

3장 Node.js를 이용한 웹 서버 프로그래밍

3.1 Node.js 소개

3.1.1 Node.js란 무엇인가?

Node.js(이하 Node)는 공식 홈페이지(<https://nodejs.org>)에서 다음과 같이 소개되고 있다.

Node.js는 크롬 브라우저의 V8 자바스크립트 엔진 위에서 작성된 자바스크립트 실행 시스템(runtime)이다. Node.js는 이벤트 중심(event-driven)의 비중단적(non-blocking) 입출력을 사용하기 때문에 매우 가벼우면서(lightweight) 효율적이다.

위의 소개를 좀 더 자세히 살펴보자. 우선 Node는 자바스크립트 프로그램을 실행시키는 시스템이다. 자바스크립트 프로그램은 이미 각종 브라우저에서 실행된다는 것을 우리는 알고 있다. 그런데 왜 또 실행 시스템이 필요한 것일까?

처음 자바스크립트 실행 시스템은 브라우저에 내장되어 프로그램을 한 줄씩 실행하는 인터프리터였기 때문에 자바스크립트 프로그램의 실행 속도는 매우 느렸다. 그러나 2008년 Google에서 크롬 브라우저가 개발되면서 자바스크립트 프로그램의 실행 속도는 매우 빠르게 개선되었다. 이는 이제 자바스크립트를 웹 브라우저에서만 사용할 것이 아니라 서버 측에서도 사용할 수 있을 정도로 충분히 실행속도가 빨라졌다는 것을 의미한다. 특히 크롬 브라우저에서 동작하던 V8 엔진은 자바스크립트 프로그램을 실행 컴퓨터의 기계어로 번역하여 실행함으로써 자바스크립트 프로그램의 실행속도를 개선하는데 결정적으로 기여하였다. Node는 V8 엔진을 이용하여 자바스크립트 프로그램을 서버 측에서 실행할 수 있도록 지원하는 실행 시스템이다.

다음으로 Node의 특징을 이벤트 중심의 비중단적 입출력을 들고 있다. 이 개념은 아래 그림이 잘 보여주고 있다.



그림 1 Node의 이벤트 중심 비중단적 입출력 개념

Node 프로그램은 서버에서 하나의 실행 흐름으로 수행된다. 이 실행 흐름에서는 요청을 받아서 처리하는 작업을 계속하는데 이 흐름이 그림의 가운데에 나와 있는 이벤트 루프이다. 이벤트 루프라고 부르는 이유는 이 각종 요청 및 입출력 작업의 완료 등이 이 실행 흐름에 이벤트로 전달되기 때문이다. 따라서 이 실행 흐름은 들어오는 이벤트를 처리하는 작업을 계속 수행한다. 즉 하나의 이벤트를 처리하고 또 다른 이벤트를 처리한다. 이렇게 이벤트를 처리를 계속하기 때문에 이벤트 루프라고 부른다.

이벤트를 처리하는 도중에 시간이 오래 걸리는 입출력을 해야 하면 어떻게 할까? 이런 예는 큰 이미지 파일을 요청하는 HTTP-요청을 처리하는 것을 들 수 있다. 서버의 디스크에서 큰 이미지 파일을 읽어오려면 시간이 많이 걸린다. 이때 이벤트 루프는 파일을 다 읽을 때까지 기다리지 않고 다른 요청을 처리한다. 나중에 파일을 다 읽어서 전송 준비가 되면 입출력 종료 이벤트가 발생하고 이때 이벤트 루프는 주어진 콜백 함수를 실행함으로써 이벤트를 처리를 완료한다. 이렇게 이벤트 처리 중에 시간이 걸리는 작업이 있다면 이벤트 루프는 이 작업이 끝날 때까지 기다리지 않고 다른 작업을 하기 때문에 “비중단적 입출력”이라고 부른다.

이제 Node 홈 페이지의 소개 내용이 어느 정도 이해가 되었을 것이다. 계속해서 Node의 특징을 살펴보면 Node는 Windows, Linux, OS X 등 운영체제에 관계없이 잘 실행되기 때문에 플랫폼 독립적(cross platform)이다. 또한 Node는 프로그램이 대중에게 공개된 개방 소스(open source) 프로그램이다.

Node 프로그램은 웹브라우저에서 실행되는 것이 아니라 웹 서버 즉 컴퓨터에서 실행되기 때문에 Node에서는 JavaScript API 중 웹브라우저와 관련된 API가 제거되었고 파일이나 네트워크와 같이 운영체제와 관련된 API가 도입되었다.

3.1.2 Node.js를 사용하는 장점

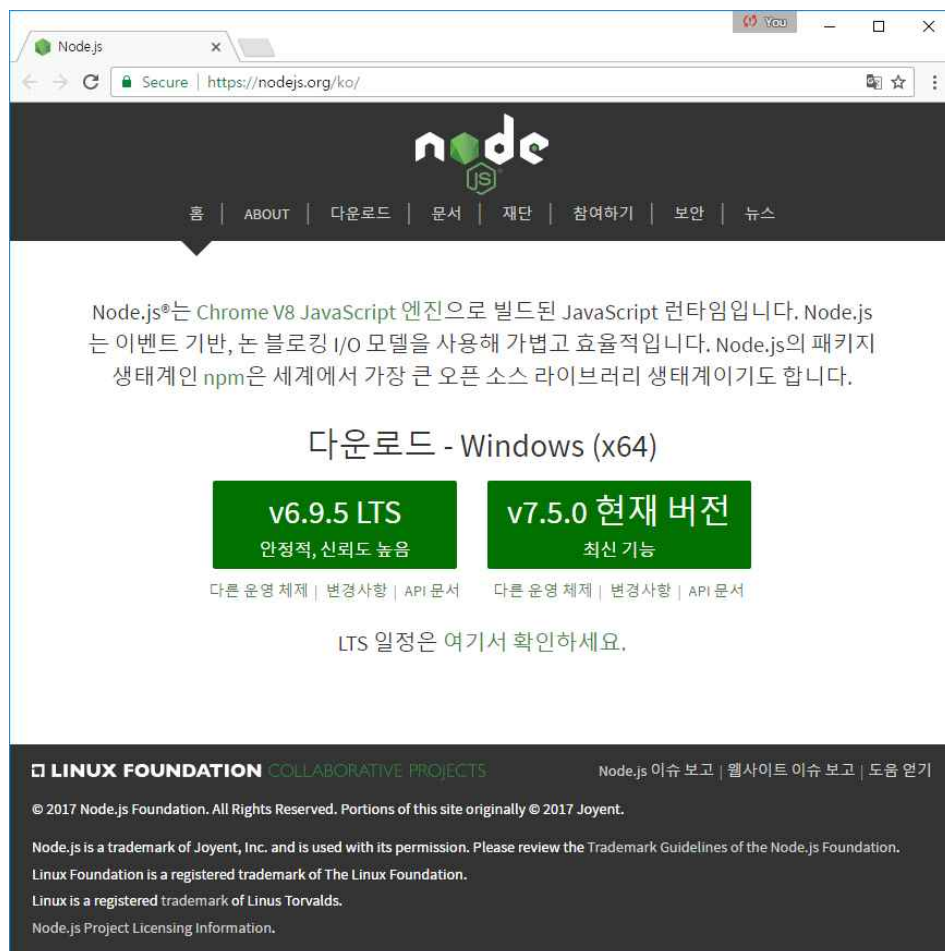
Node를 사용하는 것이 다른 방법에 비해 어떤 장점이 있을까?

- 속도가 빠르다. Node는 대량의 요청을 빠르게 처리할 수 있도록 최적화되어 있기 때문에 다양한 웹 개발 문제에 잘 사용할 수 있다.
- 코드가 JavaScript 언어로 작성된다. JavaScript 언어는 웹 클라이언트 프로그램을 위한 표준 언어이므로 웹 개발자는 이미 이 언어에 익숙한 경우가 많다. 서버 프로그램도 JavaScript로 작성할 수 있다면 새로운 언어를 학습할 필요가 없어서 편리하다.
- 다른 웹 서버 언어인 PHP, Python 등과 비교할 때 JavaScript 언어는 비교적 나중에 만들어진 언어로 최근 언어 설계 이론을 반영하여 작성되었다. 또한 CoffeeScript, LiveScript 등의 많은 최신 언어들이 JavaScript 언어로 변환된다.
- Node의 패키지 관리자(package manager) 프로그램인 npm을 이용하면 이미 개발된 수 천 개의 패키지를 사용할 수 있다. npm은 패키지 의존성을 관리해 주기 때문에 프로그램 빌드 과정이 쉬워진다.

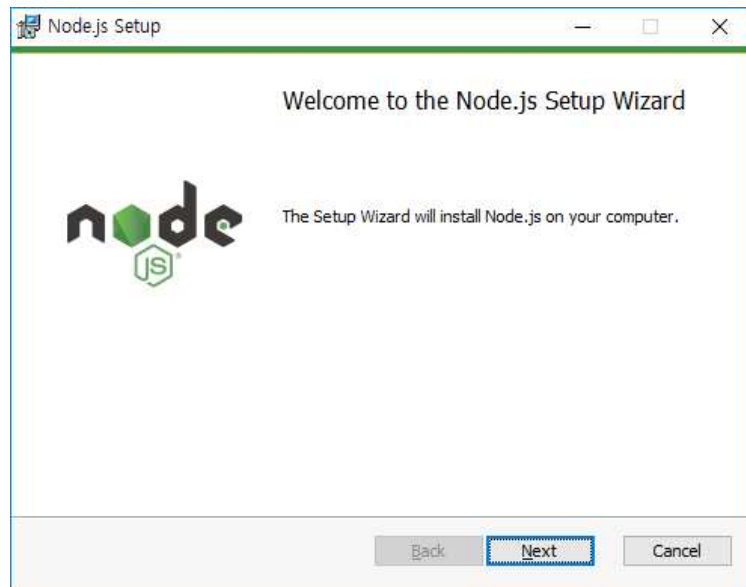
- Node는 이식성이 높아서 Windows, Linux, OS X, FreeBSD 등 다양한 운영체제에서 잘 실행된다.
- 개발자 단체(developer community)가 잘 발달되어 있어서 필요한 경우 도움을 쉽게 받을 수 있다.

3.1.3 Node 설치하기

Express를 사용하기 위해서는 먼저 Node.js와 NPM을 설치해야 한다. 이 설치하는 매우 간단하다. 먼저 Node의 공식 사이트 <https://nodejs.org/ko/>를 방문하자. 다음과 같은 화면이 나온다.



위를 보면 Node의 두 가지 버전이 나오는데 하나는 안정적인 버전이고 다른 하나는 최신 개발 중인 버전이다. 어느 것을 설치해도 문제가 없으나 여기서는 안정적인 버전인 v.6.9.5를 설치하도록 한다. 초록색 사각형을 클릭하면 윈도우 설치 파일인 node-v6.9.5-x64.msi 파일이 저장된다. 이 파일을 실행시킨다. 다음과 같은 설치 화면이 나온다.



