

Creating and setting up a MEAN project

Samkeun Kim <skim@hknu.ac.kr>

<http://cyber.hankyong.ac.kr>

This chapter covers

- Managing dependencies by using a package.json file
- Creating and configuring Express projects
- Setting up an MVC environment
- Adding Twitter Bootstrap for layout
- Publishing to a live URL and using Git and Heroku

애플리케이션인 Loc8r 구축

사용자 가까이 있는 장소 리스트를 표시하고,

각 장소에 대해 세부 정보를 제공하고,

방문해 본 사람들이 로그인하여 리뷰를 남길 수 있도록 하는 위치 인식 웹 애플리케이션

그림 3.1은 애플리케이션 아키텍처 구축 측면에서 이 장이 초점을 맞추는 부분을 설명한다.

1. 프로젝트를 만들고, DB를 제외한 다른 모든 것을 저장할 Express 애플리케이션을 **캡슐화**
2. 기본 Express 애플리케이션 설정

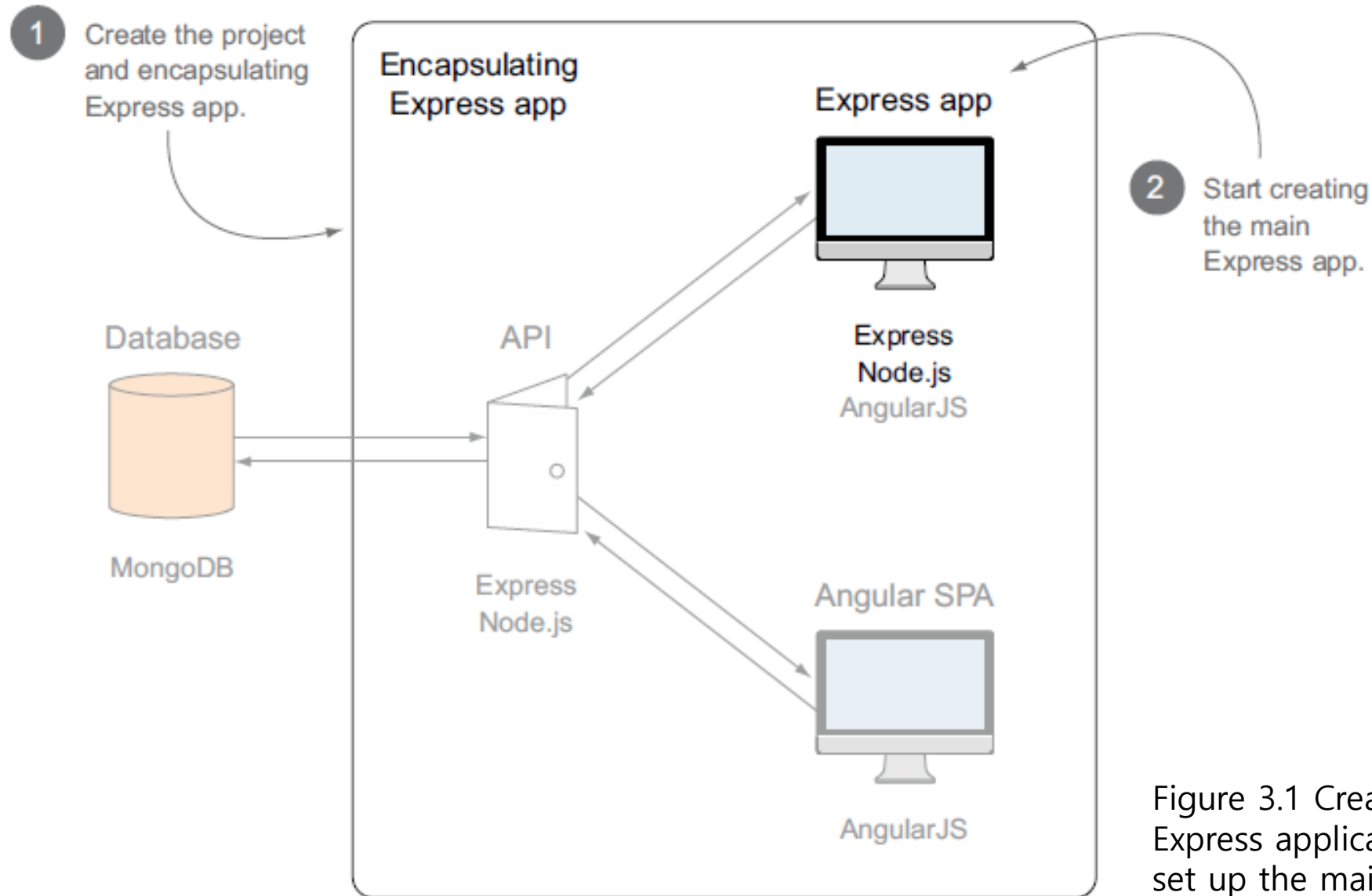


Figure 3.1 Creating the encapsulating Express application, and starting to set up the main Express application

Express 프로젝트 설정

커맨드 라인에서 새로운 **Express** 프로젝트를 생성하는 방법, 이 시점에서 지정할 수 있는 여러 가지 옵션들을 살펴본다.

Express 프로젝트를 model-view-controller(MVC) 아키텍처 형태로 수정할 수 있다.

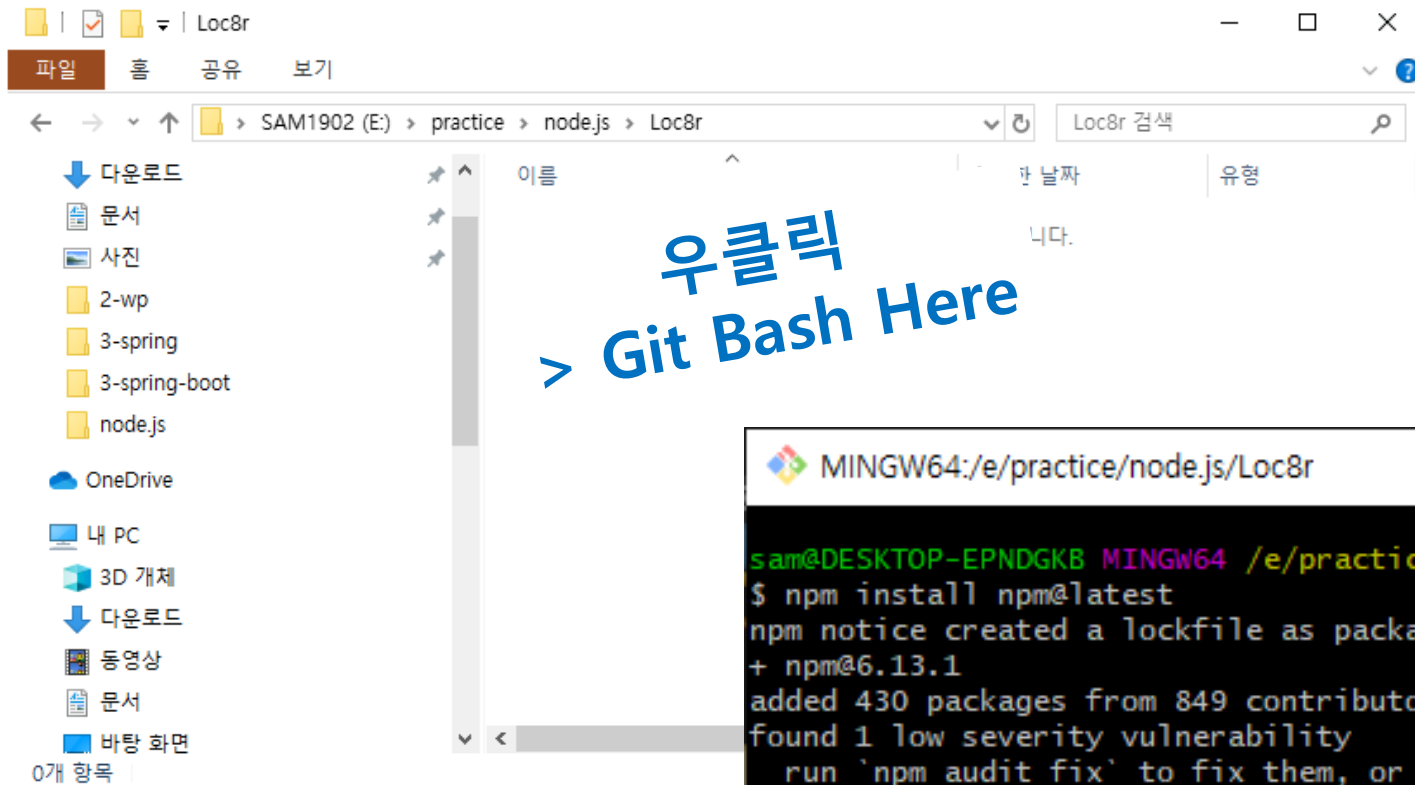
또한 **Twitter Bootstrap** 프레임워크를 포함시킬 수 있다.

⇒ Pug 템플릿을 수정하여 사이트를 반응형(responsive)으로 만들 수 있다.

이 장의 마지막 단계에서는 수정된 반응형 MVC Express 애플리케이션을 **Heroku**와 **Git**을 사용하여 실제 URL로 푸시한다.

윈도 검색 > git 입력 > **Git Bash** 클릭!

또는 작업 디렉토리에 우클릭!



```
MINGW64:/e/practice/node.js/Loc8r

sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ npm install npm@latest
npm notice created a lockfile as package-lock.json. You should commit this file.
+ npm@6.13.1
added 430 packages from 849 contributors and audited 12001 packages in 560.173s
found 1 low severity vulnerability
  run 'npm audit fix' to fix them, or 'npm audit' for details

sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ |
```

시간 오래 걸림

Installing Node dependencies with npm

```
MINGW64:/e/practice/node.js/Loc8r
sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (loc8r)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to E:\practice\node.js\Loc8r\package.json:
{
  "name": "loc8r",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this OK? (yes)
sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ |
```

생성된 package.json 파일에 아래 내용 Copy & Paste!

```
{
  "name": "loc8r",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "latest",
    "body-parser": "latest",
    "cookie-parser": "latest",
    "morgan": "latest",
    "serve-favicon": "latest",
    "debug": "latest",

    "pug": "latest"
  }
}
```

Copy & Paste!

```
sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ npm install
```

MINGW64:/e/practice/node.js/Loc8r

sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r

\$ npm install

npm WARN deprecated core-js@2.6.10: core-js@<3.0 is no longer maintained and not recommended for usage due to the number of issues. Please, upgrade your dependencies to the actual version of core-js@3.

> core-js@2.6.10 postinstall E:\practice\node.js\Loc8r\node_modules\core-js

> node postinstall || echo "ignore"

Thank you for using core-js (<https://github.com/zloirock/core-js>) for polyfilling JavaScript standard library!

