라우터 미들웨어 사용하기

번거롭게 요청 url을 일일이 확인해야 하는 문제를 해결하기 위한 것이 **라우터 미들웨어**(router middleware)입니다. router도 미들웨어 중의 하나이지만 app 객체의 속성으로 제공되며 사용자가 요청한 기능이 무엇인지 패스를 기준으로 구별해 주기 때문에 아주 중요한 역할을 담당합니다.

라우터 미들웨어는 익스프레스에 미리 등록되어 있어 바로 사용할 수 있습니다.

라우터 미들웨어를 사용하면 다음과 같은 메소드가 app 객체에 추가됩니다.

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

[표] 라우터 미들웨어를 사용하면 app 객체에 추가되는 메소드

클라이언트에서 특정 패스로 요청할 경우 GET 방식으로 요청할 때는 익스프레스에서 get( ) 메소드를 사용해 함수를 등록해야 하며, POST 방식으로 요청할 때는 post( ) 메소드를 사용해 등록해야 합니다. 라우팅 기능을 확인해 보기 위해 먼저 앞에서 만든 [public] 폴더 안의 login.html 파일을 복사하여 login2.html을 만든 후 〈form〉 태그에 action 속성을 추가합니다.

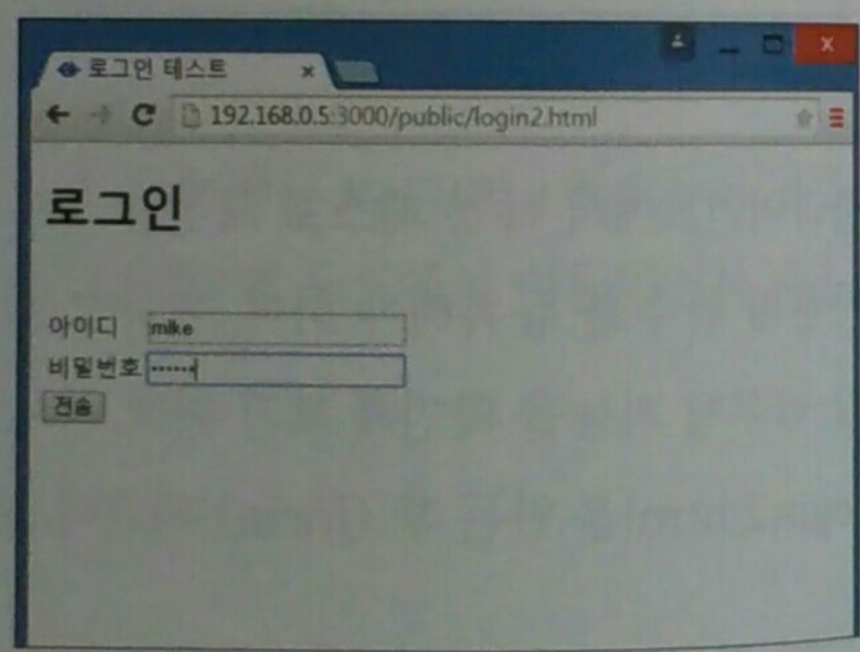
참조 파일 | ExpressExample>/public/login2.html

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

app7.js 파일을 복사해서 만든 app8.js 파일에는 이제 더 이상 use( ) 메소드로 함수를 추가하지 않겠습니다. 그 대신 post( ) 메소드를 호출하면 등록한 함수가 /process/login 요청 패스를 처리하도록 만듭니다.

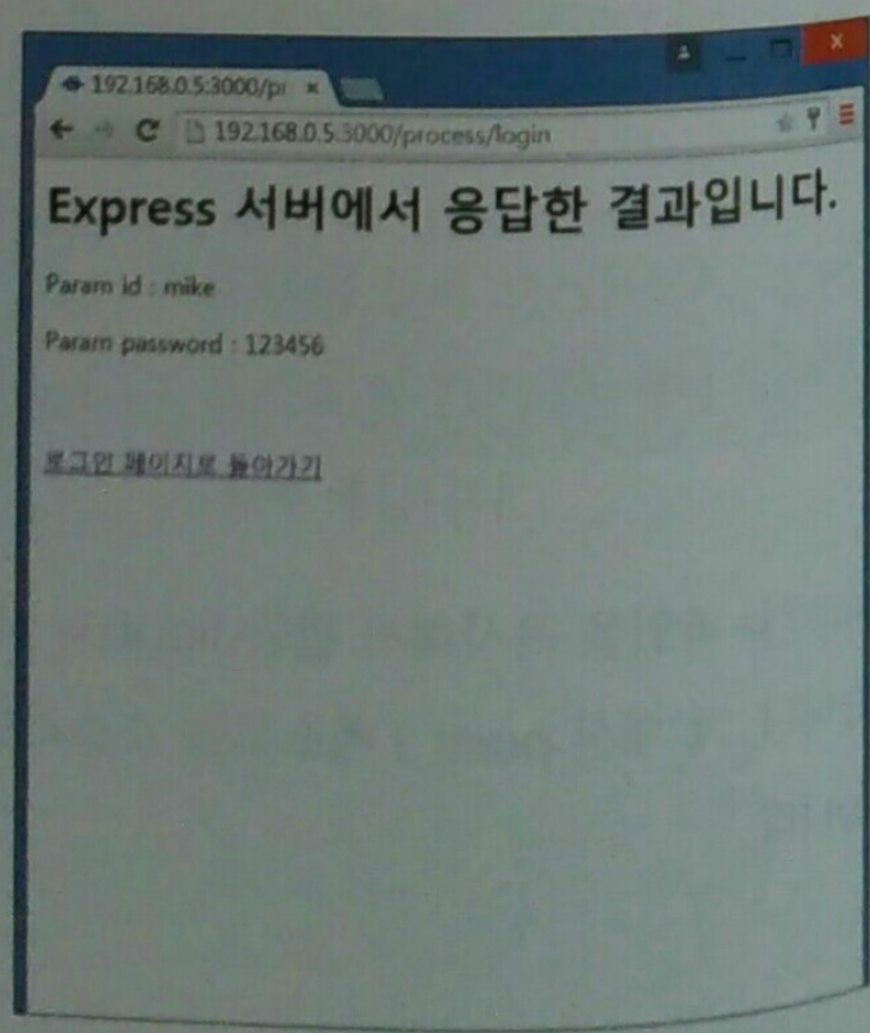
```
……

app.post('/process/login', function(req, res) {
  console.log('/process/login 처리함.');

  var paramId = req.param('id');
  var paramPassword = req.param('password');

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

……
```

로그인 페이지에서 [전송] 버튼을 누르면 /process/login 패스로 요청하므로 post( ) 메소드로 등록한 콜백 함수가 호출됩니다. app8.js 파일을 실행한 후 웹 브라우저에서 [public] 폴더 아래에 있는 login2.html 파일을 열고 아이디와 비밀번호를 입력합니다.



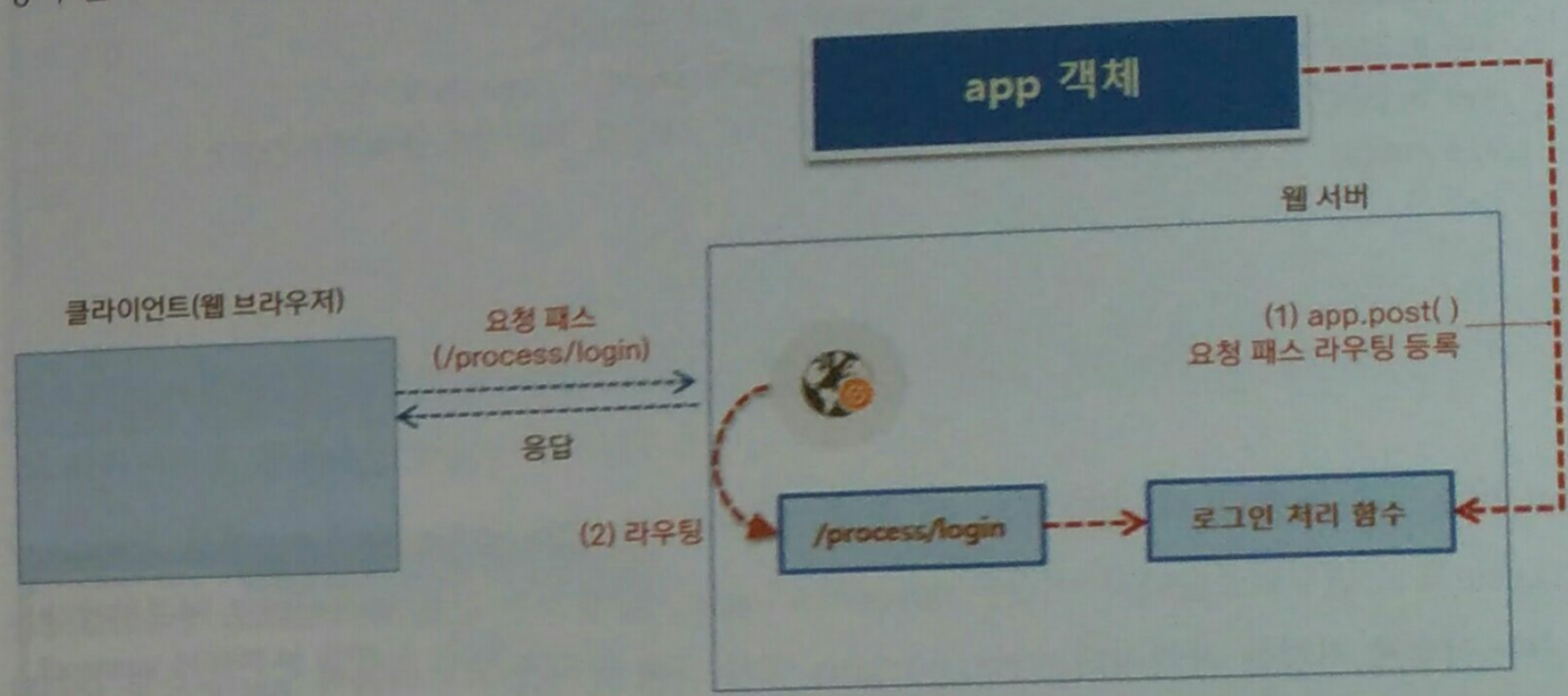
login2.html 파일에서 아이디와 비밀번호 입력 ▶

[전송] 버튼을 누르면 서버에서 처리한 후 다음 응답 페이지가 표시됩니다.



post( ) 메소드로 추가한 콜백 함수 처리 결과 ▶

다음은 클라이언트가 요청한 패스인 /process/login을 라우팅하는 과정을 보여 줍니다. 라우터 미들웨어를 등록하면 app 객체에서 get( ) 또는 post( ) 메소드를 호출할 수 있으므로 먼저 app.post( ) 메소드를 호출하여 요청 패스를 라우팅하도록 등록합니다. 그다음 클라이언트에서 /process/login으로 요청이 들어오면 이 라우팅 정보에 따라 해당 콜백 함수가 실행됩니다.



▲ 요청 패스 /process/login을 라우팅하는 과정

콜백 함수에서는 로그인에 필요한 기능을 실행한 후 클라이언트로 응답을 보내 줄 수 있습니다.

URL 파라미터 사용하기

URL 뒤에 ? 기호를 붙이면 필요에 따라 요청 파라미터(query string)를 추가하여 보낼 수 있습니다. 이런 요청 파라미터는 서버에 데이터를 전달하기 위해 사용되는데, 요청 파라미터 대신 URL 파라미터를 사용할 수도 있습니다. URL 파라미터는 요청 파라미터와 달리 URL 주소의 일부로 들어갑니다. URL 파라미터를 어떻게 전달하는지 알아보기 위해 app8.js 파일을 복사하여 app8_02.js 파일을 만듭니다. 그런 다음 app.post 메소드 호출 부분을 다음과 같이 수정합니다.

참조 파일 | ExpressExample>app8_02.js

```
……

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

        var paramName = req.params.name;

        var paramId = req.param('id');
```

```
        var paramPassword = req.param('password');

        res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
        res.write('<h1>Express 서버에서 응답한 결과입니다.</h1>');
        res.write('<div><p>Param name : ' + paramName + '</p></div>');
        res.write('<div><p>Param id : ' + paramId + '</p></div>');
        res.write('<div><p>Param password : ' + paramPassword + '</p></div>');
        res.write("<br><br><a href='/public/login3.html'>로그인 페이지로 돌아가기</a>");
        res.end();
});

......
```

app.post 메소드를 호출할 때 전달하는 첫 번째 파라미터의 값이 /process/login에서 /process/login/:name으로 변경되었습니다. 이것은 /process/login/ 뒤에 오는 값을 파라미터로 처리하겠다는 의미입니다. 이렇게 지정한 파라미터는 req.params 객체 안에 들어갑니다. 따라서 :name으로 표시된 부분에 넣어 전달된 값은 req.params.name 속성으로 접근할 수 있습니다.

이렇게 /process/login/:name 형태를 가진 URL을 처리하도록 변경했으므로 사용자가 웹 페이지에서 요청할 때 사용하는 URL도 변경해야 합니다. [public] 폴더 안에 있는 login2.html 파일을 복사하여 login3.html 파일을 만든 후 다음과 같이 요청 URL을 수정합니다.

참조 파일 | ExpressExample>/public/login3.html

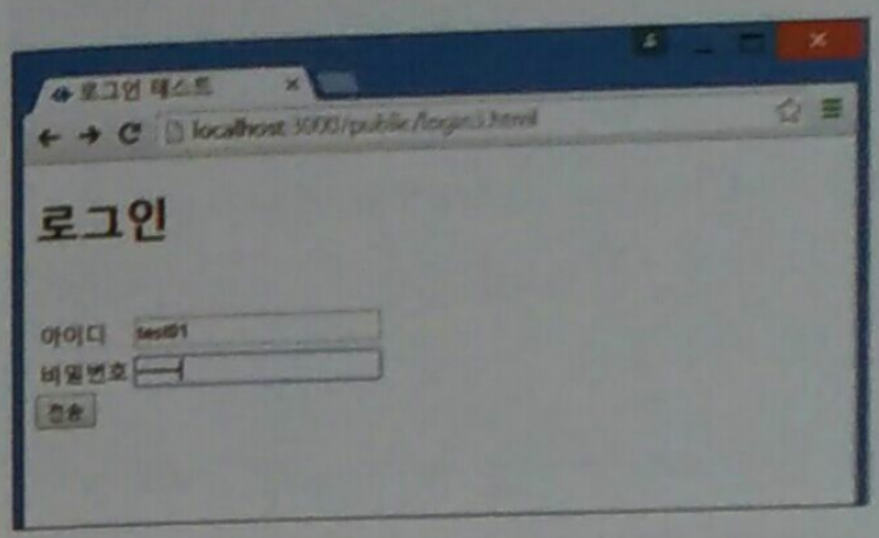
```
......

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

......
```

action 속성 값으로 /process/login/mike를 넣었으므로 mike라는 문자열이 URL 파라미터로 전달됩니다.

```
/process/login/mike
/process/login/:name
```

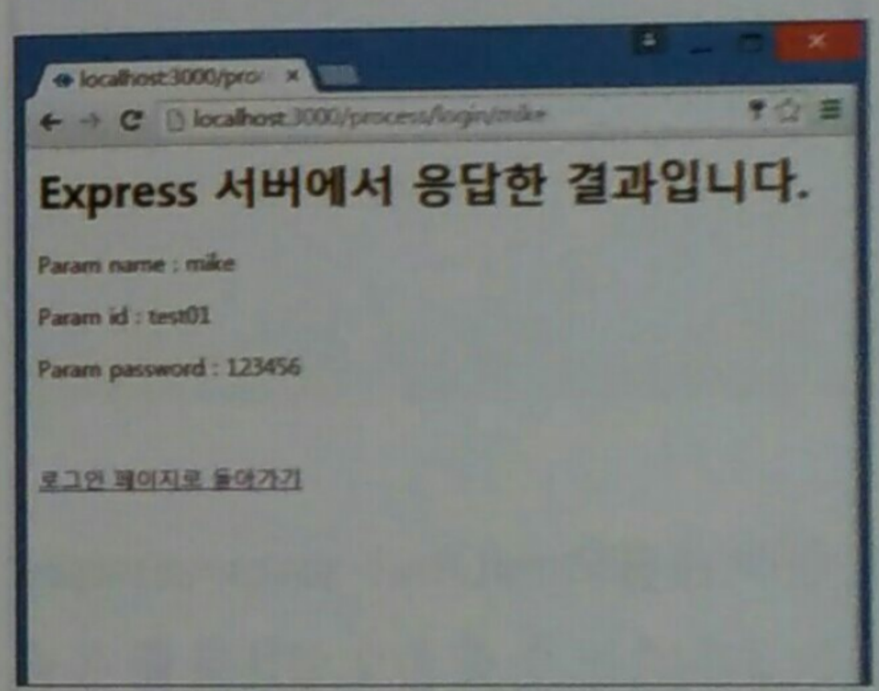
위에서 빨간색으로 표시한 부분이 서로 매칭되어 처리됩니다. 정상적으로 처리되는지 확인하기 위해 app8_02.js 파일을 실행한 후 웹 브라우저에서 login3.html 파일을 엽니다.

▶ http://localhost:3000/public/login3.html



◀ URL 파라미터를 전달하기 위한 웹 페이지

아이디와 비밀번호를 입력한 후 [전송] 버튼을 누르면 다음과 같이 아이디와 비밀번호 외에 mike라는 값도 파라미터로 전달받은 것을 알 수 있습니다.



◀ 서버에서 URL 파라미터를 전달받아 보여 준 결과

클라이언트가 보낸 URL에 포함된 문자열 중 일부를 서버에서 파라미터로 처리할 수 있다는 것을 알게 되었습니다. URL 파라미터를 사용해 데이터를 전달하는 경우가 자주 있으므로 이 방식에 대해 잘 기억해 두세요.

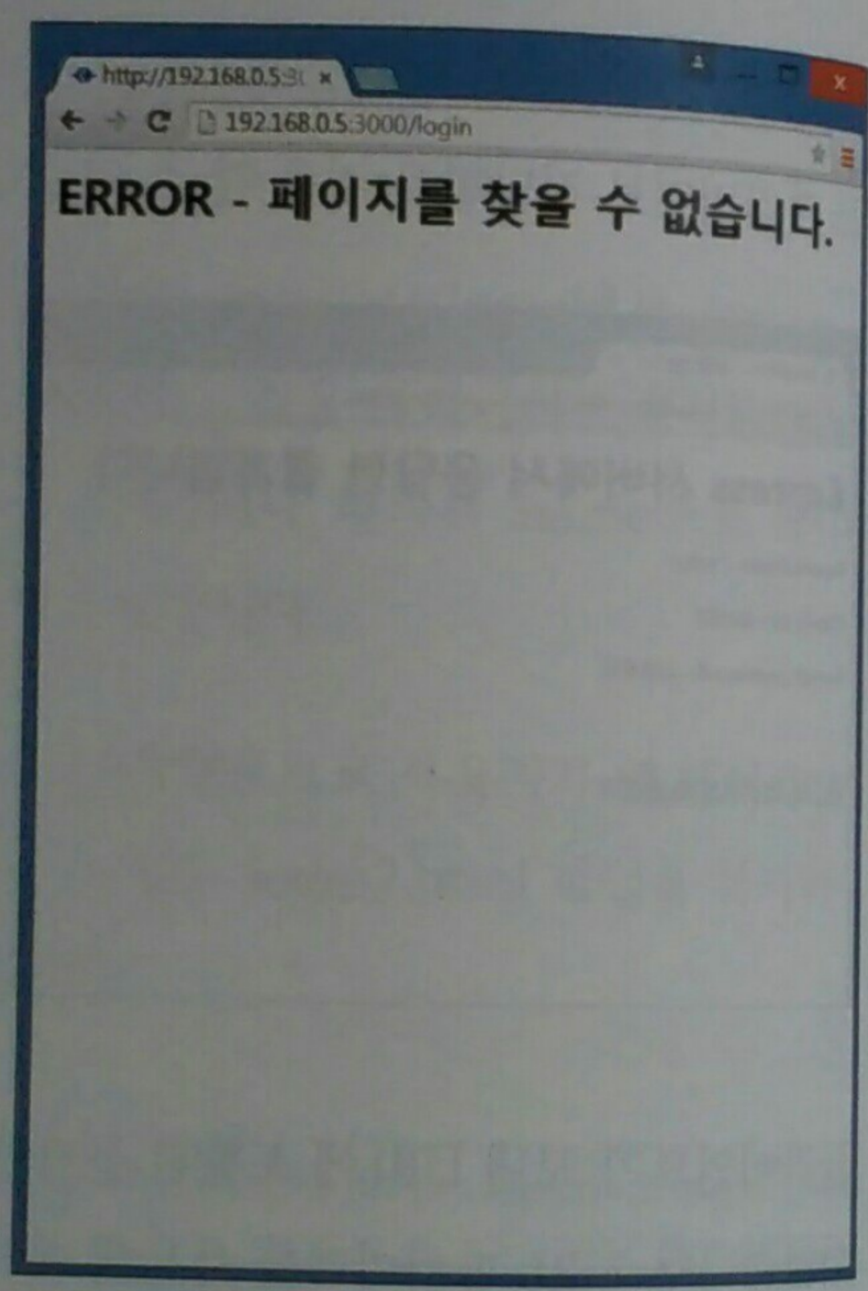
오류 페이지 보여 주기

로그인을 위해 웹 브라우저에 주소를 입력할 때 주소가 입력되는 부분을 보면 /process/login 패스로 처리된 것을 알 수 있습니다. 이 패스는 웹 서버에서 라우터 미들웨어로 등록했기 때문에 등록된 콜백 함수에서 요청을 전달받아 처리할 수 있습니다. 따라서 웹 서버에 등록되지 않은 패스인 /login과 같은 패스를 웹 브라우저에 입력해 보면 문서를 찾을 수 없다는 디폴트 메시지가 나타납니다. 그런데 문서를 찾을 수 없다는 메시지를 우리가 직접 만든 페이지로 바꾸고 싶다면 어떻게 할까요? 지정한 패스 이외의 모든 패스로 요청이 들어왔을 때 오류 페이지가 보이도록 처리해 주어야 합니다.

라우터 미들웨어는 특정 패스가 등록되어 있는지 순서대로 확인하여 처리합니다. 따라서 위에서 추가한 post( ) 메소드 아래쪽에 다음과 같이 all( ) 메소드 호출 부분을 추가합니다.

```
……

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

……
```

이제 다시 웹 브라우저에서 /login 패스를 입력하면 서버에서 전송한 오류 페이지가 화면에 표시됩니다.



all( ) 메소드를 사용해 '페이지를 찾을 수 없다'는 메시지를 보여 준 경우 ▶

이와 같이 페이지를 찾을 수 없다는 오류 메시지를 응답으로 보내는 코드를 직접 만드는 대신에 다른 사람이 미리 만들어 둔 미들웨어를 사용할 수도 있습니다.

express-error-handler 미들웨어로 오류 페이지 보내기

다음은 express-error-handler 미들웨어를 사용해 404.html 페이지를 응답으로 보내 주는 예제입니다. app8.js 파일을 복사하여 app9.js 파일로 만든 후 파일을 수정합니다.

참조 파일 | ExpressExample>app9.js

```
……

// 오류 핸들러 모듈 사용
var expressErrorHandler = require('express-error-handler');

……

// 모든 router 처리 끝난 후 404 오류 페이지 처리
var errorHandler = expressErrorHandler({
  static: {
    '404': './public/404.html'
  }
});

app.use( expressErrorHandler.httpError(404) );
app.use( errorHandler );

……
```

express-error-handler 모듈은 특정 오류 코드에 따라 클라이언트로 응답을 보내 줄 때 미리 만들어 놓은 웹 문서를 보내 줄 수 있습니다. 이 모듈은 외장 모듈이므로 코드의 위쪽에서 먼저 require( ) 메소드를 호출하여 모듈을 불러들입니다. 오류 페이지는 모든 라우터 처리가 끝난 후 처리되어야 합니다. 따라서 서버를 시작하기 위해 호출하는 코드 위쪽에 미들웨어로 추가합니다. 오류 페이지를 지정할 때는 미리 만들어 둔 파일의 위치를 지정할 수 있으므로 ./public/404.html로 지정합니다. 이제 프로젝트 폴더 안에 있는 [public] 폴더 안에 404.html 파일을 만들고 그 안에 오류 표시를 위한 코드를 입력합니다.

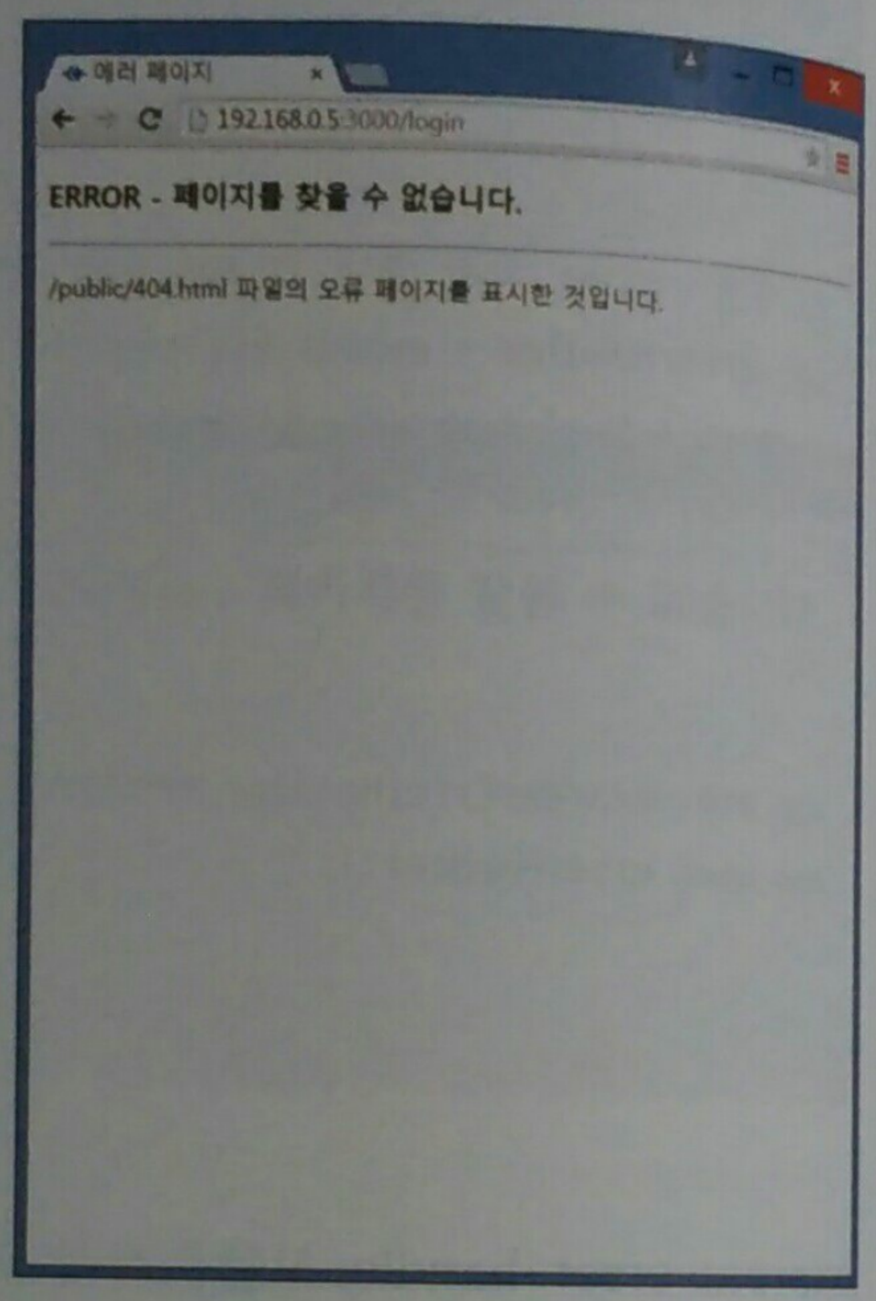
참조 파일 | ExpressExample>/public/404.html

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

express-error-handler 모듈은 외장 모듈이므로 먼저 설치가 필요합니다. 명령 프롬프트 창을 열고 다음 명령을 입력하여 설치합니다.

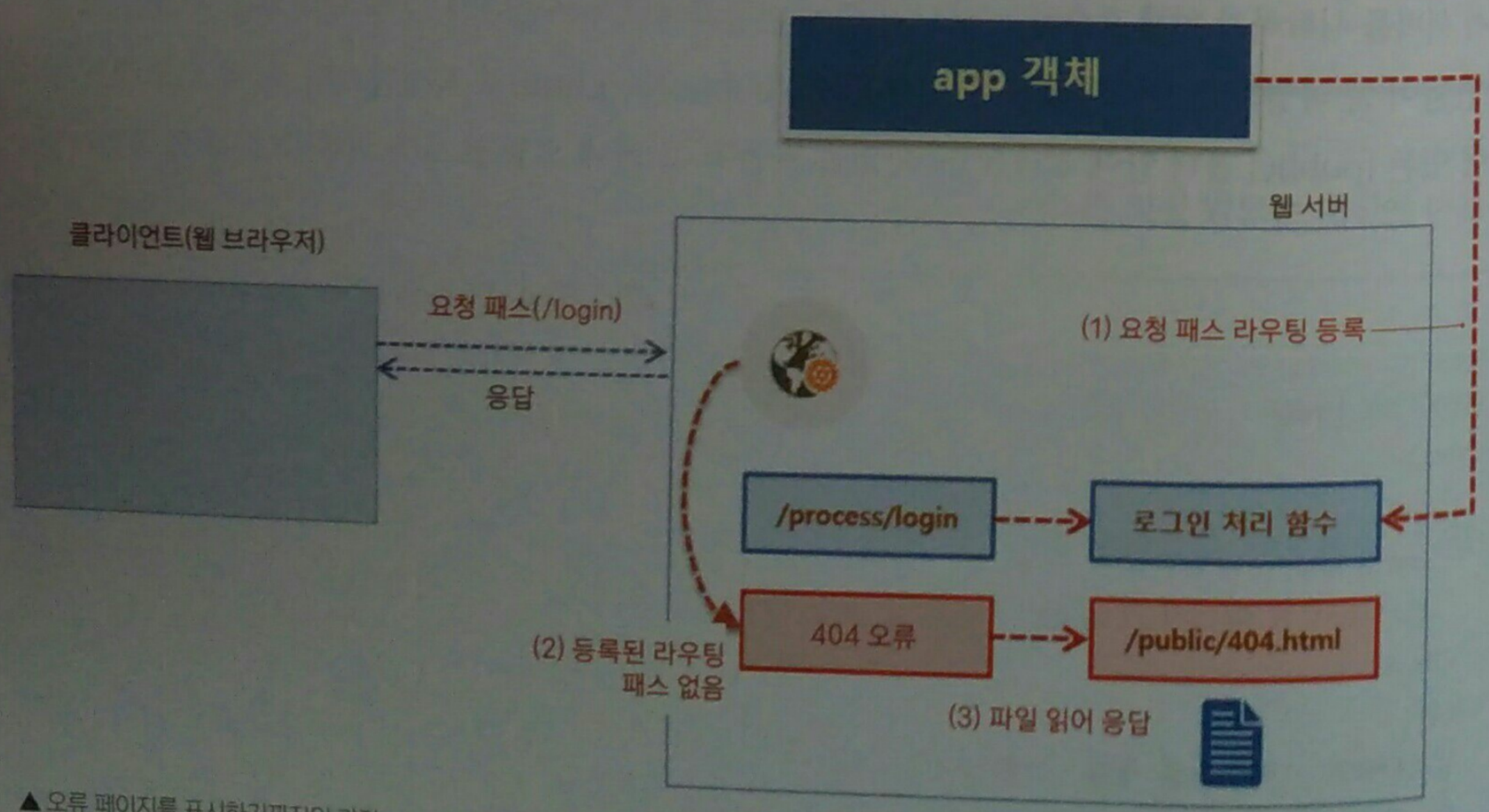
```
% npm install express-error-handler --save
```

서버를 실행한 후 웹 브라우저에서 아무 주소나 입력하면 다음과 같이 페이지를 찾을 수 없다는 메시지가 나타납니다.



오류 처리 미들웨어로 표시한 오류 페이지 ▶

문서를 찾을 수 없다는 오류 페이지를 표시하기까지의 과정은 다음과 같습니다.



▲ 오류 페이지를 표시하기까지의 과정

웹 브라우저에서 요청한 패스가 라우터에 등록한 함수 중에 없다면 404 오류가 발생합니다. 이 오류가 발생하면 미리 만들어 둔 404.html 파일을 읽어 응답으로 보냅니다.

지금까지 라우팅 미들웨어를 사용하는 방법에 대해 알아보았습니다. 사용자가 요청하는 패스는 매우 다양합니다. 그래서 각각의 요청을 담당하는 함수를 각각 만들 수 있는 라우팅 기능은 매우 편리하면서도 중요한 기능입니다.

토큰과 함께 요청한 정보 처리하기

라우터 미들웨어는 유용하게 사용할 수 있는 또 다른 기능을 포함하고 있습니다. 바로 요청 URL에 포함된 파라미터, 즉 **토큰(Token)**을 사용할 수 있는 기능입니다. 그러면 요청 패스에 토큰을 추가하여 보내 온 요청을 처리하도록 만들어 보겠습니다. app9.js 파일을 복사하여 app10.js 파일을 만들고 다음과 같이 입력합니다.

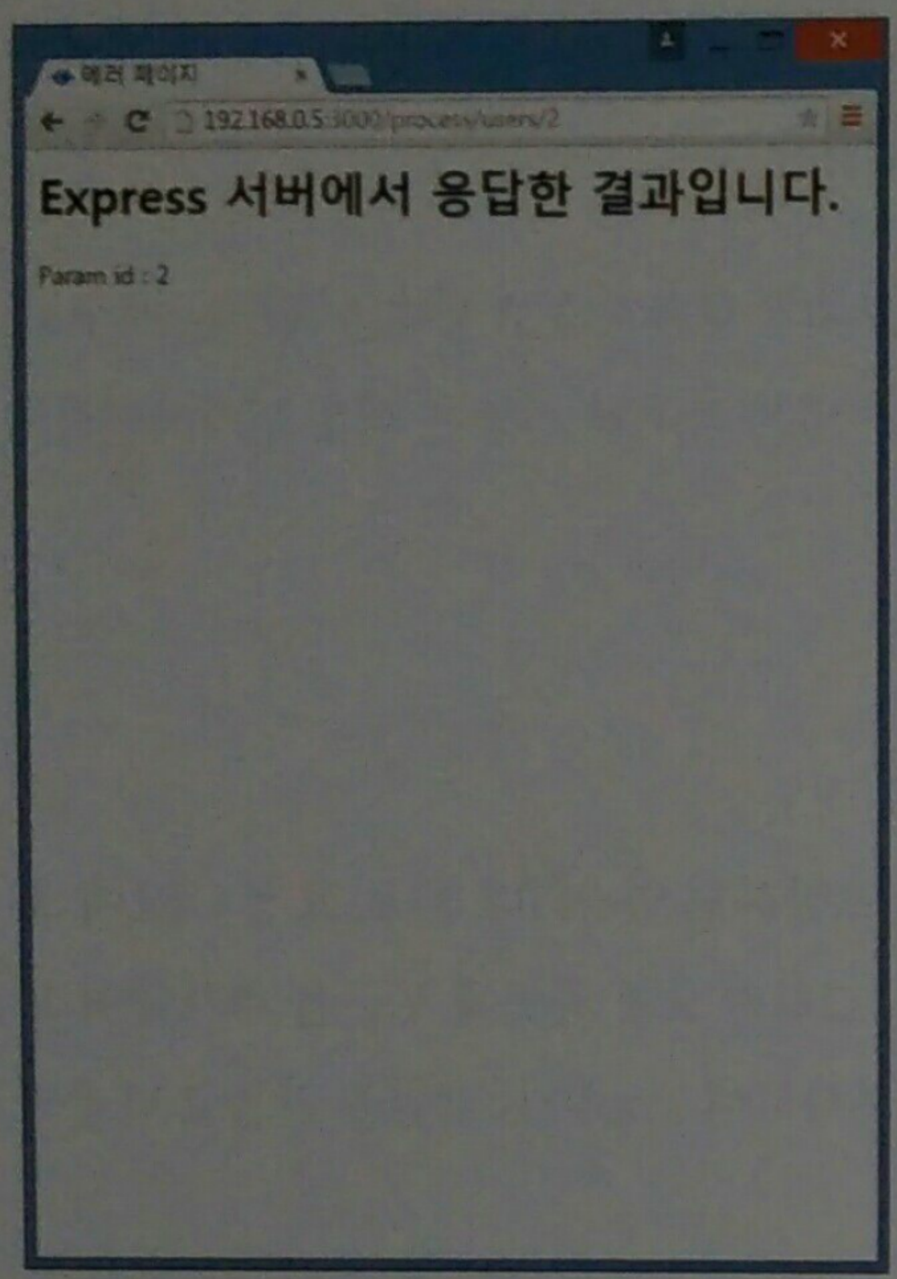
참조 파일 | ExpressExample>app10.js

```
.....
app.get('/process/users/:id', function(req, res) {
  // 토큰 정보를 가져옴
  var paramId = req.params.id;

  console.log('/process/users와 토큰 %s를 사용해 처리함.', paramId);

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

.....
```

get( ) 메소드를 호출하면서 동시에 /process/users/:id 패스를 처리하도록 코드를 입력했습니다. 여기에서 콜론(:)을 붙인 id 값이 토큰이며, 일반적인 요청 파라미터처럼 파라미터 객체의 속성으로 확인할 수 있습니다. 따라서 req.params.id 코드를 사용하면 id 속성에 접근할 수 있습니다.

이 파일을 실행하고 웹 브라우저에 다음 주소를 입력한 후 조회하면 다음 결과를 볼 수 있습니다.

▶ http://localhost:3000/process/users/2



▲ 토큰을 포함하여 GET 요청을 했을 때 보여 주는 응답 페이지

이렇게 토큰을 사용하면 사용자 리스트 중에서 특정 사용자 정보를 id 값으로 조회하기에 편리합니다.

05-5 | 쿠키와 세션 관리하기

사용자가 로그인한 상태인지 아닌지 확인하고 싶을 때에는 **쿠키**나 **세션**을 사용합니다. 쿠키는 클라이언트 웹 브라우저에 저장되는 정보이며, 세션은 웹 서버에 저장되는 정보입니다. 여기에서는 이 쿠키와 세션을 어떻게 다룰 수 있는지 알아보겠습니다.

쿠키 처리하기

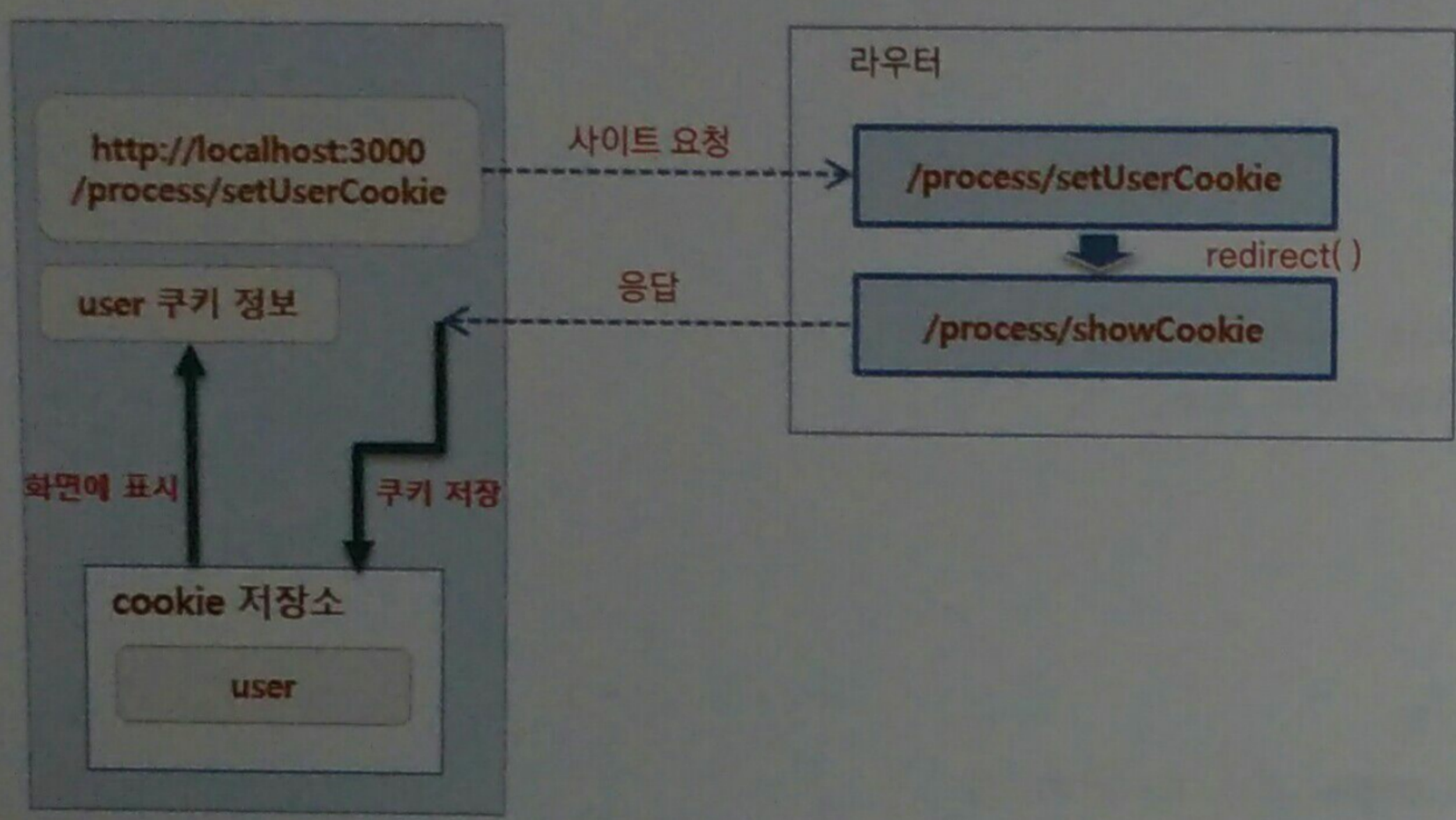
쿠키는 클라이언트 웹 브라우저에 저장되는 정보로서 일정 기간 동안 저장하고 싶을 때 사용합니다. 익스프레스에서는 **cookie-parser 미들웨어**를 사용하면 쿠키를 설정하거나 확인할 수 있습니다. 다음과 같이 use( ) 메소드를 사용해 cookie-parser 미들웨어를 사용하도록 만들면 요청 객체에 cookies 속성이 추가됩니다. 앞에서 만들어 둔 app10.js 자바스크립트 파일을 복사하여 새로운 app11.js 파일을 만들고 다음과 같이 일부분을 수정합니다.

```
……,

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

……,

app.use(express.cookieParser());

……,

// 쿠키 정보를 확인함
app.get('/process/showCookie', function(req, res) {
  console.log('/process/showCookie 호출됨.');

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

// 쿠키에 이름 정보를 설정함
app.get('/process/setUserCookie', function(req, res) {
  console.log('/process/setUserCookie 호출됨.');

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

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

……,
```

cookie-parser 미들웨어를 사용하도록 설정한 후 라우팅 미들웨어의 get( ) 메소드를 호출하여 /process/showCookie와 /process/setUserCookie 패스를 처리하는 콜백 함수를 등록합니다.

