



Unit **1** 3-6

Node.js의 시작

UT-NodeJS / 03.17.2023

[ut-nodejs.github.io](https://ut-nodejs.github.io)



# Contents / 내용

설치와 사용하는 방법



## 03. Node.js 모듈 생성

- 새로운 Node.js 모듈 생성
- npm으로 Node.js 애플리케이션 구축
- npm으로 Node.js 패키지 설치

## 04. Node.js에서 웹 서버 만들기

- Node.js와 npm을 사용한 기본 웹 서버 생성
- 브라우저에서의 요청 처리 및 결과 보내기 코드 작성
- 브라우저에서의 웹 서버 실행

## 05. 수신 데이터 다루기

- 요청 데이터의 수집과 처리
- curl 명령을 통한 POST 요청 제출
- 기본 라우트를 가지는 애플리케이션 제작

## 06. 라우트와 외부 파일

- fs 모듈을 이용한 전체 HTML 파일의 저장
- 정적 애셋 저장
- 라우트 모듈 생성

# 03

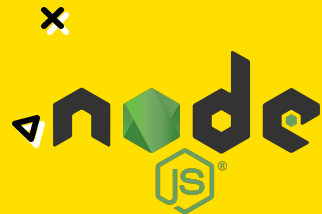
## Node.js 모듈 생성

새로운 Node.js 모듈 생성  
npm으로 Node.js 애플리케이션 구축  
npm으로 Node.js 패키지 설치

p. 69-78



## JS 모듈이 뭔가?



### 모듈, 패키지, 종속 모듈

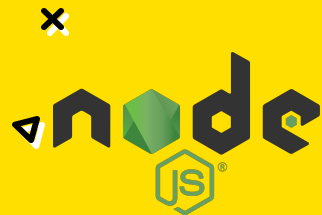
Node.js 애플리케이션은 많은 JavaScript 파일로 구성된다. 애플리케이션을 체계적이고 효율적으로 유지하려면 이러한 파일이 필요할 때 서로 접근할 수 있어야 한다. 코드 라이브러리가 들어 있는 각 JavaScript 파일 또는 폴더를 **모듈**이라고 한다.

여러분이 Node.js 개발 시 모듈, 패키지, 종속 모듈이라는 용어를 자주 접할 것이다. 용어의 정의는 다음과 같다.

- **모듈**은 단일 콘셉트, 기능 또는 라이브러리와 관련된 코드를 포함하는 개인 JavaScript 파일이다.
- **패키지**는 여러 개의 모듈을 포함할 수 있다. 연관 도구들을 제공하는 파일을 묶는 데 사용된다.
- **종속 모듈**은 애플리케이션 또는 다른 모듈에 의해 사용되는 Node.js 모듈이다. 만약 어떤 패키지가 어떤 애플리케이션에 종속 관계를 갖고 있다면 이는 애플리케이션이 동작하기 위해 (애플리케이션에서 특정된 시점의 버전으로) 반드시 설치돼야 할 것이다.



## JS 모듈이 뭐가?



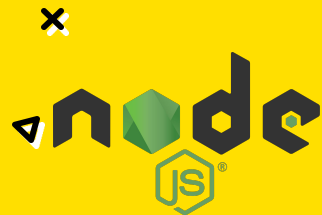
### 패키지 설치하기

애플리케이션에 일부 기능을 통합하려면 <https://www.npmjs.com>에서 이를 지원하는 패키지를 찾을 수 있다.

```
npm install <package> // 애플리케이션에 종속 모듈로 해당 패키지
npm install <pkg1> <pkg2> <pkg3> // 동시에 여러 패키지 설치
npm install <package> --global // 패키지를 컴퓨터상의 전역 패키지로 설치
npm install <package> --save-dev // 패키지가 개발 목적으로만 사용
npm install <package> --save-prod // 프로덕션을 위한 패키지
```



## JS 모듈이 뭔가?



### 모듈 사용하기

패키지에는 Node.js의 exports 객체에 속성으로 추가되는 많은 JS 모듈이 포함될 수 있습니다.

그런 다음 require를 사용하여 가져와 애플리케이션에서 사용하거나 공유할 수 있습니다.

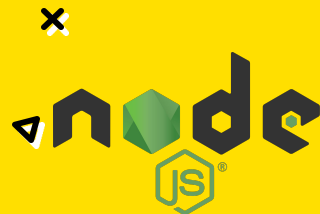
```
// messages.js
var messages = ["Hello", "Nice to meet you", "How are you?"];
exports.messages = messages;

// other.js
const messagesModule = require("./messages");
messagesModule.messages.forEach(m => console.log(m));
```

require는 다른 모듈의 메소드와 객체들을 로컬에서 사용하게 하는 또 다른 Node.js 전역 객체다. require 모듈은 모듈에 코드를 읽어들이는 역할을 하며, 읽어들이는 모듈을 우리가 만든 모듈의 export 객체에 붙이는 방식으로 수행한다. 그 결과, 가져온 코드가 어떤 방식으로 재사용될 필요가 생겨도 이를 다시 읽어들이는 필요가 없어진다.



## JS 모듈이 뭔가?



### 다른 **npm** 명령

Node.js 설치 시 Node.js의 패키지 관리자인 npm도 같이 설치된다. npm은 외부 패키지의 관리 (온라인상으로 다른 모듈을 설치하고 공개하는 일) 역할을 한다.

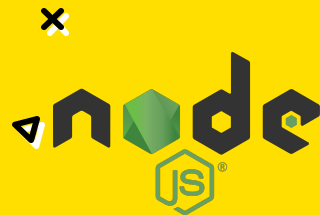
애플리케이션 개발 과정에서 npm을 통해 패키지들의 설치, 제거, 수정을 할 것이다. 터미널상에서 `npm -i` 명령으로 npm 명령에 대한간략한 설명을 볼 수 있다.

표 3.1 자주 쓰이는 npm 명령어

npm 명령어	설명
<code>npm init</code>	Node.js 애플리케이션을 초기화하고 <code>package.json</code> 파일을 생성
<code>npm install</code>	Node.js 패키지 설치
<code>npm publish</code>	사용자가 만든 패키지를 npm 패키지 커뮤니티에 저장 및 업로드
<code>npm start</code>	Node.js 애플리케이션의 실행(이 명령을 사용하려면 <code>package.json</code> 파일이 설정돼 있어야 함)
<code>npm stop</code>	실행 중인 애플리케이션 중지
<code>npm docs &lt;package&gt;</code>	지정된 패키지에 대한 가능한 문서 페이지(웹 페이지) 열기



# Node.js 애플리케이션의 초기화

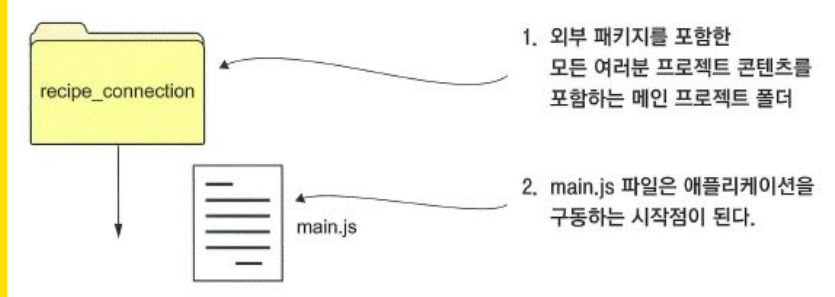


## npm init

모든 Node.js 애플리케이션이나 모듈은 프로젝트 특성을 기술하고 있는 package.json 파일을 포함하고 있으며 프로젝트의 루트 레벨에 존재한다. 일반적으로 이 파일에는 현재 릴리스 버전과 애플리케이션 이름 메인 애플리케이션 파일 등이 기술된다. 이 파일은 node 온라인 커뮤니티에 패키지를 저장하기 위한 npm에 중요한 정보를 제공한다.

```
{
  "name": "recipe_connection",
  "version": "1.0.0",
  "description": "An app to share cooking recipes",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Jon Wexler",
  "license": "ISC"
}
```

