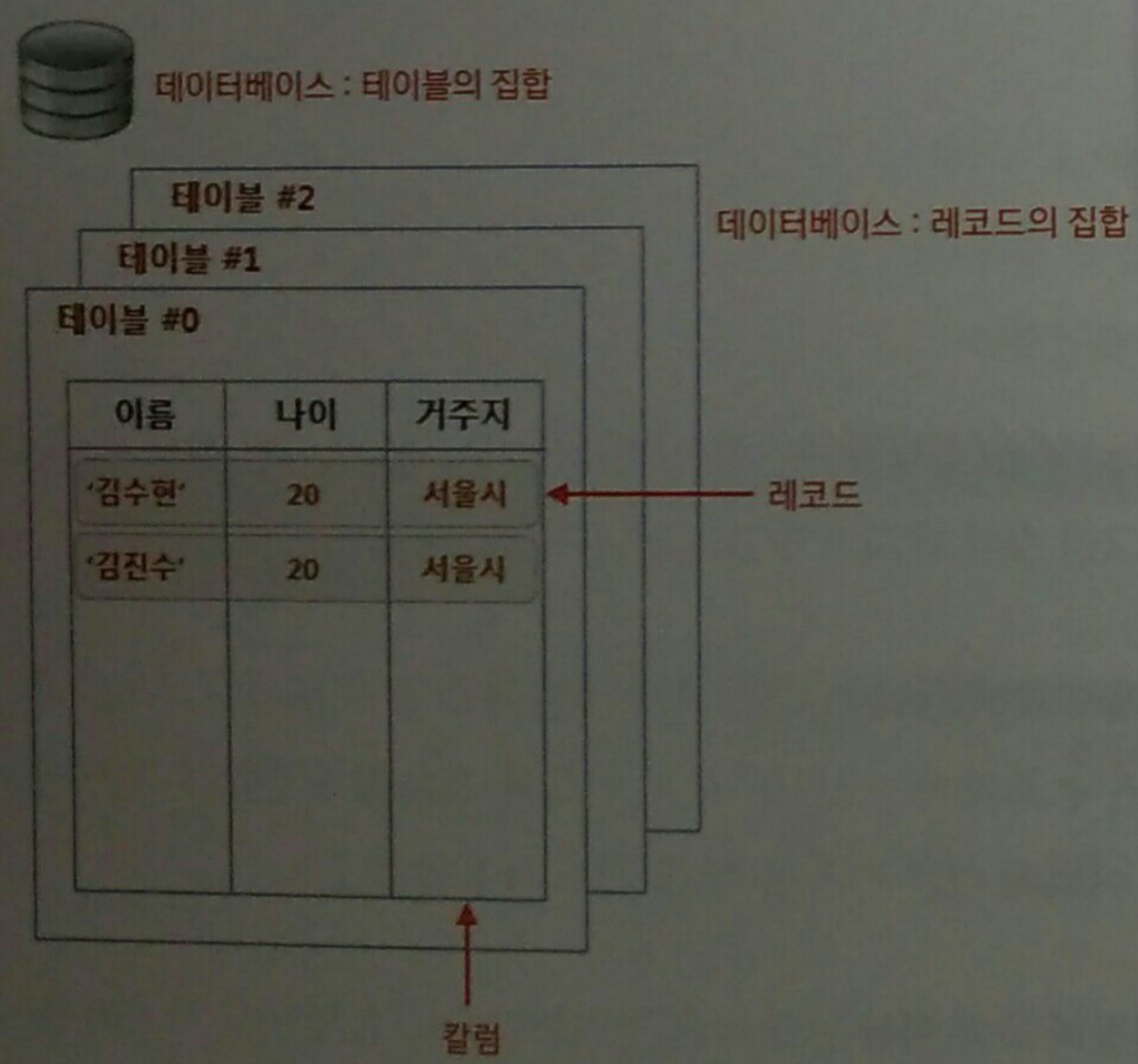
# 06-6 | MySQL 데이터베이스 사용하기

여러분이 사용하던 데이터베이스가 있다면 기존 데이터베이스의 데이터를 조회할 수 있어야 합니다. 기존 데이터베이스로는 오라클이나 MySQL 같은 관계형 데이터베이스가 많은데 노드로 프로그램을 만들 때 이런 관계형 데이터베이스들에 연결할 수 있어야겠죠? 노드에는 이런 종류의 기존 데이터베이스를 사용할 수 있는 외부 모듈이 있습니다. 여기에서는 MySQL 데이터베이스에 연결하여 데이터를 저장하거나 조회하는 방법을 알아보겠습니다.

## 관계형 데이터베이스 간단하게 이해하기

관계형 데이터베이스는 데이터베이스 저장소 안에 **테이블(Table)**을 만들어서 데이터를 저장합니다. 테이블을 이해할 때는 엑셀(Excel) 프로그램의 시트(Sheet)를 생각하면 쉽습니다. 즉, 엑셀의 시트 탭에 어떤 이름의 데이터가 어떤 **타입(Type)**으로 들어갈지 정의한 후 데이터를 한 줄씩 입력하는 과정과 비슷하기 때문입니다. 한 줄에 해당하는 데이터를 **레코드(Record)**라고 부르며, 이 레코드가 들어 있는 테이블은 검색할 수 있습니다. 다음은 **관계형 데이터베이스**의 기본 구조입니다.




▲ 관계형 데이터베이스의 기본 구조

관계형 데이터베이스를 다루는 일은 크게 네 단계로 나뉩니다. 첫 번째 단계에서는 데이터베이스에 연결하고 두 번째 단계에서는 테이블을 만듭니다. 테이블을 만들 때는 각 칼럼의 이름과 함께 어떤 타입의 값이 들어갈 수 있는지 정의합니다. 세 번째 단계에서는 레코드를 추가하고 네 번째 단계에서는 데이터를 조회합니다. 첫 번째 단계를 제외한 나머지 세 단계는 모두 SQL문을 사용합니다. 이 책을 읽는 여러분은 관계형 데이터베이스를 한 번쯤은 다뤄 본 경험이 있을 것이므로 자세한 설명은 생략합니다. 만약 관계형 데이터베이스를 전혀 접해 보지 않았더라도 대표적인 SQL문의 형태만 다시 한 번 정리하면 어렵지 않게 사용할 수 있습니다. 다음은 MySQL 사용 방법을 단계별로 나눈 후 간단한 예를 든 것입니다.

| 단계 | 설명 |
|---|---|
| 1단계:<br>데이터베이스 연결 | 어떤 데이터베이스에 연결할지 지정하는 정보와 보안 정보를 입력합니다.<br>예) connectionLimit: 10, host: 'localhost', user: 'root', password: 'OOOOO', database: 'users', debug: false |
| 2단계:<br>테이블 생성 | 테이블을 만들어 어떤 칼럼에 어떤 타입의 데이터가 들어가는지 정의합니다.<br>예) CREATE TABLE users(id text, name text, age int, password text) |
| 3단계:<br>레코드 추가 | 레코드를 한 줄 추가합니다.<br>예) INSERT INTO users(id, name, age, password) VALUES ('101', '김진수', 20, '123456') |
| 4단계:<br>데이터 조회 | 원하는 조건을 넣어 데이터를 조회합니다.<br>예) SELECT id, name, age, password FROM users WHERE age > 10 |

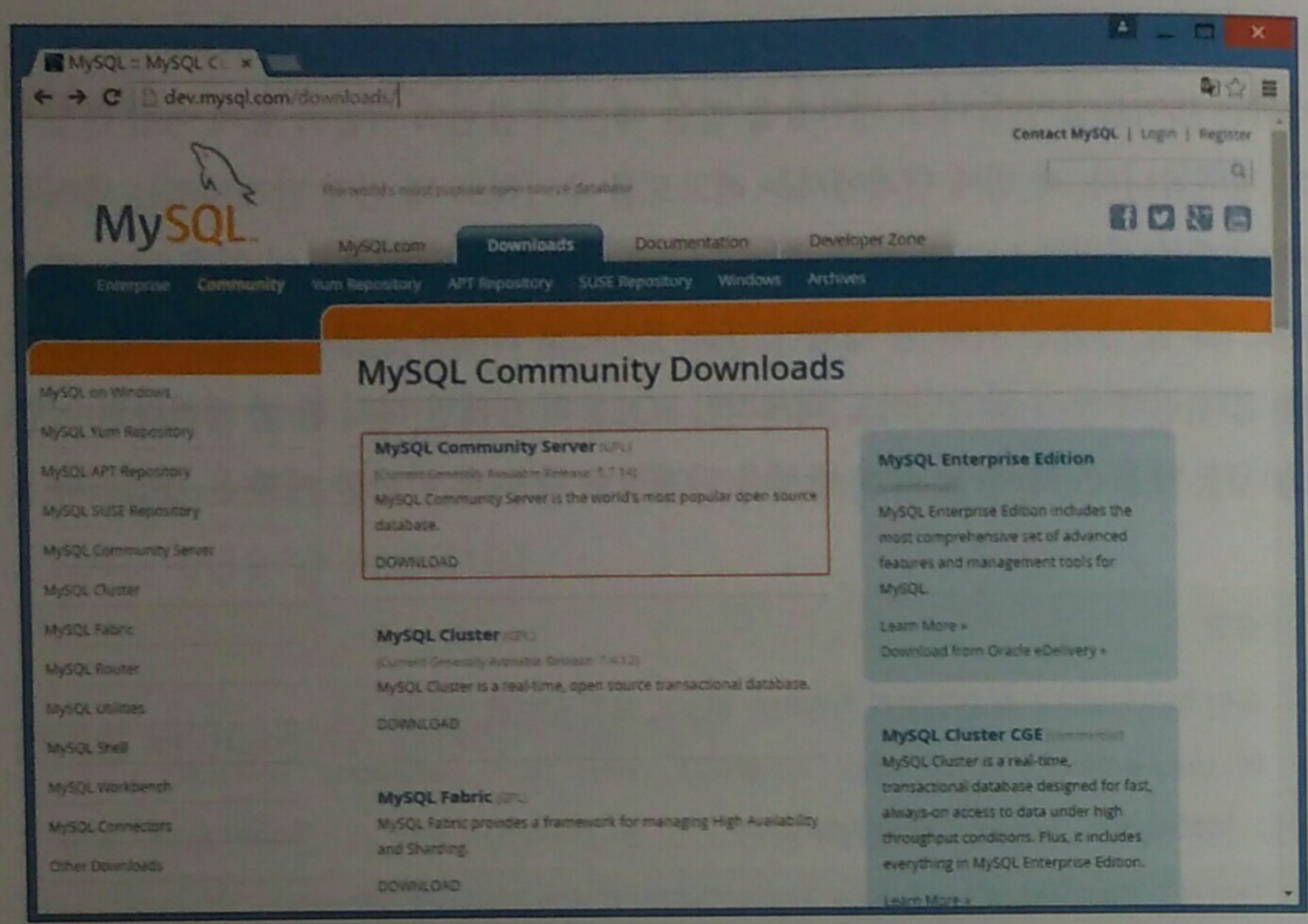
[표] MySQL 사용 방법을 단계별로 나눈 예

## MySQL 설치하기

MySQL을 먼저 설치하고 화면이 있는 관리 도구도 함께 설치합니다. 그런 다음 설치한 관리 도구를 사용해서 데이터베이스와 테이블을 만듭니다. 그리고 노드 서버로 사용자를 추가하거나 데이터베이스에 있는 사용자 정보로 로그인하는 기능을 만들어 보겠습니다.