The project needs your help! Please consider supporting of core-js on Open Collective or Patreon:

> <https://opencollective.com/core-js>

> <https://www.patreon.com/zloirock>

Also, the author of core-js (<https://github.com/zloirock>) is looking for a good job -)

added 128 packages from 173 contributors and audited 12283 packages in 345.618s

found 1 low severity vulnerability

run 'npm audit fix' to fix them, or 'npm audit' for details

sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r

\$ |

Creating an Express project

MEAN 애플리케이션을 구축하기 위한 새로운 Express 프로젝트를 생성

Express 프로젝트를 작성하려면, 개발 시스템에 다섯 가지 주요 사항이 설치되어 있어야 한다:

- Node and npm
- The Express generator installed **globally**
- Git
- Heroku
- New command-line interface (CLI) or terminal

Configuring an Express installation

Express 프로젝트 => 커맨드라인에서 설치

```
$ npm install -g express-generator
```

```
MINGW64:/e/practice/node.js/Loc8r
sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ npm install -g express-generator
C:\Users\sam\AppData\Roaming\npm\express -> C:\Users\sam\AppData\Roaming\npm\node_modules\express-generator\bin\express-cli.js
+ express-generator@4.16.1
added 10 packages from 13 contributors in 1.311s

sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ |
```

```
$ express --view=pug
```

현재 폴더에 프레임 워크가 설치된다.
먼저 몇 가지 **설정 옵션**을 살펴보자.

```
MINGW64:/e/practice/node.js/Loc8r
sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ express --view=pug
destination is not empty, continue? [y/N] y

create : public\
create : public\javascripts\
create : public\images\
create : public\stylesheets\
create : public\stylesheets\style.css
create : routes\
create : routes\index.js
create : routes\users.js
create : views\
create : views\error.pug
create : views\index.pug
create : views\layout.pug
create : app.js
create : package.json
create : bin\
create : bin\www

install dependencies:
  $ npm install

run the app:
  $ DEBUG=loc8r:* npm start

sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ |
```

CONFIGURATION OPTIONS WHEN CREATING AN EXPRESS PROJECT

Express 프로젝트를 생성할 때 다음 사항을 지정할 수 있다:

- Which HTML template engine to use
- Which CSS preprocessor to use
- Whether to add support for sessions

디폴트 설치: **pug** template engine, **no CSS** preprocessing or **session support**

표 3.1과 같이 몇 가지 옵션을 지정할 수 있다.

Table 3.1 Command-line configuration options when creating a new Express project

Configuration command	Effect
<code>--css less stylus</code>	Adds a CSS preprocessor to your project, either Less or Stylus, depending on which you type in the command.
<code>--ejs</code>	Changes the HTML template engine from Jade to EJS.
<code>--jshtml</code>	Changes the HTML template engine from Jade to JsHtml.
<code>--hogan</code>	Changes the HTML template engine from Jade to Hogan.

예를 들어, **less CSS** 전처리기와 **Hogan** 템플릿 엔진을 사용하는 프로젝트를 만들려면 터미널에서 다음 명령을 실행하면 된다:

```
$ express --css less --hogan
```

프로젝트를 단순하게 유지하기 위해 CSS 사전 처리를 사용하지 않으므로 기본 CSS를 그대로 사용할 수 있다.

먼저 **템플릿 엔진 옵션**에 대해 간략하게 살펴보자.

DIFFERENT TEMPLATE ENGINES

4개의 템플릿 옵션이 있다: Pug, EJS, JsHtml, Hogan.

- ✓ 템플릿 엔진의 기본 워크플로는 **HTML 템플릿을 생성한 후, 데이터를 전달하는 것이다.**
- ✓ 엔진은 둘(data + view)을 합하여 브라우저가 받게 될 최종 HTML 마크업을 생성하게 된다.
- ✓ 모든 엔진들은 자신만의 장단점을 갖는다.

이 책에서 우리는 **pug**를 사용하기로 한다.

- ✓ **Pug**는 매우 강력하며 우리가 필요로 하는 모든 기능을 제공해준다.
- ✓ **Pug**는 Express의 **디폴트 템플릿 엔진**이다.

A QUICK LOOK AT PUG

Pug는 템플릿에 실제로 **HTML 태그들을 포함하지 않는다**는 점에서 다른 템플릿과는 대조적이다.

- ✓ HTML 구조를 정의하기 위해 태그 이름, 들여쓰기, CSS 스타일 레퍼런스 메소드를 이용한다.
- ✓ 한 가지 예외는 **<div>** 태그이다.
- ✓ 이 태그는 너무나 흔히 사용되기 때문에 템플릿에서 태그 이름이 생략되어 있다면 **<div>** 태그라고 간주해도 된다.

Pug 템플릿과 컴파일된 결과:

```
#banner.page-header
  h1 My page
  p.lead Welcome to my page
```

**Pug template contains
no HTML tags**

```
<div id="banner" class="page-header">
  <h1>My page</h1>
  <p class="lead">Welcome to my page</p>
</div>
```

**Compiled output is
recognizable HTML**

첫 번째 줄에서 아래와 같은 차이점을 볼 수 있다:

- 태그 이름을 지정하지 않았기 때문에 <div>가 만들어진다
- Pug의 #banner는 HTML DOM에서 id="banner"가 된다
- Pug의 .page-header는 HTML의 class="page-header"가 된다

이제 위의 지식을 시작 포인트로 하여 프로젝트를 생성해 보자.

Creating an Express project and trying it out

새로운 프로젝트를 만들어 보자.

터미널에서 이 폴더로 이동하여 아래 명령을 실행한다:

```
~Loc8r> express ← 이미 수행함!
```

이렇게 하면 Loc8r 폴더 안에 폴더와 파일이 만들어져 **Loc8r** 애플리케이션의 기초가 만들어진다.

다음에는 의존 모듈들을 설치해야 한다.

package.json 파일과 같은 폴더에 있는 터미널 프롬프트에서 아래 명령 실행:

```
~Loc8r> npm install ← 이미 수행함!
```

완료했다면, 이제 애플리케이션을 테스트할 준비가 된 것이다.

TRYING IT OUT

지금까지 한 일이 어떻게 작동하는지 살펴 보자.

터미널의 Loc8r 폴더에서 아래 명령을 실행한다:

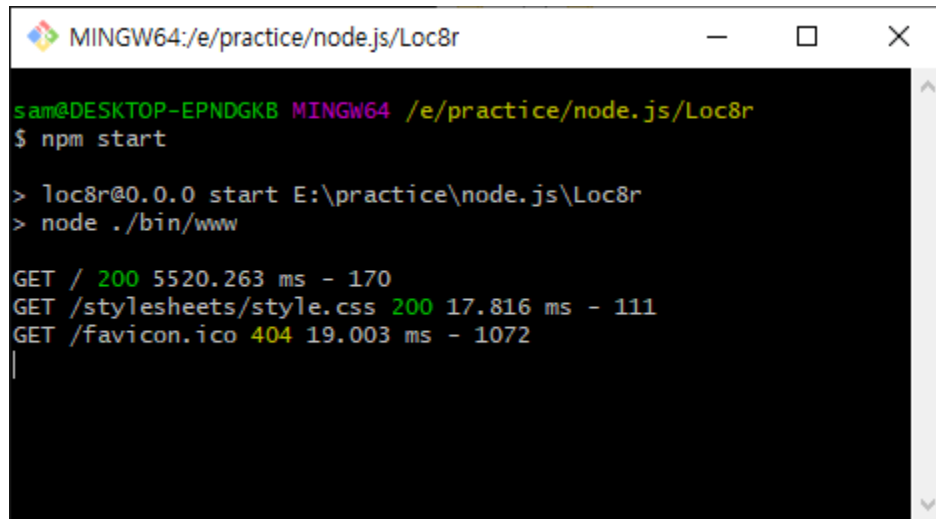
```
~Loc8r> npm start
```

```
Loc8r@0.0.0 start /path/to/your/application/폴더
```

이제 **Express** 애플리케이션이 실행 중이다! 브라우저를 열고 **localhost:3000**으로 이동하여 실제 동작을 볼 수 있다.

실습과제 23-1 My First Express application

\$npm start



```
MINGW64:/e/practice/node.js/Loc8r
sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ npm start

> loc8r@0.0.0 start E:\practice\node.js\Loc8r
> node ./bin/www

GET / 200 5520.263 ms - 170
GET /stylesheets/style.css 200 17.816 ms - 111
GET /favicon.ico 404 19.003 ms - 1072
```

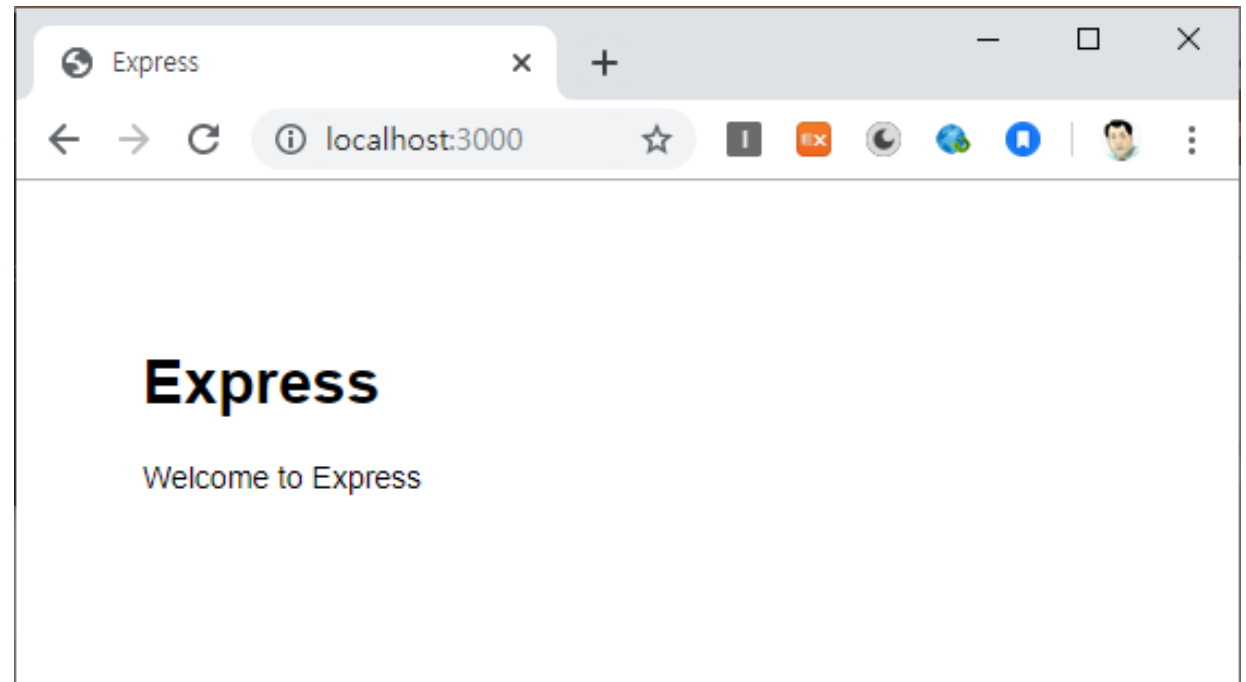


Figure 3.3 Landing page for a barebones Express project

HOW EXPRESS HANDLES THE REQUESTS

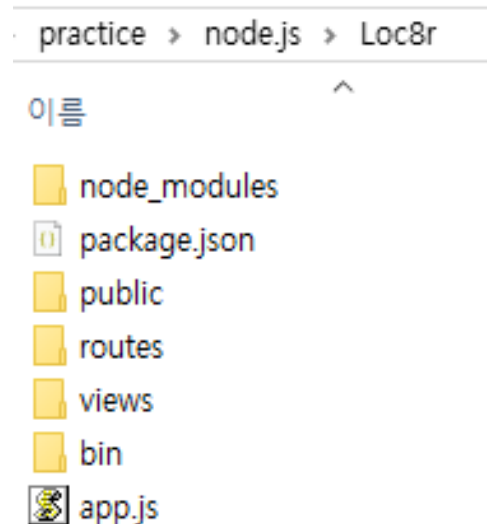
Express 서버에 전달되는 모든 요청은 app.js 파일에 정의된 미들웨어를 통하여 실행된다.

app.js 미들웨어: static 파일들에 대한 path를 찾는다.

미들웨어가 파일에 대한 패스와 매치되면, Express는 이것을 비동기적으로 반환한다.

⇒ Node.js 프로세스: Non-blocking operation

⇒ 미들웨어의 실행이 완료되면 라우트에 대한 요청 패스를 매칭시키려고 시도한다



About Express middleware

app.js 파일의 중앙 부분에 **app.use**로 시작하는 한 무더기의 라인들이 있다. 이들을 미들웨어라고 한다.

