Unit 2 8-11

Express를 통한 웹 개발

UT-NodeJS / 04.07.2023

ut-nodejs.github.io

지금까지 뭐를 배웠어요?

중간고사 미리보기

1. Git and GitHub Classroom
   - Basic Git commands
   - GitHub Classroom
2. Node.js
   - What is Node.js?
   - Node.js REPL
   - Modules, Packages, and Dependencies
   - exports Module / require keyword
   - Callback Functions
3. NPM
   - What is NPM?
   - Some important NPM commands
4. Basic Web Servers
   - HTTP Module
   - request and response objects
   - http-status-codes (200, 404, 500)
   - HTTP Headers
   - HTTP Methods
5. Request-Response Cycle
   - Listening for Requests
   - Analyzing the Request
   - Reading the Request Body
6. Routing and External Files
   - Simple Routing
   - Working with Static Files
   - Simple Route Handling in External Files

다가오는 주제 (오늘)

7. Express Framework
8. Middleware
9. Template Engine
10. MVC Pattern

Contents / 내용

설치와 사용하는 방법

08. Express.js 설정

- Express.js로 Node.js 애플리케이션 설정
- 웹 프레임워크 둘러보기기

09. Express.js에서의 라우트

- 애플리케이션에서 라우트 설정
- 다른 모듈로부터의 데이터로 응답하기
- 요청 URL 매개변수의 수집
- 라우트 콜백으로부터 컨트롤러의 이동

10. 뷰와 템플릿의 연결

- 애플리케이션과 템플릿 엔진의 연결
- 컨트롤러로부터 뷰로의 데이터 전달
- Express.js의 레이아웃 설정

11. 설정과 에러 처리

- 애플리케이션 시작 스크립트 변경
- Express.js를 통한 정적 페이지 서비스
- 에러 처리를 위한 미들웨어 생성

08

Express.js 설정

Express.js로 Node.js 애플리케이션 설정
웹 프레임워크 둘러보기기

p. 133-141

Express.js는 웹 프레임워크이다

웹 프레임워크는 뭘까?

웹 프레임워크는 웹 개발에 사용될 **통상적인 도구들을 제공한다.**

Express .js는

1. 요청 처리를 도와주고,
2. 동적 및 정적 콘텐츠를 제공하며,
3. 데이터베이스를 연결하고
4. 사용자 행동을 추적하는 모듈과 메소드를 제공한다.

8.1 Express.js 패키지 설치

다른 웹 프레임워크도 있다

Express.js는 **2010년** 처음 세상에 나왔으며, 그 후로 다른 쓸 만한 프레임워크도 계속 나왔다.

표 8.1 알아야 할 Node.js 프레임워크

Node.js 프레임워크	설명
Koa.js	Express.js 개발자들이 설계했으며, Express.js에서 제공하지 않은 메소드의 라이브러리에 초점을 맞췄다(http://koajs.com).
Hapi.js	Express.js와 유사한 구조로 설계돼 있으며 코드 작성을 덜 필요로 하는 게 특징이다(https://hapijs.com).
Sails.js	Express.js를 기반으로 만들어졌으며, 더 많은 라이브러리 제공과 함께 커스터마이징 작업이 많이 필요 없도록 돼 있다(https://sailsjs.com).
Total.js	HTTP 코어 모듈 기반으로 만들어졌으며, 빠른 요청과 응답 속도를 목적으로 하고 있다(https://totaljs.com).

[노트] Node.js 웹 프레임워크에 대한 더 많은 정보는 http://nodeframework.com/ 내의 GitHub 저장소소 볼 수 있다.

8.2 첫 Express.js 애플리케이션의 작성

listing 8.1 (p. 137)

```
// listing 8.1.js (p. 137)
"use strict";

const port = 3000,
  express = require("express"), // 애플리케이션에 express 모듈 추가
  app = express(); // 상수 app에 express 애플리케이션 할당

app
  .get("/", (req, res) => {
    // 홈페이지에 GET 라우트 세팅
    /**
     * Listing 8.2 (p. 140)
     * 요청 매개변수의 액세스
     */
    console.log(req.params); // 요청 매개변수의 로그 (토큰, ID, 이름 등)
    console.log(req.body); // POST 요청 바디의 로그 (폼 데이터)
    console.log(req.url); // 요청된 URL의 로그 (index.html)
    console.log(req.query); // Stringified 요청 쿼리의 로그 (name=John&age=30)

    res.send("Hello World!"); // res.send로 서버에서 클라이언트의 응답 발행
  })
  .listen(port, () => {
    // 3000번 포트로 애플리케이션 셋업
    console.log(`The Express server is listening on port: ${port}`);
  });
```

listing 5.1 (p. 90)

```
// listing5.1.js (p. 90)
"use strict";

const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer(); // http 모듈로 웹 서버 생성

app.on("request", (req, res) => {
  /**
   * listing5.2.js
   * 요청 로깅 (p. 92)
   */
  console.log(req.method); // 사용된 HTTP 메소드의 로그 (GET / POST)
  console.log(req.url); // 요청된 URL의 로그 (index.html)
  console.log(req.headers); // 요청 헤더 (User-Agent, Accept, Accept-Language)

  // 요청 수신
  res.writeHead(httpStatus.OK, {
    "Content-Type": "text/html",
  }); // 응답 준비

  let resMsg = "<h1>This will show on the screen.</h1>";
  res.end(resMsg); // HTML로 응답
});

app.listen(port);
console.log(`The server has started and is listening on port number: ${port}`);
```

HTTP 헤더는 요청이나 응답에서 전송되는 내용을 설명하는 정보 필드를 포함한다. 헤더 필드에는 날짜, 토큰, 요청 및 응답의 출처에 대한 정보와 연결 유형을 설명하는 데이터가 포함될 수 있다.

8.3 웹 프레임워크를 이용한 작업

노트 미들웨어 패키지는 Express.js보다 더 작을 수도 있다. 어떤 미들웨어는 코어 애플리케이션에 데이터가 전달되기 전 유입 요청에 대한 보안 체크를 한다.