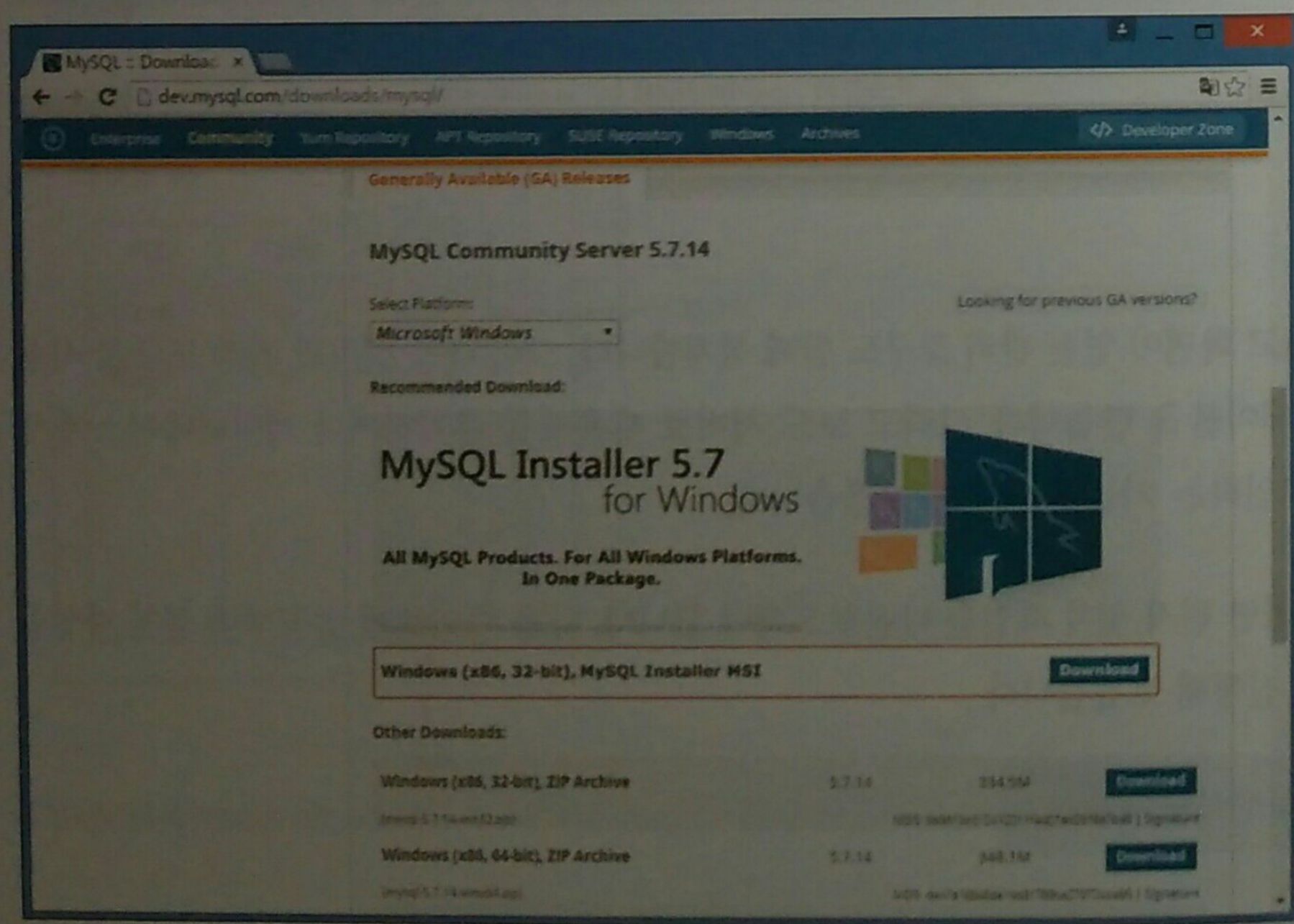
❶ MySQL을 사용하려면 먼저 설치 파일을 다운로드해야 합니다. 다음 사이트에 접속하여 설치 파일을 다운로드한 후 설치를 진행해 보겠습니다.

▶ http://dev.mysql.com/downloads/



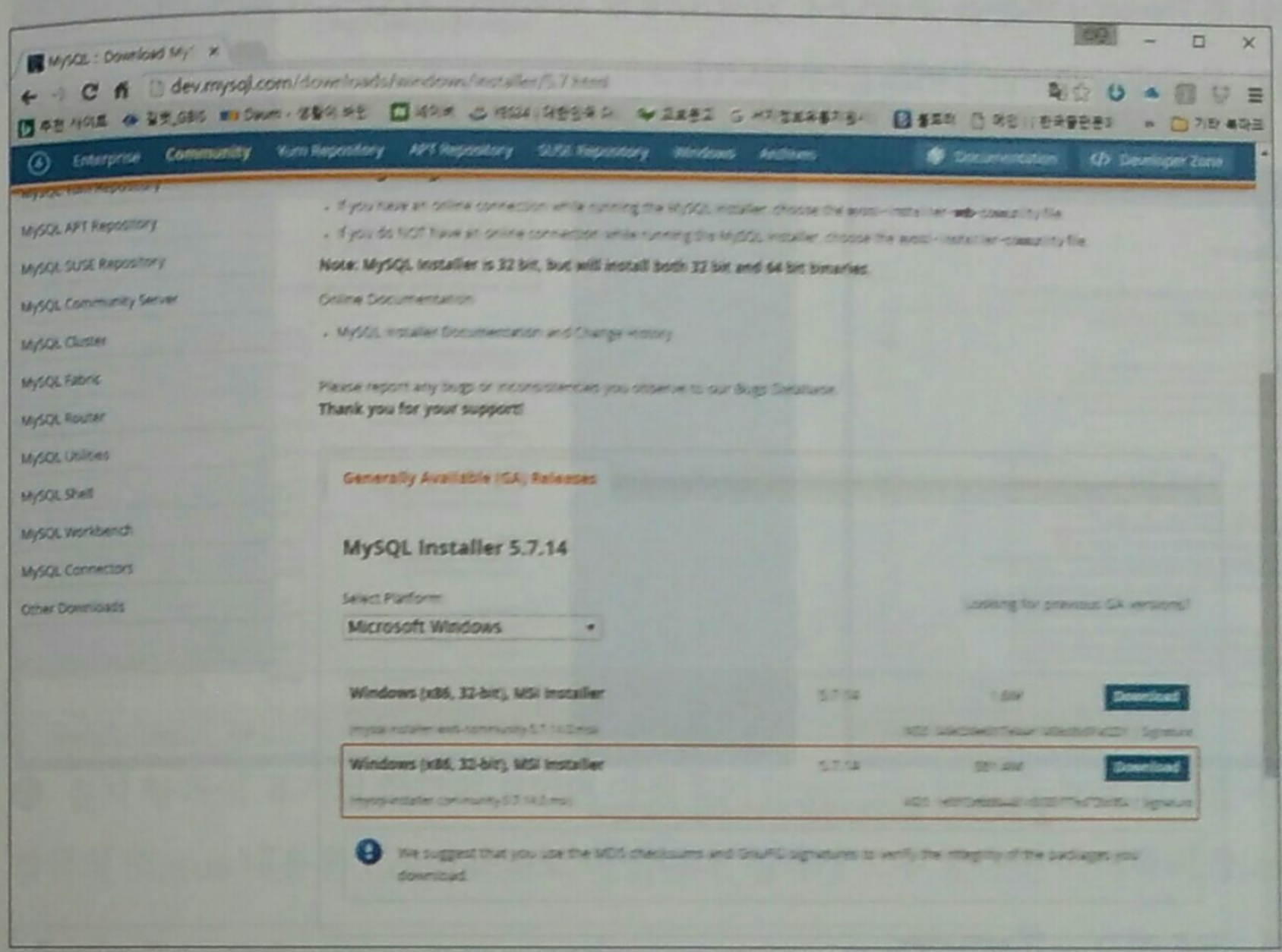

▲ MySQL 다운로드 사이트

❷ [Downloads] 메뉴 중에서 Community 버전은 무료로 사용할 수 있습니다. 다운로드 목록 위쪽에 있는 MySQL Community Server 링크를 클릭하면 다운로드 페이지로 이동합니다. 다운로드 페이지가 나타나면 아래쪽으로 스크롤하여 MySQL Installer MSI를 다운로드합니다.