← 이름, 버전, 설명, 시작 파일, 사용자 스크립트, 저자, 라이선스를 보여주는 package.json





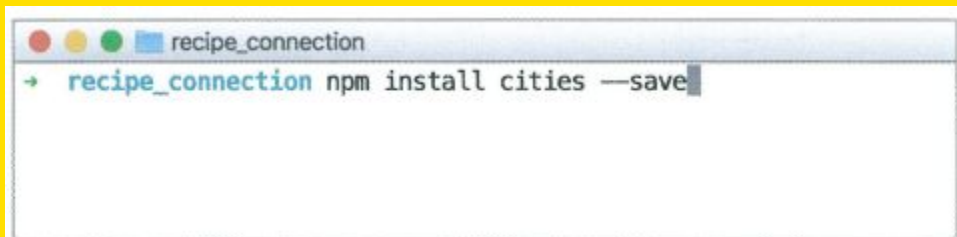


## Node.js 애플리케이션의 초기화



### npm install <package>

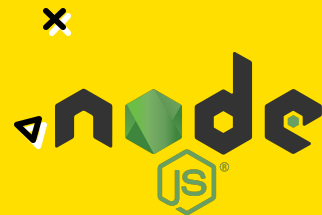
터미널에서 프로젝트 폴더로 이동하고 `npm install cities` 명령을 실행해 `cities` 패키지를 설치할 수 있다(그림 3.1).



```
recipe_connection
→ recipe_connection npm install cities --save
```

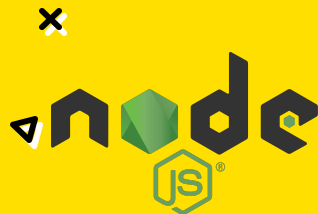
그림 3.1 터미널에서의 패키지 인스톨

이 명령을 실행하면 여러분의 `package.json`에 `cities` 패키지 설치 및 버전 정보를 포함하는 새로운 종속 모듈 섹션이 생긴다(Listing 3.3).





## Node.js 애플리케이션의 초기화

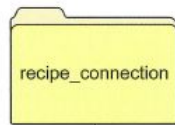


### npm install <package>

또한 이 설치에서 프로젝트 폴더에 새로운 node modules라는 폴더가 생겼다. 이 폴더에는 우리가 설치한 cities라는 패키지 코드가 있다(그림 3.2).

```
{
  "name": "recipe_connection",
  "version": "1.0.0",
  "description": "An app to share cooking recipes",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Jon Wexler",
  "license": "ISC",
  "dependencies": {
    "cities": "^1.1.2"
  }
}
```

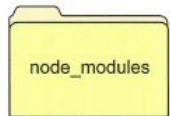
← package.json에서 종속 모듈 부분



main.js



package.json

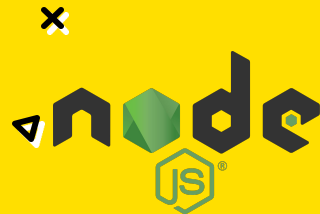


1. 외부 패키지를 포함한 모든 여러분 프로젝트 콘텐츠를 포함하는 메인 프로젝트 폴더
2. main.js 파일은 애플리케이션을 구동하는 시작점이 된다.
3. package.json 파일은 여러분의 특정 애플리케이션과 패키지 종속 관계에 대한 정보와 설정을 포함한다.
4. node\_modules 폴더에는 다운로드된 모든 외부 패키지가 들어 있다.

그림 3.2 node\_modules가 있는 Node.js 애플리케이션의 구조



## Node.js 애플리케이션의 초기화



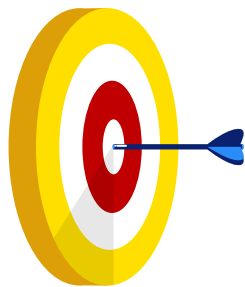
### package.lock에 대해서

아마 프로젝트 디렉터리의 루트 레벨에 package-lock.json 파일도 같이 생성될 것이다 이 파일은 자동으로 생성되며 npm이 패키지 설치 기록 관리와 프로젝트의 종속 관계의 히스토리 및 상태 관리에도 사용된다' 이 파일은 수정해서는 안 된다.

### node\_modules에 대해서

node\_modules 폴더는 점점 사이즈가 커지기 때문에 코드를 온라인에 공유할 때 이 폴더는 포함하지 말 것을 권장한다. 이 프로젝트를 다운로드하는 사람들은 npm install 명령만으로 종속 관계에 있는 모든 파일을 같이 다운로드하게 된다.

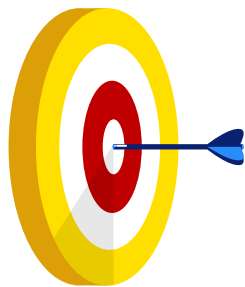
```
// .gitignore
.DS_Store
node_modules
```



## 퀵 체크 3.1

함수나 변수가 동일 모듈 내 다른 곳에서 사용하기 위해 어떤 객체가 쓰이는가?

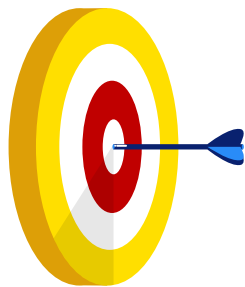
애플리케이션 내에서 모듈의 특성이나 기능의 공유를 위해서는 `exports` 객체가 사용된다. `module.exports`도 같은 기능으로 사용된다.



## 퀵 체크 3.2

패키지를 전역 패키지로 인스톨하려면  
어떤 플래그를 써야 할까?

--global 또는 -g 플래그로 여러분의 컴퓨터에 전역 패키지로 설치한다. 이렇게 설치되면 다른 프로젝트에서도 굳이 명시하지 않더라도 해당 패키지를 사용할 수 있다.



## 퀵 체크 3.3

package.json 파일을 통해 Node.js  
애플리케이션을 초기화하는 명령어는 무엇인가?

npm init으로 애플리케이션을 초기화하고 package.json  
파일 생성을 위한 프롬프트를 표시한다.

# 04

## Node.js에서 웹 서버 만들기

Node.js와 npm을 사용한 기본 웹 서버 생성  
브라우저에서의 요청 처리 및 결과 보내기 코드 작성  
브라우저에서의 웹 서버 실행

p. 79-88



## 웹 서버의 이해

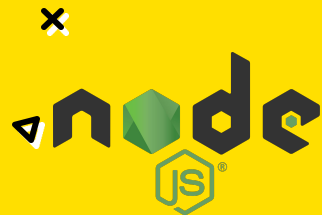


4장에서는 http 모듈의 기초를 다룬다. http 모듈은 인터넷상의 요청을 처리하는 Node.js 코드 라이브러리다. 4장에서는 여러분의 첫 웹 서버를 만들 것이다.

서버는 대부분 Node.js 웹 애플리케이션의 기본이 되며, 이미지나 HTML 웹 페이지를 앱에서 읽어들이어 사용자들에게 보여주게 된다.

웹 서버란 데이터 읽기 및 처리를 통해 인터넷상의 요청에 대한 응답을 위해 설계된 소프트웨어를 말한다.

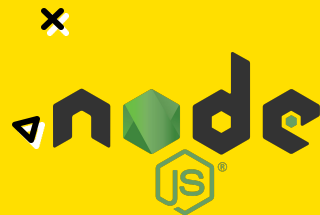
서버와 클라이언트(즉, 우리의 컴퓨터)가 통신하는 방법 중 하나는 HTTP 요청을 사용하는 것이다. 만들어진 요청이 어떤 요청인지, 예를 들어 사용자가 새로운 웹 페이지를 읽어들이는지, 또는 지금 보고 있는 페이지의 업데이트인지 나타낸다. 애플리케이션에서의 사용자 인터랙션 정황은 **요청-응답** 사이클에서 중요한 부분이다.







