

ch05. 웹 서버 만들기

05-01 간단한 웹 서버 만들기

- 노드에 기본으로 들어있는 http 모듈을 사용하여 웹 서버 객체 만듦
 - createServer() 메소드
 - http server 객체 생성
 - server 객체의 listen() 사용

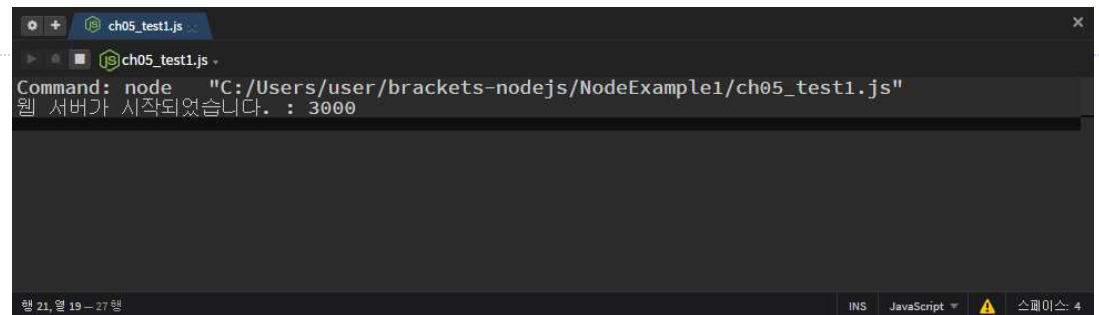
ch05_test01.js

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

```
var server = http.createServer();
```

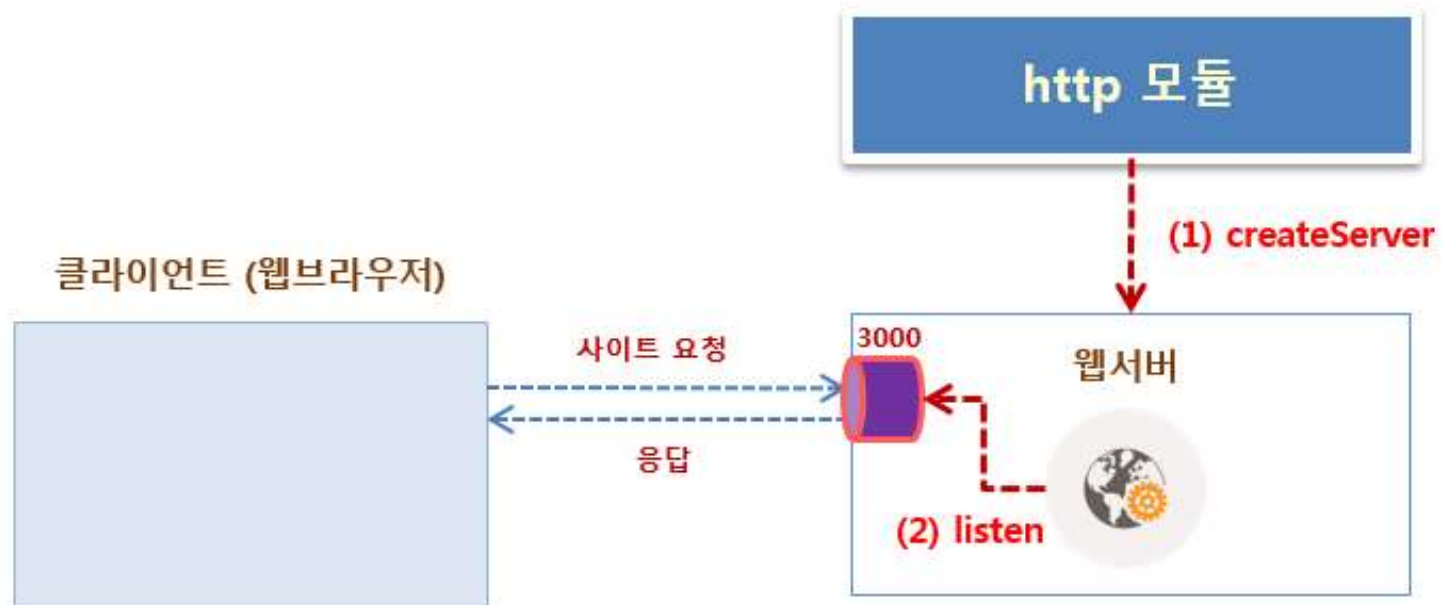
```
var port = 3000;
```

```
server.listen(port, function() {  
    console.log('웹 서버가 시작되었습니다. : %d', port);  
});
```



- `createServer()` 메소드로 웹 서버 객체 만들고 `listen()` 메소드로 대기

메소드 이름	설명
<code>listen(port[, hostname][, backlog][, callback]</code>	서버를 실행하여 대기시킵니다.
<code>close([callback])</code>	서버를 종료합니다.



- 웹 서버 실행 시 호스트와 포트 지정하는 경우

ch05_test01.js - 수정

```
var host = '192.168.0.5';  
var port = 3000;  
server.listen(port, host, '50000', function() {  
    console.log('웹 서버가 시작되었습니다. : %s, %d', host, port);  
});
```

- 웹 브라우저에서 요청할 때 상황에 따른 적절한 이벤트 발생

이벤트 이름	설명
connection	클라이언트가 접속하여 연결이 만들어질 때 발생하는 이벤트입니다.
request	클라이언트가 요청할 때 발생하는 이벤트입니다.
close	서버를 종료할 때 발생하는 이벤트입니다.

가장 많이 사용

- connection 이벤트와 request 이벤트 처리

ch05_test02.js

```
var http = require('http');
var server = http.createServer();
var port = 3000;
server.listen(port, function() {
  console.log('웹 서버가 시작되었습니다. : %d', port);
});

server.on('connection', function(socket) {
  var addr = socket.address();
  console.log('클라이언트가 접속했습니다. : %s, %d', addr.address, addr.port);
});

server.on('request', function(req, res) {
  console.log('클라이언트 요청이 들어왔습니다.');
  console.dir(req);
});

server.on('close', function() {
  console.log('서버가 종료됩니다.');
```

socket 객체

- <http://localhost:3000/> 으로 접속

```
ch05_test2.js
Command: node "C:/Users/user/brackets-nodejs/NodeExample1/ch05_test2.js"
웹서버가 시작되었습니다. : 3000
클라이언트가 접속했습니다. : ::1, 52815
클라이언트가 접속했습니다. : ::1, 52816
클라이언트 요청이 들어왔습니다.
IncomingMessage {
  _readableState:
    ReadableState {
      objectMode: false,
      highWaterMark: 16384,
      buffer: []
    }
}
```

- EADDRINUSE 에러 발생하는 경우 있음 - 기존 포트를 사용하고 있는 경우임

```
ch05_test2.js ch05_test2.js
ch05_test2.js
Command: node "C:/Users/user/brackets-nodejs/NodeExample1/ch05_test2.js"
events.js:141
    throw er; // Unhandled 'error' event
    ^
Error: listen EADDRINUSE :::3000
    at Object.exports._errnoException (util.js:907:11)
    at exports._exceptionWithHostPort (util.js:930:20)
    at Server._listen2 (net.js:1250:14)
    at listen (net.js:1286:10)
    at Server.listen (net.js:1382:5)
    at Object (C:/Users/user/brackets-nodejs/NodeExample1/ch05_test2.js:15:9)
```


- request 이벤트 처리
 - req: request 객체
 - 웹 클라이언트가 전송한 data 포함 객체
 - res: response 객체
 - 웹 서버 응답 용 객체
 - writeHead(), write(), end() 메소드로 응답 전송

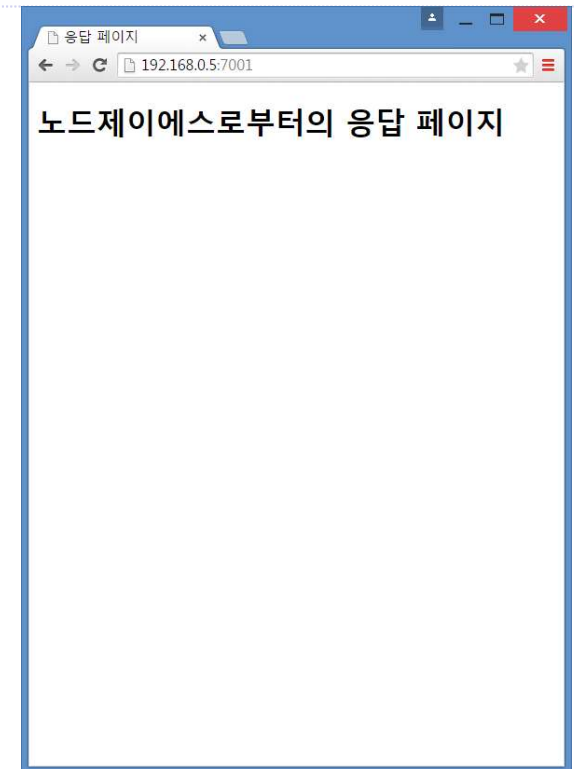
ch05_test02.js - 추가

```
server.on('request', function(req, res) {  
  console.log('클라이언트 요청이 들어왔습니다.');
```



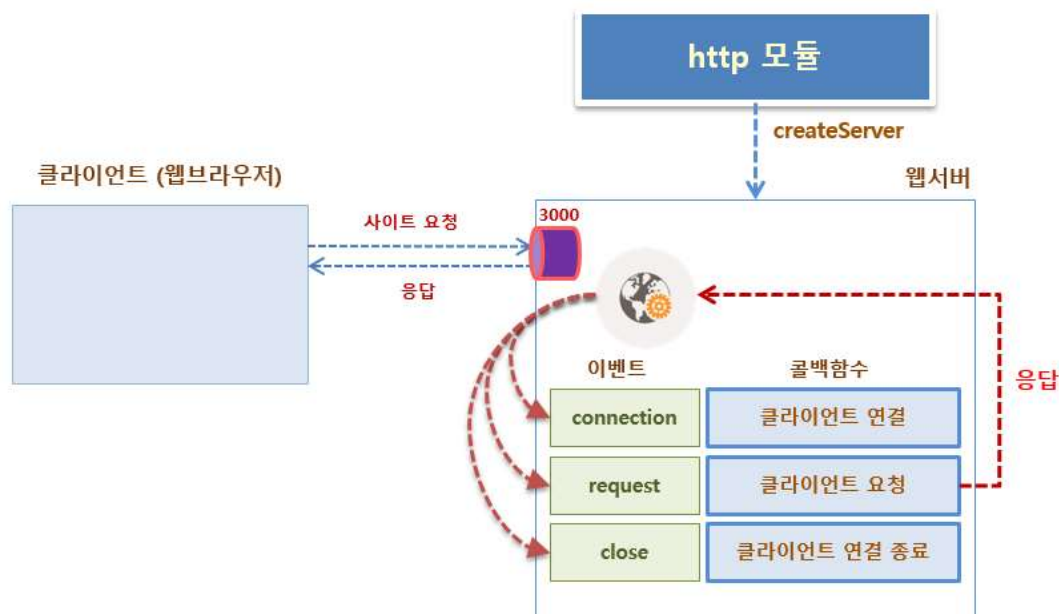
```
  res.writeHead(200, {"Content-Type": "text/html; charset=utf-8"});  
  res.write("<!DOCTYPE html>");  
  res.write("<html>");  
  res.write(" <head>");  
  res.write(" <title>응답 페이지</title>");  
  res.write(" </head>");  
  res.write(" <body>");  
  res.write(" <h1>노드제이에스로부터의 응답 페이지</h1>");  
  res.write(" </body>");  
  res.write(" </html>");  
  res.end();  
});
```

← end() 호출 시 비로소
응답 전송



- 요청을 받으면 request 이벤트가 발생되고 write() 메소드 등을 이용해 응답 전송
- response 객체 주요 메소드

메소드 이름	설명
writeHead(statusCode [, statusMessage][, headers])	응답으로 보낼 헤더를 만듭니다.
write(chunk[, encoding][, callback])	응답 본문(body) 데이터를 만듭니다. 여러 번 호출될 수 있습니다.
end([data][, encoding][, callback])	클라이언트로 응답을 전송합니다.