이 설치화면에서 Next 버튼을 클릭하고, 저작권에 동의하고 계속 Next 버튼을 클릭하면 Node의 설치가 끝난다. 그러면 Windows 운영체제의 경우 C:\Program Files\ 아래에 nodejs라는 폴더가 생성되고 필요한 파일들이 해당 폴더에 설치되었음을 볼 수 있다. 또한 Node를 설치하면 NPM도 함께 설치된다.

Node를 설치했는데 제대로 설치되었는지 확인은 어떻게 하는가? 먼저 Windows의 경우 시작 버튼을 누르고 검색 창에 cmd라고 입력한다. 그러면 다음 그림과 같이 명령 프롬프트 창이 표시될 것이다.

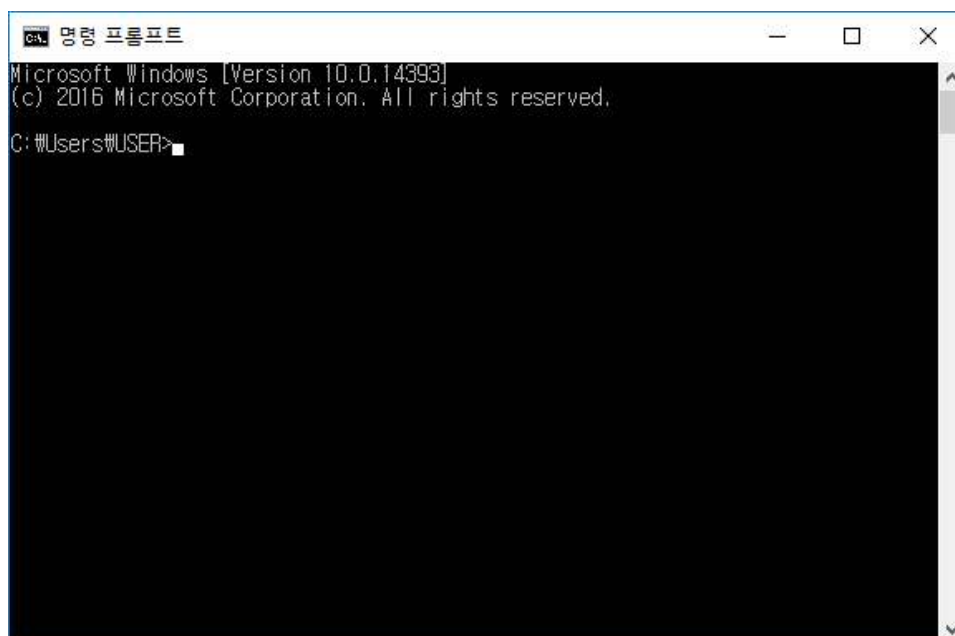


그림 4 Windows 명령 프롬프트 창 실행

이 명령 프롬프트 창에 다음 명령을 입력하자. Node와 NPM 버전이 표시되면 Node가 문제 없이 잘 설치된 것이다.

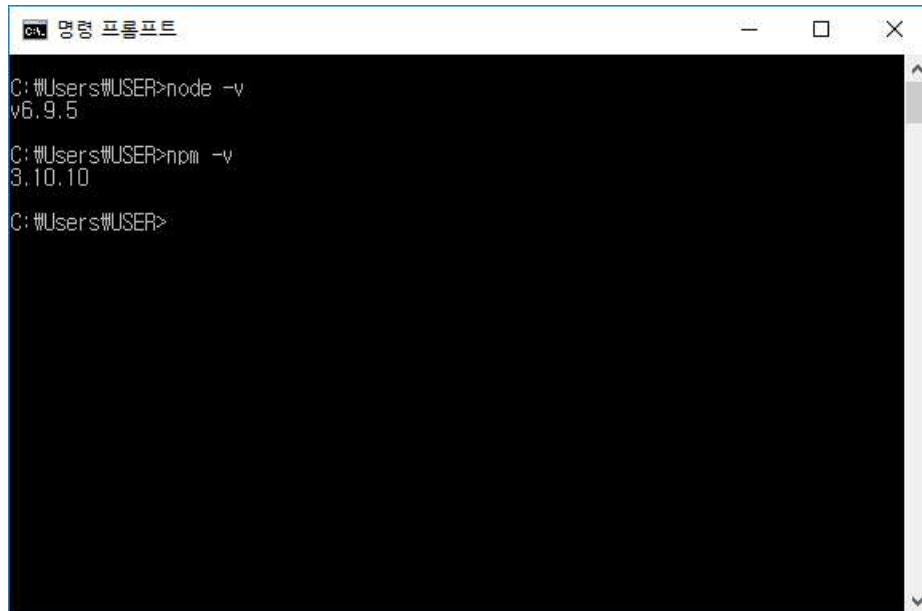


그림 5 Node와 npm 버전 확인

3.1.4 Node로 Hello World 프로그램 하기

일반적으로 프로그래밍 언어를 배우기 위해 처음으로 작성하는 프로그램이 “Hello World”라는 메시지를 화면에 출력하는 프로그램이다.

Node 프로그래밍을 배우기 위해 우리도 “Hello World”를 화면에 출력하는 프로그램을 작성해 보자. 먼저 적당한 위치에 프로젝트를 위한 폴더를 하나 만든다(본 교재에서는 helloNode라는 폴더를 새로 만들었다). 이제 이 폴더를 Code로 연 다음 새로운 파일 “hello.js”를 추가한다. “hello.js” 파일에 다음과 같이 입력한다.

```
console.log('hello world');
```

이제 이 파일을 실행해야 한다. 간단한 방법은 Code의 “보기” 메뉴에서 “통합터미널” 메뉴를 선택한다. 그러면 아래와 같이 Code 화면 아래에 명령을 입력할 수 있는 명령 창(powershell)이 나타난다. 이 명령 창에 “node hello.js”라고 입력하면 우리가 작성한 “hello.js” 프로그램이 실행되어 화면에 “hello world”라 표시된다.

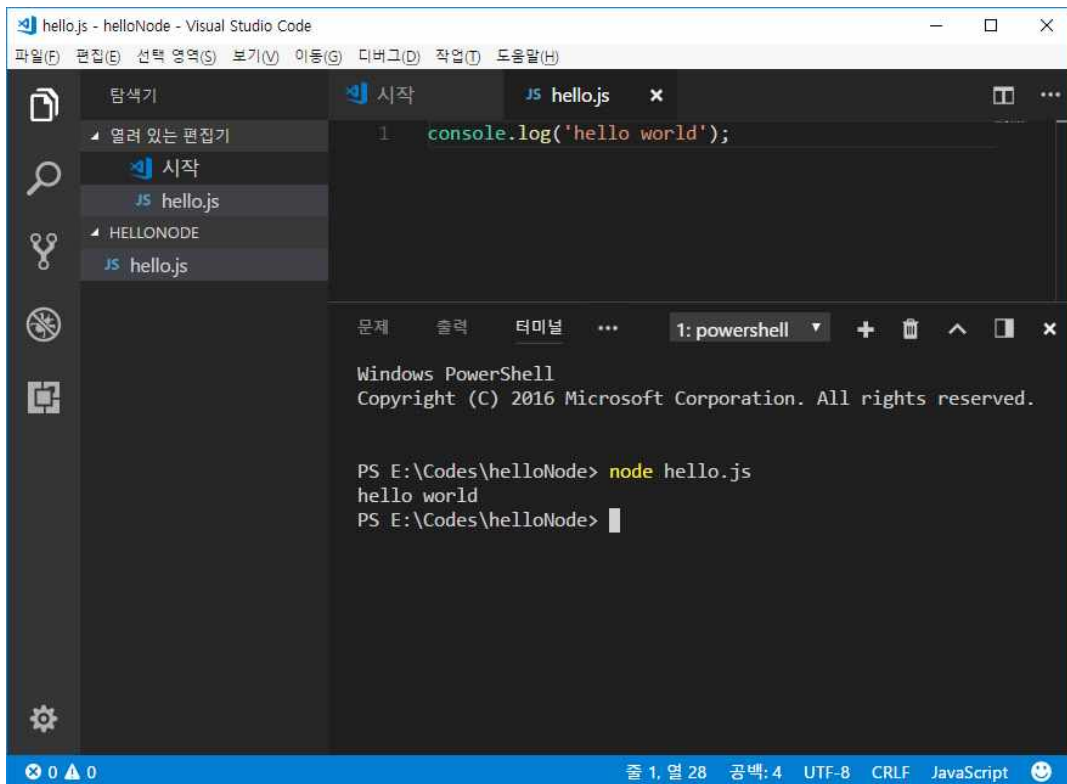


그림 6 Code에서 Node 프로그램 작성 및 실행

위의 프로그램을 보고 매우 낮이 익다는 생각이 들면 맞다. 우리는 클라이언트 측 자바스크립트 언어를 배울 때 `console.log()` 메서드를 이용하여 브라우저의 콘솔 창에 여러 가지 정보를 출력한 바 있다. 이제 우리는 서버에서 자바스크립트 프로그램을 실행시키기 때문에 콘솔은 명령 창 화면이 된다. 그리고 노드 프로그램을 실행하기 위한 명령은 “node”이다.

3.2 Node로 간단한 웹 서버 만들기

3.2.1 응답으로 “Hello World”를 보내는 웹 서버

HTTP 요청이 들어오면 “Hello World” 텍스트를 보내는 응답으로 보내는 웹 서버를 Node의 HTTP 패키지를 이용하면 쉽게 만들 수 있다. 먼저 “httpServer1.js”라는 파일을 새로 만든 후 아래의 내용을 그대로 입력한다.

```
var http = require('http');
var server = http.createServer();
server.on('request', function(req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.write('Hello World');
    res.end();
});
server.listen(8080);
console.log('서버가 127.0.0.1:8080에서 시작되었습니다.');
```

위의 프로그램을 입력한 후 node 명령을 통해 실행한다.

```
문제 출력 디버그 콘솔 터미널 1: node
PS E:\Codes\helloNode> node .\httpServer1.js
Server가 http://127.0.0.1:8080/ 에서 시작되었습니다.
```

그림 7 Node를 이용한 HTTP 서버 실행

지금은 단지 콘솔에 서버가 시작되었다는 메시지만 출력될 뿐 아무것도 달라지지 않았다. 그러나 사실은 우리가 처음으로 만든 웹 서버가 배후에서 동작하고 있다. 이를 확인하기 위하여 웹 브라우저를 이용하여 http://localhost:8080(http://127.0.0.1:8080으로 접속해도 같은 결과를 낸다)으로 접속해 보자. 그러면 웹 브라우저에 우리가 서버에서 보낸 “Hello World” 메시지가 출력될 것이다.

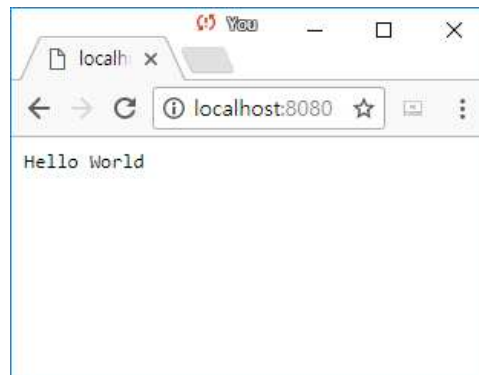


그림 8 웹 서버에 접속한 결과

웹 서버 프로그램을 좀 더 자세히 살펴보자. 가장 먼저 나오는 문장은 모듈 수입 문장이다.