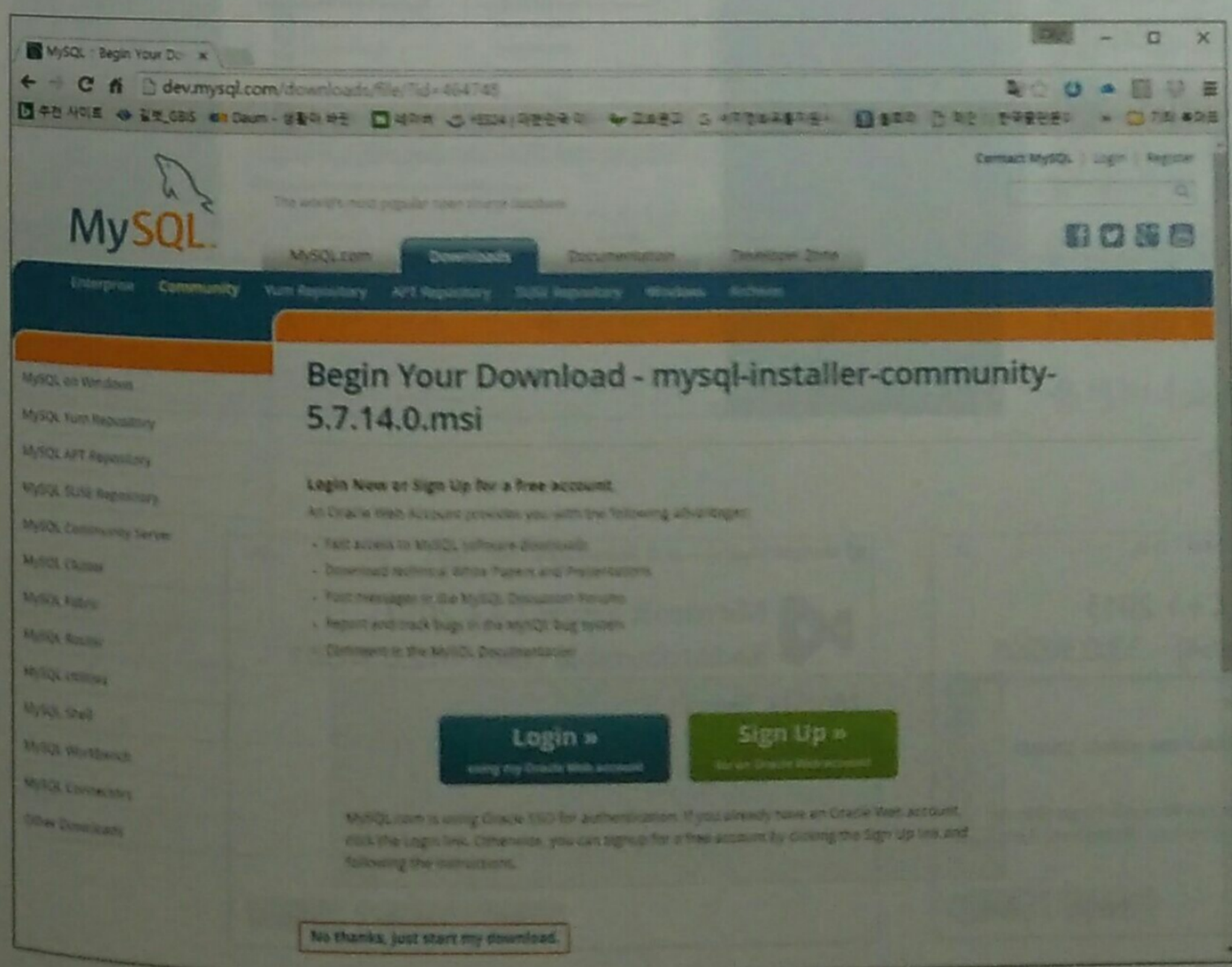



▲ MySQL 설치 파일 다운로드

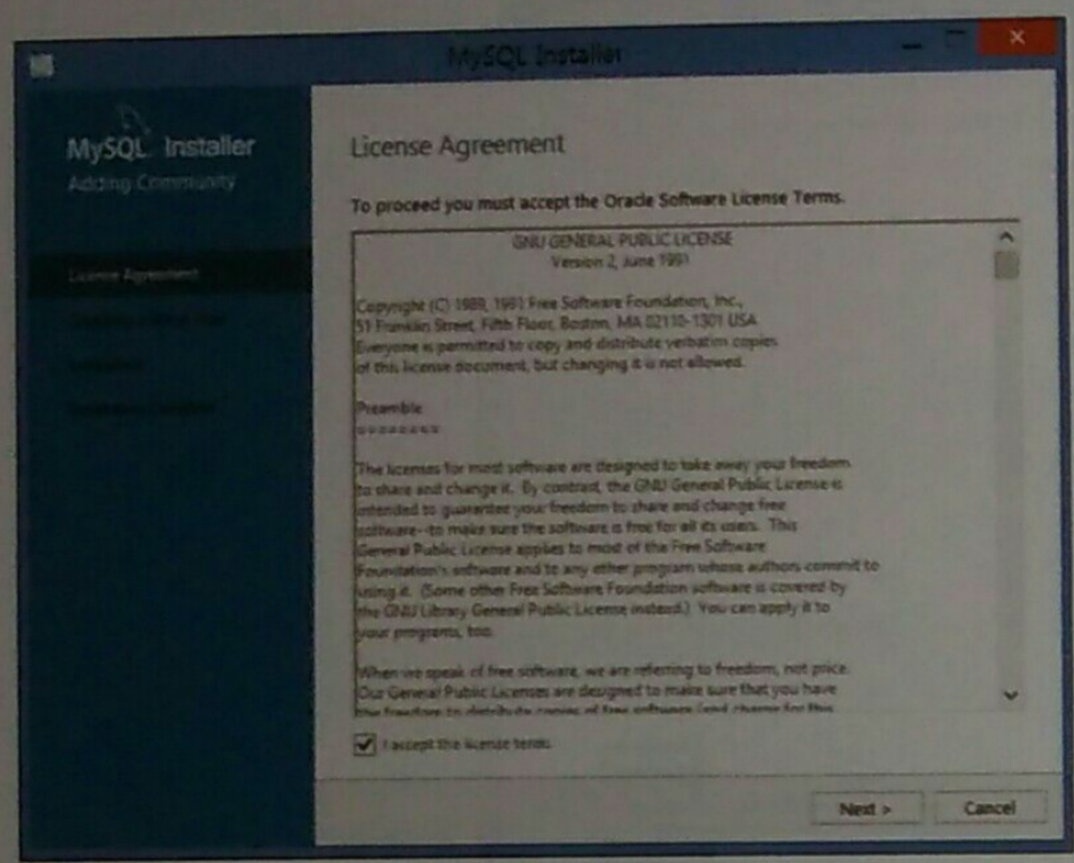
❸ Download MySQL Installer 페이지가 나타나면 역시 아래쪽으로 스크롤해서 용도에 맞는 MSI Installer를 설치합니다. 여기에서는 web에서만 사용하도록 제안된 버전이 아니라 전체 설치가 가능한 Windows(x86, 32-bit), MSI Installer를 설치하면 됩니다.

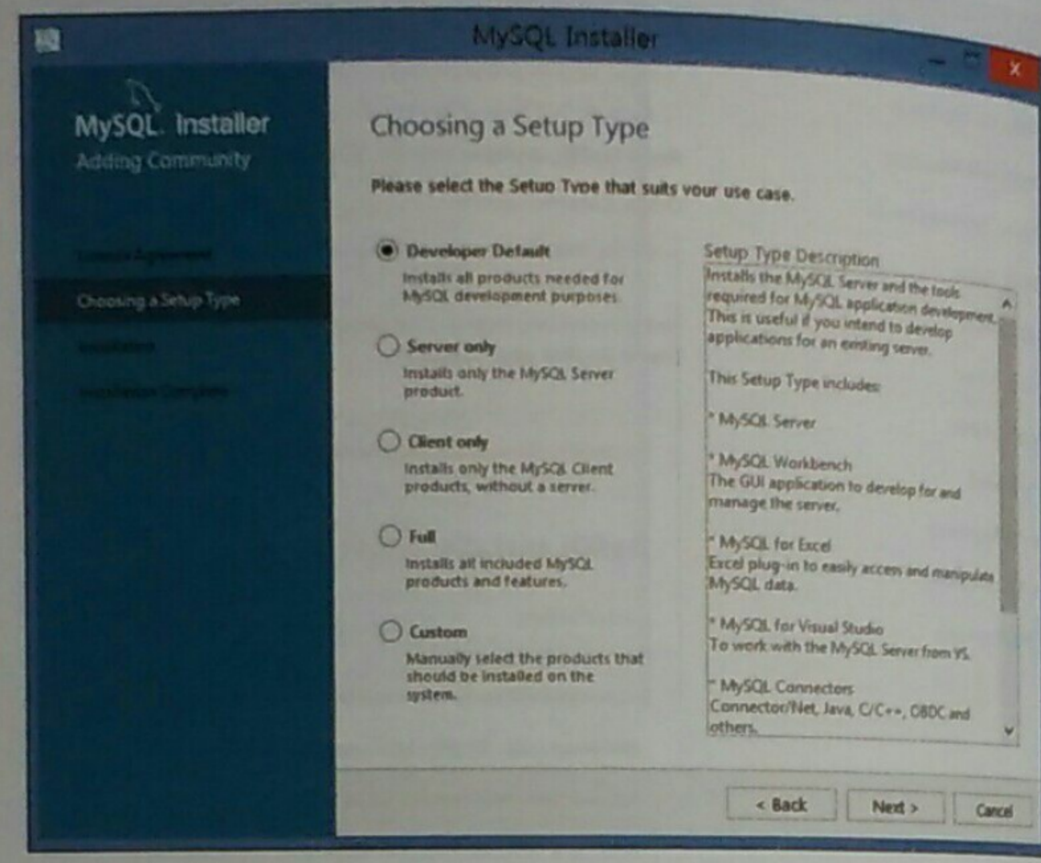



❹ 오라클 계정으로 로그인을 하거나 새로 회원가입을 진행하라는 페이지가 나타납니다. 이 과정은 나중에 진행하도록 하고 우선 하단에 있는 No thanks, just start my download. 링크를 클릭해서 설치 파일을 다운로드합니다.