- 실제로 많이 사용되지는 않음 (모든 요청을 한꺼번에 처리하는 경우는 거의 없기 때문)

ch05_test04.js – test02 복사 후 추가

```
var server = http.createServer(function(req, res) {  
  console.log('클라이언트 요청이 들어왔습니다.');  
  res.writeHead(200, {"Content-Type": "text/html; charset=utf-8"});  
  res.write("<!DOCTYPE html>");  
  res.write("<html>");  
  res.write(" <head>");  
  res.write(" <title>응답 페이지</title>");  
  res.write(" </head>");  
  res.write(" <body>");  
  res.write(" <h1>노드제이에스로부터의 응답 페이지</h1>");  
  res.write(" </body>");  
  res.write("</html>");  
  res.end();  
});
```

- fs 모듈을 이용해 파일을 읽어 응답 보낼 수 있음

ch05_test05.js – test02 복사 후 추가

```
var http = require('http');
var fs = require('fs');
.....

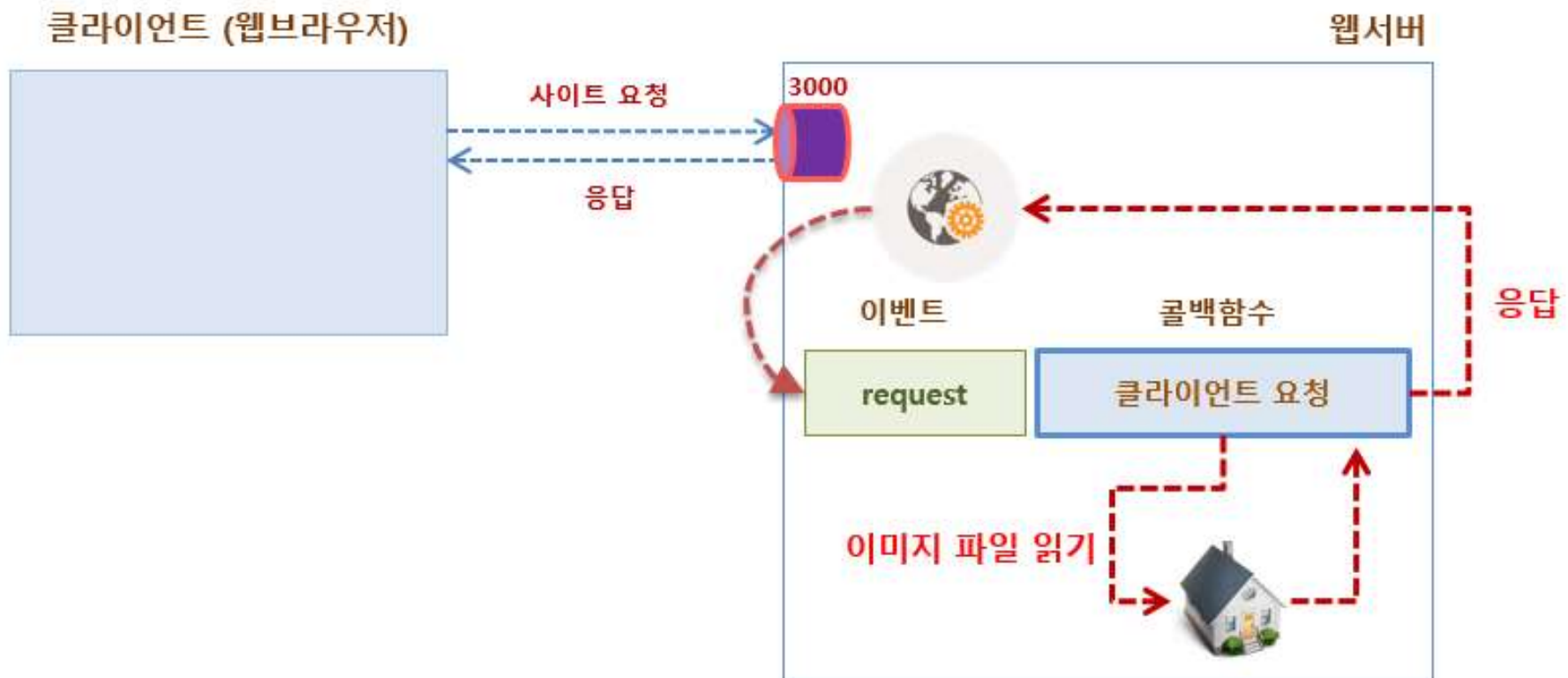
server.on('request', function(req, res) {
  console.log('클라이언트 요청이 들어왔습니다.');
```



```
  var filename = 'house.png';
  fs.readFile(filename, function(err, data) {
    res.writeHead(200, {"Content-Type": "image/png"});
    res.write(data);
    res.end();
  });
});
```



- 이미지 파일 이외에도 텍스트 파일이나 음악 파일 등을 보낼 수 있음



- 파일 읽기와 응답 객체를 스트림 파이프를 연결하여 데이터 전송
- `createReadStream` 으로 읽은 후 `pipe()` 메소드를 이용해 전송

ch05_test06.js – test05 복사 후 추가

```
server.on('request', function(req, res) {  
  console.log('클라이언트 요청이 들어왔습니다.');  
  var filename = 'house.png';  
  var infile = fs.createReadStream(filename, {flags: 'r'} );  
  
  infile.pipe(res);  
  
});
```

파일을 버퍼에 담아두고 일부분만 읽어 응답 보내기

- 파일을 버퍼 크기만큼 읽어서 응답
- 파일에서 read() 로 읽고 응답 객체의 write() 메소드를 이용해 응답 전송

ch05_test06.js -추가

```
....  
server.on('request', function(req, res){  
  var filename = 'house.png';  
  var infile = fs.createReadStream(filename, {flags:'r'});  
  var filelength = 0;  
  var curlength = 0;  
  
  fs.stat(filename, function(err, stats){  
    filelength = stats.size;  
  });  
  
  res.writeHead(200, {"Content-Type": "image/png"});
```

파일을 버퍼에 담아두고 일부분만 읽어 응답 보내기

16/75

ch05_test06.js – 계속

readable event: 스트림에서 읽을 수 있는 데이터가
있는 경우에 발생

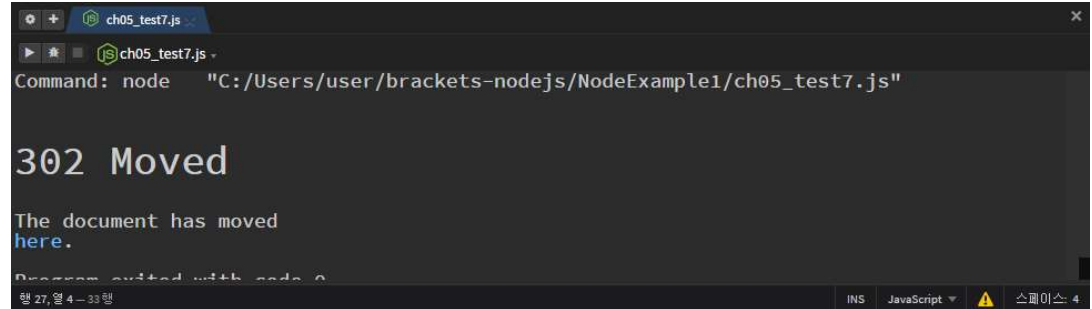
createReadStream()을 통한 스트림인 경우 read()를
위한 default buffer size는 64kb 임

```
infile.on('readable', function() {  
  var chunk;  
  while (null !== (chunk = infile.read())) {  
    console.log('읽어 들인 데이터 크기 : %d 바이트', chunk.length);  
    curlength += chunk.length;  
    res.write(chunk, 'utf8', function(err) {  
      console.log('파일 부분 쓰기 완료 : %d, 파일 크기 : %d', curlength, filelength);  
      if (curlength >= filelength) {  
        res.end();  
      }  
    });  
  }  
});  
};  
};
```

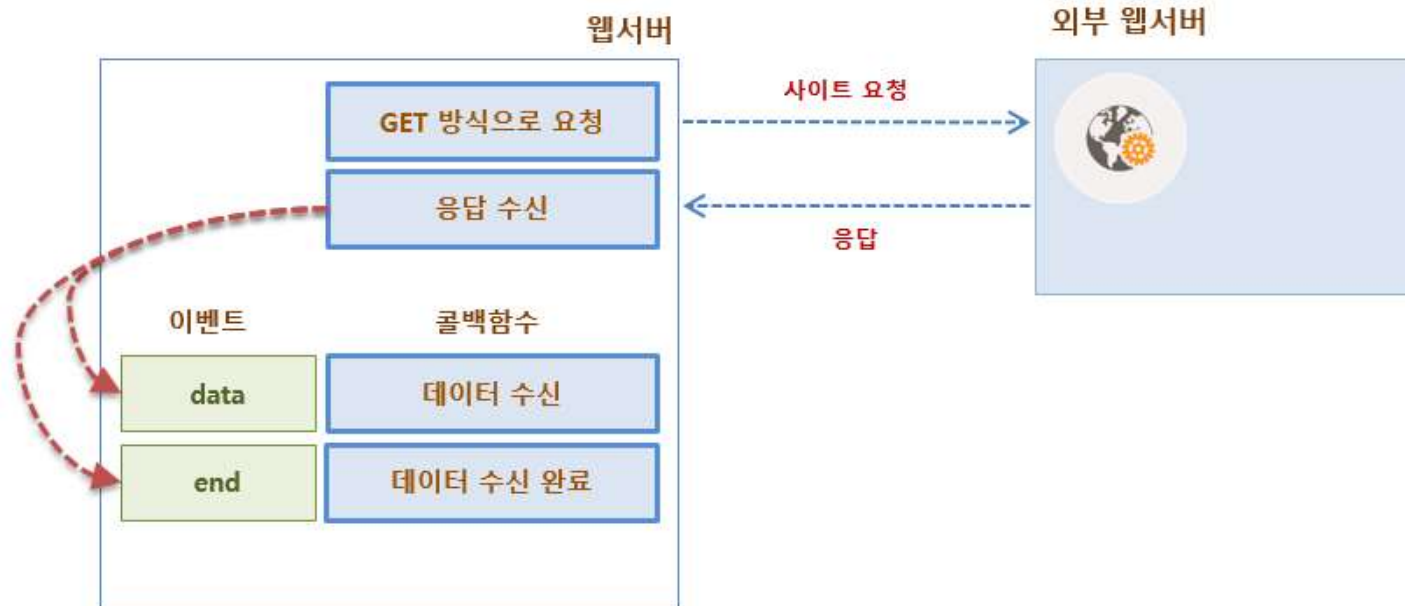

- http client 기능
 - http 모듈을 사용해 다른 웹사이트의 데이터를 가져와서 필요한 곳에 사용할 수 있음
 - GET, POST 방식 제공

ch05_test07.js

```
var http = require('http');
var options = {
  host: 'www.google.com',
  port: 80,
  path: '/'
};
var req = http.get(options, function(res) {
  var resData = '';
  res.on('data', function(chunk) {
    resData += chunk;
  });
  res.on('end', function() {
    console.log(resData);
  });
});
req.on('error', function(err) {
  console.log("에러 발생 : " + err.message);
});
```



- data이벤트
 - 다른 서버에서 응답을 받고 있을 때 발생
- end 이벤트
 - 데이터 수신이 완료되면 발생



- POST 방식으로 요청할 때는 request() 메소드 사용

ch05_test08.js

```
var http = require('http');
var opts = {
  host: 'www.google.com',
  port: 80,
  method: 'POST',
  path: '/',
  headers: {}
};

var resData = '';
var req = http.request(opts, function(res) {
  res.on('data', function(chunk) {
    resData += chunk;
  });

  res.on('end', function() {
    console.log(resData);
  });
});
```

ch05_test08.js - 계속

```
opts.headers['Content-Type'] = 'application/x-www-form-urlencoded';  
req.data = "q=actor";  
opts.headers['Content-Length'] = req.data.length;
```

} header 부분 추가 작성

```
req.on('error', function(err){  
    console.log("오류 발생:" + err.message);  
});
```

```
req.write(req.data);  
req.end();
```