## 웹 서버의 이해

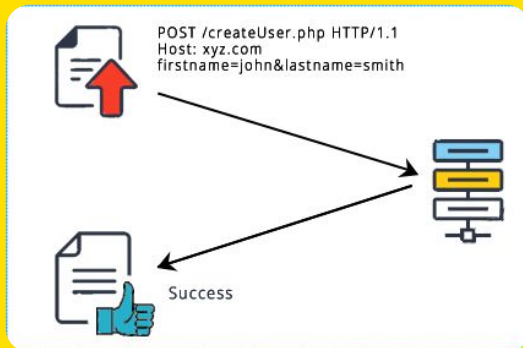


자주 접하고 가장 많이 사용하는 2개의 HTTP 메소드는 다음과 같다.



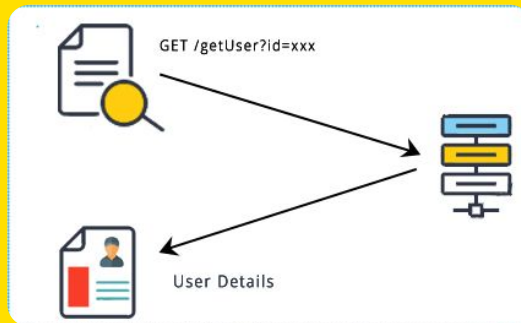
### GET

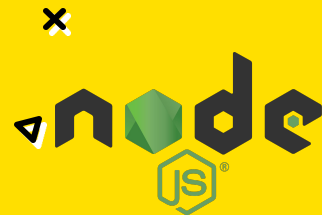
서버로부터 정보를 **요청**한다.  
보통 서버는 브라우저에서 볼 수 있는 콘텐츠(예를 들어 링크를 클릭했을 때 보이는 웹 페이지)로 **응답**한다.



### POST

서버로 정보를 **전송**한다 서버는 데이터 처리(예를 들어 등록 양식에 내용을 채워 제출) 후 HTML 페이지로 **응답**하거나 애플리케이션 내 다른 페이지로 이동시킨다.

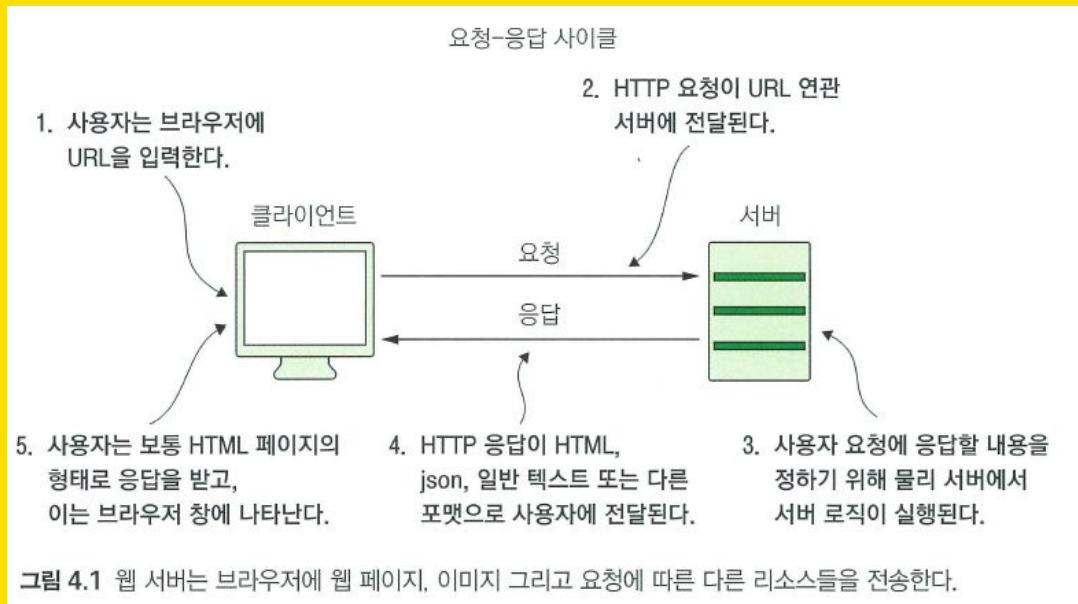




## 웹 서버의 이해

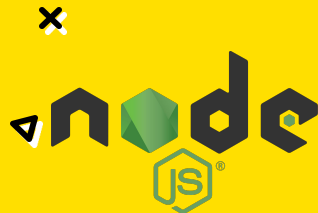


그림 4.1에서 데이터 묶음은 요청의 형태로 애플리케이션 서버로 전송되며, 서버가 **요청**을 처리 후 **응답** 형태로 데이터 묶음을 다시 만들어 보낸다. 이 과정은 인터넷상의 대부분의 상호작용에서 일어난다.



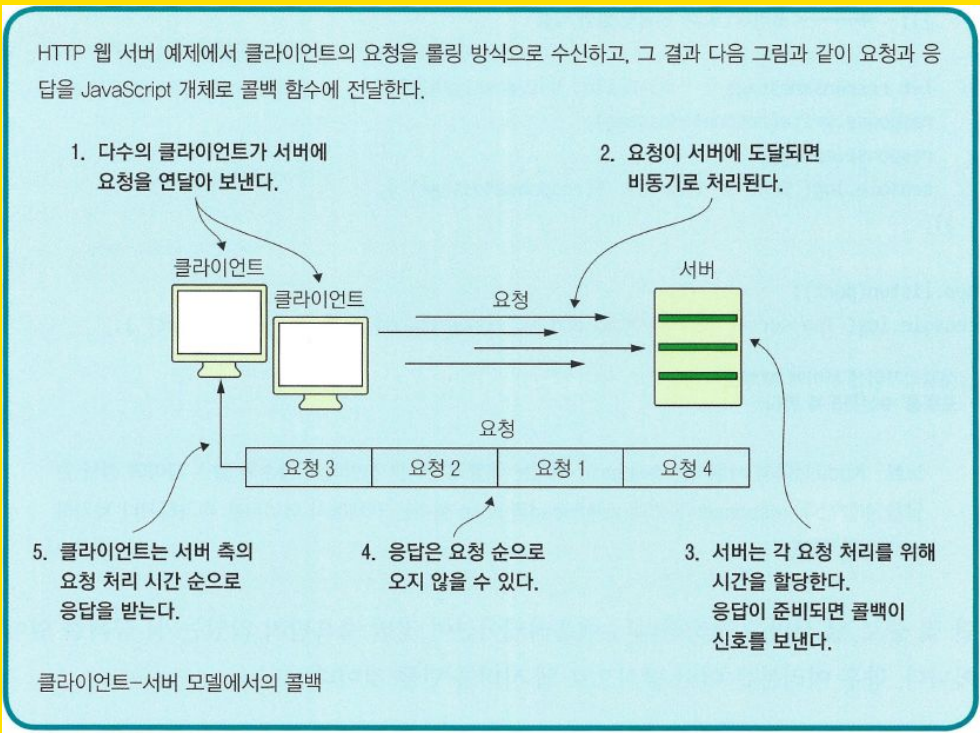


# Node.js에서의 콜백



Node.js의 높은 효율성과 속도 뒤에는 **콜백**이 있다. 콜백은

JavaScript에서는 새로운 개념은 아니며, Node.js에서는 너무나 많이 쓰이고 있어 여기에 언급하려 한다. 콜백은 익명 함수이며 다른 함수가 종료될 때 실행되도록 설정돼 있다. 콜백 사용 장점은 다른 코드를 실행하기 전에 원 함수의 실행 종료를 기다릴 필요가 없다는 것이다.