요청이 애플리케이션에 들어오면 **차례대로** 각 미들웨어를 통과하게 된다. 각 미들웨어는 요청일 수도 있고 아닐 수도 있지만, 응답을 반환하는 애플리케이션 로직에 도달할 때까지 항상 **next** 미들웨어로 넘겨진다.

예를 들어,

```
app.use(express.cookieParser());
```

는 들어오는 요청을 가져와서 쿠키 정보를 분석하여 데이터를 **request**에 첨부해 준다.

```
// view engine setup
app.set('views', path.join(__dirname, 'app_server', 'views'));
app.set('view engine', 'pug');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
// app.use('/users', users);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handlers

// development error handler
// will print stacktrace
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});
```

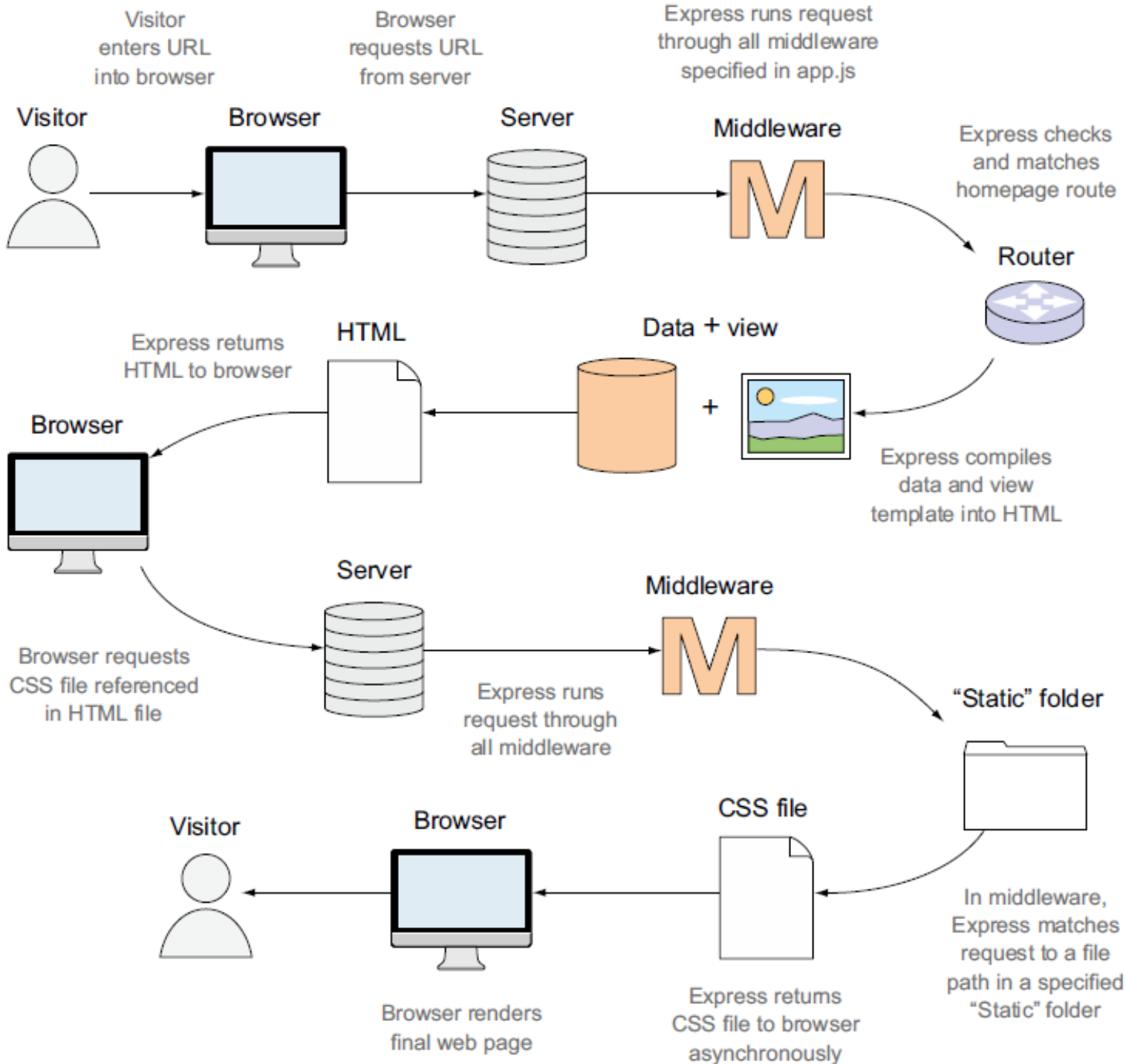


Figure 3.4. The key interactions and processes that Express goes through when responding to the request for the default landing page. The HTML page is processed by Node to compile data and a view template, and the CSS file is served asynchronously from a static folder.

Restarting the application

Node 애플리케이션은 실행되기 전에 컴파일된다.

따라서 실행 중일 때 애플리케이션 코드를 변경한다면, Node 프로세스가 재시작될 때까지 그 변경사항은 적용되지 않는다.

이것은 오로지 애플리케이션 코드에 대해서만 해당된다.

Pug 템플릿, **CSS** 파일, 클라이언트 측 **JavaScript**는 즉시 갱신될 수 있다.

Node 프로세스를 재시작 시키려면 2단계 과정을 거쳐야 한다:

첫째, 실행 프로세스를 정지시켜야 한다. 터미널에서 Ctrl-C를 누른다.

둘째, 프로세스를 터미널에서 다시 시작시킨다:

```
$ npm start
```

이렇게 해도 되지만 애플리케이션을 빈번하게 갱신하면서 테스트할 때마다 이 두 단계를 수행해야 한다는 사실은 실제로 상당히 부담스러워진다.

다행히도 더 좋은 방법이 있다.

AUTOMATICALLY RESTARTING THE APPLICATION WITH NODEMON

애플리케이션 코드를 모니터링 하다가 내용이 변경되면 프로세스를 즉시 재시작 시키는 서비스

⇒ 그 중 한 가지 서비스가 **nodemon**이다.

Nodemon을 사용하려면 Express와 마찬가지로 전역으로 설치하자:

터미널에서 **npm**을 사용하여 아래 명령을 수행하자:

```
$ npm install -g nodemon
```

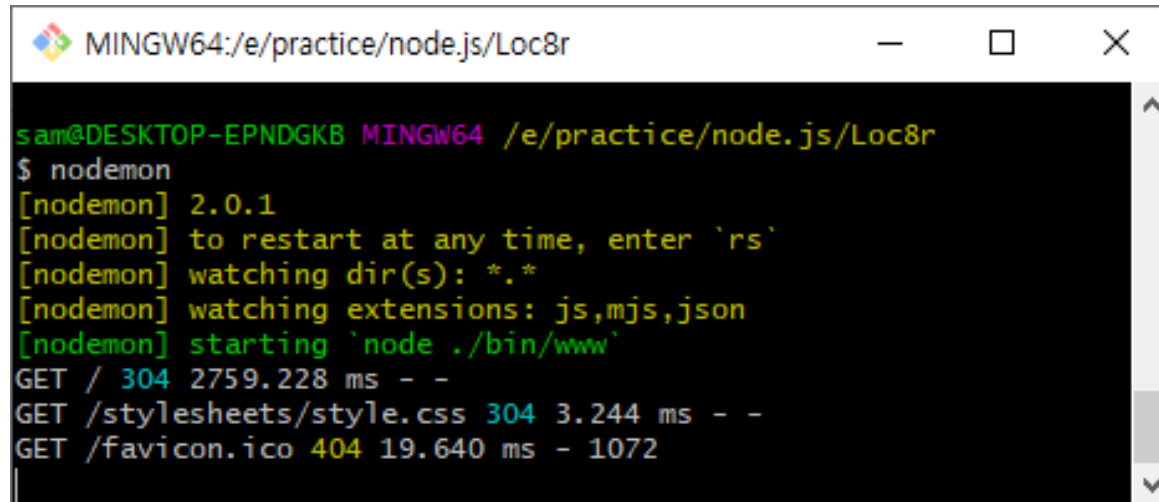
설치가 끝나면 어디에서든 원하는 대로 **nodemon**을 사용할 수 있다.

Nodemon을 사용하는 방법은 정말 간단하다.

애플리케이션을 시작시키기 위해 **node**라고 입력하는 대신 **nodemon**을 입력하면 된다.

따라서 현재 위치가 터미널의 Loc8r 폴더에 있는지 확인하고 실행중인 노드 프로세스를 중지한 상태라면 아래 명령을 입력하자:

\$ **nodemon**



```
MINGW64:/e/practice/node.js/Loc8r
sam@DESKTOP-EPNDGKB MINGW64 /e/practice/node.js/Loc8r
$ nodemon
[nodemon] 2.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./bin/www`
GET / 304 2759.228 ms - -
GET /stylesheets/style.css 304 3.244 ms - -
GET /favicon.ico 404 19.640 ms - 1072
```

Nodemon이 실행 중이고, 노드 ./bin/www가 시작되었음을 확인하는 터미널에 몇 줄의 행이 출력되는 것을 볼 수 있다.

브라우저로 돌아가서 새로 고침하면 애플리케이션이 여전히 계속 남아 있을 것이다.

Modifying Express for MVC

MVC 아키텍처로 수정하자.

MVC는 **Model-View-Controller**의 약자로, 데이터(모델), 디스플레이(뷰), 애플리케이션 로직(컨트롤러)으로 분리하는 것을 목표로 하는 아키텍처이다.

분리 목적: 컴포넌트간의 강한 결합(tight coupling)을 제거하기 위해

⇒ 이론상 관리하기 편하고 재사용이 수월하다

A bird's eye view of MVC

사용자가 구축하는 대부분의 애플리케이션은 Incoming 요청을 받아 들여 무언가를 수행하고 응답을 반환한다.

MVC 아키텍처 루프:

- 1— A **request** comes into the application.
- 2— The **request** gets routed to a controller.
- 3— The **controller**, if necessary, makes a request to the model.
- 4— The **model** responds to the controller.
- 5— The **controller** sends a response to a view.
- 6— The **view** sends a response to the original requester.

