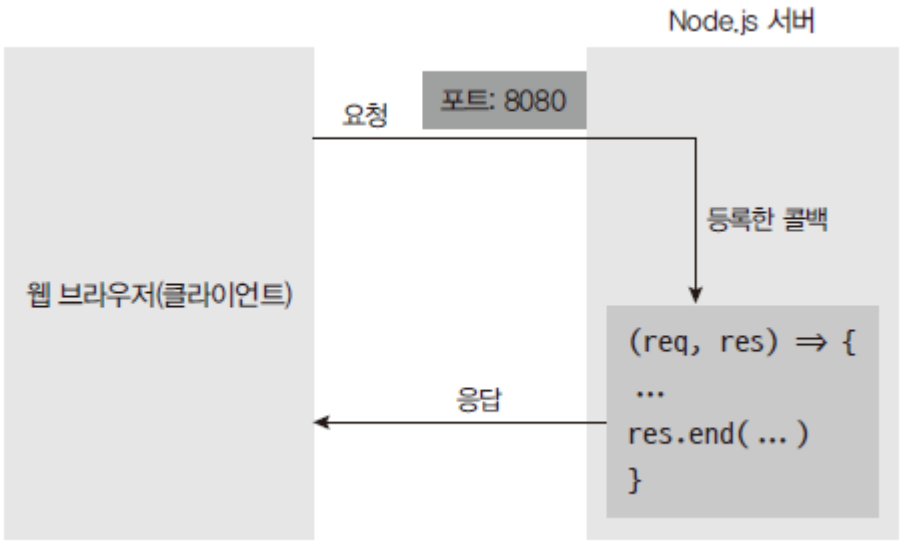
» 스크립트를 실행하면 8080 포트에 연결됨

콘솔

```
$ node server1
8080번 포트에서 서버 대기 중입니다!
```

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

그림 4-2 서버 실행 화면

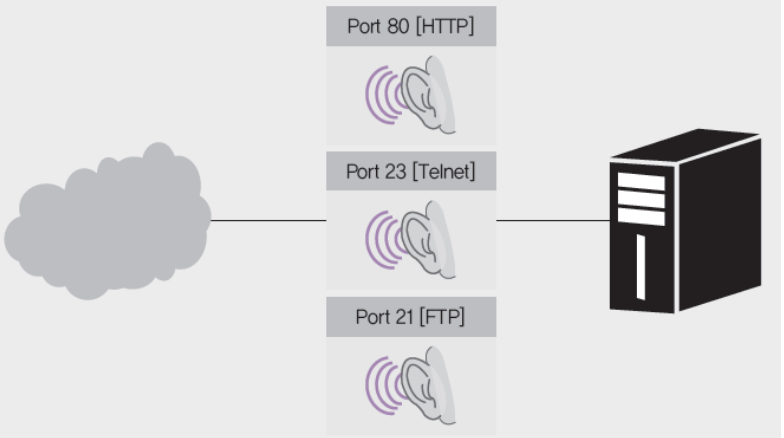


5. localhost와 포트

- localhost는 컴퓨터 내부 주소
  - 외부에서는 접근 불가능

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

그림 4-4 IP와 포트



6. 이벤트 리스너 붙이기

» listening과 error 이벤트를 붙일 수 있음.

server1-1.js

```
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', () => {
  console.log('8080번 포트에서 서버 대기 중입니다!');
});
server.on('error', (error) => {
  console.error(error);
});
```

7. 한 번에 여러 개의 서버 실행하기

» createServer를 여러 번 호출하면 됨.

- 단, 두 서버의 포트를 다르게 지정해야 함.
- 같게 지정하면 EADDRINUSE 에러 발생

server1-2.js

```
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('8080번 포트에서 서버 대기 중입니다!');
  });

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(8081, () => { // 서버 연결
    console.log('8081번 포트에서 서버 대기 중입니다!');
  });
```

8. html 읽어서 전송하기

» write와 end에 문자열을 넣는 것은 비효율적

- fs 모듈로 html을 읽어서 전송하자
- write가 버퍼도 전송 가능

server2.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Node.js 웹 서버</title>
</head>
<body>
    <h1>Node.js 웹 서버</h1>
    <p>만들 준비되셨나요?</p>
</body>
</html>
```

server2.js

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

http.createServer(async (req, res) => {
  try {
    const data = await fs.readFile('./server2.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(8081, () => {
    console.log('8081번 포트에서 서버 대기 중입니다!');
  });
```

9. server2 실행하기

» 포트 번호를 8081로 바꿈

- server1.js를 종료했다면 8080번 포트를 계속 써도 됨
- 종료하지 않은 경우 같은 포트를 쓰면 충돌이 나 에러 발생

콘솔

```
$ node server2
8081번 포트에서 서버 대기 중입니다!
```

Node.js 웹 서버

만들 준비되셨나요?