클라이언트의 쿠키 정보는 cookies 객체에 들어 있는데 사용자가 응답 문서를 조회할 때 쿠키 정보를 볼 수 있도록 /process/showCookie 패스를 처리하는 함수에서 이 쿠키 객체를 응답으로 보냅니다. 그러면 클라이언트에서는 쿠키 객체를 그대로 전달받게 됩니다.

/process/setUserCookie 패스를 처리하는 함수에서는 응답 객체의 cookie( ) 메소드로 user 쿠키를 추가합니다. 그러면 쿠키가 클라이언트 웹 브라우저에 설정되는데 마지막 코드 부분에서 redirect( ) 메소드로 /process/showCookie 패스를 다시 요청합니다. 그러므로 그 정보는 그대로 다시 클라이언트의 웹 브라우저에 표시됩니다. 다음은 쿠키가 클라이언트에 저장된 후 사용자에게 표시될 때까지의 과정입니다.

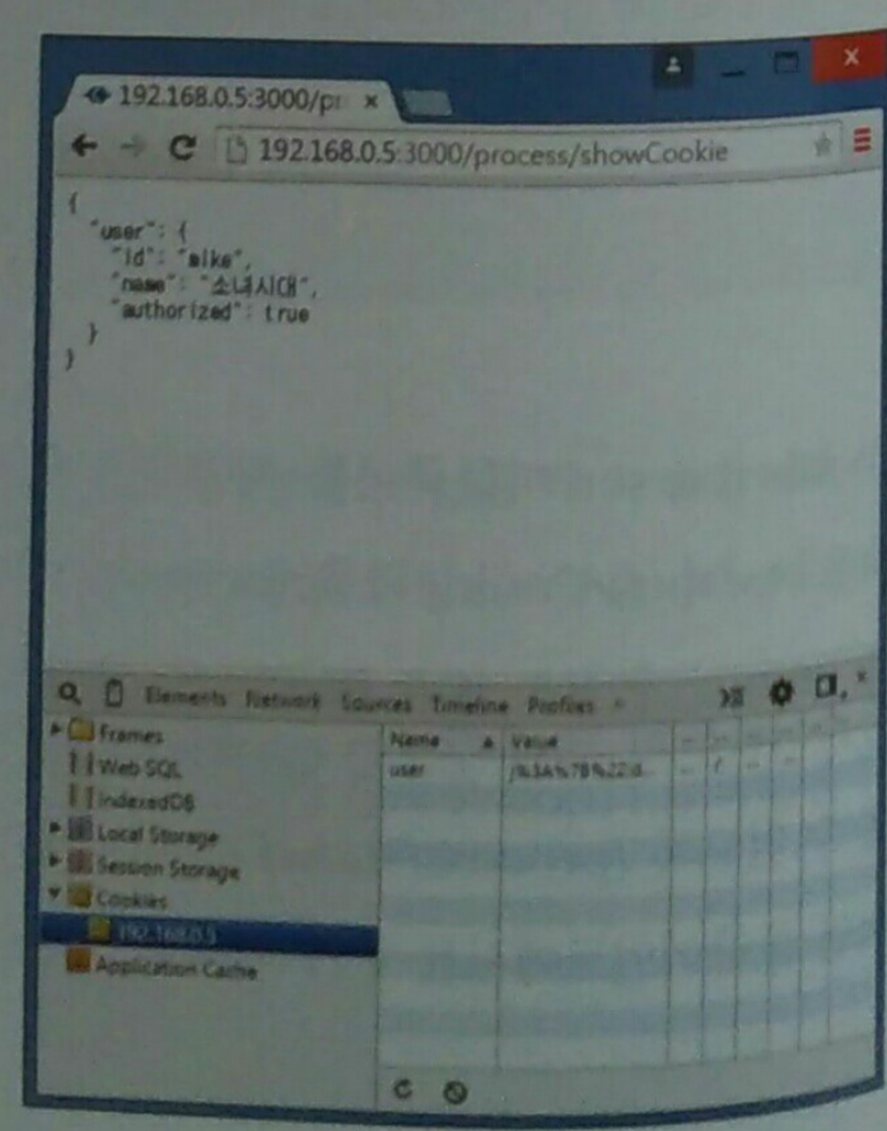


▲쿠키를 설정하고 사용자에게 보여 주는 과정

일반적으로는 쿠키를 설정하고 나면 그 정보는 다른 과정에 사용되지만 여기에서는 설정된 쿠키 정보를 단순히 확인만 하겠습니다. 쿠키를 사용할 수 있도록 만들어 주는 cookie-parser 모듈은 외장 모듈이므로 명령 프롬프트에서 다음 명령으로 설치합니다.

```
% npm install cookie-parser --save
```

그다음 app11.js를 실행한 후 웹 브라우저를 열고 /process/setUserCookie 패스로 요청합니다.



쿠키를 설정한 정보 확인▶

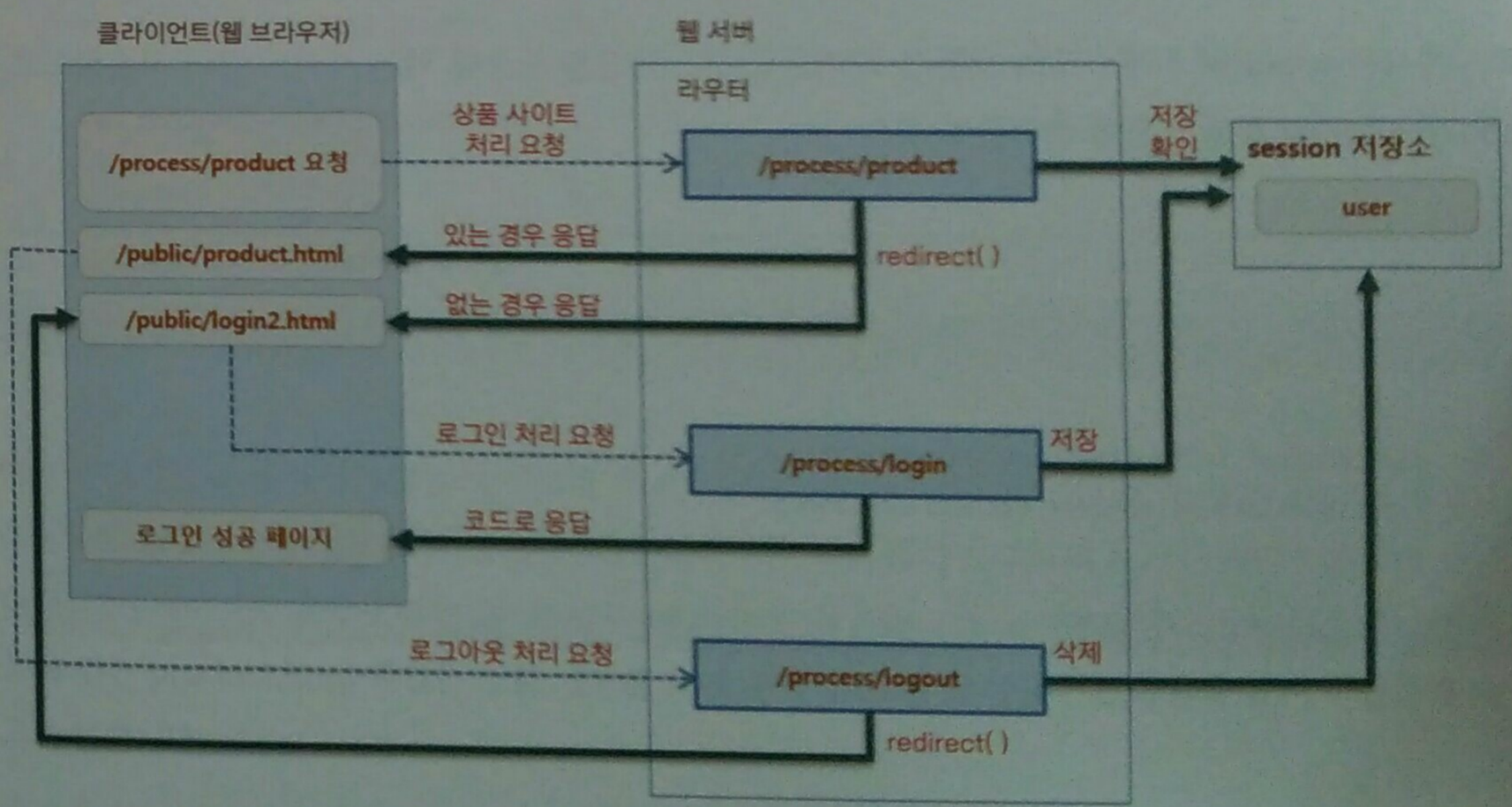
서버에서 user라는 이름으로 설정한 쿠키 정보를 웹 브라우저 화면에 보여 줍니다. 쿠키가 브라우저에 제대로 설정되었는지를 확인하기 위해 크롬 브라우저의 개발자 도구 화면을 띄웁니다. 개발자 도구는 오른쪽 위의 [설정] 버튼을 누른 후 [도구 더보기 → 개발자 도구] 메뉴를 선택합니다. 브라우저의 아래쪽에 보이는 개발자 도구 화면에는 콘솔 창을 포함해 웹 문서를 구성하는 문서 요소를 보여 주는 창, 네트워크 요청과 응답 내역을 보여 주는 창 등이 있습니다. 그중에서 [Resources] 탭을 클릭하면 브라우저에 저장된 리소스를 보여 줍니다. [Cookies] 항목을 클릭하면 현재 PC의 IP 정보가 보이는데 그 IP 정보를 클릭하면 오른쪽에 설정된 쿠키 정보가 표시됩니다. 쿠키를 저장하고 나면 이곳에서 'user'라는 이름으로 추가된 쿠키를 볼 수 있습니다.

세션 처리하기

이번에는 세션을 만들어 보겠습니다. 세션도 상태 정보를 저장하는 역할을 하지만, 쿠키와 달리 서버 쪽에 저장됩니다. 세션을 사용하는 대표적인 예로는 로그인을 했을 때 저장되는 세션을 들 수 있습니다.

사용자가 로그인하면 세션이 만들어지고 로그아웃하면 세션이 삭제되는 기능을 만들면 사용자가 로그인하기 전에는 접근이 제한된 페이지를 보지 못하도록 설정할 수 있습니다. 즉, 사용자가 상품 페이지처럼 접근이 제한된 페이지를 조회했을 때 로그인된 상태가 아니라면 로그인 페이지를 자동으로 열어 줍니다. 그러면 로그인을 해야 상품 페이지로 이동할 수 있으며 상품 페이지에서 로그아웃할 수 있습니다.

먼저 세션을 사용한 로그인 처리 과정은 다음과 같습니다.



▲ 세션을 사용한 로그인 처리 과정

세션이 있을 때와 없을 때 처리하는 방식이 달라지므로 전체 처리 과정은 약간 복잡해 보일 수 있습니다. 하나씩 차근차근 살펴보면 먼저 사용자가 상품 정보를 웹 서버에 요청합니다. 상품 정보를 요청할 때는 GET 방식으로 /process/product를 패스로 요청합니다. 웹 서버에서는 /process/product 패스를 라우팅하여 처리하는데 그 안에서 user라는 이름으로 된 세션이 있는지 확인합니다. 만약 이미 저장된 user 세션이 있다면 클라이언트에 /public/product.html 파일을 응답으로 전송합니다. 그러면 사용자는 상품 정보를 볼 수 있습니다. 만약 user 세션이 없다면 로그인을 유도하기 위해 로그인 페이지로 이동합니다. redirect( ) 메소드로 /public/login2.html 로그인 페이지를 보여 줍니다.

로그인 페이지에서 처리하는 방식은 앞에서 만든 로그인 기능과 큰 차이가 없습니다. 하지만 /process/login 패스를 라우팅하여 처리할 때 로그인에 성공했으면 user 세션을 저장하고 클라이언트에게는 로그인 성공 페이지를 보여 준다는 점이 다릅니다. 로그인 성공 페이지는 코드로 만들어서 응답할 수도 있고 별도의 파일을 만든 후 그 파일을 전송할 수도 있습니다. 여기에서는 코드로 전송하는 방식을 사용합니다. 로그인 성공 페이지에는 상품 정보를 조회하는 링크를 넣어 해당 링크를 클릭했을 때 /process/product를 요청합니다. 로그아웃을 처리하는 라우팅 함수는 /process/logout 패스를 처리합니다.

이런 처리 과정에 맞게 동작하려면 먼저 세션을 저장하는 방법에 대해 알아야 합니다. 익스프레스에서는 세션을 지원하기 위해 express-session 모듈을 사용합니다. 명령 프롬프트에서 다음과 같이 모듈을 설치합니다.

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

코드에서는 require( ) 메소드로 모듈을 불러들입니다. 세션을 사용할 때는 쿠키도 같이 사용하므로 cookie-parser 모듈도 함께 불러들입니다.

참조 파일 | ExpressExample>app12.js

```
.....

var cookieParser = require('cookie-parser');
var expressSession = require('express-session');

.....
```

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

.....
```

express-session 모듈은 미들웨어로 사용되기 때문에 use( ) 메소드를 사용해서 미들웨어에 추가합니다. use( ) 메소드에는 세션 객체를 호출하여 반환되는 객체를 전달합니다. 초기화에 전달되는 값에는 secret 속성에 키 값을 넣어 줍니다. 이제 라우팅 함수를 만들 수 있습니다. 사용자가 처음에 상품 정보를 보기 위해 요청할 /process/product 패스에 콜백 함수를 다음과 같이 추가합니다.

참조 파일 | ExpressExample>app12.js

```
.....

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

  if (req.session.user) {
    res.redirect('/public/product.html');
  } else {
    res.redirect('/public/login2.html');
  }
});

.....
```

이 함수에서는 user 세션이 있는지 확인한 후 만약 세션 객체가 있으면 /public/product.html 파일을 응답으로 보내어 상품 정보를 볼 수 있게 합니다. 하지만 user 세션이 없다면 로그인 페이지를 보여 줍니다. 로그인 페이지는 앞에서 만든 /public/login2.html 파일을 그대로 사용합니다. [public] 폴더 안에 들어 있는 product.html 파일은 상품 정보를 보여 주기 위한 웹 문서입니다. 다음과 같이 간단하게 태그를 넣습니다.

참조 파일 | ExpressExample>/public/product.html

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

문서의 가장 아래쪽에는 로그아웃할 수 있도록 〈a〉 태그를 추가하고, 클릭했을 때 /process/logout을 호출합니다. 로그인 페이지에서 로그인을 시도하는 경우에는 /process/login 패스로 라우팅되므로 다음과 같이 로그인에 필요한 함수를 추가합니다.

참조 파일 | ExpressExample>app12.js

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

  var paramId = req.param('id');
  var paramPassword = req.param('password');

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

    res.redirect('/public/product.html');
  } else {
    // 세션 저장
    req.session.user = {
      id: paramId,
      name: '소녀시대',
      authorized: true
    };
```

```
    res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
    res.write('<h1>로그인 성공</h1>');
    res.write('<div><p>Param id : ' + paramId + '</p></div>');
    res.write('<div><p>Param password : ' + paramPassword + '</p></div>');
    res.write("<br><br><a href='/process/product'>상품 페이지로 이동하기</a>");
    res.end();
  }
});
```

user 세션이 이미 있는 경우에는 /public/product.html 페이지를 보여 주고 그렇지 않으면 로그인을 시도한 후 user 세션을 저장합니다. user 객체를 세션으로 저장하고 싶다면 요청 객체 안에 있는 session 객체의 속성으로 user 객체를 넣어 주면 됩니다. user 객체에는 id, name, authorized 속성을 넣어 보았습니다. user 세션을 저장한 후에는 클라이언트로 응답을 보냅니다. 로그인이 성공했음을 알리기 위해 클라이언트로 보낸 응답 코드를 보면 가장 아래쪽에 상품 페이지로 이동할 수 있는 링크가 있습니다.

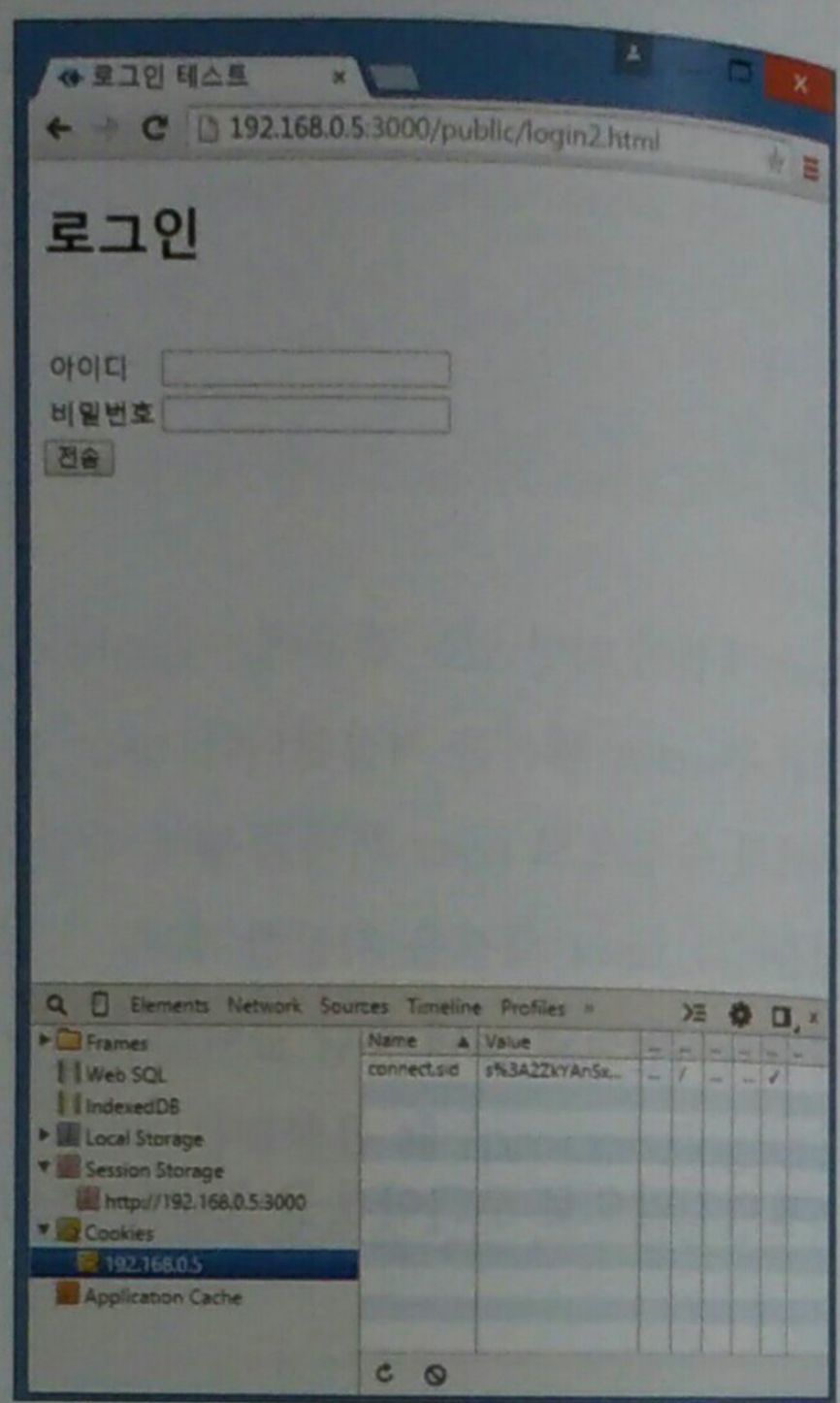
이제 마지막으로 로그아웃을 처리할 수 있는 함수를 추가합니다.

참조 파일 | ExpressExample>app12.js

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

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

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

        console.log('세션을 삭제하고 로그아웃되었습니다.');
        res.redirect('/public/login2.html');
    });
  } else {
    // 로그인 안 된 상태
    console.log('아직 로그인되어 있지 않습니다.');

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

로그아웃할 때는 session 객체에 정의된 destroy( ) 메소드를 호출하여 세션을 제거합니다. 세션을 없앤 후에는 redirect( ) 메소드로 /public/login2.html 페이지를 전송합니다.

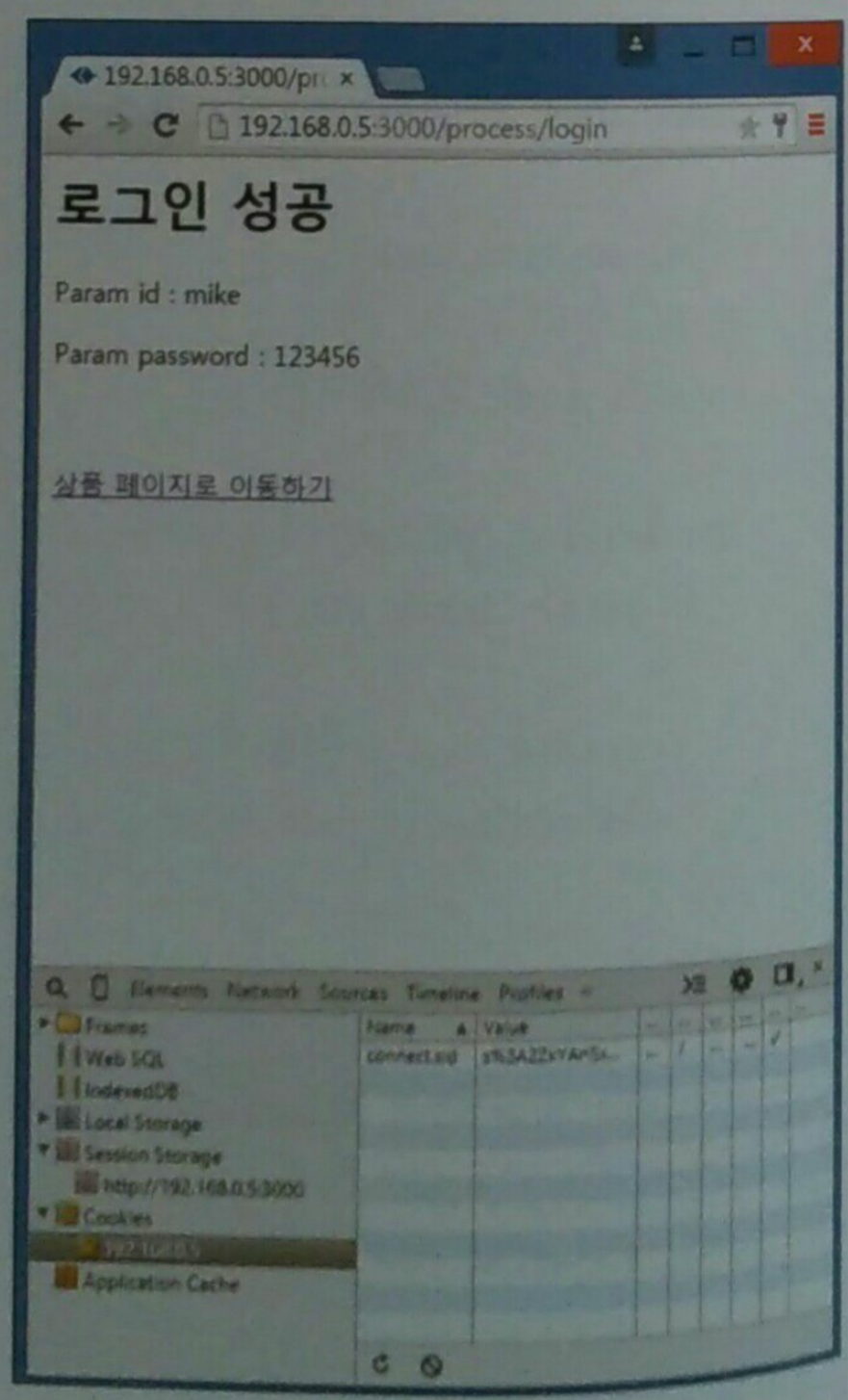
이제 서버를 실행한 후 웹 브라우저를 열고 http://localhost:3000/process/product 주소로 요청하면 처음에는 user 세션 객체가 만들어져 있지 않으므로 다음과 같이 로그인 페이지가 나타납니다.



/process/product로 요청했을 때 나타나는 로그인 페이지 ▶

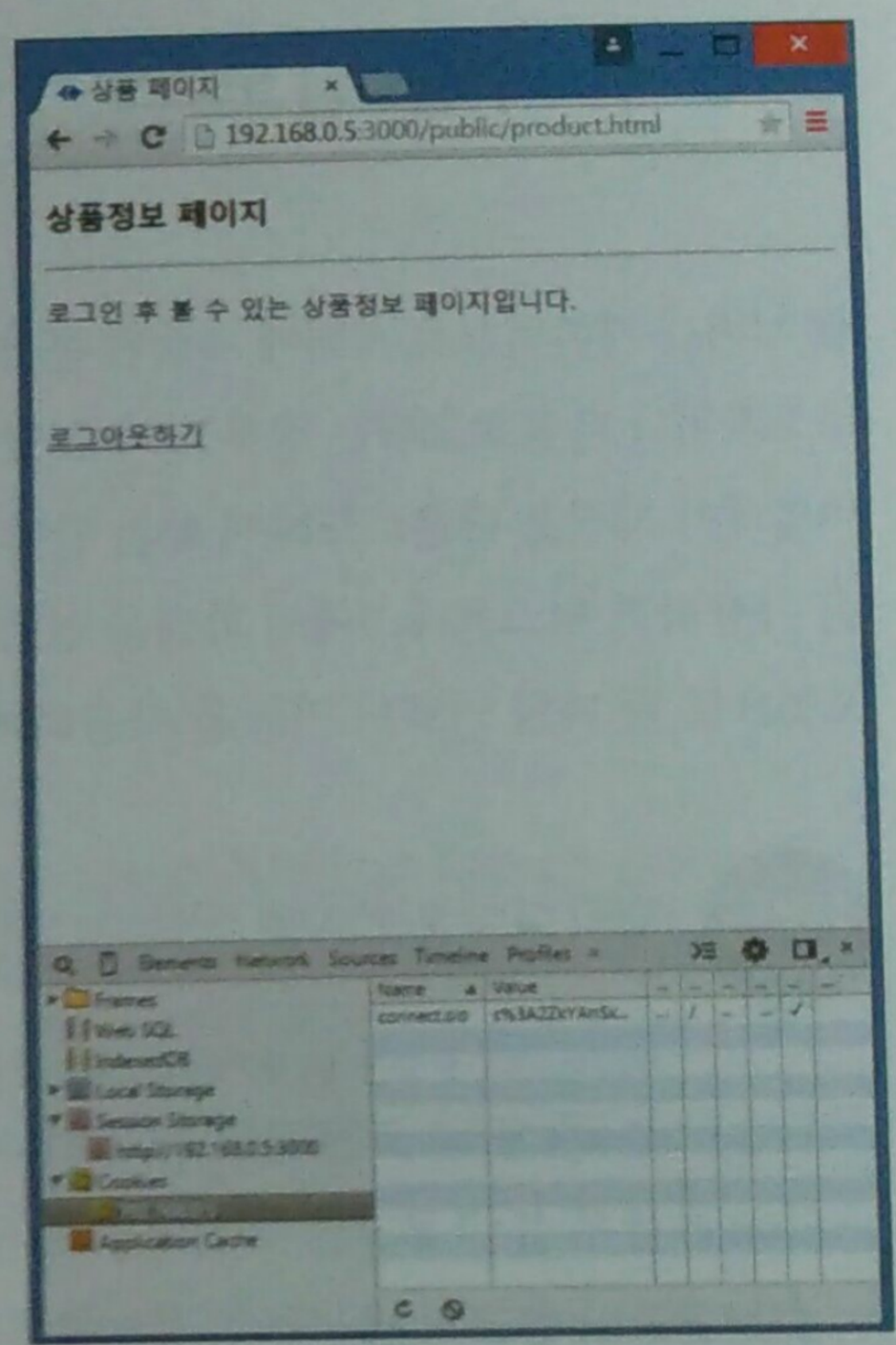
주의! 필요한 경우에는 크롬 브라우저의 [설정] 메뉴에서 개발자 도구를 보이도록 하여 화면과 같이 쿠키나 세션 객체를 확인할 수 있습니다.

아이디와 비밀번호를 입력한 후 [전송] 버튼을 누릅니다. 그러면 서버에서 로그인 상태를 유지하기 위해 user 세션을 저장한 후 로그인이 성공했다는 웹 문서를 보여 줍니다.



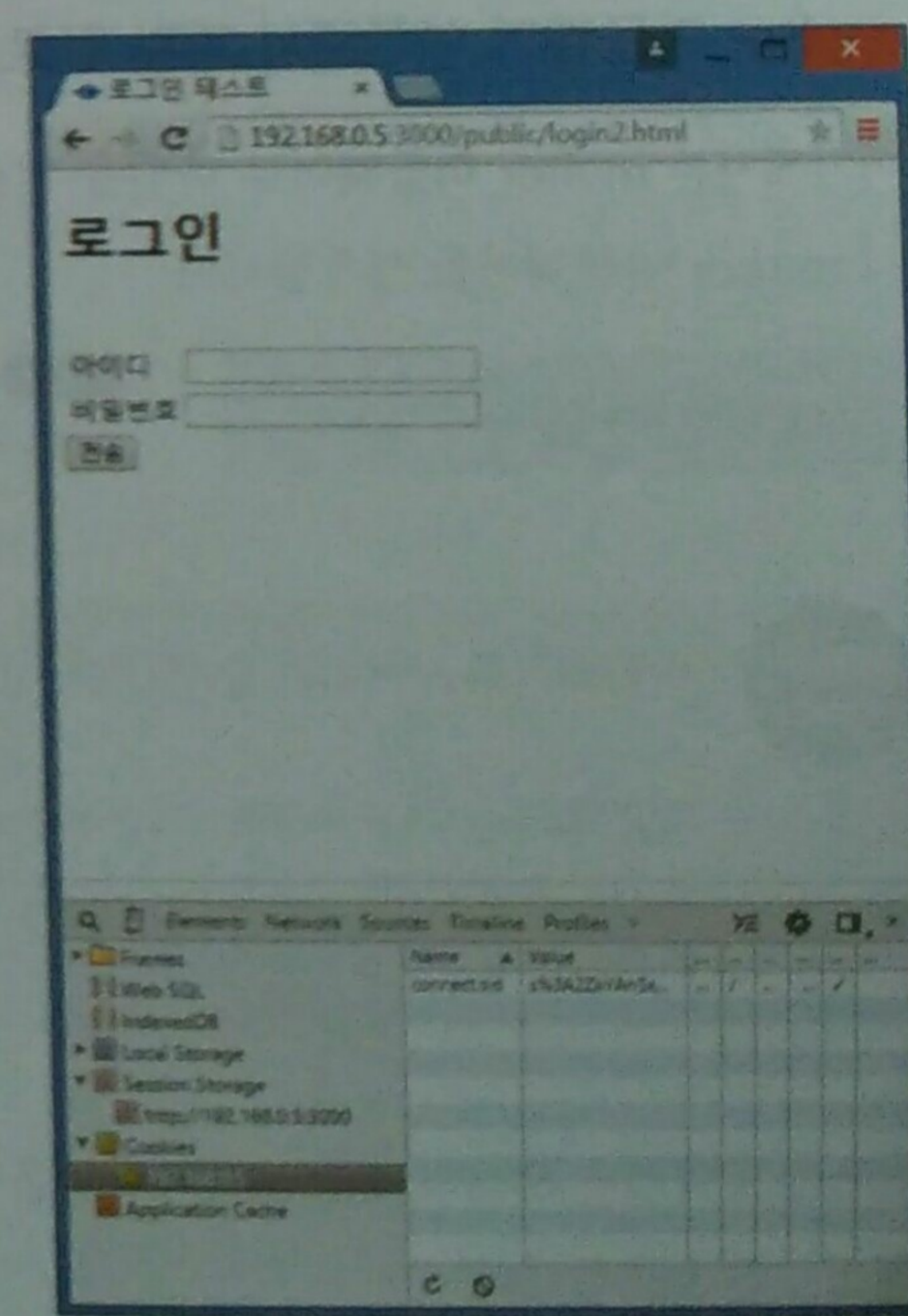
로그인을 시도했을 때 보이는 로그인 성공 메시지 ▶

화면 아래쪽에 있는 '상품 페이지로 이동하기' 링크를 누르면 다음과 같이 상품 페이지가 표시됩니다.



상품 페이지 표시▶

상품 페이지 아래쪽에 있는 '로그아웃하기' 링크를 클릭하면 user 세션 객체를 삭제한 후 다시 로그인 화면으로 이동합니다.



로그아웃한 후 보이는 로그인 화면▶

로그인 화면에서 로그인하여 세션이 만들어지면 connect.sid 쿠키가 브라우저에 저장됩니다. connect.sid 쿠키는 웹 브라우저에서 세션 정보를 저장할 때 만들어진 것입니다. 브라우저마다 이름이 다를 수는 있지만 쿠키를 사용해 세션 정보를 저장하는 방식은 같습니다.

지금까지 로그인처럼 일정 상태를 유지할 수 있도록 만들어 주는 세션에 대해 알아보았습니다.

5-6 | 파일 업로드 기능 만들기

웹 서버는 기본적으로 서버에 저장된 문서를 조회하거나 데이터를 받아 저장할 수 있지만 파일 자체를 업로드하거나 다운로드하는 경우도 자주 있습니다. 특히 모바일 단말로 찍은 사진을 업로드하고 웹이나 모바일에서 사진을 다운로드하여 보는 일이 많아지면서 이미지 파일을 다루는 경우도 많습니다. 외장 모듈을 사용하면 익스프레스에서 파일을 업로드할 수 있습니다. 파일을 업로드할 때는 **멀티 파트(multipart) 포맷**으로 된 파일 업로드 기능을 사용하며 파일 업로드 상태 등을 확인할 수 있습니다.

정박사의 한마디 멀티 파트 포맷은 웹 서버에서 파일을 업로드하기 위해 사용해요

multipart 포맷은 음악이나 이미지 파일 등을 일반 데이터와 함께 웹 서버로 보내려고 만든 표준입니다. 따라서 일반적으로 웹 서버에서 파일을 업로드할 때는 multipart 포맷을 사용합니다.

multer 미들웨어 설치해서 파일 업로드하기

여기에서는 **multer 미들웨어**로 파일을 업로드하는 방법에 대해 알아보겠습니다. 먼저 명령 프롬프트에서 multer 미들웨어를 설치합니다.

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

주의! 파일을 업로드할 때는 클라이언트에서 POST 방식으로 데이터를 전송하므로 body-parser 미들웨어를 함께 사용합니다.

정박사의 한마디 미들웨어를 사용하는 순서가 바뀌면 오류가 날 수 있어요

파일 업로드 기능을 구현할 때 body-parser, multer, router 등의 미들웨어를 사용합니다. 그런데 이렇게 여러 개의 미들웨어를 사용할 때 사용 순서가 바뀌면 제대로 동작하지 않을 수 있습니다. 따라서 미들웨어의 사용 순서가 바뀌지 않도록 주의합니다.

app12.js 파일을 복사하여 app13.js 파일을 만든 후 모듈을 불러들이는 부분에서 다음과 같이 require() 메소드를 호출합니다.

참조 파일 | ExpressExample>app13.js

```
……

var bodyParser = require('body-parser');
var cookieParser = require('cookie-parser');
var expressSession = require('express-session');

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

……
```

파일을 업로드한 후에 업로드한 파일을 다루어야 하는 경우가 많으므로 fs 모듈도 함께 불러들였습니다. body-parser, cookie-parser, express-session을 미들웨어로 사용할 수 있도록 다음 부분에 multer 미들웨어를 추가하는 코드를 넣습니다. 그다음에는 router 미들웨어를 추가하는 코드를 입력합니다.

참조 파일 | ExpressExample>app13.js

```
……

app.use(bodyParser.urlencoded({extended: true}));

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

// multer 미들웨어 사용하기
app.use(multer({
  dest: 'uploads',
  putSingleFilesInArray: true,
  limits: {
    files: 10,
    fileSize: 1024 * 1024
  },
  rename: function (fieldname, filename) {
    return filename+Date.now();
```

```
},
onFileUploadStart: function (file) {
  console.log('파일 업로드 시작 : '+ file.originalname);
},
onFileUploadComplete: function (file, req, res) {
  console.log('파일 업로드 완료 : ' + file.fieldname + ' -> ' + file.path);
},
onFileSizeLimit: function (file) {
  console.log('파일 크기 제한 초과 : %s', file.originalname);
}));

......
```

multer 미들웨어를 사용하려면 app.use( ) 메소드와 multer 미들웨어 함수를 동시에 호출하여 반환된 객체를 파라미터로 넘겨주어야 합니다. 이때 multer 함수를 호출하면서 파라미터로 전달하는 객체에는 속성이나 콜백 함수를 설정할 수 있습니다. 우리는 다양한 속성과 콜백 함수를 설정했지만, 간단하게 **dest 속성**에 파일이 저장될 폴더만 지정해도 모듈이 잘 동작합니다. 여기에서 사용한 주요 속성과 메소드의 의미는 다음과 같습니다.

속성/ 메소드 이름	설명
dest	업로드한 파일들이 저장될 폴더를 지정합니다.
putSingleFilesInArray	응답 객체의 files 속성에 업로드한 파일 정보를 클라이언트 요청 이름으로 넣을 때, 파일이 한 개라도 배열 형태로 넣을 것인지를 지정합니다.
limits	파일 크기나 파일 개수 등의 제한 속성을 설정하는 객체입니다.
rename(fieldname, filename, req, res)	업로드한 파일의 이름을 바꿉니다.
onFileUploadStart(file, req, res)	파일 업로드가 시작될 때 호출됩니다.
onFileUploadComplete(file, req, res)	파일 업로드가 완료되었을 때 호출됩니다.
onFileSizeLimit(file)	limits 객체에 설정한 제한을 넘어섰을 때 호출됩니다.

[표] multer 미들웨어를 사용할 때 설정하는 주요 속성과 메소드

dest 속성으로 지정한 폴더는 프로젝트 폴더 안에 만들어져 있어야 합니다. 여기에서는 [uploads] 폴더로 지정했으므로 업로드된 파일은 모두 이 폴더에 저장됩니다.

putSingleFilesInArray 속성의 값을 true로 설정하면 파일 업로드 기능을 처리하는 라우팅 함수 안에서 파일을 참조하기 편리해집니다. 파일은 응답 객체의 files 속성에 들어가며, 각각의 파일 이름은 클라이언트에서 보내 온 name 속성의 값이 됩니다. 그런데 파일이 여러 개라면 배열로 만들어지지만 한 개일 경우에는 배열로 만들어지지 않을 수 있습니다. 이 때문에 배열일 경우와 배열이 아닐 경우를 따로 처리해야 하는 번거로움이 생깁니다. 이때 putSingleFilesInArray 속성을 설정하면 모든 파일이 배열로 저장되므로 파일 정보를 확인할 때 더 편리합니다.

limits 객체는 파일 크기를 제한하는 등 파일을 다룰 때 필요한 제약(Limitation)을 만들어 자바스크립트 객체로 설정할 수 있습니다. 여기에서는 한 번에 업로드할 수 있는 파일의 수를 10개로 제한하고, 파일 크기는 1Mbyte까지로 허용하였습니다.

파일을 업로드하면 업로드한 파일 이름이 중복될 수 있습니다. 이 때문에 고유한 파일 이름으로 변경하여 저장하는 것이 좋습니다. rename( ) 메소드는 파일을 업로드했을 때 그 이름을 변경할 수 있습니다. 여기에서는 다른 파일과 중복되지 않도록 업로드한 파일 이름에 현재 시간을 붙여 저장했습니다.

onFileUploadStart( )와 onFileUploadComplete( ) 메소드를 사용하면 파일이 업로드되는 상태를 알 수 있습니다. 이 메소드들은 파일 업로드가 시작될 때와 완료되었을 때 각각 호출됩니다. 그 밖에 제한된 파일 크기를 넘어서는 경우에 호출되는 onFileSizeLimit( ) 메소드가 있습니다.

클라이언트의 요청 처리 함수 추가하기

이제 multer 미들웨어를 사용할 수 있게 되었으므로 클라이언트의 요청을 처리할 함수를 추가할 수 있습니다. 서버 쪽에서 처리할 함수를 추가하기 전에 먼저 사용자가 띄워 볼 웹 문서를 만듭니다. [public] 폴더 안에 photo.html 파일을 만든 후 다음 코드를 입력합니다.