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

이 문장의 의미는 Node 시스템에서 이미 정의되어 있는 “http” 모듈을 가지고 와서 프로그램 안에서 사용하겠다는 의미이다. “http” 모듈은 하나의 객체에다 수출하고자 하는 기능을 담아서 보내는데 이 객체를 받아서 http라는 변수에 저장하고 사용하겠다는 것이다. 모듈의 작성과 수입에 대해서는 다음에 좀 더 자세히 공부할 것이다.

다음 문장은 http 모듈에서 수출한 기능인 createServer() 메서드를 이용하여 http-서버(웹 서버)를 만든 다음 이 서버를 server 변수에 저장하였다. 이때 서버는 만들어졌으나 아직 작동하지는 않는 상태이다.

다음으로 on() 메서드를 이용하여 서버에 요청(request)이 들어오면 어떤 일을 해야 하는지를 무명 함수로 정의하였다. 이 함수는 두 개의 매개변수를 갖는데 첫 번째 매개변수는 요청 즉 HttpRequest 객체를 표시하며 두 번째 매개변수는 웹 서버의 응답 즉 HttpResponse 객체를 표시한다. 이 웹 서버에서는 요청에 들어오면 응답 객체 헤더에 200 코드(응답이 무사히 처리

되었음을 표시하는 코드임)와 보내는 자료 형태(text/plain)를 적는다. 다음으로 응답에다 “Hello World”라는 스트링을 적고 end() 메서드를 이용하여 응답을 브라우저에게 전송한다.

마지막 문장은 listen()메서드를 사용하여 서버를 동작시킨다. 서버를 동작시킬 때 서버가 요청을 받아들이는 포트 번호를 지정한다. 이 예에서는 8080 포트를 사용하였다. 포트가 다른 프로그램에 의해 사용되고 있을 경우(2000번 이하의 포트는 다른 프로그램이 사용할 가능성이 있다) 다른 포트 번호를 지정하면 문제가 해결된다.

위의 프로그램을 다음과 같이 간단히 작성할 수도 있다.

```
var http = require('http');
var server = http.createServer(function(req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World');
}).listen(8080);
console.log('서버가 127.0.0.1:8080에서 시작되었습니다.');
```

모듈 제작과 수입

Node에서 모듈은 JavaScript로 작성된 라이브러리나 파일을 의미한다. 이 모듈을 수입해서 사용하려면 Node의 require() 함수를 사용한다. 다른 라이브러리와 마찬가지로 Express도 하나의 모듈이다.

모듈을 수입하는 예는 위에서 본 적이 있다. 다시 쓰면 다음과 같다.

```
var mod = require("module_name");
```

여러분이 모듈을 만들어서 다른 쪽에서 수입할 수 있게 하려면 수출하고자 하는 객체들을 exports 객체에 지정하면 된다. 다음 예를 보자. 이 파일이 “square.js”라는 이름으로 저장되어 있다고 가정한다.

```
exports.area = function(width) { return width * width; };
exports.perimeter = function (width) { return 4 * width; };
```

이 모듈을 수입해서 사용하는 예는 다음과 같다.

```
var square = require("./square");
console.log("한 변의 길이가 4인 정사각형의 면적은 " + square.area(4) + "입니다");
```

위의 예에서 보듯이 수입하고자 하는 파일 이름이 “.js”로 끝이 날 경우 require() 문에서 “.js”를 써 줄 필요가 없다.

만약 모듈에서 수출하려는 것이 객체 1개 이면 다음과 같이 쓸 수도 있다.


```
module.exports = {
  area: function (width) { return width * width; },
  perimeter: function (width) { return 4 * width; }
};
```

비동기적(Asynchronous) APIs

어떤 연산을 하는데 시간이 많이 걸릴 경우 JavaScript 프로그램은 일반적으로 동기적 API보다는 비동기적 API를 사용한다. 다음 log 함수는 동기적이기 때문에 먼저 위의 log 함수가 수행되고 그 수행이 끝나야 다음 log 함수가 수행된다.

```
console.log("Hello");
console.log("World");
```

이와는 다르게 비동기적 API는 연산을 시작하고 난 후 (연산이 끝나기 전에) 바로 복귀한다. 연산이 끝나면 콜백 함수라는 개념을 이용하여 추가적인 연산을 할 수 있다. 다음 예를 보자.

```
setTimeout(function() {
  console.log("Hello");
}, 5000);
console.log("World");
```

이 프로그램의 실행 결과는 World가 먼저 출력되고 난 후 Hello가 출력된다. setTimeout() 함수가 먼저 실행이 되었기 때문에 Hello가 먼저 나와야 할 것 같지만 결과는 그렇지 않다. 그 이유는 setTimeout() 함수가 비동기적 함수이기 때문이다. 이 함수는 시작이 된 후 끝날 때까지 기다리지 않고 바로 반환된다. 이 함수가 끝나려면 5초나 걸리기 때문에 그 다음 문장 즉 World를 출력하는 함수가 실행된다. 5초가 지난 후 setTimeout 함수가 끝나면 지정된 콜백 함수를 호출하게 되고 여기서 Hello가 출력된다.

비동기적 API를 이용할 때, 연산이 끝난 후 수행해야 할 작업을 지정하는 여러 가지 방법이 있지만 콜백 함수를 지정하는 것이 가장 널리 사용되는 방법이다.

3.2.2 HTML 파일을 보내는 웹 서버 작성

위에서 작성한 웹 서버는 너무 간단하였다. 모든 요청에 대해 "Hello World"라는 스트링을 응답으로 전송하였다. 조금 더 현실적인 웹 서버를 작성해 보자. 이 웹 서버에서는 요청이 들어오면 HTML 파일을 응답으로 보낸다. 응답으로 보낼 HTML 파일 이름은 원하는 대로 정할 수 있지만 여기서는 "index.html"이라고 하겠다.

먼저 응답으로 보낼 "index.html" 파일을 웹 서버가 있는 폴더에 다음과 같이 작성한다.

```
<!doctype html>
<html>
  <head>
```

```

    <meta charset="utf-8">
    <title> Node 서버 HTML 파일 응답 페이지</title>
  </head>
  <body>
    <h2> Node 웹 서버가 보낸 HTML 파일 </h2>
    <p>
      안녕하세요? Node는 HTML 파일을 응답으로 보낼 수도
      있습니다.
    </p>
  </body>
</html>

```

다음으로 httpServer2.js란 프로그램을 다음과 같이 입력한다.

```

var http = require('http');
var fs = require('fs');
http.createServer(function(req, res) {
  fs.readFile('index.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(data);
  });
}).listen(8080);
console.log('서버가 127.0.0.1:8080에서 시작되었습니다.');
```

다음은 위의 httpServer2.js를 실행한 후 웹 브라우저로 서버에 접속한 결과이다. 웹 브라우저로 HTML 파일이 전송되어 화면에 표시됨을 알 수 있다.

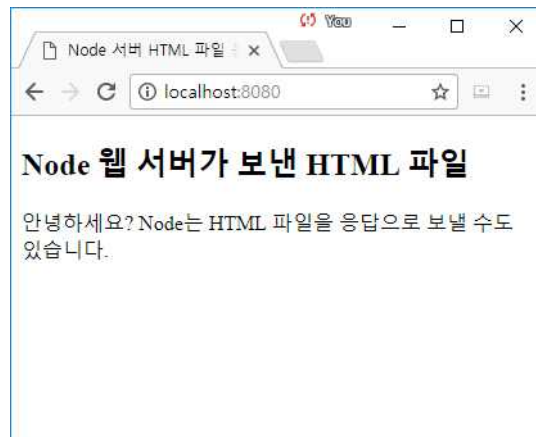


그림 9 서버에서 HTML 파일 보내기

이제 프로그램에서 변경된 부분을 주목해 보자. 먼저 다음 줄이 추가되었음을 알 수 있다.

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

fs 모듈은 파일 시스템 모듈로 이미 설치된 모듈 중 하나이다. 이 모듈에는 파일 열기, 닫기,

읽기, 쓰기에 대한 메서드들이 들어있다. 따라서 파일을 읽고 쓰기 위해서는 이 fs 모듈을 수입해서 사용하여야 한다.

다음으로 fs 모듈에 정의된 readFile() 메서드를 사용하였다. readFile() 메서드는 이름이 말해 주듯이 파일에서 자료를 읽는 메서드이다. readFile() 메서드는 읽으려는 파일의 경로와 파일을 다 읽었을 때 수행할 콜백 함수를 매개변수로 가지고 있다. 우리 웹 서버 예제에서는 파일에서 자료를 다 읽었을 경우 무명 함수를 실행하도록 지정하였다. 지정된 무명 함수는 두 개의 매개변수를 받는데 첫 번째 매개변수는 오류 여부를 표시하고 두 번째 매개변수는 읽은 데이터가 들어가 있다. 콜백 함수에서 우리는 파일에서 읽은 데이터를 http 응답 객체인 res에다가 썼다. 또한 데이터를 쓰기 전에 res의 헤더에 'Content-Type'으로 'text/html'을 지정했다는 것도 변경된 점이다.

3.2.3 정적 파일 서버 만들기

지금까지 개발한 웹 서버는 아직도 너무 단순하다. 두 번째 만든 웹 서버는 모든 요청에 대해 오직 하나의 HTML 파일만을 전송한다.

이제 우리는 웹 서버를 고쳐서 사용자가 원하는 파일을 전송하도록 구현해보자. 예를 들면 사용자가 `http://localhost:8080/bird1.jpg`라고 URL을 입력하면 서버의 프로젝트 폴더에 있는 `bird1.jpg` 파일을 전송하는 것이다.

이러한 웹 서버를 구축하는 것은 위의 간단한 웹 서버를 구축하는 것 보다 조금 복잡하다. 따라서 새로운 프로젝트 폴더를 하나 만들자. 본 교재에서는 "static file server"라는 이름으로 프로젝트 폴더를 생성하였다.

이제 프로젝트가 점점 복잡해질수록 많은 패키지들을 사용하게 된다. 패키지는 모듈의 집합 정도로 생각하자. 사용하는 패키지 중에는 Node를 설치할 때 함께 설치된 모듈도 있고 새롭게 설치해야 할 패키지도 있다. 즉 현재 프로젝트는 여러 패키지에 의존하게 된다. 현재 프로젝트에 필요한 패키지를 개발자가 하나하나 찾아서 설치할 수도 있다. 그러나 개발자가 만든 프로그램을 사용하는 사용자에게 필요한 패키지를 찾아서 설치하기를 요구하는 것은 사용자 입장에서 매우 어려운 작업이 될 수 있다.

Node에서는 이러한 문제를 해결하기 위해 Node Package Manager(NPM)을 사용한다. NPM을 사용하기 위해서는 먼저 지금 개발하고 있는 프로젝트가 어떤 패키지를 요구하는지를 명시할 필요가 있다. 현재 프로젝트가 의존하고 있는 패키지를 명시하기 위해서 `package.json` 파일을 사용한다.