} 요청 전송

05-02 익스프레스로 웹 서버 만들기

- express module
 - 웹 서버 구축 모듈
 - 미들웨어와 라우터 기능 제공
- express 예제 실습 전
 - 새로운 프로젝트 폴더를 ExpressExample로 만들
 - 파일>폴더 열기 메뉴를 눌러 프로젝트 폴더 지정
 - 명령프롬프트에서 npm init 명령 실행하여 package.json 파일 생성

app.js

```
var express = require('express');  
var http = require('http');
```

← express 모듈은 http 모듈 기반으로
운영되므로 필히 http를 함께 불러야 함

```
// 익스프레스 객체 생성  
var app = express();  
// 기본 포트를 app 객체에 속성으로 설정  
app.set('port', process.env.PORT || 3000);  
// Express 서버 시작  
http.createServer(app).listen(app.get('port'), function(){  
  console.log('익스프레스 서버를 시작했습니다 : ' + app.get('port'));  
});
```

← express 객체를 파라미터로 전달

• 익스프레스 객체의 주요 메소드와 주요 속성 이름

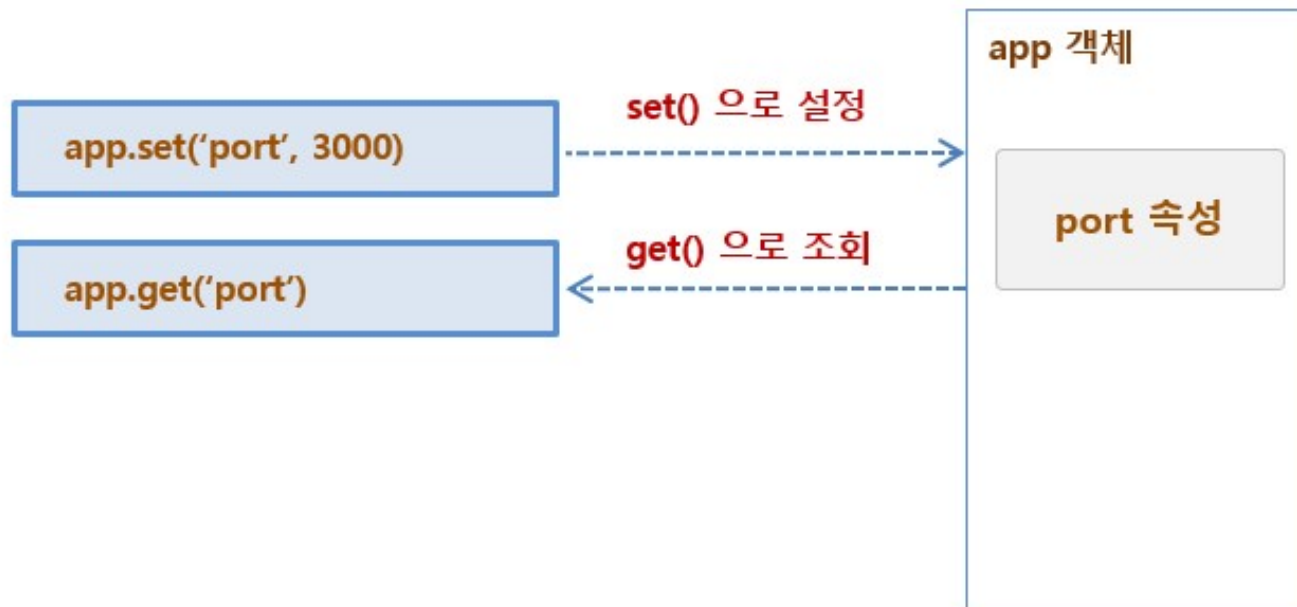
메소드 이름	설명
set(name, value)	서버 설정을 위한 속성을 지정합니다. set() 메소드로 지정한 속성은 get() 메소드로 꺼내어 확인할 수 있습니다.
get(name)	서버 설정을 위해 지정한 속성을 꺼내 옵니다.
use([path,] function [, function...])	미들웨어 함수를 사용하도록 합니다.
get([path,] function)	특정 패스로 요청된 정보를 처리합니다.

속성 이름	설명
env	서버 모드를 설정합니다.
views	뷰들이 들어 있는 폴더 또는 폴더 배열을 설정합니다.
view engine	디폴트로 사용할 뷰 엔진을 설정합니다.

- port 속성 설정

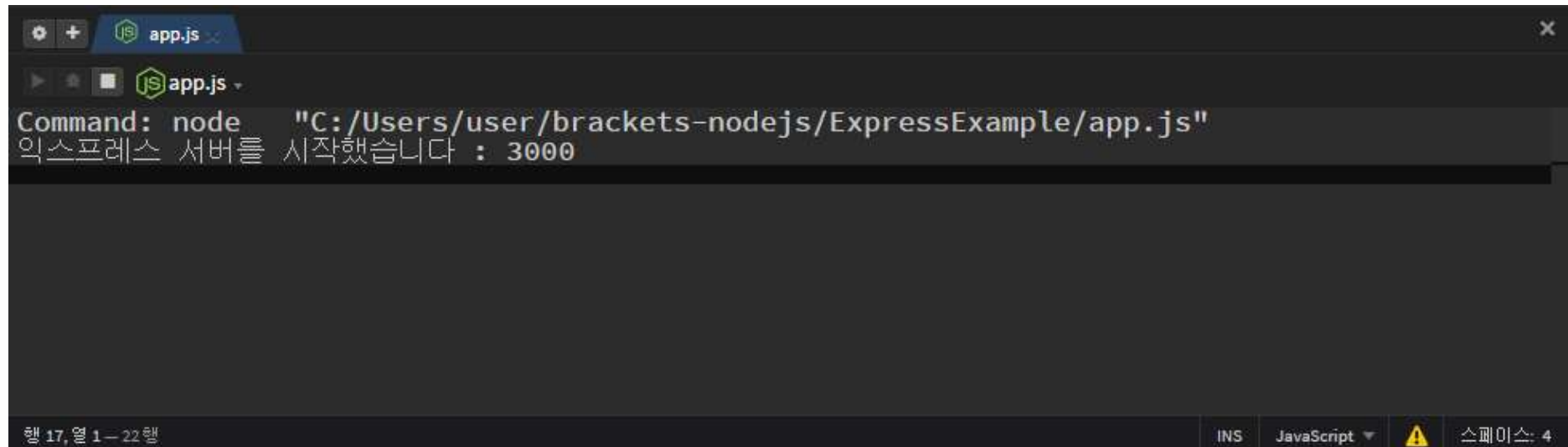
```
// 기본 포트를 app 객체에 속성으로 설정  
app.set('port', process.env.PORT || 3000);  
.....
```

```
http.createServer(app).listen(app.get('port'), function(){ ...
```



- Express 외장 모듈 설치하고 실행

```
% npm install express --save
```



The screenshot shows a code editor window with a dark theme. The top bar indicates the file is 'app.js'. Below the editor, a command prompt is visible, showing the command 'node "C:/Users/user/brackets-nodejs/ExpressExample/app.js"' being executed. The output of the command is '익스프레스 서버를 시작했습니다 : 3000', which translates to 'Express server started : 3000'. The status bar at the bottom shows '행 17, 열 1 - 22 행', 'INS', 'JavaScript', a warning icon, and '스페이스: 4'.

```
Command: node "C:/Users/user/brackets-nodejs/ExpressExample/app.js"
익스프레스 서버를 시작했습니다 : 3000
```

- 미들웨어

- 특정 path에 대한 요청을 처리
 - path: regular expression 사용
- 요청에 대하여 라우터 실행 이전에 필요한 작업을 하기 위한 독립된 함수

- 예

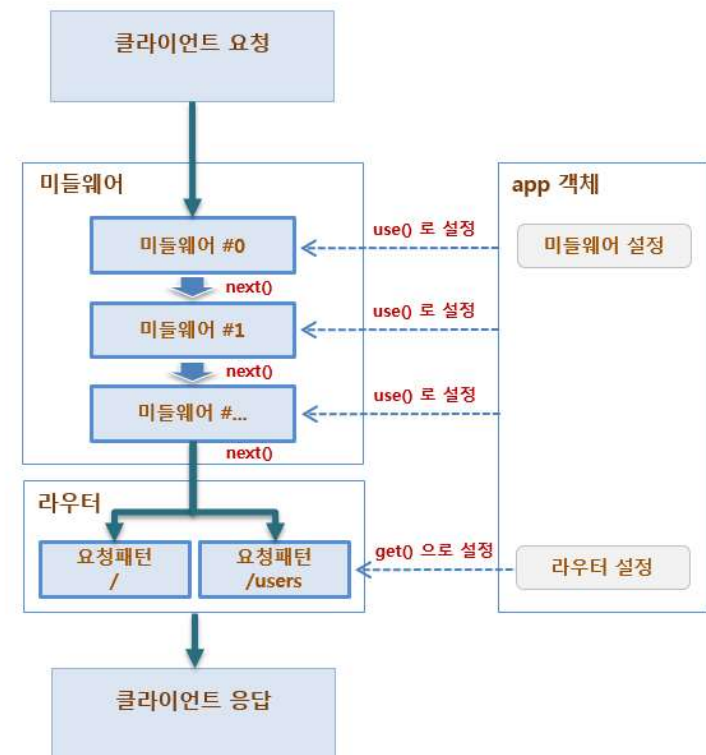
- 클라이언트 요청 로그 함수의 미들웨어 등록
-> 모든 클라이언트 요청에 대하여 로그 기록 남김

- next()

- 다음 미들웨어로 작업 처리를 넘김

- 라우터

- 요청 패스에 따라 해당 요청을 처리하기 위한 함수로 기능을 전달하는 역할



- use() 메소드
 - 미들웨어 함수를 등록

app2.js

```
var express = require('express')
, http = require('http');

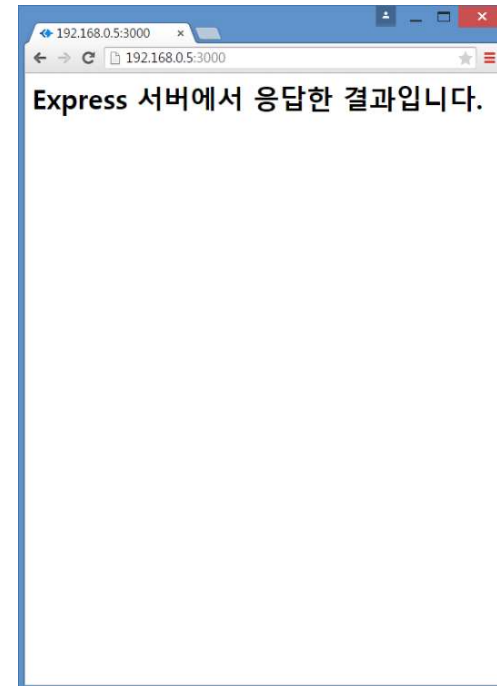
var app = express();
app.use(function(req, res, next) {
  console.log('첫 번째 미들웨어에서 요청을 처리함.');
```

res.writeHead('200', {'Content-Type': 'text/html; charset=utf8'});