참조 파일 | ExpressExample>/public/photo.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>파일 업로드 테스트</title>
  </head>
<body>
  <h1>파일 업로드</h1>
  <br>
  <form method="post" enctype="multipart/form-data" action="/process/photo">
```

```
    <table>
      <tr>
        <td><label>파일</label></td>
        <td><input type="file" name="photo"></td>
      </tr>
    </table>
    <input type="submit" value="업로드" name="submit">
  </form>
</body>
</html>
```

<form> 태그 안에 들어가는 <input> 태그의 type 속성 값으로 file을 입력하면 파일을 선택하여 멀티파트 포맷으로 서버에 업로드를 요청할 수 있습니다. <form> 태그의 method 속성 값은 post로 설정하고 enctype 속성 값은 multipart/form-data로 설정합니다. 요청할 URL 정보인 /process/photo는 action 속성의 값으로 넣어 줍니다.

주의! 주의할 것은 <input> 태그의 name 속성입니다. 이 속성의 값은 서버로 보내져 서버에서 처리할 때 요청 객체에 들어 있는 files 객체의 속성 이름이 됩니다. 따라서 이 name 속성의 값을 바꾸면 서버에서 참조하는 파일 객체의 속성 이름도 바뀌어야 합니다.

이제 사용자가 업로드한 사진 파일을 받아서 처리할 수 있도록 다음과 같이 /process/photo 패스를 라우팅하는 함수를 추가합니다.

참조 파일 | ExpressExample>app13.js

```
……

app.post('/process/photo', function(req, res) {
  console.log('/process/photo 호출됨.');

  var files = req.files.photo;

  // 현재의 파일 정보를 저장할 변수 선언
  var originalname = '',
          name = '',
          mimetype = '',
          size = 0;

  if (Array.isArray(files)) {   // 배열에 들어 있는 경우
    console.log("배열에 들어 있는 파일 개수 : %d", files.length);
    for (var index = 0; index < files.length; index++) {
      originalname = files[index].originalname;
      name = files[index].name;
      mimetype = files[index].mimetype;
```

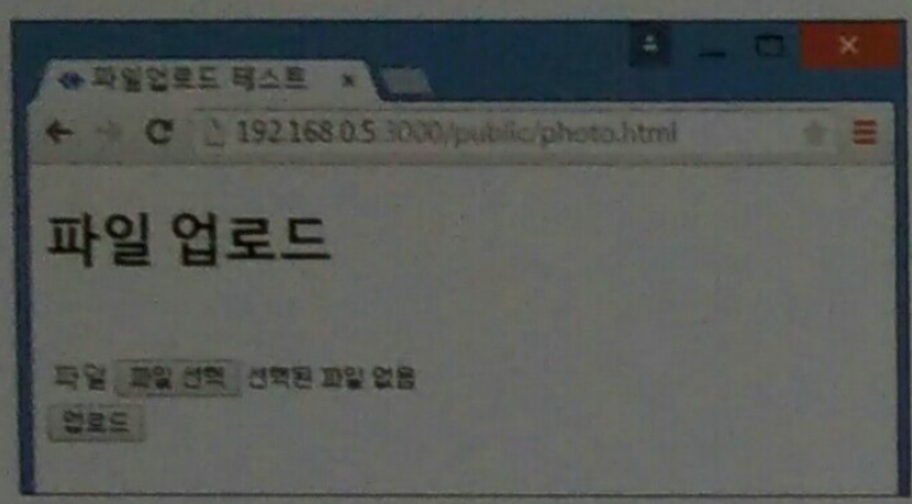
```
    size = files[index].size;
  }

} else {   // 배열에 들어가 있지 않은 경우 (현재 설정에서는 해당 없음)
  console.log("파일 개수 : 1 ");

  originalname = files[index].originalname;
  name = files[index].name;
  mimetype = files[index].mimetype;
  size = files[index].size;
}

console.log('현재 파일 정보 : ' + originalname + ', ' + name + ', '
            + mimetype + ', ' + size);

// 클라이언트에 응답 전송
res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
res.write('<h3>파일 업로드 성공</h3>');
res.write('<hr/>');
res.write('<p>원본 파일 이름 : ' + originalname + ' -> 저장 파일 이름 : ' + name + '</p>');
res.write('<p>MIME TYPE : ' + mimetype + '</p>');
res.write('<p>파일 크기 : ' + size + '</p>');
res.end();

});

.....
```

코드가 제법 길지만 실제로는 파일의 정보를 확인하고 클라이언트에 응답을 보내는 것이 전부입니다. 업로드한 파일의 정보를 확인할 때는 req.files.photo 객체를 참조하는데, 앞에서 이야기한 것처럼 photo라는 속성 이름은 photo.html 문서에 들어 있는 〈input〉 태그에서 지정한 것입니다.

파일을 업로드했을 때 업로드한 파일의 정보는 배열 객체로 저장됩니다. for문으로 배열 객체의 요소들인 파일 이름이나 크기를 하나씩 확인합니다. originalname 속성은 클라이언트에서 보낼 때의 원본 파일 이름이며, name은 서버에 저장될 파일 이름입니다. 업로드된 파일이 어떤 MIME TYPE으로 전달된 것인지 mimetype 속성으로 확인할 수 있으며, 파일 크기는 size 속성으로 확인할 수 있습니다.

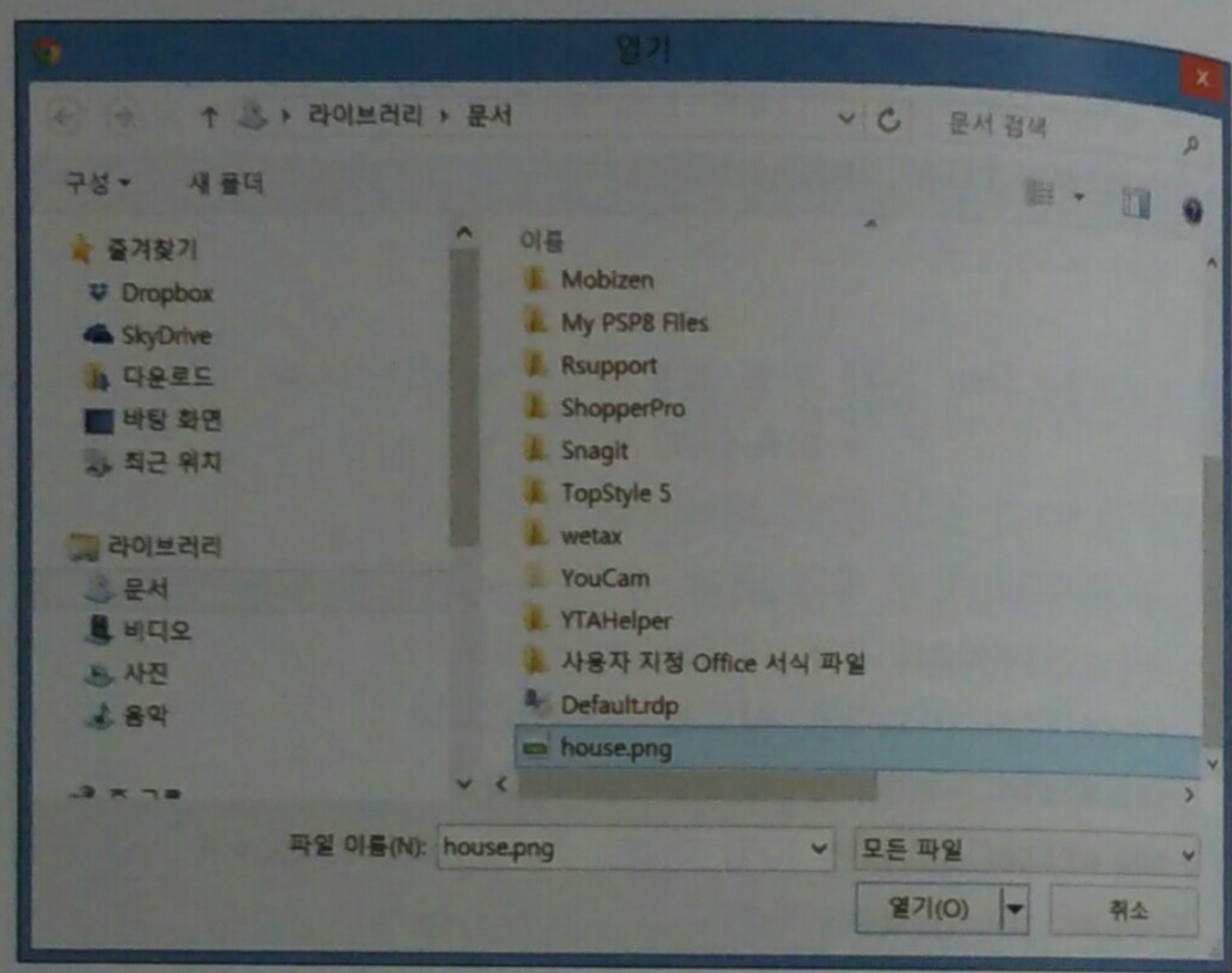
이제 서버를 실행한 후 웹 브라우저를 열고 다음 주소로 요청합니다. 그러면 파일을 업로드하기 위한 웹 문서가 표시됩니다.

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



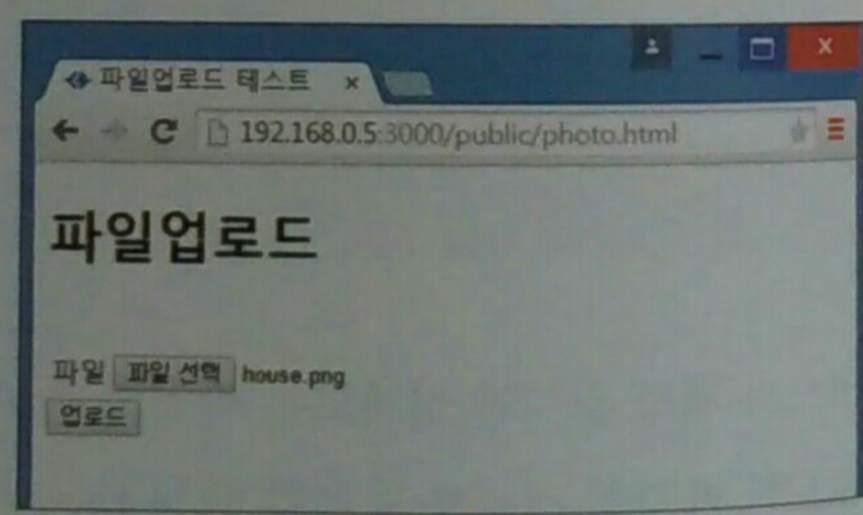
◀ 파일 업로드를 위한 웹 문서

〈input〉 태그의 type 속성 값을 file로 했기 때문에 [파일 선택] 버튼이 보입니다. [파일 선택] 버튼을 눌러 PC에 저장되어 있는 이미지 파일 하나를 선택합니다.



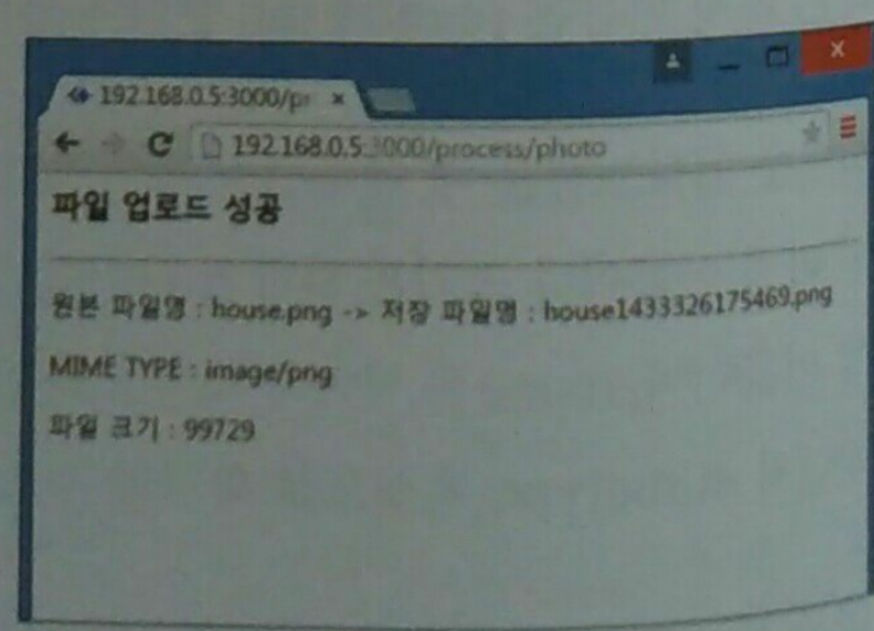
▲ PC에 있는 이미지 파일 선택하기

파일을 선택하면 선택한 파일의 이름이 웹 문서에 표시됩니다.



파일을 선택한 후의 웹 문서 ▶

[업로드] 버튼을 누르면 파일을 서버로 업로드한 후 다음과 같이 업로드 성공 메시지를 보여 줍니다. 서버에서는 업로드가 성공했을 때 파일 정보를 확인하고 응답을 보내도록 설정되어 있습니다. 따라서 응답 웹 문서에는 업로드한 파일의 정보가 함께 표시되었습니다.



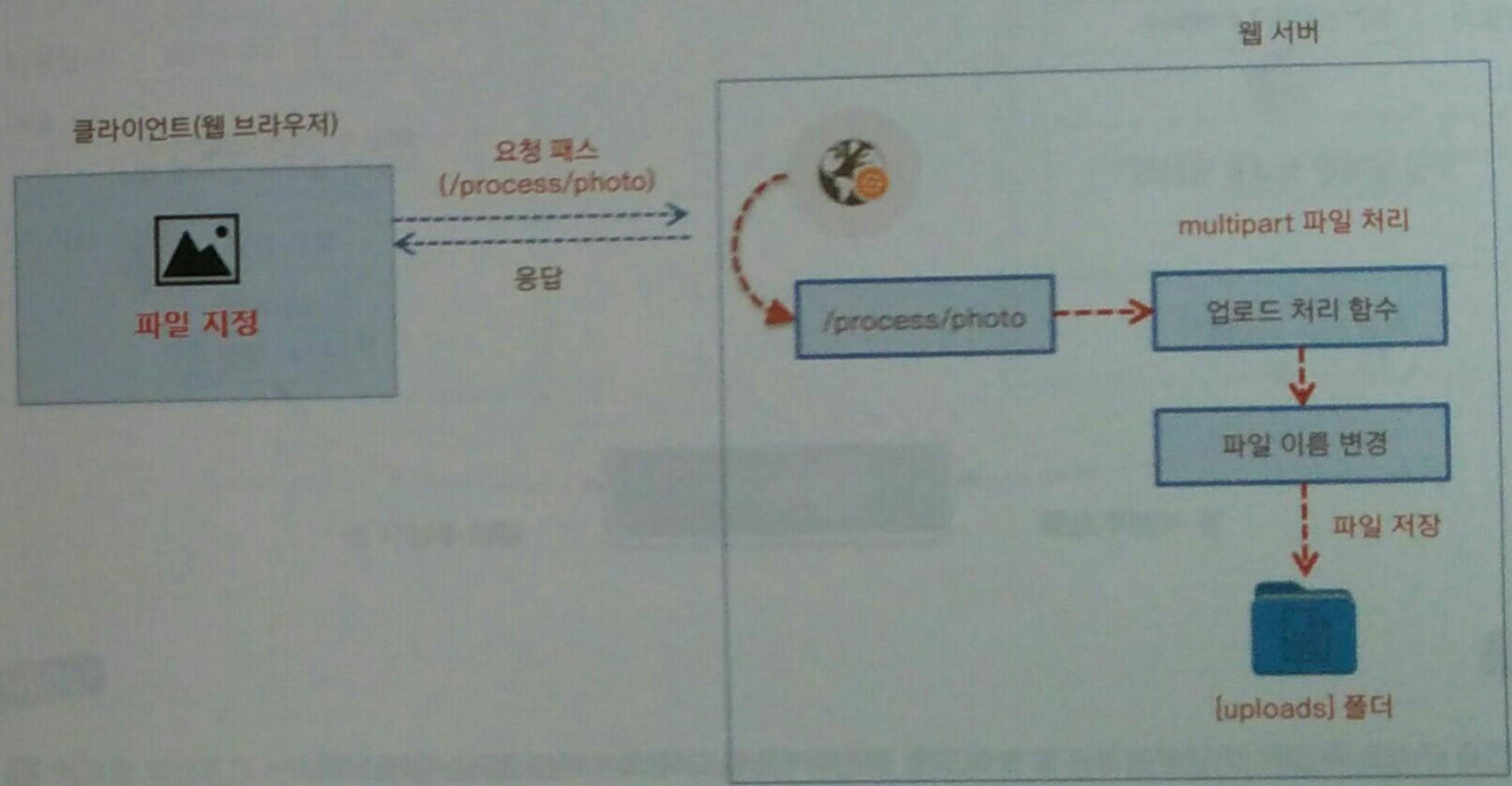
업로드한 후 보이는 파일 정보 ▶

서버의 콘솔 창을 보면 클라이언트에서 파일을 업로드했을 때의 과정과 함께 업로드한 파일이 서버의 [uploads] 폴더에 어떤 이름으로 저장되었는지 알 수 있는 로그가 출력됩니다.

```
Console  Debug
NodeExample2-app13.js [Node Application] Node.js Process
Express server listening on port 3000
파일 업로드 시작 : house.png
파일 업로드 완료 : photo ->  uploads\house1433327292722.png
/process/photo 호출됨.
배열에 들어있는 파일 갯수 : 1
현재 파일 정보 : house.png, house1433327292722.png, image/png, 99729
```

▲ 업로드한 후 서버 쪽에 보이는 콘솔 창의 로그 정보

이미지 파일을 업로드하는 과정을 정리하면 다음과 같습니다.



▲ 이미지 파일을 업로드하는 과정

웹 브라우저에서 이미지 파일을 선택하고 [업로드] 버튼을 눌러 웹 서버로 요청하면 업로드 기능을 담당하는 multer 미들웨어를 거쳐 라우팅 함수 쪽으로 전달됩니다. 이 함수에서는 파일 이름을 변경하여 [uploads] 폴더에 저장됩니다. 이 과정에서 확인된 정보는 응답으로 보내지므로 웹 브라우저에서 조회하는 응답 웹 문서에서 업로드 결과를 볼 수 있습니다.

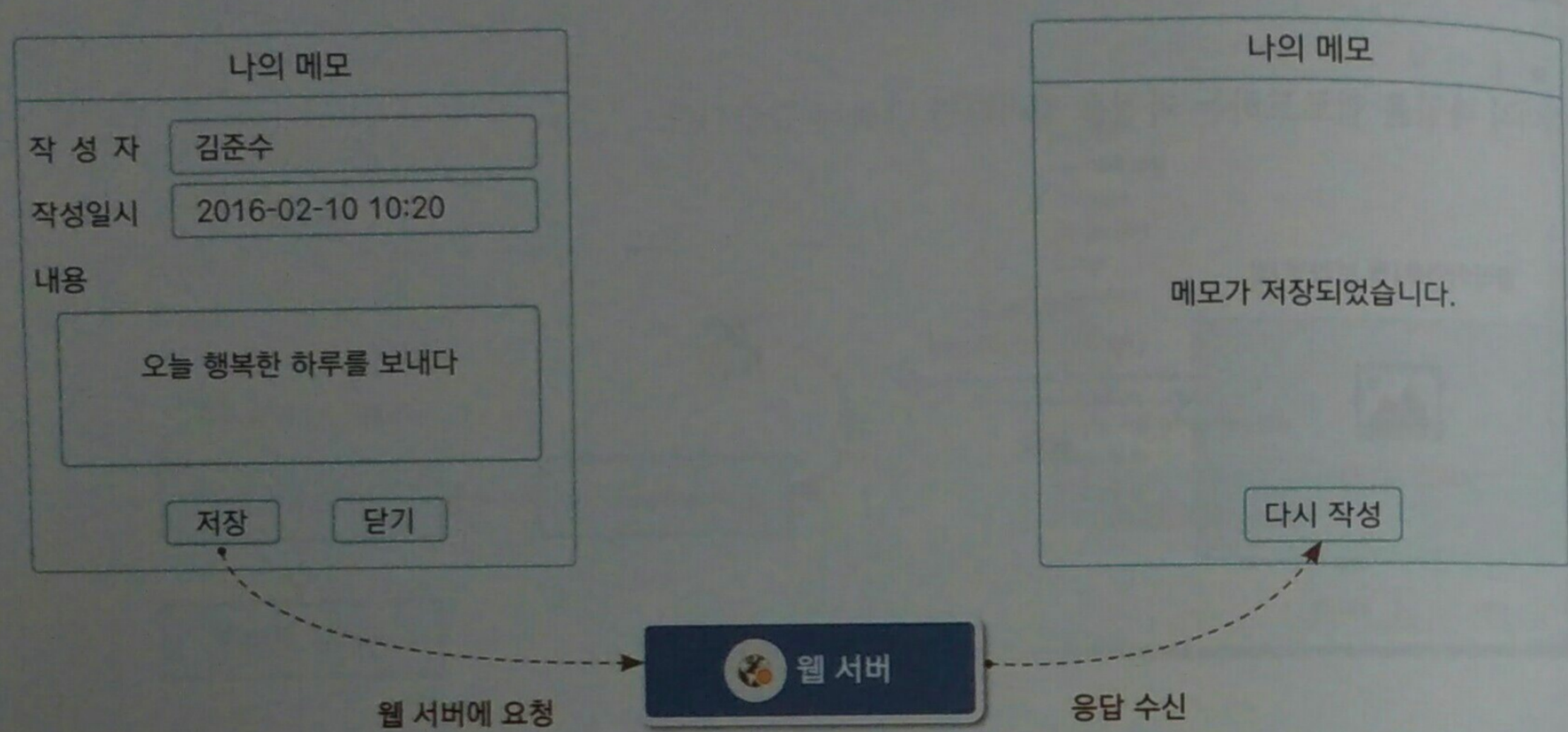
이제 이미지 파일을 웹 서버에 업로드할 수 있게 되었습니다. 만약 여러분이 쇼핑몰 사이트를 운영하는 판매자라고 가정해 보겠습니다. 판매자인 여러분이 상품 정보를 올렸을 때 일반 사용자가 업로드된 상품 정보를 조회할 수 있는 기능을 만들고 싶다면 지금까지 익힌 업로드 예제를 활용하면 충분히 해결할 수 있습니다.

다음 장에서는 데이터베이스에 데이터를 저장하는 방법에 대해 알아보겠습니다.

Do it!
도전 문제

01. 메모를 기록하여 웹 서버로 보내는 기능을 만들어 보세요.

프로젝트 소스 Mission03 난이도 초급 ★☆☆☆



Hint!

1. 화면을 구성할 때 모바일 단말에서도 잘 보이도록 간단하게 만들되 예쁘게 만들 필요는 없습니다.

2. 화면은 작성 화면과 응답 화면으로 구성합니다. 작성 화면에는 작성자와 작성 날짜 그리고 내용을 입력할 수 있는 입력 상자를 추가합니다. 가장 아래쪽에는 [저장] 버튼과 [닫기] 버튼을 배치합니다.

3. [저장] 버튼을 누르면 웹 서버로 메모 내용을 보냅니다. 웹 서버에서는 메모 내용을 확인한 후 정상적으로 저장되었다는 응답 메시지를 클라이언트로 보냅니다.

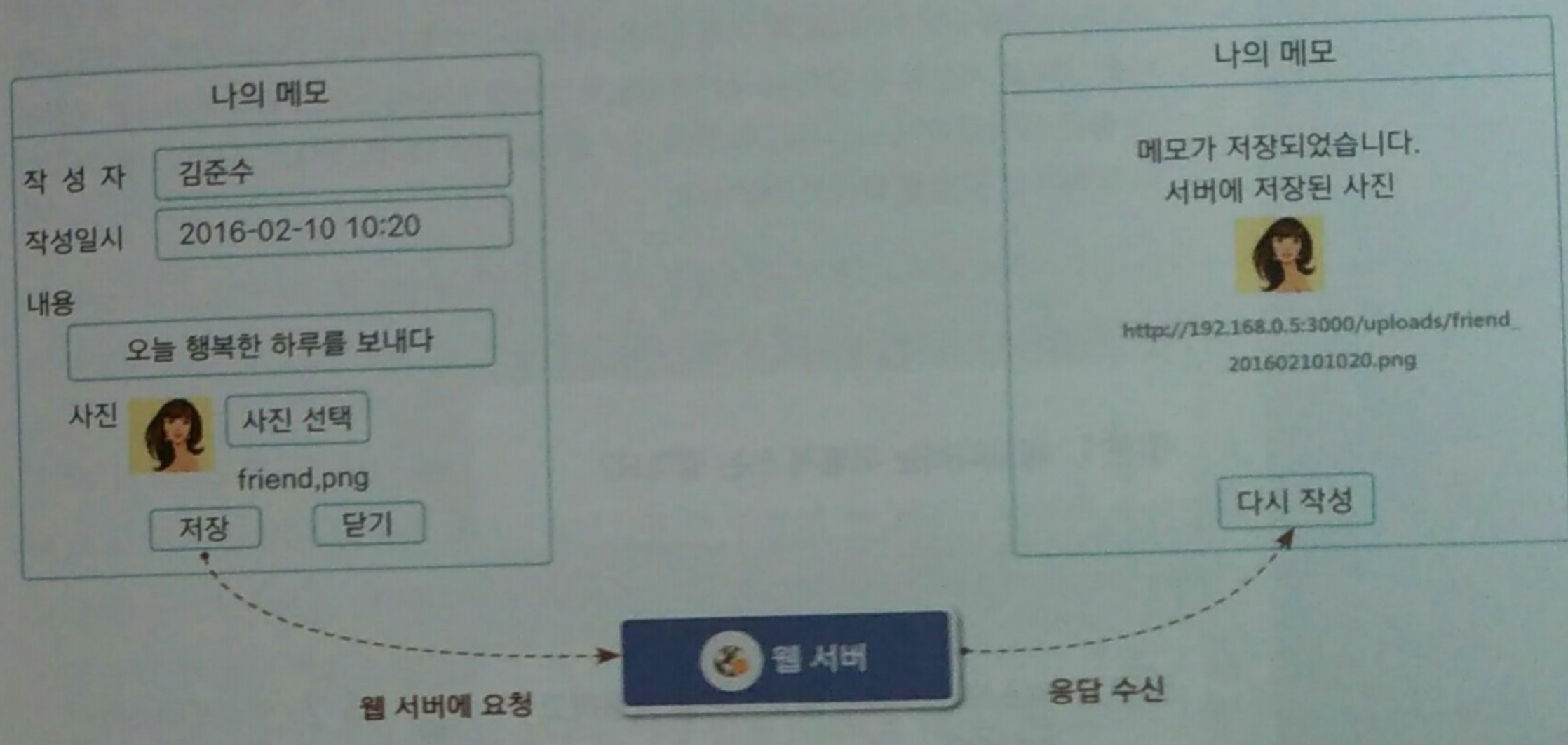
4. 클라이언트에서 응답 메시지를 받으면 응답 화면에 메시지를 보여 줍니다. 응답 메시지 아래쪽에는 [다시 작성] 버튼을 만들고, 이 버튼을 누르면 작성 화면이 표시됩니다.

MEMO

Do it!
도전 문제

02. 메모를 기록하고 사진과 함께 웹 서버로 보내는 기능을 만들어 보세요.

프로젝트 소스 Mission04 난이도 중급 ★★☆☆



Hint!

1. 메모를 작성하고 서버에 보낸 후 응답을 받아 표시하는 기본 화면은 앞의 초급 과제와 같습니다. 파일을 복사하여 만들고 한두 가지 기능을 추가합니다.

2. 작성 화면의 아래쪽에는 사진을 선택해서 표시할 수 있도록 구성합니다. [사진 선택] 버튼을 누르면 PC나 모바일 단말에서 사진을 선택한 후 보여 줍니다.

3. [저장] 버튼을 누르면 웹 서버로 메모 내용을 보내고 사진도 함께 보냅니다. 웹 서버에서는 메모 내용을 확인하고 사진도 업로드한 후 정상적으로 저장되었다는 응답 메시지를 클라이언트로 보냅니다.

4. 클라이언트에서 응답 메시지를 받으면 응답 화면에 메시지를 보여 줍니다. 응답 메시지 아래쪽에는 서버에 업로드한 사진을 보여 줍니다.

MEMO

06 데이터베이스 사용하기

웹 서버가 사용자 요청을 받으면 데이터베이스에 있는 데이터를 조회하여 응답하거나 또는 사용자가 보낸 데이터를 데이터베이스에 저장합니다. 노드로 서버를 만들 때도 데이터를 저장하기 위해 다양한 데이터베이스를 사용합니다. 데이터베이스의 유형은 데이터를 메모리에 저장하는 형태부터 오라클이나 MySQL 같은 관계형 데이터베이스, 몽고디비(MongoDB) 같은 NoSQL에 이르기까지 다양합니다. 특히 몽고디비는 자바스크립트 객체를 그대로 저장할 수 있어서 자바스크립트 언어를 사용하는 노드에서 데이터를 저장하기 좋은 데이터베이스입니다. 이 장에서는 몽고디비로 데이터베이스에 데이터를 저장하고 조회하는 방법을 알아보겠습니다.

질문으로 시작하기

질문1. 몽고디비는 어떻게 쓰는 걸까요? ➡ 06-1. 몽고디비 시작하기

질문2. 익스프레스에서 몽고디비를 사용하고 싶나요? ➡ 06-2. 익스프레스에서 몽고디비 사용하기

질문3. 스키마를 만들어서 사용할 수도 있나요? ➡ 06-3. 몽구스로 데이터베이스 다루기

질문4. 인덱스와 메소드 사용법을 알고 싶나요? ➡ 06-4. 인덱스와 메소드 사용하기

질문5. 비밀번호를 저장할 때 암호화는 어떻게 하나요? ➡ 06-5. 비밀번호 암호화하여 저장하기

질문6. 사용하던 관계형 데이터베이스에 연결할 수 있나요? ➡ 06-6. MySQL 데이터베이스 사용하기

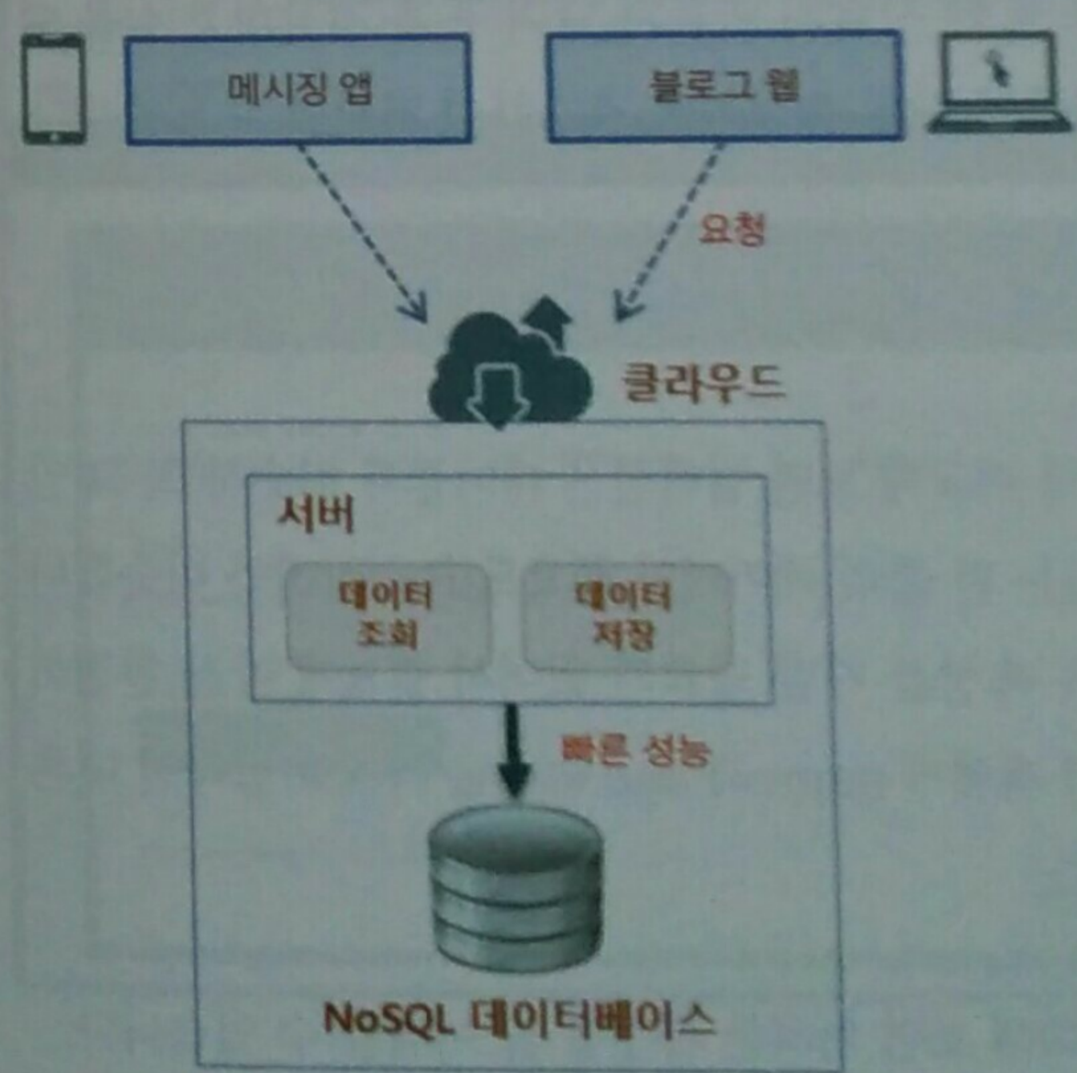
06-1 | 몽고디비 시작하기

노드에서 데이터 저장이 필요할 때 몽고디비를 사용하는 경우가 많습니다. 몽고디비는 기존에 자주 사용하던 **관계형 데이터베이스(Relational Database)**와 달라 SQL을 사용하지 않습니다. 또 자바스크립트 객체를 그대로 저장할 수 있어서 데이터 조회 방식도 SQL과 다릅니다. 하지만 데이터를 저장하거나 조회하는 방법을 따로 제공하기 때문에 몇 가지 사용법만 알아 두면 쉽게 사용할 수 있습니다.

몽고디비란?

실무에서는 오라클이나 MySQL과 같은 관계형 데이터베이스를 많이 사용합니다. 그러나 최근 몇 년 사이에 **비관계형 데이터베이스**를 적용하는 곳이 늘고 있으며, 이런 시스템을 NoSQL 또는 Not Only SQL이라고 합니다.

SQL 문을 사용하는 관계형 데이터베이스를 더 편하게 생각하는 사람이 많은데도 왜 NoSQL에 주목하는 걸까요? 그 이유는 대부분 성능 때문입니다. 관계형 데이터베이스는 시스템의 신뢰도를 높이는 데 필요한 장치를 많이 가지고 있습니다. 또 SQL문을 읽어 들이고 실행하는 데 많은 리소스를 사용하며 이 때문에 성능이 떨어지는 경우가 많습니다. 이에 반해 NoSQL 데이터베이스는 성능을 최우선으로 생각하기 때문에 실시간으로 처리해야 하는 경우나 대용량 트래픽을 감당할 수 있는 메시징 시스템 등에 활용됩니다. 특히 클라우드 서비스로 서버를 구성하는 경우가 많아지면서 많은 사용자를 수용하거나 시스템 자원을 적게 소모하는 NoSQL 데이터베이스에 점점 더 관심을 갖게 되었습니다.

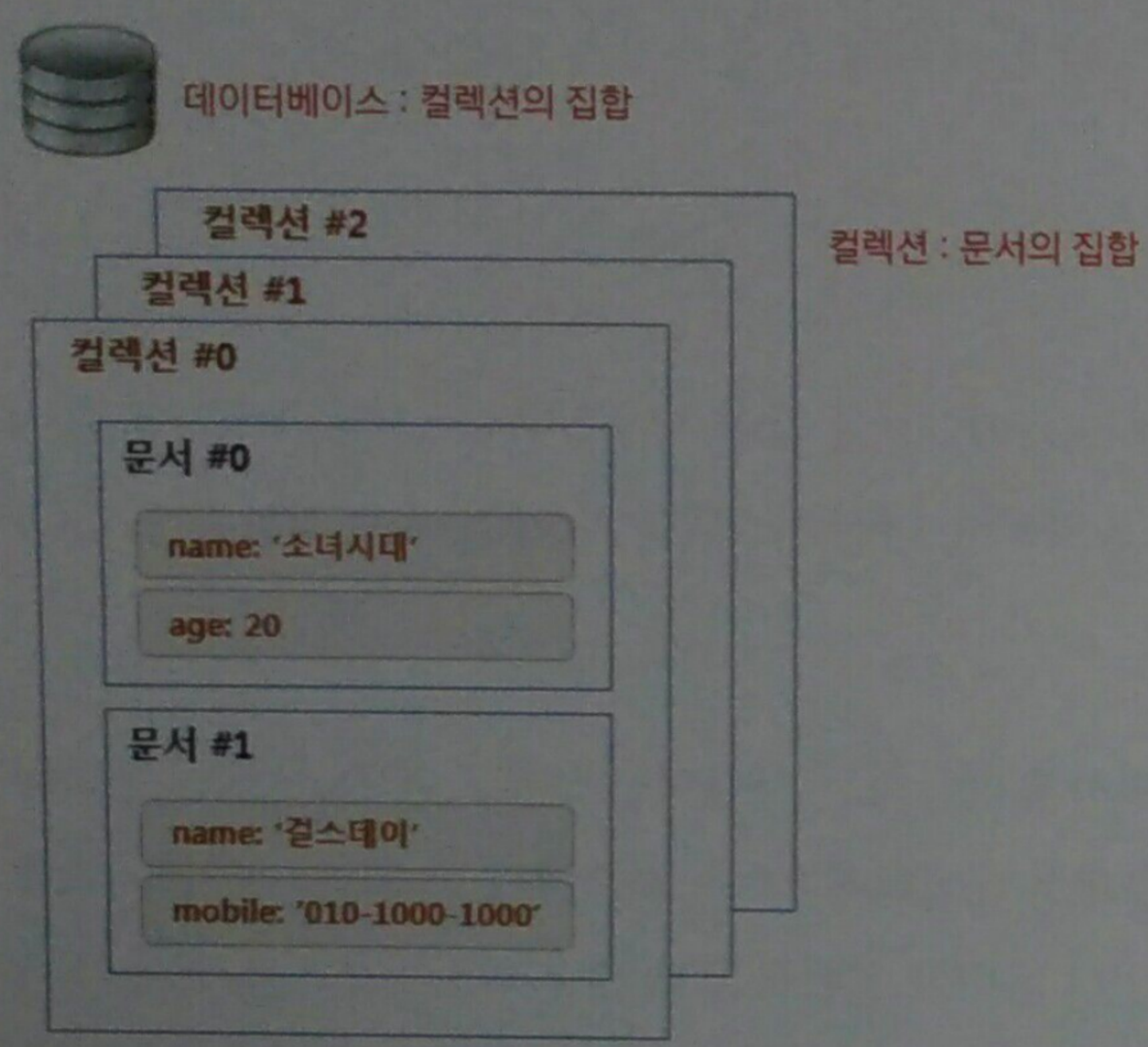


▲ NoSQL 데이터베이스의 사용

용어 **관계형 데이터베이스란** 데이터 항목들을 모아 놓은 집합체인 정형화된 테이블을 여러 개 포함하고 있으면서 다양한 방법으로 데이터에 접근하거나 조회할 수 있도록 만든 데이터 저장소를 말합니다. 관계형 데이터베이스는 데이터를 조작하거나 접근할 수 있도록 표준 검색 언어 SQL을 제공하며, SQL 문장은 관계형 데이터베이스의 데이터를 직접 조회하거나 보고서를 추출하는 데 주로 사용됩니다.

몽고디비는 NoSQL이기 때문에 관계형 데이터베이스의 테이블 개념이 없습니다. 그 대신 몽고디비는 여러 데이터가 모인 하나의 단위를 **컬렉션**(Collection)이라고 부릅니다. 관계형 데이터베이스의 테이블이 컬렉션이라고 단순하게 생각할 수도 있습니다. 하지만 테이블과 달리, 데이터를 정해 놓은 칼럼의 형태대로 컬렉션에 넣어야 한다는 제약은 없습니다.

다음은 몽고디비의 기본 구조입니다.



▲ 몽고디비의 기본 구조

몽고디비는 데이터 저장소를 가지고 있으며 그 안에 여러 개의 컬렉션을 넣을 수 있습니다. 즉, 데이터베이스는 **컬렉션의 집합**이라고 할 수 있습니다. 그리고 각각의 컬렉션은 여러 개의 **문서 객체**(Document)를 가질 수 있습니다. 문서 객체는 속성들의 집합으로서 한 사람의 이름과 나이 등을 저장하고 싶을 때 하나의 문서 객체를 만든 후 그 안에 자바스크립트 객체와 같이 속성들을 추가하여 저장할 수 있습니다.

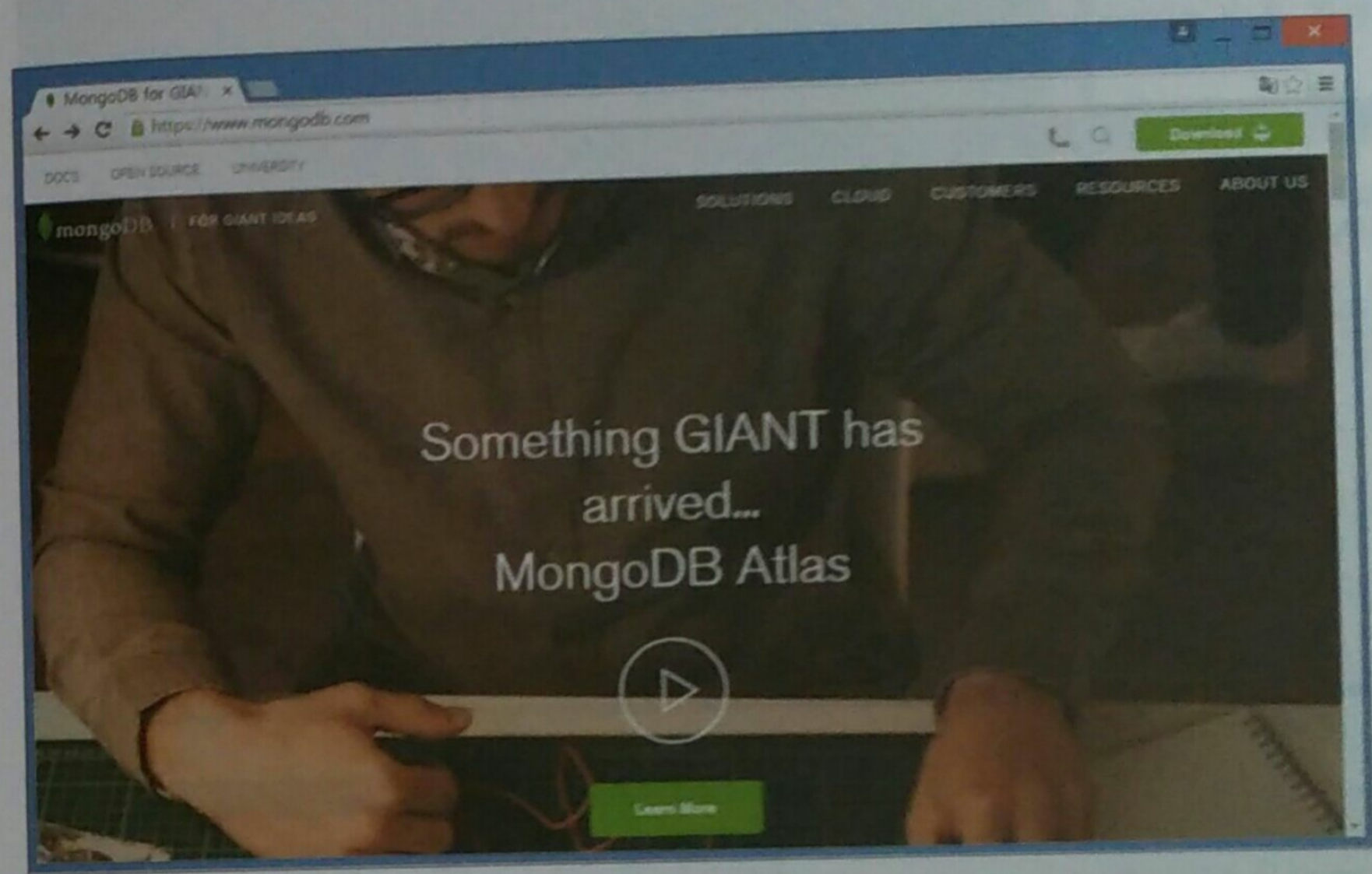
몽고디비 사용을 위한 프로그램 설치하기

몽고디비의 기본 구조를 관계형 데이터베이스의 구조와 비교해 보면 컬렉션은 테이블과 비슷하고, 그 안에 들어 있는 각각의 문서 객체는 테이블 안에 들어 있는 한 줄의 데이터인 **레코드**(Record)와 비슷합니다. 하지만 이 문서 객체들은 다른 문서 객체와 똑같은 속성을 가질 필요가 없어서 필요에 따라 완전히 다른 속성을 넣어 둘 수도 있습니다. 예를 들어, 한 문서 객체가 name과 age 속성을 가지고 있다면 다른 문서 객체는 name과 mobile 속성을 가질 수 있습니다.

몽고디비를 설치한 후 간단하게 데이터를 저장하고 조회해 보면 이러한 특성을 잘 이해할 수 있습니다.

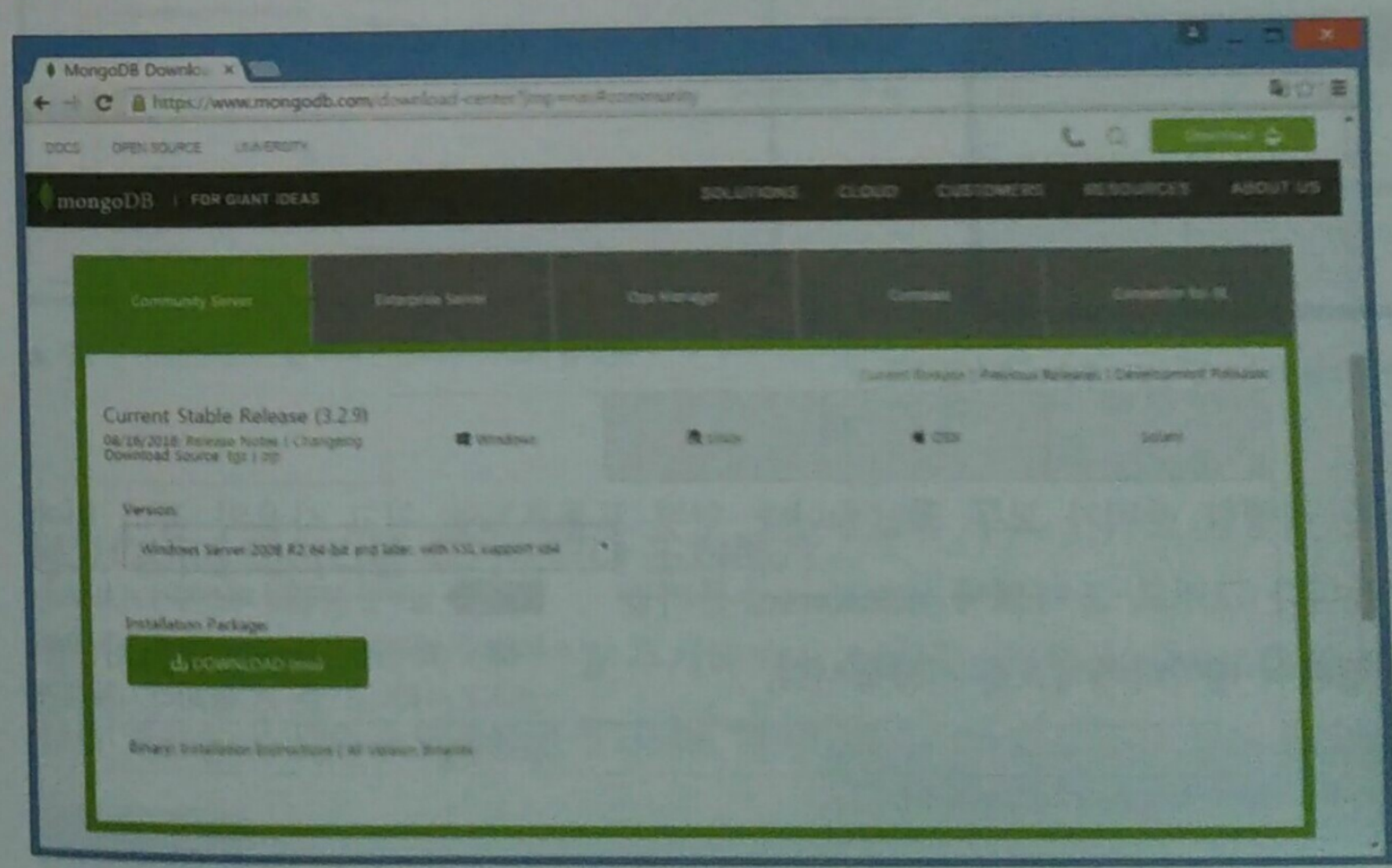
❶ 몽고디비를 사용하려면 먼저 설치 파일을 다운로드하여 설치해야 합니다. 다음 사이트에 접속하여 설치 파일을 다운로드한 후 설치를 진행합니다. 사이트에 접속한 후 오른쪽 위에 있는 [DOWNLOAD] 버튼을 클릭합니다.

▶ http://mongodb.org



◀ 몽고디비 사이트

❷ 다운로드 페이지가 나타나면 [Community Server] 탭 안에 있는 [DOWNLOAD] 버튼을 클릭합니다. 그러면 설치 파일을 다운로드할 수 있습니다. 윈도우용 설치 파일을 다운로드한 후 설치합니다.

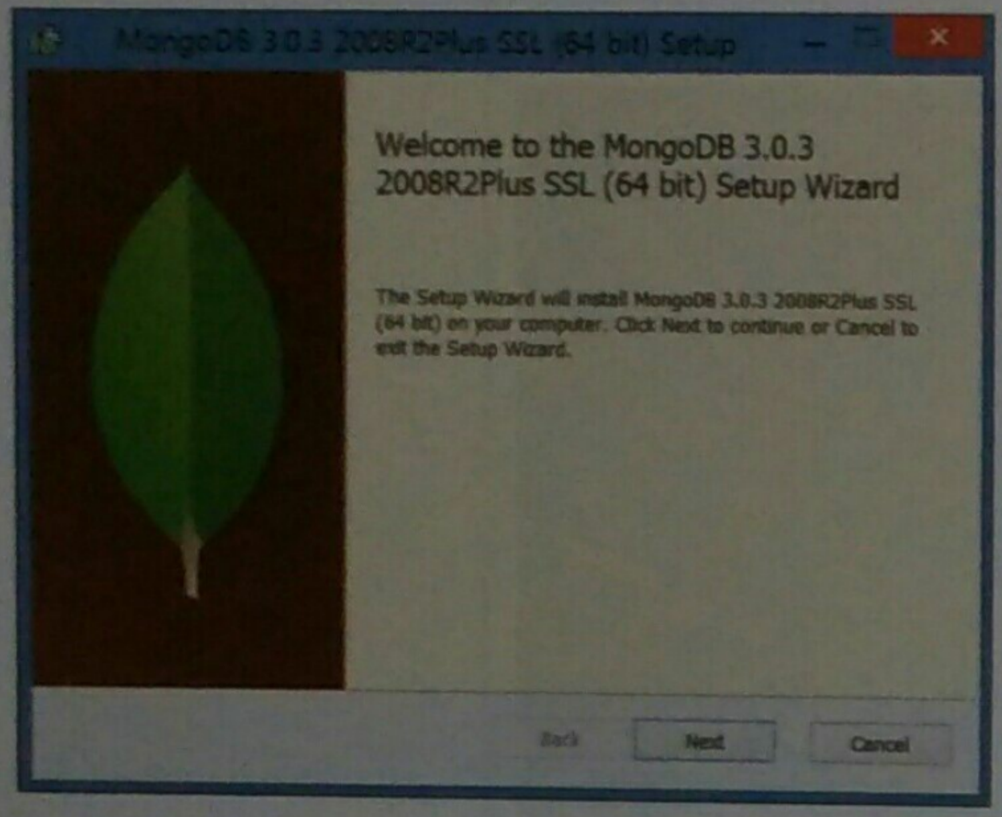


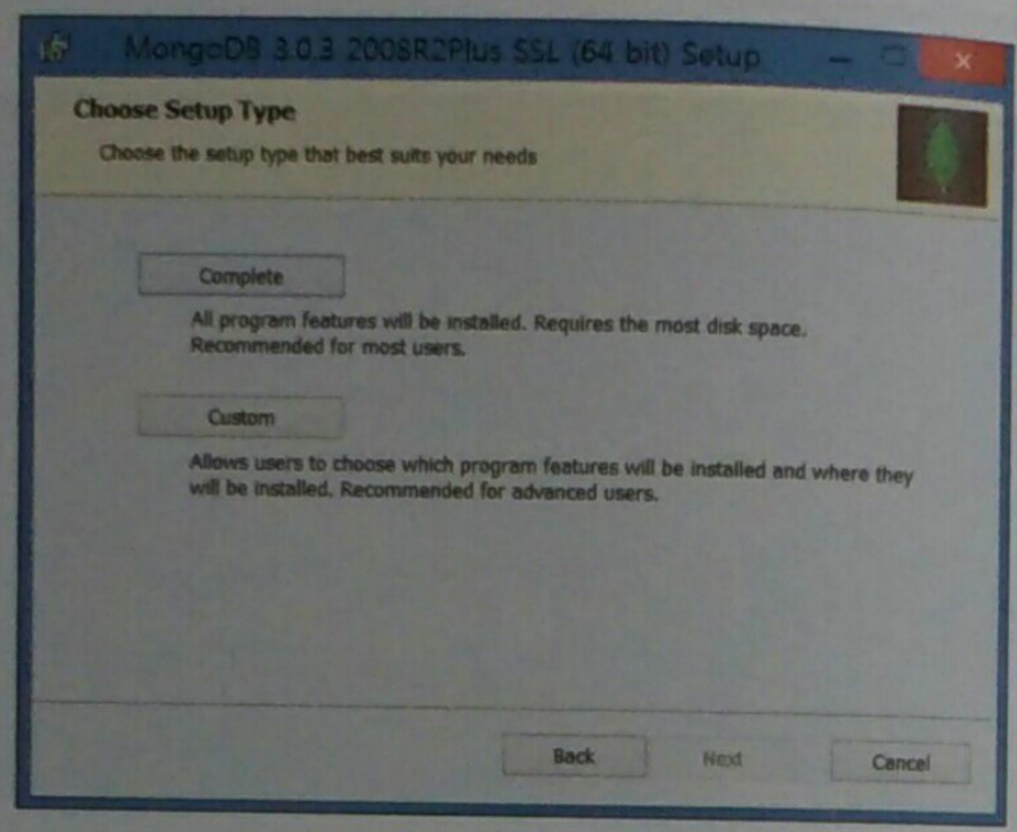
◀ 몽고디비 설치 파일 다운로드

정박사의 한마디 설치 시점에 따라 최신 버전이 달라질 수 있어요

설치 시점에 따라 버전은 달라집니다. 사이트에 접속했을 때 보이는 최신 버전의 설치 파일을 다운로드합니다.

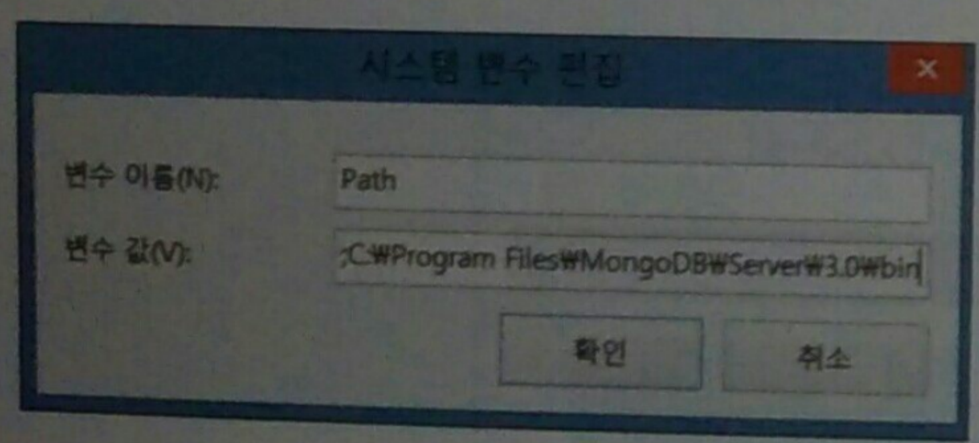
❸ 파일을 다운로드한 후 설치하는 과정은 일반 프로그램의 설치 과정과 비슷합니다. [Next] 버튼을 눌러 다음 화면으로 이동한 후 [Complete] 버튼을 눌러 모든 기능을 설치합니다.





▲ 몽고디비 설치 진행 화면

❹ 몽고디비 설치가 완료되면 설치 폴더 안에 있는 [bin] 폴더를 시스템 PATH에 추가해야 합니다. 윈도우에서는 보통 C:₩Program Files₩MongoDB₩Server₩3.0 폴더에 몽고디비가 설치되므로 다음과 같이 시스템 PATH에 추가합니다.



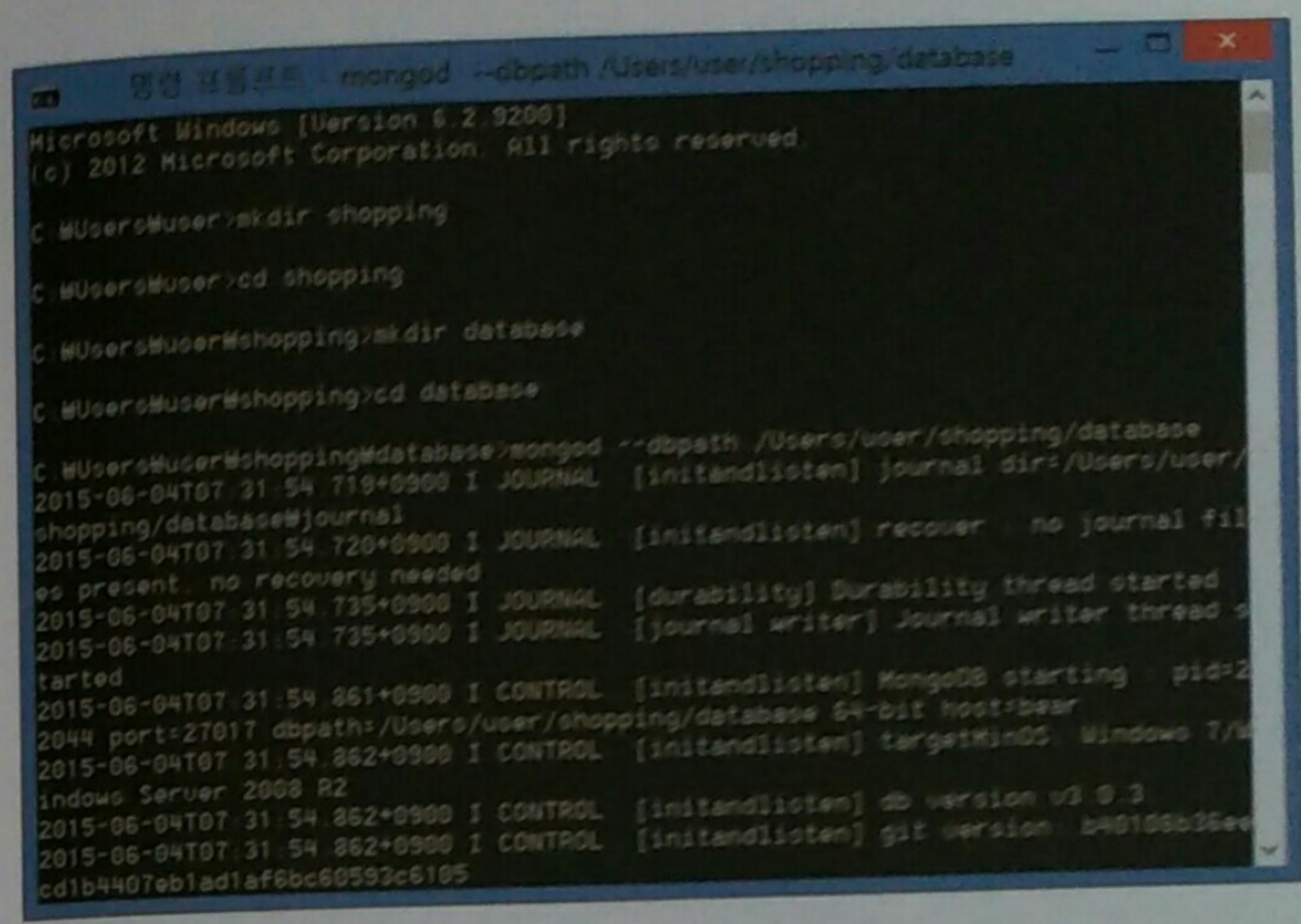
▲ 시스템 변수 중 PATH 환경 변수의 몽고디비 설치 폴더에 [bin] 폴더를 추가

주의! 폴더 이름에서 3.0은 버전을 의미하므로 설치한 버전에 따라 달라질 수 있습니다

❺ 이제 데이터베이스를 실행할 준비가 모두 끝났습니다. 명령 프롬프트를 열고 사용자 폴더 밑에 [shopping] 폴더를 만듭니다. 그리고 그 아래에 [database] 폴더를 만든 후 다음 명령을 실행하여 데이터베이스를 시작합니다.