`package.json` 파일에는 JSON 형태로 프로젝트에 대한 여러 가지 정보가 저장된다. 정적 파일 웹 서버 개발을 위해 프로젝트 폴더에 `package.json` 파일을 만들고 다음 내용을 입력하자.

```
{
```

```

    "name" : "staticFileServer",
    "version" : "0.1",
    "description": "A Web Server That Servers Static Files",
    "dependencies" : {
        "mime": "~1.2.7"
    }
}

```

위의 package.json 파일에 우리가 개발하려는 프로젝트는 “mime” 모듈을 사용하는데, “mime” 모듈은 1.2.7 이후 버전이면 된다는 것을 명사하였다.

이제 프로젝트 폴더에 가서 다음과 같이 npm을 실행하여 필요한 외부 모듈을 설치한다.

```
\Node\static file server> npm install
```

위의 명령을 실행하면 npm 프로그램이 우리 프로젝트에서 필요한 “mime” 모듈을 찾아서 설치해 준다. 설치된 결과로 다음과 같이 프로젝트 폴더에 “node_modules”라는 폴더가 생기고 그 폴더 아래에 “mime” 폴더가 생겨난 것을 확인할 수 있다.

프로젝트 폴더에 웹 서버의 기능을 담당하는 “server.js” 파일을 생성한 후 다음과 같이 입력하자.

```

var http = require('http');
var fs = require('fs');
var path = require('path');
var mime = require('mime');

function sendError(response) {
    response.writeHead(404, {'Content-Type': 'text/plain'});
    response.write('Error 404: File Not Found');
    response.end();
}

function sendFile(response, filePath, fileContent) {
    console.log('Sending File: ' + filePath);
    response.writeHead(
        200,
        {'Content-Type': mime.lookup(path.basename(filePath))}
    );
    console.log('Content-Type: ' + mime.lookup(path.basename(filePath)));
    response.write(fileContent);
    response.end();
}

function serverStatic(response, absPath) {
    fs.exists(absPath, function (exists) {
        if(exists) {

```

```

        fs.readFile(absPath, function(err, data) {
            if(err) {
                sendError(response);
            }
            else {
                sendFile(response, absPath, data);
            }
        });
    }
    else {
        sendError(response);
    }
});
}

var server = http.createServer(function(request, response) {
    console.log("Sever serving: " + request.url);
    var filePath = null;
    if(request.url == '/') {
        filePath = '/index.html'
    }
    else {
        filePath = request.url;
    }
    var absPath = '.' + filePath;
    serverStatic(response, absPath);
});

server.listen(8080, function () {
    console.log('Static File Server started at port 8080...');
});

```

이 프로그램에서는 이전과는 다르게 “path” 모듈과 “mime” 모듈을 사용하고 있다. “path” 모듈은 파일 경로를 처리하기 위해 필요한 기능들이 들어있는 모듈이다. “mime” 모듈은 파일의 이름을 주면 적절한 MIME 타입(예를 들면 text/html과 같은)을 결정해 주는 모듈이다. 이 모듈을 이용하여 HttpResponse의 “Content-Type” 항목을 적절하게 지정할 수 있다.

파일을 서비스하기 위해 간단한 보조 함수들을 작성하였다. sendError() 함수는 사용자가 요청한 파일이 없을 경우 오류 메시지를 전달하는 함수이다. 응답으로 “Error 404: File Not Found”라는 스트링을 전달한다.

다음으로 sendFile() 함수를 정의하였다. sendFile() 함수는 response 객체와 파일의 경로 그리고 파일의 내용을 입력으로 받는다. 이 함수에서는 먼저 mime.lookup() 메서드를 이용하여 response 헤더에 응답 유형을 지정한다. 다음으로 전달받은 파일 내용을 response 객체에 쓴다.

다음 함수는 serveStatic() 함수이다. 이 함수는 response 객체와 파일의 경로를 입력으로 전

달받는다. 먼저 지정된 파일이 존재하는지를 fs.exists() 메서드를 이용하여 검사한다. 파일이 존재하면 해당 파일을 읽어서 sendFile() 함수를 이용하여 브라우저에 전달한다. 만약 파일이 존재하지 않으면 sendError() 함수를 이용하여 오류 메시지를 전달한다.

마지막으로 서버를 생성하는 부분이 수정되었다. request.url은 요청하는 파일에 대한 정보를 가지고 있는데 request에서 파일을 지정하지 않을 경우 자동으로 “index.html” 파일을 전달한다. 사용자가 파일을 지정하면 해당 파일을 전달한다. 마지막으로 server.listen() 메서드를 이용하여 서버를 8080 포트를 대상으로 실행한다.

서버를 실행시킨 후 웹 브라우저를 이용하여 접속해 보자. 그러면 다음과 같이 “index.html” 페이지가 표시됨을 알 수 있다.

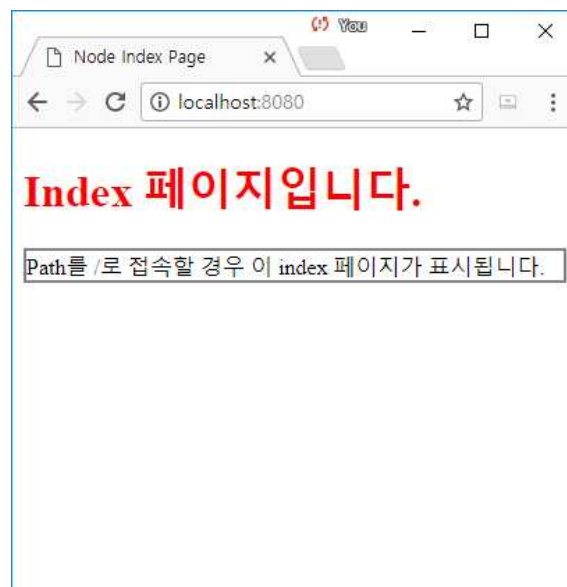


그림 10 index.html 출력

URL 주소를 변경하여 “http://127.0.0.1:8080/bird1.jpg”를 입력하면 다음과 같이 이미지 파일이 출력됨을 알 수 있다. 물론 접속하기 전에 “bird1.jpg” 파일을 프로젝트 폴더에 옮겨놓아야 한다.

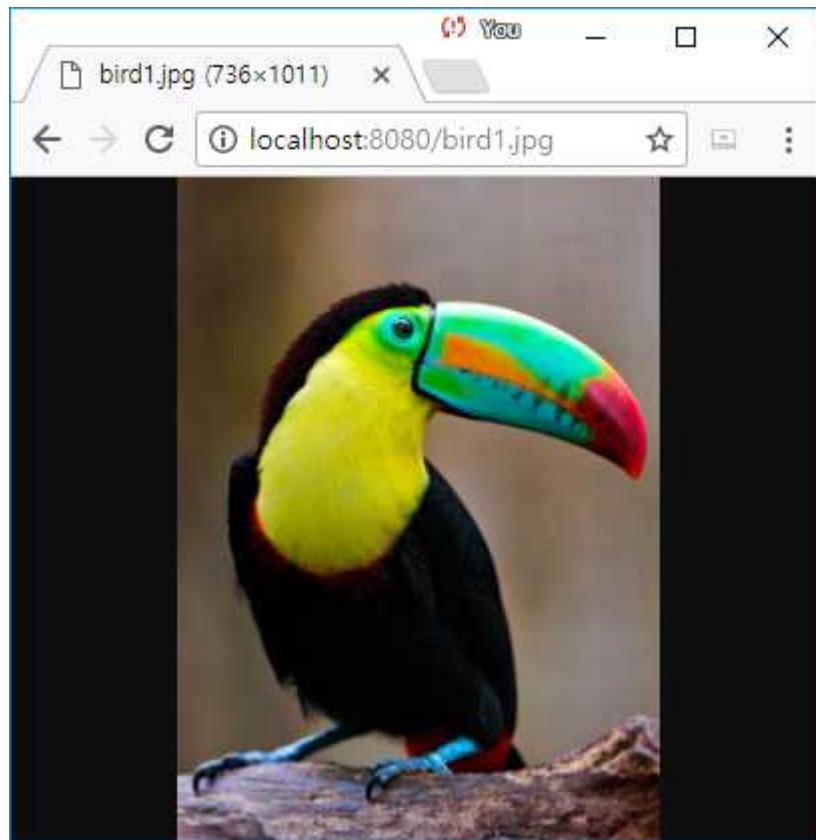


그림 11 Node를 이용한 정적 파일 서비스

3.3 Express 프레임워크

이제 Node를 사용하면 간단하게 웹서버를 만들 수 있다는 것을 알았다. 이것으로 충분한가? 그렇지 않다. 예를 들면 HTTP 요청 중 GET 방식으로 보낸 요청과 POST 방식으로 보낸 요청을 다르게 처리하기, 서버 파일에 대한 요청, 템플릿을 사용하여 동적으로 웹페이지를 생성하기 등의 기능은 Node에서 지원하지 않는다. 물론 프로그램을 많이 작성하면 이런 기능들을 Node에서 구현할 수 있지만 다른 사람이 이미 다 해 놓은 것을 또 다시 할 필요는 없다. 이러한 라이브러리와 개발 패턴을 제공하는 것이 프레임워크이다.

Express는 가장 널리 사용되는 Node 웹 프레임워크이다. Express는 다른 여러 가지 Node 웹 프레임워크의 기반 시스템이다. Express는 다음과 같은 기능을 제공한다.

- 다른 HTTP 동사(POST, GET, DELETE 등)와 다른 URL 경로에 대한 처리기 작성
- 템플릿에 자료를 삽입함으로써 결과를 생성하는 뷰(view) 생성 엔진이 포함되어 있음
- 접속을 위한 포트 번호라든가 응답을 생성하기 위한 템플릿 경로 등을 지정할 수 있게 한다.

- 요청 처리 중간에 “middleware”를 삽입함으로써 필요한 기능을 요청 처리 과정에 더할 수 있다.

3.3.1 Express의 설치

Express 개발환경은 Node.js, NPM 패키지 관리자, Express Application Generator 등을 포함한다. 우리는 이미 Node.js와 NPM을 우리 컴퓨터에 설치하였다. 이제 Express를 설치하자. Express를 현재 프로젝트에만 설치할 수도 있고 모든 프로젝트에서 사용할 수 있도록 전역적으로 설치할 수도 있다. 전역적으로 설치하려면 npm 명령에 -g 옵션을 준다. 다음과 같이 Express를 전역적으로 설치하자.

```
npm install express -g
```

또한 다음 npm 을 사용하여 Express Application Generator를 전역적으로 설치하자.

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

이 프로그램은 Express 웹 응용의 골격을 MVC 패턴에 따라서 자동으로 생성해 주는 프로그램이다. Express Application Generator는 필수적으로 설치해야 하는 것은 아니다. 여러분이 MVC 모델이 아닌 다른 모델을 이용하여 웹 응용을 개발할 수도 있고, 여러분 스스로 웹 응용의 골격을 만들어도 문제는 없다. 그러나 이렇게 개발하면 더 많은 노력이 필요하기 때문에 가능하면 이 도구를 사용하는 것이 좋다.