4.2 REST API와 라우팅

1. REST API

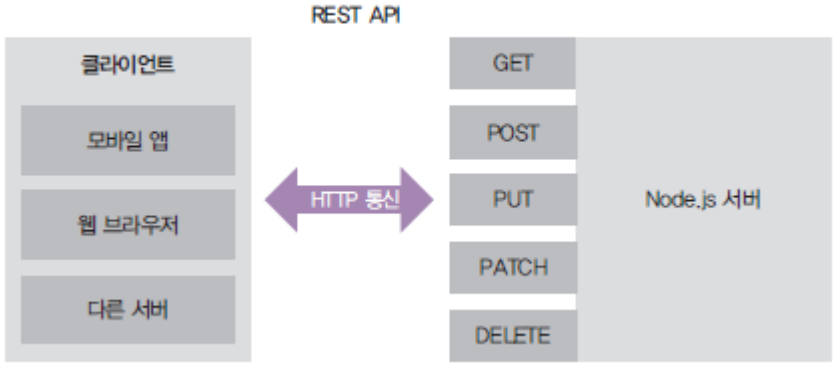
- 서버에 요청을 보낼 때는 주소를 통해 요청의 내용을 표현
  - /index.html이면 index.html을 보내달라는 뜻
  - 항상 html을 요구할 필요는 없음
  - 서버가 이해하기 쉬운 주소가 좋음
- REST API(Representational State Transfer)
  - 서버의 자원을 정의하고 자원에 대한 주소를 지정하는 방법
  - /user이면 사용자 정보에 관한 정보를 요청하는 것
  - /post면 게시글에 관련된 자원을 요청하는 것
- HTTP 요청 메서드
  - GET: 서버 자원을 가져오려고 할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



» 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 만들기

- GitHub 저장소(https://github.com/zerocho/nodejsbook) ch4 소스 참조
- restServer.js에 주목
  - GET 메서드에서 /, /about 요청 주소는 페이지를 요청하는 것이므로 HTML 파일을 읽어서 전송합니다. AJAX 요청을 처리하는 /users에서는 users 데이터를 전송합니다. JSON 형식으로 보내기 위해 JSON.stringify를 해주었습니다. 그 외의 GET 요청은 CSS나 JS 파일을 요청하는 것이므로 찾아서 보내주고, 없다면 404 NOT FOUND 에러를 응답합니다.
  - POST와 PUT 메서드는 클라이언트로부터 데이터를 받으므로 특별한 처리가 필요합니다. req.on('data', 콜백)과 req.on('end', 콜백) 부분인데요. 3.6.2절의 버퍼와 스트림에서 배웠던 readStream입니다. readStream으로 요청과 같이 들어오는 요청 본문을 받을 수 있습니다. 단, 문자열이므로 JSON으로 만드는 JSON.parse 과정이 한 번 필요합니다.
  - DELETE 메서드로 요청이 오면 주소에 들어 있는 키에 해당하는 사용자를 제거합니다.
  - 해당하는 주소가 없을 경우 404 NOT FOUND 에러를 응답합니다.

4. REST 서버 실행하기

localhost:8085에 접속

콘솔

```
$ node restServer
8085번 포트에서 서버 대기 중입니다.
```

그림 4-7 Home 클릭 시

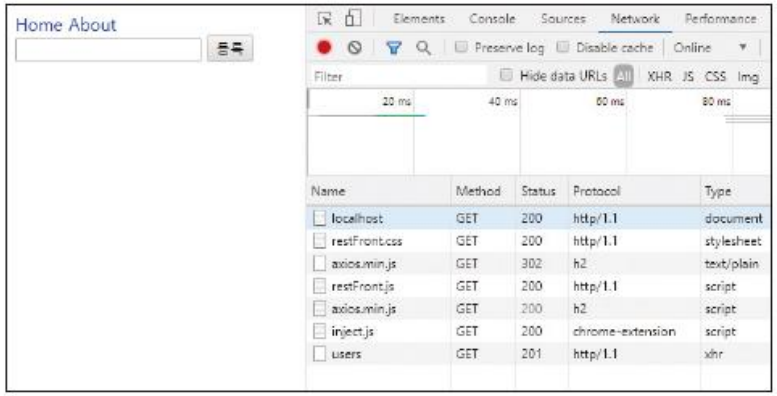
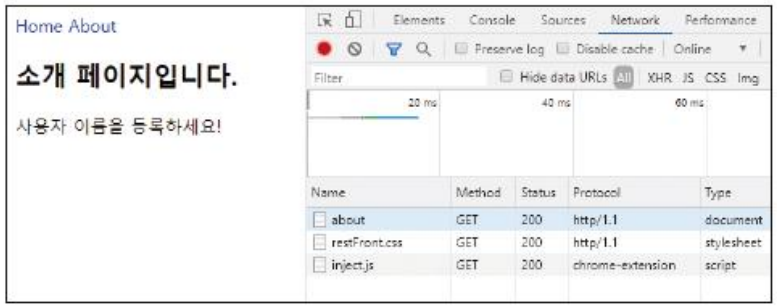