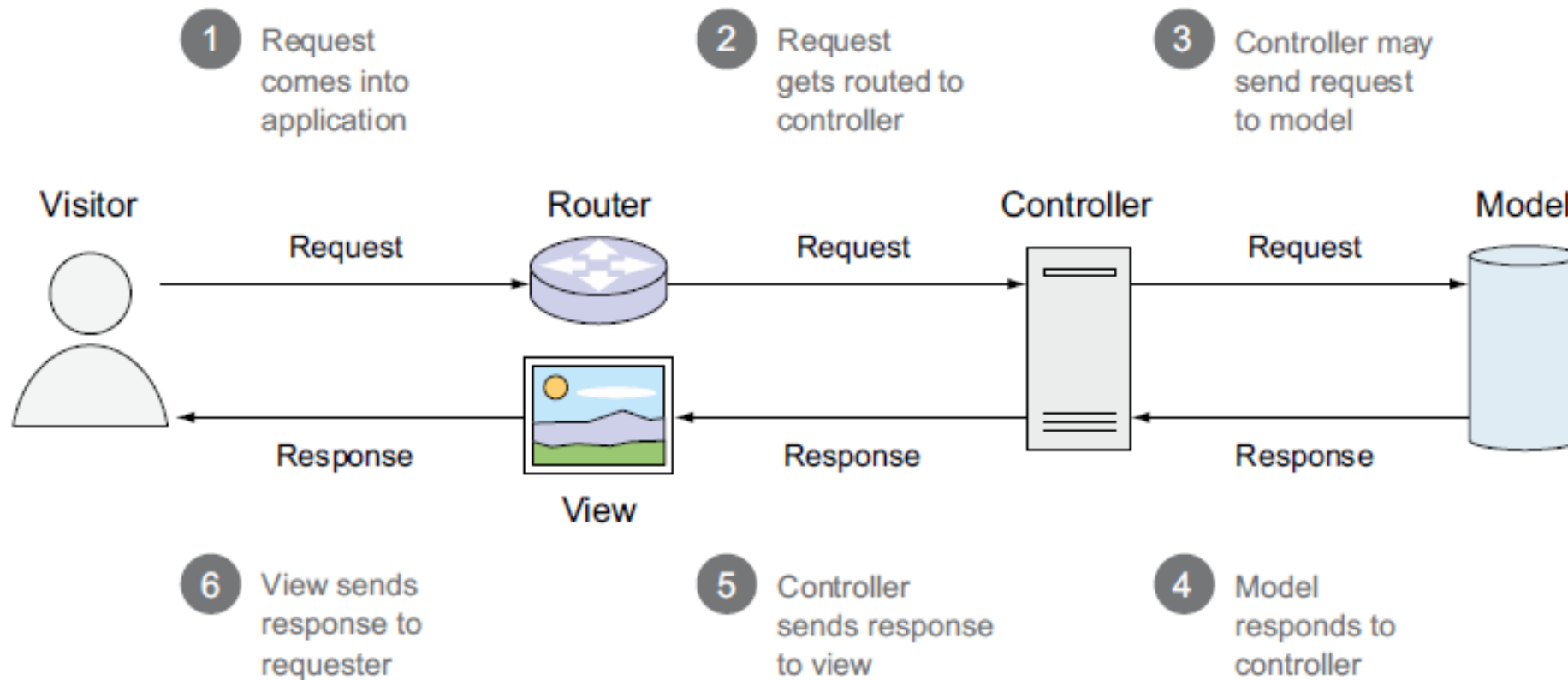


Figure 3.5
Request-response
flow of a basic
MVC architecture

그림 3.5는 MVC 아키텍처의 각 부분과 이들이 어떻게 링크되어 있는지를 보여준다.

또한 모델, 뷰, 컨트롤러 컴포넌트와 함께 라우팅 메커니즘의 필요성을 보여준다.

이제 **Loc8r** 애플리케이션의 기본 흐름이 어떻게 동작되는지 살펴 보았으므로 Express 설치를 수정해보자.

Changing the folder structure

Loc8r 폴더에서 새로 만들어진 Express 프로젝트를 보면, `views` 폴더와 `routes` 폴더를 포함하는 파일 구조가 보이지만, `models`나 `controllers`는 없다.

MVC 아키텍처로 폴더 정리 - 아래 세 가지 단계를 따라해보자:

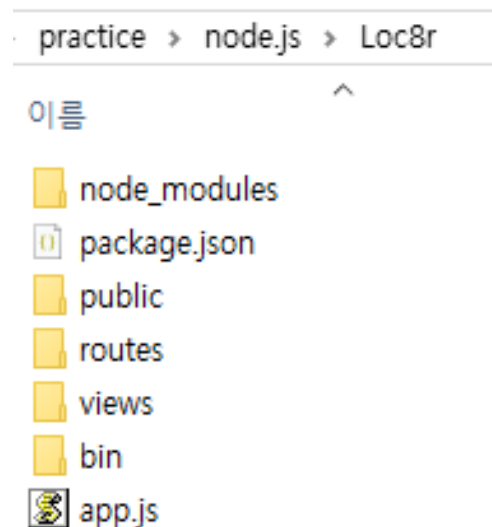
1— `app_server`라는 새 폴더를 만든다

2— `app_server`에서 `models`와 `controllers`라는 2개의 새 폴더를 만든다

3— 애플리케이션의 루트의 `views`와 `routes` 폴더를 `app_server` 폴더로 이동

Figure 3.6은 위의 변경사항을 보여준다(수정 전후의 폴더구조).

이제 애플리케이션에 **MVC** 셋업을 시켰다 => 관심사항들을 분리시켰다!



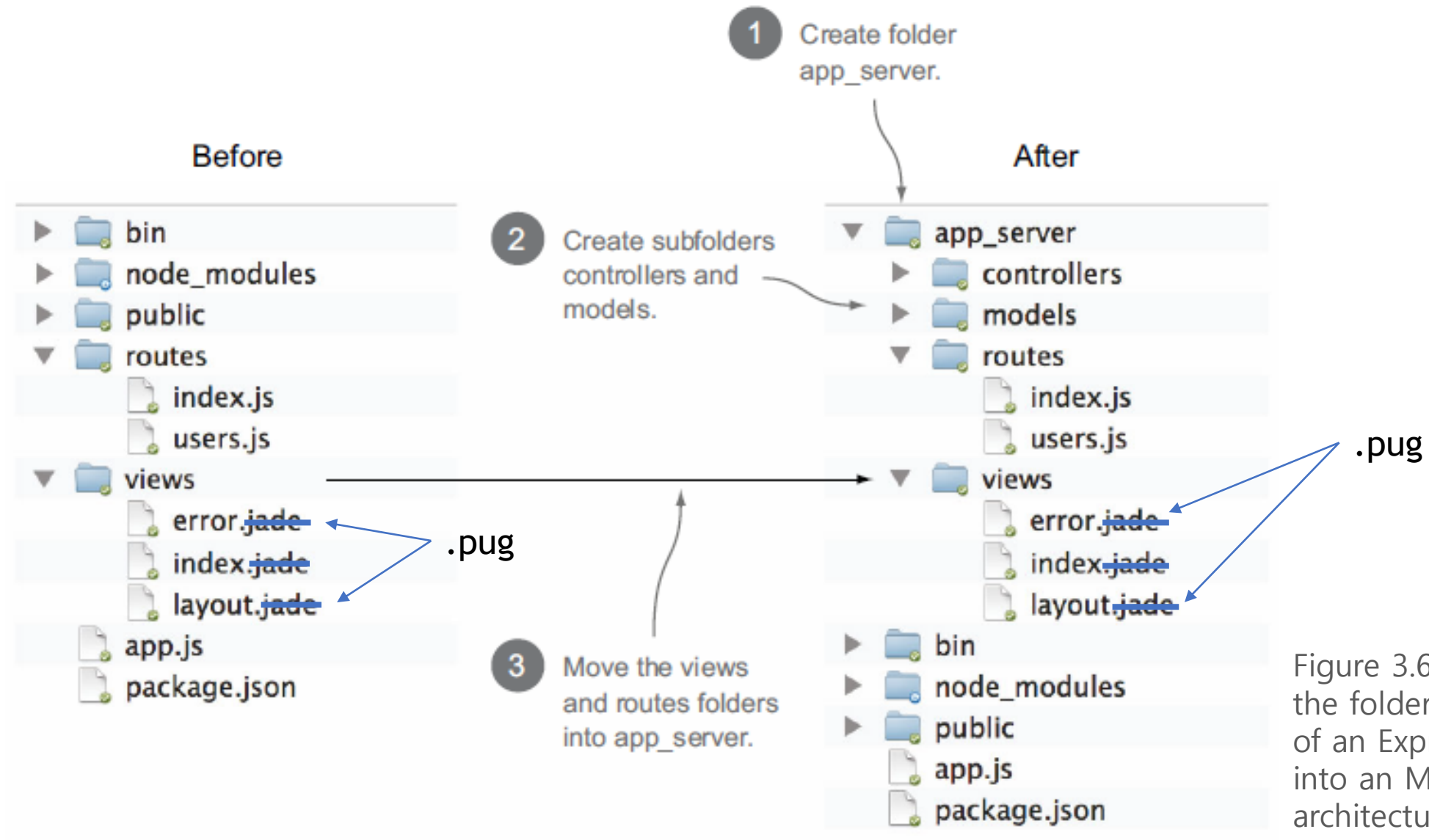


Figure 3.6 Changing the folder structure of an Express project into an MVC architecture

Using the new views and routes folders

첫 번째 할 일: Express에게 `views`와 `routes` 폴더를 이동시켰다는 사실을 알리는 것

USING THE NEW VIEWS FOLDER LOCATION

Express가 `/views`를 찾겠지만, 방금 `/app_server/views`로 이동되었다.

app.js에서 아래 행을 찾는다:

```
app.set('views', path.join(__dirname, 'views'));
```

아래와 같이 변경한다:

```
app.set('views', path.join(__dirname, 'app_server', 'views'));
```

routes를 이동했기 때문에 애플리케이션은 아직 동작하지 않는다.

그래서 이것도 Express에게 알려주도록 하자.

USING THE NEW ROUTES FOLDER LOCATION

마찬가지로 Express가 `/routes`를 찾겠지만, 방금 `/app_server/routes`로 이동했다.

app.js에서 아래 행을 찾자:

```
var indexRouter = require('./routes/index');  
var usersRouter = require('./routes/users');
```

아래와 같이 변경한다:

```
var indexRouter = require('./app_server/routes/index');  
var usersRouter = require('./app_server/routes/users');
```

이것을 저장하고 애플리케이션을 다시 실행하면 애플리케이션이 동작될 것이다!

```
($ nodemon)
```

Splitting controllers from routes

Controllers와 routes는 분리시키는 것이 좋다.

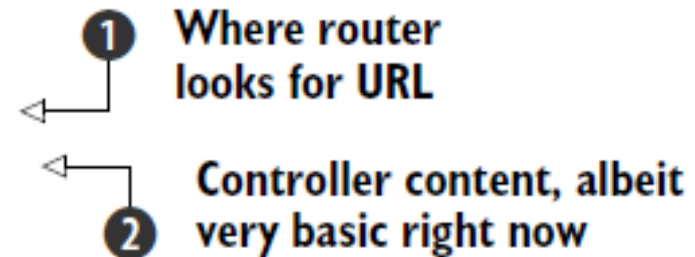
컨트롤러는 애플리케이션 로직을 관리하고, 라우팅은 URL 요청을 컨트롤러에 매핑시켜 준다.

UNDERSTANDING ROUTE DEFINITION

라우팅을 이해하기 위해 디폴트 Express homepage를 전달하는 라우트 설정을 살펴보자:

`index.js` in `app_server/routes` :

```
/* GET home page. */  
router.get('/', function(req, res) {  
  res.render('index', { title: 'Express' });  
});
```



컨트롤러 코드를 `controllers` 폴더에 두기 전에 먼저 같은 파일 내에서 테스트부터 해보자.

이를 위해 라우트 정의에서 anonymous 함수를 이름있는(named) 함수로 정의한다.

⇒ 라우트 정의에서 콜백으로서 함수 이름을 전달한다.

Inside `app_server/routes/index.js`:

Listing 3.2 Taking the controller code out of the route: step 1

```
var homepageController = function (req, res) {  
  res.render('index', { title: 'Express' });  
};
```

Take anonymous
function and define it
as a named function

```
/* GET home page. */  
router.get('/', homepageController);
```

Pass name of function
through as a callback
in route definition

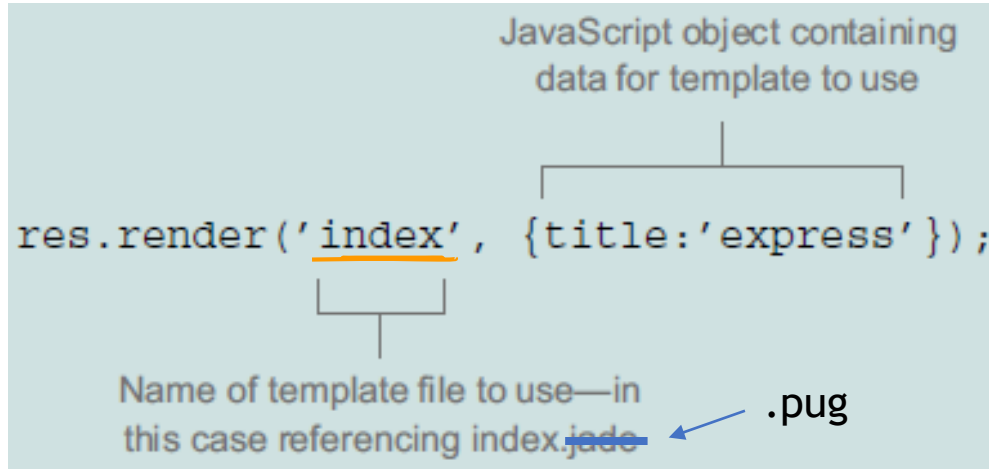
홈페이지를 새로 고침하면 이전과 마찬가지로 계속 작동할 것이다.