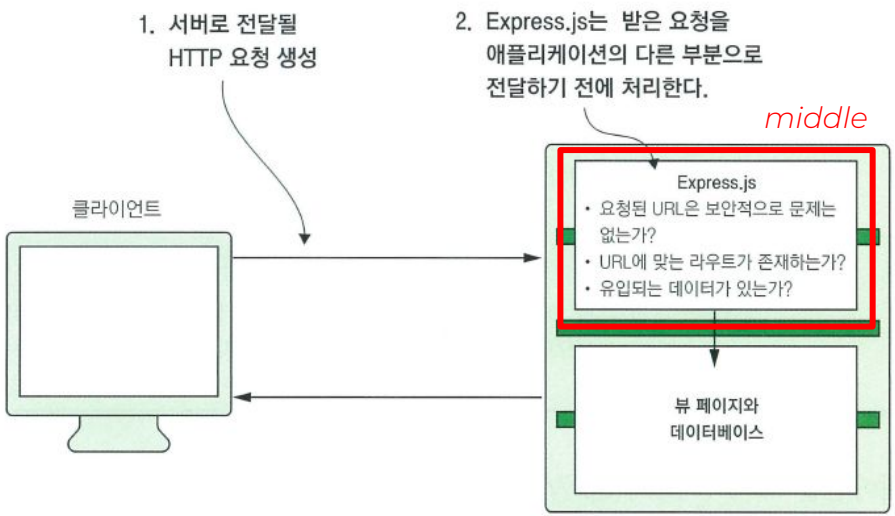


그림 8.1 HTTP 요청과 애플리케이션 코드 사이의 Express.js의 위치

미들웨어란 **(middleware)?**

Express.js와 같은 웹 프레임워크는 보통 **미들웨어**로 운영된다. 프레임워크의 위치가 웹 상의 **HTTP와 Node.js 플랫폼 사이**이기 때문이다.

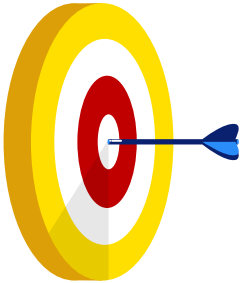
*미들웨어*는 애플리케이션 로직과의 데이터 교환 전에 대기하고, 분석하고 필터링하고 HTTP 통신을 다루는 코드를 일걷는 일반용어다.

request object 데이터 아이템

[팁] 쿼리 스트링은 URL 내에서 키-값 쌍으로 표현되는 텍스트이며 호스트 이름 다음의 물음표 뒤에 붙는다. 예를 들어 http://localhost:3000?name=jon에서는 이름(name = key)와 값(jon = value)를 쿼리 스트링으로 보내는 것이다. 이 데이터는 라우트 처리기에서 추출돼 사용된다.

표 8.2 request object 데이터 아이템

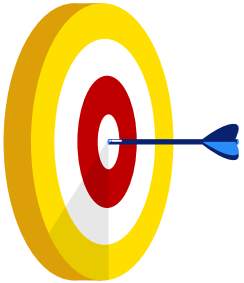
요청 데이터 객체	설명
params	URL로부터 ID와 토큰을 추출하게 해준다. 4부에서 RESTful 라우트를 배우고 나면 e-커머스 사이트에서 어떤 아이템이 많이 요청됐는지, 또는 어떤 사용자 프로필을 봐야 하는지 정의가 가능해진다.
body	요청 콘텐츠의 대부분을 포함한다. 때로는 제출된 폼과 같이 POST 요청으로부터의 데이터를 포함하기도 한다. body 객체로부터 정보들을 쉽게 추출하고 데이터베이스에 저장할 수 있다.
url	방문된 URL에 대한 정보를 제공한다(1부에서 기본 웹 서버의 req.url과 유사)
query	body와 마찬가지로 애플리케이션 서버로 제출된 데이터를 추출할 수 있다. 이 데이터는 반드시 POST 요청으로부터 오는 것은 아니며, URL의 쿼리 스트링 같은 곳에서 요청되기도 한다.



퀵 체크 8.1

Express.js 설치 시 save 플래그를
붙이지 않으면 어떤 일이 일어날까?

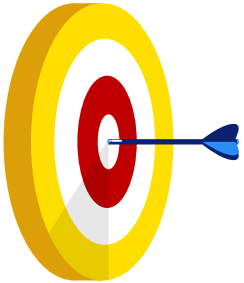
--save 플래그를 붙이지 않으면 Express .js는 dependencies 목록에 들어가지 않게 된다 이렇게 되면 로컬 환경에서 애플리케이션은 동작하지만, 운영 환경 등에 배포할 때 로컬의 Express.js 몰더를 같이 배포하지 않으면 애플리케이션은 동작하지 않게 된다 package.json에 해당 모률이 필요하다는 내용이 없기 때문이다.



퀵 체크 8.2

express와 app 상수의 차이점은 무엇일까?

app은 대부분의 라우트, 다른 모듈로의 액세스, 애플리케이션을 의미한다. express는 애플리케이션에 국한되지 않고 더 넓은 범위의 메소드를 의미한다 텍스트의 파싱이나 분석 등의 기능을 제공한다.



퀵 체크 8.3

왜 대부분의 개발자들은 웹 애플리케이션을 처음부터 직접 만들지 않고 웹 프레임워크를 사용하는가?

웹 프레임워크는 개발자들에게 많은 편의를 준다. 웹 개발은 흥미로운 작업이며 에러를 유발시키는 지루한 작업들이 본 모습은 아니다. 웹 프레임워크를 이용하면 개발자와 기업 모두 더욱 흥미로운 부분에 집중할 수 있다.

09

Express.js에서의 라우트

애플리케이션에서 라우트 설정
다른 모듈로부터의 데이터로 응답하기
요청 URL 매개변수의 수집
라우트 콜백으로부터 컨트롤러의 이동

p. 143-154

9.1 Express.js GET과 POST 라우트

listing 9.3 (p. 145)

```
// listing 9.3.js (p. 145)
"use strict";

const port = 3000,
    express = require('express'),
    app = express();

app.get('/', (req, res) => {
    res.send('This is the HOME page.');
});

app.get('/contact', (req, res) => {
    res.send('This is the CONTACT page.');
});

/**
 * Listing 9.1 (p. 144)
 * Express.js post 메소드를 이용한 요청 처리
 */
app.post('/contact', (req, res) => {
    res.send('CONTACT info submitted successfully.');
});
```

built-in handling

listing 6.5 (p. 112)

```
// 라우트에 따른 콜백 함수를 처리하기 위한 함수 handle의 생성
exports.handle = (req, res) => {
  try {
    if (routes[req.method][req.url]) {
      routes[req.method][req.url](req, res);
    } else {
      res.writeHead(httpStatus.NOT_FOUND, htmlContentType);
      res.end("<h1>No such file exists</h1>");
    }
  } catch (ex) {
    console.log("error: " + ex);
  }
};

// main.js로부터 routes에 등록하기 위한 get 및 post 함수 생성
exports.get = (url, action) => {
  routes["GET"][url] = action;
};

exports.post = (url, action) => {
  routes["POST"][url] = action;
};
```

GET 라우트를 다른 방법으로 기술하면 HTTP 메소드와 Path (URL)를 갖고 있는 애플리케이션의 **엔드포인트**라고 할 수 있다. 1 부에서 유사한 라우트 구조를 만들어봤기 때문에 Express.js에서의 라우트는 친숙하게 보일 것이다.