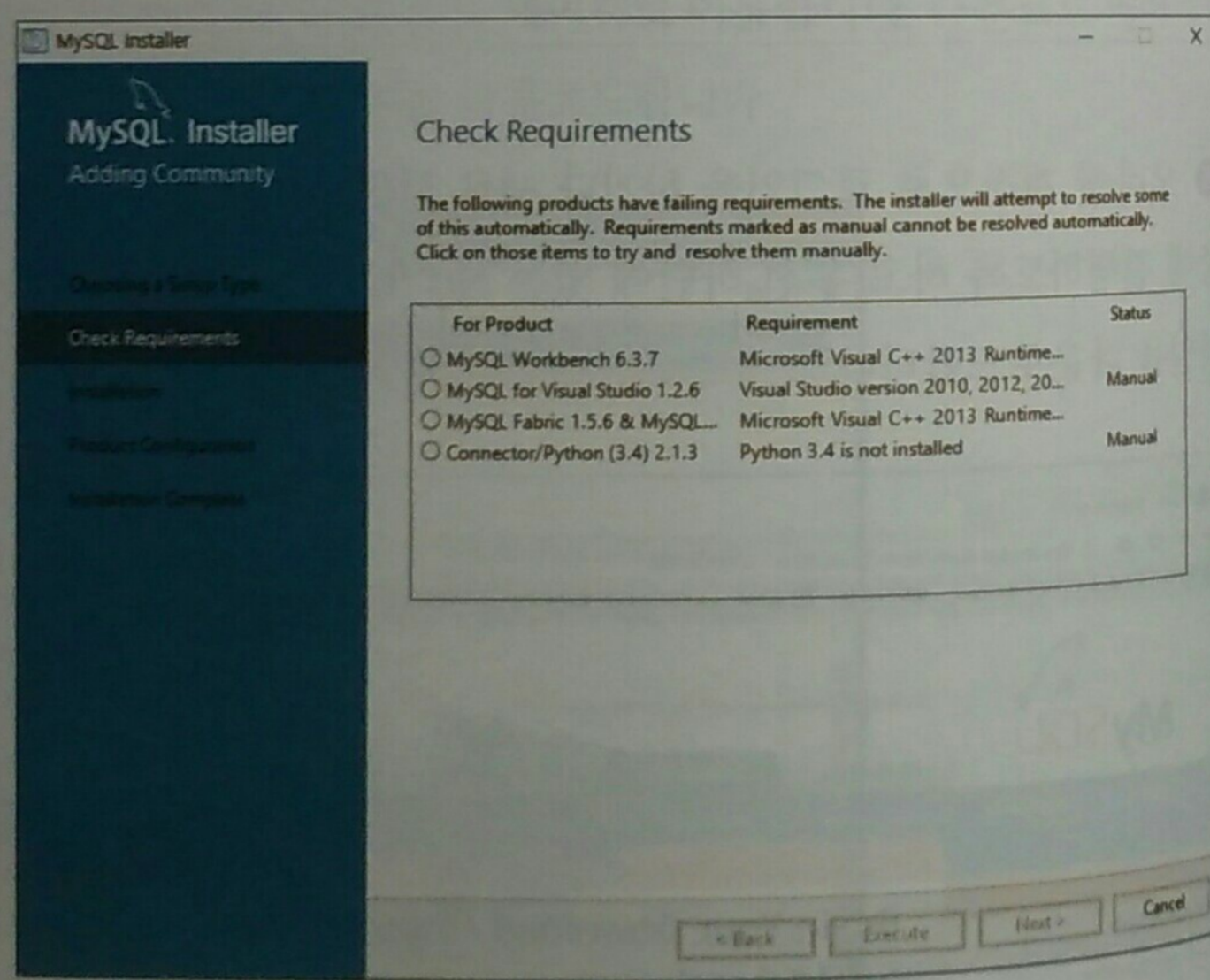
❺ 다운로드한 파일을 설치하는 과정은 일반 프로그램의 설치 과정보다 조금 복잡합니다. 데이터베이스를 설치할 때 여러 가지 설정도 함께 진행하기 때문인데, 조금 복잡하지만 순서대로 설치하면 그리 어렵지 않게 끝낼 수 있습니다. 설치 파일을 실행하고 아래쪽에 있는 I accept th license terms 항목을 클릭해서 라이선스에 동의한 후 [Next›] 버튼을 누릅니다. 설치 유형을 선택하는 화면이 나오면 'Developer Default' 항목을 선택한 상태에서 [Next›] 버튼을 누릅니다.