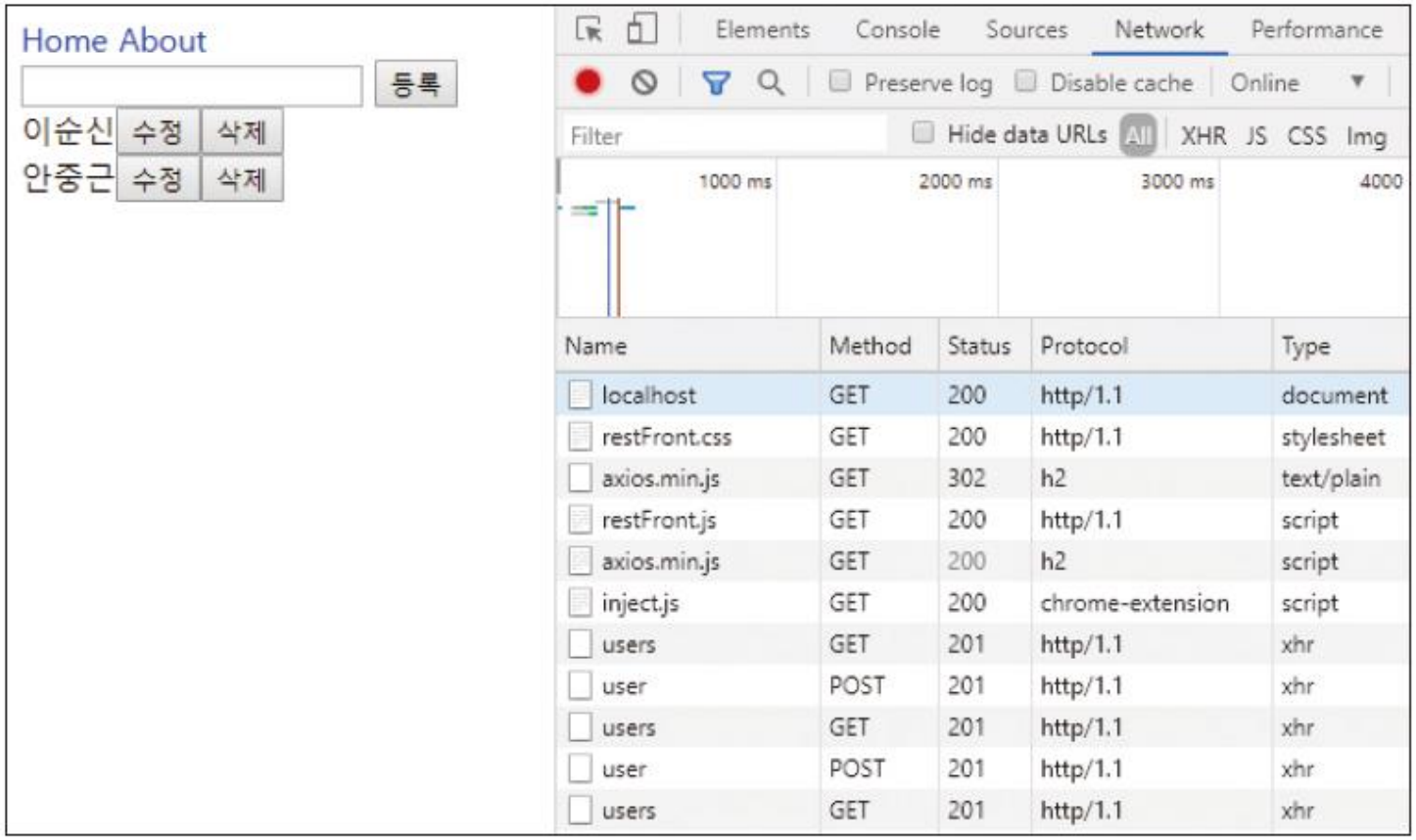
그림 4-8 About 클릭 시



5. REST 요청 확인하기

» 개발자도구(F12) Network 탭에서 요청 내용 실시간 확인 가능

- Name은 요청 주소, Method는 요청 메서드, Status는 HTTP 응답 코드
- Protocol은 HTTP 프로토콜, Type은 요청 종류(xhr은 AJAX 요청)