매개변수 들어있는 라우트

```
24  /**
25   * Listing 9.2 (p. 145)
26   * 경로 매개변수로 응답하기
27   */
28  app.get('/items/:vegetable', (req, res) => { // URL 매개변수를 얻기 위한 라우트 추가
29      let veg = req.params.vegetable.toUpperCase();
30      res.send(`This is the ${veg} page.`);
31  });
```

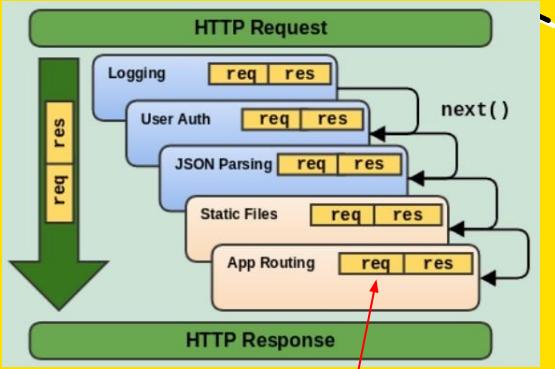
REST위해

라우트 매개변수는 애플리케이션 내 지정 데이터 객체를 확정할 때 간편하다. 사용자 계정과 코스 목록을 데이터베이스에 저장하려고할 때 /users/:id와 /course/:type으로 사용자 정보와 코스의 종류를 액세스할 것이다.

이 구조는 **REST**Representional State Transfer 아키텍처의 개발을 위해 필요하다.
(4부에서 다룰 것이다.)

매개변수 들어있는 라우트

```
33  /**
34   * Listing 9.4 (p. 147)
35   * 요청 경로의 로깅을 위한 Express.js 미들웨어 함수
36   */
37  app.use((req, res, next) => { // 미들웨어 함수의 정의
38      console.log(`request to: ${req.url}`); // 콘솔 화면에 요청 경로를 로깅
39      next(); // next 함수의 호출
40  });
41  /**
42   * [주위] 코드가 종료됐음을 Express.js에 알리기 위해 함수 마지막 부분의 next
43   * 함수의 호출은 필요하다. 이렇게 하지 않으면 요청은 hang 상태로 남아 버린다.
44   * 미들웨어는 순차적으로 처리하기 때문에 next를 호출하지 않으면 흐름이 막혀
45   * blocking 현상이 발생한다.
46   */
```



앞에서 Express.js는 요청 유업 단계와 요청 처리 단계 **중간**에 위치한다고 했다.
다른 정의 미들웨어를 추가하고 싶다면 next 매개변수를 사용하는 미들웨어 함수를 정의하다.

- **next**는 요청-응답 실행 흐름에서 다음 함수를 호출하기 위해 제공된다. 요청이 서버로 유입되면 요청은 미들웨어 함수를 액세스한다. 어디에 사용자 정의 미들웨어 함수를 추가했느냐에 따라 함수가 종료됐음을 Express.js에게 알리고 다음에 어떤 함수가 오든지 간에 계속 이어 실행하기 위해 **next**를 쓸 수 있다.

미들웨어 함수의 역할

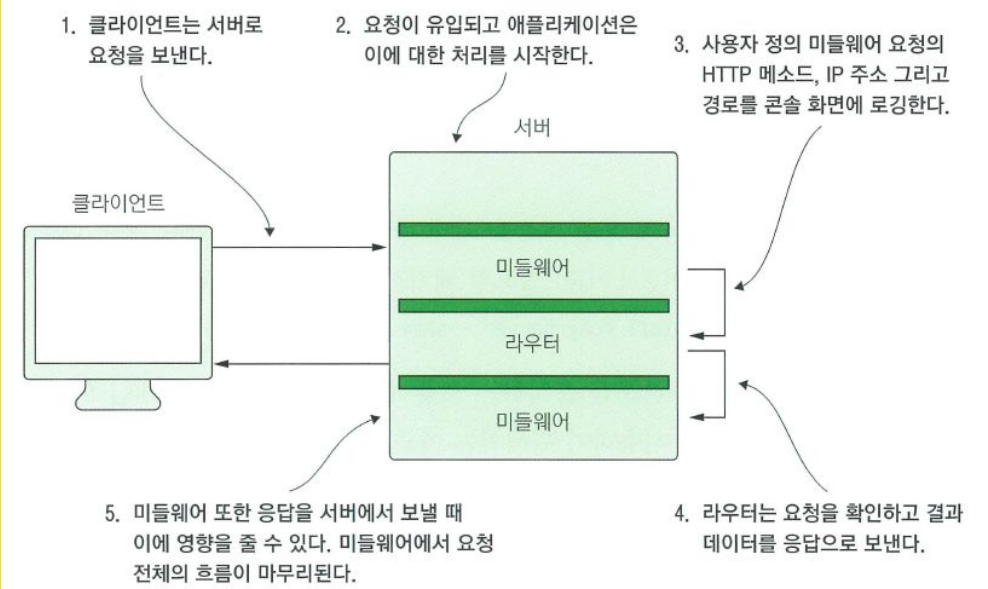
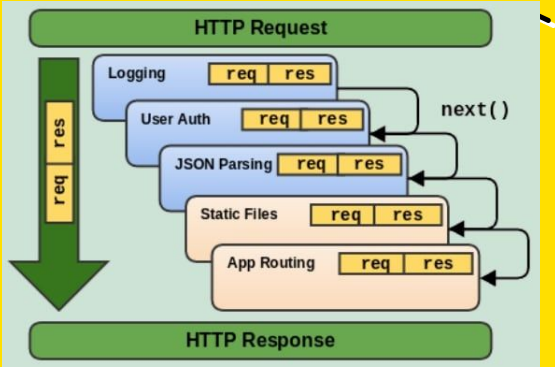


그림 9.1 미들웨어 함수의 역할



또한 미들웨어 함수의 실행을 위한 경로를 지정할 수 있다. 예를 들어 app.use("/items", <callback>).

9.2 요청 데이터의 분석

사용자로부터 데이터를 추출하는 방법은 2가지다.

1. **POST 요청**에서 요청 본문을 통해
2. URL에서의 **쿼리 스트링**을 통해

첫 번째 캡스톤 프로젝트에서 POST 라우트(특정 URL에 대한 포스트를 대기하는 라우트)로 데이터를 제출하는 양식을 만들었다. *하지만* http의 유입 데이터는 **버퍼 스트림** 형태이며 *사람이 읽을 수 있는 형태가 아니다.* 그래서 이를 위한 변환 단계가 별도로 추가돼야 한다.

Express.js는 body 속성을 이용해 요청 본문을 쉽게 추출할 수 있도록 한다. 본문 내용을 읽어들이기 위해 별도의 패키지 설치가 필요하다.

npm install body-parser

(1) 요청 데이터의 분석 (POST)

body-parser는 유입되는 요청 본문을 분석하는 모듈이다. 이 모듈은 Express.js의 프리 패키지 형태로 설치되며 이를 사용하기 위해서는 애플리케이션에서 사용을 요청하면 된다.

코드를 보면 먼저 body-parser 모듈을 요청하고 이를 상수에 할당한다. Express.js의 **app.use**를 통해 URL 인코딩(보통 utf-8 인 POST 양식) JSON 양식으로 된 유입 요청의 파싱을 지정할 수 있다.