```
% mongod --dbpath /Users/user/shopping/database
```

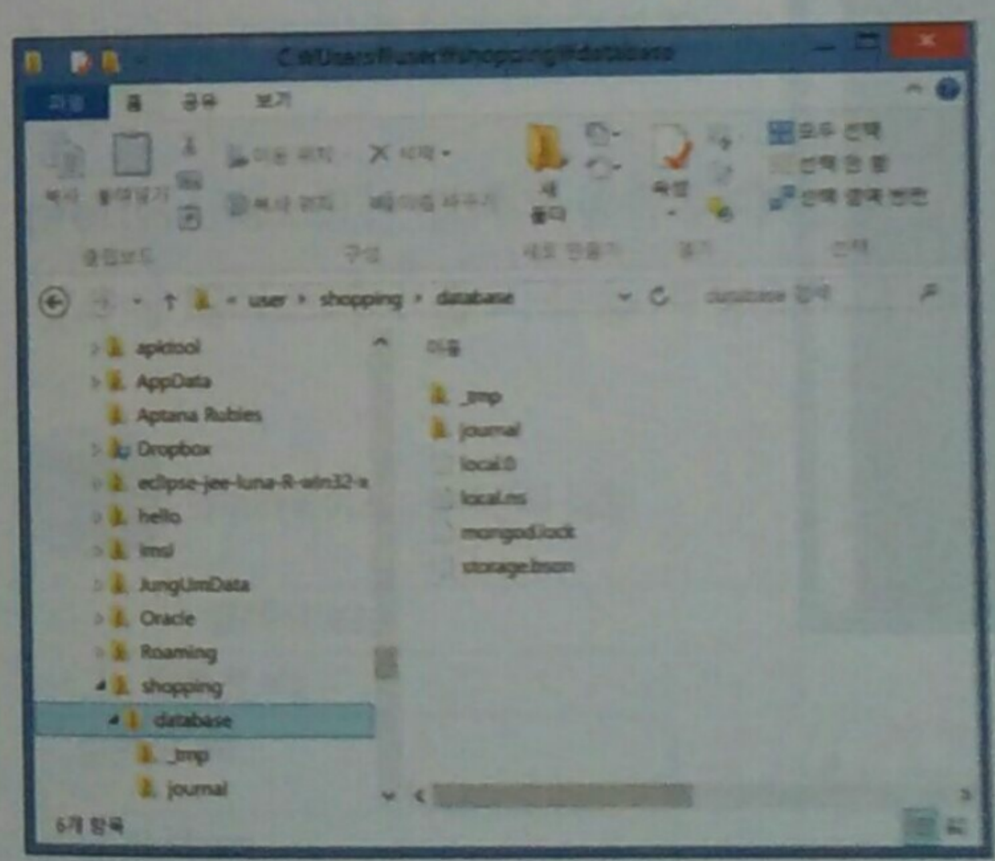
주의! 입력한 명령의 패스에서 /Users는 [사용자] 폴더이며, user는 사용자 계정 이름입니다. 사용하고 있는 PC에 따라 사용자 계정 이름을 다르게 입력해야 합니다.



▲ 명령 프롬프트에서 새로운 폴더를 만들고 데이터베이스를 실행한 화면

주의! 실행한 데이터베이스를 종료하고 싶나요? 데이터베이스를 실행했다가 종료하고 싶다면 Ctrl + C 키를 누르면 됩니다.

❻ 데이터베이스가 시작되면 27017 포트에서 연결을 기다린다는 메시지가 마지막에 표시됩니다. 파일 탐색기를 열고 [C:\Users\user\shopping\database] 폴더를 열어 보면 새로운 데이터베이스를 위해 여러 개의 파일과 폴더가 만들어졌음을 알 수 있습니다.



▲ 새로운 데이터베이스를 위해 만들어진 파일과 폴더들

몽고디비에 데이터를 추가하거나 조회하기

데이터베이스에 접속하여 데이터를 추가하거나 조회하는 방법을 알아보겠습니다. 먼저 명령 프롬프트 창을 열고 다음 명령을 입력하면 몽고디비 데이터베이스에 연결할 수 있는 셸(Shell) 상태로 들어갑니다.

```
% mongo
```

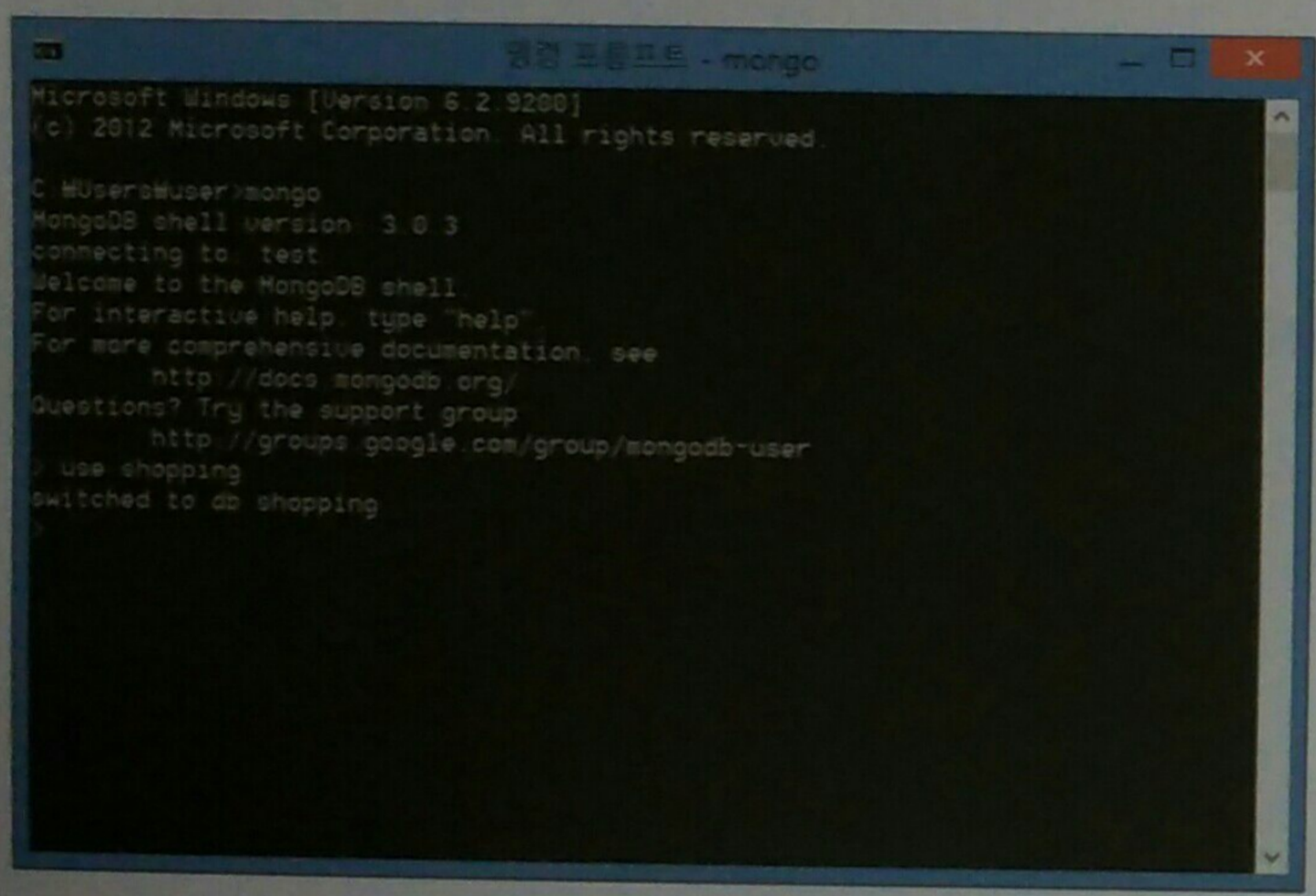
정박사의 한마디 셸이 무엇일까요?

셸이란 명령을 받아서 실행하는 프로그램입니다. 어떤 명령을 실행하면 명령을 받아들일 준비가 되는데 이 상태에서 명령을 실행하면 셸이 그 명령을 이해하고 처리합니다.

명령 프롬프트에서 이 명령을 실행하면 mongo 셸 상태로 전환됩니다. 이렇게 셸 상태로 전환되면 명령을 이용해서 데이터베이스를 조작할 수 있습니다. 다음과 같이 use 명령을 입력해서 데이터베이스를 지정합니다.

```
% use shopping
```

shopping 데이터베이스는 [shopping] 폴더 안의 [database] 폴더에 있습니다.



▲ use 명령으로 shopping 데이터베이스에 연결한 화면

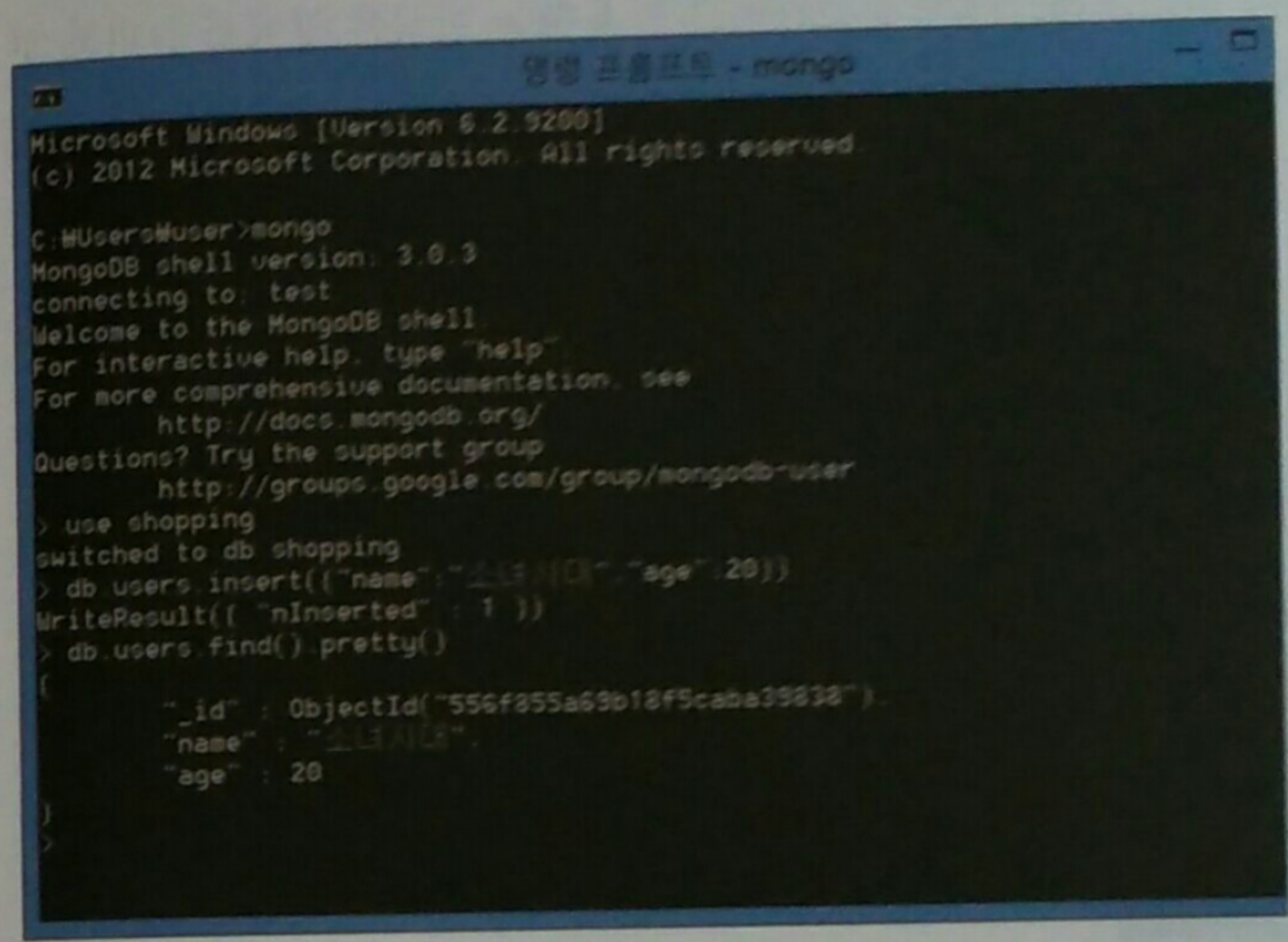
데이터베이스는 db라는 이름으로 접근할 수 있으며 데이터베이스 안에는 컬렉션을 만들고 그 안에 문서를 저장할 수 있습니다. 그런데 컬렉션을 별도로 만들지 않고 해당 컬렉션에 필요한 작업을 실행하도록 지정만 해도 자동으로 새로 만들어집니다. 예를 들어, 다음과 같이 insert 명령을 실행하면 users 컬렉션이 자동으로 만들어지며 그 안에 파라미터로 전달한 JSON 객체가 추가됩니다. 이 JSON 객체가 바로 문서 객체가 됩니다.

```
% db.users.insert({"name":"소녀시대","age":20})
```

이 명령을 보면 문서 객체를 추가할 때는 컬렉션 객체의 insert( ) 메소드를 사용하고, insert( ) 메소드에는 중괄호를 이용해 만든 JSON 객체를 파라미터로 전달한다는 것을 알 수 있습니다. 이렇게 만든 문서

객체는 find( ) 메소드를 이용해 특정 조건으로 찾아낼 수 있습니다. 다음 명령을 입력하면 users 컬렉션에 있는 모든 문서 객체들을 반환(Return)합니다.

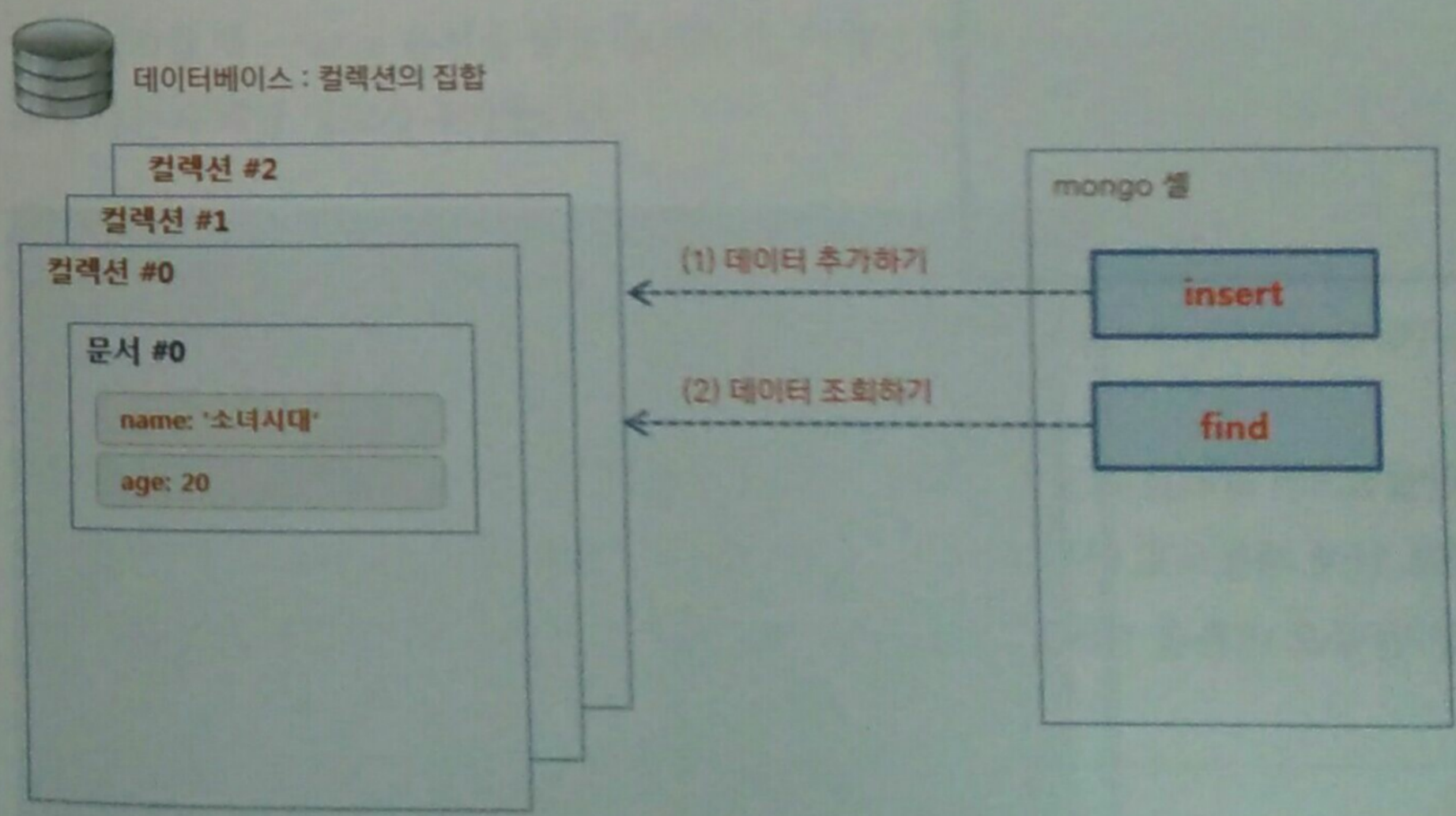
```
% db.users.find().pretty()
```



▲ 'users' 컬렉션에 문서를 추가한 후 조회한 경우

pretty( ) 메소드는 결과가 출력될 때 예쁘게 보이도록 조회된 결과에 띄어쓰기나 줄 바꿈을 자동으로 넣어 줍니다.

주의! 몽고디비에 데이터를 추가할 때는 insert( ), 데이터를 조회할 때는 find( ) 메소드를 사용한다는 것을 반드시 기억해 두세요.



▲ mongo 셸에서 데이터를 추가하고 조회하기

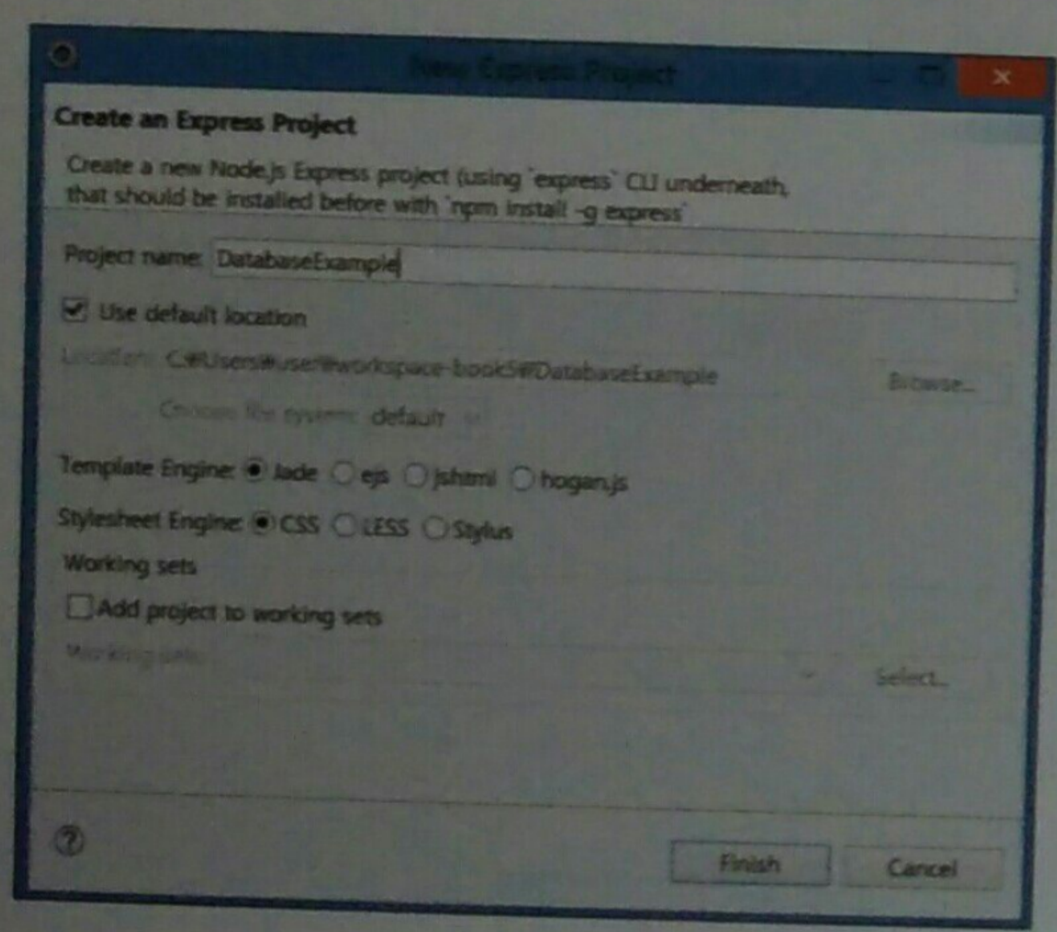
이렇듯 몽고디비에 데이터를 추가하거나 조회하는 것은 아주 쉽습니다. 데이터를 추가하는 것도 한 줄, 조회하는 것도 한 줄이면 가능합니다.

06-2 | 익스프레스에서 몽고디비 사용하기

익스프레스는 웹 서버를 만들 때 필요한 파일들을 자동으로 만들어 주는데 이런 웹 서버에서도 데이터베이스를 사용할 경우가 많습니다. 그런데 익스프레스에서 몽고디비를 사용하는 방식은 앞에서 살펴본 것과 약간 다릅니다. 특히 익스프레스가 적은 양의 코드로 빠르게 웹 서버를 만들 수 있는 것처럼 익스프레스 안에서 사용되는 몽고디비도 적은 양의 코드로 데이터를 저장하거나 조회할 수 있습니다.

새로운 프로젝트 만들기

이제 이클립스에서 새로운 익스프레스 프로젝트를 만들고 몽고디비 데이터베이스에 접근할 수 있는 코드를 만들어 보겠습니다. [File → New → Node.js Express Project] 메뉴를 눌러 새로운 프로젝트를 만들 수 있는 대화상자를 띄웁니다. 대화상자의 Project name 입력 창에 DatabaseExample을 입력한 후 [Finish] 버튼을 누릅니다.



▲ 이클립스에서 새로운 프로젝트 DatabaseExample 만들기

프로젝트를 새로 만들었으면 app.js 파일을 열고 앞 장에서 사용한 코드를 다시 입력합니다. 앞 장에서 익스프레스 프로젝트 안에 자동으로 만들어진 코드를 하나하나 살펴보았지만, 데이터베이스 기능을 추가하기 전에 다시 한 번 주요 내용을 정리해 보겠습니다. 코드의 주요 부분을 살펴보면 다음과 같습니다.

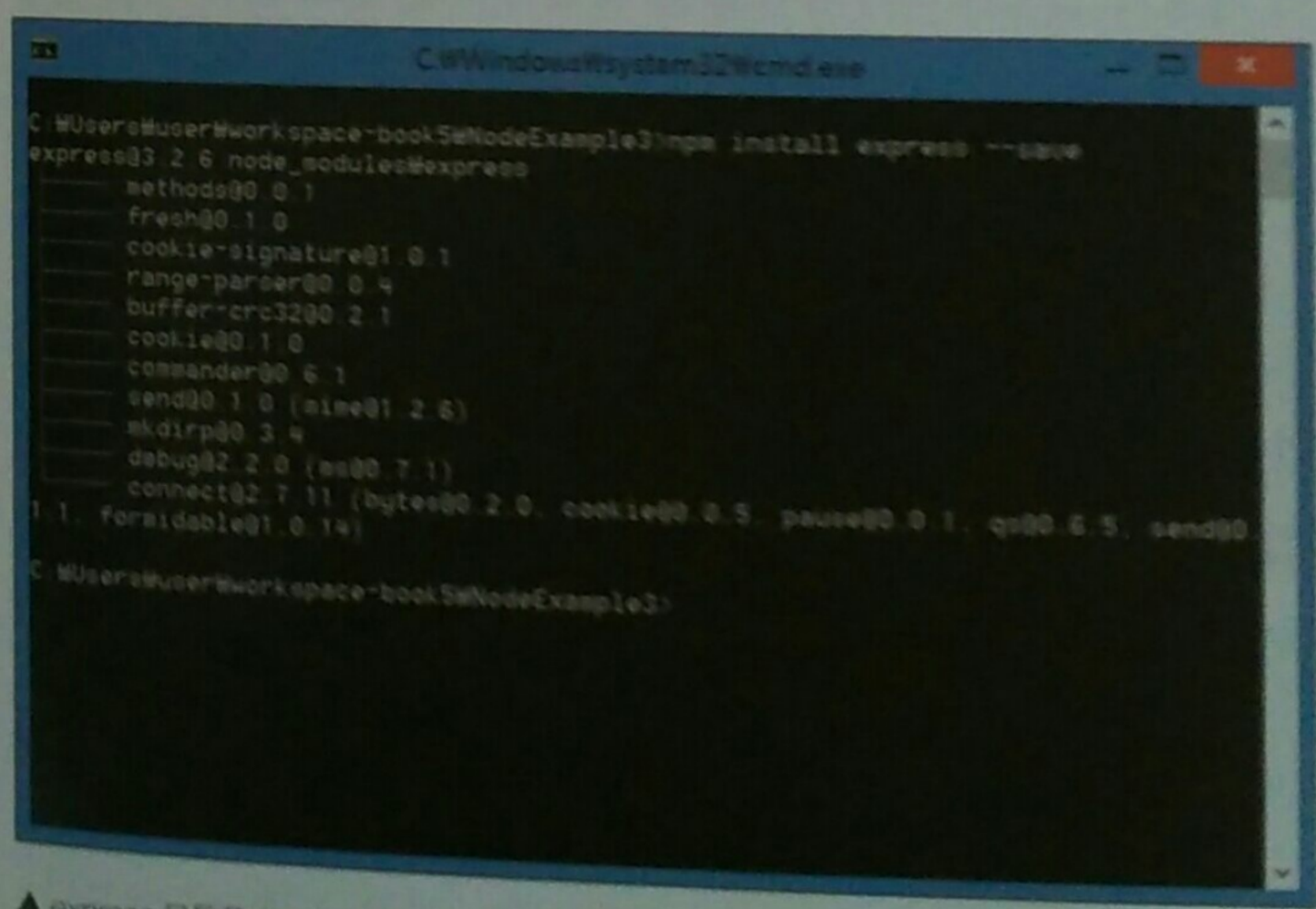
참조 파일 | DatabaseExample>app.js

```
//===== 모듈 불러들이기 =====//
var express = require('express')
  , http = require('http')
  , path = require('path');
```

```
var bodyParser = require('body-parser');
var cookieParser = require('cookie-parser');
var expressSession = require('express-session');
var expressErrorHandler = require('express-error-handler');



//===== Express 서버 객체 만들기 =====//
var app = express();

.....
```

코드의 첫 부분에서는 필요한 모듈을 불러들이고 express( ) 메소드를 호출하여 익스프레스 서버 객체를 만듭니다. 익스프레스에서 사용하는 기본 모듈인 express 모듈, http 모듈과 함께 패스 문자열을 다루는 path 모듈, POST 요청 처리를 다루는 body-parser 모듈, 쿠키와 세션을 다루는 cookie-parser와 express-session 모듈을 불러들입니다. 마지막으로 오류 처리를 위해 사용했던 express-error-handler 모듈도 불러들입니다.

이 부분까지 확인한 후 app.js 파일을 실행합니다. app.js 파일을 선택한 상태에서 마우스 오른쪽 버튼을 클릭한 뒤 [Node Application] 메뉴를 클릭하면 실행할 수 있습니다. 그런데 실행했을 때 모듈을 찾을 수 없다는 오류가 발생할 수도 있습니다. 이럴 때에는 명령 프롬프트에서 모듈을 설치해야 합니다. 모듈을 설치할 때 --save 옵션을 붙일 수 있는데, 이렇게 하면 package.json 파일 안에 어떤 모듈을 설치해야 하는지 해당 정보가 추가됩니다.



▲ express 모듈을 설치할 때 --save 옵션을 붙인 경우

필요한 모듈을 설치할 때마다 --save 옵션을 붙였다면 최종적으로 다음과 같은 package.json 파일이 만들어집니다.

참조 파일 | DatabaseExample/package.ison

```
{
  "name": "DatabaseExample",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "body-parser": "^1.12.4",
    "cookie-parser": "^1.3.5",
    "express": "^3.2.6",
    "express-error-handler": "^1.0.1",
    "express-session": "^1.11.2",
    "http": "0.0.0",
    "jade": "*",
    "path": "^0.11.14"
  }
}
```

이 파일에 포함된 버전 정보는 설치 시점에 따라 달라질 수 있으니 주의하세요. 하지만 사용한 모듈의 이름은 같습니다. 이렇게 만든 package.json 파일의 dependencies 정보는 새로운 프로젝트를 만들었을 때 그대로 사용할 수 있습니다. 또 이 정보가 있으면 각각의 모듈을 따로 설치할 필요 없이 다음 명령만으로 자동 설치됩니다.