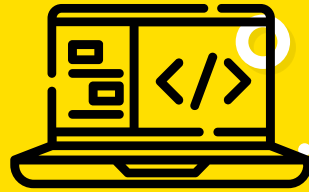


# 애플리케이션 초기화 + 코딩

[https://github.com/  
ut-nodejs/](https://github.com/ut-nodejs/)

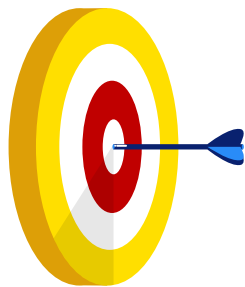
2023.1-UT-NodeJS > 과제 > solutions > 1-node-webserver > practice > JS listing4.1.js > ...

```
1 // listing4.1.js
2
3 /**
4  * simple_server 예시
5  *
6  * 코드: p. 84-85
7  * 설명: p. 83-84 (여기 주석에서도 포함)
8  */
9
10 // 애플리케이션에서 사용할 포트 번호인 3000을 지정한다. (80번호는 HTTP, 443번호는 HTTPS)
11 const port = 3000;
12 // http라는 특정 Node.js 모듈을 가져와 상수로 저장한다.
13 const http = require("http");
14 // http 상태 코드를 나타내는 상수를 제공하려면 http-status-codes 패키지가 필요하다.
15 const httpStatus = require("http-status-codes");
16
17 // http 변수를 HTTP 모듈에 대한 참조로 사용하고 해당 모듈의 createServer
18 // 함수를 사용해 서버를 만들고 결과 서버를 app 이라는 변수에 저장한다.
19 // createServer 함수는 새로운 http.Server 인스턴스를 생성한다.
20 const app = http.createServer((req, res) => { // (req, res) 새로 만들어진 서버 인스턴스로 열린 HTTP 요청(req)을 받을 준비를 하고 HTTP 응답(res)을 전송한다.
21
22   // 시스템은 클라이언트로부터 요청을 받았음을 기록하고
23   console.log("Message received!");
24   // 콜백 함수의 response 매개변수를 사용해 처음 요청을 받은 사용자에게 다시 내용을 보낸다.
25   // 첫 번째 줄에서는 writeHead 메소드를 사용해 응답의 HTTP 헤더의 기본 속성을 정의한다.
26   res.writeHead(httpStatus.OK, { // 이 경우 시스템은 httpStatus.OK를 돌려준다. 이는 응답 코드 200으로 표현된다.
27     // HTTP 헤더는 요청이나 응답에서 전송되는 내용을 설명하는 정보 필드를 포함한다.
28     "Content-Type": "text/html",
29   });
30
31   // 이 코드 블록에 의해 시스템은 로컬 변수인 resMsg를 HTML의 메시지 응답에 할당한다.
32   let resMsg = "<h1>Hello, Everybody!</h1>";
33   // 바로 다음 줄에 write를 써서 HTML 형식의 출력문으로 응답을 시작하고 end를 써서 응답을 종료시킨다.
34   res.write(resMsg);
35   // 응답이 종료될 때에는 반드시 end를 사용해 더 이상 응답 출력은 없다고
36   // 서버에 확인시켜 줘야 하며, 그렇지 않을 경우 서버는 커넥션을 끊지 않고
37   // 계속 연결하고 있어 요청을 받을 수 없는 상태로 남게 된다.
38   res.end();
39   // 이 시점에서 응답로그를 남기게 되면 여러분은 어떤 서버에서 어떤 응답을 보냈는지 확인이 가능하다.
40   console.log(`Sent response: ${resMsg}`);
41 });
42
43 // 애플리케이션 서버에 3000번 포트를 수신하도록 한다.
44 app.listen(port);
45 // 만일 포트 넘버를 지정하지 않는다면 운영체제는 임의로 포트를 지정해줄 것이다
46 // 이 포트 넘버는 웹 브라우저를 통해 웹 서버가 실행되고 있는지 확인하는 데 사용된다.
47 console.log(`Server listening on port: ${port}`);
```



# Coding!

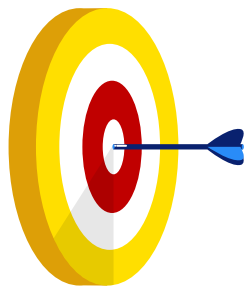
Listing 4.1  
p. 84-85



## 퀵 체크 4.1

웹 서버가 클라이언트로부터 받는 것은 무엇인가? 또 무엇을 돌려주는가?

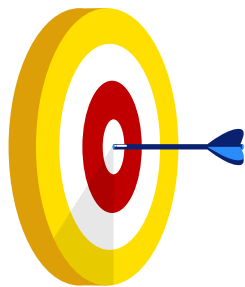
웹 서버는 클라이언트로부터 요청 (request)을 받고  
응답(response)을 돌려준다



## 퀵 체크 4.2

왜 애플리케이션에서 HTTP 서버를 저장하기 위해 `var` 대신 `const`를 써야 할까?

서버는 계속 클라이언트로부터의 수신 대기 상태이기 때문에 서버를 나타내는 이 변수는 재할당을 하면 안 된다. ES6에서는 이런 객체들은 재할당 가능한 `var`가 아닌 `const`로 사용하도록 규칙을 정하고 있다.



## 퀵 체크 4.3

서버가 구동되고 `http://localhost:3000/`으로 접속했다. 어떤 타입의 HTTP 요청으로 접속한 것일까?

현재 애플리케이션 개발 단계에서의 모든 요청은 HTTP GET 요청이다.



# 05

## 수신 데이터 다루기

요청 데이터의 수집과 처리  
curl 명령을 통한 POST 요청 제출  
기본 라우트를 가지는 애플리케이션 제작

p. 89-100

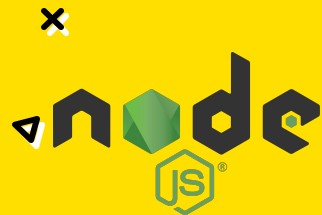


## 서버 코드의 수정



코드에는 콜백 함수 (req, res) => {} 가 있는 server 객체를 갖고 있으며 콜백 함수는 요청이 서버로 들어올 때마다 실행된다. 서버가 실행될 때 localhost:3000으로 접속하고 페이지를 리프레시하면 콜백 함수는 리프레시 될 때에도 호출되므로 모두 2번 호출된다. 다시 말하면 요청을 받을 때 서버는 **요청 및 응답** 객체를 실행할 함수가 들어 있는 코드에 전달한다.

**[노트]** req와 res는 HTTP의 요청 (request)과 응답 (response)을 의미한다. 여기에 다른 변수명으로써도 무방하지만 순서는 기억하기 바란다 요청이 항상 먼저 오며 응답은 그 뒤에 온다. (요청 = yo, 응답 = ng)?