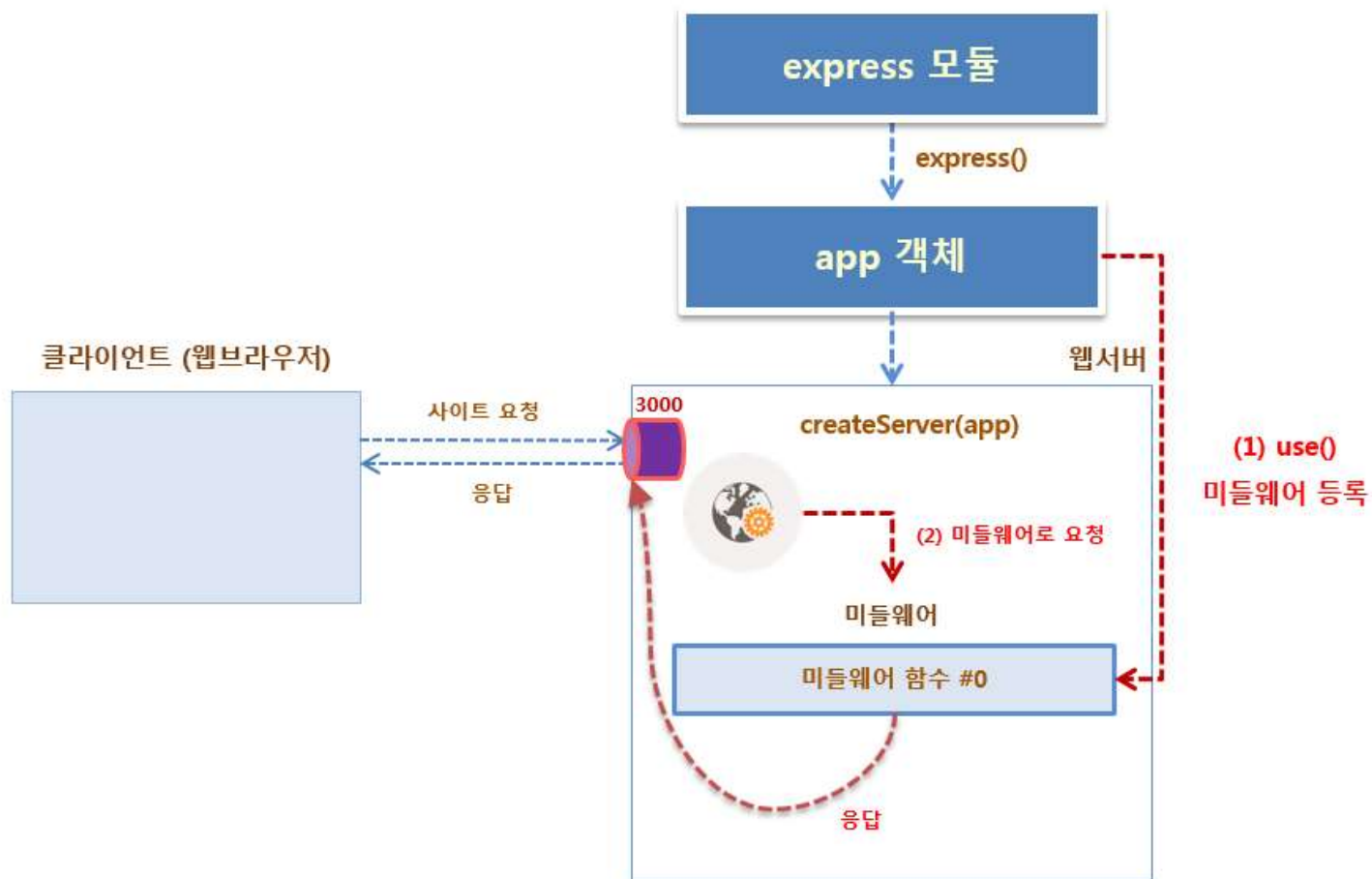
res.end('<h1>Express 서버에서 응답한 결과입니다.</h1>');

```
});
http.createServer(app).listen(3000, function() {
  console.log('Express 서버가 3000번 포트에서 시작됨.');
```

```
});
```



- ① express 객체를 만듬.
- ② use 메소드를 이용해 미들웨어를 등록함

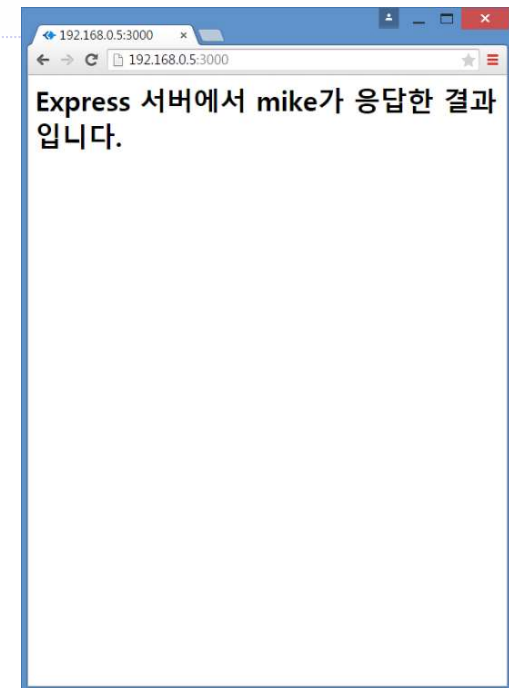


- use() 메소드를 여러 번 사용
 - 등록한 순서대로 호출됨

app3.js – app2 복사 후 추가

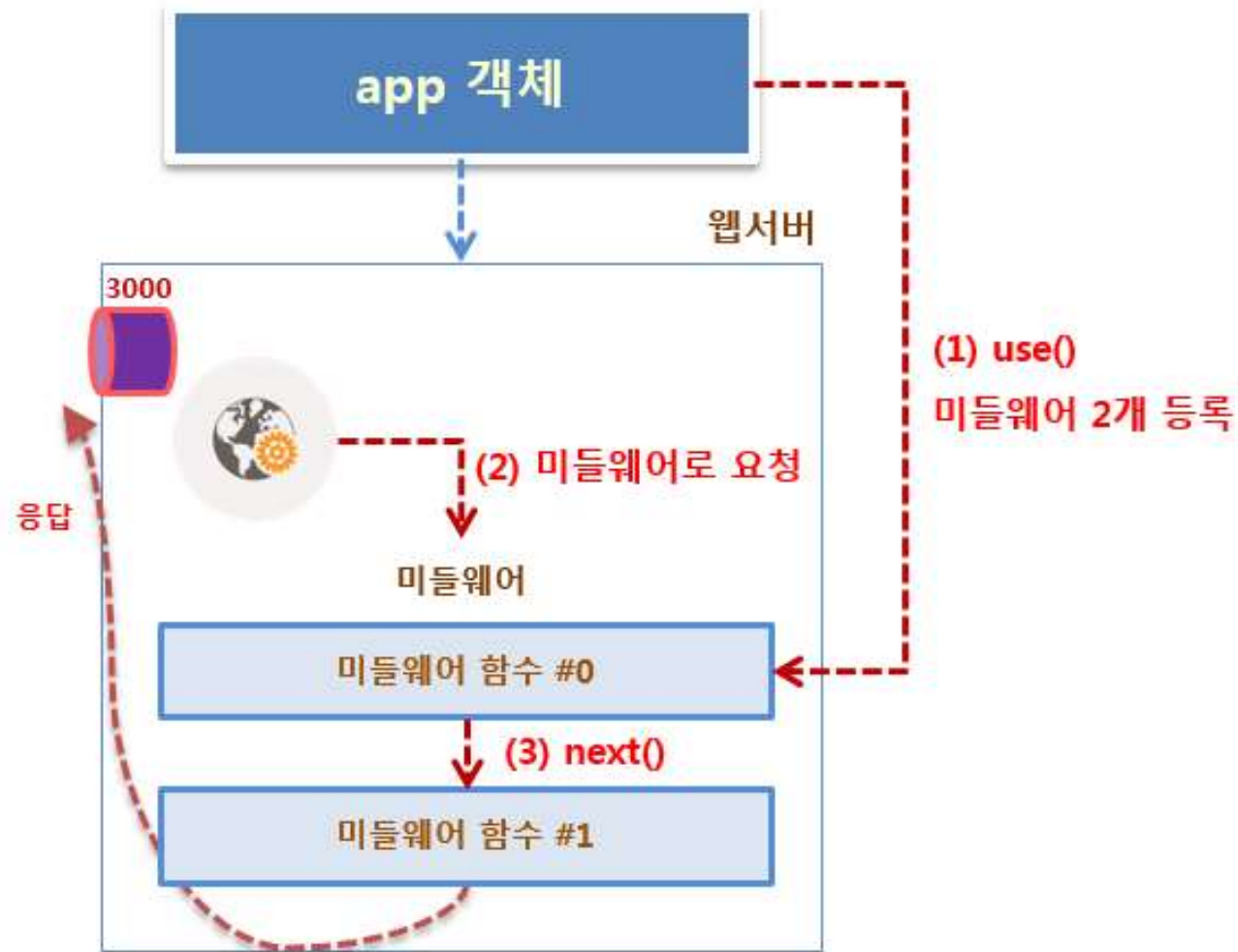
```
app.use(function(req, res, next) {  
  console.log('첫 번째 미들웨어에서 요청을 처리함.');  
  req.user = 'mike';  
  next();  
});
```

```
app.use('/', function(req, res, next) {  
  console.log('두 번째 미들웨어에서 요청을 처리함.');  
  res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});  
  res.end('<h1>Express 서버에서 ' + req.user + '가 응답한 결과입니다.</h1>');});
```



두 개의 미들웨어를 사용할 때의 구성

30/75



- 익스프레스의 응답 객체 추가 사용가능 메소드

메소드 이름	설명
send([body])	클라이언트에 응답 데이터를 보냅니다. 전달할 수 있는 데이터는 HTML 문자열, Buffer 객체, JSON 객체, JSON 배열입니다.
status(code)	HTTP 상태 코드를 반환합니다. 상태 코드는 end()나 send() 같은 전송 메소드를 추가로 호출해야 전송할 수 있습니다.
sendStatus(statusCode)	HTTP 상태 코드를 반환합니다. 상태 코드는 상태 메시지와 함께 전송됩니다.
redirect([status,] path)	웹 페이지 경로를 강제로 이동시킵니다.
render(view [, locals][, callback]	뷰 엔진을 사용해 문서를 만든 후 전송합니다.

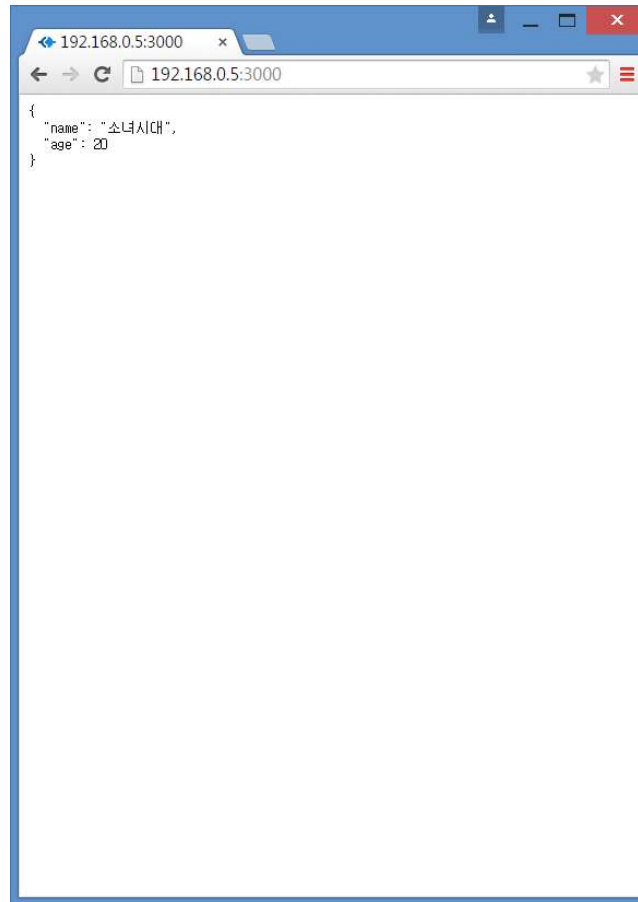
app4.js – app3 복사 후 수정

```
app.use(function(req, res, next) {  
  console.log('첫 번째 미들웨어에서 요청을 처리함.');
```

```
  res.send({name:'소녀시대', age:20});  
});
```

- 브라우저에서 응답 확인

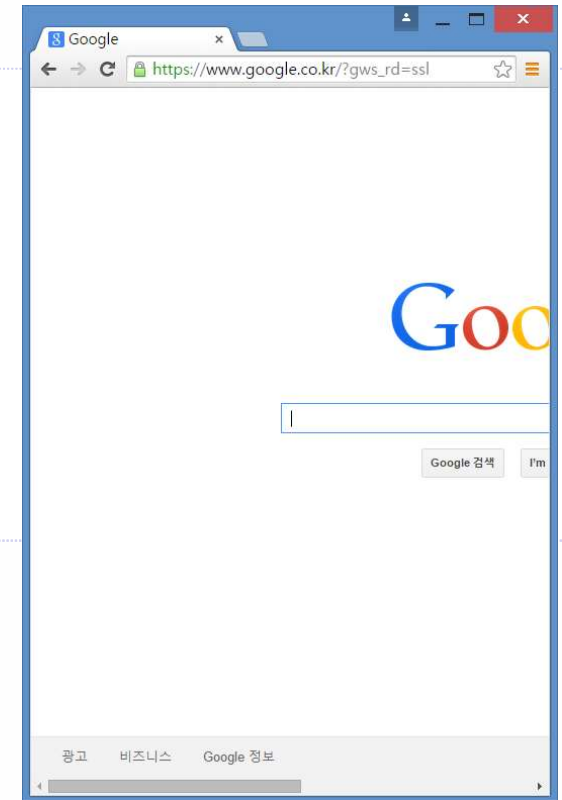
http://localhost:3000 또는 PC의 IP를 확인한 후 <http://192.168.0.5:3000>과 같이 조회



- redirect() 메소드
 - 다른 페이지로 이동

app5.js – app4 복사 후 수정

```
app.use(function(req, res, next) {  
  console.log('첫 번째 미들웨어에서 요청을 처리함.');  
  res.redirect('http://google.co.kr');  
});
```



- 헤더를 확인하는 방법과 요청 파라미터를 확인하는 방법
 - 요청 객체 사용

추가한 정보	설명
query	클라이언트에서 GET 방식으로 전송한 요청 파라미터를 확인합니다. 예) req.query.name
body	클라이언트에서 POST 방식으로 전송한 요청 파라미터를 확인합니다. 단, body-parser와 같은 외장 모듈을 사용해야 합니다. 예) req.body.name
header(name)	헤더를 확인합니다.