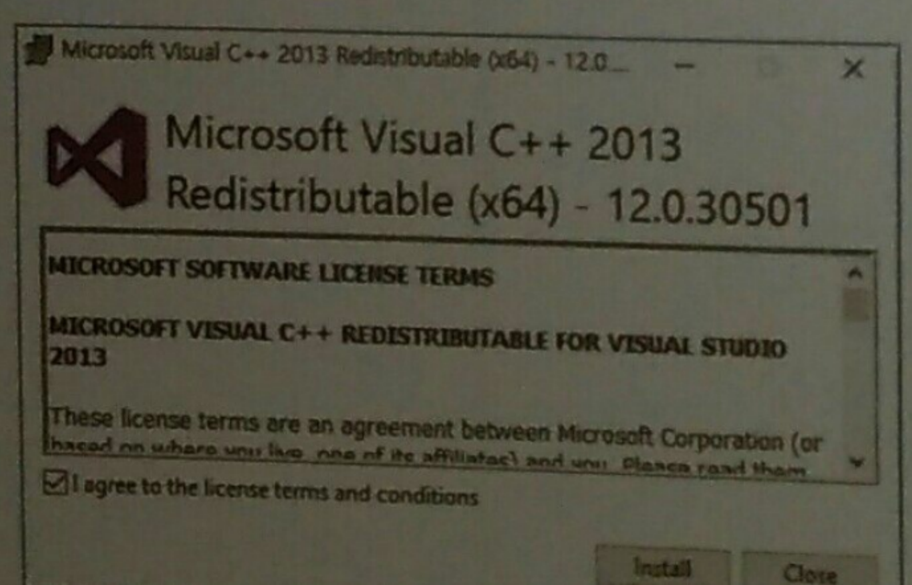


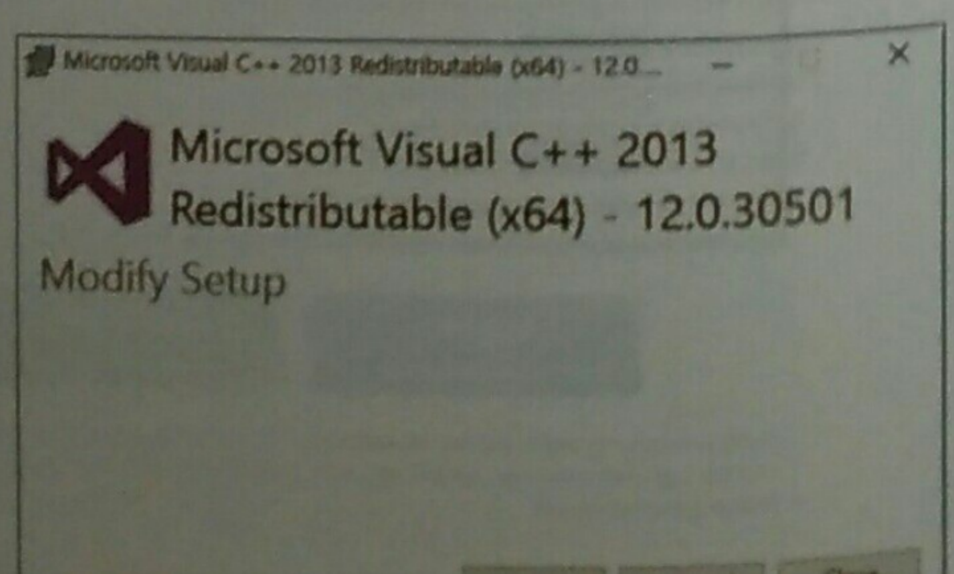




▲ 라이선스 동의 화면과 설치 유형 선택 화면

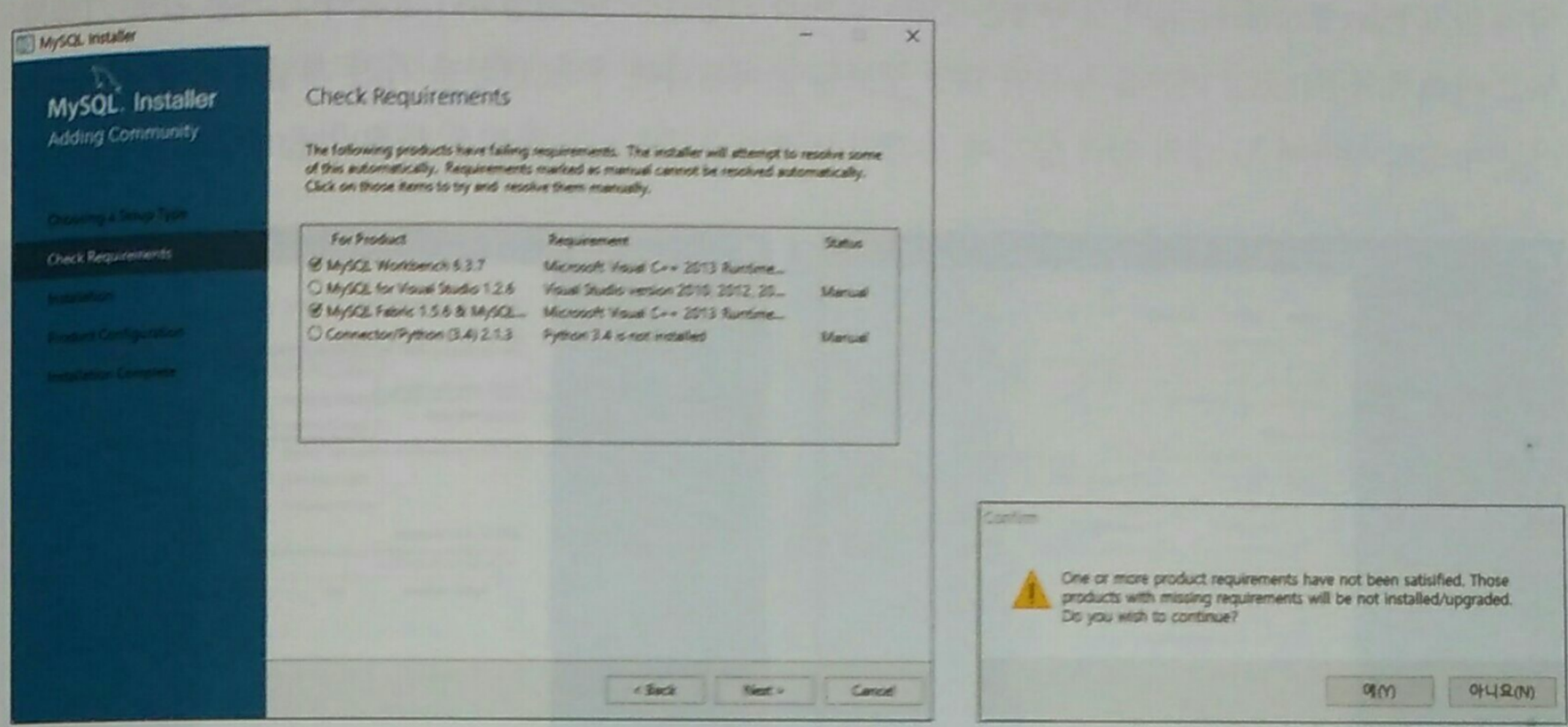
❻ MySQL을 설치하기 전에 필요한 다른 제품들을 설치하기를 권하는 Check Requirements 화면이 나타납니다. 아래쪽의 [Execute] 버튼을 누릅니다. 그러면 설치가 필요한 제품이 자동으로 설치되는 창이 나타납니다. 라이선스 동의를 체크한 후 [Install] 버튼을 눌러서 설치합니다. 설치가 끝났는데 Modify 화면이 나타나면 [Repair] 버튼을 누른 다음 [Close] 버튼을 누르면 설치가 완료됩니다.



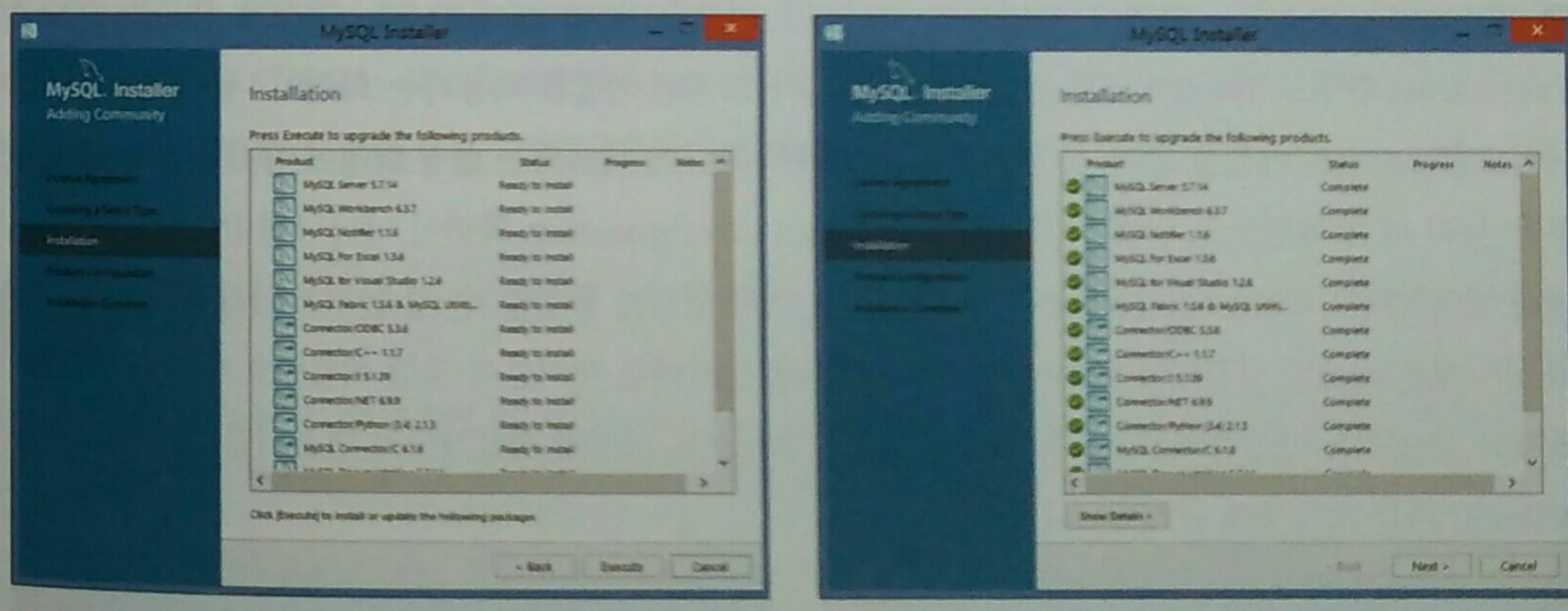



❼ 설치 항목이 체크되면 [Next›] 버튼을 누릅니다. 이때 하나 이상의 제품을 요청하면 만족하지 못한 결과를 가져올 수 있다는 경고 창이 나타납니다. 우선 [예(Y)] 버튼을 눌러서 넘어갑니다.

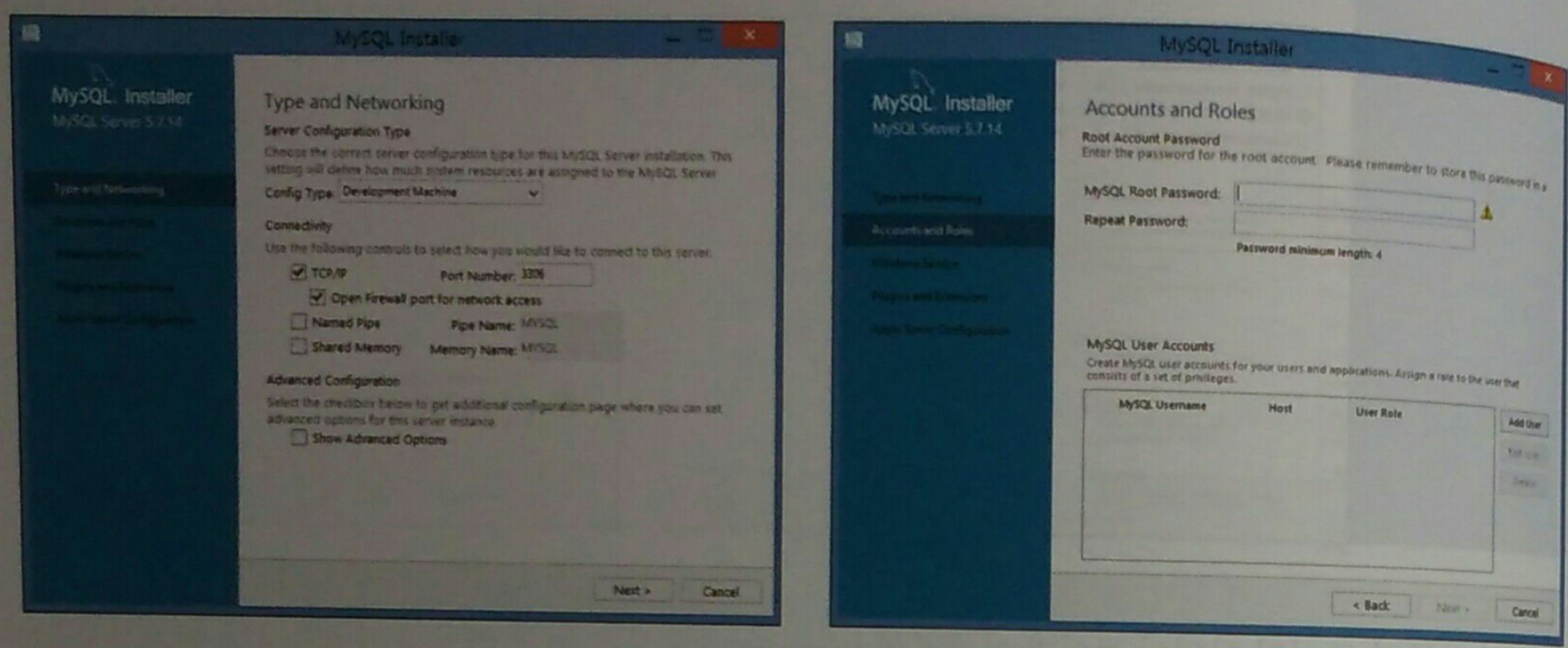



❽ 설치 항목이 표시된 화면이 보이면 아래쪽의 [Execute] 버튼을 눌러 설치를 진행합니다. 설치가 끝난 항목의 Status 내용은 Complete로 바뀝니다. 설치가 모두 끝나면 아래쪽의 [Next›] 버튼을 누릅니다.