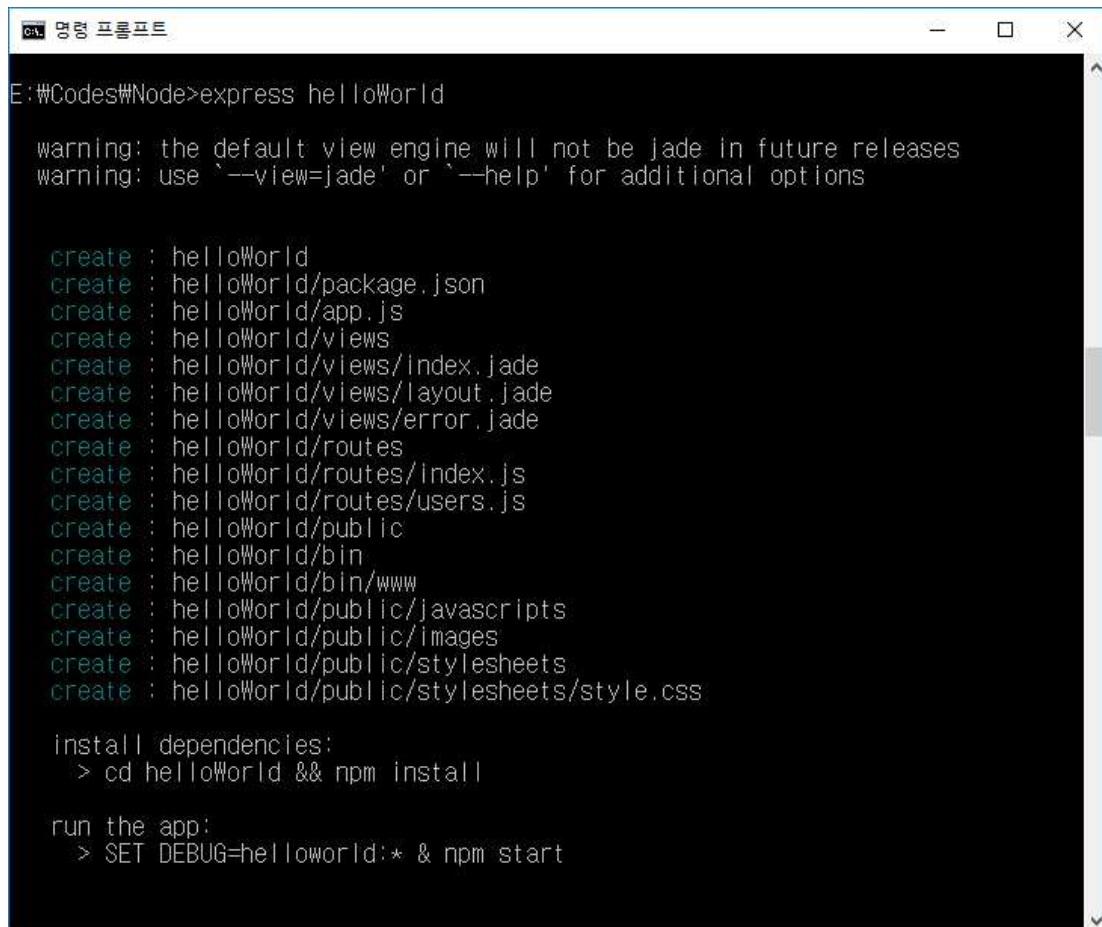
다음으로 express 명령을 이용하여 Express Application Generator를 실행한다.

3.3.2 Express 응용 프로그램 만들기

이제 Express를 이용하여 간단한 웹 서버를 구축해보자. 적당한 폴더로 이동하여 다음 Express Application Generator 명령을 사용하여 프로젝트의 골격을 만든다.

```
express helloWorld
```

이 명령이 수행되면 현재 폴더에 helloWorld라는 새로운 하위 폴더가 만들어지고, 그 폴더 내부에 다양한 하위 폴더와 파일들(특히 app.js, package.json 파일)이 생성된다.



```
E:\#Codes\#Node>express helloWorld

warning: the default view engine will not be jade in future releases
warning: use '--view=jade' or '--help' for additional options

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

install dependencies:
  > cd helloWorld && npm install

run the app:
  > SET DEBUG=helloworld:* & npm start
```

그림 12 Express Application Generator를 이용한 프로젝트 골격 생성

특히 마지막에 나오는 메시지에 주의하라. 프로젝트에 필요한 모듈들을 설치하기 위해서는 helloWorld 폴더로 이동한 후 “npm install”을 실행하라고 알려 주고 있다. 또한 프로젝트를 실행시키기 위해서는 “SET DEBUG=helloworld:* & npm start”를 입력하라고 알려주고 있다.

편집기를 열어서 package.json 파일을 살펴보면 이 프로젝트에서는 다양한 다른 모듈을 필요로 하는 것을 알 수 있다.

```
{  "name": "helloworld",
    "version": "0.0.0",
    "private": true,
    "scripts": {
      "start": "node ./bin/www"
    },
    "dependencies": {
      "body-parser": "~1.16.0",
      "cookie-parser": "~1.4.3",
      "debug": "~2.6.0",
      "express": "~4.14.1",
```

```
    "jade": "~1.11.0",
    "morgan": "~1.7.0",
    "serve-favicon": "~2.3.2"
  }
}
```

이 파일의 의존성 부분을 보면 body-parser, cookie-parser, debug, express 등 다양한 a 모듈을 사용하는 것을 볼 수 있다. 이 모듈들은 아직 우리 프로젝트에 설치되지 않았다. 필요한 모듈들을 한꺼번에 설치하려면 프로젝트 폴더로 가서 다음 명령을 실행한다.

```
npm install
```

이 명령의 실행 결과로 필요한 패키지들이 현재 폴더의 하위 폴더인 node_modules에 설치된다.

이제 프로젝트를 실행시키지 위해 다음 명령을 실행한다.

```
SET DEBUG=helloworld:* & npm start
```

실행 결과는 다음과 같다.

```
C:\Users\USER\Desktop\WebProgBook\Codes\Node\helloWorld>SET DEBUG=helloworld:* & npm start
start
> helloworld@0.0.0 start C:\Users\USER\Desktop\WebProgBook\Codes\Node\helloWorld
> node ./bin/www

helloworld:server Listening on port 3000 +0ms
```

이제 웹브라우저에서 <http://localhost:3000>으로 접속해보자. 접속 결과는 다음과 같다.

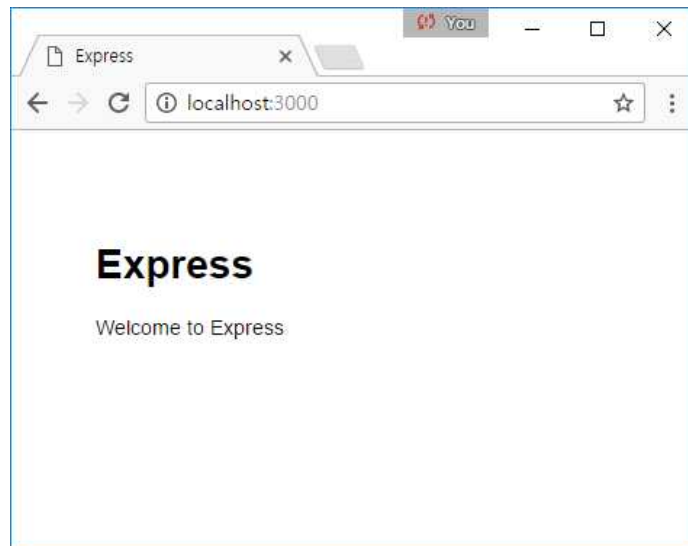


그림 14 Express 응용의 기본 화면

이제 URL 주소를 localhost:3000/users로 입력해 보자. 그러면 다음 페이지를 볼 수 있을 것이다.

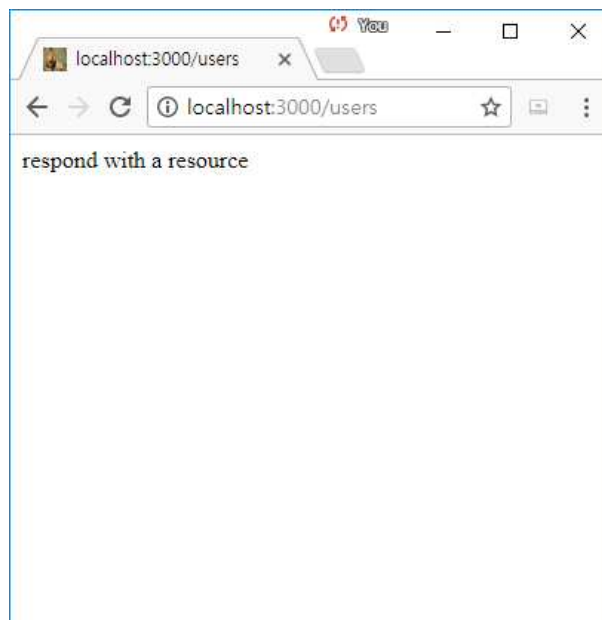


그림 15 Express 응용의 다른 화면

이 화면들은 어떻게 만들어지는 것일까? 다음 절에서 Express Application Generator가 설치한 파일들을 검토해보자.
서버를 중지시키려면 서버가 실행되고 있는 명령창에서 Ctr-C(컨트롤 키를 누른 채 c 키를 누른다)를 입력한다.

3.3.3 Express 코드 분석

프로그램 실행 코드

프로젝트를 실행시키기 위해 우리가 입력한 명령을 기억하는가? 다음 명령이었다.

```
SET DEBUG=helloworld:* & npm start
```

여기서 DEBUG를 지정하는 것은 요청 처리에 대한 정보를 서버의 콘솔 화면에 출력하기 위한 것이고 실제로 프로그램을 실행시키는 명령은 “npm start”이다. 그런데 앞 절에서 제시한 package.json 파일에 다음과 같은 부분이 있다.

```
“scripts”: {  
  “start”: “node ./bin/www”  
}
```

Npm은 우리가 입력한 start 단어를 보고 package.json 파일의 script 부분에 start로 정의된 속성을 찾아서 그 명령을 실행한다. 따라서 실제로 우리가 실행한 명령은 다음 명령이다.

```
node ./bin/www
```

따라서 우리 프로젝트의 실행 결과를 이해하려면 bin 폴더에 있는 www 파일을 보아야한다. 다음은 www 파일의 일부이다.

```
var app = require('../app');  
var debug = require('debug')('helloworld:server');  
var http = require('http');  
/**  
 * Get port from environment and store in Express.  
 */  
var port = normalizePort(process.env.PORT || '3000');  
app.set('port', port);  
/**  
 * Create HTTP server.  
 */  
var server = http.createServer(app);  
/**  
 * Listen on provided port, on all network interfaces.  
 */  
server.listen(port);
```

이 프로그램을 보면 우리가 Node를 이용해서 간단한 웹 서버를 구축한 것과 같은 일을 하고 있다. Http 모듈을 수입하고, 서버를 만들고, 특정 포트에서 대기하게 한다. 한 가지 다른 점은 서버를 만들 때 app라는 객체를 인수로 제공하고 있다는 점이다. 이 app 객체는 helloWorld 폴더에 있는 app.js라는 모듈이 수출하는 객체이다. 이상으로 대부분의 필요한 동작은 app.js에 있음을 알 수 있다.

다음은 app.js 파일 내용 중 일부이다.