Book

```
/**
 * Listing 9.5 (p. 149)
 * 요청 본문으로부터 포스팅된 데이터 캡처
 */
app.use(
    express.urlencoded({
        extended: false
    })
); // Express.js에 body-parser를 이용해 URL-encoded 데이터를 파싱하도록 요청
app.use(express.json());

app.post('/', (req, res) => { // 홈페이지를 위한 새로운 라우트 생성
    console.log(req.body); // 요청 본문의 로깅
    console.log(req.query);
    res.send('POST successful!');
});
```

expressjs.com

```
var express = require('express')
var bodyParser = require('body-parser')

var app = express()

// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }))

// parse application/json
app.use(bodyParser.json())

app.use(function (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.write('you posted:\n')
  res.end(JSON.stringify(req.body, null, 2))
})
```

(1) 요청 데이터의 분석 (POST)

Git Bash

```
Aaron@DESKTOP-PTD9GI3 MINGW64 ~/Desktop/UT/Classes/2023.1 NodeJS
$ curl --data "first-name=Aaron&last-name=Snowberger" http://localhost:3000
POST successful!
Aaron@DESKTOP-PTD9GI3 MINGW64 ~/Desktop/UT/Classes/2023.1 NodeJS
$
```

Terminal

```
PS C:\Users\Aaron\Desktop\UT\Classes\2023.1 NodeJS\과제\solutions\3-express-webserver-soln\lesson-9> node .\9.3.js
The server is running on port: 3000
request to: /favicon.ico
request to: /
[Object: null prototype] {
  'first-name': 'Aaron',
  'last-name': 'Snowberger'
}
{}
```

이 프로세스는 **POST 메소드**와 URL의 지정만큼이나 간단하다.
포스팅된 요청 객체와 body 속성 양식의 콘텐츠를출력한다.

curl --data "firstname=Jon&lastname=Wexler" http://localhost:3000로 입력해 코드를 테스트해보자.

(2) 요청 데이터의 분석 (URL 매개변수를)

```
48  /**
49   * Listing 9.5 (p. 149)
50   * 요청 본문으로부터 포스팅된 데이터 캡처
51   */
52  app.use(
53      express.urlencoded({
54          extended: false
55      })
56  ); // Express.js에 body-parser를 이용해 URL-encoded 데이터를 파싱하도록 요청
57  app.use(express.json());
58
59  app.post('/', (req, res) => { // 홈페이지를 위한 새로운 라우트 생성
60      console.log(req.body); // 요청 본문의 로깅
61      console.log(req.query);
62      res.send('POST successful!');
63  });
```

app.get('/', ...)으로 옮김

또 다른 데이터 수집 방법은 URL 매개변수를 이용하는 것이다. 추가 페이지 없이 Express.js는 **URL 경로의 끝 부분에 물음표**를 붙이고 이 뒤에 연결된 값들을 취할 수 있게 한다. 이 값들을 **쿼리 스트링 (query string)**이라고 하며, 사이트에서 사용자의 행동을 추적하거나 사용자 방문 페이지의 일시적인 정보 저장을 위해 사용된다.

http://localhost:3000?cart=3&pagesVisited=4&utmcode=837623로 한번 테스트해보자.
이 쿼리 스트링을 서버에서 보려면 **console.log(req.query)**를 main.js의 미들웨어 함수에 추가하라.

```
PS C:\Users\Aaron\Desktop\UT\Classes\2023.1 NodeJS\과제\solutions\3-express-webserver-soln\lesson-9> node .\9.3.js
The server is running on port: 3000
{ cart: '3', pagesVisited: '4', utmcode: '837623' }
```

9.3 MVC의 사용

MVC 아키텍처는 **모델(M), 뷰(V), 컨트롤러(C)**라는 애플리케이션의 3가지 메인 기능에 포커스를 맞춘다.

표 9.1 Model–View–Controller의 세부 정의

뷰(Views)	애플리케이션에서 데이터를 이용해 화면 표시를 한다. 3부에서 모델에 관해 알아보고 자신의 것을 만들어본다.
모델(Models)	애플리케이션 및 데이터베이스에서 객체지향 데이터를 나타내는 클래스다. 레시피 애플리케이션에서 고객 주문을 나타내는 모델을 생성할 수 있다. 이 모델에서 주문에 포함해야 하는 데이터와 해당 데이터에서 실행할 수 있는 함수 유형을 정의한다.
컨트롤러(Controllers)	뷰와 모델 사이의 접착제 역할을 한다. 컨트롤러는 요청 본문 데이터를 처리하는 방법과 모델 및 뷰를 포함시키는 방법을 결정하기 위해 요청의 유입 시 대부분의 로직을 수행한다. Express.js 애플리케이션에서 라우트 콜백 함수가 컨트롤러 역할을 하기 때문에 이 프로세스에는 익숙해질 것이다.

9.3 MVC의 사용

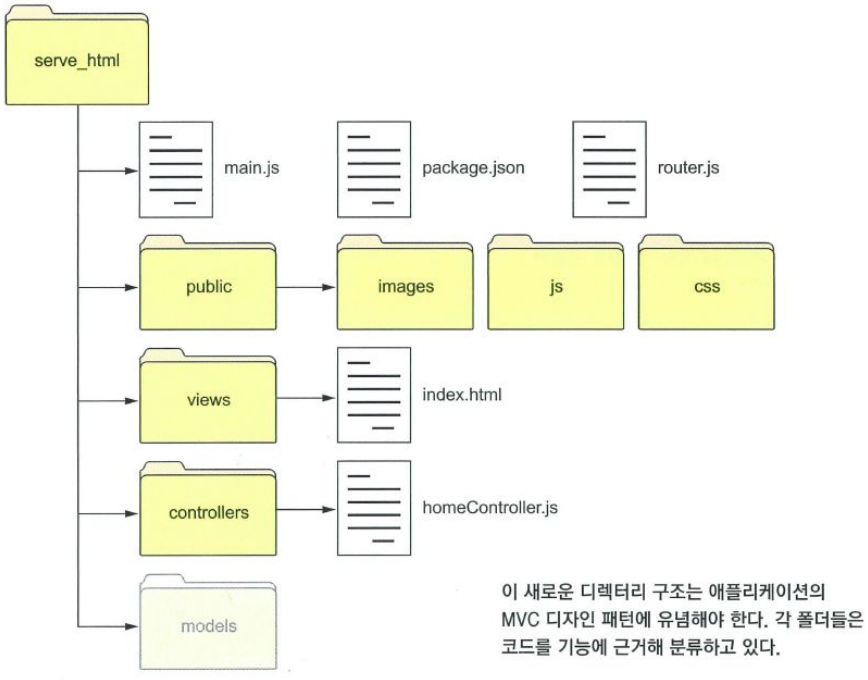


그림 9.2 Express.js의 MVC 구조

MVC 디자인 패턴을 따르려면 콜백 함수를 해당 함수의 용도를 나타내는 모듈로 분리하자. 예를 들어 사용자 계정 생성, 삭제 또는 변경과 관련된 콜백 함수는 controllers 폴더 내의 usersController.js 파일로 이동시킨다. 홈페이지 또는 기타 정보 페이지를 렌더링하는 라우트 함수는 규칙에 따라 homeController.js 에 들어갈 수 있다.