```
% npm install
```

npm 프로그램은 package.json 파일의 dependencies 정보를 확인한 후 필요한 모듈을 알아서 설치합니다. 이렇게 설치한 모듈들은 [node_modules] 폴더 안에 들어갑니다.

주의! [node_modules] 폴더가 프로젝트 폴더보다 상위에 있을 경우에 프로젝트 폴더에 직접 모듈을 설치하지 않아도 참조하여 사용할 수 있습니다. 이 점은 다시 한 번 기억해 두기 바랍니다.

app.js 파일에 들어갈 기본 코드를 좀 더 살펴보겠습니다. 다음은 모듈을 불러온 후 초기화하는 부분입니다.

```
……

//====== 서버 변수 설정 및 static으로 public 폴더 설정 ======//
app.set('port', process.env.PORT || 3000);
app.use('/public', express.static(path.join(__dirname, 'public')));

//====== body-parser, cookie-parser, express-session 사용 설정 ======//
app.use(bodyParser.urlencoded({extended: true}));

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

……
```

서버에서 사용할 포트 정보는 set( ) 메소드를 이용해서 port라는 이름으로 설정합니다. 그리고 [/public] 폴더를 특정 패스로 접근할 수 있도록 static 미들웨어를 사용해 등록합니다. 그다음에는 body-parser, cookie-parser, express-session 모듈을 사용할 수 있도록 use( ) 메소드를 호출합니다. 그 아래 코드에서 라우터 미들웨어를 사용하면 이제부터 사용자가 요청한 패스 정보에 따라 라우팅이 가능해집니다.


```
……

//===== 404 오류 페이지 처리 =====//
var errorHandler = expressErrorHandler({
 static: {
   '404': './public/404.html'
 }
});

app.use(expressErrorHandler.httpError(404));
app.use(errorHandler);


//====== 서버 시작 ======//
http.createServer(app).listen(app.get('port'), function() {
  console.log('서버가 시작되었습니다. 포트 : ' + app.get('port'));
});
```

필요한 모듈을 설치할 때마다 --save 옵션을 붙였다면 최종적으로 다음과 같은 package.json 파일이 만들어집니다.

참조 파일 | DatabaseExample>package.json

```
{
  "name": "DatabaseExample",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "body-parser": "^1.12.4",
    "cookie-parser": "^1.3.5",
    "express": "^3.2.6",
    "express-error-handler": "^1.0.1",
    "express-session": "^1.11.2",
    "http": "0.0.0",
    "jade": "*",
    "path": "^0.11.14"
  }
}
```

이 파일에 포함된 버전 정보는 설치 시점에 따라 달라질 수 있으니 주의하세요. 하지만 사용한 모듈의 이름은 같습니다. 이렇게 만든 package.json 파일의 dependencies 정보는 새로운 프로젝트를 만들었을 때 그대로 사용할 수 있습니다. 또 이 정보가 있으면 각각의 모듈을 따로 설치할 필요 없이 다음 명령만으로 자동 설치됩니다.

```
% npm install
```

npm 프로그램은 package.json 파일의 dependencies 정보를 확인한 후 필요한 모듈을 알아서 설치합니다. 이렇게 설치한 모듈들은 [node_modules] 폴더 안에 들어갑니다.

주의! [node_modules] 폴더가 프로젝트 폴더보다 상위에 있을 경우에 프로젝트 폴더에 직접 모듈을 설치하지 않아도 참조하여 사용할 수 있습니다. 이 점은 다시 한 번 기억해 두기 바랍니다.

app.js 파일에 들어갈 기본 코드를 좀 더 살펴보겠습니다. 다음은 모듈을 불러온 후 초기화하는 부분입니다.

참조 파일 | DatabaseExample>app.js

```
……,

//===== 서버 변수 설정 및 static으로 public 폴더 설정 =====//
app.set('port', process.env.PORT || 3000);
app.use('/public', express.static(path.join(__dirname, 'public')));

//===== body-parser, cookie-parser, express-session 사용 설정 =====//
app.use(bodyParser.urlencoded({extended: true}));

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

……
```

서버에서 사용할 포트 정보는 set( ) 메소드를 이용해서 port라는 이름으로 설정합니다. 그리고 [/public] 폴더를 특정 패스로 접근할 수 있도록 static 미들웨어를 사용해 등록합니다. 그다음에는 body-parser, cookie-parser, express-session 모듈을 사용할 수 있도록 use( ) 메소드를 호출합니다. 그 아래 코드에서 라우터 미들웨어를 사용하면 이제부터 사용자가 요청한 패스 정보에 따라 라우팅이 가능해집니다.

참조 파일 | DatabaseExample>app.js

```
……,

//===== 404 오류 페이지 처리 =====//
var errorHandler = expressErrorHandler({
 static: {
   '404': './public/404.html'
 }
});

app.use(expressErrorHandler.httpError(404));
app.use(errorHandler);


//===== 서버 시작 =====//
http.createServer(app).listen(app.get('port'), function() {
  console.log('서버가 시작되었습니다. 포트 : ' + app.get('port'));
});
```

가장 아래쪽에는 서버를 실행하는 코드가 있으며, 그 위에는 오류 처리에 필요한 express-error-handler 모듈을 사용할 수 있게 하는 코드가 추가되어 있습니다. 앞 장에서 만든 코드가 머릿속에 정리되었다면, 이제 사용자가 로그인했을 때 데이터베이스에 저장된 사용자 정보와 일치하는지 확인하는 기능을 만들어 보겠습니다.

mongodb 모듈을 사용하여 로그인 기능 만들기

먼저 사용자가 로그인하는 데 사용할 웹 문서를 [public] 폴더 안에 만듭니다. 앞에서 만든 Express Example 프로젝트의 [public] 폴더 안에 있는 login2.html 파일과 같은 것을 사용할 것이므로 그 파일을 복사하여 DatabaseExample 프로젝트의 [public] 폴더에 넣은 후 이름을 login.html로 변경해도 됩니다.

참조 파일 | DatabaseExample>/public/login.html

```
.....

<form method="post" action="/process/login">
  <table>
    <tr>
      <td><label>아이디</label></td>
      <td><input type="text" name="id"></td>
    </tr>
    <tr>
      <td><label>비밀번호</label></td>
      <td><input type="password" name="password"></td>
    </tr>
  </table>
  <input type="submit" value="전송" name="">
</form>

.....
```

사용자는 웹 브라우저를 열고 http://localhost:3000/public/login.html 주소를 입력하면 로그인 페이지를 볼 수 있습니다. 로그인 페이지에서 아이디와 비밀번호를 입력한 후 웹 서버에 요청하면 사용자의 아이디와 비밀번호는 서버 쪽에 요청 파라미터로 전달됩니다. 요청 파라미터로 전달받은 데이터는 데이터베이스에 저장된 사용자 아이디와 비밀번호를 비교하여 두 값이 일치하는지 확인하는 데 사용될 것입니다. 이전에 명령 프롬프트에서 데이터베이스에 추가한 데이터에는 아이디와 비밀번호 정보가 없습니다. 따라서 다음 명령을 차례대로 입력하여 기존 문서 객체를 없애고 아이디와 비밀번호를 속성으로 포함하는 새로운 문서 객체를 추가합니다.

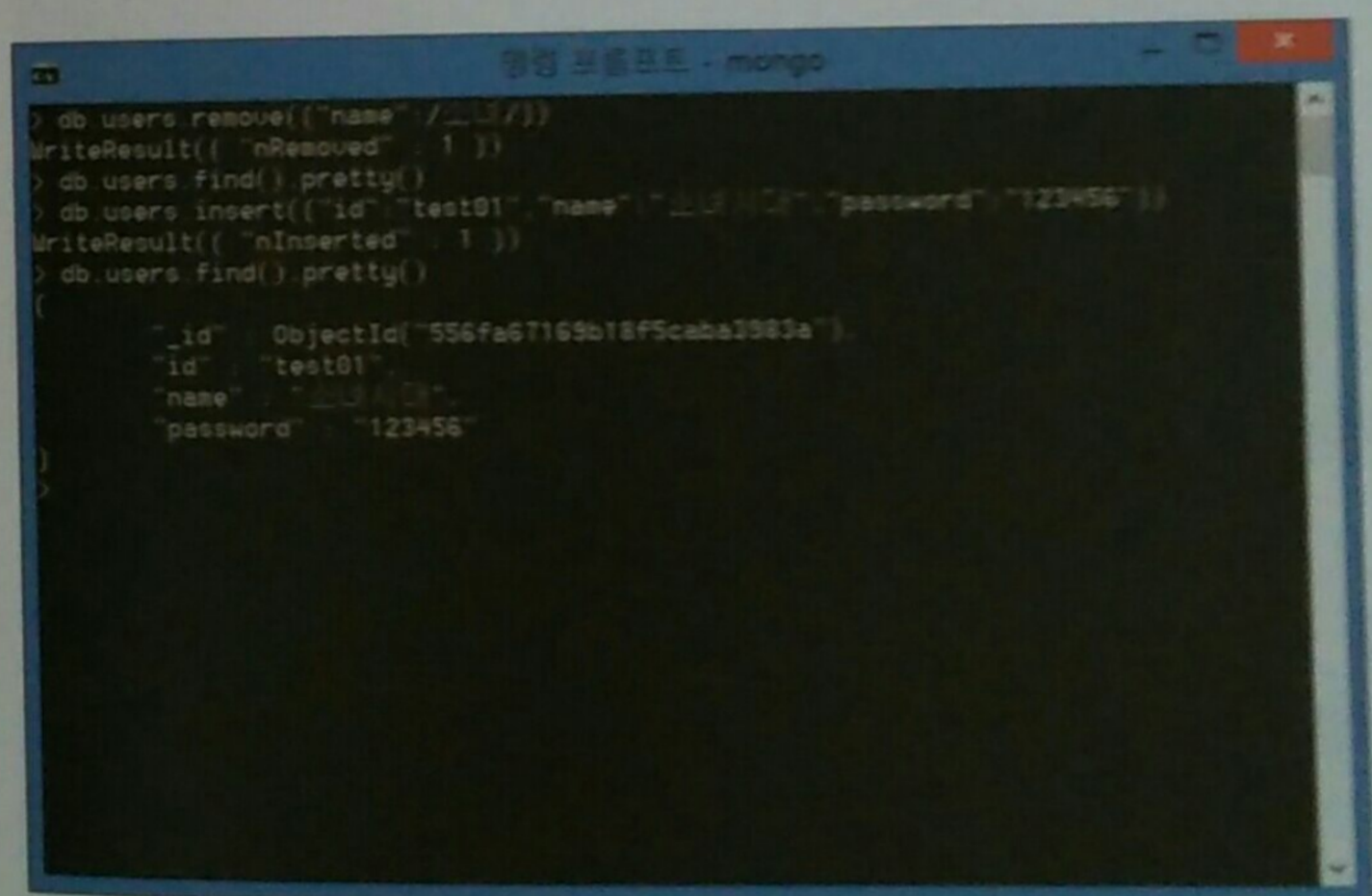
```
% cls
% db.users.remove({"name":/소녀/})
% db.users.find().pretty()
% db.users.insert({"id" : "test01" , "name" : "소녀시대", "password" : "123456"})
% db.users.find().pretty()
```

cls 명령은 명령 프롬프트 화면에 출력된 정보를 화면에서 지워 줍니다. 컬렉션 객체의 remove( ) 메소드를 사용하면 기존 데이터를 삭제할 수 있는데 이때 파라미터로 전달되는 JSON 객체에는 **regexp 표현식**으로 표시 조건을 넣을 수 있습니다. 예를 들어, {"name":/소녀/} 객체를 전달하면 name 속성의 값에 소녀라는 단어가 들어간 모든 데이터를 찾아낸 후 삭제합니다.

주의! 여기에서 /소녀/라는 regexp 표현식에는 큰따옴표나 작은따옴표를 붙이지 않는다는 점에 주의하세요.

용어 **regexp 표현식이란 무엇일까요?** regexp는 정규 표현식(Regular Expression)의 약자인데 특정한 규칙을 가진 문자열의 집합을 말합니다. 다시 말해, 일정한 규칙을 정해 둔 글자들이라고 할 수 있는데 이 글자들을 사용해 표현한 조건에 맞는 문자열을 검색하거나 치환합니다. 예를 들어, ^x라고 표현된 정규 표현식은 x로 시작하는 문자열을 의미합니다.

데이터를 삭제한 후 find( ) 메소드로 데이터를 조회하면 모든 데이터가 삭제되었기 때문에 어떤 데이터도 조회되지 않습니다. 데이터가 삭제된 것을 확인한 다음에는 insert( ) 메소드로 데이터를 새로 추가합니다. 새로 추가하는 문서 객체에는 id, name, password 속성을 포함시킵니다.



▲ 기존 문서 객체를 삭제한 후 id, name, password를 포함한 새로운 문서 객체 추가

문서 객체에 id와 password 속성을 추가했으므로 app.js 파일에 로그인을 처리하는 코드를 추가할 수 있습니다. 앞에서는 mongo 셸을 사용하여 직접 명령을 입력했지만, 소스 코드에서는 셸을 사용하지 않고 직접 몽고디비 데이터베이스에 연결해야 하므로 mongodb 모듈을 사용합니다. 먼저 명령 프롬프트에서 mongodb 모듈을 설치합니다.

```
% npm install mongodb --save
```

app.js 파일에서 라우팅 코드 윗 부분에 다음 코드를 추가합니다.

참조 파일 | DatabaseExample>app.js

```
.....

var mongodb = require('mongodb');

//===== 데이터베이스 연결 =====//
var database;

// 데이터베이스에 연결하고 응답 객체의 속성으로 db 객체 추가
function connectDB() {
  // 데이터베이스 연결 정보
  var databaseUrl = 'mongodb://localhost:27017/shopping';

  // 데이터베이스 연결
  mongodb.connect(databaseUrl, function(err, db) {
    if(err) throw err;

    console.log('데이터베이스에 연결되었습니다. : ' + databaseUrl);

    // database 변수에 할당
    database = db;
  });
}

.....
```

connectDB 함수는 mongodb 모듈을 사용해 몽고디비 데이터베이스에 연결하도록 새로 만든 것입니다. 데이터베이스에 연결하려면 먼저 연결 정보를 문자열로 정의해야 합니다. 문자열의 형식은 다음과 같습니다.

```
mongodb://%IP정보%:%포트정보%/%데이터베이스이름%
```

앞뒤에 %기호가 붙은 것은 여러분이 지정해야 하는 정보라는 의미입니다. 앞의 코드에서는 로컬 PC의 27017포트에서 실행되고 있는 shopping 데이터베이스에 연결하도록 연결 정보를 만들었습니다.

데이터베이스에 연결하려면 mongodb 객체의 connect( ) 메소드를 호출합니다. 첫 번째 파라미터로 연결 URL을 전달하고, 두 번째 파라미터로 콜백 함수를 전달합니다. 데이터베이스가 정상적으로 연결되면 db 객체가 전달되는데 이 객체는 database 변수에 저장합니다. 서버가 실행되면 connectDB 함수를 호출하여 데이터베이스에 연결합니다. 다음과 같이 서버가 시작된 후 호출되는 콜백 함수 안에서 connectDB 함수를 호출합니다.

참조 파일 | DatabaseExample>app.js

```
……,

//===== 서버 시작 =====//
http.createServer(app).listen(app.get('port'), function() {
  console.log('서버가 시작되었습니다. 포트 : ' + app.get('port'));

  // 데이터베이스 연결
  connectDB();

});
```

사용자가 보내온 아이디와 비밀번호 비교하기

이제 데이터베이스에 연결되었고 database 변수에 저장된 db 객체를 사용해 데이터베이스를 사용할 수 있는 상태가 되었습니다. 그러면 사용자가 보내온 아이디와 비밀번호를 사용해 데이터베이스 안에 일치하는 문서 객체가 있는지 조회합니다. 이 기능을 담당하는 **authUser 함수**를 다음과 같이 만듭니다.

참조 파일 | DatabaseExample>app.js

```
……,

// 사용자를 인증하는 함수
var authUser = function(database, id, password, callback) {
  console.log('authUser 호출됨.');

  // users 컬렉션 참조
  var users = database.collection('users');
```

```
// 아이디와 비밀번호를 사용해 검색
users.find({"id" : id, "password" : password}).toArray(function(err, docs) {
  if(err) {
    callback(err, null);
    return;
  }

  if(docs.length > 0) {
    console.log('아이디 [%s], 비밀번호 [%s]가 일치하는 사용자 찾음.', id, password);
    callback(null, docs);
  } else {
    console.log("일치하는 사용자를 찾지 못함.");
    callback(null, null);
  }
});
}

.....
```

authUser( ) 함수는 데이터베이스 객체, 아이디, 비밀번호를 파라미터로 받으며 콜백 함수도 파라미터로 전달받습니다. 데이터베이스의 users 컬렉션 안에 사용자 정보가 들어 있으므로 database.collection( ) 메소드로 users 컬렉션 객체를 참조합니다. 그다음에는 users 컬렉션 객체의 find( ) 메소드를 호출하여 데이터를 조회합니다. id와 password 속성의 값을 파라미터로 전달하여 조회한 후 조회 결과는 toArray( ) 메소드를 사용해서 배열 객체로 변환합니다. toArray( ) 메소드의 파라미터로 전달되는 콜백 함수에는 toArray( )로 변환된 문서 객체가 전달됩니다. docs의 값이 0보다 크면 사용자 데이터가 조회된 것이므로 callback 함수를 호출하면서 docs 객체를 파라미터로 전달합니다. 이렇게 하면 이 함수를 호출한 쪽에서 콜백 함수를 사용해 docs 객체를 전달받을 수 있습니다.

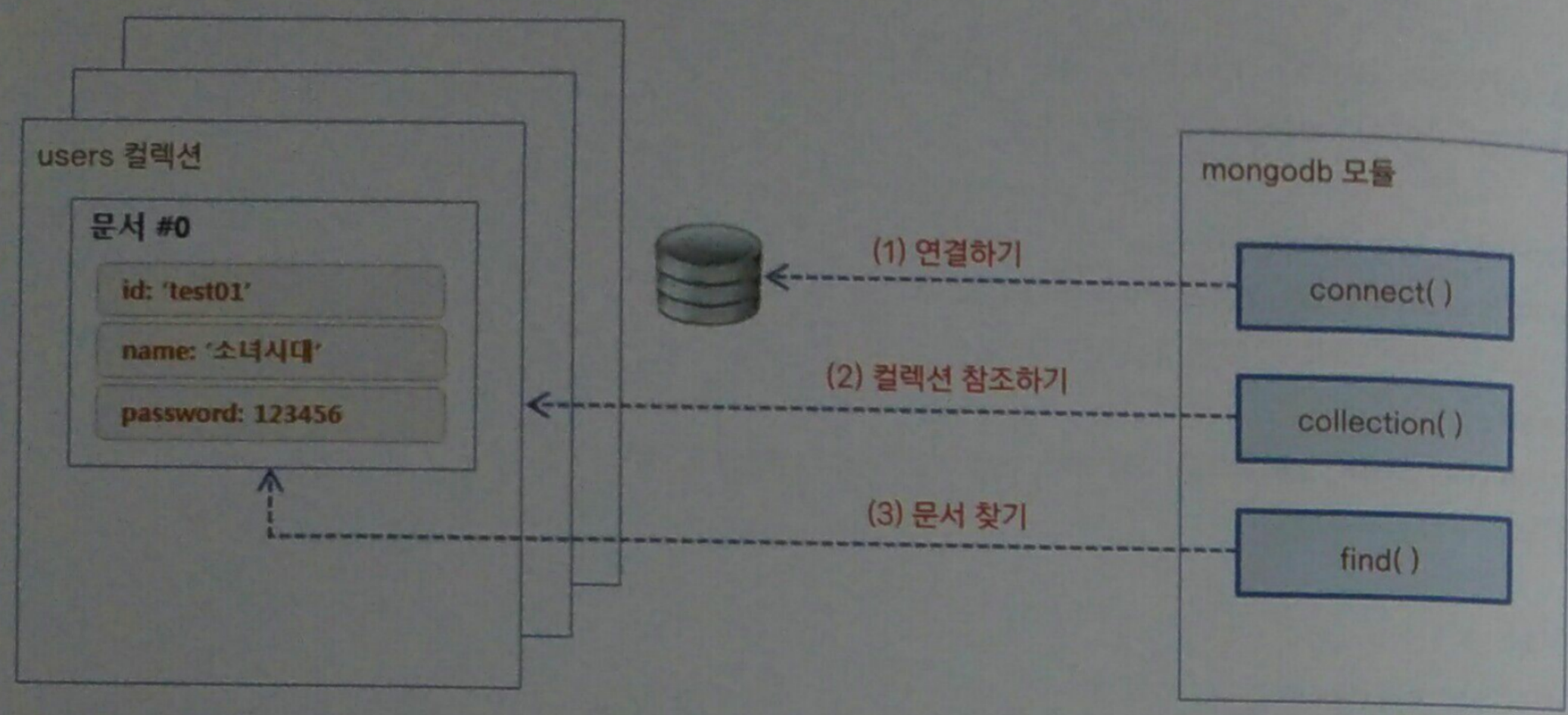
로그인 처리를 요청하는 패스에 라우팅 함수 추가하기

이제 로그인 처리를 요청하는 /process/login 패스에 라우팅 함수를 추가할 때가 되었습니다. 다음과 같이 사용자가 있는지 확인한 후 그에 맞는 응답을 보내 주는 코드를 입력합니다.

```
……,

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

  var paramId = req.param('id');
  var paramPassword = req.param('password');

  if(database) {
    authUser(database, paramId, paramPassword, function(err, docs) {
      if(err) {throw err;}

      if(docs) {
        console.dir(docs);

        res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
        res.write('<h1>로그인 성공</h1>');
        res.write('<div><p>사용자 아이디 : ' + paramId + '</p></div>');
        res.write('<div><p>사용자 이름 : ' + username + '</p></div>');
        res.write("<br><br><a href='/public/login.html'>다시 로그인하기</a>");
        res.end();
      } else {
        res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
        res.write('<h1>로그인  실패</h1>');
        res.write('<div><p>아이디와 비밀번호를 다시 확인하십시오.</p></div>');
        res.write("<br><br><a href='/public/login.html'>다시 로그인하기</a>");
        res.end();
      }
    });
  } else {
    res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
    res.write('<h2>데이터베이스 연결 실패</h2>');
    res.write('<div><p>데이터베이스에 연결하지 못했습니다.</p></div>');
    res.end();
  }

});

……,
```

사용자가 요청한 id와 password 파라미터를 확인한 후 authUser( ) 메소드를 호출하여 전달합니다. 사용자가 조회되면 docs 문서 객체가 콜백 함수의 파라미터로 전달되므로 docs 문서 객체의 첫 번째 배

열 요소를 참조한 후 사용자 이름을 확인합니다. 지금까지 만든 것은 웹 서버에서 몽고디비에 연결한 후 컬렉션을 참조하고 문서 객체를 찾아내는 과정입니다. 이 과정을 순서대로 그려 보면 다음과 같습니다.



▲ 몽고디비에 연결한 후 문서 객체를 찾아내는 과정

이 처리 과정을 보면 mongodb 모듈로 몽고디비 데이터베이스에 연결하고 문서 객체를 찾는 과정을 쉽게 이해할 수 있을 것입니다. 이 과정을 생각하면서 코드 형태를 다시 한 번 살펴보기 바랍니다.

이제 app.js 파일을 실행한 후 웹 브라우저에서 다음 주소로 요청합니다.

```
▶ http://localhost:3000/public/login.html
```

정박사의 한마디 bson 모듈과 관련된 오류가 보이나요?

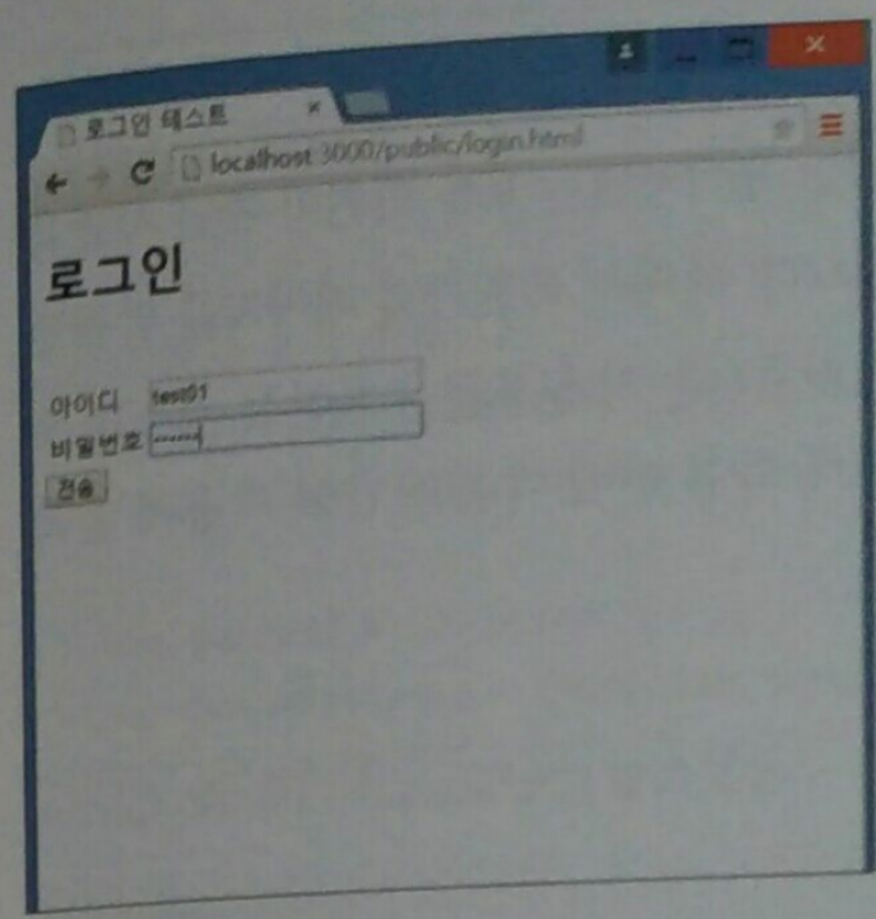
app.js 파일을 실행했을 때 다음 오류 메시지가 나타나는 경우도 있습니다.

```
js-bson: Failed to load c++ bson extension, using pure JS version
{ [Error: Cannot find module '../build/Release/bson'] code: 'MODULE_NOT_FOUND' }
```

이것은 mongodb 모듈 안에 포함된 bson 모듈의 위치를 제대로 인식하지 못했기 때문입니다. 다음과 같이 첫 번째 폴더에 있는 파일들을 두 번째 폴더를 만들어서 복사하여 넣으면 됩니다.

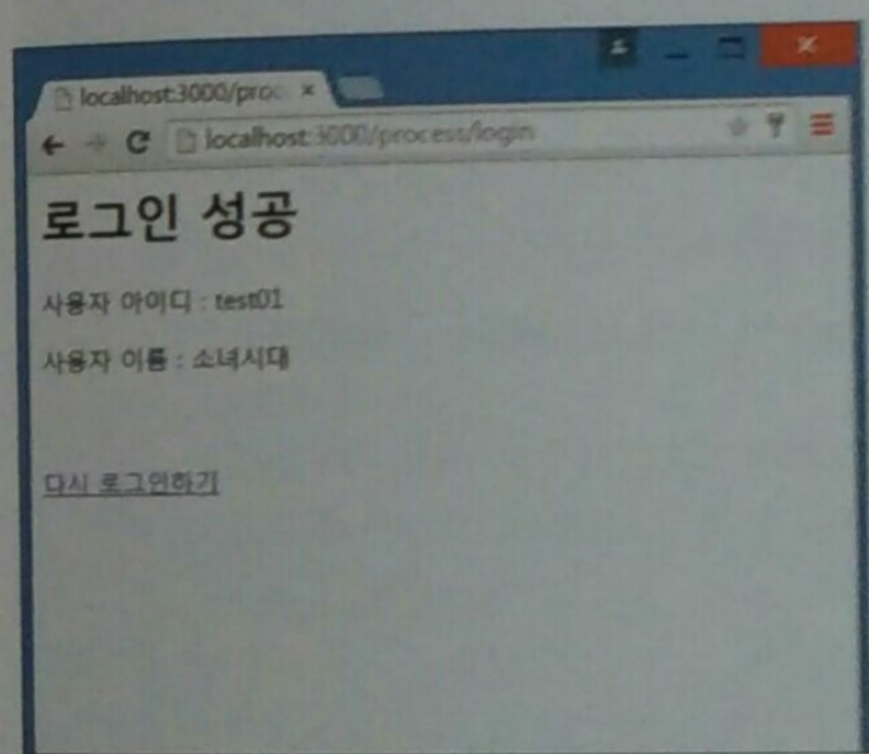
[프로젝트 폴더 아래의 폴더 중에서]

```
\node_modules \ mongodb \ node_modules \ mongodb-core \ node_modules \ bson \ browser_build
→
\ node_modules \ mongodb \ node_modules \ mongodb-core \ node_modules \ bson \ node_modules \ bson-ext \ build \ Release
```



◀ 사용자 인증을 위해 간단하게 만든 로그인 화면

로그인 화면이 보이면 아이디 입력란에 test01, 비밀번호 입력란에 123456을 입력한 후 [전송] 버튼을 누릅니다. 그러면 다음과 같이 로그인에 성공했다는 화면이 나타나면서 사용자 이름이 표시됩니다.



◀ 사용자 인증 결과를 보여 주는 화면

서버 쪽의 콘솔을 보면 다음과 같이 사용자 인증을 요청했을 때 데이터베이스에 연결하고 문서를 조회한 과정을 알 수 있는 로그가 표시됩니다.

```
Console  Debug
NodeExample3-app.js [Node Application] Node.js Process
서버가 시작되었습니다. 포트 : 3000
데이터베이스에 연결되었습니다. : mongodb://localhost:27017/shopping
/process/login 호출됨.
authUser 호출됨.
아이디 [test01], 패스워드 [123456] 가 일치하는 사용자 찾음.
[ { _id: { _bsontype: 'ObjectID', id: 'Uo¦qi±\\«£:' },
    id: 'test01',
    name: '소녀시대',
    password: '123456' } ]
```

▲ 사용자 인증 결과를 보여 주는 서버 쪽 콘솔 화면

주의! 만약 사용자 아이디나 비밀번호를 잘못 입력하면 사용자 인증에 실패했다는 메시지가 표시됩니다.

이렇게 데이터베이스에 저장된 사용자 정보를 확인하는 기능을 만들어 보았습니다.

사용자 추가 기능 만들기

지금부터는 사용자를 추가하는 기능을 만들어 보겠습니다. 로그인 페이지에는 보통 회원이 아닌 사람을 위해 사용자 등록 과정을 만들고 링크를 걸어 두는 경우가 많습니다. 따라서 회원가입 페이지를 만들면 웹 서버로 사용자 등록 요청을 하고 사용자 등록 요청을 받은 서버에서는 사용자를 추가하는 기능을 수행해야 합니다. app.js 파일을 복사하여 app2.js 파일을 만들고 이 파일 안에 다음과 같이 사용자 등록 기능을 처리하는 함수를 추가합니다.

참조 파일 | DatabaseExample>app2.js

```
……

// 사용자를 등록하는 함수
var addUser = function(database, id, password, name, callback) {
  console.log('addUser 호출됨.');

  // users 컬렉션 참조
  var users = database.collection('users');
  // id, password, username을 사용해 사용자 추가
  users.insert([{"id" : id, "password" : password, "name" : name}], function(err, result) {
    if(err) {
      callback(err, null);
      return;
    }

    console.log("사용자 데이터 추가함.");
    callback(null, result);

  });
}

……
```

addUser 함수에는 데이터베이스 객체와 함께 사용자에게 요청받은 id, password, name 파라미터 값을 전달합니다. 마지막 파라미터는 콜백 함수로서 이 함수를 호출하는 쪽에 결과 객체를 보내기 위해서 전달받습니다. users 컬렉션을 참조한 후 insert( ) 메소드를 호출하여 데이터를 추가합니다.

그다음에는 클라이언트에서 사용자 추가를 위해 요청할 /process/adduser 패스를 라우팅하는 함수를 다음과 같이 추가합니다.

```
……,

// 사용자 추가 함수
app.post('/process/adduser', function(req, res) {
  console.log('/process/adduser 호출됨.');

  var paramId = req.param('id');
  var paramPassword = req.param('password');
  var paramName = req.param('name');

  if(database) {
    addUser(database, paramId, paramPassword, paramName, function(err, result) {
      if(err) {throw err;}

      if(result) {
        console.dir(result);

        res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
        res.write('<h2>사용자 추가 성공</h2>');
        res.end();
      } else {
        res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
        res.write('<h2>사용자 추가 실패</h2>');
        res.end();
      }
    });
  } else {
    res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
    res.write('<h2>데이터베이스 연결 실패</h2>');
    res.end();
  }

});

……,
```

addUser( ) 함수를 호출하면서 동시에 사용자로부터 전달받은 id, password, name 파라미터를 전달합니다. 콜백 함수에 결과 객체가 전달되면 정상적으로 사용자가 데이터베이스에 추가된 것이므로 클라이언트에 성공했다는 응답을 보냅니다.

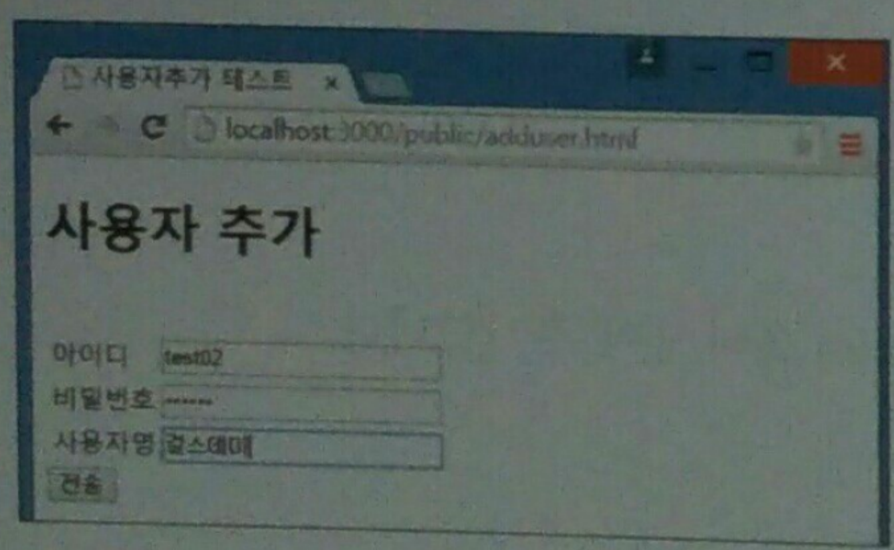
이제 다음과 같이 사용자 등록을 위한 웹 문서를 만들어 [public] 폴더 밑에 추가합니다.

참조 파일 | DatabaseExample>/public/adduser.html

```
……

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>사용자 추가 테스트</title>
  </head>
<body>
    <h1>사용자 추가</h1>
    <br>
    <form method="post" action="/process/adduser">
      <table>
        <tr>
          <td><label>아이디</label></td>
          <td><input type="text" name="id" /></td>
        </tr>
        <tr>
          <td><label>비밀번호</label></td>
          <td><input type="password" name="password" /></td>
        </tr>
        <tr>
          <td><label>사용자명</label></td>
          <td><input type="text" name="name" /></td>
        </tr>
      </table>
      <input type="submit" value="전송" name="" />
    </form>
</body>
</html>
```

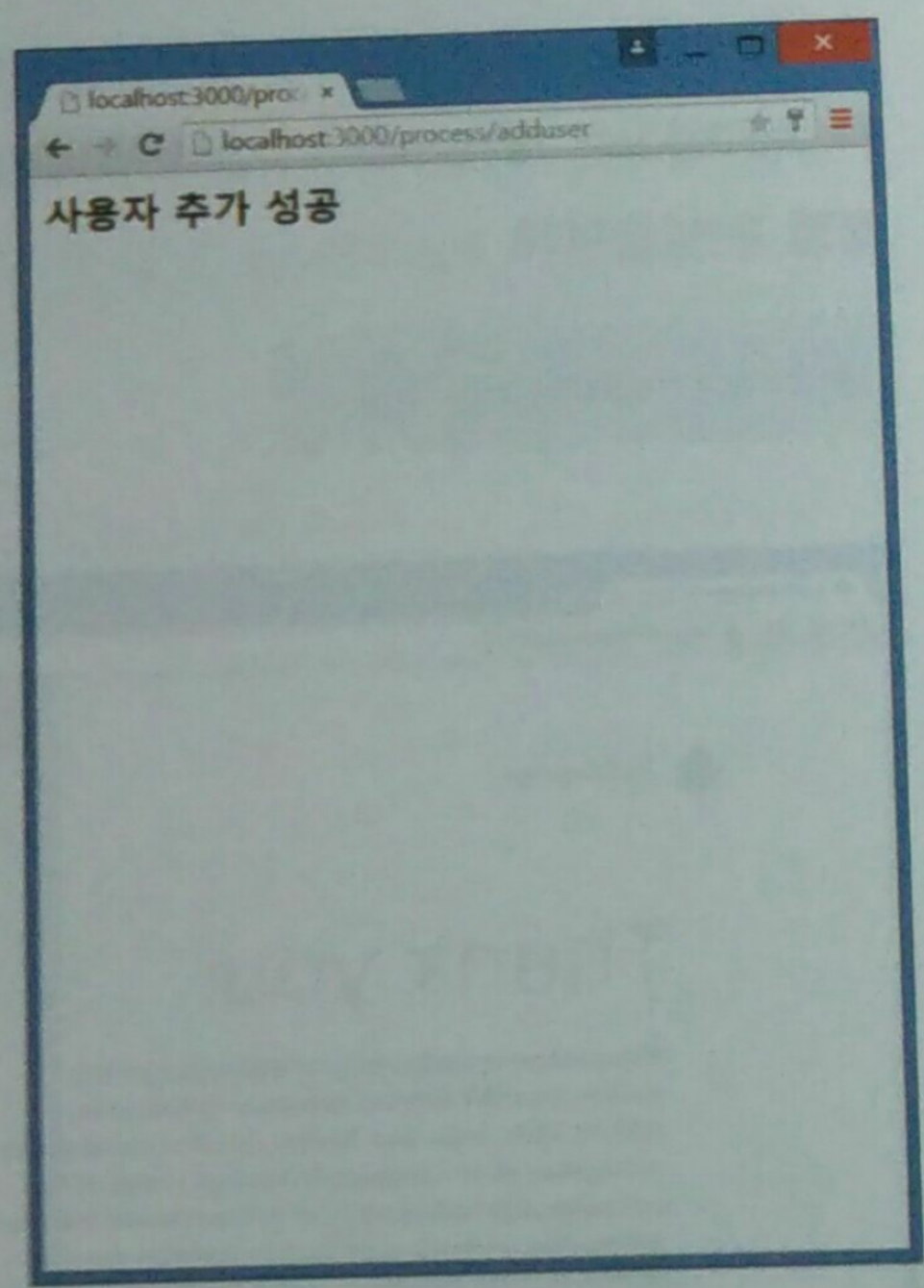
이전에 만들었던 로그인 화면과 거의 비슷하므로 login.html 파일을 복사한 후 수정해도 됩니다. app2.js 파일을 실행한 후 웹 브라우저에서 다음 주소로 요청합니다.

▶ http://localhost:3000/public/adduser.html



◀ 사용자 추가를 위해 만든 화면

사용자를 추가할 수 있는 화면이 보이면 아이디 입력란에 test02, 비밀번호 입력란에 123456, 사용자 이름 입력란에 걸스데이를 입력한 후 [전송] 버튼을 누릅니다. 사용자가 정상적으로 추가되면 다음과 같이 성공 메시지가 표시됩니다.



사용자를 추가한 후의 결과 화면▶

서버 쪽 콘솔을 보면 서버 쪽에서 사용자를 데이터베이스에 추가했다는 알림 메시지가 표시됩니다.