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var index = require('./routes/index');
var users = require('./routes/users');
var app = express();
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
// 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('/', index);
app.use('/users', users);
...
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};
  // render the error page
  res.status(err.status || 500);
  res.render('error');
});
module.exports = app;
```

조금 긴 프로그램이지만 몇 가지 이해해야 할 부분이 있다. 먼저 이 프로그램에서 수출하는 객체는 app 객체이고(맨 마지막 줄에서 수출하고 있음), 이 객체는 express() 생성자를 이용해서 만들어진 객체이라는 점이다. 즉 express 응용 객체이다.

다음으로 나오는 것은 템플릿을 사용하여 동적으로 웹페이지를 만들어주는 엔진을 jade로 지정하고 이 엔진이 사용할 템플릿이 있는 폴더 위치를 지정하는 것이 나온다. 이때 사용하고 있는 __dirname 전역변수는 프로젝트 폴더의 위치를 가지고 있고, path.join() 메서드는 두 개의 부분 경로를 합쳐서 하나의 경로로 만든다. 따라서 이 프로그램에서는 템플릿들이 프로젝트 폴더 내 Views라는 폴더에 있음을 app에게 알려주고 있다.

미들웨어 사용

다음으로 다양한 app.use() 문장들이 나온다. 이 문장들은 미들웨어를 사용하겠다고 선언하는 문장들이다. 미들웨어는 들어온 요청에 대해 특정한 처리를 한 다음 응답을 보내거나 혹은 다

음 미들웨어로 처리를 넘기는 소프트웨어 모듈을 의미한다. 이 프로그램에서는 bodyParser, cookieParser 등의 미들웨어를 사용한다.

미들웨어 선언 후반에 다음 두 문장이 나온다.

```
app.use('/', index);
app.use('/users', users);
```

이 문장들은 라우터(router)를 미들웨어로 선언한 것이다. 라우터란 관련 있는 라우트(route) 들을 모아놓은 것을 말한다. 라우트는 사용자가 보낸 HTTP 동사와 경로에 대해 처리할 함수를 지정하는 Express 프로그램 코드를 말한다. 다음 예를 보자.

```
app.get('/', function(req, res) {
  res.send('Hello World');
});
```

이 라우트는 HTTP get 동사에 대해 경로로 /가 들어올 경우 처리하는 함수를 지정하고 있다. HTTP 동사는 “get”, “post”, “delete”, “header” 등 매우 많은 종류가 있다.

마지막 미들웨어는 오류 처리 미들웨어이다. 오류가 발생하면 이 미들웨어가 실행되는데 다음 문장을 이용하여 오류 페이지를 생성하고 있다.

```
res.render('error');
```

render() 메서드는 템플릿 엔진을 이용하여 HTML 페이지를 동적으로 생성한 후 이 HTML 파일을 클라이언트에게 전송하는 역할을 한다. 동적으로 HTML 페이지를 생성하는 것은 다음에 자세히 살펴보겠다.

라우터 사용

웹 사이트를 개발할 때 많은 수의 라우트를 이용하게 된다. 이러한 라우트 중 서로 관련이 있는 라우트들을 모아서 처리하는 프로그램을 라우터라고 한다. 우리의 app.js에서는 index와 users 두 개의 라우터를 사용하고 있다.

라우터들은 routes 폴더에 있다. 현재 이 폴더에는 index와 users 두 개의 라우터가 존재할 것이다.

그 중 index.js 라우터를 살펴보면 다음과 같다.

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

```
});  
module.exports = router;
```

프로그램의 앞부분에서 express의 Router 생성자를 사용하여 라우터 객체를 생성한다. 그 다음 각각의 라우트를 라우터에 추가한다. 위 라우터는 get 동사를 이용한 / 경로에 대한 요청을 처리하는 1개의 라우트만 가지고 있으나 일반적으로 라우터는 여러 개의 라우트를 가지게 된다.

app.js에서는 다음과 같이 /users로 시작하는 모든 요청 경로에 대해 이 라우터를 사용할 것을 지정하고 있다.

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

문서 템플릿 사용

routes 폴더에 있는 index.js 파일을 파일을 살펴보면 다음과 같은 부분이 있다.

```
res.render('index', { title: 'Express' });
```

앞에서도 잠깐 설명했듯이 res.render() 메서드는 템플릿을 이용하여 HTML 파일을 만든 후 클라이언트에게 전달하는 역할을 한다. 위에서 'index'는 사용할 템플릿 파일 이름을 말하고 {title: 'Express'}는 템플릿 엔진에 전달할 인수 객체이다.

이제 index 템플릿을 이용하여 HTML 페이지를 어떻게 작성하는지 보자. index 템플릿은 views 폴더 아래에 index.jade란 이름으로 저장되어 있다(jade 확장자는 우리가 템플릿 엔진으로 jade를 사용하기 때문이다. 만약 다른 템플릿 엔진을 사용한다면 확장자가 달라질 것이다).

내용은 다음과 같다.

```
extends layout  
block content  
  h1= title  
  p Welcome to #{title}
```

먼저 extends 문장이 나온다. 이 문장은 현재 페이지가 layout 페이지를 확장하는 것이라는 것을 알려준다. 다음으로 "block content"라는 문장이 나오는데 이 문장은 layout 페이지에서 "block content"라고 표시된 부분을 현재 문서의 "block content" 부분으로 교체하라는 의미이다.

jade 템플릿에서 보통 문장의 처음에 나오는 것은 HTML 태그를 의미한다. 위에서 h1, p는 각각 <h1>과 <p> 태그를 의미한다. 또한 태그와 태그 사이의 포함 관계는 들여쓰기로 표시한다. 위의 예에서 "block content"는 h1 요소와 p 요소를 자식으로 갖는다.

태그 이름 다음에 =이 나오는 경우는(h1=과 같이) = 다음에 나오는 식을 자바스크립트 식으로 계산하여 그 결과를 넣으라는 의미이다. 여기서는 title이라는 식을 사용했는데 이 식은 우리가 템플릿 엔진을 호출하면서 인수로 넘겨준 값이며 그 값은 현재 Express로 되어 있다.

태그 이름 다음에 =이 나오지 않으면 jade는 이것을 일반 문자열로 취급한다. 문자열 내에서 자바스크립트 식을 사용할 필요가 있을 경우 #{ } 표현을 사용한다. 위의 p에서 #{title}은 Express로 대체된다.

layout.jade의 내용은 다음과 같다.

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

이 템플릿에서는 HTML 문서의 기본 구조를 정의하고 있음을 알 수 있다. 문서 타입을 선언한 후 html 요소를 정의하고 있다. html 요소 자식으로 head와 body 요소가 있다. jade에서 어떤 요소의 속성을 표시하기 위해서는 link(rel='stylesheet')와 같이 괄호를 사용한다.

3.3.4 사용자 정보 관리를 위한 웹 응용 개발

웹 응용에서는 정보를 데이터베이스에 저장한 다음, 새로운 항목을 추가(Create)하거나 검색(Retrieve)하거나 수정(Update)하거나 삭제>Delete)하는 연산을 주로 하게 된다. 이러한 연산을 지원하는 응용 프로그램을 연산들의 앞 글자를 따서 CRUD 응용이라고 부른다.

본 절에서는 Express Application Generator가 생성한 골격 프로젝트를 사용자 정보를 관리하기 위한 CRUD 응용으로 개발해 보자.

개발하는 과정에서 우리는 모델(Model)-뷰(View)-컨트롤러(Controller) 패턴을 사용하려고 한다. 이 패턴은 약자로 MVC 패턴이라고 하는데 모델은 우리가 다루는 데이터의 형태를 정의하는 부분이다. 뷰는 데이터가 주어졌을 때 이 데이터를 어떻게 화면에 표시하는가를 담당하는 부분이고 마지막 컨트롤러는 데이터에 대한 요청을 처리하기 위해서 모델로부터 데이터를 가지고와서 뷰에서 넘겨주는 역할을 한다. MVC 패턴에서는 이 3가지 역할을 분리하여 전체 응용을 개발한다.

웹 응용 개발을 시작하면서 먼저 생각해야 하는 것은 어떤 정보를 우리가 다루어야 하는가이다. 우리 응용에서는 사용자 정보를 저장하고 처리하려고 한다. 사용자 정보는 단순히 이름, 휴대전화번호, 전자우편 주소로 구성된다고 가정한다. 각 사용자를 구별해 줄 아이디가 필요

한데 우리는 uid라는 필드를 사용할 것이다. uid는 문자 'u'와 해당 사용자가 생성된 시간(밀리 초로 측정한)이 합쳐진 스트링이다. 이를 위하여 먼저 프로젝트 폴더에 models 폴더를 새로 만든 다음 이 models 폴더에 다음과 같이 user.js를 작성한다.

```
function User(name, phone, email) {
  var now = new Date();
  this.uid = 'u' + now.getTime();
  this.name = name;
  this.phone = phone;
  this.email = email;
}
User.prototype.getUrl = function() {
  return '/users/info/' + this.uid;
}
module.exports = User;
```

다음으로 생각할 것은 사용자들에 대한 정보를 어떻게 저장할 것인가라는 문제이다. 물론 데이터베이스를 이용하여 저장하는 것이 편리하다. 그러나 데이터베이스 다루는 것이 본 절에서 공부하고자 하는 범위를 벗어나기 때문에 우리는 단순하게 데이터를 객체의 리스트로 저장할 것이다. 먼저 app.js가 있는 폴더에 database.js를 새로 만들고 다음과 같이 입력한다.

```
var User = require('./models/user')
var database = [];
database.push(new User('사용자1', '010-111-1111', 'user1@mycom'));
database.push(new User('사용자2', '010-222-2222', 'user2@myhome'));
module.exports = database;
```

이 코드에서 우리는 데이터베이스로 빈 리스트를 만든 후 데이터베이스에 2명의 사용자를 추가하였다.

모델 부분이 정리가 되었으면 컨트롤러 부분을 개발한다. 컨트롤러 부분은 요청이 들어오면 적절한 처리 코드를 연결해 주는 라우트와 실제 요청을 처리하여 응답을 생성하는 컨트롤러 부분으로 구성된다.

먼저 라우트 부분을 정리하자. routes 폴더에 있는 users.js를 다음과 같이 수정한다.

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

/* GET users listing. */
router.get('/', userController.userList);
router.get('/add', userController.addUser_get);
router.post('/add', userController.addUser_post);
router.get('/delete/:uid', userController.deleteUser);
router.get('/update/:uid', userController.updateUser);
```

```
router.get('/info/:uid', userController.userInfo);
module.exports = router;
```

이 라우터 모듈에서는 사용자 목록을 출력하는 것, 사용자 추가, 삭제, 수정에 대한 처리 모듈을 각각 지정하고 있다.