9.3 MVC의 사용

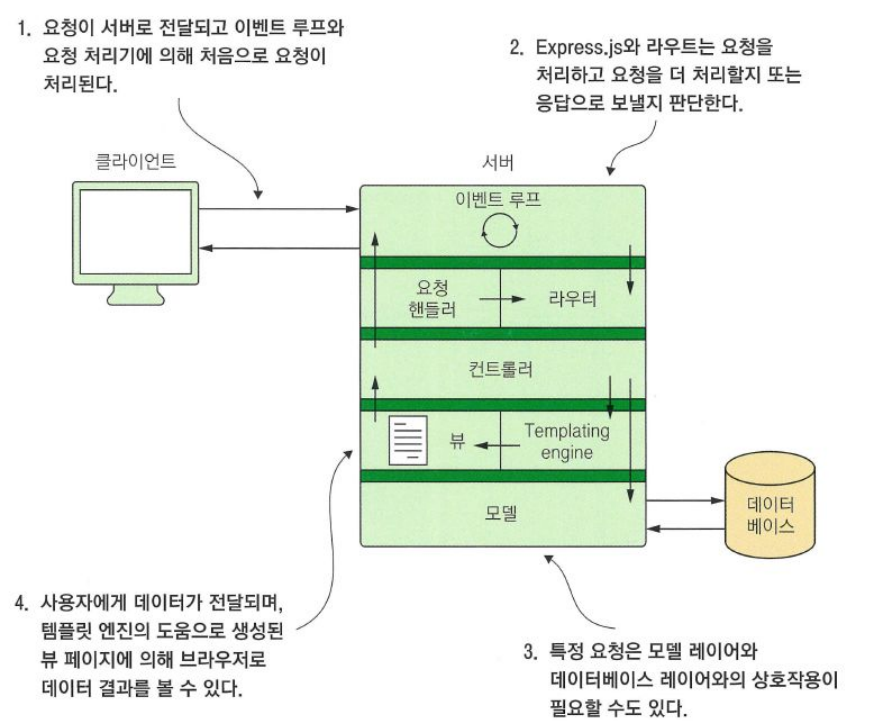


그림 9.3 라우트 피딩 컨트롤러를 사용해 Express.js는 MVC 구조를 따른다.

그림 9 .3은 요청을 처리하고 애플리케이션의 컨트롤러에도 요청을 전달하는 애플리케이션 레이어로서의 Express.js를 보여준다. 콜백은 뷰를 렌더링할지 또는 일부 데이터를 클라이언트에 다시 보내야 하는지 결정한다.

라우트 콜백 함수를 흠 컨트롤러로 옮겨 해당 모듈의 export 객체에 추가하자.
예를 들어 vegetable 매개변수로 응답하는 라우트는 흠 컨트롤러로 이동해 Listing 9.6과 같이 표시할 수 있다. homeController .js에서 exports.sendReqParam

9.3 MVC의 사용

```
// homeController.js
"use strict";

/**
 * Listing 9.6 (p. 153)
 * 콜백 함수를 homeController.js의 홈 컨트롤러로 이동
 */
exports.sendReqParam = (req, res) => { // 지정 라우트 요청 처리를 위한 함수 생성
    let veg = req.params.vegetable.toUpperCase();
    res.send(`This is the ${veg} page.`);
};
```

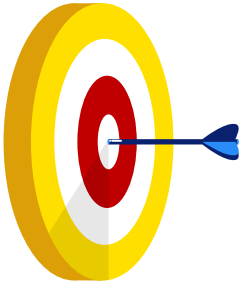
라우트 콜백 함수를 **홈 컨트롤러로 옮겨** 해당 모듈의 export 객체에 추가하자.

sendReqParam은 변수 이름이며 함수를 설명하는 다른 이름을 선택할수도있다.

예를 들어 vegetable 매개변수로 응답하는 라우트는 홈 컨트롤러로 이동해 Listing 9.6과 같이 표시할 수 있다. homeController.js에서 **exports.sendReqParam**을 콜백 함수로 지정한다.

```
/**
 * Listing 9.7 (p. 153)
 * 컨트롤러 함수의 참조를 통한 콜백의 대체
 */
// "/items/:vegetable"로의 GET 요청 처리
app.get('/items/:vegetable', homeController.sendReqParam);
```

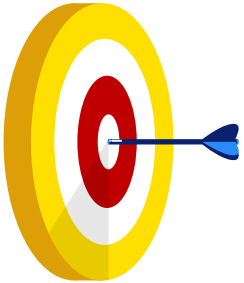
main.js로돌아와서 Listing 9. 7과같이 라우트를 변경한다.



퀵 체크 9.1

Express.js에서 use 메소드는 어떤 역할을 하나?

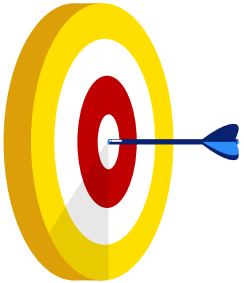
use 메소드는 Express에서 같이 사용될 미들웨어 함수를 정의하는 데 사용된다.



퀵 체크 9.2

Express.js에서 유입된 요청 본문 데이터를
파싱하는 데 필요한 추가 패키지는 무엇인가?

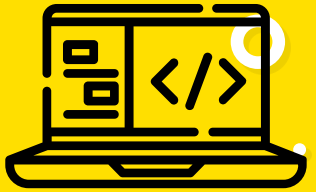
body-parser 패키지는 서버로 유입되는 데이터의 파싱 라이브러리 코드를 제공한다. 다른 패키지들은 미들웨어처럼 동작하면서 동일한 작업을 한다.



퀵 체크 9.3

MVC에서 컨트롤러의 역할은 무엇인가?

컨트롤러는 모델(Models)과 통신하고, 코드 로직을 수행하며, 서버 응답에서 뷰(Views)를 렌더링하도록 요정함으로써 데이터를 처리한다.



Coding!

Listing 9.7 + homeController.js

p. 153

10

뷰와 템플릿의 연결

애플리케이션과 템플릿 엔진의 연결
컨트롤러로부터 뷰로의 데이터 전달
Express.js의 레이아웃 설정

p. 155-164

뷰와 템플릿의 연결

<header>

<body>

<footer>

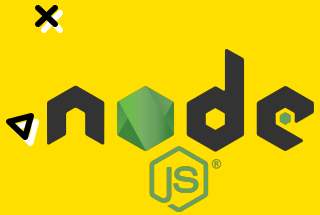
그림 10.2 name page의 예제 뷰

고려 사항
애플리케이션 페이지가 어떻게 보일지 배치를 지원하는 와이어프레임이 몇 가지가 있으며 **많은 페이지가 이를 통해 구성 요소를 공유하고 있는 점**에 주목해야 한다.