```
Console  Debug
NodeExample3-app2.js [Node Application] Node.js Process
서버가 시작되었습니다. 포트 : 3000
데이터베이스에 연결되었습니다. : mongodb://localhost:27017/shopping
/process/adduser 호출됨.
addUser 호출됨.
사용자 데이터 추가함.
{ result: { ok: 1, n: 1 },
  connection:
   { domain: null,
     _events:
      { close: [Object],
```

▲사용자 추가 결과를 보여 주는 서버 쪽 콘솔 화면

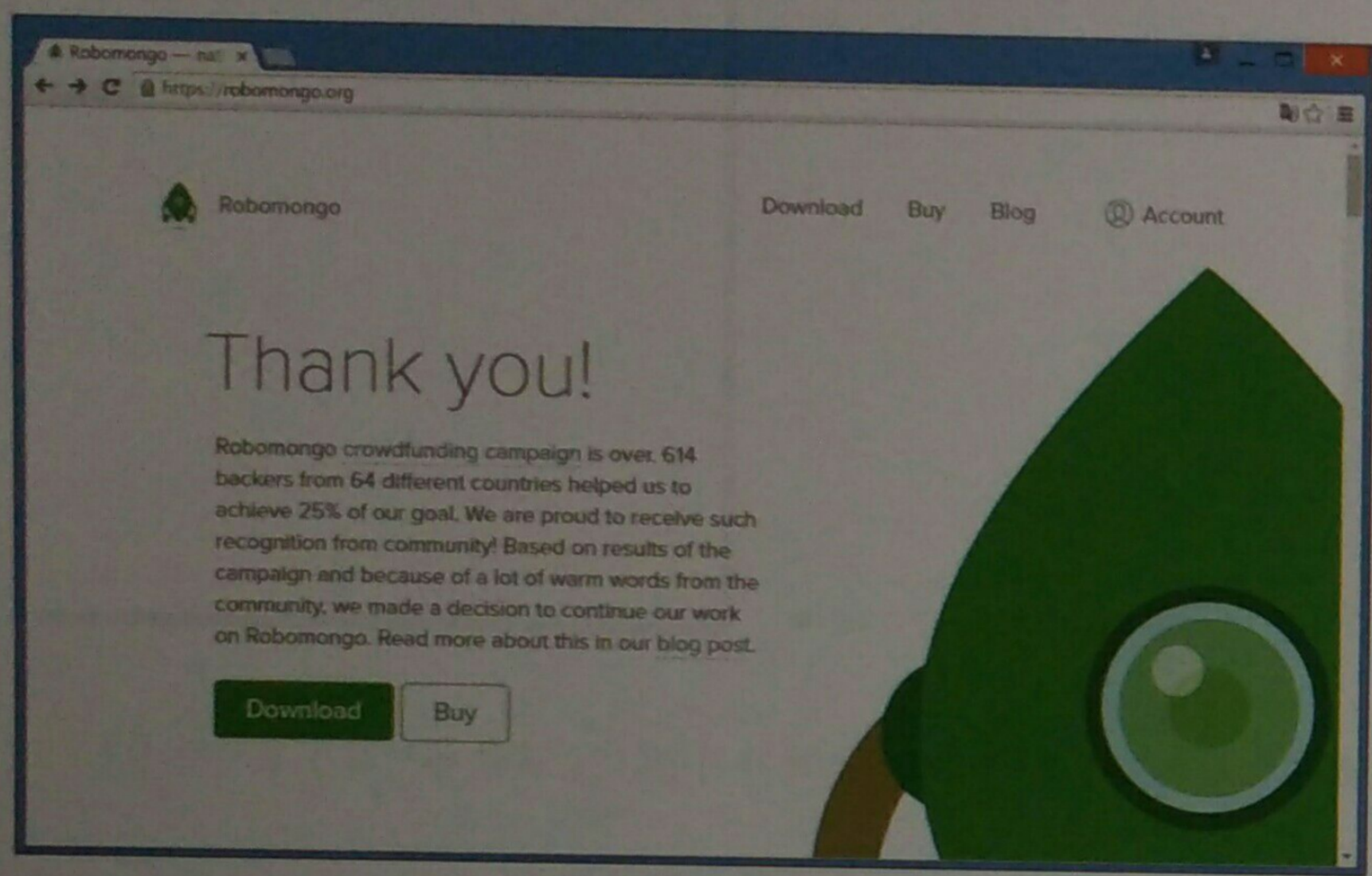
지금까지 사용자가 웹 브라우저에서 사용자 인증이나 등록을 요청하면 익스프레스 서버의 라우팅 함수가 실행되면서 몽고디비 데이터베이스에 저장되어 있는 사용자를 확인하거나 새로운 사용자 정보를 추가하는 방법에 대해 알아보았습니다. 코드가 조금 길지만 코드가 사용되는 패턴을 이해하면 서버 기능을 쉽게 구현할 수 있습니다.

데이터베이스 관리 도구 사용하기

몽고디비 데이터베이스도 다른 데이터베이스들처럼 편하게 화면을 보면서 데이터를 조회하거나 추가할 수 있습니다. 몽고디비 도구는 여러 가지가 있지만 여기에서는 오픈 소스로 제공하거나 무료로 제공하는 도구 중 하나를 사용하여 데이터를 확인해 보겠습니다.

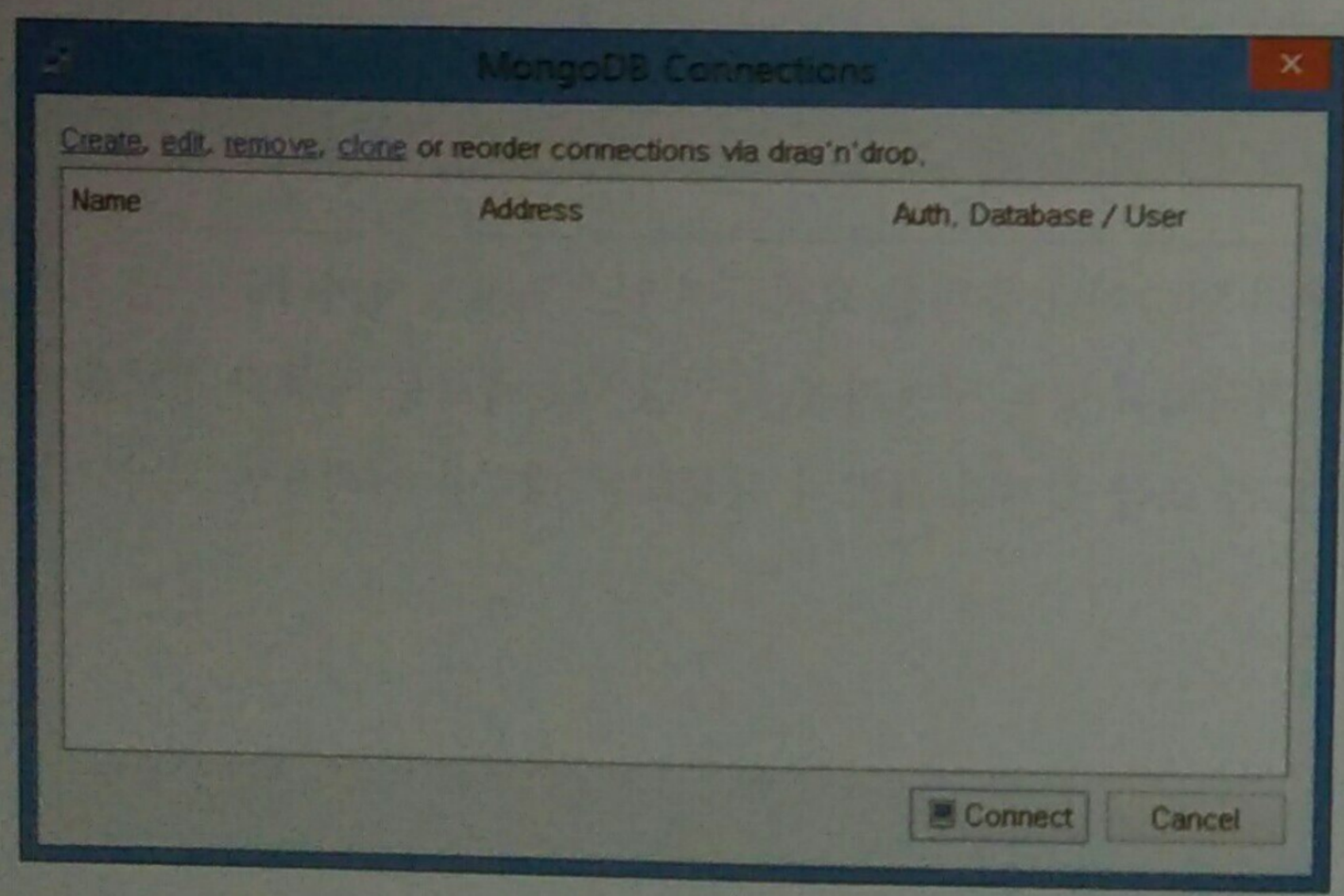
❶ 웹 브라우저에서 다음 사이트에 접속하면 UI 도구 로보몽고(Robomongo)를 다운로드할 수 있습니다. 로보몽고는 데이터베이스 안에 들어 있는 컬렉션과 문서 객체를 몽고디비 관리 도구로 확인하거나 수정할 수 있습니다.

▶ http://robomongo.org



◀ 로보몽고(Robomongo) 웹 사이트

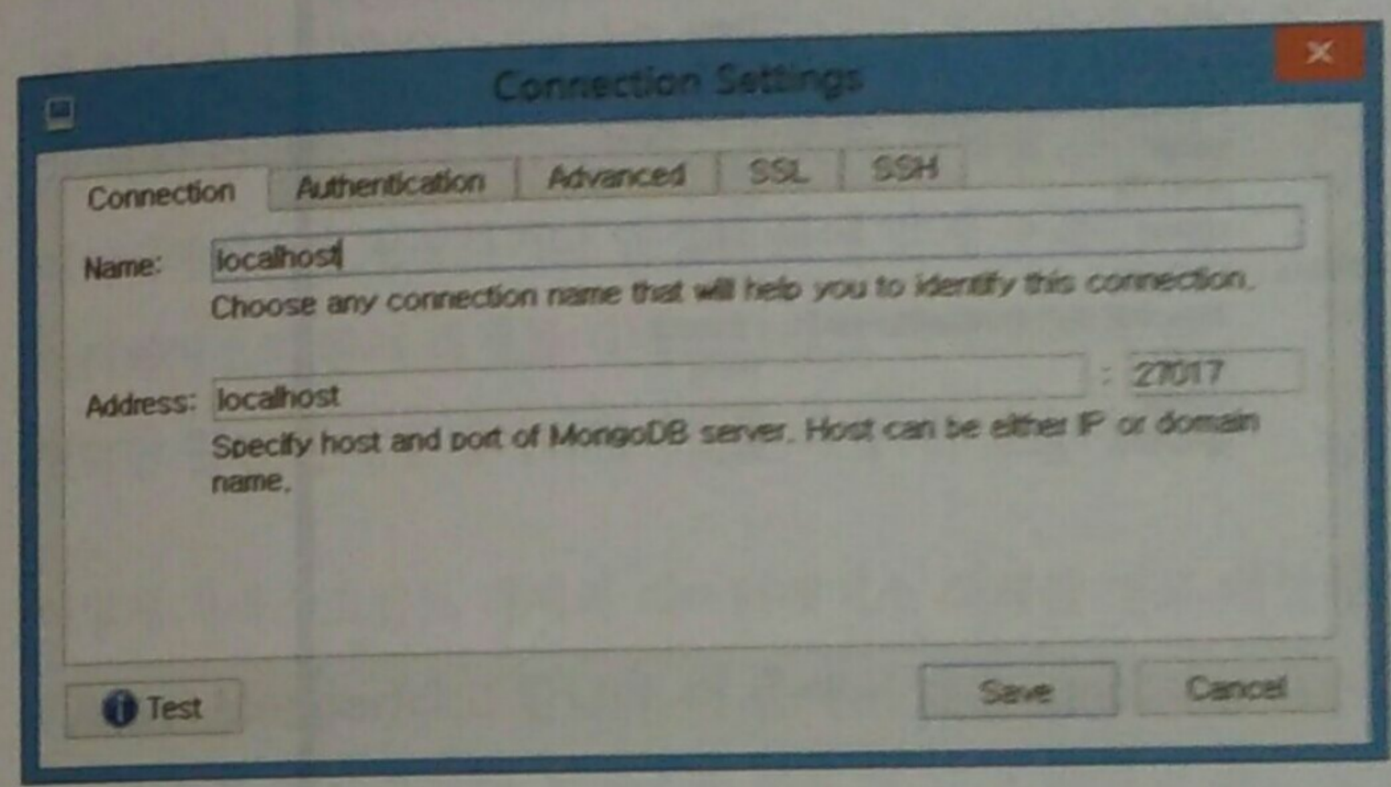
❷ [Download] 버튼을 클릭하면 다운로드 페이지로 이동합니다. 최신 버전은 유료지만 이전 버전은 무료이므로 이전 버전을 찾아 [Download] 버튼을 누릅니다. 설치 파일 다운로드가 끝나면 클릭하여 설치합니다. 로보몽고를 설치한 후 실행하면 메인 화면이 보이고 다음과 같은 연결 정보 설정 화면도 나타납니다. 만약 연결 정보 설정 화면이 없다면 위쪽 메뉴 바에서 [File → Connect...] 메뉴를 눌러 [MongoDB Connections] 대화상자를 불러옵니다.



▲ [MongoDB Connections] 대화상자

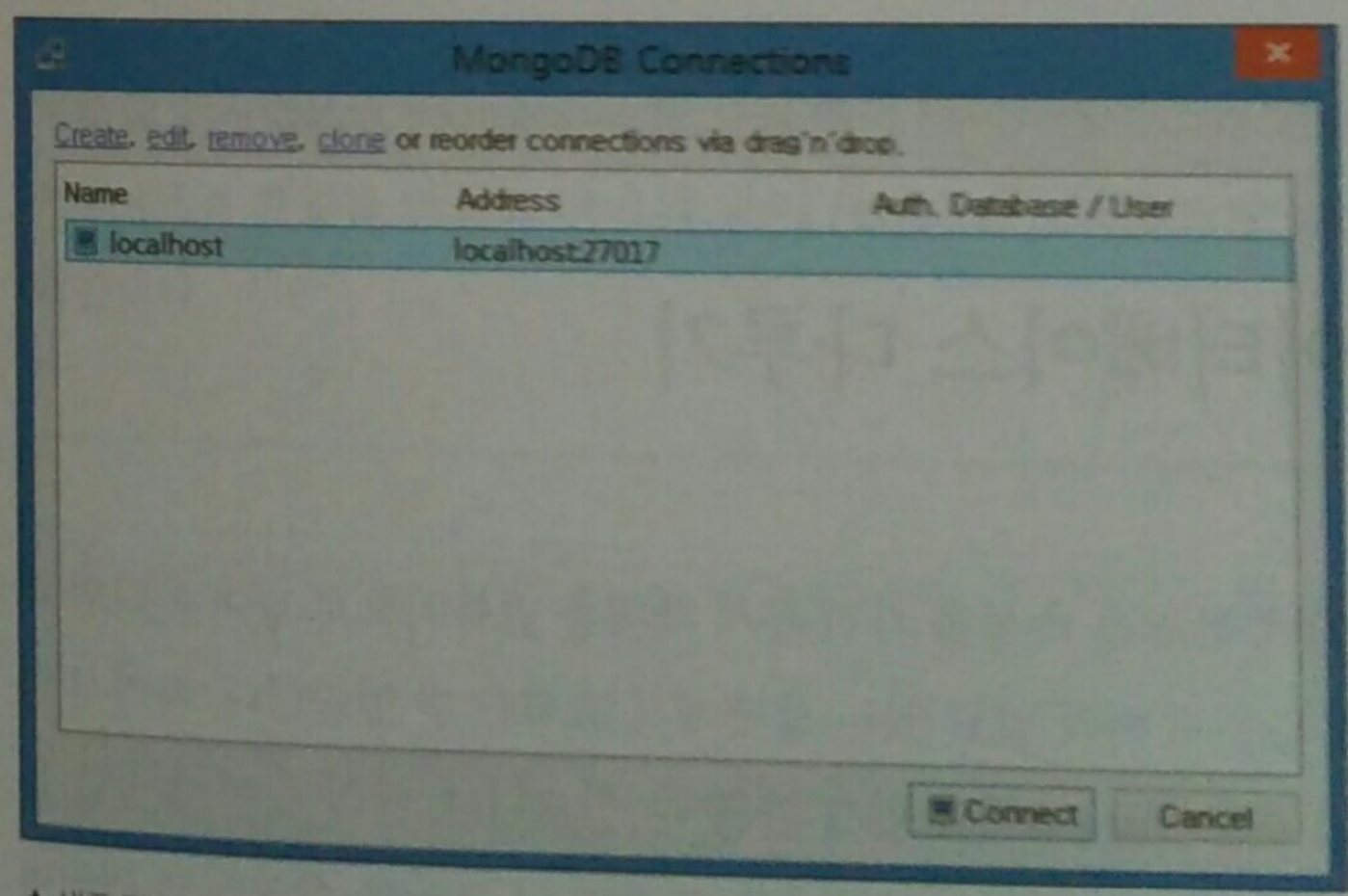
❸ 왼쪽 위의 'Create' 링크를 누르면 [Connection Settings] 대화상자가 나타납니다. 이 대화상자에 연결 정보를 입력합니다. mongodb 모듈에서 몽고디비 데이터베이스에 접속하는 URL은 다음과 같습니다. 디폴트 값인 localhost와 27017은 그대로 두고 Name: 항목의 값만 localhost로 바꿔서 연결 정보 이름을 localhost로 새롭게 만듭니다.

```
mongodb://localhost:27017/shopping
```



▲연결 정보를 입력한 화면

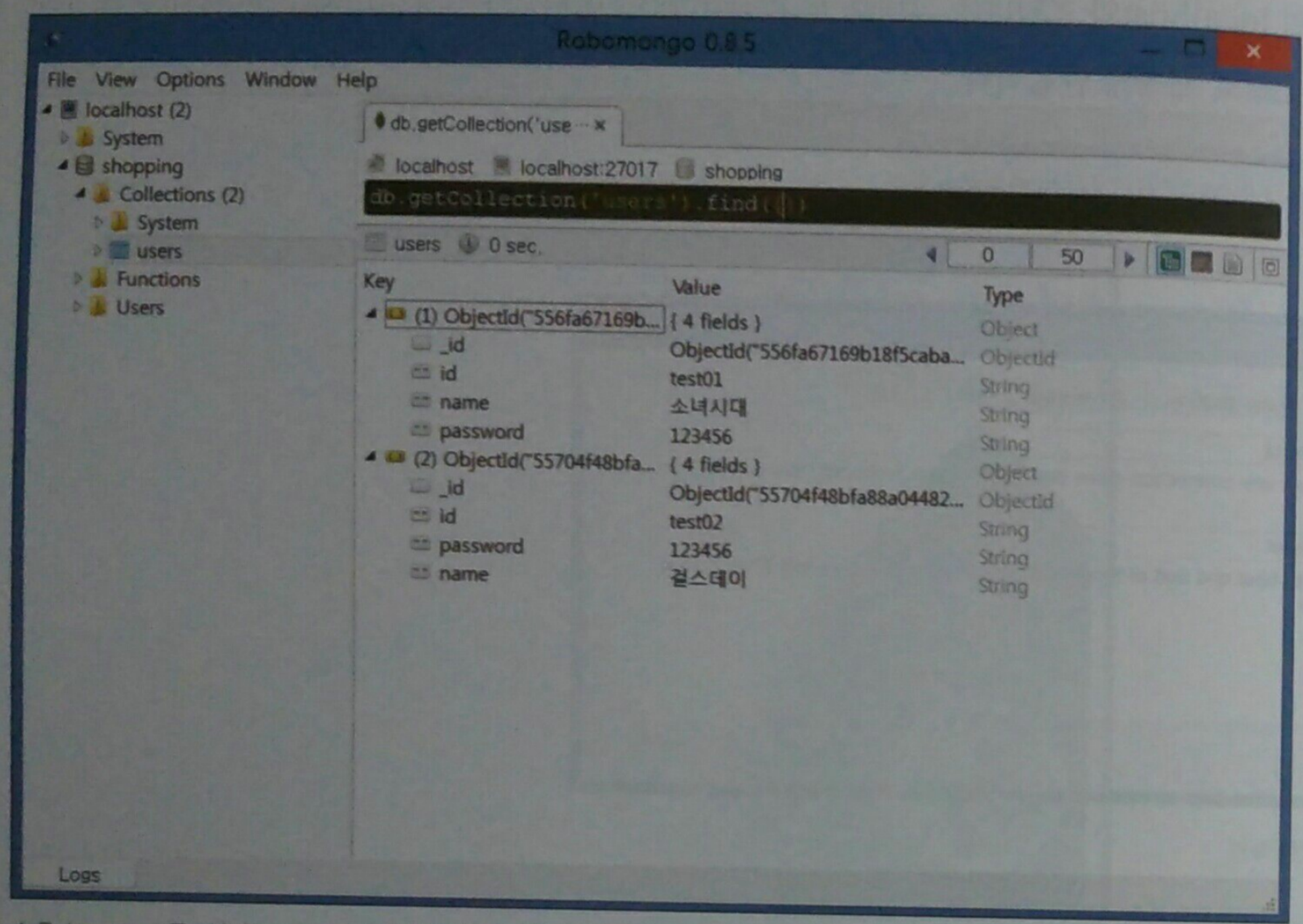
❹ Connection Settings 화면에서 [Advanced] 탭을 눌러 보면 어떤 데이터베이스로 연결할 것인지를 지정할 수 있습니다. shopping이라고 입력한 후 [Save] 버튼을 누르면 연결 정보가 저장되고 MongoDB Connections 화면에는 다음과 같이 연결 정보가 리스트에 나타납니다.



▲새로 저장된 연결 정보

❺ localhost 항목을 선택한 후 [Connect] 버튼을 눌러 데이터베이스에 연결하면 메인 화면 왼쪽에 데이터베이스 구조를 볼 수 있는 트리가 표시됩니다. localhost 아래에 shopping 항목이 보이면 그 아래에 있는 Collections를 선택합니다. 그리고 그 안에 users 컬렉션이 보이면 더블클릭합니다. 그러면 데이터

가 저장되어 있는 문서 객체들이 오른쪽 영역에 표시됩니다. 문서 객체들은 ObjectId를 기준으로 구분되며 ObjectId를 클릭하면 저장된 데이터를 확인할 수 있습니다.



▲ Robomongo를 사용해 users 컬렉션 안에 있는 데이터 확인하기

지금까지 확인한 것처럼 Robomongo 도구를 사용하면 아주 간단하게 컬렉션이나 문서 객체를 추가, 수정, 삭제할 수 있습니다.

06-3 | 몽구스로 데이터베이스 다루기

몽고디비를 사용하면서 하나의 컬렉션 안에 똑같은 속성을 가진 문서 객체를 반복적으로 넣어 둔다면 데이터를 조회할 때도 어떤 속성들이 들어 있는지 미리 알고 있는 상태에서 조회할 수 있습니다. 따라서 관계형 데이터베이스의 테이블이나 엑셀의 시트(Sheet)처럼 쉽게 다룰 수 있습니다. 이와 같이 일정한 틀을 제공하는 모듈 중에서 대표적인 것이 **몽구스(mongoose)**입니다. 그러면 이 모듈을 사용하면 데이터를 저장하거나 조회하는 과정이 어떻게 달라지는지 알아보겠습니다.

몽구스 모듈 사용하기

NoSQL 데이터베이스 중 하나인 몽고디비를 사용하면 문서 객체 안에 들어가는 속성을 마음대로 바꿀 수 있으므로 매우 유연하게 데이터를 저장할 수 있습니다. 하지만 컬렉션 안에 들어 있는 여러 개의 문서 객체를 조회할 때는 제약이 생길 수도 있습니다. 예를 들어, 같은 users 컬렉션 안에 들어 있는 문서 객체라도 어떤 문서 객체에는 name 속성이 있는 데 반해 다른 문서 객체에는 name 속성이 없을 수도 있습니다. 그래서 관계형 데이터베이스처럼 조회 조건을 공통 적용하기 어려운 문제점이 있습니다. 이런 문제 때문에 **스키마(Schema)**를 만들고, 그 스키마에 따라 문서 객체를 저장하는 것이 때로는 편리합니다. 특히 일정한 틀에 맞는 자바스크립트 객체를 그대로 데이터베이스에 저장하거나 일정한 틀에 맞게 구축된 데이터베이스의 문서 객체를 자바스크립트 객체로 바꿀 수 있다면 훨씬 편하게 데이터베이스를 다룰 수 있습니다.

용어! **스키마(Schema)란 무엇일까요?** 스키마란 데이터베이스의 구조를 정의한 것을 말합니다. 예를 들어, 엑셀 시트에 이름과 나이 데이터를 넣으려고 한다면 첫 번째 칼럼에는 이름을 문자로 넣고, 두 번째 칼럼에는 나이를 숫자로 넣는다는 것을 미리 정해 둘 수 있습니다. 이런 구조를 데이터베이스에서는 스키마라고 부릅니다.

이렇게 자바스크립트 객체와 데이터베이스 객체를 서로 매칭하여 바꿀 수 있게 하는 것을 **오브젝터 맵퍼(Object Mapper)**라고 합니다. 이 중에서 가장 많이 사용하는 것이 바로 몽구스(mongoose) 모듈입니다. 이 모듈을 사용하면 스키마를 만들고 이 스키마에 맞는 모델(Model)을 만들어서 데이터를 손쉽게 저장하거나 조회할 수 있습니다.

몽구스를 사용하려면 먼저 명령 프롬프트에서 다음과 같이 mongoose 모듈을 설치합니다.

```
% npm install mongoose --save
```

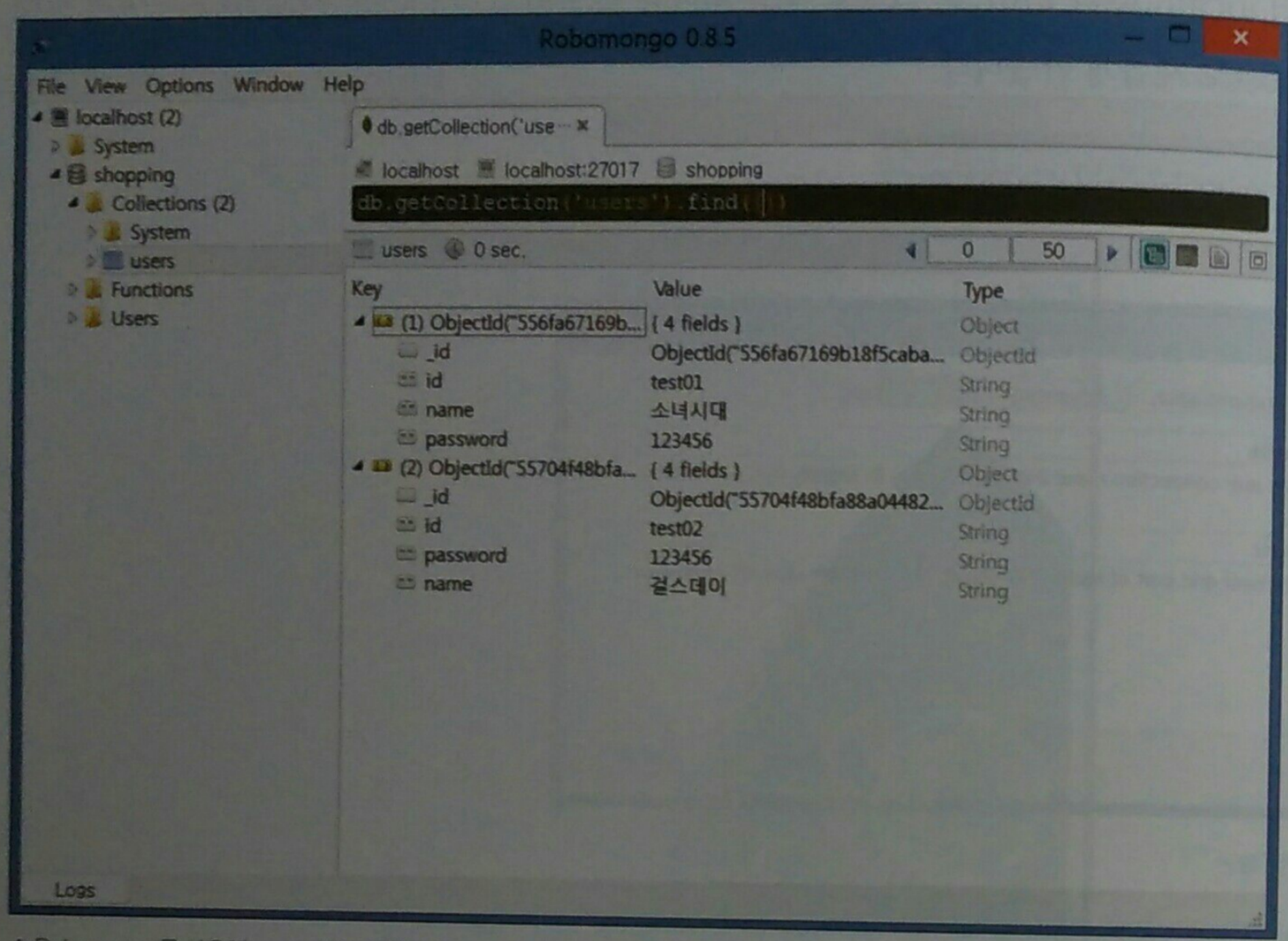
모듈을 설치했으면 mongoose를 사용해서 데이터베이스에 연결할 수 있습니다. app2.js 파일을 복사하여 app3.js 파일을 만들고 다음과 같이 require( ) 메소드로 mongoose 모듈을 불러들이는 코드를 입력합니다.

참조 파일 | DatabaseExample>app3.js

```
......

// mongoose 모듈 불러들이기
var mongoose = require('mongoose');

......
```

mongoose 모듈에서 사용할 수 있는 대표적인 메소드들은 다음과 같습니다.

가 저장되어 있는 문서 객체들이 오른쪽 영역에 표시됩니다. 문서 객체들은 ObjectId를 기준으로 구분되며 ObjectId를 클릭하면 저장된 데이터를 확인할 수 있습니다.



▲ Robomongo를 사용해 users 컬렉션 안에 있는 데이터 확인하기

지금까지 확인한 것처럼 Robomongo 도구를 사용하면 아주 간단하게 컬렉션이나 문서 객체를 추가, 수정, 삭제할 수 있습니다.

06-3 | 몽구스로 데이터베이스 다루기

몽고디비를 사용하면서 하나의 컬렉션 안에 똑같은 속성을 가진 문서 객체를 반복적으로 넣어 둔다면 데이터를 조회할 때도 어떤 속성들이 들어 있는지 미리 알고 있는 상태에서 조회할 수 있습니다. 따라서 관계형 데이터베이스의 테이블이나 엑셀의 시트(Sheet)처럼 쉽게 다룰 수 있습니다. 이와 같이 일정한 틀을 제공하는 모듈 중에서 대표적인 것이 **몽구스(mongoose)**입니다. 그러면 이 모듈을 사용하면 데이터를 저장하거나 조회하는 과정이 어떻게 달라지는지 알아보겠습니다.

몽구스 모듈 사용하기

NoSQL 데이터베이스 중 하나인 몽고디비를 사용하면 문서 객체 안에 들어가는 속성을 마음대로 바꿀 수 있으므로 매우 유연하게 데이터를 저장할 수 있습니다. 하지만 컬렉션 안에 들어 있는 여러 개의 문서 객체를 조회할 때는 제약이 생길 수도 있습니다. 예를 들어, 같은 users 컬렉션 안에 들어 있는 문서 객체라도 어떤 문서 객체에는 name 속성이 있는 데 반해 다른 문서 객체에는 name 속성이 없을 수도 있습니다. 그래서 관계형 데이터베이스처럼 조회 조건을 공통 적용하기 어려운 문제점이 있습니다. 이런 문제 때문에 **스키마(Schema)**를 만들고, 그 스키마에 따라 문서 객체를 저장하는 것이 때로는 편리합니다. 특히 일정한 틀에 맞는 자바스크립트 객체를 그대로 데이터베이스에 저장하거나 일정한 틀에 맞게 구축된 데이터베이스의 문서 객체를 자바스크립트 객체로 바꿀 수 있다면 훨씬 편하게 데이터베이스를 다룰 수 있습니다.

용어! **스키마(Schema)란 무엇일까요?** 스키마란 데이터베이스의 구조를 정의한 것을 말합니다. 예를 들어, 엑셀 시트에 이름과 나이 데이터를 넣으려고 한다면 첫 번째 칼럼에는 이름을 문자로 넣고, 두 번째 칼럼에는 나이를 숫자로 넣는다는 것을 미리 정해 둘 수 있습니다. 이런 구조를 데이터베이스에서는 스키마라고 부릅니다.

이렇게 자바스크립트 객체와 데이터베이스 객체를 서로 매칭하여 바꿀 수 있게 하는 것을 **오브젝터 맵퍼(Object Mapper)**라고 합니다. 이 중에서 가장 많이 사용하는 것이 바로 몽구스(mongoose) 모듈입니다. 이 모듈을 사용하면 스키마를 만들고 이 스키마에 맞는 모델(Model)을 만들어서 데이터를 손쉽게 저장하거나 조회할 수 있습니다.

몽구스를 사용하려면 먼저 명령 프롬프트에서 다음과 같이 mongoose 모듈을 설치합니다.

```
% npm install mongoose --save
```

모듈을 설치했으면 mongoose를 사용해서 데이터베이스에 연결할 수 있습니다. app2.js 파일을 복사하여 app3.js 파일을 만들고 다음과 같이 require( ) 메소드로 mongoose 모듈을 불러들이는 코드를 입력합니다.

참조 파일 | DatabaseExample>app3.js

```
……

// mongoose 모듈 불러들이기
var mongoose = require('mongoose');

……
```

mongoose 모듈에서 사용할 수 있는 대표적인 메소드들은 다음과 같습니다.

메소드 이름	설명
connect(uri(s), [options], [callback])	mongoose를 사용해 데이터베이스에 연결합니다. 연결 후에는 mongoose.connection 객체를 사용해 연결 관련 이벤트를 처리할 수 있습니다.
Schema( )	스키마를 정의하는 생성자입니다.
model(name, [schema], [collection], [skipInit])	모델을 정의합니다. [collection]이 지정되면 이 컬렉션을 사용하며, 지정하지 않으면 name으로 유추한 컬렉션을 사용합니다.

[표] mongoose 모듈에서 사용할 수 있는 대표적인 메소드

몽구스도 데이터베이스에 연결한 후에 사용할 수 있습니다. 따라서 connectDB( ) 함수의 코드를 다음과 같이 수정하여 mongoose 모듈로 데이터베이스에 연결하고, 연결했을 때 전달받는 이벤트를 처리하도록 만듭니다.

참조 파일 | DatabaseExample>app3.js

```
.....

var database;
var UserSchema;
var UserModel;

// 데이터베이스에 연결하고 응답 객체의 속성으로 db 객체 추가
function connectDB() {
  // 데이터베이스 연결 정보
  var databaseUrl = 'mongodb://localhost:27017/shopping';

  // 데이터베이스 연결
  mongoose.connect(databaseUrl);
  database = mongoose.connection;

  database.on('error', console.error.bind(console, 'mongoose connection error.'));
  database.on('open', function() {
    console.log('데이터베이스에 연결되었습니다. : ' + databaseUrl);

    // 스키마 정의
    UserSchema = mongoose.Schema({
      id: String,
      name: String,
      password: String
    });
```

```
    console.log('UserSchema 정의함.');

    // User 모델 정의
    UserModel = mongoose.model("users", UserSchema);
    console.log('users 정의함.');
  });
  database.on('disconnected', connectDB);
}

......
```

데이터베이스 연결 정보는 mongodb 모듈로 연결했을 때의 연결 정보와 같습니다. mongoose 모듈로 데이터베이스에 연결할 때는 connect( ) 메소드를 호출하면서 동시에 데이터베이스 연결 정보를 파라미터로 넘겨줍니다. 데이터베이스에 연결되었는지 여부는 mongoose 객체에 들어 있는 connection 객체로 전달되는 이벤트를 통해 확인할 수 있습니다.

open 이벤트는 데이터베이스가 연결되었을 때 발생하며, **error 이벤트**는 데이터베이스 연결이 제대로 되지 않을 때 발생합니다. **disconnected 이벤트**는 데이터베이스 연결이 끊어졌을 때 호출되는데 데이터베이스 연결이 끊어져서 disconnected 이벤트가 발생하면 데이터베이스에 다시 연결하도록 connectDB 함수를 호출합니다.

데이터베이스에 연결되면 이미 만들어 둔 users 컬렉션을 위해 스키마(Schema)를 정의합니다. 스키마는 컬렉션이 가지고 있는 속성을 정의한 것으로 mongoose.Schema( ) 메소드를 호출하면 새로운 스키마를 만들 수 있습니다. 스키마를 만들 때는 각 속성의 정보를 넣을 수 있는데 users 컬렉션에서 사용하는 id, name, password 속성을 넣어 줍니다.

스키마에 들어가는 각 속성은 **스키마 타입(Schema Type)**을 가집니다. 데이터 타입과 같은 역할을 하는 스키마 타입은 다음과 같습니다.

스키마 타입	설명
String	문자열 타입
Number	숫자 타입
Boolean	이진 타입
Array	배열 타입
Buffer	버퍼 타입 바이너리 데이터를 저장할 수 있습니다.
Date	날짜 타입
ObjectId	각 문서(Document)마다 만들어지는 ObjectId를 저장할 수 있는 타입
Mixed	혼합 타입

[표] mongoose에서 스키마를 정의할 때 각 속성에 사용할 수 있는 스키마 타입

사용자 정보를 저장하는 스키마에 id, password, name 이외에 나이, 데이터 생성 날짜, 데이터 수정 날짜까지 저장하고 싶다면 다음과 같이 정의하면 됩니다.

```
var UserSchema = new mongoose.Schema({
    id: {type: String, required: true, unique: true},
    password: {type: String, required: true},
    name: String,
    age: Number,
    created_at: Date,
    updated_at: Date
});
```

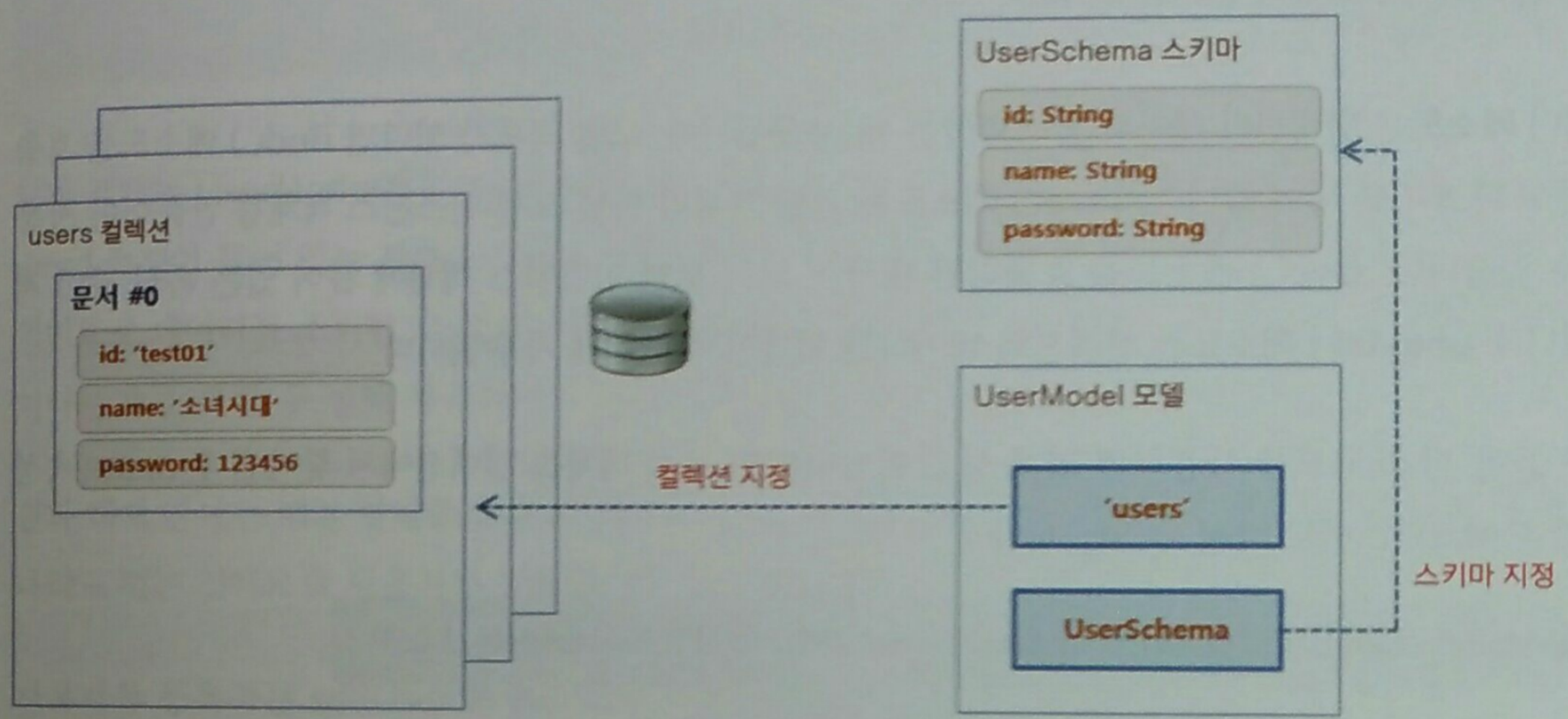
문자열이 들어가는 속성의 스키마 타입을 단순히 String이라고 지정할 수 있습니다. 하지만 좀 더 구체적으로 지정하기 위해 자바스크립트 객체를 전달할 수도 있습니다. 자바스크립트 객체는 중괄호를 사용해 만들게 되므로 중괄호 안에 각 속성이 가져야 하는 정보를 구체적으로 지정합니다. 중괄호 안에 넣은 속성 이름이 갖는 의미는 다음과 같습니다.

속성 이름	설명
- type	자료형을 지정합니다.
- required	값이 true이면 반드시 들어가야 하는 속성이 됩니다.
- unique	값이 true이면 이 속성에 고유한 값이 들어가야 합니다.

[표] 중괄호 안에 넣은 속성 이름이 갖는 의미

id 속성의 경우 문자열이 들어가면서 값을 NULL로 둘 수 없고 각 데이터들이 고유한 값을 가져야 한다면 type, required, unique 속성이 모두 들어 있는 자바스크립트 객체를 만들어 스키마 타입으로 지정할 수 있습니다.

여기에서는 이미 만들어 둔 users 컬렉션의 문서 속성과 같도록 맞추기 위해 id, password, name 속성만으로 정의합니다. 스키마는 단순히 구조만 정의하므로 데이터베이스에 들어 있는 컬렉션을 지정하려면 모델을 만들어야 합니다. mongoose 객체의 model( ) 메소드를 사용하면 데이터베이스의 컬렉션을 지정하는 모델 객체를 만들 수 있습니다.



▲ mongoose의 스키마와 모델 간의 관계

models( ) 메소드의 첫 번째 파라미터로는 모델 이름이 전달되고 두 번째 파라미터에는 스키마 객체가 전달됩니다. 이렇게 모델 객체를 만들면 모델 객체에 지정한 users라는 이름은 데이터베이스의 users 컬렉션과 매칭되고 UserSchema 객체는 mongoose.Schema( ) 메소드로 만든 스키마 객체를 가리키게 됩니다. 스키마 객체와 모델 객체 모두 다른 함수에서도 사용할 수 있도록 변수로 정의해 두었습니다.

몽구스로 사용자 인증하기

스키마를 만들었으므로 이제 데이터베이스에 들어 있는 users 컬렉션의 데이터를 조회하여 사용자 인증 과정에 적용해 볼 차례입니다. 그러려면 데이터 조회 방법을 알아야 합니다. 모델 객체 또는 모델을 사용해서 만들어 낸 인스턴스 객체에는 데이터베이스의 컬렉션에 들어 있는 문서 객체를 조회, 수정, 삭제할 수 있는 메소드가 들어 있습니다. 다음은 데이터 처리에 사용하는 대표적인 메소드들입니다.

메소드 이름	설명
find([criteria], [callback])	조회 조건을 사용해 컬렉션의 데이터를 조회합니다. 조회 결과는 콜백 함수로 전달됩니다.
save([options], [callback])	모델 인스턴스 객체의 데이터를 저장합니다. 저장 결과는 콜백 함수로 전달됩니다.
update([criteria], [doc], [options], [callback])	컬렉션의 데이터를 조회한 후 업데이트합니다. where( ) 메소드와 함께 사용됩니다.
remove([criteria], [callback])	컬렉션의 데이터를 삭제합니다.

[표] mongoose에서 데이터 처리를 위해 사용하는 대표적인 메소드

find( ) 메소드는 컬렉션의 데이터를 조회하는 데 사용됩니다. 모델 객체가 있다면 find( ) 메소드를 호출해서 조회 조건을 넣어 줍니다. **save( ) 메소드**는 모델 객체가 아닌 모델 인스턴스 객체를 만들어야 사용할 수 있습니다. save( ) 메소드를 호출하면 대상이 되는 모델 인스턴스 객체에 들어 있는 속성 값이 저장됩니다. **update( ) 메소드**는 컬렉션의 데이터를 업데이트하는 데 사용됩니다.

예를 들어, 다음 코드를 사용하면 id 속성 값이 test01인 문서 객체를 컬렉션에서 찾아낸 후 name 속성의 값을 애프터스쿨로 바꿀 수 있습니다.