- 파라미터를 전달하고 그 파라미터를 다시 응답으로 받을 수 있음

app6.js – app5 복사 후 수정

.....

```
app.use(function(req, res, next) {  
  console.log('첫 번째 미들웨어에서 요청을 처리함.');
```



```
  var userAgent = req.header('User-Agent');
```



```
  var paramName = req.query.name;
```



```
  res.writeHead('200', {'Content-Type': 'text/html; charset=utf8'});
```



```
  res.write('<h1>Express 서버에서 응답한 결과입니다.</h1>');
```



```
  res.write('<div><p>User-Agent : ' + userAgent + '</p></div>');
```



```
  res.write('<div><p>Param name : ' + paramName + '</p></div>');
```



```
  res.end();
```



```
});
```

.....

- 파라미터를 전달하고 그 파라미터를 다시 응답으로 받을 수 있음

▶ http://localhost:3000/?name=mike

주소 문자열

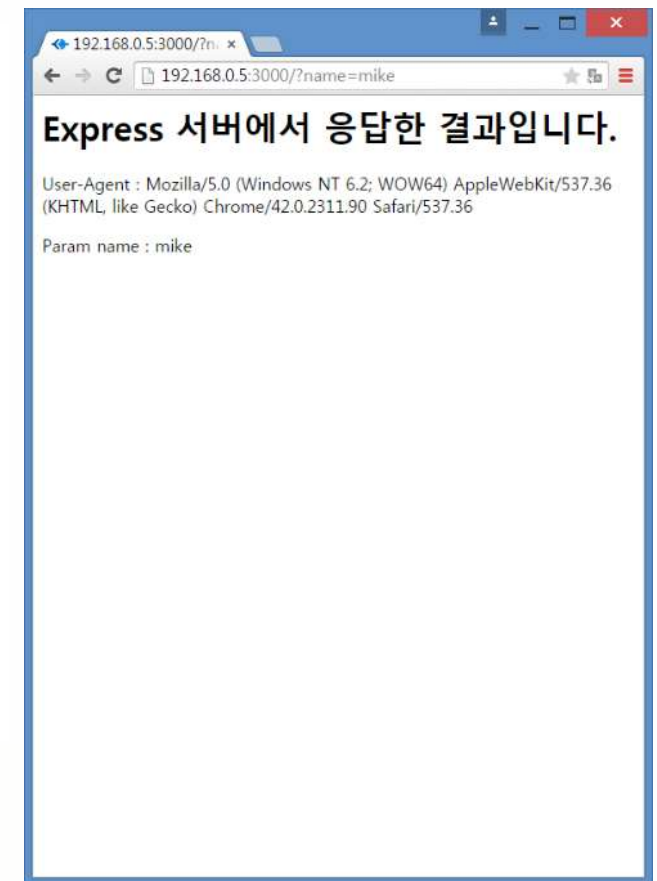
http://localhost:3000/?name=mike

req 객체



mike

req.query.name;



05-03 미들웨어 사용하기

- express는 사용자를 위해 여러가지 미들웨어를 제공함
- static 미들웨어
 - 특정 폴더의 파일들을 특정 패스로 접근할 수 있도록 열어주는 역할을 함

```
var static = require('serve-static');
```

```
...
```

```
app.use(static(path.join(__dirname, 'public')));
```

./public 폴더를 접근할 수 있도록
open 함

ExpressExample/public/index.html

ExpressExample/public/images/house.png

ExpressExample/public/css/style.css

./public 폴더의 파일을 아래와 같이
접근할 수 있음

http://localhost:3000/index.html

http://localhost:3000/images/house.png

http://localhost:3000/css/style.css

- /public/images/house.png 도 다음과 같이 패스를 지정할 수 있음

```
res.end("<img src='/images/house.png' width='50%'");
```

- /public 폴더의 파일을 /public 패스를 이용하여 접근하도록 지정할 경우 다음과 같이 설정

```
app.use('/public', static(path.join(__dirname, 'public')));
```



- body-parser 미들웨어
 - POST로 요청했을 때의 요청 파라미터 확인 방법 제공
 - 클라이언트가 POST 방식으로 요청하였을 때 body 영역에 들어있는 요청 파라미터를 파싱하여 request객체의 body 속성에 넣어줌
- ExpressExample 폴더 안에 있는 public 폴더 안에 login.html 파일 생성

login.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset = "UTF-8">
    <title>로그인 테스트</title>
  </head>
  <body>
    <h1>로그인</h1>
    <br>
```

```
<form method= "post">
  <table>
    <tr>
      <td><label>아이디</label></td>
      <td><input type= "text" name= "id" /></td>
    </tr>
    <tr>
      <td><label>비밀번호</label></td>
      <td><input type= "password" name= "password" /></td>
    </tr>
  </table>
  <input type= "submit" value= "전송" name= "" />
</form>
</body>
</html>
```


app7.js – app6 복사 후 수정

```
// Express 기본 모듈 불러오기
var express = require('express')
  , http = require('http')
  , path = require('path');

// Express의 미들웨어 불러오기
var bodyParser = require('body-parser')
  , static = require('serve-static');

// 익스프레스 객체 생성
var app = express();

// 기본 속성 설정
app.set('port', process.env.PORT || 3000);

// body-parser를 이용해 application/x-www-form-urlencoded 형식 파싱
app.use(bodyParser.urlencoded({ extended: false }));

// body-parser를 이용해 application/json 형식 파싱
app.use(bodyParser.json());

app.use(static(path.join(__dirname, 'public')));

...
```

- Content-Type이 특정 값(html)이고 urlencoded 형태의 body를 해석하는 body-parser객체를 만듦
- extended: URL encoded된 데이터를 해석하는 라이브러리 선택
 - . false: querystring 사용
 - . true: qs 사용

- 미들웨어에서 파라미터 확인 후 응답 전송

app7.js – 계속

...

// 미들웨어에서 파라미터 확인

```
app.use(function(req, res, next) {  
  console.log('첫번째 미들웨어에서 요청을 처리함.');
```



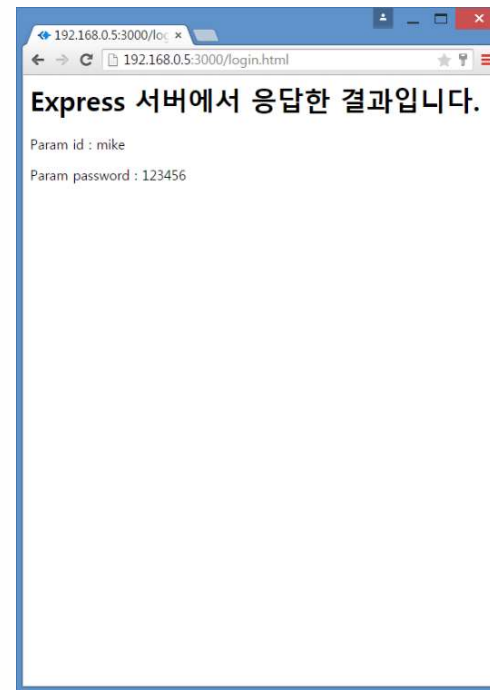
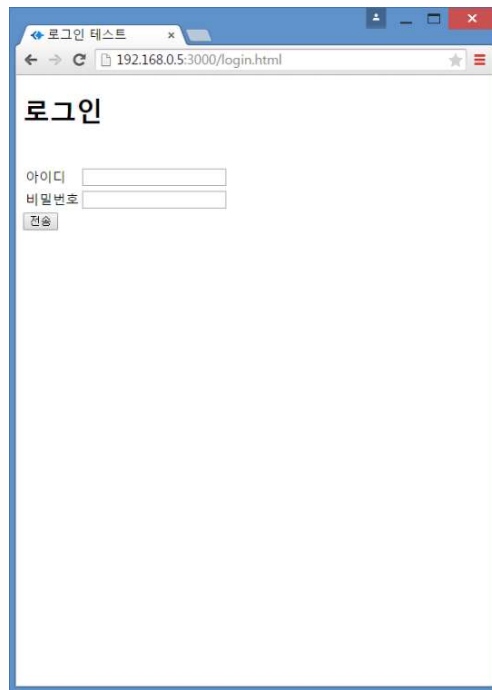
```
  var paramId = req.body.id || req.query.id;  
  var paramPassword = req.body.password || req.query.password;
```



```
  res.writeHead('200', {'Content-Type': 'text/html; charset=utf8'});  
  res.write('<h1>Express 서버에서 응답한 결과입니다.</h1>');  
  res.write('<div><p>Param id : ' + paramId + '</p></div>');  
  res.write('<div><p>Param password : ' + paramPassword + '</p></div>');  
  res.end();  
});
```

- npm으로 설치

```
% npm install body-parser --save
```



05-04 요청 라우팅하기

- 라우터 미들웨어
 - 클라이언트가 요청한 기능을 패스를 기준으로 구별하여 실행
- 사용방법
 - Router 객체를 참조한 후 route() 메소드를 이용해 라우팅

```
// 1. router 객체 reference  
var router = express.Router(); ...
```

```
// 2. routing path와 function 등록  
router.route('/process/login').get(function(req, res) { ...  
router.route('/process/login').post(function(req, res) { ...
```

```
// 3. router 객체 express에 등록  
app.use('/', router);
```

메소드 이름	설명
get(callback)	GET 방식으로 특정 패스 요청이 발생했을 때 사용할 콜백 함수를 지정합니다.
post(callback)	POST 방식으로 특정 패스 요청이 발생했을 때 사용할 콜백 함수를 지정합니다.
put(callback)	PUT 방식으로 특정 패스 요청이 발생했을 때 사용할 콜백 함수를 지정합니다.
delete(callback)	DELETE 방식으로 특정 패스 요청이 발생했을 때 사용할 콜백 함수를 지정합니다.
all(callback)	모든 요청 방식을 처리하며, 특정 패스 요청이 발생했을 때 사용할 콜백 함수를 지정합니다.

login2.html – login.html 복사 후 수정

```
.....  
<form method="post" action="/process/login">  
.....
```

app8.js – app7 복사 후 수정

```
.....  
var router = express.Router();  
.....  
router.route('/process/login').post(function(req, res) {  
  console.log('/process/login 처리함.');  
  var paramId = req.body.id || req.query.id;  
  var paramPassword = req.body.password || req.query.password;  
  
  res.writeHead('200', {'Content-Type': 'text/html; charset=utf8'});  
  res.write('<h1>Express 서버에서 응답한 결과입니다.</h1>');  
  res.write('<div><p>Param id : ' + paramId + '</p></div>');  
  res.write('<div><p>Param password : ' + paramPassword + '</p></div>');  
  res.write("&<br><br><a href='/public/login2.html'>로그인 페이지로 돌아가기</a>");  
  res.end();  
});  
.....  
app.use('/', router);
```

- 등록한 요청 패스에 해당하는 함수를 실행

`http://localhost:3000/public/login2.html`



- GET 방식의 파라미터 → query 객체
- POST 방식의 파라미터 → body 객체
- URL 파라미터 → params 객체
 - URL 주소의 일부로 파라미터 포함됨

app8_02.js – app7 복사 후 수정