홈페이지와 연락처 페이지는 **모두 동일한 내비게이션 바**를 사용한다. 각 뷰의 내비게이션 바를 나타내는 HTML을 다시 작성하는 대신 코드를 한 번 작성하고 각 뷰에서 재사용하려고 할 것이다. Node.js 어플리케이션에서 **템플릿** 기능을 사용하면 이것이 가능해진다.

모든 애플리케이션 페이지에 대해 단일 레이아웃을 렌더링하거나 partials라는 코드 스니펫에서 **뷰 콘텐츠를 공유할 수 있다**.

10.1 템플릿 엔진의 연결



템플릿 사용으로 뷰에 동적 데이터를 삽입하는 코딩을 할 수 있다.

이 책에서는 특수 구문을 사용해 페이지에 삽입된 JavaScript 객체의 형태로 **EJS** 데이터를 사용해 HTML로 뷰를 작성한다. 이 파일의 확장자는 **.ejs** 이다. (ejs.co)

EJS와 비슷한 많은 템플릿 언어가 있다.

표 10.1 템플릿 엔진

템플릿 엔진	설명
Mustache.js	Handlebars.js가 제공하는 사용자 정의 헬퍼(helpers)가 없는 이 템플릿 엔진은 간단하고 가벼우며 JavaScript 이외의 많은 언어로 컴파일이 가능하다(https://mustache.github.io/).
Handlebars.js	EJS와 비슷한 기능을 하는 이 템플릿 엔진은 뷰에 동적 콘텐츠를 삽입하기 위해 중괄호나 handlebar를 사용하는 데 중점을 둔다(http://handlebarsjs.com/).
Underscore.js	다른 JavaScript 함수 및 라이브러리 외에도 이 엔진은 사용자 정의가 가능한 구문과 기호로 템플릿을 제공한다(http://underscorejs.org/).
Pug.js	Ruby의 Jade와 유사한 구문을 제공하며 단순화를 위해 HTML 태그 이름을 약자로 사용하며 들여쓰기에 민감한 게 특징이다(https://pugjs.org).

10.1 템플릿 엔진의 연결

템플릿 엔진은 Express.js가 뷰를 처리하고 **브라우저에서 읽을 수 있는 HTML** 페이지로 변환하는 데 사용하는 엔진이다. HTML 이 아닌 모든 행은 HTML로 변환되며, 내장 변수가 있는 곳에서 값이 렌더링된다.

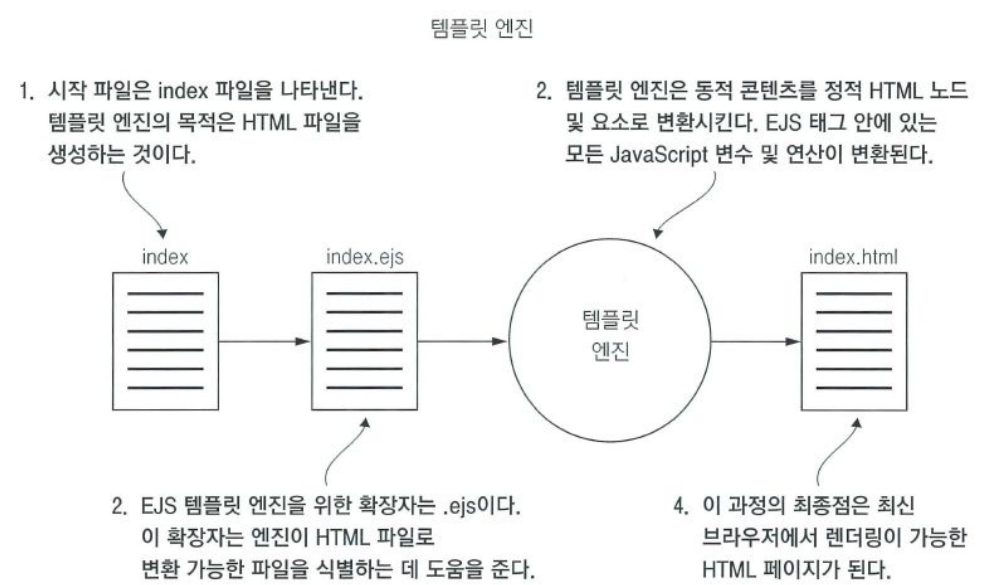


그림 10.1 EJS의 HTML로의 변환

EJS
Embedded
JavaScript

10.1 템플릿 엔진의 연결

새로운 프로젝트에서 애플리케이션을 초기화하고 종속 모듈로 express와 ejs를 설치하자.

npm install express ejs

이제 ejs 패키지가 설치됐으므로 Express.js 애플리케이션에 ejs를 템플릿용으로 사용할 계획임을 알려야 한다. 이렇게 하려면 main.js의 요청 행 아래에 app.set("view engine", "ejs")를 추가한다.

Set 메소드
set은 종종 애플리케이션에서 사용되는 사전 정의된 구성 변수에 값을 할당하는 데 사용된다 애플리케이션 설정 속성 *(application setting properties)*이라고 하는 변수다.

앞에서 애플리케이션 포트를 3000(기존 포트 번호)으로 설정했지만 애플리케이션이 온라인으로 배포될 때 포트 번호는 바뀌게 된다. app.set을 사용하면 애플리케이션에서 재사용할 키에 값을 할당할 수 있다. 예를 들어 **app.set("port", process.env.PORT || 3000)**은 이전 값이 정의되지 않은 경우 port를 환경변수 PORT 값 또는 3000으로 설정한다.

이 설정값을 사용하려면 애플리케이션의 main.js 파일 끝에 있는 하드코딩된 3000을 app.get("port")로 바꿔야 한다. 마찬가지로 app.get("view engine")을 실행할 수 있다. 이제 console.log(`Server running at http://localhost:**${app.get("port")}**`);)와 같은 좀 더 동적인 명령문으로 대체할 수도 있다.

10.1 템플릿 엔진의 연결

views 폴더에서 **index.ejs** 파일을 만들고 **EJS 구문 <% %>**을 사용해 뷰 내에서 변수를 정의하고 할당한다. 이 문자 내의 모든 것은 유효한 JavaScript로 실행된다. HTML의 각 줄에는 포함된 변수가 들어 있다. **<%= %>**를 사용하면 HTML 태그 내에 해당 변수의 값을 인쇄할 수 있다.

```
1  <!-- index.ejs (p. 159) -->
2
3  <!-- EJS language support 설치 -->
4  <!-- https://marketplace.visualstudio.com/items?itemName=DigitalBrainstem.javascript-ejs-support -->
5
6  <% let name = "Aaron"; %><!-- EJS의 변수 할당 정의 -->
7  <h1>Hello, <%= name %></h1><!-- HTML내에서의 내장 변수 사용 -->
8
```

homeController.js

```
4  /**
5   * Listing 10.2 (p. 159)
6   * 컨트롤러 동작에 의한 뷰 렌더링
7   */
8  exports.respondWithName = (req, res) => {
9      res.render('index'); // 사용자 정의 EJS 뷰를 사용한 응답
10 }
```

10.2 컨트롤러로부터의 데이터 전달

템플릿이 렌더링됐으므로 **뷰에서 변수를 직접 정의하는 대신 컨트롤러에서 뷰로 데이터를 전달하는 형태가 이상적이다.** 그렇게 하기 위해 index.ejs 내에서 name 변수를 정의하고 할당하면서 H1 태그와 그 EJS 콘텐츠를 유지하는 내용의 행을 제거하자.