```
UserModel.where({id : 'test01'}).update({name: '애프터스쿨'}, function(err...) {...})
```

사용자를 인증하는 authUser( ) 함수의 코드는 다음과 같이 수정합니다.

참조 파일 | DatabaseExample>app3.js

```
.....

// 사용자를 인증하는 함수
var authUser = function(database, id, password, callback) {
  console.log('authUser 호출됨.');

  // 아이디와 비밀번호를 사용해 검색
  UserModel.find({"id" : id, "password" : password}, function(err, results) {
    if(err) {
      callback(err, null);
      return;
    }

    console.log('아이디 [%s], 비밀번호 [%s]로 사용자 검색 결과', id, password);
    console.dir(results);
```

```
    if(results.length > 0) {
      console.log('일치하는 사용자 찾음.', id, password);
      callback(null, results);
    } else {
      console.log("일치하는 사용자를 찾지 못함.");
      callback(null, null);
    }
  });
};

……
```

모델 객체에 UserModel에는 find( ) 메소드가 들어 있으므로 find( ) 메소드를 호출하면서 id와 pass word 속성이 들어 있는 자바스크립트 객체를 파라미터로 전달합니다. 이렇게 하면 지정된 아이디와 비밀번호로 데이터를 조회할 수 있습니다. 조회 결과는 콜백 함수로 전달되는데 콜백 함수의 첫 번째 파라미터는 오류 객체, 두 번째 파라미터는 결과 객체가 됩니다.

결과 객체 안에는 배열 형태로 조회된 객체들이 들어 있는데, 배열의 크기가 0보다 크면 조회 결과가 하나라도 있는 것이므로 사용자가 인증된 것으로 처리합니다.

사용자를 등록하는 addUser 함수는 다음과 같이 수정합니다.

참조 파일 | DatabaseExample>app3.js

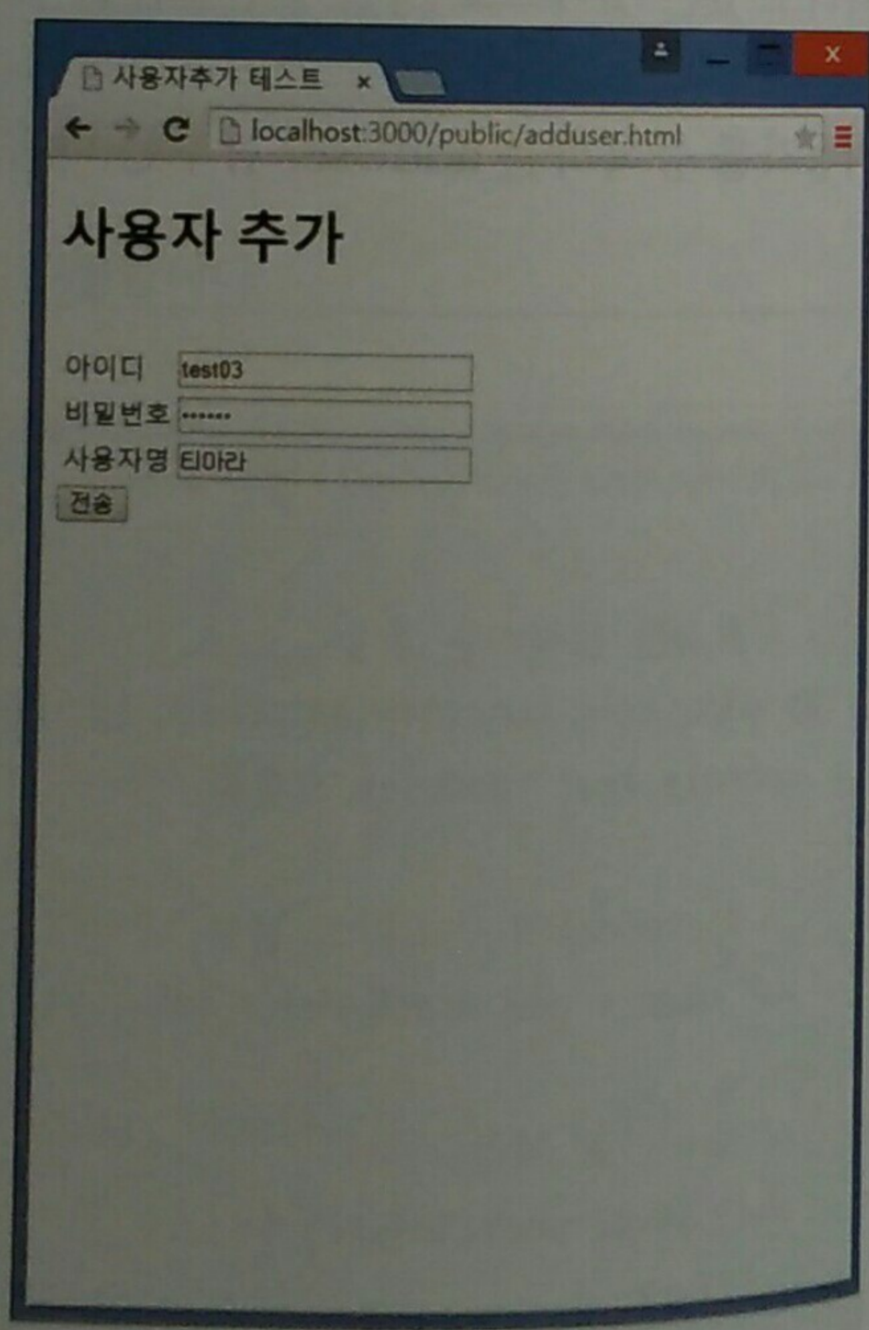
```
……

// 사용자를 등록하는 함수
var addUser = function(database, id, password, name, callback) {
  console.log( ' addUser 호출됨.');

  // UserModel의 인스턴스 생성
  var user = new UserModel({"id" : id, "password" : password, "name" : name});

  // save()로 저장
  user.save(function(err) {
    if(err) {
      callback(err, null);
      return;
    }
```

```
    console.log("사용자 데이터 추가함.");
    callback(null, user);

  });
};

.......
```

데이터를 저장하기 위해 new 연산자와 모델 객체인 UserModel로 새로운 모델 인스턴스를 만듭니다. new 연산자로 모델 인스턴스를 만들면 그 객체의 save( ) 메소드를 호출하여 저장합니다. 사용자가 브라우저에서 전달한 id와 password 값으로 모델 인스턴스를 만들었기 때문에 그 값이 그대로 데이터베이스의 users 컬렉션에 추가됩니다.

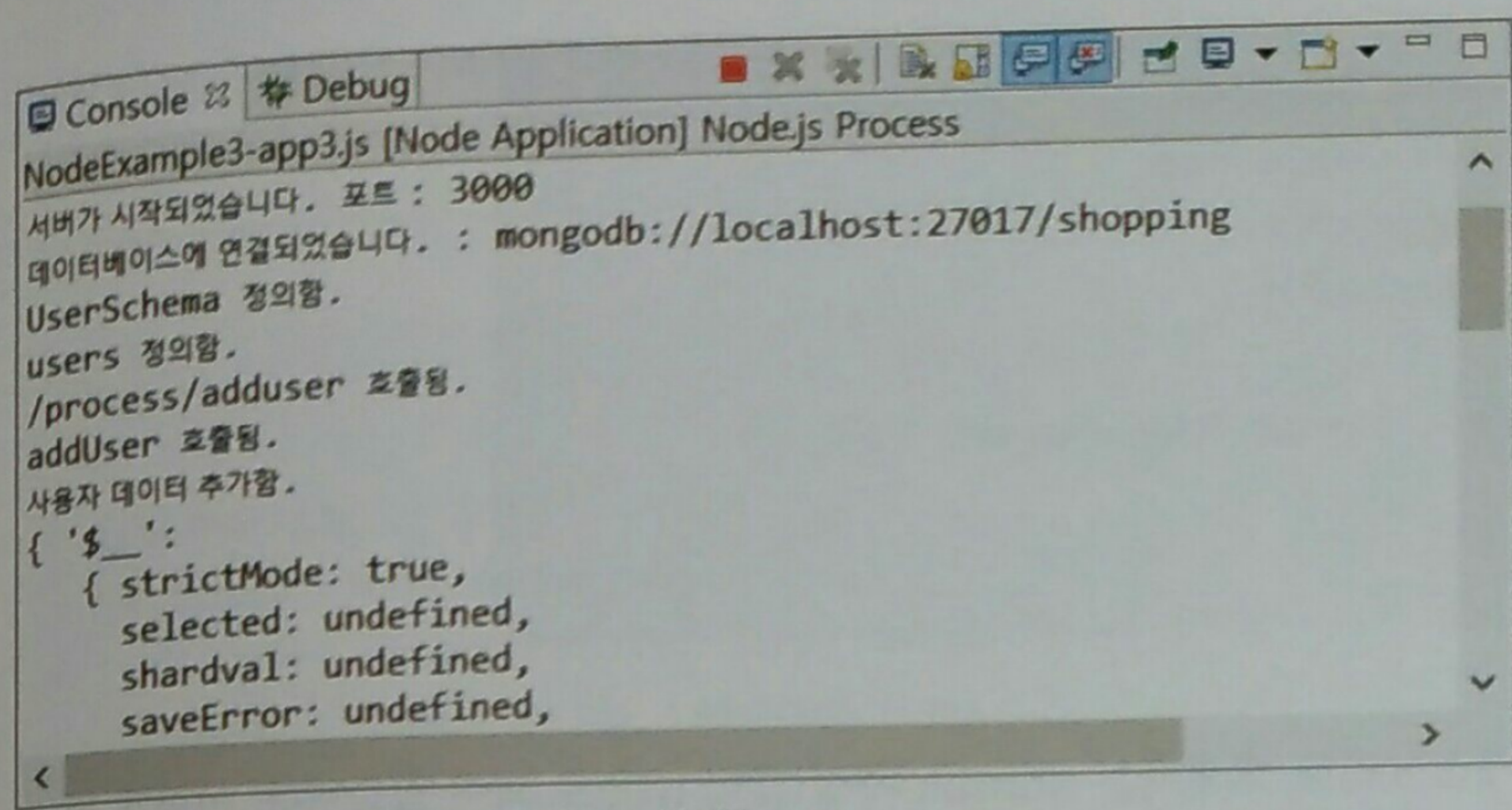
이제 app3.js 파일을 실행한 후 브라우저에서 http://localhost:3000/public/login.html 사이트를 엽니다. 아이디와 비밀번호를 입력한 후 [전송] 버튼을 누르면 로그인 성공 페이지가 나타납니다. 그리고 http://localhost:3000/public/adduser.html 사이트를 열고 사용자 등록을 요청하면 데이터베이스의 users 컬렉션에 사용자가 추가됩니다.

다음은 웹 브라우저에서 사용자 등록을 요청하는 화면입니다. 아이디 입력란에는 test03, 비밀번호 입력란에는 123456 그리고 사용자명에는 티아라를 입력합니다.



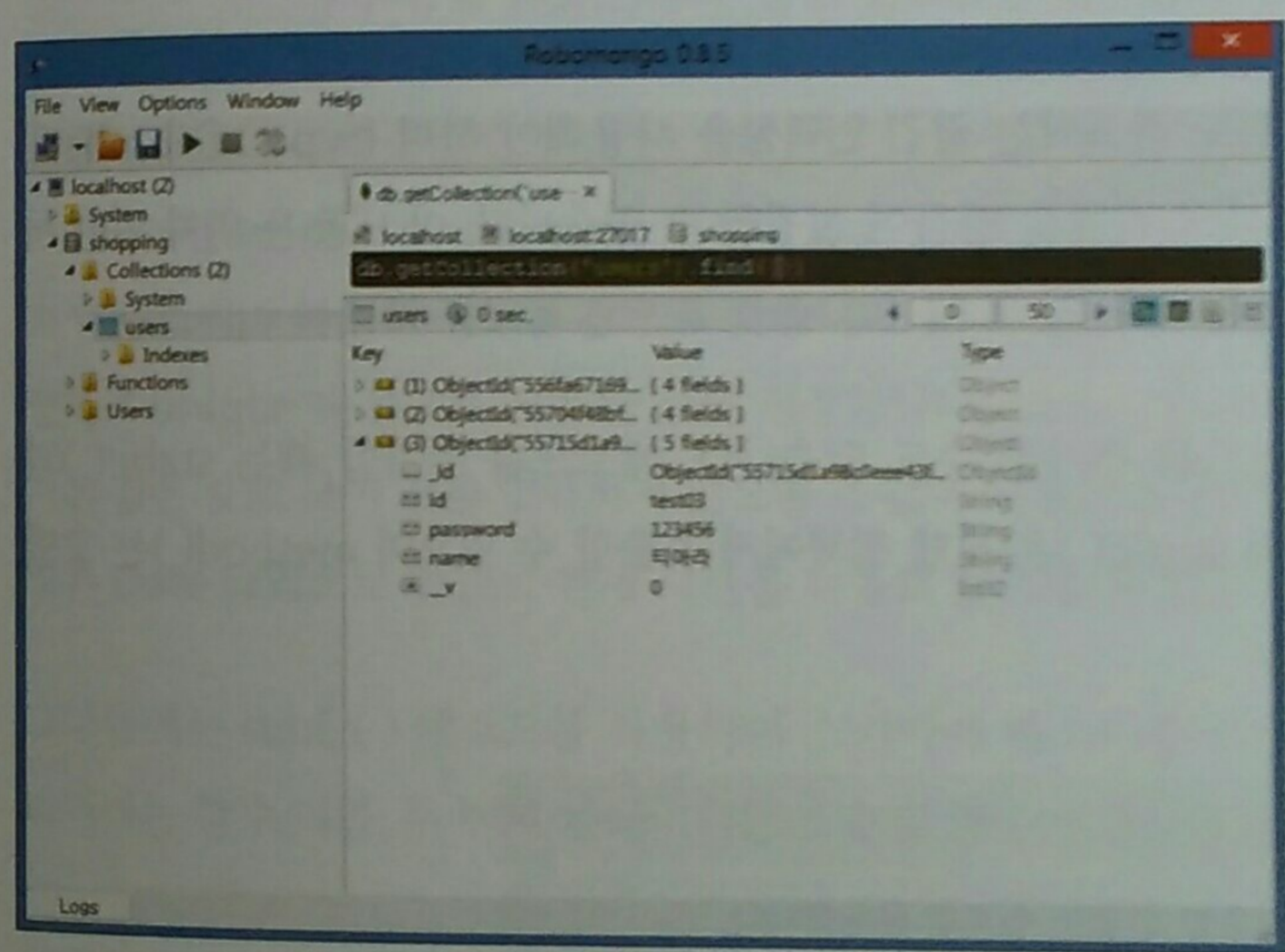
사용자 등록 화면에서 test03 사용자 정보를 입력한 경우 ▶

[전송] 버튼을 누르면 서버에 사용자 등록을 요청하고 정상적으로 처리되면 성공했다는 메시지가 표시됩니다. 다음은 사용자 등록이 성공적으로 처리된 후에 서버의 콘솔 창에서 볼 수 있는 메시지입니다.



▲사용자를 등록한 후 서버의 콘솔 창에 나타나는 메시지

사용자를 등록한 후 Robomongo 화면에서 users 컬렉션에 들어 있는 데이터를 조회하면 다음과 같이 id 속성 값이 test03인 문서 객체가 추가되어 있는 것을 확인할 수 있습니다.



▲id 속성 값이 test03인 사용자가 추가된 상태

06-4 | 인덱스와 메소드 사용하기

몽고디비는 데이터베이스이므로 빠른 검색을 위해 각각의 속성에 인덱스를 만들 수 있습니다. 스키마를 만들 때 각 속성의 스키마 타입에는 여러 정보가 들어갈 수 있는데 인덱스도 그중 하나입니다. 예를 들어, 다음과 같이 사용자 스키마를 만들면서 인덱스 정보를 추가할 수 있습니다.

```
var UserSchema = new mongoose.Schema({
    id: {type: String, required: true, unique: true},
    password: {type: String, required: true},
    name: {type: String, index: 'hashed'},
    age: Number,
    created_at: {type: Date, index: {unique: false, expires: '1d'}},
    updated_at: Date
});
```

id 속성의 값으로 전달된 자바스크립트 객체에는 unique 속성 값이 true로 되어 있습니다. 이렇게 하면 자동으로 인덱스가 만들어집니다. name 속성 값으로 들어가는 객체에는 index 속성 값이 hashed로 되어 있습니다. 이 속성을 사용하면 인덱스가 만들어집니다. 만들어진 날짜를 의미하는 created_at 속성에도 인덱스가 설정되었으며, 인덱스가 설정되는 값의 유효 기간까지 지정했습니다. 인덱스를 사용하면 검색 속도가 빨라지므로 조회가 필요한 속성에는 인덱스를 만들어 두는 것이 좋습니다.

위치 기반 서비스를 위해 저장되는 경위도 좌표에는 **공간 인덱싱**을 사용해야 하며 {type : [Number], index : '2d', sparse : true}와 같은 형태로 객체를 만들어 설정할 수 있습니다. 이 내용은 위치 기반 서비스 서버를 다룰 때 자세하게 설명하겠습니다.

mongoose에서는 스키마 객체에 메소드를 추가할 수도 있습니다. 메소드를 추가할 때는 static( )과 method( ) 중 하나를 사용할 수 있는데 static( )은 모델 객체에서 호출할 수 있으며 method( )는 모델 인스턴스 객체에서 호출할 수 있습니다.

메소드 이름	설명
static(name, fn)	모델 객체에서 사용할 수 있는 함수를 등록합니다. 함수의 이름과 함수 객체를 파라미터로 전달합니다.
method(name, fn)	모델 인스턴스 객체에서 사용할 수 있는 함수를 등록합니다. 함수의 이름과 함수 객체를 파라미터로 전달합니다.

[표] mongoose 스키마에서 메소드를 추가하는 두 가지 방법

사용자 리스트 조회 기능 추가하기

app3.js 파일을 복사하여 app4.js 파일을 만든 후 사용자 리스트를 조회하는 기능을 추가해 보겠습니다. users 컬렉션은 이미 만들어져 있으므로 삭제하지 않도록 주의합니다. 그리고 users보다 더 많은 칼럼을 가지고 있고 인덱스도 더 많이 만들어져 있는 users2 컬렉션을 새로운 스키마로 정의할 것이며, 사용자 리스트를 조회하는 기능도 추가해 보겠습니다.

먼저 앞에서 만든 스키마와 모델 객체를 다음과 같이 수정합니다.

참조 파일 | DatabaseExample>app4.js

```
.....

// 스키마 정의
UserSchema = mongoose.Schema({
  id: {type : String, required : true, unique: true},
  password : {type : String, required: true},
  name: {type : String, index : 'hashed'},
  age: {type : Number, 'default' : -1},
  created_at : {type : Date, index : {unique : false}, 'default' : Date.now},
  updated_at : {type : Date, index : {unique : false}, 'default' : Date.now}
});

.....
```

스키마에는 id, password, name 이외에 age, created_at, updated_at 속성을 더 추가합니다. 각각의 속성은 스키마 타입의 정보를 값으로 갖습니다. id 속성은 required 속성을 사용해서 필수 속성으로 지정하고 unique 속성으로 고유한 값을 지정합니다. name 속성은 인덱스를 만들어 둡니다. age 속성의 기본 값은 -1로 하여 값을 입력하지 않으면 -1 값이 디폴트로 들어갑니다. created_at과 updated_at 속성은 Date 타입으로 만들고 현재 시간을 디폴트 값으로 지정합니다.

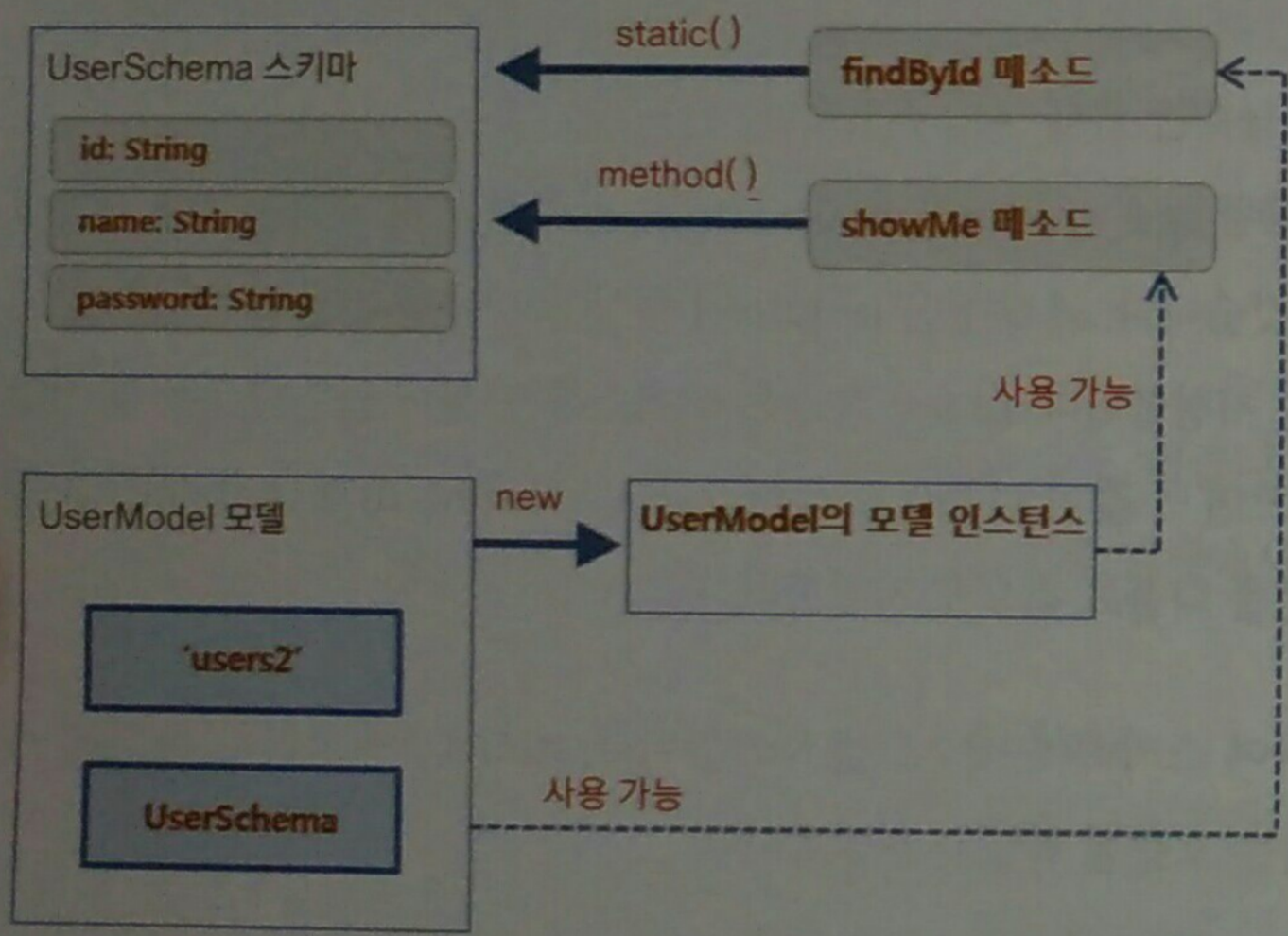
그다음에는 static( ) 메소드를 사용하여 스키마에 메소드를 추가합니다. static( ) 메소드의 첫 번째 파라미터는 함수 이름, 두 번째 파라미터는 호출될 함수가 전달됩니다.

참조 파일 | DatabaseExample>app4.js

```
.....

// 스키마에 static 메소드 추가
UserSchema.static('findById', function(id, callback) {
  return this.find({id : id}, callback);
});

UserSchema.static('findAll', function(callback) {
  return this.find({ }, callback);
});
```

```
console.log('UserSchema 정의함.');

// UserModel 모델 정의
UserModel = mongoose.model("users2", UserSchema);
console.log('UserModel 정의함.');

……
```

먼저 모델 객체에서 호출할 수 있도록 findById 함수를 추가합니다. 이 함수는 id 속성을 전달받아 데이터베이스에서 검색한 후 콜백 함수 쪽으로 결과를 넘겨줍니다. users2 컬렉션의 모든 문서 데이터를 반환하는 findAll 함수도 추가합니다. 모델 객체를 만들 때는 컬렉션의 이름을 users2로 지정합니다. 코드에는 추가하지 않았지만 만약 showMe 메소드를 method( ) 함수로 추가하면 모델 인스턴스 객체에서 사용할 수 있습니다.



▲ 스키마에 static( )으로 추가한 메소드와 method( )로 추가한 메소드의 사용 예

주의! 스키마에 추가한 메소드는 static( )으로 추가했다면 모델 객체에서 호출하여 사용할 수 있습니다. 그리고 method( )로 추가한 경우에는 모델 인스턴스 객체에서 호출하여 사용할 수 있습니다.

스키마에 사용자를 찾는 기능을 메소드로 추가했습니다. 따라서 authUser 함수 안에서 데이터베이스의 users2 컬렉션을 조회할 때 스키마에 새로 추가한 메소드를 사용하도록 코드를 변경합니다.

```
……,

// 사용자를 인증하는 함수 : 아이디로 먼저 찾고 비밀번호를 그다음에 비교
var authUser = function(database, id, password, callback) {
  console.log('authUser 호출됨.');

  // 1. 아이디를 사용해 검색
  UserModel.findById(id, function(err, results) {
    if(err) {
      callback(err, null);
      return;
    }

    console.log('아이디 [%s]로 사용자 검색 결과', id);
    console.dir(results);

    if(results.length > 0) {
      console.log('아이디와 일치하는 사용자 찾음.');

      // 2. 비밀번호 확인
      if(results[0]._doc.password == password) {
        console.log('비밀번호 일치함');
        callback(null, results);
      } else {
        console.log('비밀번호 일치하지 않음');
        callback(null, null);
      }

    } else {
      console.log("아이디와 일치하는 사용자를 찾지 못함.");
      callback(null, null);
    }

  });
}

……,
```

모델 객체의 findById( ) 메소드를 호출할 때는 id 값과 콜백 함수를 전달합니다. 콜백 함수에서 결과 데이터를 배열로 받으면 그 배열 객체에 데이터가 있는지 확인합니다. 데이터가 있는 경우에는 첫 번째 배열 요소의 _doc 속성을 참조합니다. **_doc 속성**은 각 문서 객체의 정보를 담고 있어 그 안에 있는 password

속성 값을 확인할 수 있습니다. findById( ) 메소드는 사용자의 아이디만으로 조회하는 메소드이기 때문에 비밀번호는 그다음에 일치 여부를 확인합니다.

이번에는 사용자 리스트를 조회하는 메소드를 추가합니다. 클라이언트에서는 /process/listuser로 요청하므로 이 패스를 라우팅하여 처리하는 함수를 추가합니다.

참조 파일 | DatabaseExample>app4.js

```
.....

// 사용자 리스트 함수
app.post('/process/listuser', function(req, res) {
  console.log('/process/listuser 호출됨.');

  if(database) {
    // 1. 모든 사용자 검색
    UserModel.findAll(function(err, results) {
      if(err) {
        callback(err, null);
        return;
      }

      if(results) {
        console.dir(results);

        res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
        res.write('<h2>사용자 리스트</h2>');
        res.write('<div><ul>');

        for(var i = 0; i < results.length; i++) {
          var curId = results[i]._doc.id;
          var curName = results[i]._doc.name;
          res.write('    <li>#' + i + ' : ' + curId + ', ' + curName + '</li>');
        }

        res.write('</ul></div>');
        res.end();
      } else {
        res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
        res.write('<h2>사용자 리스트 조회 실패</h2>');
        res.end();
      }

.....
```

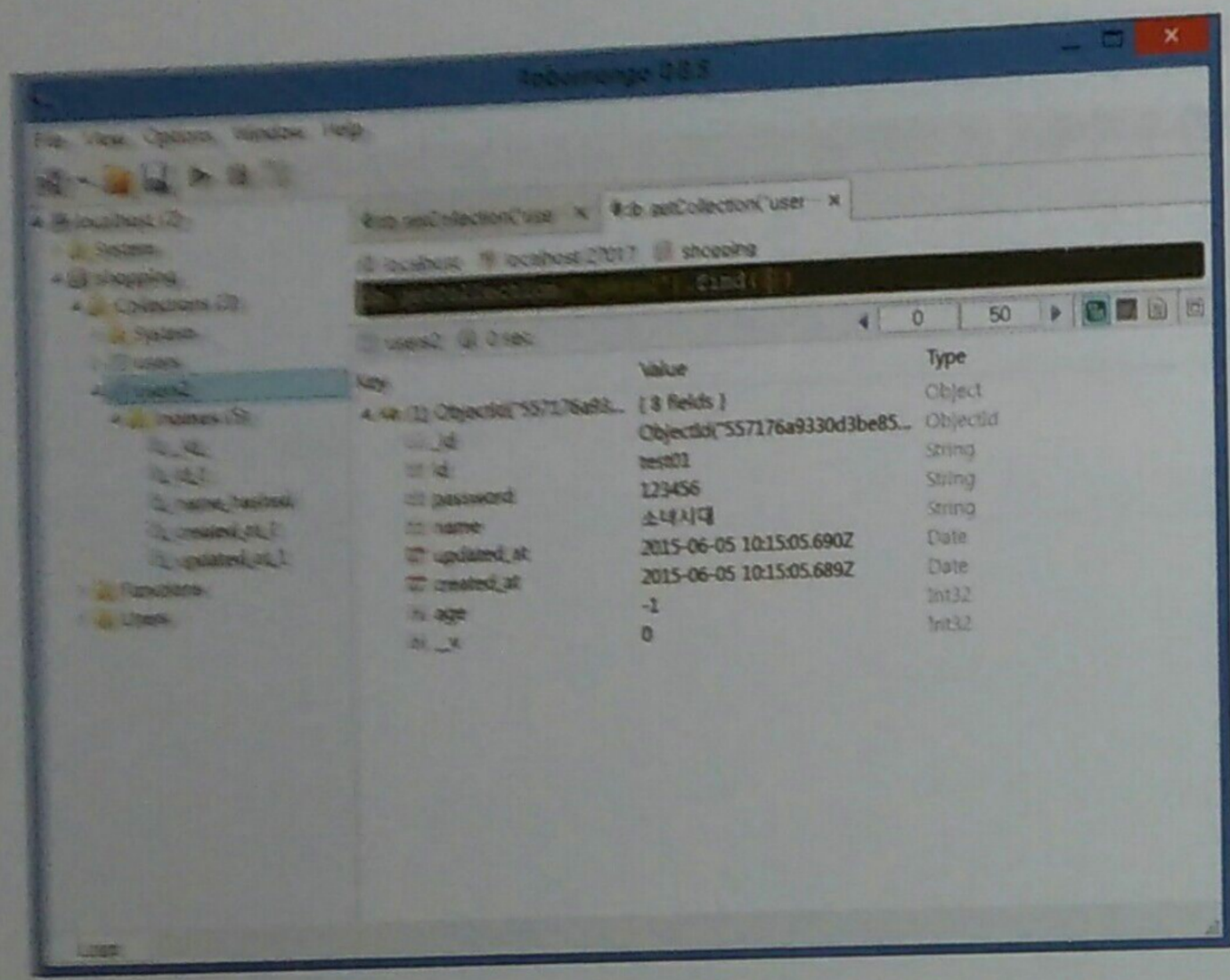
이 함수 안에서는 모델 객체의 findAll( ) 함수를 호출하고 그 결과를 응답으로 전송합니다. 서버 쪽에 사용자 리스트 기능을 추가했으니 웹 브라우저에서 조회할 웹 문서를 [public] 폴더에 만듭니다. listuser.html 파일을 만든 후 다음과 같이 웹 문서를 구성하는 코드를 입력합니다.

참조 파일 | DatabaseExample>/public/listuser.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>사용자 리스트 테스트</title>
  </head>
<body>
  <h1>사용자 리스트</h1>
  <br>
  <form method="post" action="/process/listuser">
    <table>
      <tr>
        <td><label>아래 [전송] 버튼을 누르세요.</label></td>
      </tr>
    </table>
    <input type="submit" value="전송" name="">
  </form>
</body>
</html>
```

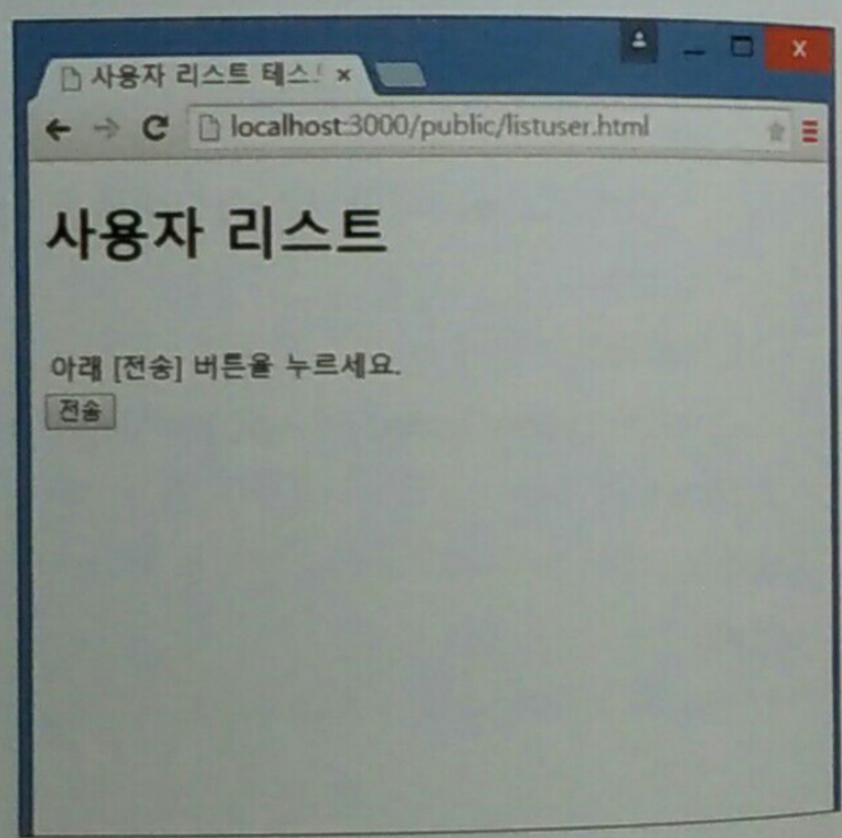
간단한 안내글과 함께 [전송] 버튼이 화면에 표시되는 웹 문서가 만들어집니다. [전송] 버튼을 누르면 POST 방식으로 서버에 등록된 /process/listuser 패스를 호출합니다.

이제 app4.js 파일을 실행한 후 웹 브라우저에서 http://localhost:3000/public/adduser.html 사이트에 접속한 후 사용자를 등록합니다. app4.js 파일에는 users2 컬렉션에 문서 데이터를 추가하도록 설정되어 있으므로 아이디에 test01, 비밀번호에 123456, 이름에 소녀시대를 입력하고 [전송] 버튼을 누릅니다. 그러면 서버의 데이터베이스에 데이터가 저장됩니다. 데이터를 성공적으로 저장한 후 Robomongo에서 컬렉션을 조회하면 다음과 같이 id가 test01인 문서 데이터가 조회됩니다.



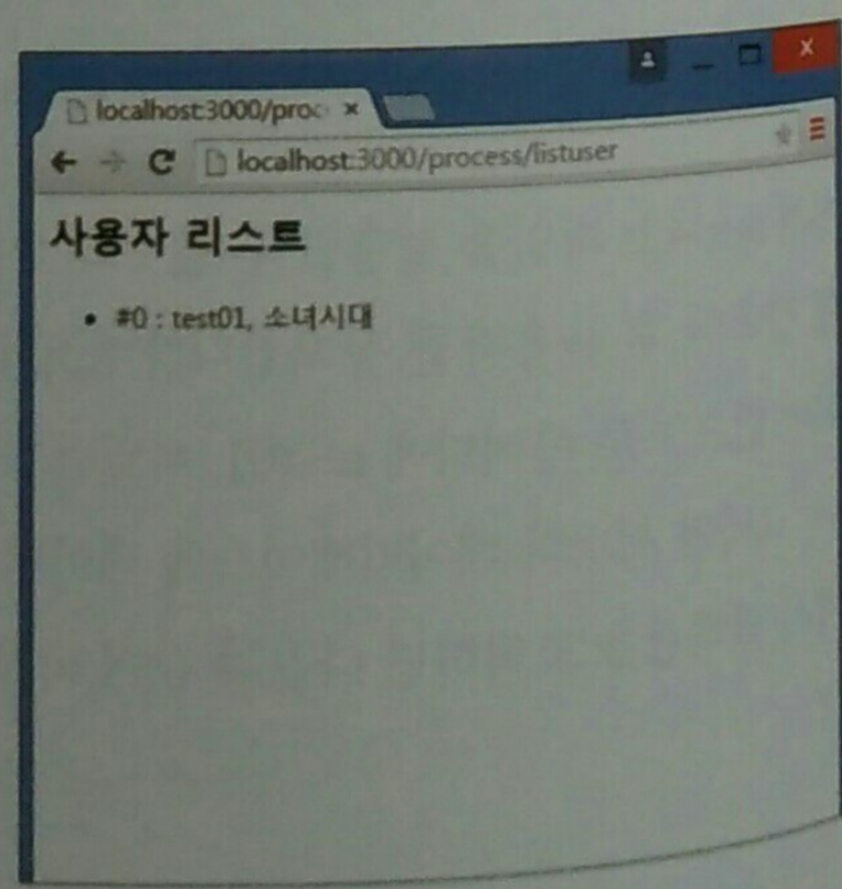
▲ /public/adduser.html 페이지에서 사용자 등록 요청하여 데이터 추가된 것을 Robomongo에서 조회한 화면

users 컬렉션에 추가했던 문서 객체와 달리 users2 컬렉션에는 속성과 인덱스가 더 많이 만들어진 것을 확인할 수 있습니다. 속성 값 중에서 created_at과 updated_at 속성 값에는 현재 시간이 들어 있습니다. 이제 새로 추가한 사용자 리스트 기능을 확인해 보겠습니다. 웹 브라우저에서 http://localhost:3000/public/listuser.html 사이트에 접속하면 다음과 같은 화면이 표시됩니다.



/public/listuser.html 사이트를 조회했을 때 나타나는 사용자 리스트 요청 화면 ▶

[전송] 버튼을 누르면 서버의 데이터베이스에 저장된 사용자 리스트가 표시합니다. 한 명만 등록되어 있으므로 한 명의 사용자만 확인할 수 있습니다.



사용자 리스트 조회 결과 화면 ▶

이제 사용자를 조회하는 기능까지 동작하니 사용자 정보를 관리하는 기본 기능은 어느 정도 만들었습니다.

06-5 | 비밀번호 암호화하여 저장하기

로그인에 사용되는 비밀번호를 암호화하려면 어떻게 해야 할까요? 몽구스를 사용하면 스키마나 모델 객체에 함수를 추가하고 필요할 때 그 함수를 실행할 수 있으니 비밀번호를 암호화하여 저장하는 기능을 구현하면 됩니다. 이 기능을 어떻게 구현하는지 잘 살펴보면 몽구스와 몽고디비에 대해 더 깊이 이해할 수 있을 것입니다.

virtual 함수 사용하기

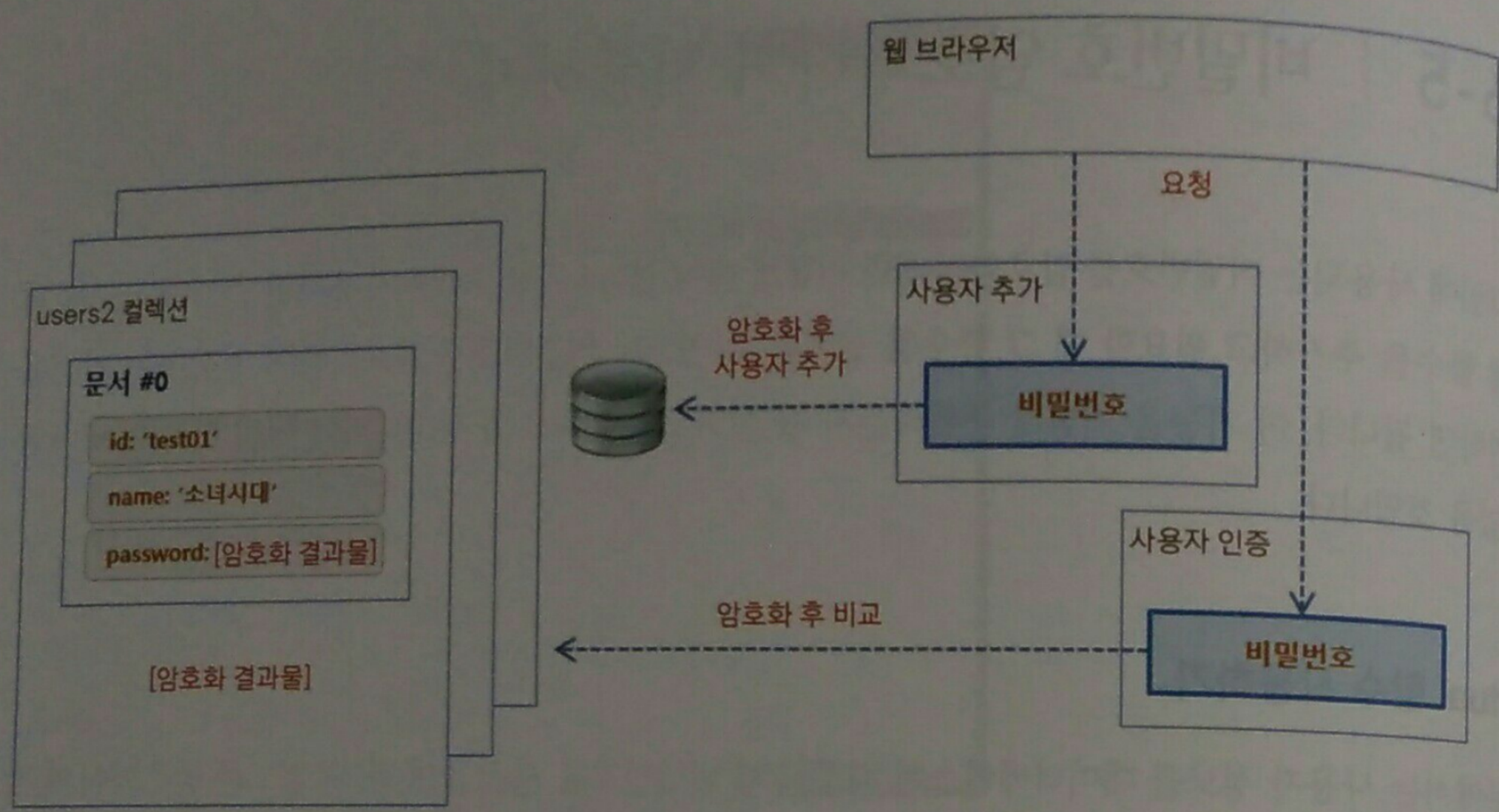
실무에서는 사용자 정보를 데이터베이스에 저장할 때 비밀번호를 사람들이 볼 수 없도록 암호화하여 저장하는 경우가 많습니다. 이때 비밀번호는 단방향으로 암호화하여 원본 비밀번호 문자열을 알 수 없도록 만듭니다.