사이트 동작은 아무 것도 바뀐 것이 없으며, 관심사를 분리한 것 뿐이다.

Understanding res.render

Render는 view template을 HTML 응답으로 보내기 위해 컴파일하기 위한 Express 함수이다.

- 파라미터로 view template의 이름과 JavaScript 데이터 객체를 갖는다.



- 템플릿 파일은 확장자를 갖지 않는다. (`index.pug` can just be referenced as `index`.)
- You also don't need to specify the path to the view folder because you've already done this in the main Express setup.

Creating and using Node modules

module.exports 메소드를 사용하여 원하는 코드를 노출시킬 수 있다(**yourModule.js**) :

```
module.exports = function () {  
    console.log("This is exposed to the requester");  
};
```

main 파일에서 require:

```
require(' ./yourModule');
```

독립된 이름있는 메소드로 모듈을 노출시키고 싶다면:

```
module.exports.logThis = function(message) {  
    console.log(message);  
};
```

오리지널 파일에서 이 메소드를 참조하고 싶다면 모듈을 변수명에 할당하고 메소드를 호출하면 된다:

```
var yourModule = require(' ./yourModule');  
yourModule.logThis("Hooray, it works!");
```

MOVING THE CONTROLLER OUT OF THE ROUTES FILE

Node에서는 외부파일의 코드를 참조하기 위해 새로운 파일 안에 module을 생성하고, 원래의 파일 안에서 require 하면 된다.

첫 번째 할 일: 컨트롤러 코드를 저장할 파일을 생성한다.

- ✓ app_server/controllers에서 **main.js**라는 새 파일을 만든다.
- ✓ **index 메소드 생성**: 이것을 이용하여 res.render 코드를 아래 리스팅처럼 저장한다.

Listing 3.3 Setting up the homepage controller in app_server/controllers/main.js

```
/* GET home page */
module.exports.index = function(req, res) {
  res.render('index', { title: 'Express' });
};
```

← Create an index
export method

← Include controller
code for homepage

- ✓ 컨트롤러 내보내기를 만드는 것이 전부이다.

다음 단계: 라우트 정의에서 노출한 메소드를 사용할 수 있도록 라우트 파일에 컨트롤러 모듈을 `require`!!

`index.js`:

Listing 3.4 Updating the routes file to use external controllers

```
var express = require('express');
var router = express.Router();
var ctrlMain = require('../controllers/main');

/* GET home page. */
router.get('/', ctrlMain.index);

module.exports = router;
```

1 Require main controllers file

2 Reference index method of controllers in route definition

그림 3.7에서와 같이 `app.js`에서는 `routes/index.js`를 `require`하고, 라우트의 `index.js`에서는 `controllers/main.js`를 `require`한다.

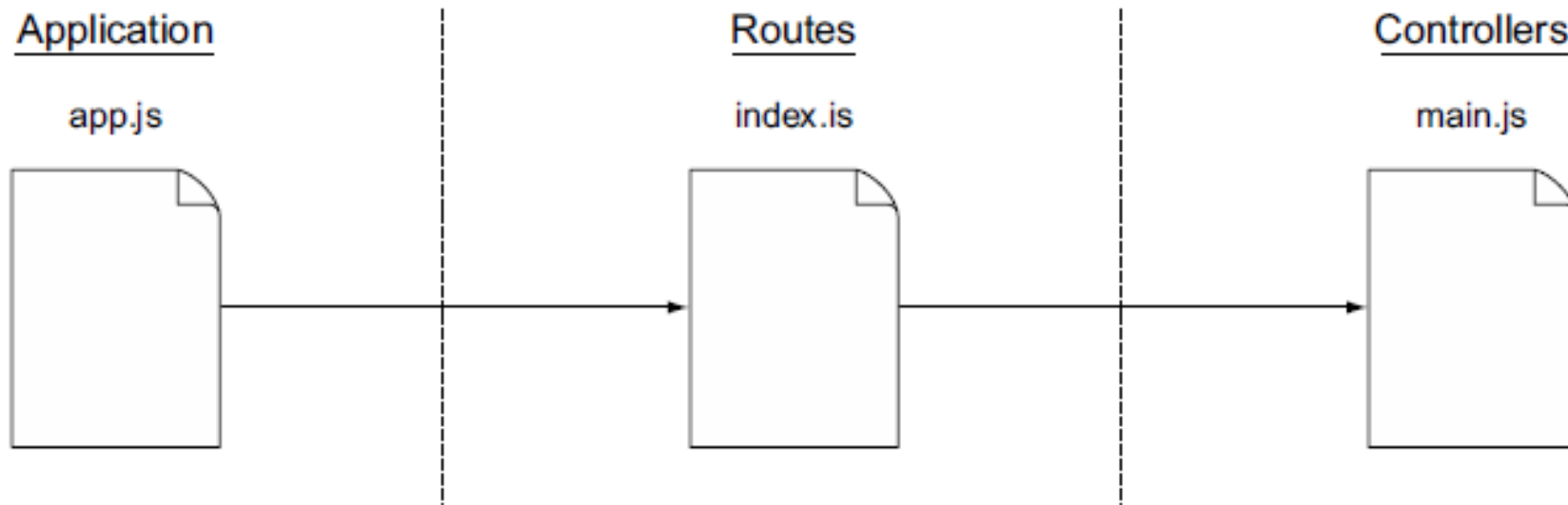
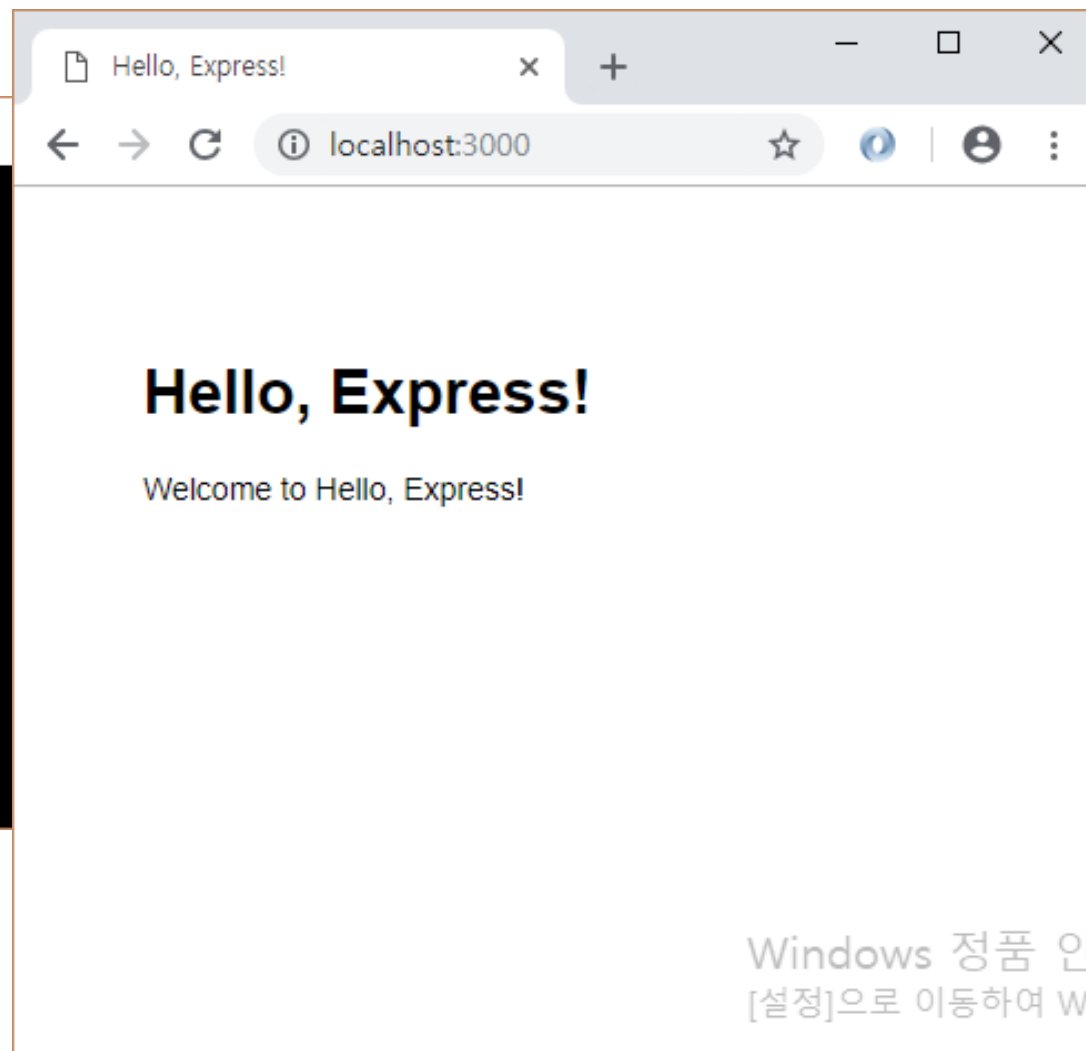


Figure 3.7 Separating the controller logic from the route definitions

실습과제 23-2

```
MINGW64:/f/practice/wsc/Loc8r
[nodemon] restarting due to changes...
[nodemon] starting 'node ./bin/www'
[nodemon] restarting due to changes...
[nodemon] starting 'node ./bin/www'
GET / 304 1028.608 ms - -
GET /stylesheets/style.css 304 2.666 ms - -
[nodemon] restarting due to changes...
[nodemon] starting 'node ./bin/www'
[nodemon] restarting due to changes...
[nodemon] starting 'node ./bin/www'
GET / 200 308.412 ms - 194
GET /stylesheets/style.css 304 1.898 ms - -
[nodemon] restarting due to changes...
[nodemon] starting 'node ./bin/www'
[nodemon] restarting due to changes...
[nodemon] starting 'node ./bin/www'
[nodemon] restarting due to changes...
[nodemon] starting 'node ./bin/www'
GET / 304 297.606 ms - -
GET /stylesheets/style.css 304 1.721 ms - -
```



Windows 정품 인
[설정]으로 이동하여 Wi

이제 Express와 관련된 모든 것이 준비되었으므로 구축 프로세스를 시작할 시간이다.

하지만 그 전에 우리가 해야 할 일이 몇 가지 더 있는데, 그 중 첫 번째는 **Twitter Bootstrap**을 애플리케이션에 추가하는 것이다.

Import Bootstrap for quick, responsive layouts

Loc8r 애플리케이션은 트위터의 **Bootstrap** 프레임워크를 사용하여 반응형 디자인의 개발을 가속화한다.

또한 테마(theme)를 사용하여 애플리케이션을 돋보이게 만든다.

Download Bootstrap and add it to the application

Bootstrap 다운로드: <https://getbootstrap.com/>

키 포인트: Bootstrap 파일들이 모두 브라우저에 직접 보내어지는 **static** 파일이라는 점이다;
즉 Node 엔진에 의한 어떤 처리도 요구하지 않는다.