```
router.route('/process/login/:name').post(function(req, res) {
  console.log('/process/login/:name 처리함.');
```



```
  var paramName = req.params.name;
```



```
  var paramId = req.body.id || req.query.id;
  var paramPassword = req.body.password || req.query.password;
```



```
  res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
  res.write('<h1>Express 서버에서 응답한 결과입니다.</h1>');
  res.write('<div><p>Param name : ' + paramName + '</p></div>');
  res.write('<div><p>Param id : ' + paramId + '</p></div>');
  res.write('<div><p>Param password : ' + paramPassword + '</p></div>');
  res.write("&<br><br><a href='/public/login2.html'>로그인 페이지로 돌아가기</a>");
  res.end();
});
```


- 호출할 때 URL 안에 파라미터가 포함되도록 요청

login3.html – login2.html 복사 후 수정

```
.....  
<form method="post" action="/process/login/mike">  
.....
```

- URL 안에 들어간 파라미터가 매핑되는 형식

/process/login/mike ←----- Token이라고 함



/process/login/:name

http://localhost:3000/public/login2.html

로그인 테스트

localhost:3000/public/login3.html

로그인

아이디 test01

비밀번호

전송

localhost:3000/process/login/mike

Express 서버에서 응답한 결과입니다.

Param name : mike

Param id : test01

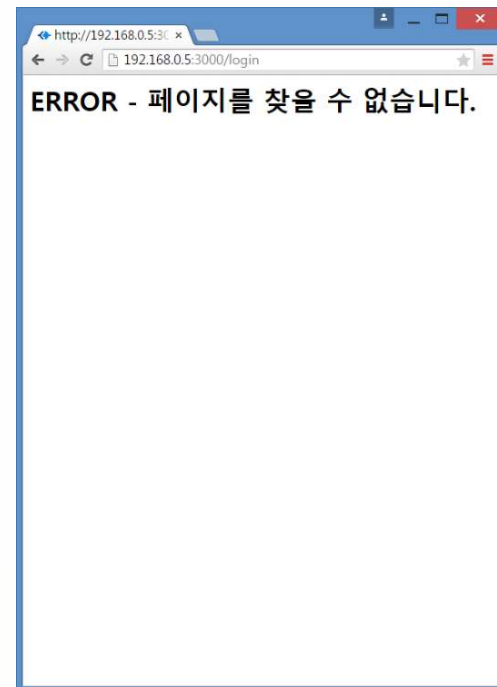
Param password : 123456

[로그인 페이지로 돌아가기](#)

- 등록되지 않은 요청 패스일 경우
 - default error page가 표시됨
 - 자체적인 오류 페이지 표시할 수 있음

app8.js – app8.js 수정

```
.....  
app.all('*', function(req, res) {  
  res.status(404).send('<h1>ERROR - 페이지를 찾을 수 없습니다.</h1>');  
});
```



- express-error-handler
 - error 처리용 미들웨어
 - http 에러에 대해서도 처리 가능

app9.js – app8.js 복사 후 수정

```
.....  
var expressErrorHandler = require('express-error-handler');  
  
.....  
// 모든 라우터 처리 후 404 오류 페이지 처리  
var errorHandler = expressErrorHandler({  
  static: {  
    '404': './public/404.html'  
  }  
});  
  
// 서버가 시작하기 전에 미들웨어로 추가되어야 함  
app.use( expressErrorHandler.handleError(404) );  
app.use( errorHandler );  
  
.....
```

- public 폴더 안에 404.html 파일 생성하고 모듈 설치

404.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset= "UTF-8">
  <title>오류 페이지 </title>
</head>
<body>
  <h3>ERROR - 페이지를 찾을 수 없습니다.</h3>
  <hr/>
  <p>/public/404.html 파일의 오류 페이지를 표시한 것입니다.</p>
</body>
</html>
```

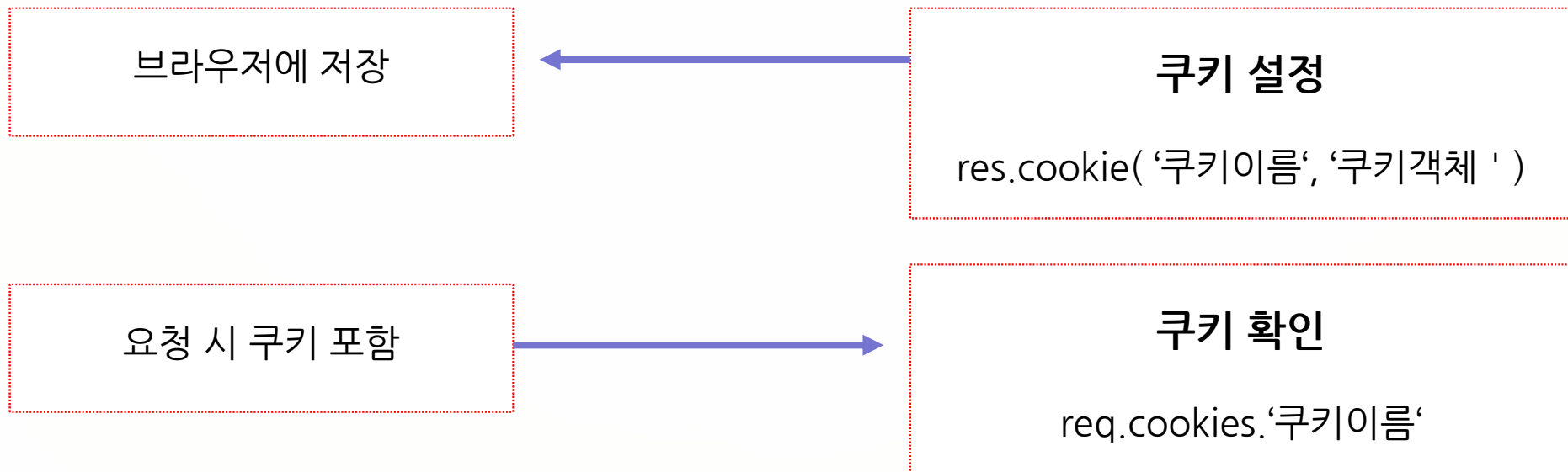
```
% npm install express-error-handler --save
```

- 등록된 요청 패스를 찾지 못하는 경우



05-05 쿠키와 세션 관리하기

- 웹 에서 상태 정보의 사용
 - 쿠키나 세션 사용
- 쿠키: 클라이언트 웹 브라우저에 저장되는 정보
 - 웹서버에서 응답할 때 쿠키를 설정하면 그 정보를 받은 웹브라우저에서 쿠키 저장
 - 응답할 때 응답 객체에 쿠키 설정 : `res.cookie()` 메소드 호출
 - 웹브라우저에서 웹서버로 요청할 때 쿠키 정보 전송
 - 요청 객체에 들어있는 쿠키 확인 : `req.cookies` 객체 안의 속성으로 확인
- 세션: 웹 서버에 저장되는 정보



- cookie-parser 미들웨어 사용

app11.js – app10.js 복사 후 추가

```
.....  
var cookieParser = require('cookie-parser');
```

```
.....  
// cookie-parser 설정  
app.use(cookieParser());  
  
.....
```


- 응답 객체의 cookie() 메소드 호출

app11.js – 계속

```
router.route('/process/setUserCookie').get(function(req, res) {
  console.log('/process/setUserCookie 호출됨.');
```



```
  // 쿠키 설정
  res.cookie('user', {
    id: 'mike',
    name: '소녀시대',
    authorized: true
  });
```



```
  // redirect로 응답
  res.redirect('/process/showCookie');
});
```

- showCookie 요청 패스에서 쿠키 정보 표시
 - 요청 객체의 cookies 속성 사용

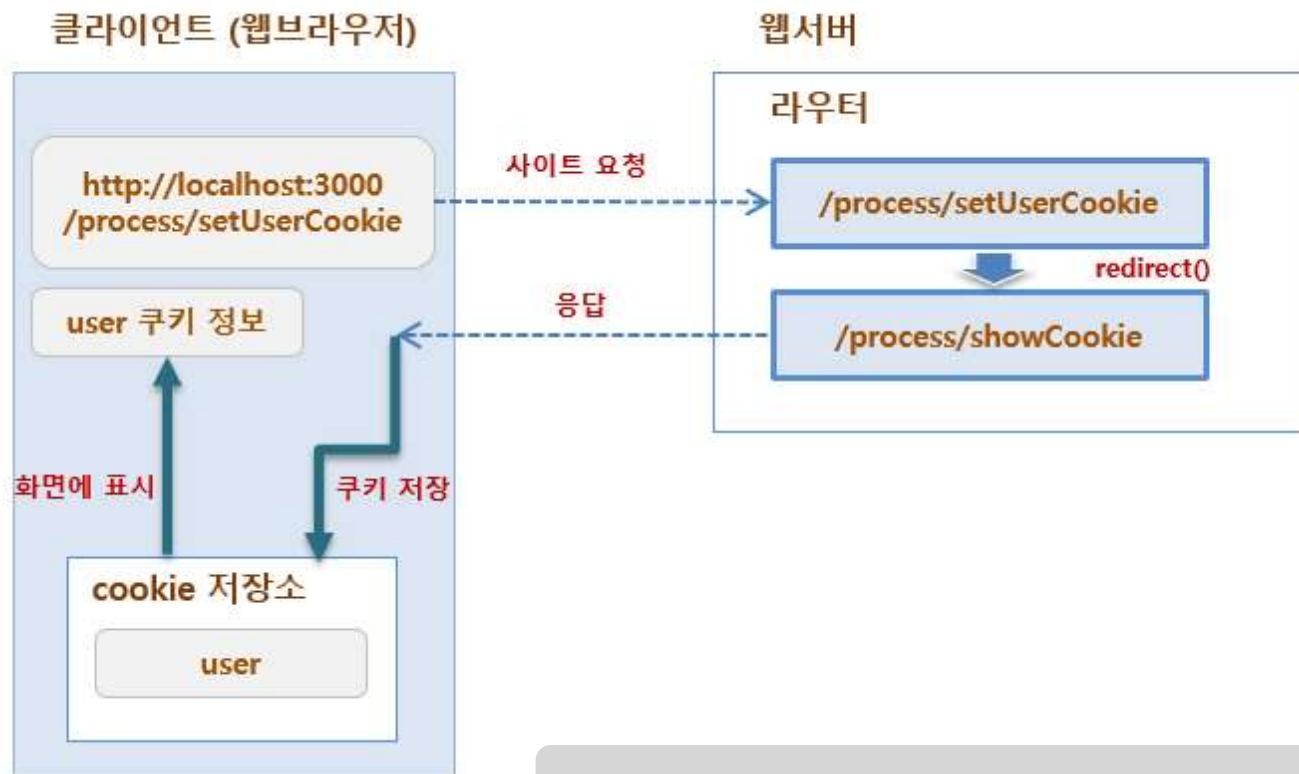
app11.js – 계속

```
router.route('/process/showCookie').get(function(req, res) {  
  console.log('/process/showCookie 호출됨.');
```

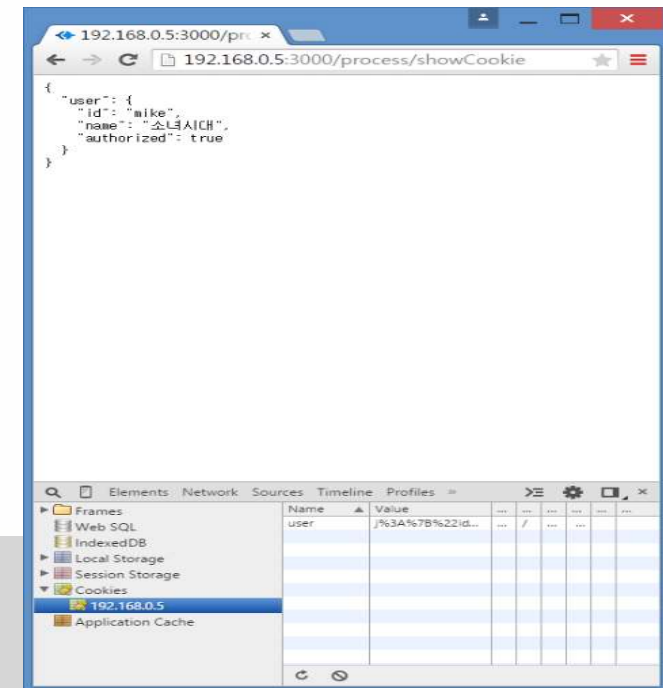


```
  res.send(req.cookies);  
});
```