이제 처리 모듈을 만들어야 한다. 먼저 프로젝트 폴더에 controllers라는 폴더를 새로 만들고 그 내부에 userController.js 파일을 생성한 후 다음과 같이 입력한다. 각 처리 함수는 나중에 자세히 개발하기로 하고 일단 '아직 구현되지 않았음' 메시지를 전송하는 것으로 구현한다.

```
var database = require('../database');
var User = require('../models/user');

exports.userList = function(req, res) {
  res.send('userList: 아직 구현되지 않았음');
}

exports.addUser_get = function(req, res) {
  res.send('addUser_get: 아직 구현되지 않았음');
}

exports.addUser_post = function(req, res) {
  res.send('addUser_post: 아직 구현되지 않았음');
}

exports.deleteUser = function(req, res) {
  res.send('deleteUser: 아직 구현되지 않았음!');
}

exports.updateUser = function(req, res) {
  res.send('updateUser: 아직 구현되지 않았음!');
}

exports.userInfo = function(req, res) {
  res.send('uerInfor: 아직 구현되지 않았음!');
}
```

다음으로 이 라우터를 /users 경로에 사용하도록 app.js에서 미들웨어로 다음과 같이 지정해 준다.

```
app.use('/', index);
app.use('/users', users);
```

이제 다음 URL을 브라우저에서 입력해 보면 그림과 같이 아직 구현되지 않았다는 메시지가 출력될 것이다.

localhost:3000/users

localhost:3000/users/add

localhost:3000/users/delete/u123

localhost:3000/users/update/u123

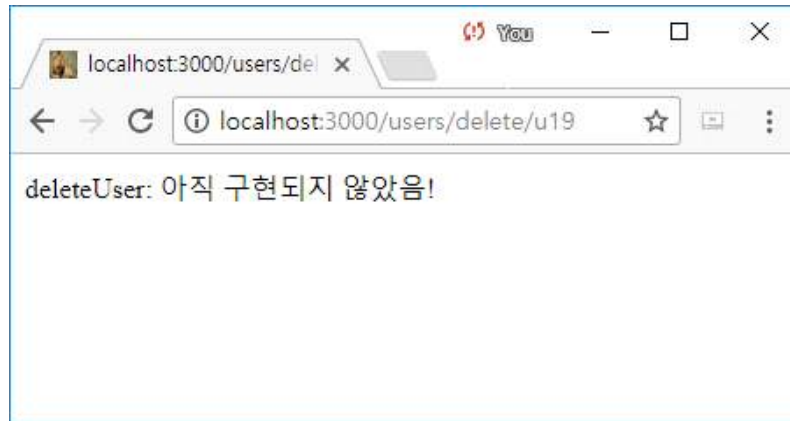


그림 16 URL 동작 점검

이제 우리의 홈 페이지를 변경해보자. 홈 페이지는 Views에 index.jade를 사용하여 출력되는데 이 템플릿 페이지를 호출하는 것은 routes에 있는 index.js 파일에서 호출하고 있다. 먼저 routes 폴더에 있는 index.js 파일을 다음과 같이 수정하자.

```
var express = require('express');
var database = require('../database');
var router = express.Router();
/* GET home page. Users 페이지로 redirect한다*/
router.get('/', function(req, res, next) {
  res.render('index', {title:'사용자 정보 관리 웹 사이트',
    numUsers:database.length});
});
module.exports = router;
```

다음으로 Views에 있는 index.jade 파일을 다음과 같이 수정한다.

```
extends layout
block content
  h1= title
  p 사용자 정보 관리 사이트에 오신 것을 환영합니다.
  if numUsers > 0
    p 현재 #{numUsers} 사용자가 등록되어 있습니다.
  else
    p 현재 등록된 사용자가 없습니다.
```

이 템플릿에서는 간단한 환영 인사를 출력한 후 현재 등록된 사용자 수에 대한 정보를 알려준다.

다음으로 다른 모델 템플릿의 기본이 되는 layout.jade를 다음과 같이 수정하자.

```

doctype html
html
  head
    title= title
    meta(charset='utf-8')
    meta(name="viewport", content="width=device-width, initial-scale=1")
    link(rel='stylesheet', href='/stylesheets/style.css')
    link(rel="stylesheet", href="https://maxcdn.bootstrapcdn.com/
      bootstrap/3.3.7/css/bootstrap.min.css")
    script(src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/
      jquery.min.js")
    script(src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/
      js/bootstrap.min.js")
  body
    div(class='container')
      div(class='row')
        div(class='col-sm-4')
          ul
            li
              a(href='/') 홈으로 가기
            li
              a(href='/users') 사용자 목록
            li
              a(href='/users/add') 사용자 추가
        div(class='col-sm-8')
          block content

```

위의 layout.jade 파일에서는 홈 페이지를 2단으로 표시하기 위하여 bootstrap이라는 라이브러리를 사용한다. bootstrap에서는 한 화면에 12개의 단을 표시하는데 어떤 div의 클래스를 col-sm-4로 지정하면 12 단 중 4 단을 합한 너비로 표시된다. 이 파일에서는 4 단 크기의 div 하나와 8단 크기의 div 하나를 사용하여 2단으로 표시하였으며, 왼쪽에는 네비게이션 메뉴를, 오른쪽에서 정보를 표시하였다. 브라우저 출력 결과는 다음과 같다.

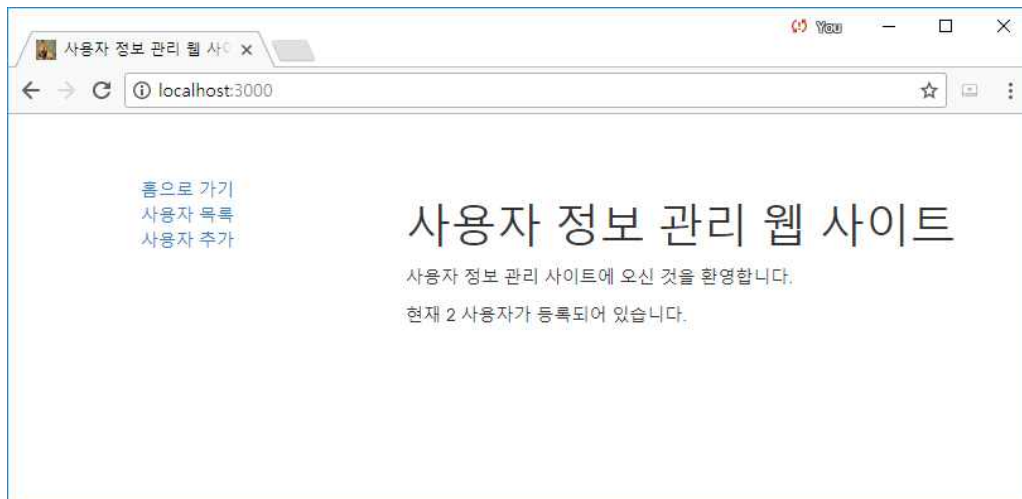


그림 17 수정된 홈페이지

이제 사용자 목록을 출력하기 위한 `UserController` 내부 함수를 다음과 같이 수정하자.

```
exports.userList = function(req, res) {
  res.render('userList', {title: '사용자 목록', users: database});
}
```

여기서는 `userList`라는 템플릿 파일을 이용하고 있는데 `userList.jade` 템플릿 파일을 `views` 폴더 아래에 만들고 다음과 같이 입력하자.

```
extends layout

block content
  h1= title
  hr
  ul
    each user in users
      li
        a(href=user.getUrl()) #{user.uid}
        | #{user.name}
    else
      li 사용자가 없습니다.
```

위에서는 `jade` 명령어 중 `each-in` 문장을 이용하여 각 사용자의 정보를 출력한다. `localhost:3000/users` URL로 접속하면 다음과 같이 사용자 목록이 표시됨을 알 수 있다. 만약 사용자가 어떤 `uid`를 클릭하다면 해당 사용자에 대한 자세한 정보를 표시해 준다.

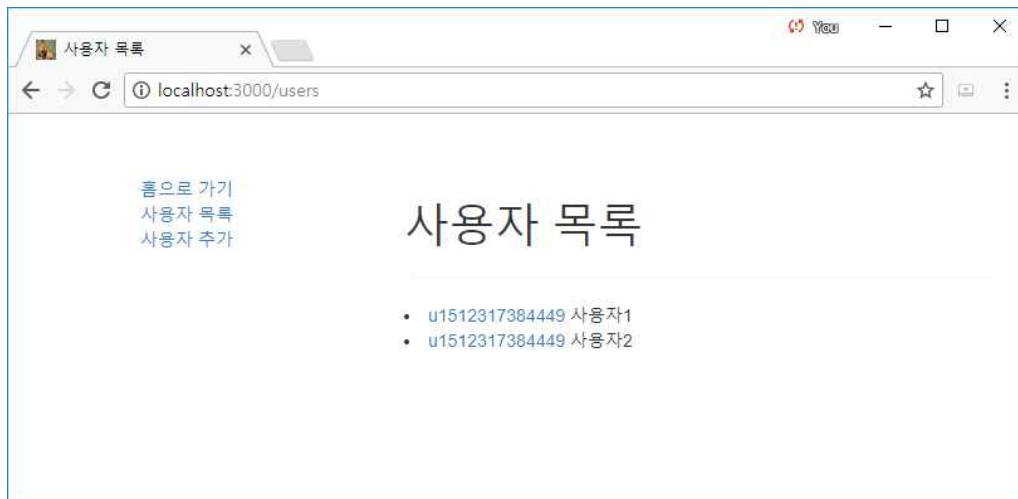


그림 18 사용자 목록 출력

이제 각 사용자의 세부 정보를 표시하는 컨트롤러와 템플릿을 작성해 보자. routes 폴더 아래에 있는 users.js 파일을 살펴보면 다음 부분이 있다.

```
router.get('/info/:uid', userController.userInfo);
```

이 문장에서는 라우트 매개변수를 사용하고 있다. 경로에 나오는 :uid가 바로 라우트 매개변수이다. 만약 사용자가 /info/u123을 경로로 입력한다면 위의 uid는 u123 값을 갖게 된다. 즉 라우트 매개 변수는 URL의 특정 위치에 나오는 값을 저장하는 변수이다. 이 변수는 request.param.uid로 프로그램에서 접근할 수 있다.

이제 userController에 userInfo 부분을 작성해 보자.

```
exports.userInfo = function(req, res) {
  var uid = req.params.uid;
  var i=0;
  // search for that user information.
  for(i=0; i<database.length; i++) {
    if(database[i].uid == uid) {
      break;
    }
  }
  res.render('showUserInfo', {title:'사용자 정보', user:database[i]});
}
```

사용자 정보를 표시하기 위한 showUserInfo.jade 파일을 다음과 같이 작성하자.

```
extends layout

block content
  h1= title
  table()
```

```
|  |  |
| --- | --- |
| td uid | td= user.uid |
| td name | td= user.name |
| td phone | td= user.phone |
| td emial | td= user.email |

```

이제 사용자 정보 표시 화면에서 uid를 클릭하면 다음과 같이 사용자에게 대한 상세 정보 페이지가 표시될 것이다.