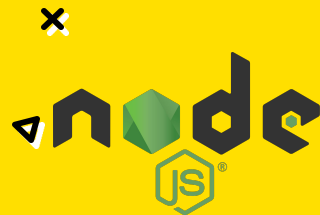


## 서버 코드의 수정

### <Listing 5.1 보기>

서버는 요청 이벤트가 트리거될 때 콜백 함수의 코드를 시작한다.

```
5  /**
6   * listing5.1.js
7   * 이벤트를 리스너를 추가한 간단한 웹 서버 (p. 90)
8   */
9  const port = 3000,
10     http = require("http"),
11     httpStatus = require("http-status-codes"),
12     app = http.createServer();
13
14  app.on("request", (req, res) => { // 요청 수신
15     res.writeHead(http.STATUS_CODES.OK, {
16       "Content-Type": "text/html",
17     }); // 응답 준비
18
19     let resMsg = "<h1>This will show on the screen.</h1>";
20     res.end(resMsg); // HTML로 응답
21  });
22
23  app.listen(port);
24  console.log(`The server has started and is listening on port number: ${port}`);
```





## 요청 데이터의 분석

<Listing 5.2 보기>

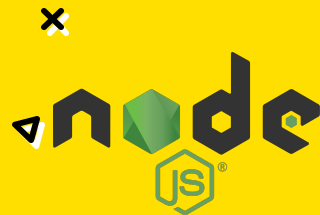


화면상에 콘텐츠를 출력한다는 것은 놀라운 일이지만, 여러분은 수신하는 요청의 종류에 따라 콘텐츠를 변경시키는 것을 원할 것이다. 예를 들면 사용자가 계약이나 등록을 위한 폼 페이지를 방문할 때 상황에 따라 서로 다른 콘텐츠를 보여주는 기능이 필요할 수 있다. 이를 위한 첫 번째 단계는 요청 헤더에서 어떤 HTTP 메소드와 어떤 URL을 사용할지 정하는 것이다.

라우팅 Routing은 애플리케이션이 요청하는 클라이언트에 어떻게 반응하는지 정하는 방식이다. 어떤 라우트 route는 요청 객체에 있는 URL에 연결돼 있는데 이 라우트를 5장에서 만들어본다.

각 요청 객체는 url 속성을 갖고 있으며 req.url 속성을 통해 클라이언트가 어떤 URL 요청을 하고 있는지 파악할 수 있다. 콘솔의 로깅을 통해 이 속성과 다른 두 가지 속성들을 확인해보라. app.on("request") 코드 블록에 다음과 같다.

```
26  /**
27   * listing5.2.js
28   * 요청 로깅 (p. 92)
29   */
30  console.log(req.method); // 사용된 HTTP 메소드의 로그
31  console.log(req.url);    // 요청된 URL의 로그
32  console.log(req.headers); // 요청 헤더
```





## 요청 데이터의 분석

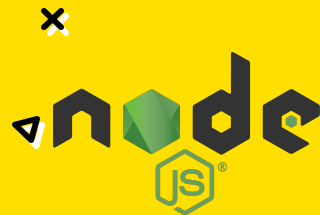
<Listing 5.3 보기>



요청의 일부 개체는 다른 중첩 개체를 포함할 수 있으므로, Listing 5.3과 같이 `JSON.stringify`를 사용해 개체를 더 읽기 쉬운 문자열로 변환할 수 있다. 이 함수는 JavaScript 객체를 인수로 사용해 문자열을 반환한다. 이제 이 함수를 사용하기 위해 로깅 코드를 변경할 수 있다. 예를 들어 `console.log(`Method: ${getJSONString(req.method)}`)`를 사용해 `request` 메소드를 출력할 수 있다.

```
34  /**
35   * listing5.3.js
36   * 요청 데이터 로깅 (p. 92)
37   */
38  const getJSONString = (obj) => {
39    return JSON.stringify(obj, null, 2); // JS 객체의 스트링 변환
40  };
41  console.log(`Method: ${getJSONString(req.method)}`);
```

우리가 다룰 대부분의 요청 종류는 GET 요청이다.





## 요청 데이터의 분석



요청 객체는 Node.js의 대부분의 객체와 마찬가지로 서버와 유사하게 이벤트를 수신할 수 있다. 누군가가 서버에 POST 요청을 하는 경우(서버에 데이터를 보내려고 시도), 그 POST의 내용은 요청한 본문에 저장된다. 서버가 전송되는 데이터의 양을 알지 못하기 때문에 게시된 데이터는 데이터 청크chunk를 통해 http 서버로 들어온다.

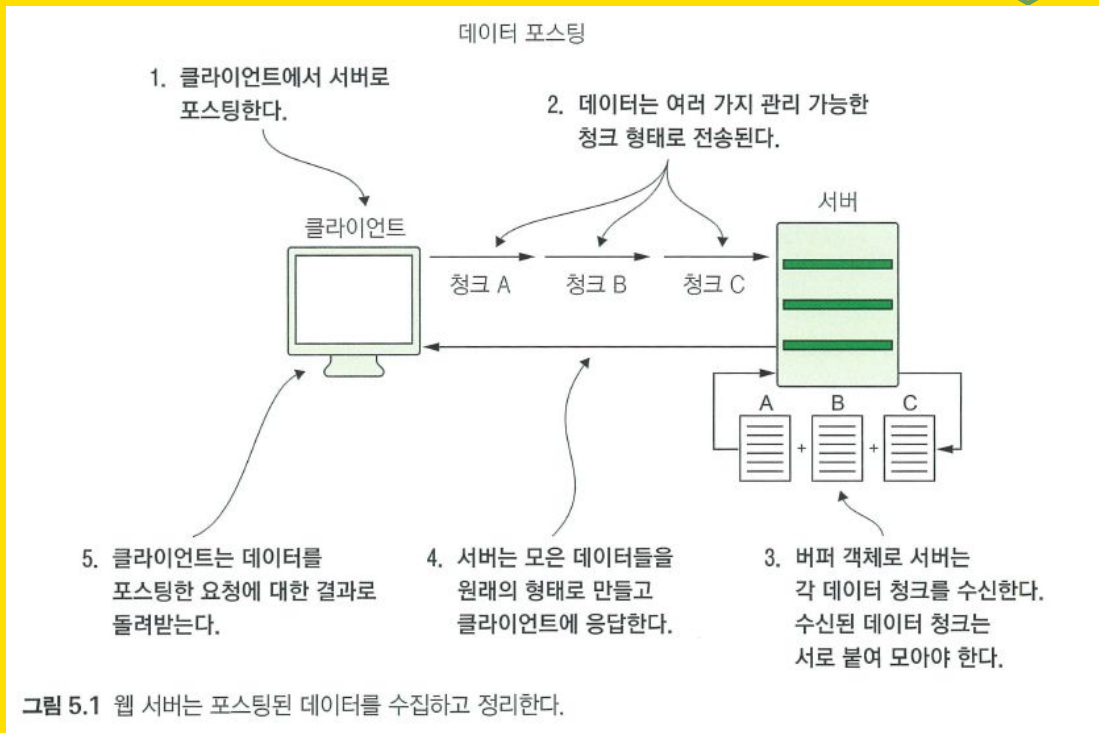
게시된 모든 데이터를 서버로 수집하려면 수신된 각 데이터의 내용을 직접 신하고 정리해야 한다. 다행히도 요청은 특정 data 이벤트를 수신한다. `req.on("data")`는 특정 요청에 대한 데이터를 수신할 때 활성화된다. 이 이벤트 핸들러 외부에서 새로운 배열 `body`를 정의하고 서버에 도착할 때 순차적으로 데이터 청크를 추가해야 한다.

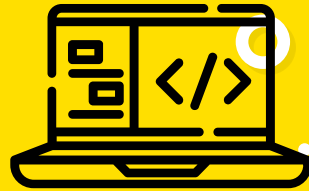




## 요청 데이터의 분석

그림 5.1에서 게시된 데이터의 교환에 주목하라. 모든 데이터 청크가 모아지면 단일 데이터 항목으로 다룰 수 있다.

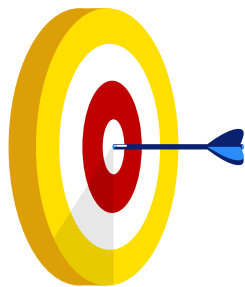




# Coding!

Listing 5.4  
p. 94

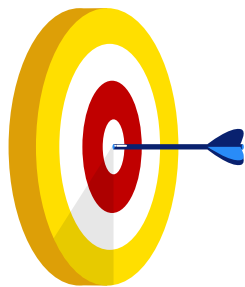




## 퀵 체크 5.1

요청이 들어올 때마다 서버에서  
호출하는 함수 이름은 무엇인가?