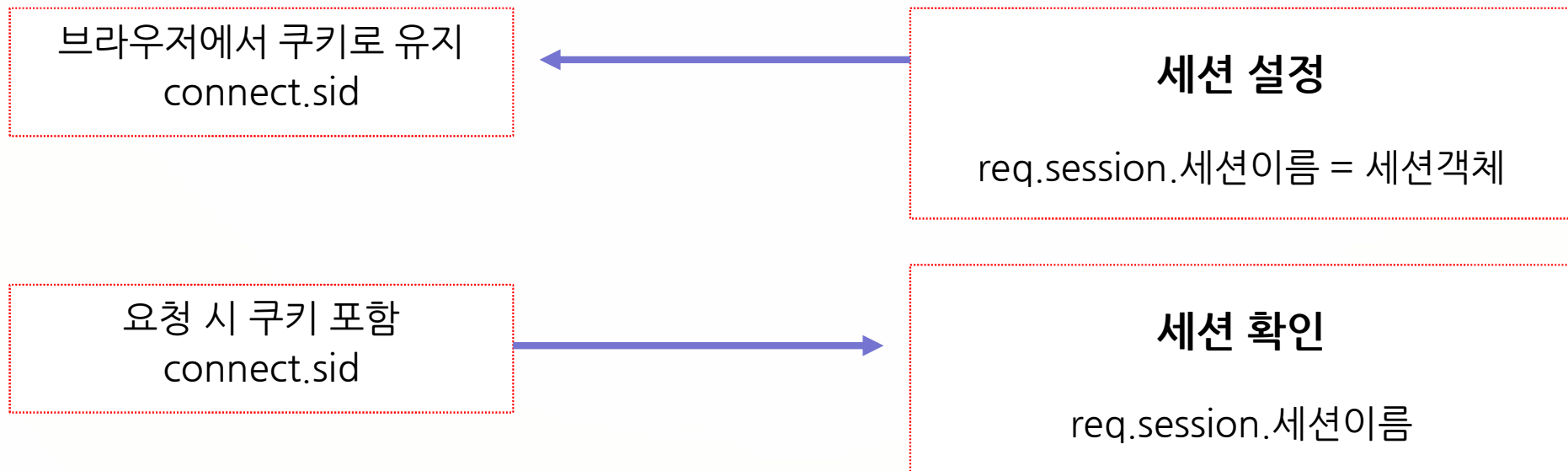
- 웹 브라우저의 쿠키 저장소에 저장됨
- 요청 객체의 cookies 속성 사용



% npm install cookie-parser --save

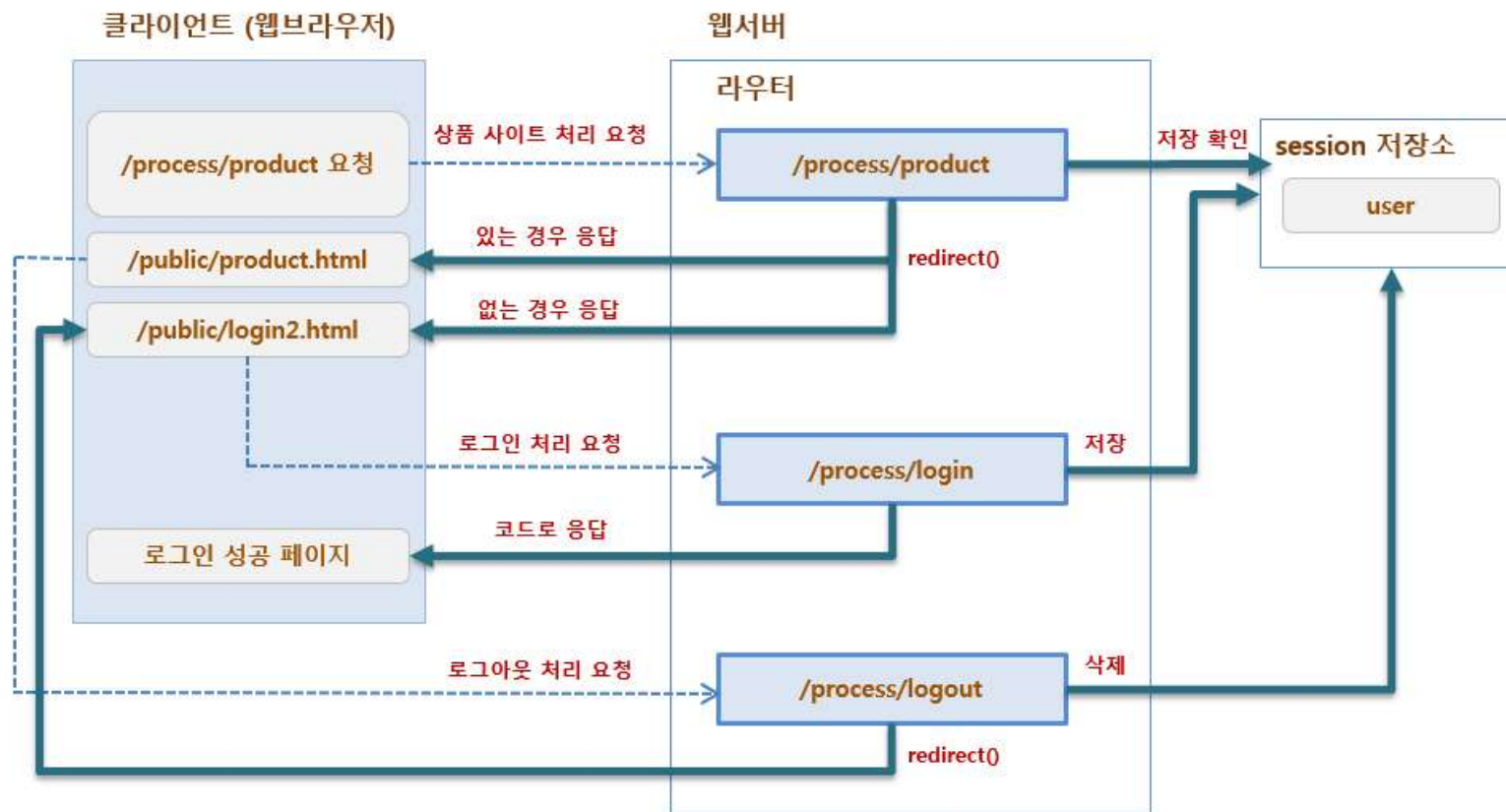


- 세션: 웹 서버에 저장되는 정보
 - 웹서버에서 요청 객체에 세션을 설정하면 유지됨 → req.session.세션이름 = 세션객체
 - 그 정보를 받은 웹브라우저에서도 connect.sid 쿠키 저장
 - 웹브라우저에서 웹서버로 요청할 때 connect.sid 쿠키 정보 전송
 - 요청에 들어있는 세션 정보 확인 → req.session.세션이름



- 세션 예제 코드

- 로그인하면 세션이 만들어지고 로그아웃하면 세션이 삭제되도록 만들 수 있음



• 모듈을 불러들이고 미들웨어로 사용

app12.js – app11 복사 후 추가

```
// Session 미들웨어 불러오기  
var expressSession = require('express-session');
```

```
// 세션 설정
```

```
app.use(expressSession({  
  secret: 'my key',  
  resave: true,  
  saveUninitialized: true  
}));
```

- secret: 쿠키의 임의 변조를 방지하기 위한 값, 이 값을 이용하여 암호화 후 저장함
- resave: 세션을 항상 저장할 지 설정
- saveUninitialized: 세션이 저장되기 전 uninitialized 상태로 미리 만들어서 저장

• 요청 객체의 session 속성에 세션 정보 추가

app12.js – 계속

```
router.route('/process/login').post(function(req, res) {
  console.log('/process/login 호출됨.');
```



```
  var paramId = req.body.id || req.query.id;
  var paramPassword = req.body.password || req.query.password;
```



```
  if (req.session.user) {
    // 이미 로그인된 상태
    console.log('이미 로그인되어 상품 페이지로 이동합니다.');
```



```
    res.redirect('/public/product.html');
```

```
  } else {
    // 세션 저장
    req.session.user = {
      id: paramId,
      name: '소녀시대',
      authorized: true
    };
    // - 요청 객체의 session 변수에 user 로 세션 값 저장
    // - session.[키 이름] = 값 으로 설정함
```



```
    res.writeHead('200', {'Content-Type': 'text/html; charset=utf-8'});
    res.write('<h1>로그인 성공</h1>');
    res.write('<div><p>Param id: ' + paramId + '</p></div>');
    res.write('<div><p>Param password: ' + paramPassword + '</p></div>');
    res.write("&<br><br><a href='/process/product'>상품페이지로 이동하기</a>");
    res.end();
  });
```

- 요청 객체의 session 속성에 세션 정보가 있는지 확인

app12.js – 계속

```
router.route('/process/logout').get(function(req, res) {
  console.log('/process/logout 호출됨.');
```



```
  if (req.session.user) {
    // 로그인된 상태
    console.log('로그아웃합니다.');
```



```
    req.session.destroy(function(err) {
      if (err) {throw err;}


      console.log('세션을 삭제하고 로그아웃되었습니다.');
```



```
      res.redirect('/public/login2.html');
    });
  } else {
    // 로그인 안된 상태
    console.log('아직 로그인되어있지 않습니다.');
```



```
    res.redirect('/public/login2.html');
  }
});
```



- 상품 정보 확인 페이지에서 세션 정보 확인

app12.js – 계속

```
router.route('/process/product').get(function(req, res) {  
  console.log('/process/product 호출됨.');
```



```
  if (req.session.user) {  
    res.redirect('/public/product.html');
```



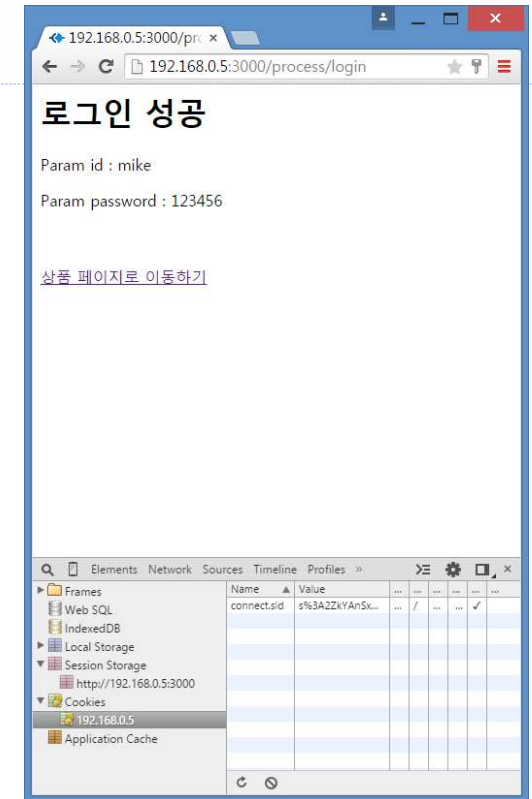
```
  } else {  
    res.redirect('/public/login2.html');
```



```
  }  
});
```

- product.html 페이지 등 생성

```
<!DOCTYPE html>
<html>
<head>
  <meta charset= "UTF-8">
  <title>상품 페이지</title>
</head>
<body>
  <h3>상품정보 페이지</h3>
  <hr/>
  <p>로그인 후 볼 수 있는 상품정보 페이지입니다.</p>
  <br><br>
  <a href= '/process/logout'>로그아웃하기</a>
</body>
</html>
```