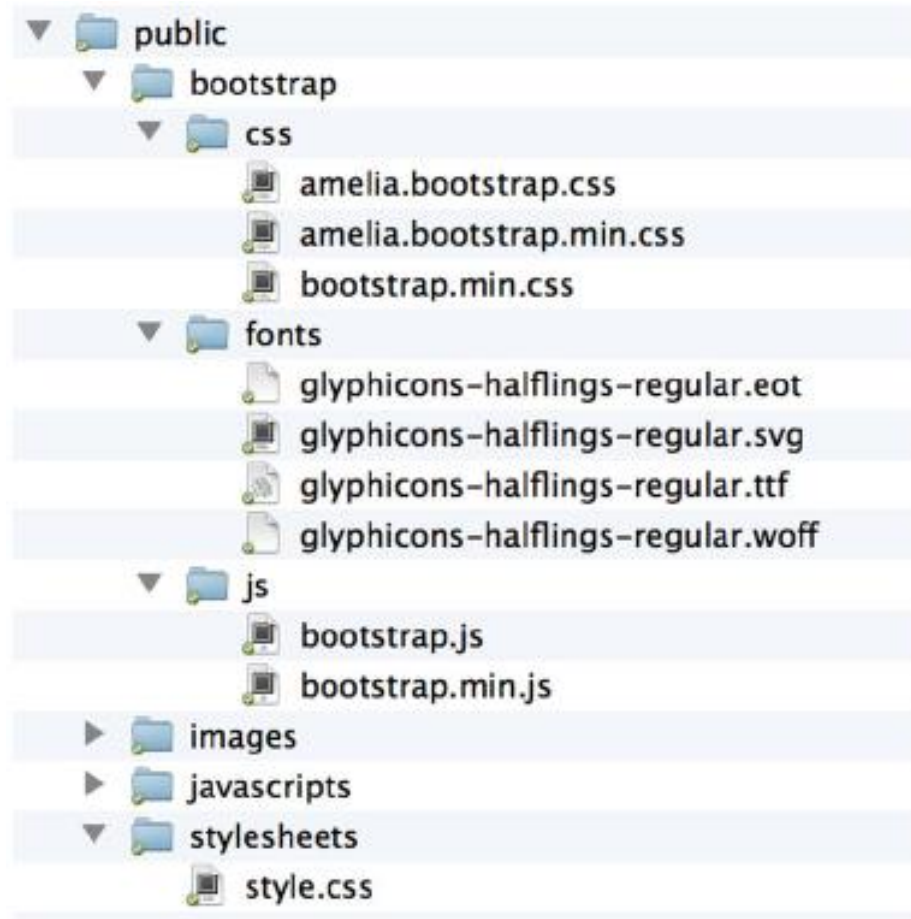
현재 Express 애플리케이션에 이런 용도로 사용할 수 있는 폴더가 이미 있다: **public**

- ✓ public 폴더는 그림 3.8과 같아야 한다.

Bootstrap은 일부 대화식 컴포넌트를 로드하기 위해 **jQuery**가 필요하다.

- ✓ CDN에서 직접 참조할 수 있지만 <http://jquery.com/download/> (or <https://code.jquery.com/jquery-3.3.1.min.js>)에서 다운로드하여 응용 프로그램에 추가
- ✓ jQuery 다운로드 => 애플리케이션의 `public/javascripts` 폴더에 저장

Using Bootstrap in the application



Amelia download:

<https://github.com/simonholmes/amelia>

Glyphicons download:

<https://github.com/Darkseal/bootstrap4-glyphicons>

(<https://github.com/Darkseal/bootstrap4-glyphicons/tree/master/bootstrap4-glyphicons/fonts/glyphicons>)

Figure 3.8 Structure of the public folder in the Express application after adding Bootstrap

WORKING WITH PUG TEMPLATES

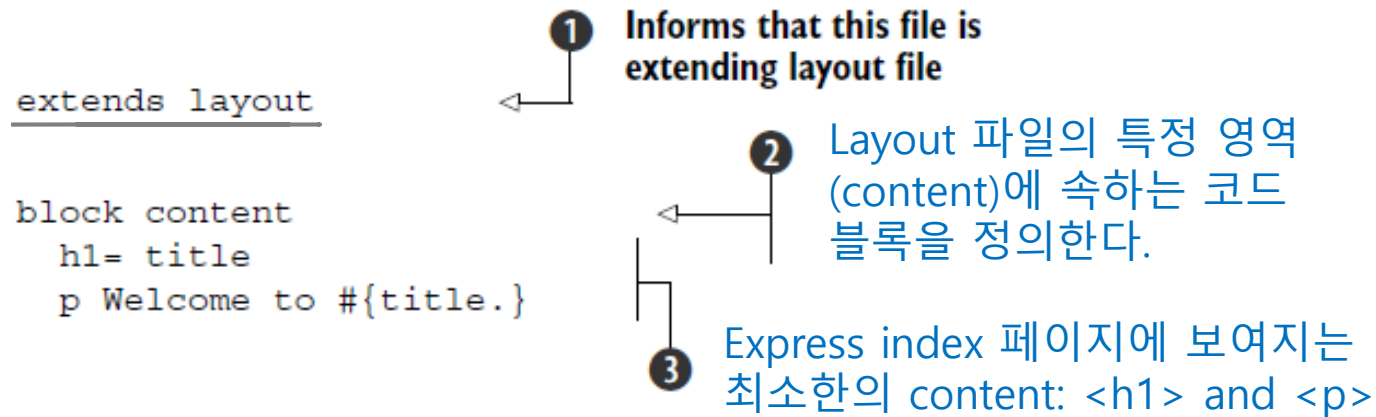
Pug 템플릿:

다른 Pug 파일을 확장할 수 있다(extends) => 웹 애플리케이션 구축 시 큰 의미

애플리케이션의 views 폴더를 보면 `layout.pug`와 `index.pug`라는 두 개의 파일이 있다.

`index.pug` 파일은 애플리케이션의 `index` 페이지 내용을 제어한다.

Listing 3.5 The complete index.pug file



여기에 `<head>` 또는 `<body>` 태그에 대한 참조나 스타일시트 참조가 없다.

⇒ Layout 파일에서 제어!

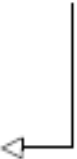
layout.pug

- Express 설치 시에 디폴트로 생성된 레이아웃 파일 (index 페이지를 위해 사용됨)

Listing 3.6 Default layout.pug file

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

Empty named
block can be used
by other templates



- block은 리스트 3.5의 index.pug 파일처럼 다른 Pug 템플릿에서 참조할 수 있다.
- 뷰 파일을 컴파일할 때 index 파일의 **block content**가 layout 파일의 **block content** 영역으로 푸시된다.

ADDING BOOTSTRAP TO THE ENTIRE APPLICATION

전체 애플리케이션에 외부파일을 추가할 때 `layout` 파일을 이용한다.

따라서 `layout.pug`에서 아래의 4가지를 수행해야 한다:

- Bootstrap CSS 파일 참조
- Bootstrap JavaScript 파일 참조
- jQuery 참조 - Bootstrap이 require한다
- 모바일 장치에서 페이지 크기가 잘 조정되도록 viewport 메타 데이터 추가

CSS 파일과 viewport 메타 데이터는 모두 문서의 머리 부분에 있어야 하며, 2개의 스크립트 파일은 body 섹션의 끝에 있어야 한다.

다음 리스트는 `layout.pug`에서 위의 모든 부분을 굵게 표시해준다:

Listing 3.7 Updated layout.jade Including Bootstrap references

```
doctype html
html
  head
    meta(name='viewport', content='width=device-width, initial-scale=1.0')
    title= title
    link(rel='stylesheet', href='/bootstrap/css/amelia.bootstrap.css')
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content

    script(src='/javascripts/jquery-1.11.1.min.js')
    script(src='/bootstrap/js/bootstrap.min.js')
```

Set viewport metadata for better display on mobile devices

Include themed Bootstrap CSS

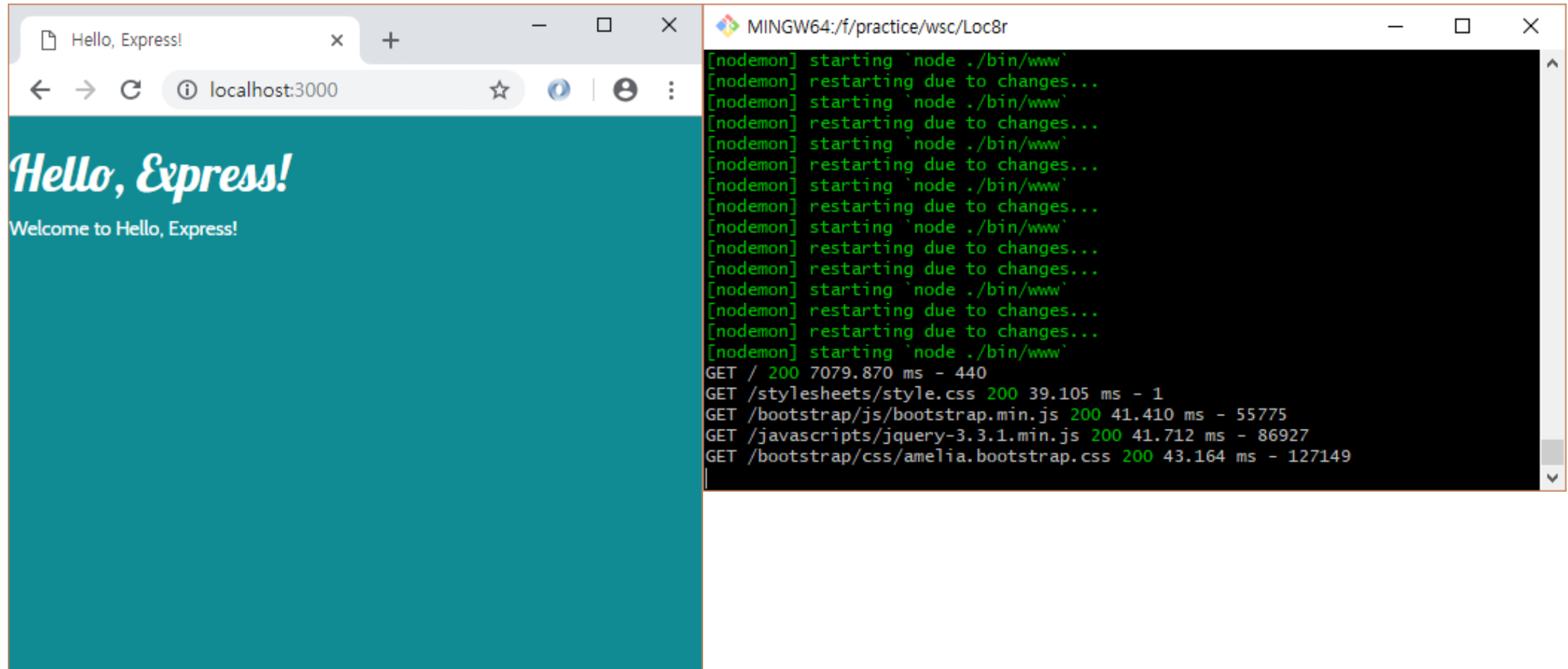
Bring in jQuery as it's needed by Bootstrap

Bring in Bootstrap JavaScript file

마지막으로, 테스트하기 전에 /public/stylesheets/에 있는 **style.css** 파일의 **내용만** 삭제하자.

- ⇒ **Bootstrap** 파일을 디폴트 Express 스타일(**style.css**)로 대체하지 않는다.
- ⇒ 추후 Loc8r 애플리케이션을 개발해 가다가 어느 곳에서 약간의 라인을 추가하여 자신의 스타일을 추가할 것이기 때문에 파일 자체를 삭제할 필요는 없다.