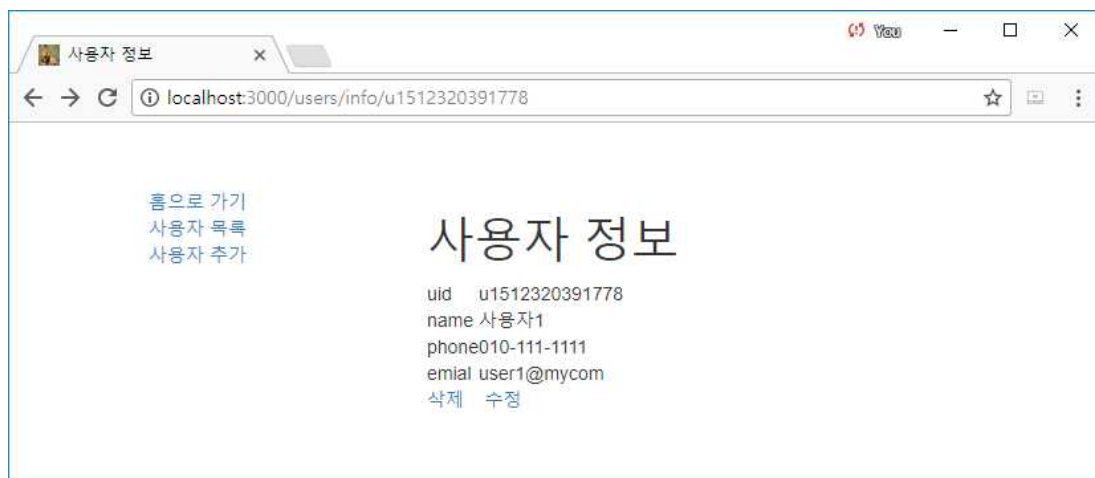


그림 19 사용자 상세 정보 화면

이제 사용자를 추가하기 위한 기능을 작성해보자. `UserController`에 있는 함수 중 사용자 추가와 관련이 있는 함수는 `addUser_get` 함수와 `addUser_post` 함수이다. 사용자가 웹 브라우저에서 '사용자 추가' 링크를 클릭하면 처음에는 아무 것도 채워지지 않은 사용자 등록 폼을 출력하여야 한다. 그러나 사용자가 필드 값을 잘못 입력하였거나 필수 필드 값을 누락해서 다시 표시할 경우는 지금까지 사용자가 입력한 정보를 표시하고 오류 정보도 함께 표시해야 한다.

사용자가 입력한 폼 내부 필드들을 검사하기 위해서는 `Express Validator`를 사용하면 편리하다. 먼저 우리 프로젝트에 다음 명령으로 설치하자.

```
npm install express-validator --save
```

다음으로 app.js에 express-validator를 미들웨어로 다음과 같이 등록하자. 이때 body-parser가 동작한 이후에 validator가 동작하므로 그 밑에 두는 것이 타당하다.

```
var expressValidator = require('express-validator');
...
app.use(bodyParser.urlencoded({ extended: false }));
...
app.use(expressValidator());
```

이제 userControll.js 내부에 addUser_get 부분부터 다음과 같이 수정하자.

```
exports.addUser_get = function(req, res) {
  res.render('getUserInfo', {title:'사용자 추가'});
}
```

이제 getUserInfo.jade 파일을 views 폴더 아래에 만들고 다음과 같이 입력하자.

```
extends layout

block content
  h1= title
  hr
  form(action='',method='post')
    label(for='name') 이름:
    input(id='name',name='name',type='text',
      value=(user==undefined)? '':user.name)
    br
    label(for='phone') 휴대폰:
    input(id='phone',name='phone',type='text',
      value=(user==undefined)? '':user.phone)
    br
    label(for='email') 이메일:
    input(id='email',name='email',type='text',
      value=(user==undefined)? '':user.email)
    br
    button(type='submit') 제출
  if errors
    h2 Errors!
    ul
      each error in errors
        li= error.msg
```

이제 홈페이지 화면에서 “사용자 추가” 링크를 누르면 다음과 같이 사용자 정보를 입력할 수 있는 페이지가 표시된다.



그림 20 사용자 추가 화면

사용자가 정보를 입력하고 제출 버튼을 누르면 어떻게 되는가? 먼저 이 요청은 어떤 URL로 전송되는가? 위의 `getUserInfo.jade` 코드에서 form이 나오는데 이 form의 action은 빈 스트링이다. form의 액션이 빈 스트링일 경우 요청은 현재 페이지로 전달된다. 다만 우리가 method를 post 방식으로 주었기 때문에 `/users/add` URL로 post 방식으로 정보가 전달된다.

우리의 라우터에서 post 방식으로 `/users/add`에 접근할 때 처리하는 함수는 `addUser_post` 함수이다. 이제 이 함수를 완성할 차례이다.

```
exports.addUser_post = function(req, res) {
  console.log('Adding a user by post: ');
  req.checkBody('name', 'Invalid Name').notEmpty();
  req.checkBody('phone', 'Invalid phone').notEmpty();
  req.checkBody('email', 'Invalid email').notEmpty();

  req.sanitize('name').escape().trim();
  req.sanitize('phone').escape().trim();
  req.sanitize('email').escape().trim();

  console.log(req.body.name + ':' + req.body.phone);
  var errors = req.validationErrors();

  var user = new User(req.body.name, req.body.phone, req.body.email);

  if(errors) {
    res.render('getUserInfo', {title: '사용자 추가', user: user,
    errors: errors});
  }
  else {
```

```

        database.push(user);
        res.redirect('/users');
    }
}

```

위의 코드에서 몇 가지 주목할 점은 다음과 같다. 우선 req.checkBody라는 메서드를 사용하여 폼에서 필수 필드가 빠졌는지 검사하도록 하였다. 우리는 모든 필드를 검사하도록 지정하였다. 두 번째는 사용자가 폼 입력 칸에 악성 자바스크립트 코드를 입력할 수 있다. 이런 경우 서버에 악영향을 줄 수 있으므로 사용자가 입력한 값들을 소독(sanitize)한다. 사용자가 폼의 입력에 넣은 값은 예를 들면 name='phone'에 넣은 값은 req.body.phone으로 접근할 수 있다. 이를 위해서 먼저 body-parser 미들웨어가 실행되어야 한다.

위의 코드에서는 req.validationError() 메서드를 이용하여 사용자 입력에 오류가 있는지를 검사하고 만약 오류가 있다면 오류 객체를 현재 페이지로 보내서 출력하도록 한다.

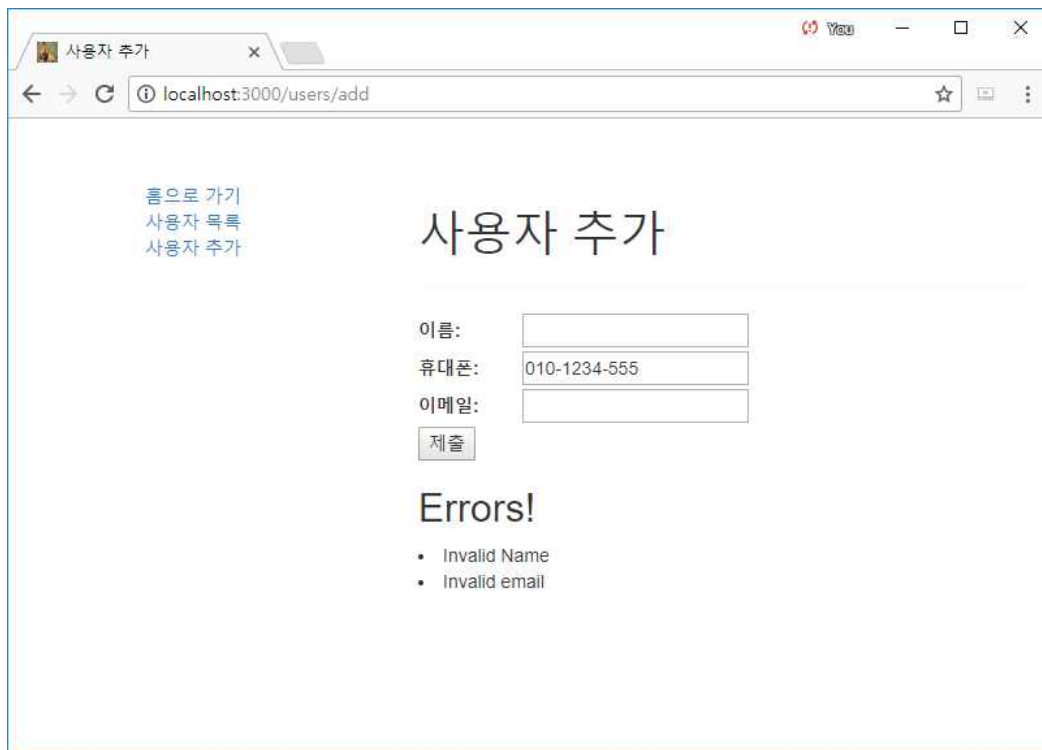


그림 21 사용자 입력에 오류가 있는 경우

만약 오류가 없는 경우 사용자를 새로 만들어서 database에 추가한 후 페이지를 res.redirect() 메서드를 이용하여 '사용자 목록' 화면으로 이동한다.

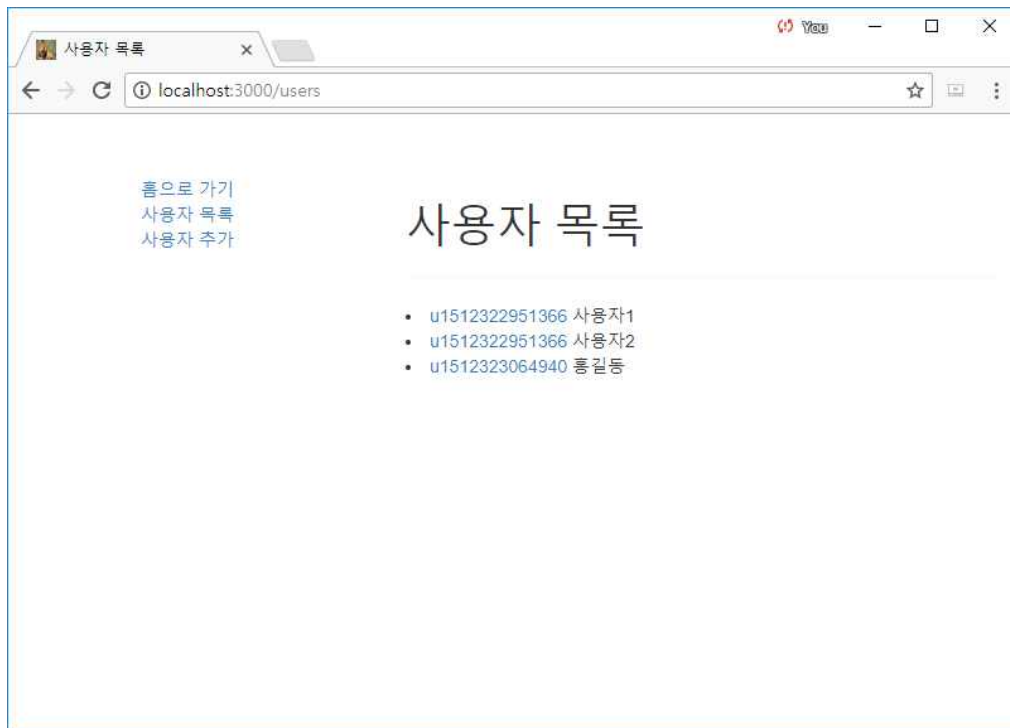


그림 22 새로운 사용자가 문제없이 추가된 화면

사용자 삭제와 사용자 수정은 지금까지 배운 내용을 토대로 스스로 작성해 보기를 바란다.