콜백 함수라고 한다. 이 함수는 이름을 정의하지  
않기 때문에 익명 함수로 간주된다.



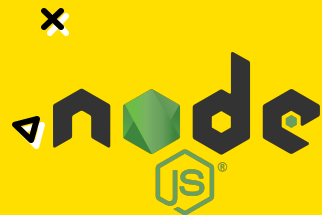
## 퀵 체크 5.2

참 또는 거짓! 제출된 품의 모든 콘텐츠는  
단일 정크(덩어리) 데이터로 전달된다.

**거짓이다.** 데이터는 여러 개의 덩어리로 나뉘서 스트리밍된다.  
이를 통해 서버는 수집된 데이터의 크기나 수신된 데이터  
부분에 기초해 응답할 수 있다.



## 웹 애플리케이션에 라우트 붙이기

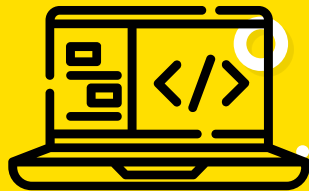


라우트는 특정 URL을 위한 요청에 어떻게 애플리케이션이 응답해야 하는지 정의하는 방식이다. 애플리케이션에서 홈페이지에 대한 요청 라우트는 로그인 정보 제출을 위한 그 것과도 달라야 할 것이다.

사용자가 안내 페이지를 보길 원한다면 사용자들이 /info URL(<http://localhost:3000/>)에서 정보 페이지를 보도록 할 것이다.

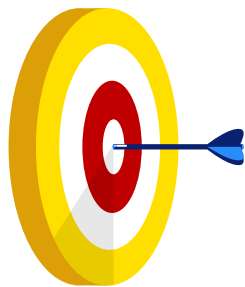
다음 단계는 클라이언트 요청을 체크하고 요청 콘텐츠에 따른 응답 본문을 구성하는 일이다. 이 구조는 애플리케이션 라우팅으로도 알려져 있다. 라우트는 특정 URL 패스를 식별하는데 이 라우트는 애플리케이션 로직에서 타깃이 될 수 있으며, 클라이언트로 보내는 정보를 특정 지을 수 있게 된다.





# Coding 과제!

Listing 5.6  
p. 97-98



## 퀵 체크 5.3

홈페이지의 요청을 위해 어떤 라우트가 필요한가?

라우트 "/"는 애플리케이션의 홈페이지를 의미한다.

# 06

## 라우트와 외부 파일

fs 모듈을 이용한 전체 HTML 파일의 저장  
정적 에셋 저장  
라우트 모듈 생성

p. 101-116



## fs 모듈을 이용한 정적 파일 제공



HTML의 일부분을 사용해 페이지를 구성한다면 귀찮을뿐더러 main.js 파일이 복잡해진다. 대신 나중에 응답할 HTML 파일을 따로 작성한다. 이 애플리케이션 구조에서는 사용자에게 표시할 모든 내용이 views 폴더에 저장되며 표시할 내용을 결정하는 모든 코드는 main.js 파일에 저장된다.

views 폴더에 HTML 파일들을 분류해 놓는 이유는 두 가지다. 일단 모든 HTML 페이지들은 한곳에서 관리될 것이다. 이 규칙은 2부의 웹 프레임워크 부분에서 사용된다.

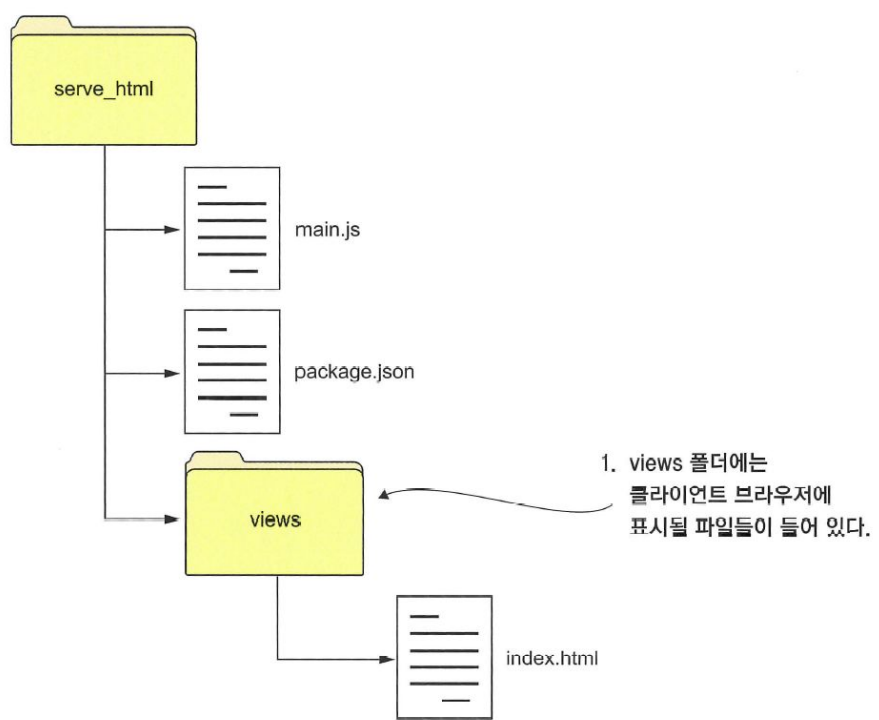
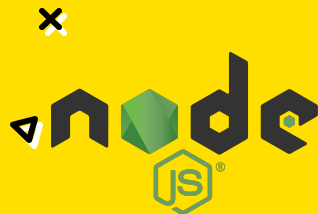


그림 6.1 views가 있는 애플리케이션 구조



## fs 모듈을 이용한 정적 파일 제공

<Listing 6.2 보기>



### 기본 코드를 **index.html** 코드에 추가하기

클라이언트는 애플리케이션을 대신해 일시스템과 상호작용하는 다른 Node.js 코어 모듈인 fs의 도움으로 브라우저에서 이 페이지를 볼 수 있다. fs 모듈을 통해 서버는 index.html에 액세스한다.

```
<!DOCTYPE html>
```

```
<html>
```

```
  <head>
```

```
    <meta charset="utf-8">
```

```
    <title>Home Page</title>
```

```
  </head>
```

```
  <body>
```

```
    <h1>Welcome!</h1>
```

```
  </body>
```

```
</html>
```

views에 기본  
HTML 구조 만들기

```
12 // HTML 파일에 매핑되는 라우트 설정
13 const routeMap = {
14   "/": "views/index.html",
15 };
16
17 http
18   .createServer((req, res) => {
19     res.writeHead(http.STATUS_CODES.OK, {
20       "Content-Type": "text/html",
21     });
22
23     if (routeMap[req.url]) {
24       // 매핑된 파일들의 콘텐츠 읽기
25       fs.readFile(routeMap[req.url], (error, data) => {
26         // 밑에 있는 [노트]를 보세요.
27         res.write(data); // 파일 콘텐츠로 응답
28         res.end();
29       });
30     } else {
31       res.end("<h1>Sorry, not found.</h1>");
32     }
33   })
34   .listen(port);
```





## fs 모듈을 이용한 정적 파일 제공

<Listing 6.3 보기>



### 동적인 읽기와 파일 제공을 위한 'fs'와 라우팅 사용

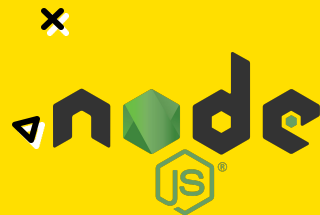
다음 예제에서는 특정 요청의 URL에 대해서만 파일을 보여준다. 누군가가 `http://localhost:3000/sample.html`로 접속했다면 코드는 `/sample.html`을 취하고, `views`를 추가해 `views/sample.html`이라는 문자열을 만든다. 이렇게 구성된 라우트는 사용자가 요청하는 파일을 동적으로 찾을 수 있게 된다.