05-06 파일 업로드 만들기

- multer 미들웨어
 - 파일 업로드 시에는 multipart 포맷으로 된 파일 업로드 기능을 사용해야 함
- 파일 업로드 시 POST 방식으로 요청해야 함
 - body-parser 미들웨어 사용 필요
- fs, cors 모듈도 사용

```
% npm install multer --save
% npm install cors --save
```

app13.js – app12 복사 후 추가

```
...
// 파일 업로드용 미들웨어
var multer = require('multer');
var fs = require('fs');

//클라이언트에서 ajax로 요청 시 CORS(다중 서버 접속) 지원
var cors = require('cors');
...
```

- 미들웨어 사용 순서 조심할 것
 - body-parser -> multer -> router 순으로 사용
- 업로드 가능한 파일 크기 등을 설정할 수 있음
- 파일 저장을 위한 폴더 설정 및 파일명 변경 가능

app13.js – 계속

```
var storage = multer.diskStorage({
  destination: function (req, file, callback) {
    callback(null, 'uploads')
  },
  filename: function (req, file, callback) {
    callback(null, file.originalname + Date.now())
  }
});
```

multer 모듈을 이용하여 post 방식으로 전송된 파일의 저장 경로와 파일명을 처리하기 위하여 DiskStorage 설정 필요

```
var upload = multer({
  storage: storage,
  limits: {
    files: 10,
    fileSize: 1024 * 1024 * 1024
  }
});
```

- multer 모듈을 option 객체로 설정
- storage: 파일을 저장할 위치
- limits: 업로드할 파일의 제한사항

- /process/photo로 오는 요청 처리

- 요청 객체의 files 속성으로 업로드된 파일을 확인할 수 있음

- 파일 업로드 수행
- upload.array 다음에 callback 함수 수행
- HTML name 속성이 'photo'로 된 것을 처리
- 여러 개의 파일인 경우에 array로 만들 ,req.files 로 파일 속성 참조 가능
- 파일 1개인 경우 upload.single() 호출, req.file 로 파일 속성 참조

app13.js – 계속

```
router.route('/process/photo').post(upload.array('photo', 1), function(req, res) {  
  console.log('/process/photo 호출됨.');
```

```
  try {  
    var files = req.files; ← 업로드한 파일들은 files 배열 객체로 저장됨
```

```
    console.dir('#==== 업로드된 첫번째 파일 정보 ====#')  
    console.dir(req.files[0]);  
    console.dir('#====#')
```

```
    // 현재의 파일 정보를 저장할 변수 선언
```

```
    var originalname = '',  
        filename = '',  
        mimetype = '',  
        size = 0;
```

```
    if (Array.isArray(files)) {  
      // 배열에 들어가 있는 경우 (설정에서 1개의 파일도 배열에 넣게 했음)  
      console.log("배열에 들어있는 파일 갯수 : %d", files.length);
```

- 파일 객체에는 **originalname**, **filename**, **mimetype**, **size** 등의 속성이 들어있음

```
...  
    for (var index = 0; index < files.length; index++) {  
        originalname = files[index].originalname;  
        filename = files[index].filename;  
        mimetype = files[index].mimetype;  
        size = files[index].size;  
    }  
} else { // 배열에 들어가 있지 않은 경우 (현재 설정에서는 해당 없음)  
    console.log("파일 갯수 : 1 ");  
  
    originalname = files[index].originalname;  
    filename = files[index].name;  
    mimetype = files[index].mimetype;  
    size = files[index].size;  
}  
  
console.log('현재 파일 정보 : ' + originalname + ', ' + filename +  
            ', ' + mimetype + ', ' + size);
```

- **<form>** 태그를 사용하고 **enctype**을 **multipart/form-data**로 함

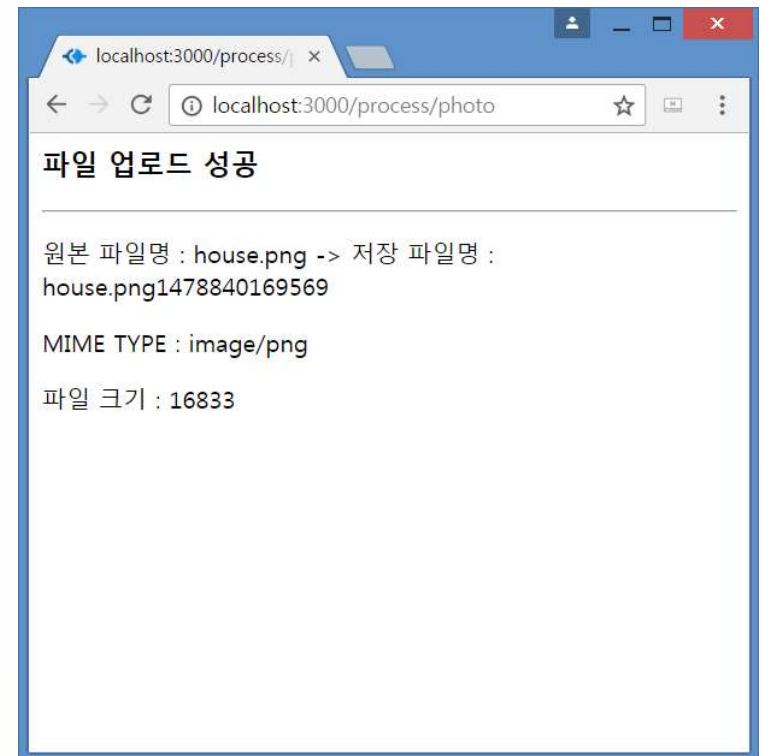
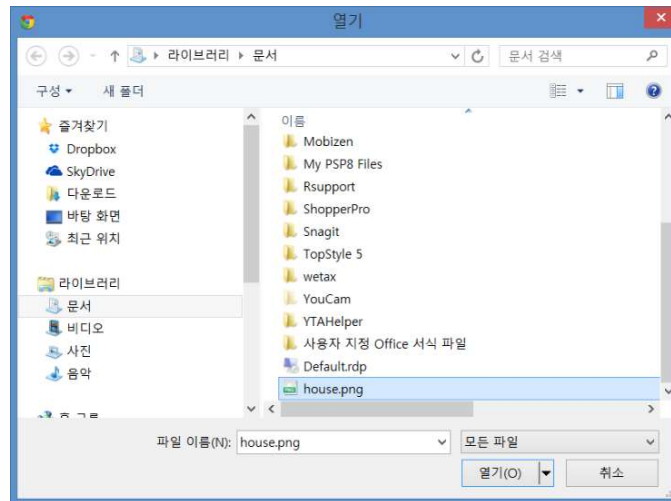
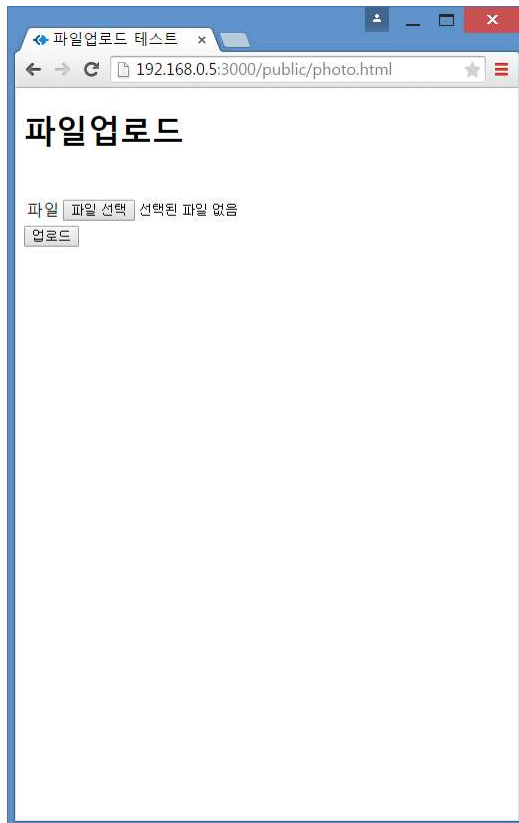
photo.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset= "UTF-8">
  <title>파일 업로드 테스트</title>
</head>
<body>
  <h1>파일 업로드</h1>
  <br>
  <form method= "post" enctype= "multipart/form-data" action= "/process/photo">
    <table>
      <tr>
        <td><label>파일</label></td>
        <td><input type= "file" name= "photo" /></td>
      </tr>
    </table>
    <input type= "submit" value= "업로드" name= "submit" />
  </form>
</body>
</html>
```

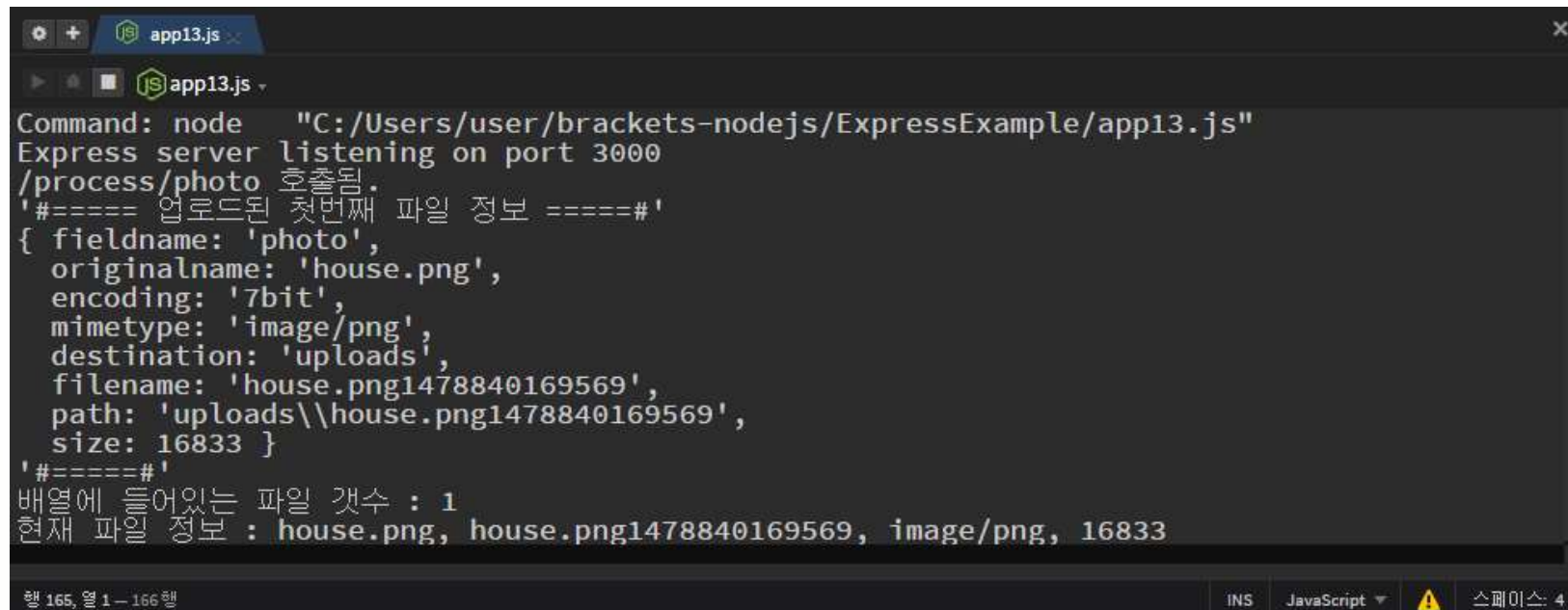

실행하여 파일 업로드 확인

73/75

▶ <http://localhost:3000/public/photo.html>



- 업로드된 파일 정보 확인 가능



```
Command: node "C:/Users/user/brackets-nodejs/ExpressExample/app13.js"
Express server listening on port 3000
/process/photo 호출됨.
'#==== 업로드된 첫번째 파일 정보 ====#'
{ fieldname: 'photo',
  originalname: 'house.png',
  encoding: '7bit',
  mimetype: 'image/png',
  destination: 'uploads',
  filename: 'house.png1478840169569',
  path: 'uploads\\house.png1478840169569',
  size: 16833 }
'#====#'
배열에 들어있는 파일 갯수 : 1
현재 파일 정보 : house.png, house.png1478840169569, image/png, 16833
```

- multipart 포맷으로 전송하는 일반적인 표준 업로드 방식 사용