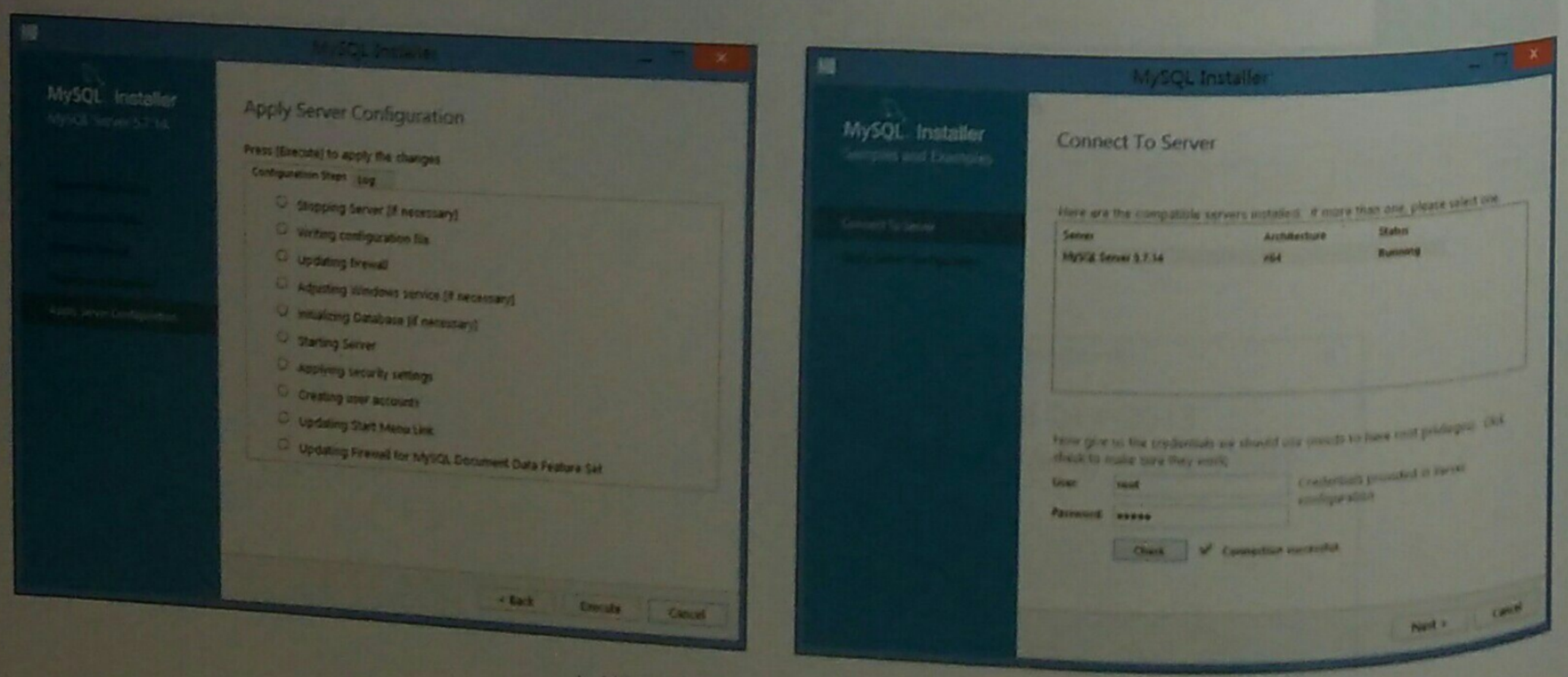

▲ 설치 항목 표시 화면과 설치가 끝났을 때의 화면

❾ Product Configuration 화면이 나타나면 아래쪽 [Next〉] 버튼을 누릅니다. 네트워크 설정을 어떻게 할 것인지 물어보는 화면이 나옵니다. Config Type 콤보 박스에는 Development Machine이 선택되어 있고 Port Number 입력 상자에는 3306 포트가 기본 값으로 설정되어 있습니다. 기본 값을 그대로 두고 아래쪽의 [Next〉] 버튼을 누르면 계정 정보를 물어봅니다. 루트 권한을 가진 계정의 비밀번호는 여러분이 직접 정하여 두 개의 입력 상자에 똑같이 입력한 후 [Next〉] 버튼을 누릅니다.



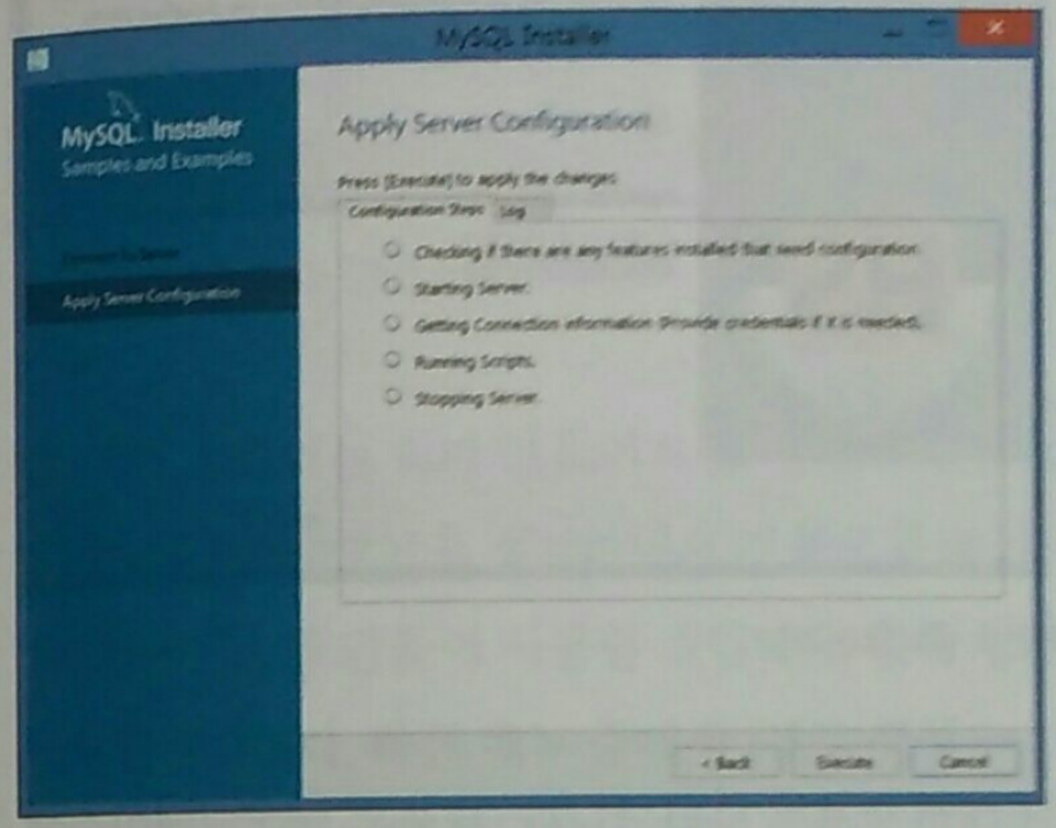

▲ 네트워킹 설정 화면과 계정 설정 화면

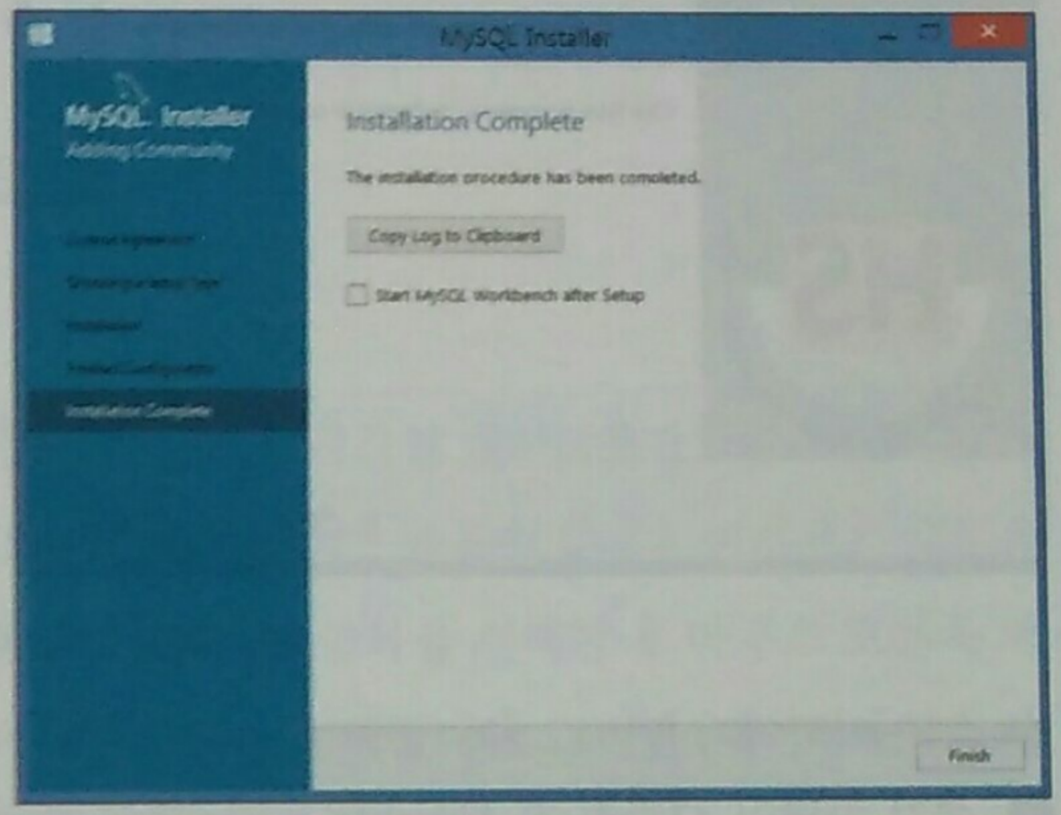
❿ Windows Service 화면이 나타나면 [Next〉] 버튼을 눌러 넘어갑니다. 그리고 Plugins and Extensions 화면도 [Next〉] 버튼을 눌러 넘어갑니다. 그러면 설정을 적용하는 화면이 나타나는데 여기에서도 [Execute] 버튼을 눌러 진행합니다. 각 항목이 하나씩 체크되는 설정 과정이 끝나면 [Finish] 버튼을 눌러 다음 화면으로 넘어갑니다. 다시 Product Configuration 화면이 나타나면 [Next〉] 버튼을 눌러 넘어갑니다. Connect To Server 화면에서는 서버 연결을 확인할 수 있습니다. 아래쪽의 User 입력 상자에는 root가 입력되어 있으며, Password 입력 상자에는 여러분이 정한 비밀번호가 자동으로 등록되어 있습니다. 아래쪽의 [Check] 버튼을 누르면 MySQL 서버에 정상적으로 연결되었다는 메시지가 [Check] 버튼 오른쪽에 표시됩니다.