**[노트]** ES6에서의 문자열 보간 기능은 사용자들에게 `$[...]`을 사용해 텍스트, 숫자, 또는 함수의 결과치를 삽입할 수 있게 한다. 이 새로운 문법을 통해 좀 더 쉽게 문자열과 타입이 다른 데이터를 연결할 수 있다.

```
18 const getViewUrl = (url) => {
19   // URL을 파일 경로에 보간하기 위한 함수 생성
20   return `views${url}.html`;
21 };
22
23 http
24 .createServer((req, res) => {
25   let viewUrl = getViewUrl(req.url); // 파일 경로 문자열 추출
26   fs.readFile(viewUrl, (error, data) => {
27     // 요청 URL을 fs file 탐색에 보간
28     if (error) {
29       // 404 에러 코드 처리
30       res.writeHead(http.STATUS_CODES.NOT_FOUND);
31       res.write("<h1>404: FILE NOT FOUND</h1>");
32     } else {
33       // 파일 내용으로 응답
34       res.writeHead(http.STATUS_CODES.OK, { You, 1 hour
35         "Content-Type": "text/html",
36       });
37       res.write(data);
38     }
39     res.end();
40   });
41 });
42 .listen(port);
```



## fs 모듈을 이용한 정적 파일 제공



## views를 보여주기 위한 라우팅 로직

**[주의]** 요청이 들어올 때 발생할 수 있는 모든 에러에 대해서도 처리해야 한다. 없는 파일에 대한 요청이 들어올 가능성이 높기 때문이다.

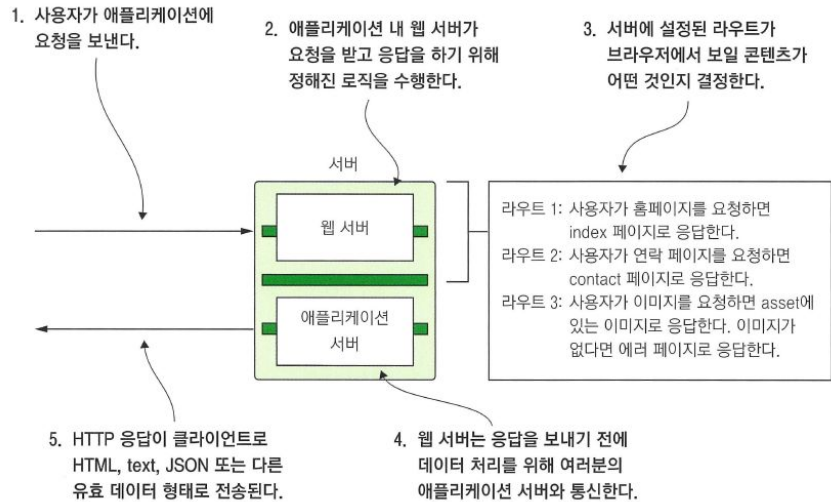
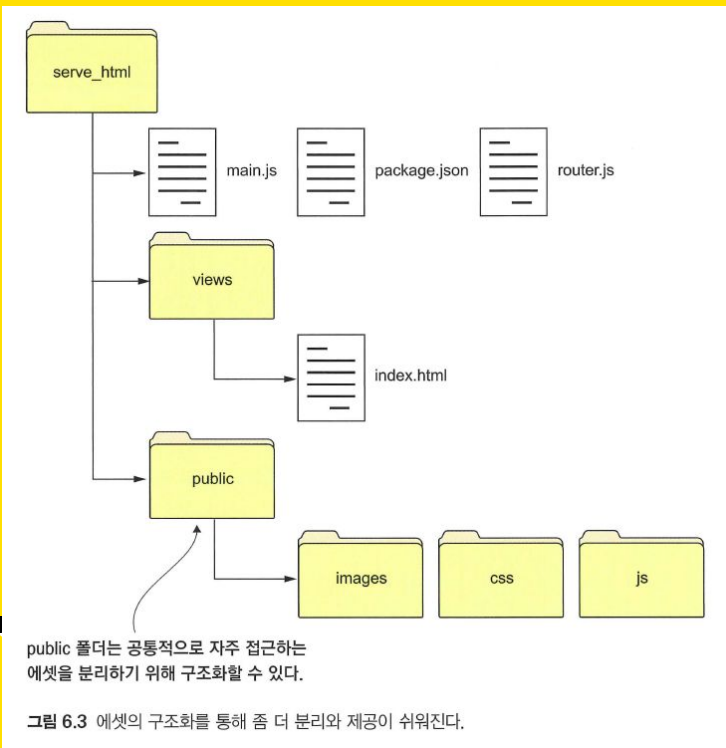
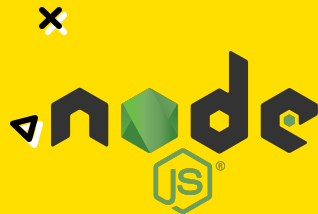


그림 6.2 views를 보여주기 위한 라우팅 로직



## 에셋 제공

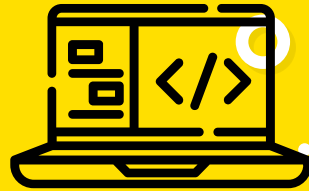


애플리케이션에서의 에셋이란 이미지, 스타일시트, JavaScript 파일을 뜻하며 클라이언트 측에서 뷰 페이지와 함께 동작한다. HTML 파일과 마찬가지로 애플리케이션 내에서 파일타입, 예를 들어 jpg 그리고 .css도 각자 고유 라우트가 필요하다.

이 프로세스의 시작을 위해 프로젝트 루트 디렉터리에 public 폴더를 만들고 모든 에셋을 이 폴더로 이동시킨다. public 폴더 하위에 image, css, js라는 폴더를 각각 만들고 그 쪽으로 해당하는 에셋들로 분류한다.

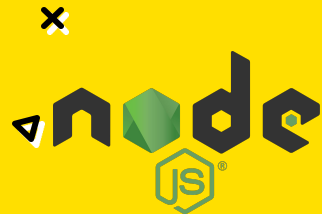
전체 로직은 단지 파일을 읽어들이는 로직을 함수로 구현해놓고 if 구문으로 지정된 파일 요청 유형에 맞게 불러들이는 것뿐이다.





# Coding!

Listing 6.4  
p. 109-110



## 라우트를 다른 파일로 바꿔 연결하기



이번 절에서는 라우트의 수정과 관리를 쉽게 하는 법을 알아본다. 만일 모든 라우트를 if-else 구문으로 구분해 처리한다면 라우트를 변경하거나 삭제할 경우 번거로움과 수정에 의한 부작용도 무시할 수 없을 것이다. 그리고 대상 라우트가 증가하게 되면 사용된 HTTP 메소드로 분리하는 게 더 편함을 알게 될 것이다. 예를 들어 /contact 패스가 GET과 POST 요청에 모두 응답을 할 수 있다면 여러분의 코드는 요청자의 메소드가 식별되자마자 바로 적절한 함수를 찾아가게 된다.

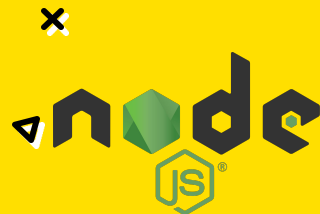
main.js 파일의 사이즈가 커짐에 따라, 이런 요청을 구분해야 할 코드도 점점 복잡해진다. 코드를 운영하다 보면 이런 구분을 위한 코드가 100라인이 훌쩍 넘어가버리는 일이 다반사다.

이런 문제를 완화시키려면 라우트 부분을 router.js라는 새로운 파일로 분리한다. 그리고 이 라우트들을 저장하고 사용하는 방법들을 재구성한다.