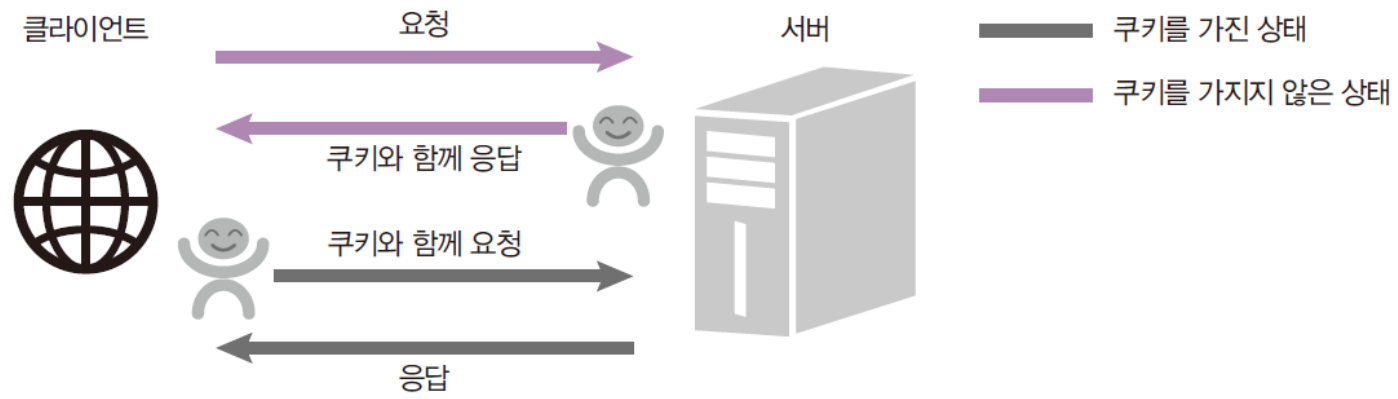


4.3 쿠키와 세션 이해하기

1. 쿠키의 필요성

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

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=zerocho
  - 매 요청마다 서버에 동봉해서 보냄

cookie.js

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

http.createServer((req, res) => {
  console.log(req.url, req.headers.cookie);
  res.writeHead(200, { 'Set-Cookie': 'mycookie=test' });
  res.end('Hello Cookie');
})
  .listen(8083, () => {
    console.log('8083번 포트에서 서버 대기 중입니다!');
  });
```

콘솔

```
$ node cookie
8083번 포트에서 서버 대기 중입니다!
```

3. 쿠키 서버 실행하기

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

콘솔