정박사의 한마디 단방향 암호화라는 것이 어떤 의미일까요?

단방향 암호화란 말 그대로 한 방향으로만 암호화가 가능한 것입니다. 이와 반대되는 것이 양방향 암호화입니다. 예를 들어, 단방향 암호화는 hello라는 글자를 암호화했다면 암호화된 데이터는 다시 원본인 hello로 복구할 수 없습니다. 원본 글자로 복구하는 과정을 복호화(Description)라고 하는데 복호화할 수 있는 방법이 없으므로 한쪽 방향으로만 암호화(Encryption)가 가능하다는 의미로 단방향 암호화라고 부릅니다.

이에 반해 양방향 암호화는 hello라는 글자를 암호화했다면 암호화된 데이터를 다시 원본인 hello로 복호화할 수 있습니다.

사용자 인증 과정에서 사용자가 입력한 비밀번호를 암호화한 후 데이터베이스에 저장되어 있는 암호화된 비밀번호와 비교하는 과정을 거칩니다.

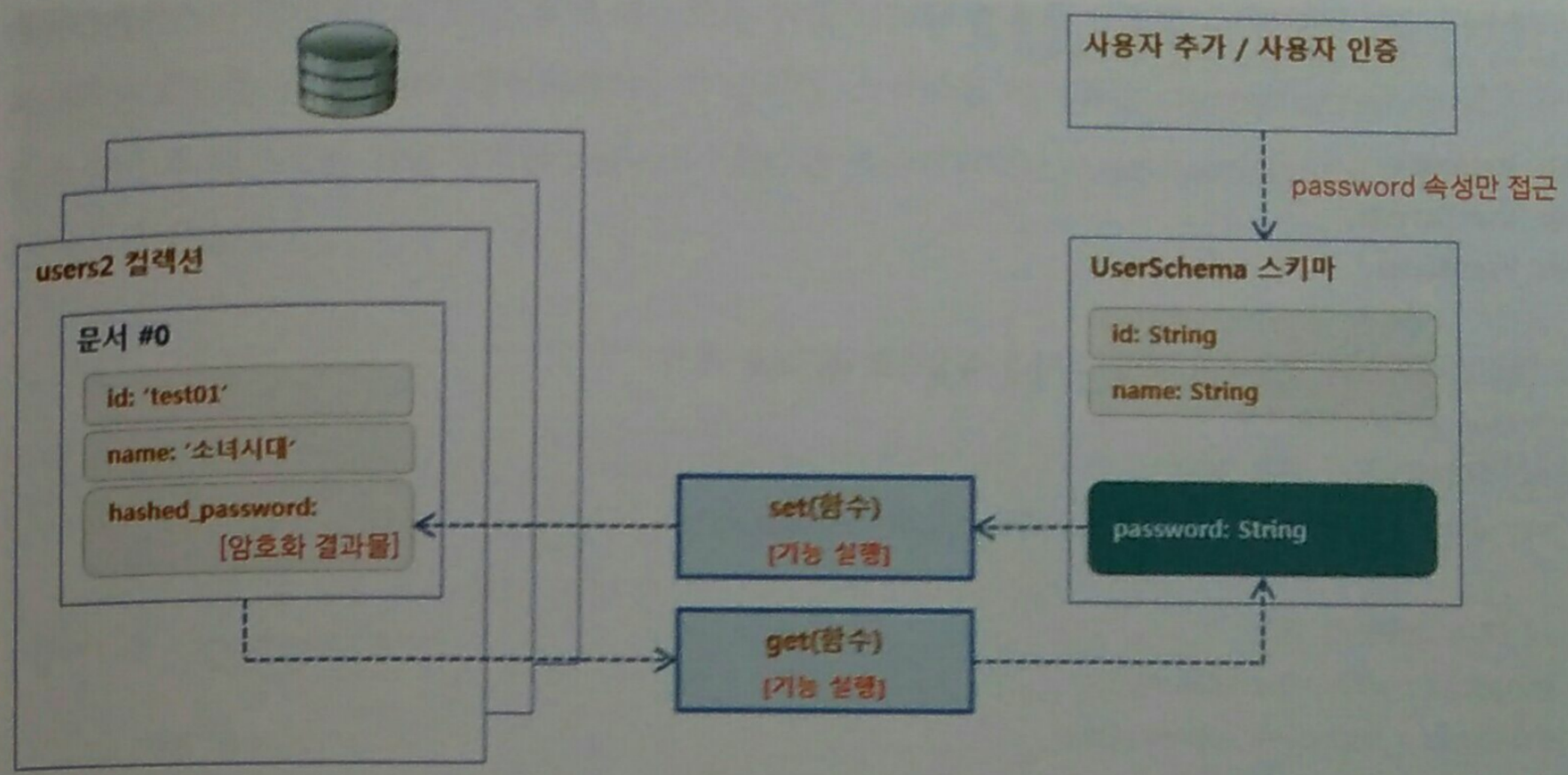


▲ 비밀번호를 암호화하여 저장한 후 사용자 인증에서 비교하는 과정

사용자를 추가할 때는 비밀번호를 암호화한 후 데이터베이스에 저장합니다. 그다음 사용자 인증을 진행할 때는 클라이언트가 보내 온 비밀번호를 암호화한 후 데이터베이스에 저장된 암호화된 비밀번호와 비교합니다.

비밀번호 암호화 작업은 사용자 정보를 저장할 때 이루어져야 합니다. mongoose에서 제공하는 virtual() 함수를 사용하면 이 과정을 더 쉽게 처리할 수 있습니다. virtual( ) 함수는 문서 객체에 실제로 저장되는 속성이 아니라 가상의 속성을 지정할 수 있습니다. 예를 들어, 비밀번호를 저장하기 위해 문서 객체에 만들어지는 속성 이름이 hashed_password라면, 실제로 저장되는 속성은 아니지만 password 속성을 만든 후 이 속성에 값을 넣어 저장할 때 hashed_password 속성으로 저장되도록 만들 수 있습니다. 물론 password 속성 값은 hashed_password 속성 값과 같을 필요는 없으며, 필요한 과정을 거친 후에 저장할 수 있습니다. 또 password 속성 값을 조회할 때도 원하는 과정을 거친 후에 조회할 수 있습니다.

다음 그림을 보면 사용자를 추가하거나 인증할 때 password 속성에 접근하지만 실제 데이터베이스에는 password 속성이 없고 hashed_password 속성만 존재합니다.



▲ virtual( ) 함수로 지정된 속성으로 데이터베이스에 저장하고 조회하기

password 속성에 값을 넣은 후 문서 객체를 저장할 때 set( ) 메소드로 지정한 함수가 필요한 작업을 수행하며, 문서 객체를 조회할 때는 get( ) 메소드로 지정한 함수가 실행됩니다.

스키마 객체의 virtual( ) 함수 사용법 알아보기

스키마 객체의 virtual( ) 함수를 사용하는 방법을 알아보기 위해 프로젝트 폴더 안에 virtual_test1.js 파일을 만든 후 데이터베이스에 문서 객체를 저장하거나 조회하는 코드를 입력합니다.

이 파일에는 express 모듈을 사용하지 않고 단순히 노드로 테스트하는 코드를 입력합니다. 데이터베이스에는 users4 컬렉션으로 문서 데이터를 저장하거나 조회하고 id와 name 속성을 포함하도록 만들 것입니다. virtual( ) 함수를 사용하여 info 속성을 추가한 후 띄어쓰기(' ')문자로 이어진 사용자 아이디와 비밀번호 값을 info 속성에 저장합니다. info 속성에 값을 저장하면 그 속성 값을 띄어쓰기 문자로 분리한 후 id와 name 속성에 각각 값을 나누어 저장합니다. 그리고 info 속성 값을 조회하여 가져올 때는 id와 name 속성 값을 붙여서 가져옵니다.

참조 파일 | DatabaseExample>virtual_test1.js

```
.....

//===== 모듈 불러들이기 =====//
var mongodb = require('mongodb');
var mongoose = require('mongoose');
```

```
//====== 데이터베이스 연결 ======//

var database;
var UserSchema;
var UserModel;

// 데이터베이스에 연결하고 응답 객체의 속성으로 db 객체 추가
function connectDB() {
  // 데이터베이스 연결 정보
  var databaseUrl = 'mongodb://localhost:27017/shopping';

  // 데이터베이스 연결
  mongoose.connect(databaseUrl);
  database = mongoose.connection;

  database.on('error', console.error.bind(console, 'mongoose connection error.'));
  database.on('open', function() {
    console.log('데이터베이스에 연결되었습니다. : ' + databaseUrl);

    // user 스키마 및 모델 객체 생성
    createUserSchema();

    // test 진행함
    doTest();

  });
  database.on('disconnected', connectDB);
}

---
```

먼저 require( ) 메소드로 mongodb와 mongoose 모듈을 불러들입니다. 데이터베이스 객체, 스키마 객체, 모델 객체를 할당할 변수를 선언한 후 connectDB( ) 함수를 정의합니다. 이 connectDB( ) 함수는 코드의 가장 마지막에 호출합니다. connectDB( ) 함수 안에서 데이터베이스에 연결하는 코드는 app4.js 파일에서 데이터베이스를 연결하기 위해 사용한 코드와 비슷합니다.

데이터베이스가 정상적으로 열린 후에 테스트를 위해 정의한 함수를 호출할 수 있도록 open 이벤트를 처리하는 코드를 입력합니다. 데이터베이스가 열리면, 스키마를 정의하는 createUserSchema( ) 함수를 호출한 후 테스트를 위해 정의한 **doTest( ) 함수**를 호출합니다. createUserSchema( ) 함수의 코드는 다음과 같습니다.

참조 파일 | DatabaseExample>virtual_test1.js

```
……

// user 스키마 및 모델 객체 생성
function createUserSchema() {

  // 스키마 정의
  // password를 hashed_password로 변경, default 속성 모두 추가, salt 속성 추가
  UserSchema = mongoose.Schema({
    id : {type : String, required : true, unique : true},
    name : {type : String, index : 'hashed', 'default':''},
    age : {type : Number, 'default': -1},
    created_at : {type : Date, index: {unique : false}, 'default' : Date.now},
    updated_at : {type : Date, index: {unique : false}, 'default' : Date.now}
  });

  // info를 virtual 메소드로 정의
  UserSchema
    .virtual('info')
    .set(function(info) {
      var splitted = info.split(' ');
      this.id = splitted[0];
      this.name = splitted[1];
      console.log('virtual info 설정함 : %s, %s', this.id, this.name);
    })
    .get(function() {return this.id + ' ' + this.name});

  console.log('UserSchema 정의함.');

  // UserModel 모델 정의
  UserModel = mongoose.model("users4", UserSchema);
  console.log('UserModel 정의함.');
}

……
```

스키마 객체에는 id, name을 포함해 다섯 개의 속성이 정의됩니다. 그다음에는 스키마 객체의 virtual() 함수를 사용해 info 속성을 지정합니다. 이 속성은 지금부터 스키마에 정의된 다른 속성처럼 사용할 수 있습니다. virtual( ) 함수로 속성을 지정하면 set( ) 메소드와 get( ) 메소드를 호출할 수 있습니다. set( ) 메소드에서는 전달된 파라미터를 띄어쓰기 기호로 분리하기 위해 **split( ) 메소드**를 사용합니다. 예를 들어, 'test01 소녀시대'라는 글자를 info 속성 값으로 지정하면 두 개의 아이템을 가지는 배열 객체가 만들어집니다. 그 첫 번째 아이템은 id 속성에, 두 번째 아이템은 name 속성에 저장됩니다. get( ) 메소드에서는 id와 name 속성 값을 붙여서 반환합니다. 스키마를 정의했다면 이 스키마로 모델 객체를 만듭니다. 모델 객체는 데이터베이스의 users4 컬렉션을 사용하도록 지정합니다.

데이터베이스가 연결된 후에 호출되는 doTest( ) 함수의 코드는 다음과 같습니다.

참조 파일 | DatabaseExample>virtual_test1.js

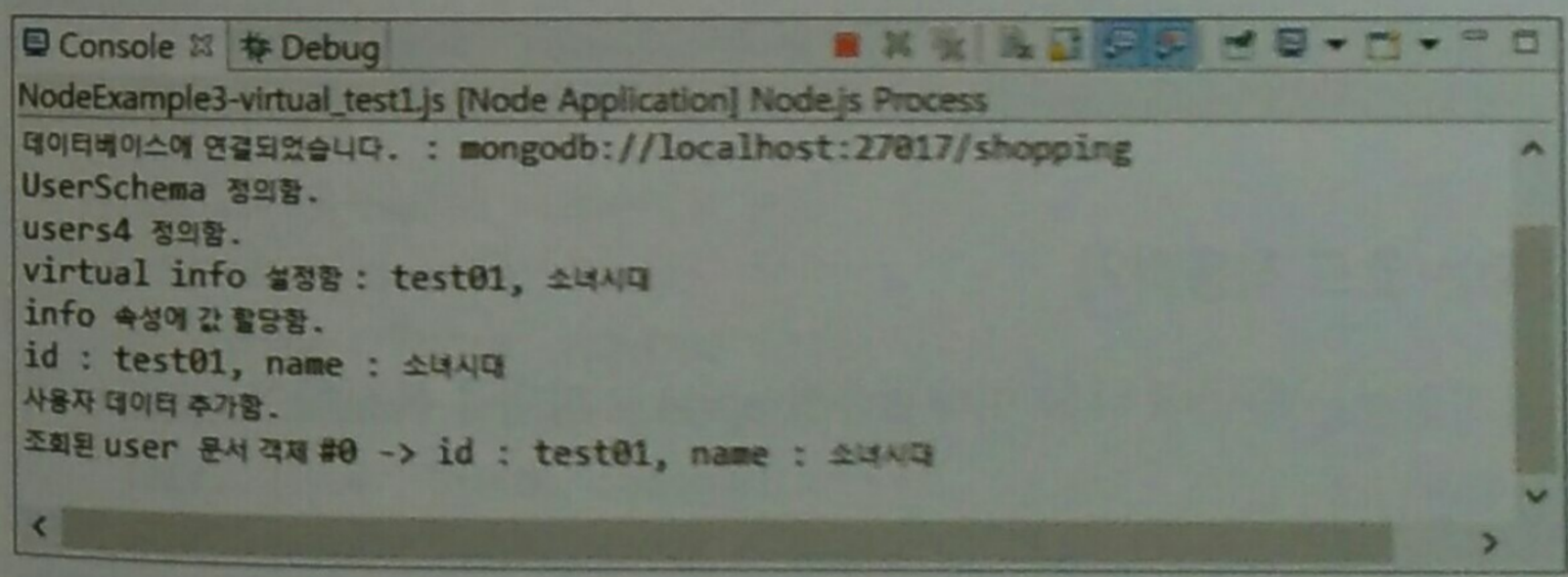
```
.....

function doTest() {
  // UserModel 인스턴스 생성
  // id, name 속성은 할당하지 않고 info 속성만 할당함
  var user = new UserModel({"info" : 'test01 소녀시대'});

  // save()로 저장
  user.save(function(err) {
    if(err) {throw err;}

    console.log("사용자 데이터 추가함.");

    findAll();
  });

  console.log('info 속성에 값 할당함.');
  console.log('id : %s, name : %s', user.id, user.name);
}

.....
```

데이터베이스에 문서 데이터를 저장하기 위해 new 연산자로 모델 인스턴스 객체를 만듭니다. 모델 인스턴스 객체를 만들 때 전달하는 속성에는 info 속성만 지정합니다. 모델 인스턴스 객체를 저장할 때 info 속성의 값을 띄어쓰기 기호로 분리하여 저장하므로 id나 name 속성은 지정할 필요가 없습니다. 문서 데이터가 저장되면 findAll( ) 메소드를 실행하여 데이터를 조회합니다. 다음은 findAll( ) 메소드의 코드입니다.

```
……,

function findAll() {
  UserModel.find({}, function(err, results) {
    if(err) {throw err;}

    if(results) {
      console.log('조회된 user 문서 객체 #0 -> id : %s, name : %s',
                  results[0]._doc.id, results[0]._doc.name);
    }
  });
}

……,
```

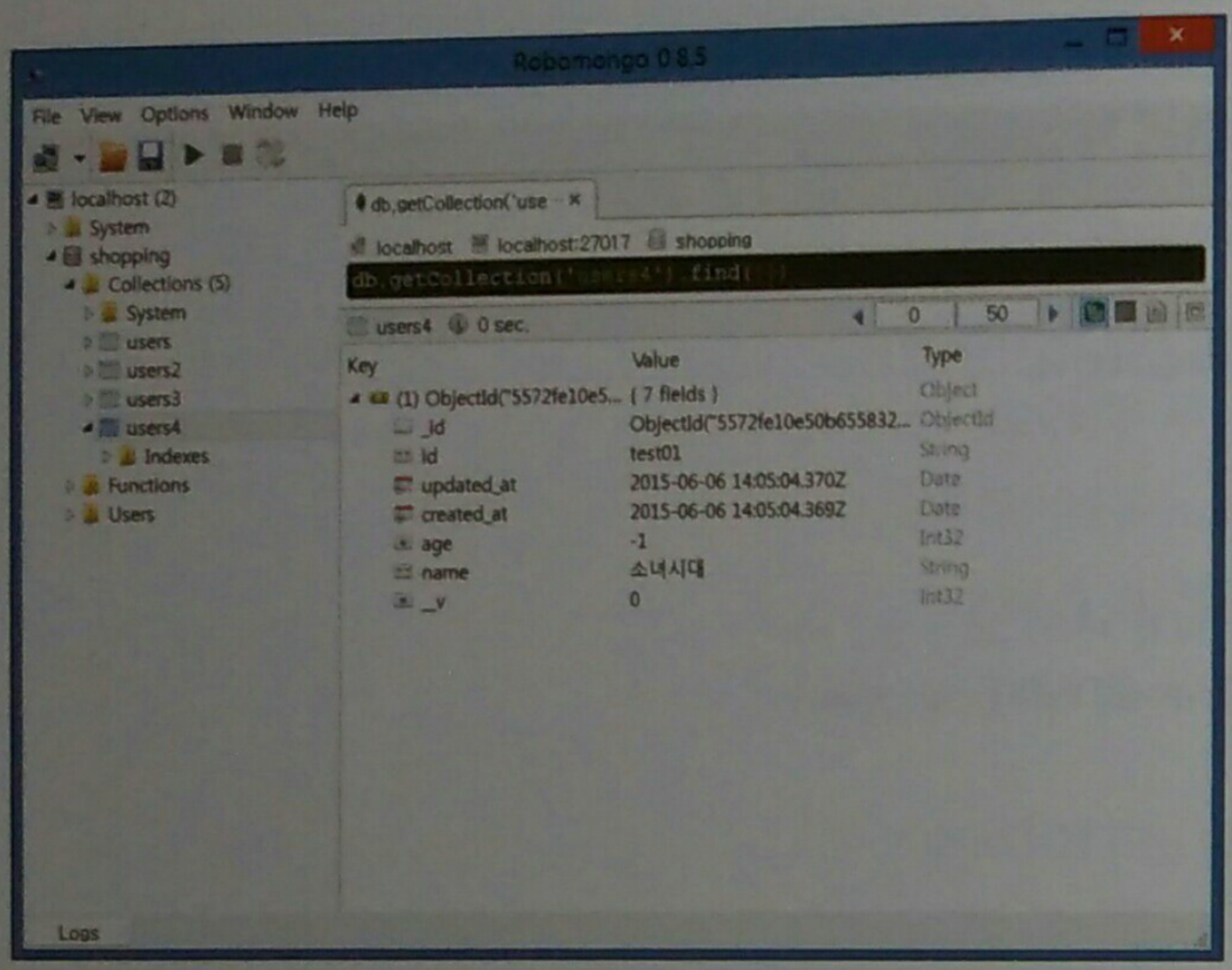
모든 데이터를 조회한 후에는 id와 name 속성 값을 확인하도록 작성되어 있습니다. 코드의 마지막 부분에서는 connectDB( ) 메소드를 호출하여 데이터베이스 연결, 문서 스키마 정의, 문서 데이터 저장, 문서 데이터 조회를 순서대로 실행합니다.

virtual_test1.js 파일을 실행하면 다음과 같이 users4 테이블에 문서 객체를 저장한 후 조회하면서 콘솔 창에 메시지가 표시됩니다.



▲ virtual( ) 함수로 지정한 info 속성을 사용해 데이터 저장할 때의 콘솔 메시지 표시

데이터베이스에 실제로 저장된 속성들은 Robomongo 화면에서 확인할 수 있습니다.



▲ 새로 만든 users4 컬렉션과 저장된 문서 데이터

id와 name 속성 값이 잘 저장된 것을 볼 수 있습니다.

정박사의 한마디 오류가 발생하고 문서 데이터가 저장되지 않나요?

스키마를 정의할 때 id 속성을 unique한 속성, 즉 같은 값을 가진 값이 중복되어 저장될 수 없도록 설정했습니다. 따라서 virtual_test1.js 파일을 한 번 실행한 후 다시 실행하면 키가 중복된다는 오류 메시지가 표시되며 데이터는 저장되지 않을 수 있습니다. 이런 경우에는 Robomongo 프로그램에서 users4 컬렉션에 저장된 문서 데이터를 삭제한 후 virtual_test1.js 파일을 다시 실행하면 됩니다.

비밀번호 암호화하여 저장하는 코드 적용하기

virtual( ) 함수로 가상 속성을 지정하는 방법에 대해 이해했다면 app4.js 파일을 복사하여 app5.js 파일을 만들고 crypto 모듈을 설치합니다.

```
% npm install crypto -- save
```

그리고 비밀번호를 암호화하여 저장하는 다음 코드를 입력합니다.

참조 파일 | DatabaseExample>app5.js

```
……,

// crypto 모듈 불러들이기
var crypto = require('crypto');

……,

function connectDB() {

……,

  database.on('open', function() {
    console.log('데이터베이스에 연결되었습니다. : ' + databaseUrl);

    // user 스키마 및 모델 객체 생성
    createUserSchema();

  });

……,

}

// user 스키마 및 모델 객체 생성
function createUserSchema() {

  // 스키마 정의
  // password를 hashed_password로 변경, default 속성 모두 추가, salt 속성 추가
  UserSchema = mongoose.Schema({
    id : {type : String, required : true, unique : true, 'default' : ' '},
    hashed_password : {type : String, required : true, 'default' : ' '},
    salt : {type : String, required : true},
    name : {type : String, index : 'hashed', 'default' : ' '},
    age : {type : Number, 'default' : -1},
    created_at : {type : Date, index : {unique : false}, 'default' : Date.now},
    updated_at : {type : Date, index : {unique : false}, 'default' : Date.now}
  });

……,
```

노드는 암호화를 위해 crypto 모듈을 제공합니다. 이 모듈을 사용해서 암호화할 것이므로 require( ) 메소드로 crypto 모듈을 불러들입니다. 스키마를 정의할 때는 기존에 만들어 둔 id, name, age, created_at, updated_at 속성 외에 hashed_password와 salt 속성을 추가합니다. hashed_password 속성에

는 비밀번호를 암호화하여 저장하고 salt 속성에는 암호화에 사용되는 salt 값을 저장합니다. 암호화 과정에서는 일종의 키(Key) 값으로 salt 값을 사용하는데 이 값이 계속 변경되도록 만들 것입니다. 따라서 암호화된 비밀번호와 함께 데이터베이스에 저장합니다. 기존에 사용한 password 속성은 더 이상 사용하지 않도록 설정하고 virtual( ) 함수를 사용해 가상 속성으로 추가합니다.

다음은 virtual( ) 함수를 사용해 password 속성을 스키마에 추가하는 코드입니다.

참조 파일 | DatabaseExample>app5.js

```
……

// password를 virtual 메소드로 정의 : MongoDB에 저장되지 않는 편리한 속성임. 특정 속성을 지정하고
set, get 메소드를 정의함
  UserSchema
    .virtual('password')
    .set(function(password) {
      this._password = password;
      this.salt = this.makeSalt();
      this.hashed_password = this.encryptPassword(password);
      console.log('virtual password 호출됨 : ' + this.hashed_password);
    })
    .get(function() {return this._password});

……
```

virtual( ) 메소드를 호출할 때는 가상 속성의 이름으로 password를 전달합니다. set( ) 함수는 전달된 문자열을 암호화된 비밀번호로 만든 후 hashed_password 속성으로 저장합니다. 암호화를 진행하기 위해 makeSalt( )와 encryptPassword( ) 메소드를 호출합니다. 이 두 메소드는 다음과 같이 정의합니다.

참조 파일 | DatabaseExample>app5.js

```
……

// 스키마에 모델 인스턴스에서 사용할 수 있는 메소드 추가
// 비밀번호 암호화 메소드
UserSchema.method('encryptPassword', function(plainText, inSalt) {
  if(inSalt) {
    return crypto.createHmac('sha1', inSalt).update(plainText).digest('hex');
  } else {
```

```
    return crypto.createHmac('sha1', this.salt).update(plainText).digest('hex');
  }
});

// salt 값 만들기 메소드
UserSchema.method('makeSalt', function() {
  return Math.round((new Date().valueOf() * Math.random())) + '';
});

// 인증 메소드 - 입력된 비밀번호와 비교 (true/false 리턴)
UserSchema.method('authenticate', function(plainText, inSalt, hashed_password) {
  if(inSalt) {
    console.log('authenticate 호출됨 : %s -> %s : %s', plainText,
                this.encryptPassword(plainText, inSalt), hashed_password);
    return this.encryptPassword(plainText, inSalt) == hashed_password;
  } else {
    console.log('authenticate 호출됨 : %s -> %s : %s', plainText,
                this.encryptPassword(plainText), this.hashed_password);
    return this.encryptPassword(plainText) == this.hashed_password;
  }
});

.....
```

makeSalt( ) 메소드와 encryptPassword( ) 메소드는 스키마 객체의 method( ) 메소드를 사용해 모델 인스턴스 객체에서 호출할 수 있는 메소드로 추가됩니다. makeSalt( ) 메소드는 Math.random( ) 메소드를 호출하면서 랜덤 값을 하나 만들어 냅니다. encryptPassword( ) 메소드는 비밀번호와 salt 값을 파라미터로 전달받은 후 crypto 모듈로 암호화합니다.

authenticate 메소드가 하나 더 추가되었는데 이 메소드는 파라미터로 전달된 비밀번호와 암호화된 비밀번호가 같은지 비교합니다. 데이터베이스에 저장할 문서 데이터 중에서 필수 속성인데도 값이 없는 경우를 체크하기 위해 validate( ) 함수를 사용할 수 있습니다.

다음 코드는 id나 name 속성 값이 없는 경우를 확인하기 위해 추가된 것입니다.

참조 파일 | DatabaseExample>app5.js

```
……

// 필수 속성에 대한 유효성 확인(길이 값 체크)
UserSchema.path('id').validate(function(id) {
  return id.length;
}, 'id 칼럼의 값이 없습니다.');

UserSchema.path('name').validate(function(name) {
  return name.length;
}, 'name 칼럼의 값이 없습니다.');

……
```

스키마 객체의 path( ) 메소드를 호출한 후 validate( ) 메소드를 호출하면 유효한 값인지 확인할 수 있습니다. id 속성 값을 확인하여 입력된 값이 유효한지 알려 줍니다.

이제 사용자를 추가하는 addUser( ) 함수의 코드를 수정합니다.

참조 파일 | DatabaseExample>app5.js

```
……

// 사용자를 등록하는 함수
var addUser = function(database, id, password, name, callback) {
  console.log('addUser 호출됨.');

  // UserModel 인스턴스 생성
  var user = new UserModel({"id" : id, "password" : password, "name" : name});

  // save()로 저장
  user.save(function(err) {
    if(err) {
      callback(err, null);
      return;
    }

    console.log("사용자 데이터 추가함.");
    callback(null, user);

  });
}
```

먼저 new 연산자로 모델 인스턴스 객체를 만듭니다. 모델 인스턴스 객체를 만들 때는 password 속성에 클라이언트로부터 전달받은 비밀번호 값을 넣습니다. save( ) 메소드를 호출하여 저장하면 비밀번호가 암호화된 후 hashed_password 속성에 저장됩니다. 사용자를 인증하는 authUser( ) 함수의 코드도 스키마에 추가한 authenticate 함수를 호출하는 방식으로 수정합니다.

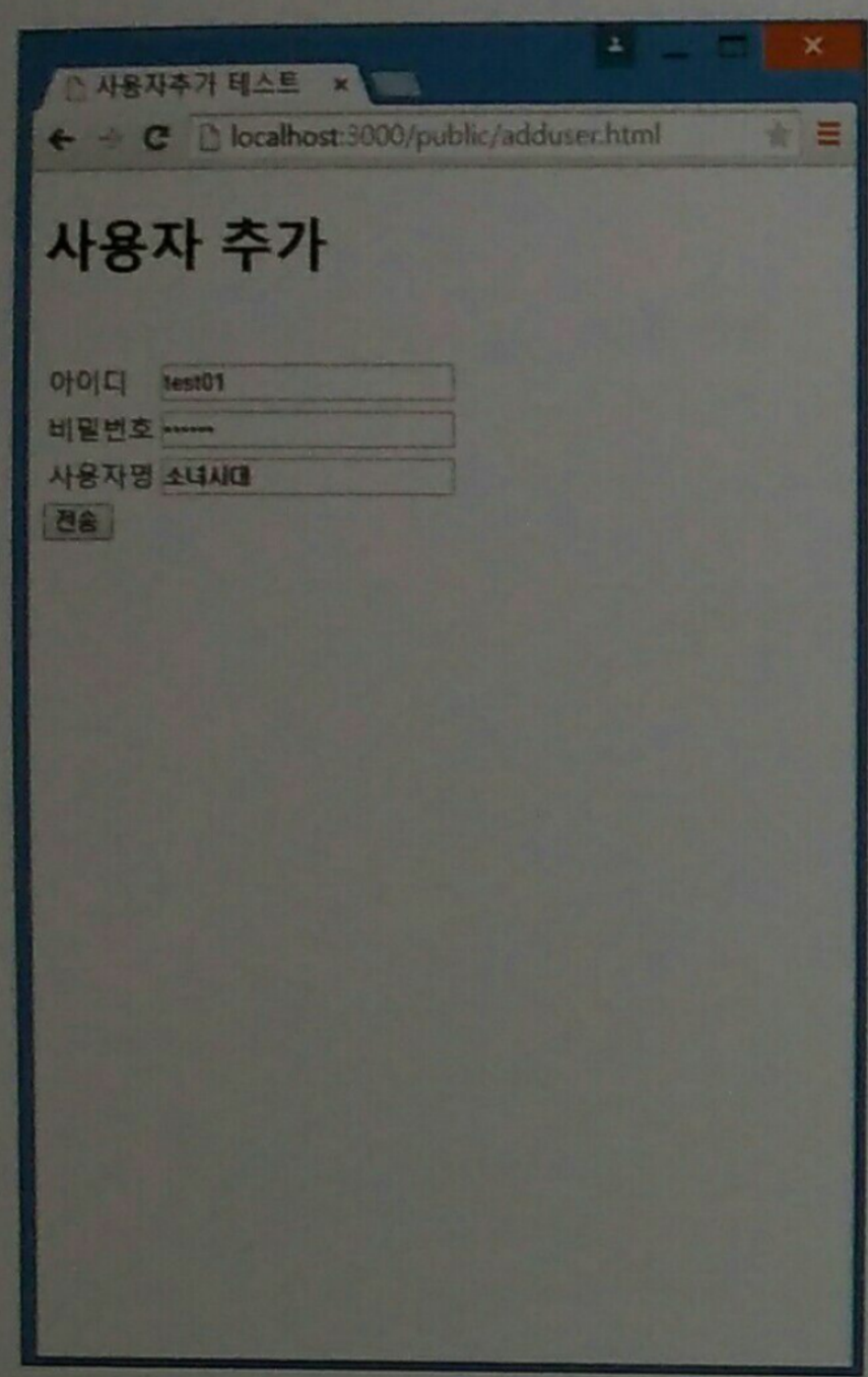
참조 파일 | DatabaseExample>app5.js

```
……,

// 사용자를 인증하는 함수 : 아이디로 먼저 찾고 비밀번호를 그다음에 비교
var authUser = function(database, id, password, callback) {
  console.log('authUser 호출됨.');

  // 1. 아이디를 사용해 검색
  UserModel.findById(id, function(err, results) {

  ……,

    if(results.length > 0) {
      console.log('아이디와 일치하는 사용자 찾음.');

      // 2. 비밀번호 확인 : 모델 인스턴스를 객체를 만들고 authenticate() 메소드 호출
      var user = new UserModel({id : id});
      var authenticated = user.authenticate(password, results[0]._doc.salt,
                                            results[0]._doc.hashed_password);

      if(authenticated) {
        console.log('비밀번호 일치함');
        callback(null, results);
      } else {
        console.log('비밀번호 일치하지 않음');
        callback(null, null);
      }

  ……,
```

모델 인스턴스 객체를 만들 때는 id 속성에만 값을 넣어 둡니다. 모델 인스턴스 객체의 authenticate 메소드를 호출할 때는 클라이언트로부터 전달받은 암호와 함께 아이디를 이용해 찾아낸 문서 객체의 salt 값과 hashed_password 값을 파라미터로 전달합니다.

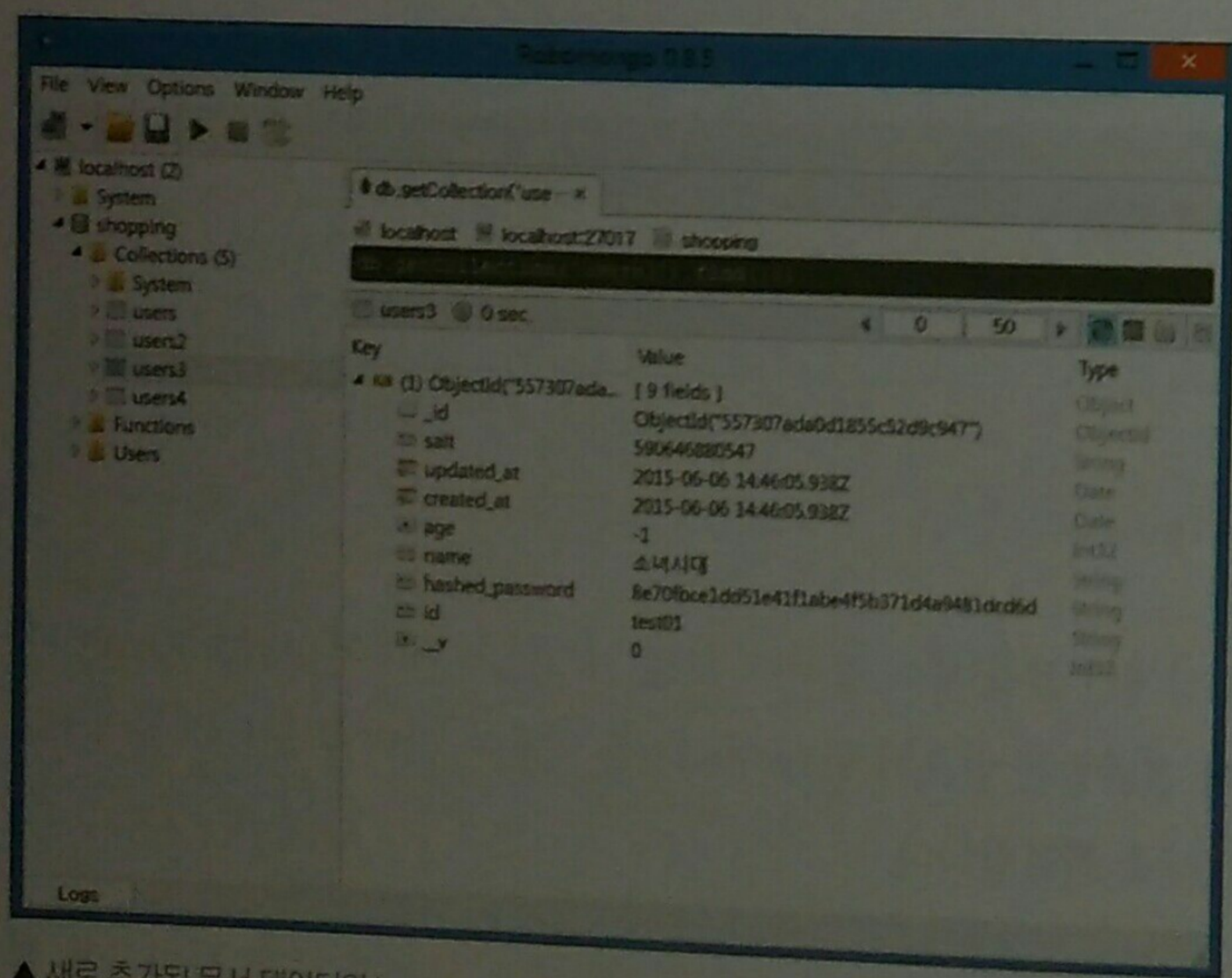
이제 코드를 모두 수정했으니 app5.js 파일을 실행한 후 웹 브라우저를 열고 사용자 등록 사이트에 접속합니다.

▶ http://localhost:3000/public/adduser.html



▲ 웹 브라우저에서 사용자 등록 사이트를 열고 값을 입력한 화면

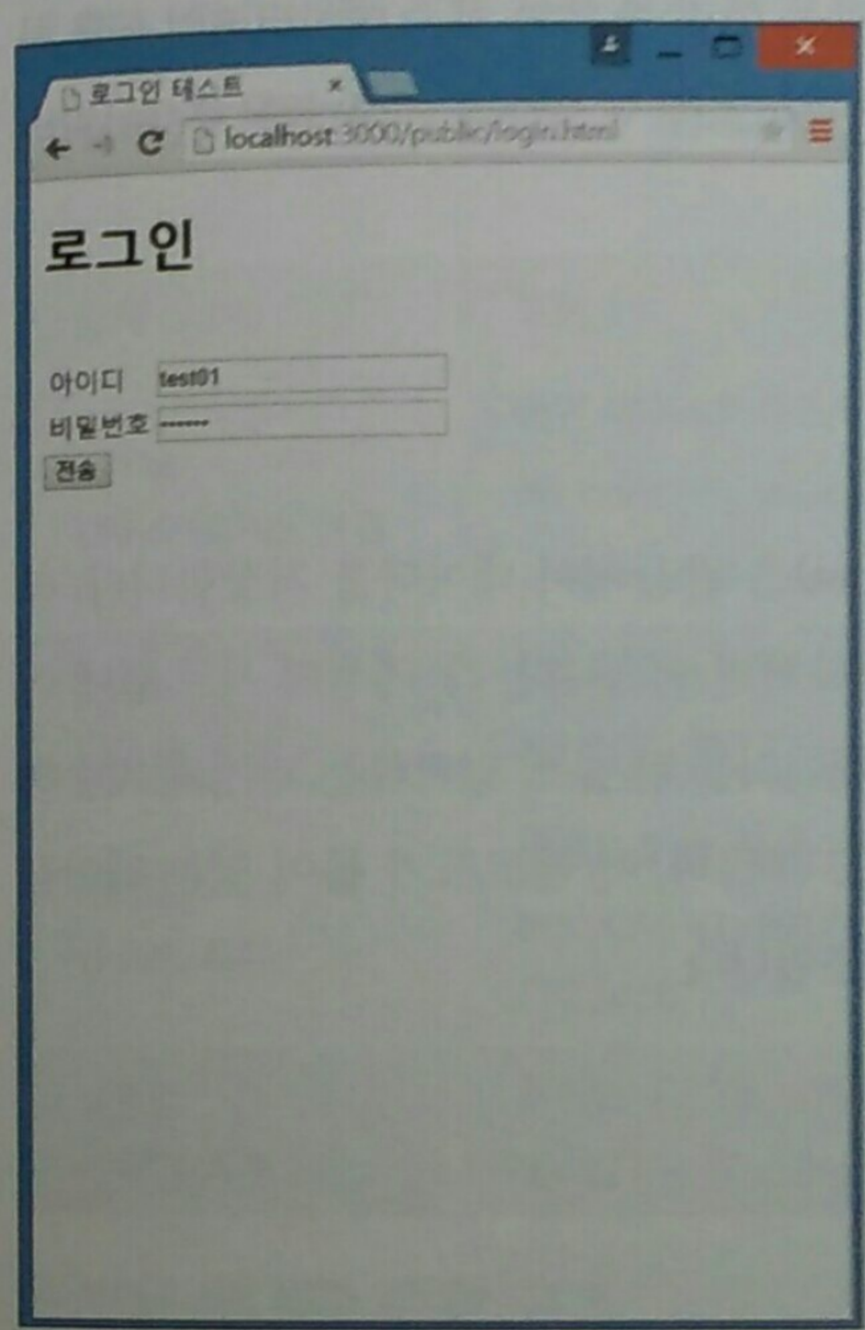
이미 여러 번 사용한 페이지이므로 익숙한 화면일 것입니다. 아이디 입력란에 test01, 비밀번호 입력란에 123456, 사용자명에 소녀시대를 입력한 후 [전송] 버튼을 누릅니다. 성공 응답이 오면 Robomongo 화면에서 users3 컬렉션 안에 문서 객체가 추가되었는지 확인합니다.



▲ 새로 추가된 문서 데이터와 hashed_password의 값

hashed_password 속성 값이 암호화되어 있다는 것을 확인할 수 있습니다. salt 속성에도 값이 들어 있는데 이 값은 문서 객체가 저장될 때마다 변경됩니다. 다시 한 번 웹 브라우저를 열고 로그인 사이트에 접속합니다.

▶ http://localhost:3000/public/adduser.html



▲ 로그인 화면에 값을 입력한 경우

아이디와 비밀번호 입력란에 test01과 123456을 입력한 후 [전송] 버튼을 누르면 로그인 성공 메시지를 볼 수 있으며, 틀린 비밀번호를 넣으면 로그인 실패 메시지가 표시됩니다.

virtual 메소드를 이용해 스키마에 가상 속성을 추가하는 것은 조금 복잡해 보일 수 있으나 코드를 한번 만들어 두면 단순히 속성에 값을 설정하거나 조회하는 것만으로도 암호화 기능을 수행할 수 있어서 매우 유용합니다. 물론 이 메소드는 암호화 이외의 다른 기능을 만들 때도 사용합니다.

지금까지 몽고디비에서 사용자 정보를 저장하고 조회하는 방법을 알아보았습니다. 단순히 사용자를 추가한 후 사용자 리스트를 조회하는 기능에서부터 사용자 비밀번호를 암호화하는 기능까지 여러 번 반복해서 실습해 보세요. 데이터베이스를 다루는 방법에 익숙해지는 것이 최선입니다.