## router.js에서 exports 객체에 함수 추가



1. router.js를 필요로 하는 다른 파일들은 exports 객체 내에서 접근 권한을 갖는다.

2. get, post, handle 함수는 router.js 내에서 접근이 가능하다.

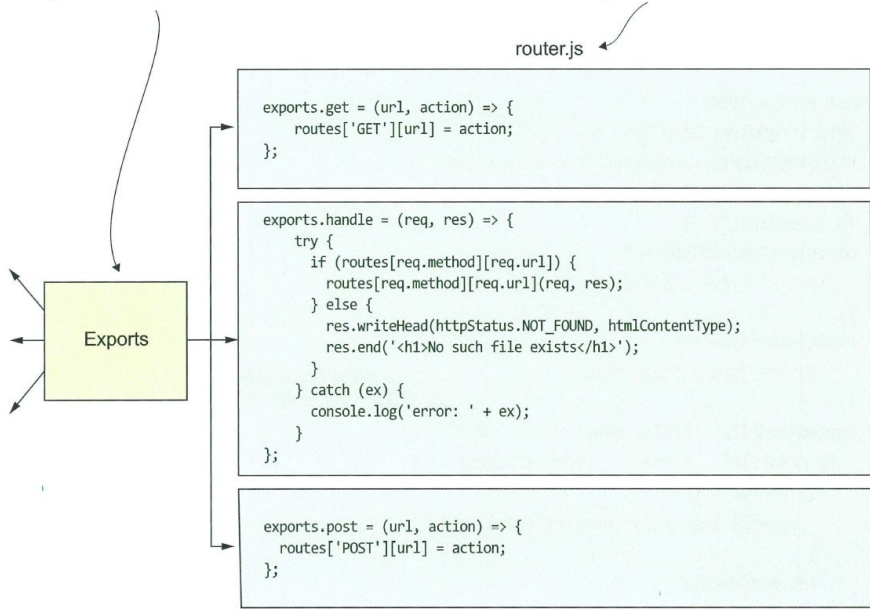


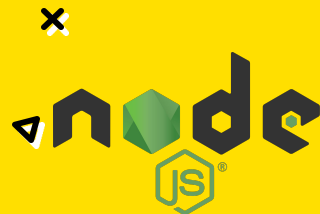
그림 6.4 exports 객체는 다른 파일에서 특정 함수를 접근하도록 한다.

get 또는 post를 호출하면 해당 라우트에 도달할 때 실행할 라우트와 함수를 전달해야 한다. 이 함수는 라우트를 routes 객체에 추가해 등록하며, handle 함수에 의해 사용된다.

그림 6.4에서 주목할 것은 routes 객체는 handle, get 그리고 post 함수에 의해 내부적으로 사용된다는 점이다. 이들 함수는 모듈의 exports 객체를 통해 쓸 수 있다.



# router.js에서 exports 객체에 함수 추가



1. router.js를 필요로 하는 다른 파일들은 exports 객체 내에서 접근 권한을 갖는다.

2. get, post, handle 함수는 router.js 내에서 접근이 가능하다.

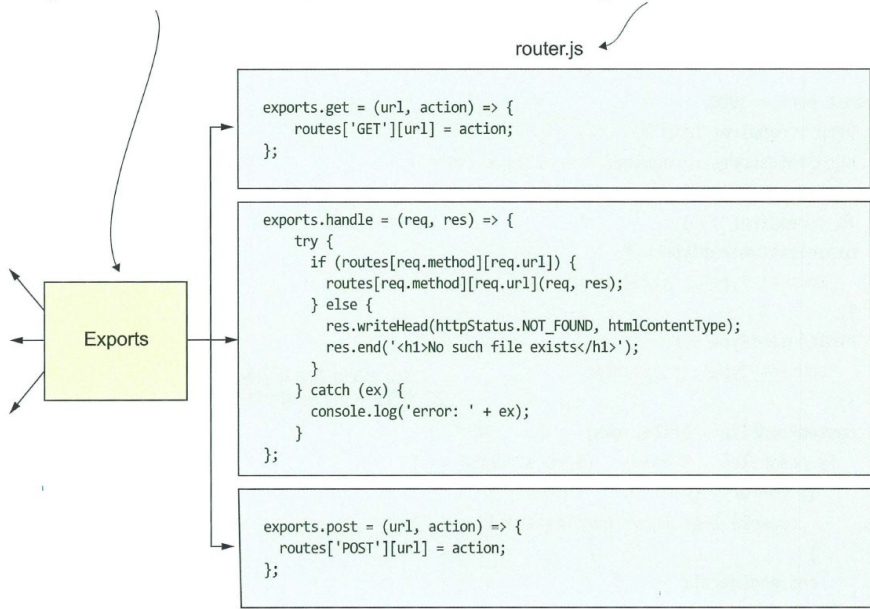
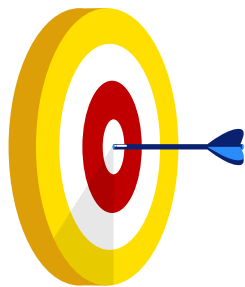


그림 6.4 exports 객체는 다른 파일에서 특정 함수를 접근하도록 한다.

마지막 단계는 main.js로 router.js를 가져오는 것이다.

main.js에서 수행하는 모든 함수 호출에 router를 추가해야 한다. 이 함수는 이제 router.js에 속한다.

이 서버 생성을 통해 모든 요청들은 라우트 모듈의 처리 함수에 의해 처리되며, 처리 함수에는 콜백 함수가 뒤따른다. 이제 `router.get` 이나 `router.post`를 사용해 요청에서 라우트로 사용될 HTTP 메소드를 지정할 수 있다. 두 번째 변수는 요청을 받았을 때 수행되기를 원하는 콜백이다.

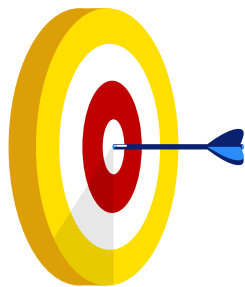


## 퀵 체크 6.1

컴퓨터에 없는 파일을 읽으려고 하면  
어떤 일이 일어날까?

컴퓨터에 없는 파일을 읽으려고 하면 fs 모듈은 콜백에 에러를 보낸다.  
어떻게 이 에러를 처리할지는 독자의 몫이다. 그냥 애플리케이션이  
죽게 둘 수도 있고 로그를 콘솔에 남길 수도 있다.

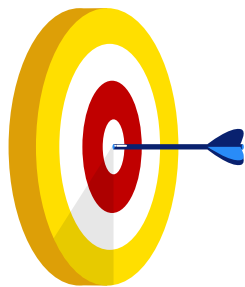




## 퀵 체크 6.2

라우트를 찾지 못한다면 어떻게 응답하게 될까?

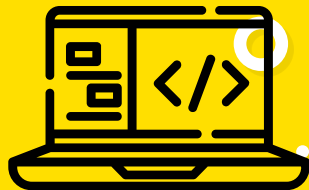
애플리케이션이 요청에 대한 라우트를 찾지 못하게 되면 404 HTTP 상태 코드를 되돌려주게 된다. 이는 찾는 페이지가 없음을 의미한다.



## 퀵 체크 6.3

참 또는 거짓! 모듈의 exports 객체에 추가되지 않은 함수와 객체는 다른 파일에서 계속 액세스할 수 있다.

**거짓이다.** exports 객체는 모듈이 함수와 개체를 공유할 수 있도록 하기 위한 것이다. 객체가 모듈의 exports 객체에 추가되지 않으면 CommonJS에 정의된 대로 해당 모듈에 로컬로 유지된다.



# Coding 과제!

Listing 6.5 + 6.6  
p. 112-113 + 114-115

# 과제 타임!

한번 해보자~