```
$ node server3
8082번 포트에서 서버 대기 중입니다!
```

- localhost:8082에 접속
  - 요청이 전송되고 응답이 왔을 때 쿠키가 설정됨
  - favicon.ico는 브라우저가 자동으로 보내는 요청
  - 두 번째 요청인 favicon.ico에 쿠키가 넣어짐

콘솔

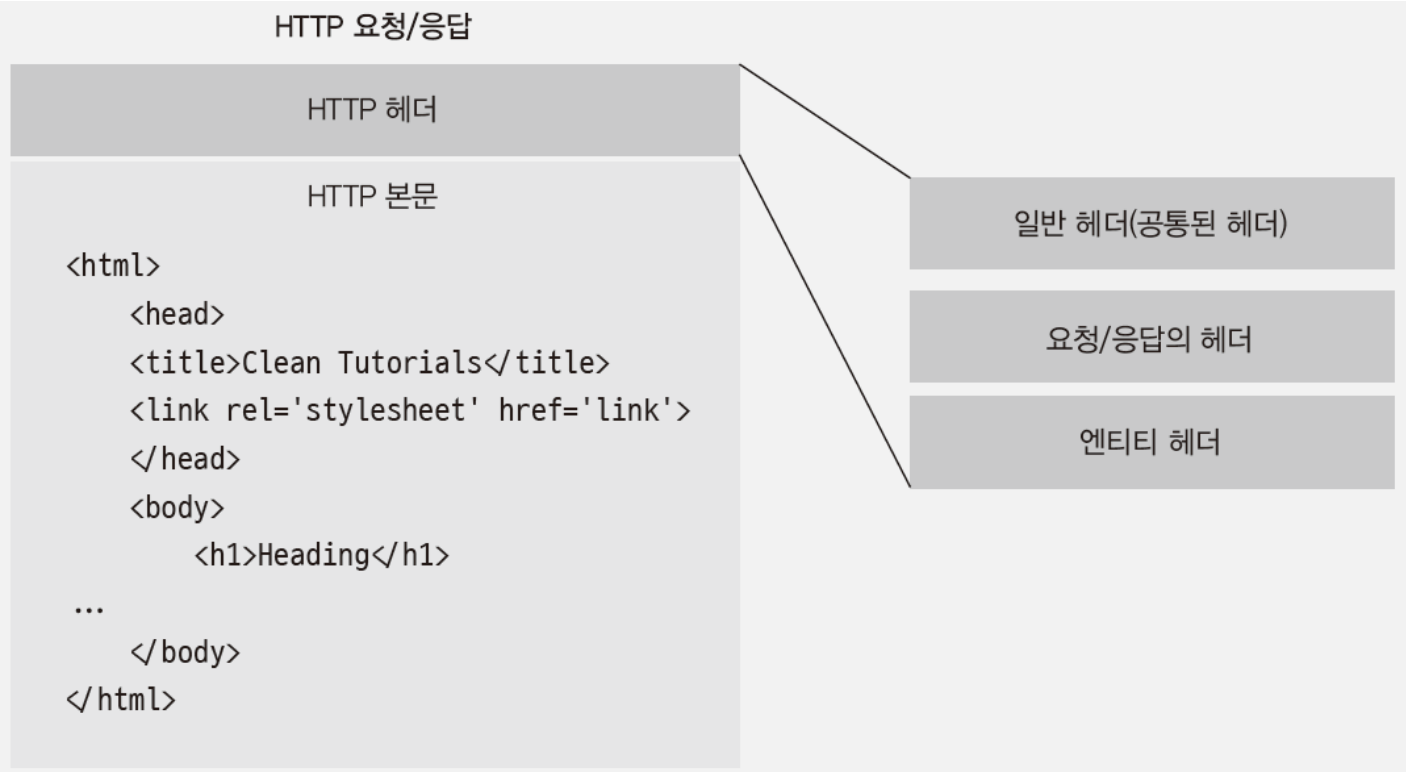
```
/ undefined
/favicon.ico { mycookie: 'test' }
```

그림 4-14 파비콘

4. 헤더와 본문

» http 요청과 응답은 헤더와 본문을 가짐

- 헤더는 요청 또는 응답에 대한 정보를 가짐
- 본문은 주고받는 실제 데이터
- 쿠키는 부가적인 정보이므로 헤더에 저장



5. http 상태 코드

- writeHead 메서드에 첫 번째 인수로 넣은 값
  - 요청이 성공했는지 실패했는지를 알려줌
  - 2XX: 성공을 알리는 상태 코드입니다. 대표적으로 200(성공), 201(작성됨)이 많이 사용됩니다.
  - 3XX: 리다이렉션(다른 페이지로 이동)을 알리는 상태 코드입니다. 어떤 주소를 입력했는데 다른 주소의 페이지로 넘어갈 때 이 코드가 사용됩니다. 대표적으로 301(영구 이동), 302(임시 이동)가 있습니다.
  - 4XX: 요청 오류를 나타냅니다. 요청 자체에 오류가 있을 때 표시됩니다. 대표적으로 401(권한 없음), 403(금지됨), 404(찾을 수 없음)가 있습니다.
  - 5XX: 서버 오류를 나타냅니다. 요청은 제대로 왔지만 서버에 오류가 생겼을 때 발생합니다. 이 오류가 뜨지 않게 주의해서 프로그래밍해야 합니다. 이 오류를 클라이언트로 res.writeHead로 직접 보내는 경우는 없고, 예기치 못한 에러 발생 시 서버가 알아서 5XX대 코드를 보냅니다. 500(내부 서버 오류), 502(불량 게이트웨이), 503(서비스를 사용할 수 없음)이 자주 사용됩니다.

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

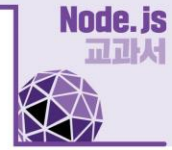
cookie2.html

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>쿠키&세션 이해하기</title>
</head>
<body>
<form action="/login">
    <input id="name" name="name" placeholder="이름을 입력하세요" />
    <button id="login">로그인</button>
</form>
</body>
</html>
```

cookie2.js

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

const parseCookies = (cookie = '') =>
  cookie
    .split(';')
    .map(v => v.split('='))
    .reduce((acc, [k, v]) => {
      acc[k.trim()] = decodeURIComponent(v);
      return acc;
    }, {});

http.createServer(async (req, res) => {
  const cookies = parseCookies(req.headers.cookie);