라우트를 변경해 경로에서 매개변수를 가져온 다음 해당 매개변수를 뷰로 보낸다. 라우트의 코드는 app.get("/name/:myName", homeController.respondWithName) 과 같이 된다. 이제 라우트는 /name 다음에 매개변수를 붙여 사용한다.

```
15  // main.js
16  app.get('/name/:myName', homeController.respondWithName);
```

```
1  <!-- index.ejs (p. 159) -->
2  <h1>Hello, <%= name %></h1><!-- HTML내에서의 내장 변수 사용 -->
3
```

```
12  // homeController.js
13  exports.respondWithName2 = (req, res) => {
14      let paramsName = req.params.myName; // 요청 매개변수로 지역 변수 할당
15      res.render('index', { name: paramsName }); // 렌더링된 뷰로 지역 변수 전달
16  };
```

10.3 요소 및 레이아웃 설정

<header>

<body>

<footer>

그림 10.2 name page의 예제 뷰

이 절에서는 뷰를 약간 다르게 설정해 **여러 페이지에서 뷰 내용을 공유하겠다.**

시작하려면 애플리케이션 레이아웃을 만든다. 레이아웃은 뷰가 렌더링되는 셸이다. 레이아웃은 웹사이트를 탐색할 때 페이지마다 **변경되지 않는 콘텐츠**라고 생각하라. 원한다면 페이지 바닥글 또는 네비게이션 바는 동일하게 유지시킬 수 있다. 이 컴포넌트를 위한 HTML의 재작성 대신, 다른 뷰가 공유할 수 있는 layout.ejs 에 추가한다.

npm install express-ejs-layouts

```
const layouts = require("express-ejs-layouts"); // main.js에서
app.use(layouts);
```

10.3 요소 및 레이아웃 설정

<%- include('partials/header') %>

<%- body %>

<%- include('partials/footer') %>

그림 10.2 name page의 예제 뷰

그런 다음 views 폴더에 **layout.ejs** 파일을 만든다. **body 키워드**는 다른 뷰의 내용을 채우기 위해 Express.js와 레이아웃 express-ejs-layouts 에 의해 사용된다.

<%- include('header') %>
<%- body %>
<%- include('footer') %>

```
1  <!-- layout.ejs (p. 162) -->
2
3  <body>
4      <header id="nav">
5          <%- include('partials/navigation') %>
6      </header>
7      <%- body %><!-- 표준 HTML로 본문 둘러싸기 -->
8      <footer id="footer">FOOTER</footer>
9  </body>
10
```

10.3 요소 및 레이아웃 설정

<%- include('partials/header') %>

<%- body %>

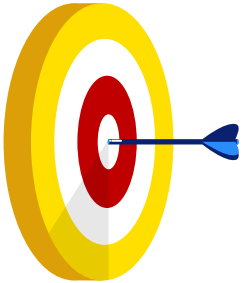
<%- include('partials/footer') %>

그림 10.2 name page의 예제 뷰

요소들**partial**은 레이아웃과 유사하게 작동한다. 요소들은 다른 뷰에 포함될 수 있는 뷰 콘텐츠 스니펫이다.

EJS 섹션 내에서 **include 키워드**를 사용해 뷰에 대한 상대경로를 사용하자. 레이아웃이 이미 views 폴더에 있기 때문에 내비게이션 요소를 찾기 위해 동일한 디렉터리 수준의 **partials 폴더**를 바라볼 필요가 있다.

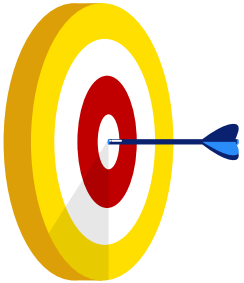
[노트] 뷰가 변경된 경우라면 애플리케이션을 재시작할 필요는 없다.



퀵 체크 10.1

템플릿 엔진이란 무엇인가?

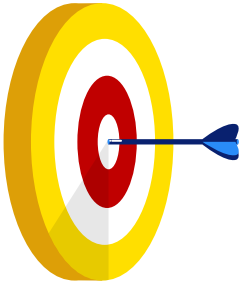
템플릿 엔진은 Express.js가 애플리케이션에서 템플릿 뷰를 처리하는 데 사용하는 도구다. 템플릿 뷰에는 HTML과 JavaScript 콘텐츠가 혼합돼 있기 때문에 브라우저가 이 정보를 브라우저에서 사용할 수 있는 HTML 파일로 변환해야 한다.



퀵 체크 10.2

어떤 형식으로 데이터를 컨트롤러에서
뷰로 보내는가?

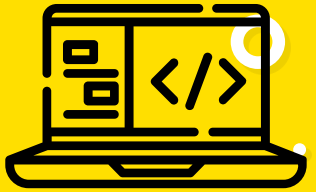
컨트롤러에서 데이터를 보내려면 JavaScript 객체 내에서
변수를
전달하면 된다 컨트롤러의 컨텍스트에 로컬인 변수는 키의
이름을 따르며, 이름은 뷰의 변수 이름과 일치해야 한다



퀵 체크 10.3

여러 뷰에서 요소들을 공유하기 위해
사용하는 키워드는 무엇인가?

include 키워드는 제공된 상대경로에서
요소들을 찾아 렌더링한다.



Coding!

Lesson 10

p. 155-164

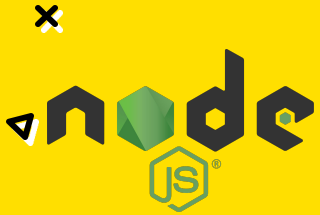
11

설정과 에러 처리

애플리케이션 시작 스크립트 변경
Express.js를 통한 정적 페이지 서비스
에러 처리를 위한 미들웨어 생성

p. 165-171

11.1 시작 스크립트 수정



// 애플리케이션을 재시작할 필요 없을 것이다.

npm install nodemon --save-dev

```
6   "scripts": {
7     "test": "jest",
8     "start": "nodemon main.js"
9   },
```

~~node main.js~~

npm start

11.2 Express.js의 에러 처리

Controllers 폴더 에 errorController.js 파일을 만들자.

npm install http-status-codes

이 함수는 **일반 미들웨어 함수보다 하나 더 많은 변수를 갖고 있어**, 요청-응답 사이클에서 **에러가 발생하면 이 에러는 첫 번째 변수**에 나타난다. console.error로로깅할수 있고 이를통해 어디가 잘못됐는지 알 수 있다. 앞에서의 미들웨어 함수에서 next 매개변수는 체인 내에서의 다음 함수 또는 라우트를 호출하며, 좀 더 처리돼야 할 에러 객체가 전달된다.

```
6  /**
7   * Listing 11.2 (p. 168)
8   * 에러 컨트롤러 추가
9   */
10 exports.logErrors = (err, req, res, next) => { // 에러 처리를 위한 미들웨어 추가
11     console.error(err.stack); // 에러 스택 로깅
12     next(err); // 다음 미들웨어 함수로 에러 전달
13 };
```

[노트] 이 에러 처리에서 **4개의 매개변수가 필요로 하며 항상 error가 제일 먼저 온다.** 이 4개의 매개변수가 모두 없다면 next 객체가 존재하지 않을 것이며, 이 함수 내 객체가 다음 미들웨어 함수의 호출을 하게 할 필요는 없을 것이다.