▲ 설정 적용 화면과 서버 연결 확인 화면

⓫ [Next>] 버튼을 누르면 다시 설정을 적용하는 화면이 나타납니다. 아래쪽의 [Execute] 버튼을 누르면 각 항목이 체크되면서 설정 과정이 진행됩니다. 모든 설정이 끝나면 아래쪽의 [Finish] 버튼을 눌러 다음 화면으로 넘어갑니다. 다시 Product Configuration 화면이 보이면 [Next>] 버튼을 눌러 넘어갑니다. 그러면 설치가 완료되었다는 화면이 나타납니다. 'Start MySQL Workbench after Setup' 체크 박스를 해제한 후 [Finish] 버튼을 누릅니다.




▲ 설정 적용 화면과 서버 연결 확인 화면

MySQL 설치가 끝났으니 MySQL 데이터베이스를 사용할 수 있습니다. 그런데 데이터베이스를 사용할 때 일일이 명령을 입력하는 작업은 번거롭겠죠? 명령을 입력하기보다는 화면이 있는 관리 도구를 사용하는 것이 더 편리합니다. 그래서 관리 도구 중 하나인 HeidiSQL을 설치해 보겠습니다.

**정박사의 한마디** 관리 도구는 여러 가지가 있어요

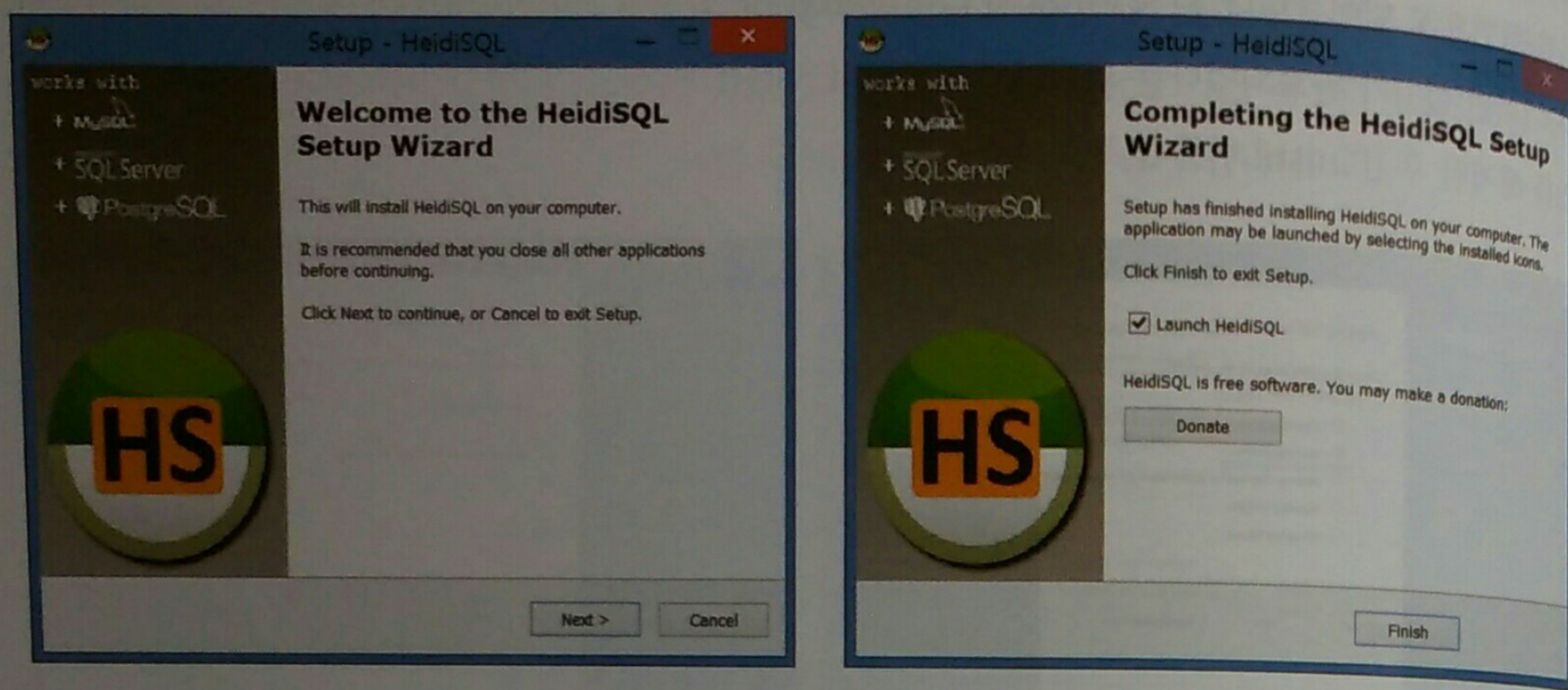
MySQL을 설치할 때 함께 설치된 Workbench도 화면이 있는 관리 도구입니다. 이 외에도 SQLYog 같은 다른 관리 도구도 있습니다. 하지만 이 책에서는 오픈 소스로 만들어진 HeidiSQL을 사용합니다.

## 화면이 있는 관리 도구 HeidiSQL 설치하기

HeidiSQL을 사용하려면 먼저 다음 사이트에 접속하여 설치 파일을 다운로드한 후 설치를 진행합니다. 화면 왼쪽 위에 있는 [HeidiSQL 0.0 Installer] 버튼을 클릭해서 다운로드합니다.

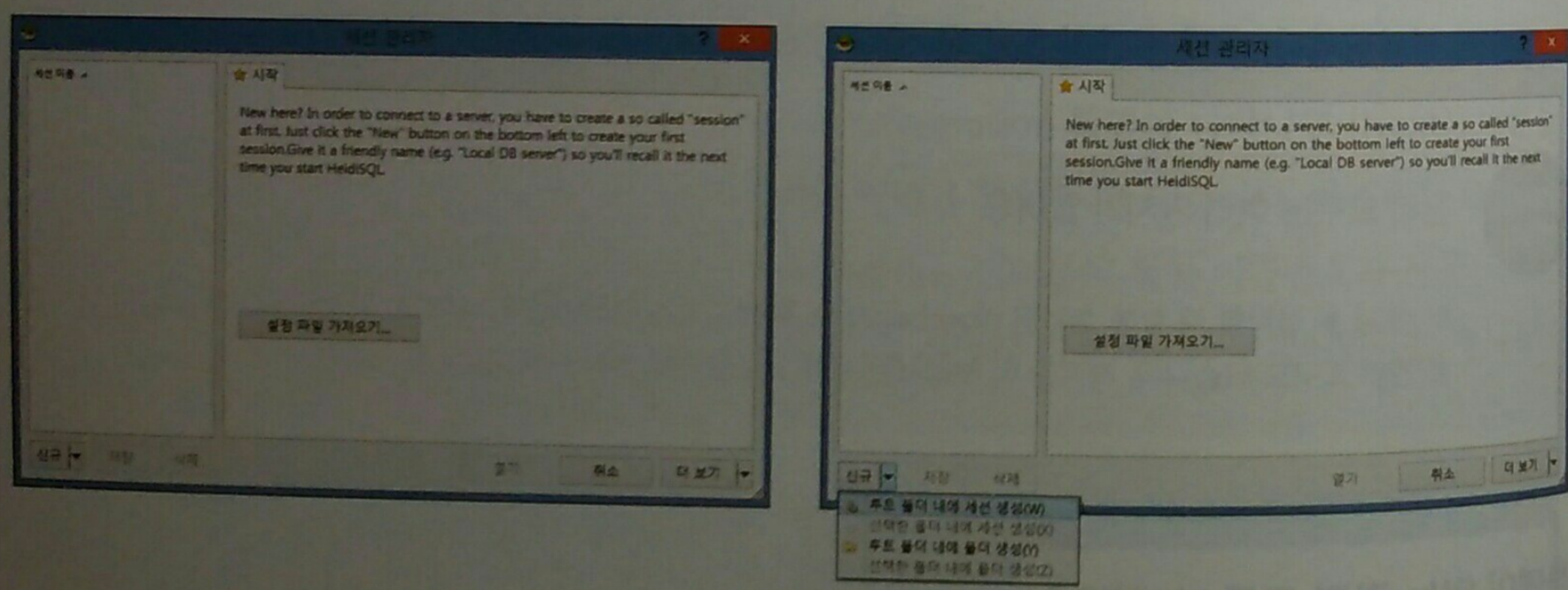
▶ http://www.heidisql.com/download.php

❶ 설치 파일을 실행하면 [Setup - HeidiSQL] 대화상자가 나타납니다. [Next>] 버튼을 눌러 진행한 후 설치가 끝나면 [Finish] 버튼을 누릅니다.