  // 주소가 /login으로 시작하는 경우
  if (req.url.startsWith('/login')) {
    const { query } = url.parse(req.url);
    const { name } = qs.parse(query);
    const expires = new Date();
    // 쿠키 유효 시간을 현재 시간 + 5분으로 설정
    expires.setMinutes(expires.getMinutes() + 5);
    res.writeHead(302, {
      Location: '/',
      'Set-Cookie': `name=${encodeURIComponent(name)}; Expires=
➥ ${expires.toGMTString()}; HttpOnly; Path=/`,
    });
    res.end();

  // name이라는 쿠키가 있는 경우
  } else if (cookies.name) {
    res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
    res.end(`${cookies.name}님 안녕하세요`);
  } else {
    try {
      const data = await fs.readFile('./cookie2.html');
      res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
      res.end(data);
    } catch (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain; charset=utf-8' });
      res.end(err.message);
    }
  }

})
  .listen(8084, () => {
    console.log('8084번 포트에서 서버 대기 중입니다!');
  });
```

7. 쿠키 옵션



- Set-Cookie 시 다양한 옵션이 있음
  - 쿠키명=쿠키값: 기본적인 쿠키의 값입니다. mycookie=test 또는 name=zerocho 같이 설정합니다.
  - Expires=날짜: 만료 기한입니다. 이 기한이 지나면 쿠키가 제거됩니다. 기본값은 클라이언트가 종료될 때까지입니다.
  - Max-age=초: Expires와 비슷하지만 날짜 대신 초를 입력할 수 있습니다. 해당 초가 지나면 쿠기가 제거됩니다. Expires보다 우선합니다.
  - Domain=도메인명: 쿠키가 전송될 도메인을 특정할 수 있습니다. 기본값은 현재 도메인입니다.
  - Path=URL: 쿠키가 전송될 URL을 특정할 수 있습니다. 기본값은 '/'이고 이 경우 모든 URL에서 쿠키를 전송할 수 있습니다.
  - Secure: HTTPS일 경우에만 쿠키가 전송됩니다.
  - HttpOnly: 설정 시 자바스크립트에서 쿠키에 접근할 수 없습니다. 쿠키 조작을 방지하기 위해 설정하는 것이 좋습니다.

8. 쿠키 서버 실행하기

콘솔

```
$ node cookie2
8084번 포트에서 서버 대기 중입니다!
```

- localhost:8084 포트에 접속
  - Application 탭(F12) 열기
  - 로그인을 하면 쿠키가 생성됨

그림 4-15 로그인 이전

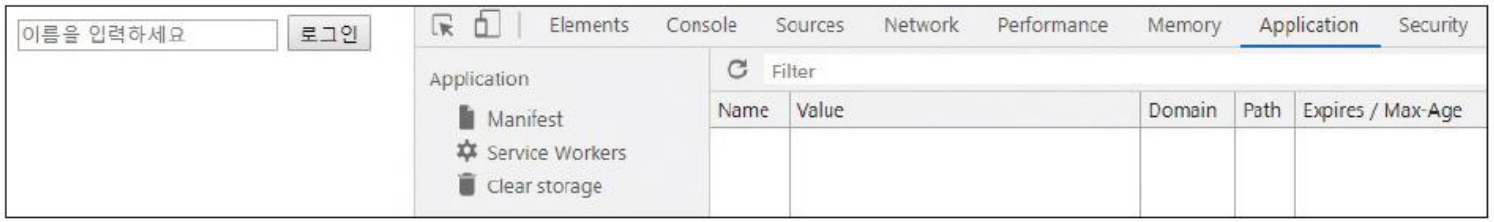
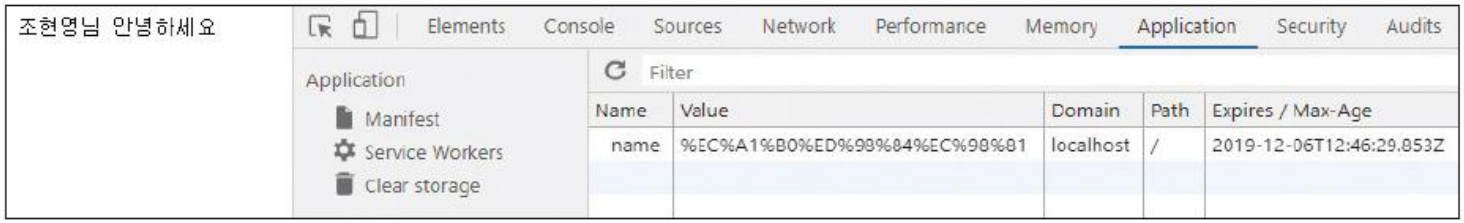


그림 4-16 로그인 이후



9. 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

session.js

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

const parseCookies = (cookie = '') =>
  cookie
    .split(';')
    .map(v => v.split('='))
    .reduce((acc, [k, v]) => {
      acc[k.trim()] = decodeURIComponent(v);
      return acc;
    }, {});

const session = {};

http.createServer(async (req, res) => {
  const cookies = parseCookies(req.headers.cookie);
  if (req.url.startsWith('/login')) {
    const { query } = url.parse(req.url);
    const { name } = qs.parse(query);
    const expires = new Date();
    expires.setMinutes(expires.getMinutes() + 5);
    const uniqueInt = Date.now();
    session[uniqueInt] = {
      name,
      expires,
    };
      res.writeHead(302, {
        Location: '/',
        'Set-Cookie': `session=${uniqueInt}; Expires=${expires.toGMTString()};
➥ HttpOnly; Path=/`,
      });
      res.end();
    // 세션 쿠키가 존재하고, 만료 기간이 지나지 않았다면
    } else if (cookies.session && session[cookies.session].expires > new Date()) {
      res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
      res.end(`${session[cookies.session].name}님 안녕하세요`);
    } else {
      try {
        const data = await fs.readFile('./cookie2.html');
        res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
        res.end(data);
      } catch (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain; charset=utf-8' });
        res.end(err.message);
      }
    }
  })
    .listen(8085, () => {
      console.log('8085번 포트에서 서버 대기 중입니다!');
    });
```