11.2 Express.js의 에러 처리

다음으로 app.use(errorController.logErrors) 를 main.js 파일에 추가해 Express.js에게 이 미들웨어를 사용한다고 알려야 한다.

[주의] main.js에서 일반 라우트의 정의 다음에 미들웨어 라인을 추가하라. **main.js에서 순서가 중요하다.** 기본적으로는 Express.js는 요청이 **끝나는 시점**에서 에러들을 처리한다.

```
15  /**
16   * Listing 11.3 (p. 169)
17   * 사용자 정의 메시지로 빠진 라우트 및 에러 대응
18   */
19  exports.resNotFound = (req, res) => { // 404 상태 코드로 응답
20      let errorCode = httpStatus.NOT_FOUND;
21      res.status(errorCode);
22      res.send(`${errorCode} | The page does not exist!`);
23  };
24
25  exports.resInternalError = (err, req, res, next) => { // 500 상태 코드로 모든 에러 처리
26      let errorCode = httpStatus.INTERNAL_SERVER_ERROR;
27      console.error(`ERROR occurred: ${err.stack}`);
28      res.status(errorCode);
29      res.send(`${errorCode} | Sorry, our app is experiencing a problem!`);
30  }
```

애플리케이션이 처리 과정에서 에러가 발생했다면 404 오류(페이지 부재)나 500 오류 코드 처리를 위한 범용 라우트를 **마지막에 둬** 사용자 정의 메시지로 응답을 보낼 수도 있다.

11.2 Express.js의 에러 처리

에러 페이지를 변경하고 싶으면 기본 HTML이 들어 있는 public 폴더에 **404.html**과 **500.html** 파일을 추가할 수 있다. 그 후 일반 텍스트 메시지로 응답하는 대신 이 파일들로 응답할 수 있으며 응답 과정에서 템플릿 엔진을 사용하지 않을 것이다.

```
15  /**
16   * Listing 11.3 (p. 169)
17   * 사용자 정의 메시지로 빠진 라우트 및 에러 대응
18   */
19  exports.resNotFound = (req, res) => { // 404 상태 코드로 응답
20      let errorCode = httpStatus.NOT_FOUND;
21      res.status(errorCode);
22      res.sendFile(`./public/${errorCode}.html`, { // 404.html 파일의 콘텐츠 전송
23          root: './'
24      });
25  };
```

이 코드에서 res.sendFile은 에러 페이지의 절대 경로를 특정하는 데 사용하며, 일반 템플릿 렌더링이 작동하지 않을때 유용하다.

```
23  // main.js
24  app.use(errorController.logErrors);
25  app.use(errorController.resNotFound); // main.js에 에러 처리 미들웨어 추가
26  app.use(errorController.resInternalError);
```

11.3 정적 파일의 제공

1부에서 시작된 애플리케이션에서 모든 다른 타입의 정적 파일과 에셋은 **수백 라인의 코드**를 요구하고 있다. **Express .js를 사용하면 이 파일 타입은 자동으로 설정된다.** 필요한 작업은 **Express.js에 정적 파일의 위치를 알려주는 것이다.**

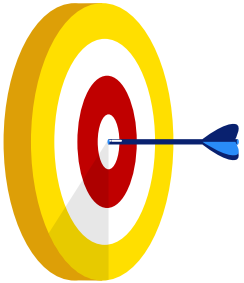
```
15   app.use(express.static('public'));
```

이 코드라인은 정적 파일 제공을 위해 애플리케이션에게 main.js와 같은 레벨에 위치하는 public 폴더에 접근하도록 한다.

이 코드를 삽입하고, http://localhost/404.html로 바로 접속할 수 있다.

URL의 메인 도메인 뒤에 파일명을 붙여 public 폴더에 있는 이미지나 다른 정적 에셋에 접근할 수 있고 다른 파일들을 추가할 수도 있다. 만일 cat.jpg같이 다른 하위 디렉터리 (images)에 있는 이미지를 호출한다면 이미지만 단독으로 http://localhost/images/ cat.jpg로 호출할 수 있다.

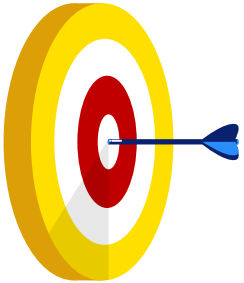
[노트] 정적 파일은 404.html과 500.html과 같은 사용자 정의 에러 페이지와 에셋을 포함한다 **이 HTML 페이지는 EJS 값들을 갖고 있기 않기 때문에 템플릿 엔진을 거치지 않는다.**



퀵 체크 11.1

package.json 파일에서 scripts의
역할은 무엇일까?

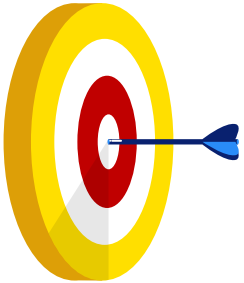
scripts 객체로 npm으로 실행하려는 명령의
별정 (aliases)을 정의할 수 있다.



퀵 체크 11.2

왜 없는 라우트의 처리를 위한 미들웨어는
일반 애플리케이션 라우트의 뒤에 올까?

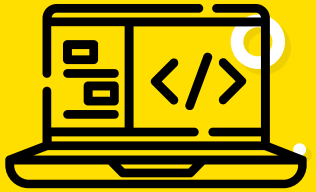
정직 파일은 404.html과 500.html과 같은 사용자 정의 에러 페이지와 에셋을 포함한다 이 HTML 페이지는 EJS 값들을 갖고 있기 않기 때문에 템플릿 엔진을 거치지 않는다.



퀵 체크 11.3

Public 폴더에서 중요한 정적 파일에는 무엇이 있는가?

Public 폴더에는 에러 페이지를 위한 정적 HTML 파일이 들어 있다, 애플러케이션에서 뭔가 에러가 발생하면 이 파일들이 사용자에게 보이게 된다.



Coding!

Lesson 10-11 연속

p. 165-171

과제 타임!

한번 해 보자!~