실습과제 23-3



Make it live on Heroku

Node 애플리케이션을 실제 프로덕션 서버에 배포

- Loc8r 애플리케이션을 라이브 URL로 푸시
- 반복해서 빌드할 때 업데이트를 계속해서 푸시할 수 있다
- 서비스 제공 업체 - Google Cloud Platform, Nodejitsu, OpenShift, **Heroku**

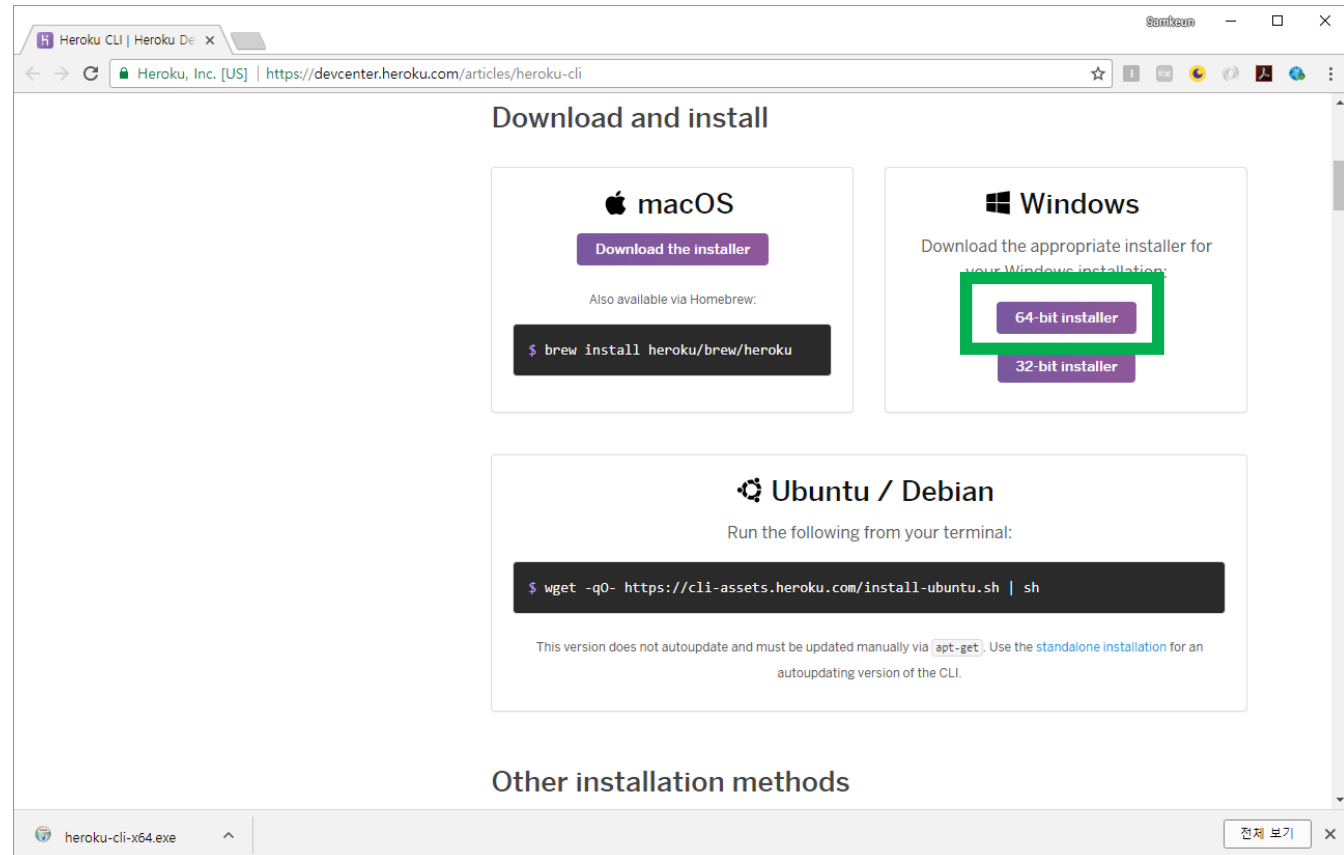
여기서는 Heroku를 사용하기로 한다:

Heroku Sign up: <https://signup.heroku.com/dc> => 회원가입하기!!

Getting Heroku set up

Heroku를 사용하려면 먼저 계정에 등록하고, 개발 컴퓨터에 Heroku Toolbelt를 설치해야 한다.

<https://toolbelt.heroku.com>



UPDATING PACKAGE.JSON

Heroku는 다양한 형태의 코드로 구성된 애플리케이션들을 실행시킬 수 있다.

- ⇒ **Heroku**에게 어떤 애플리케이션이 실행되어야 하는 가를 알려줘야 한다.
- ⇒ **npm**을 패키지 매니저로 사용하는 Node 애플리케이션을 실행 중임을 알리는 것뿐만 아니라, 프로덕션 설정이 개발 설정과 동일한 지 확인하기 위해 **실행중인 버전을 알려줘야** 한다.

실행중인 **Node** 및 **npm** 버전 확인:

```
$ node --version
```

```
$ npm --version
```

Listing 3.8 Adding an engines section to package.json

```
{
  "name": "Loc8r",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "engines": {
    "node": "~4.2.1",
    "npm": "~2.2.0"
  },
  "dependencies": {
    "express": "~4.9.0",
    "body-parser": "~1.8.1",
    "cookie-parser": "~1.3.3",
    "morgan": "~1.3.0",
    "serve-favicon": "~2.1.3",
    "debug": "~2.0.0",
    "jade": "~1.6.0"
  }
}
```

자신이 설치한 버전엔 맞춘다!

Add an engines section to package.json to tell Heroku which platform your application is on, and which version to use

Heroku에 푸시할 때, 애플리케이션이 최신 패치 버전인 Node 4.2와 최신 패치 버전 npm 2.2를 사용한다는 것을 Heroku에게 알려줄 것이다.

CREATING A PROCFILE

Package.json 파일은 Heroku에게 '애플리케이션이 Node 애플리케이션이다'라고 말해 주기는 하지만, 그것을 **시작하는 방법**은 말해주지 않는다.

- ⇒ 이를 위해 **Procfile**을 사용한다.
- ⇒ Procfile은 애플리케이션에 의해 사용된 프로세스 타입과 애플리케이션을 시작시키는데 사용되는 **커맨드**들을 선언하는데 사용된다.

Loc8r의 경우 웹 프로세스를 원하고, Node 애플리케이션을 실행하고 싶다.

- ⇒ 애플리케이션의 루트 폴더에 Procfile이라는 파일을 생성한다. (대소문자 구분/파일 확장자 없이)
- ⇒ Procfile 파일에 다음 행을 입력하자:

```
web: npm start
```

- ⇒ Heroku에 푸시될 때 이 파일은 Heroku에게 애플리케이션이 웹 프로세스를 필요로 하며 `npm start`를 실행해야 한다고 알린다.

Pushing the site live using Git

Heroku는 배치 방법으로 **Git**을 사용한다.

- **Git**의 세계가 복잡하지만 매우 매력적인 방법임에는 틀림없다!

STORING THE APPLICATION IN GIT

① 첫 번째 액션 - 로컬 머신의 Git에 애플리케이션을 저장하는 것이다 (3단계 프로세스):

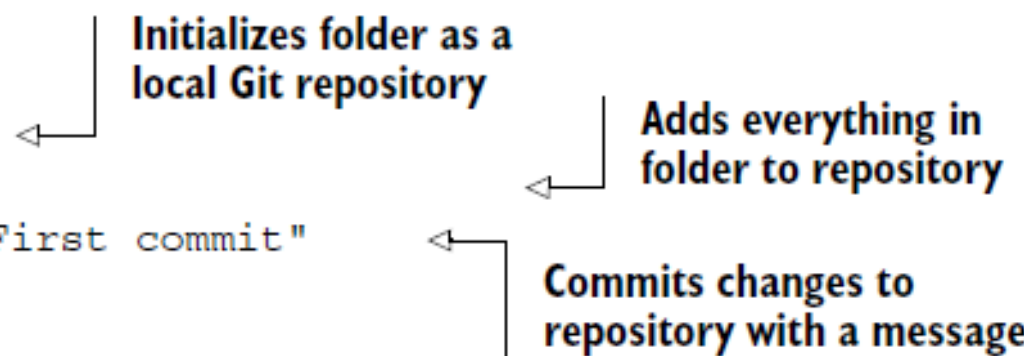
1. 애플리케이션 폴더를 **Git** 저장소로 초기화한다.
2. 저장소에 추가할 파일을 **Git**에게 알려준다.
3. 이러한 변경 사항을 저장소에 적용한다.

② .gitignore 파일 만들기:

```
# Dependency directory
node_modules
```

먼저 애플리케이션이 로컬에서 실행되고 있다면, 터미널에서 **stop**시킨다(Ctrl-C).

다음은 터미널에서 애플리케이션의 **root** 폴더로 옮겨 가서 아래 커맨드들을 수행한다:



```
$ git init
$ git add .
$ git commit -m "First commit"
```

Initializes folder as a local Git repository

Adds everything in folder to repository

Commits changes to repository with a message

위 세 명령을 수행하면, 애플리케이션의 전체 코드베이스를 포함하는 로컬 **Git** 저장소가 만들어진다.

나중에 애플리케이션을 업데이트하고 일부 변경 사항을 적용하려면, 두 번째부터 두 명령을 메시지 내용만 바꾸어서 저장소를 업데이트하면 된다.

로컬 저장소가 준비되었다. 이제 **Heroku** 애플리케이션을 만들 차례이다.


```
Application Root $ git init
```

```
Application Root $ git add .
```

```
Application Root $ git config --global user.email "YOUR EMAIL"
```

```
Application Root $ git config --global user.name "YOUR NAME"
```

```
Application Root $ git commit -m "First commit"
```

CREATING THE HEROKU APPLICATION

다음 단계는 Heroku에 애플리케이션을 생성한다:

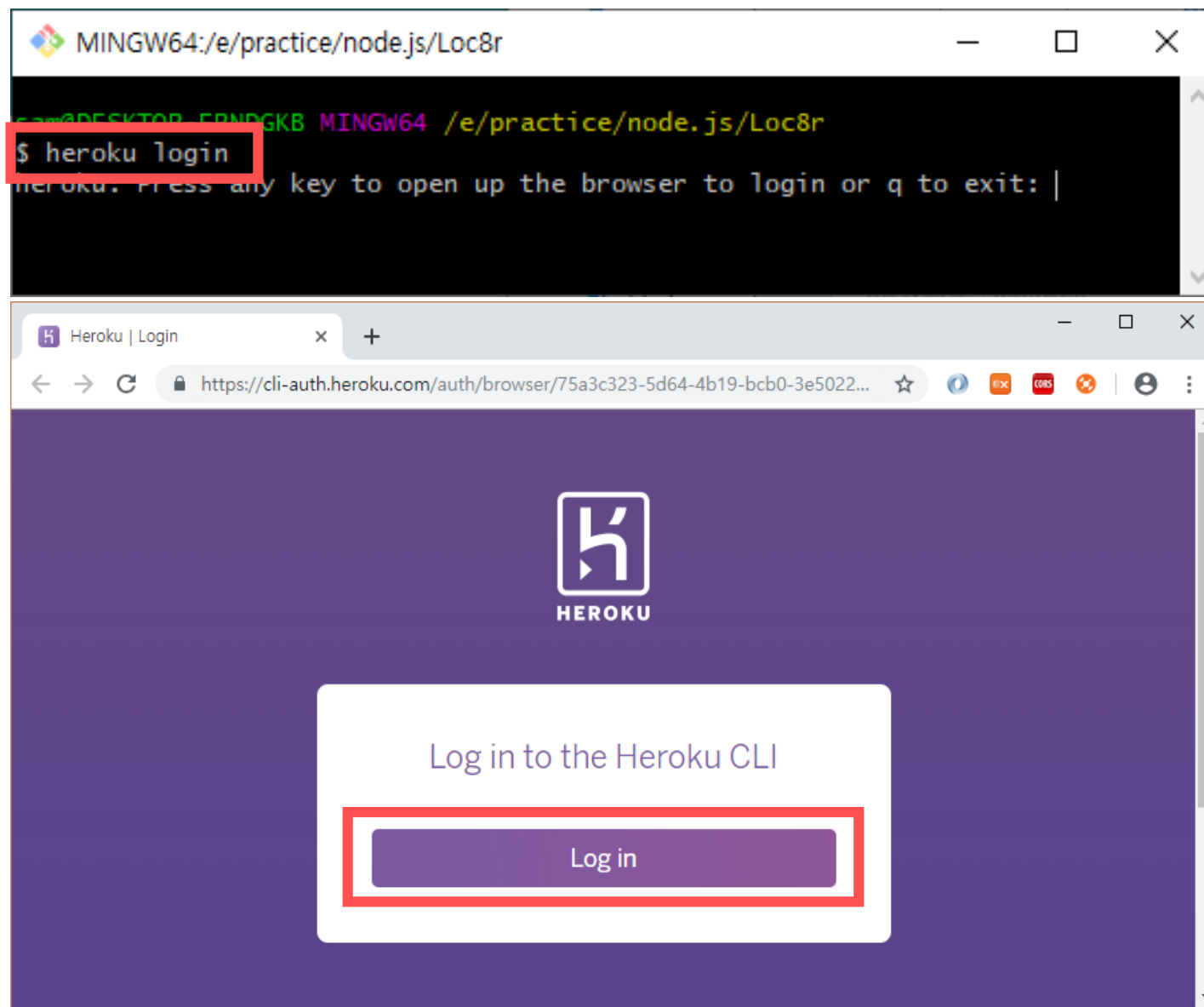
방법 ① - 터미널 명령으로 수행:

```
$ heroku login
```

```
$ heroku create YOUR_APP_NAME
```

방법 ② - Heroku GUI에서 애플리케이션 생성

방법 ② - Heroku GUI에서 애플리케이션 생성



방법 ② - Heroku GUI에서 애플리케이션 생성

1. Heroku Login page. The Heroku logo is highlighted.

2. Heroku dashboard (Personal apps). The 'Create new app' button is highlighted.

3. 'Create New App' page. The 'App name' field contains 'neardust'. A label '애플리케이션 명' (Application Name) points to the field. The 'Choose a region' dropdown is set to 'United States'. The 'Create app' button is visible.

4. Heroku dashboard (Personal apps) showing the newly created app 'neardust' in the list, highlighted.

Heroku에 애플리케이션을 생성했다면:

터미널에 애플리케이션이 상주해 있는 URL, Git 저장소 주소, 원격 저장소 이름이 표시된다.
현재는 로컬 Git 저장소에 저장되어 있다.

Heroku 클라우드 상의 Git 저장소를 Remote 저장소로 지정해야 한다:

```
$ heroku git:remote -a YOUR_APP_NAME
```

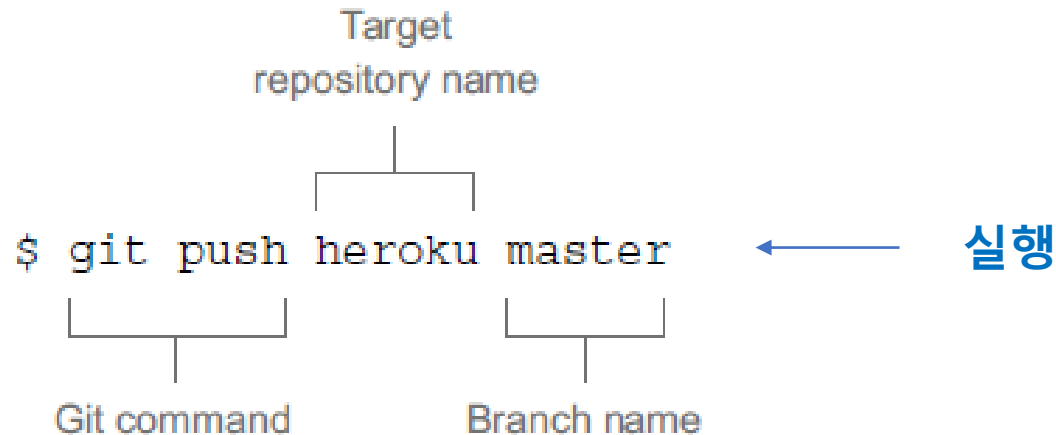
```
set git remote heroku to https://git.heroku.com/YOUR_APP_NAME.git
```

브라우저에서 Heroku 계정에 로그인해 보면, 애플리케이션이 거기 존재하고 있는 것을 볼 수 있다.
이제 Heroku 상에 애플리케이션을 위한 컨테이너를 갖게 된 것이다.

DEPLOYING THE APPLICATION TO HEROKU

지금까지 로컬 Git 저장소에 저장된 애플리케이션을 만들었고, Heroku에 새로운 원격 저장소를 생성했다.
원격 저장소는 현재 비어 있다.

로컬 저장소의 내용을 Heroku 원격 저장소에 push할 필요가 있다:



이 명령은 로컬 Git 저장소의 내용을 heroku 원격 저장소로 푸시한다.
현재 저장소에 하나의 브랜치(master 브랜치)만 있으므로 Heroku로 푸시할 수 있다.

STARTING A WEB DYNO ON HEROKU

Heroku는 애플리케이션 실행 및 크기 조절을 위해 dynos 개념을 사용한다.

- dynos가 많을수록 애플리케이션에서 사용할 수 있는 시스템 자원과 프로세스가 늘어난다.
- 무료로 하나의 웹 dyno를 사용할 수 있다.

애플리케이션을 온라인으로 보기 전에 단일 웹 dyno를 추가해야 한다.

이것은 간단한 터미널 명령으로 쉽게 할 수 있다:

```
$ heroku ps:scale web=1
```

```
Scaling web dynos... done, now running 1
```

이제 라이브 URL을 확인해 보자.

VIEWING THE APPLICATION ON A LIVE URL

모든 것이 이제 완료되었으며, 애플리케이션이 인터넷에 게시되었다!

Heroku 웹 사이트의 계정을 통해, 또는 아래 터미널 명령을 사용하여 볼 수 있다:

```
$ heroku open
```

이렇게 하면 기본 브라우저에서 애플리케이션이 시작되고 그림 3.10과 같은 화면이 나타난다.

사용자마다 URL은 물론 다를 수 있다.

프로토타입을 액세스 가능한 URL로 유지하는 것은 브라우저간/장치간 테스트는 물론 동료/파트너에게 보내는 것이 매우 편리하다.

A SIMPLE UPDATE PROCESS

Heroku 애플리케이션이 설정되어 있으므로 업데이트하는 것이 정말 쉽다.

새로운 변경 사항을 적용할 때마다 세 가지 터미널 명령만 있으면 된다:

```
$ git add .  
$ git commit -m "Commit message here"  
$ git push heroku master
```

Add all changes to local Git repository (points to `$ git add .`)

Commit changes to local repository with a useful message (points to `$ git commit -m "Commit message here"`)

Push changes to Heroku repository (points to `$ git push heroku master`)

적어도 지금은 이게 전부다!!

여러 개발자와 브랜치들을 다루어야 하는 경우 상황이 다소 복잡해 질 수 있지만, Git을 사용하여 Heroku에 코드를 푸시하는 실제 과정은 동일하게 유지된다.

실습과제 23-4

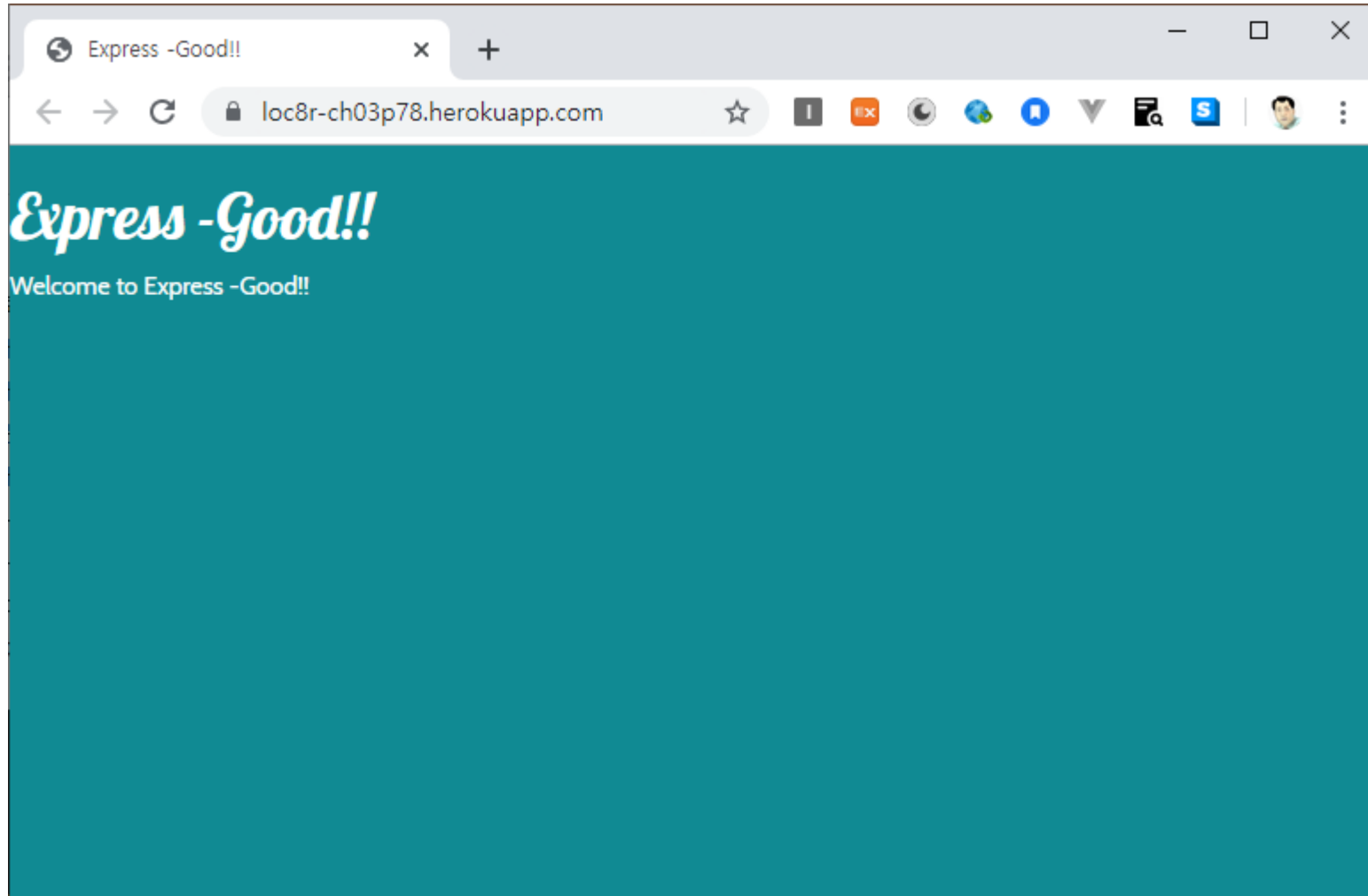


Figure 3.10

<https://loc8r-ch03p78.herokuapp.com/>