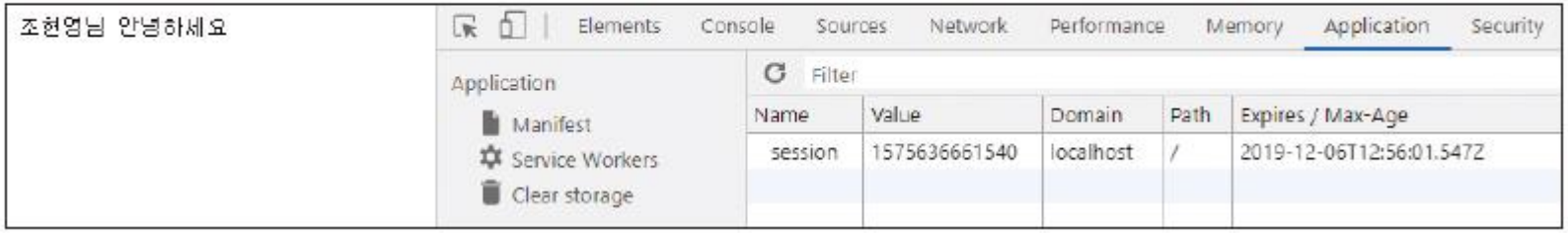
10. 세션 서버 실행하기

» localhost:8085

콘솔

```
$ node session
8085번 포트에서 서버 대기 중입니다!
```

그림 4-17 로그인 이후



» 실 서버에서는 세션을 직접 구현하지 말자

- 6장에서 나오는 express-session 사용하기

4.4 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)

그림 4-18 https 적용 화면

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

server1.js

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

http.createServer((req, res) => {
  res.writeHead(500, { '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('8080번 포트에서 서버 대기 중입니다!');
  });
```

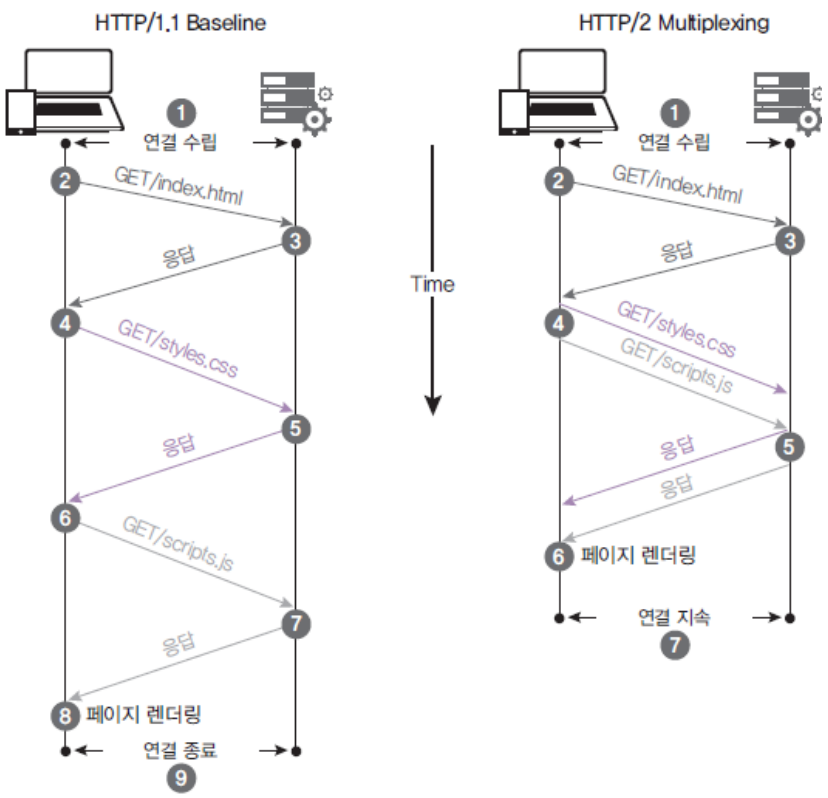
server1-3.js

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

https.createServer({
  cert: fs.readFileSync('도메인 인증서 경로'),
  key: fs.readFileSync('도메인 비밀키 경로'),
  ca: [
    fs.readFileSync('상위 인증서 경로'),
    fs.readFileSync('상위 인증서 경로'),
  ],
}, (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

그림 4-20 http/1.1과 http/2의 비교



4. http2 적용 서버

» https 모듈을 http2로, createServer 메서드를 createSecureServer 메서드로

server1-4.js

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

http2.createSecureServer({
  cert: fs.readFileSync('도메인 인증서 경로'),
  key: fs.readFileSync('도메인 비밀키 경로'),
  ca: [
    fs.readFileSync('상위 인증서 경로'),
    fs.readFileSync('상위 인증서 경로'),
  ],
}, (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!');
  });
```

4.5 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 분산됨
  - 서버에 무리가 덜 감
  - 코어가 8개인 서버가 있을 때: 보통은 코어 하나만 활용
  - cluster로 코어 하나당 노드 프로세스 하나를 배정 가능
  - 성능이 8배가 되는 것은 아니지만 개선됨
  - 단점: 컴퓨터 자원(메모리, 세션 등) 공유 못 함
  - Redis 등 별도 서버로 해결

2. 서버 클러스터링

마스터 프로세스와 워커 프로세스

- 마스터 프로세스는 CPU 개수만큼 워커 프로세스를 만듦(worker_threads랑 구조 비슷)
- cluster.js 분배

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

if (cluster.isMaster) {
  console.log(`마스터 프로세스 아이디: ${process.pid}`);
  // CPU 개수만큼 워커를 생산
  for (let i = 0; i < numCPUs; i += 1) {
    cluster.fork();
  }
  // 워커가 종료되었을 때
  cluster.on('exit', (worker, code, signal) => {
    console.log(`${worker.process.pid}번 워커가 종료되었습니다.`);
    console.log('code', code, 'signal', signal);
  });
} else {
  // 워커들이 포트에서 대기
  http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
    res.write('<h1>Hello Node!</h1>');
    res.end('<p>Hello Cluster!</p>');
  }).listen(8086);

  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

cluster.js

```
...
} else {
  // 워커들이 포트에서 대기
  http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
    res.write('<h1>Hello Node!</h1>');
    res.end('<p>Hello Cluster!</p>');
    setTimeout(() => { // 워커가 존재하는지 확인하기 위해 1초마다 강제 종료
      process.exit(1);
    }, 1000);
  }).listen(8086);

  console.log(`${process.pid}번 워커 실행`);
}
```

콘솔

```
$ node cluster
마스터 프로세스 아이디: 21360
7368번 워커 실행
11040번 워커 실행
9004번 워커 실행
16452번 워커 실행
17272번 워커 실행
16136번 워커 실행
6836번 워커 실행
15532번 워커 실행
```

콘솔

```
16136번 워커가 종료되었습니다.
code 1 signal null
17272번 워커가 종료되었습니다.
code 1 signal null
16452번 워커가 종료되었습니다.
code 1 signal null
9004번 워커가 종료되었습니다.
code 1 signal null
11040번 워커가 종료되었습니다.
code 1 signal null
7368번 워커가 종료되었습니다.
code 1 signal null
```

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.

cluster.js

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

콘솔

```
28592번 워커가 종료되었습니다.
code 1 signal null
10520번 워커 실행
10520번 워커가 종료되었습니다.
code 1 signal null
23248번 워커 실행
```