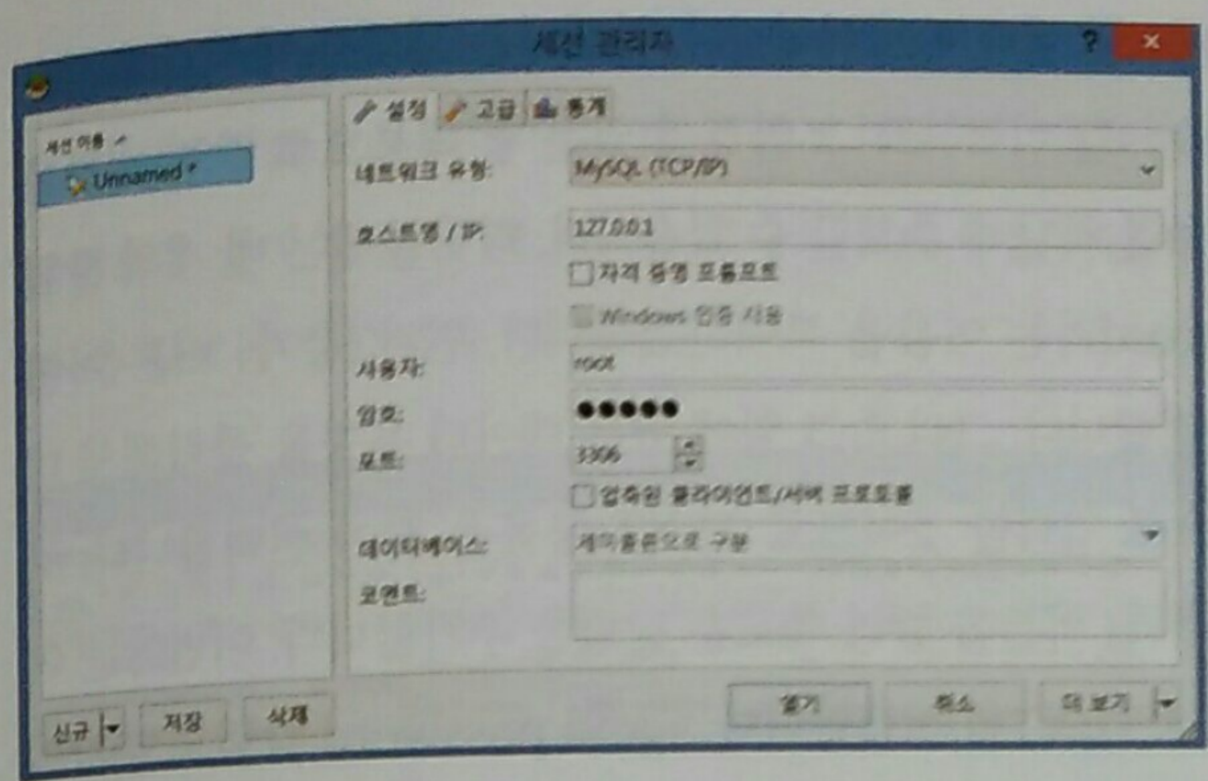

▲ HeidiSQL 설치 시작 화면과 설치 완료 화면

❷ 설치 화면이 사라진 후 HeidiSQL의 [세션 관리자] 대화상자가 나타납니다. 세션은 데이터베이스에 연결하는 하나의 단위를 말합니다. 따라서 이 화면에서 연결 정보를 만들면 그 정보로 데이터베이스에 연결할 수 있습니다. 왼쪽 아래에 '신규'라고 표시된 콤보 박스의 ▼ 버튼을 클릭하면 [루트 폴더 내에 세션 생성] 메뉴를 선택할 수 있습니다.



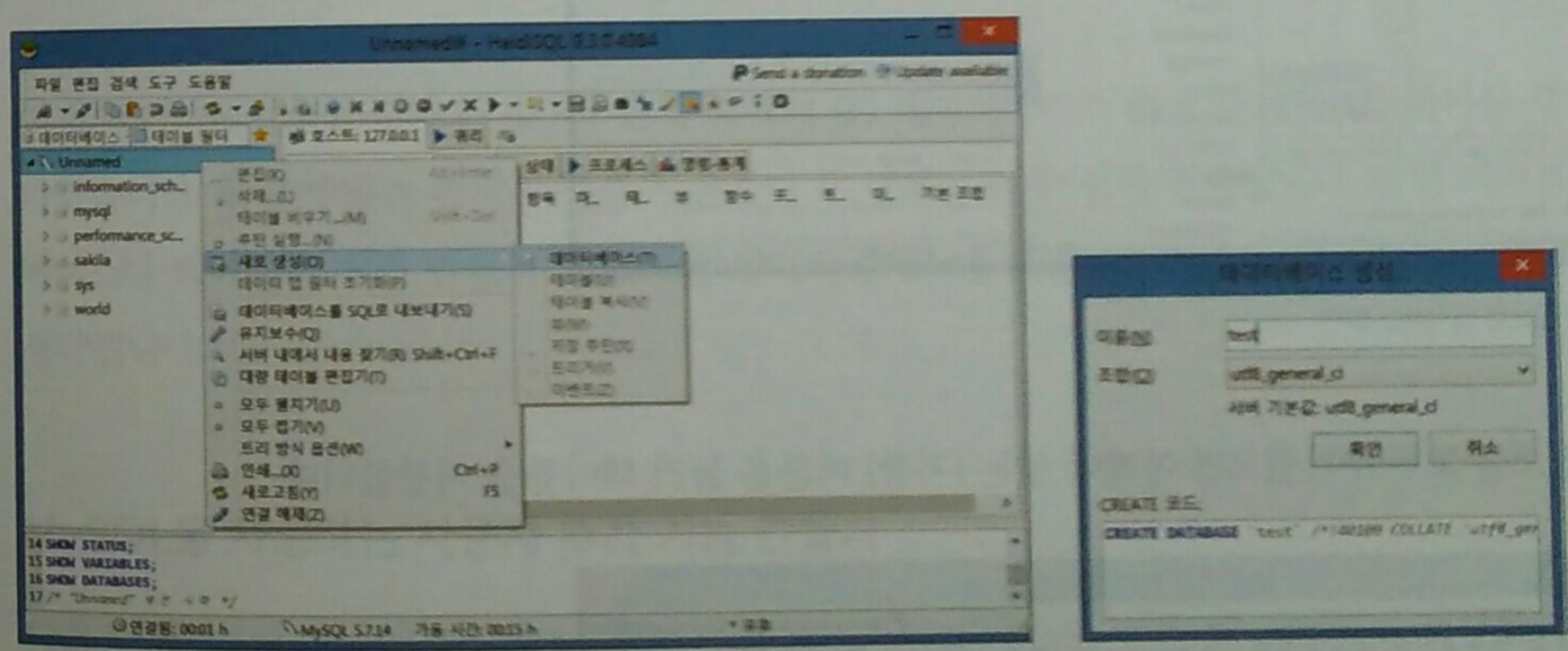

▲ [세션 관리자] 대화상자에서 새로운 세션을 생성하는 메뉴

❸ 새로운 세션 정보를 입력하는 화면이 보입니다. 다른 항목들은 그대로 두고 사용자 입력 상자에 root, 암호 입력 상자에 여러분이 만든 암호를 입력한 후 왼쪽 아래의 [저장] 버튼을 누릅니다. 그러면 세션 정보가 저장됩니다. 그다음 오른쪽 아래에 있는 [열기] 버튼을 누르면 데이터베이스에 연결한 후 새로운 창이 뜹니다.



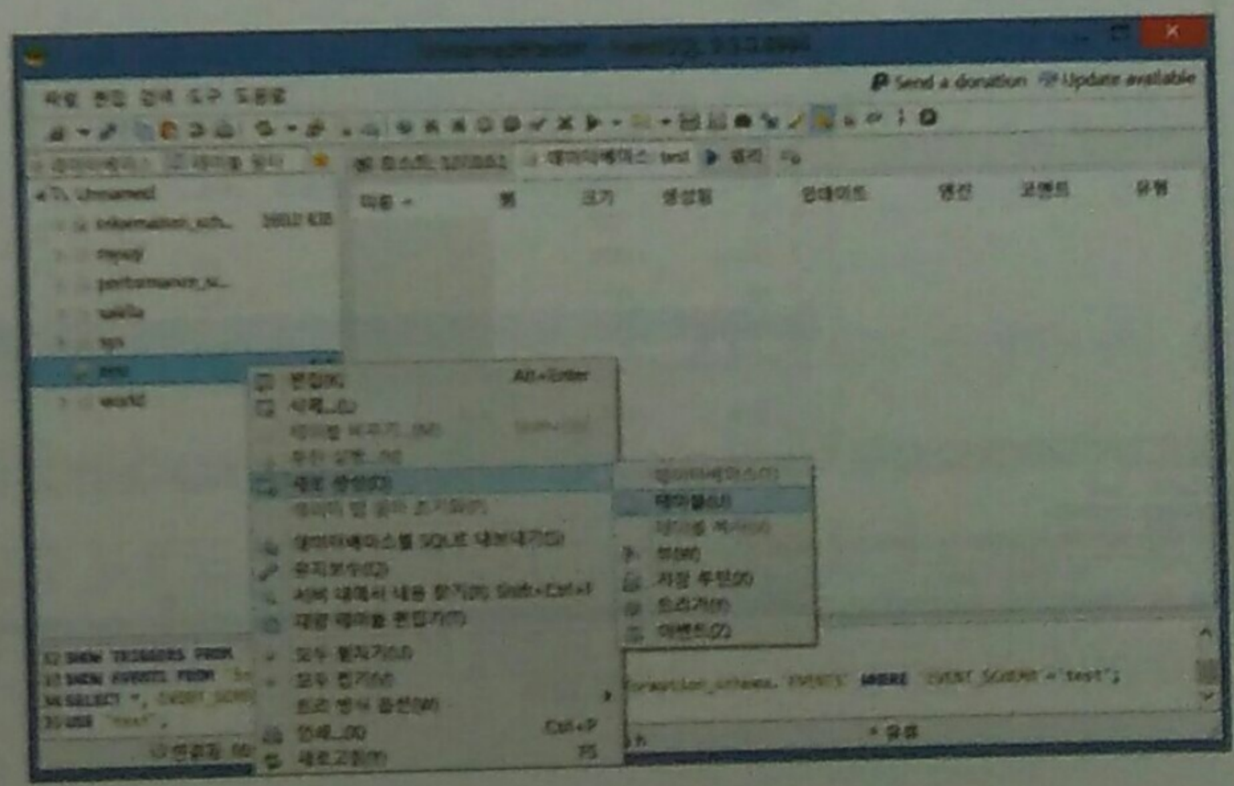

▲ 새로운 세션을 만드는 [세션 관리자] 대화상자

❹ 화면 왼쪽에는 이미 만들어져 있는 데이터베이스들이 보입니다. 각각의 데이터베이스는 저장소를 의미하는데 HeidiSQL을 설치하면서 자동으로 여러 개의 데이터베이스가 만들어집니다. 여러분은 새로운 데이터베이스를 만들어 사용할 것입니다. 왼쪽 영역에서 Unnamed 항목을 선택한 후 마우스 오른쪽 버튼을 클릭합니다. 빠른 메뉴 중에서 [새로 생성 → 데이터베이스]를 선택합니다. 그러면 [데이터베이스 생성] 대화상자가 보입니다. 이름 입력 상자에 test를 입력한 후 [확인] 버튼을 누릅니다.



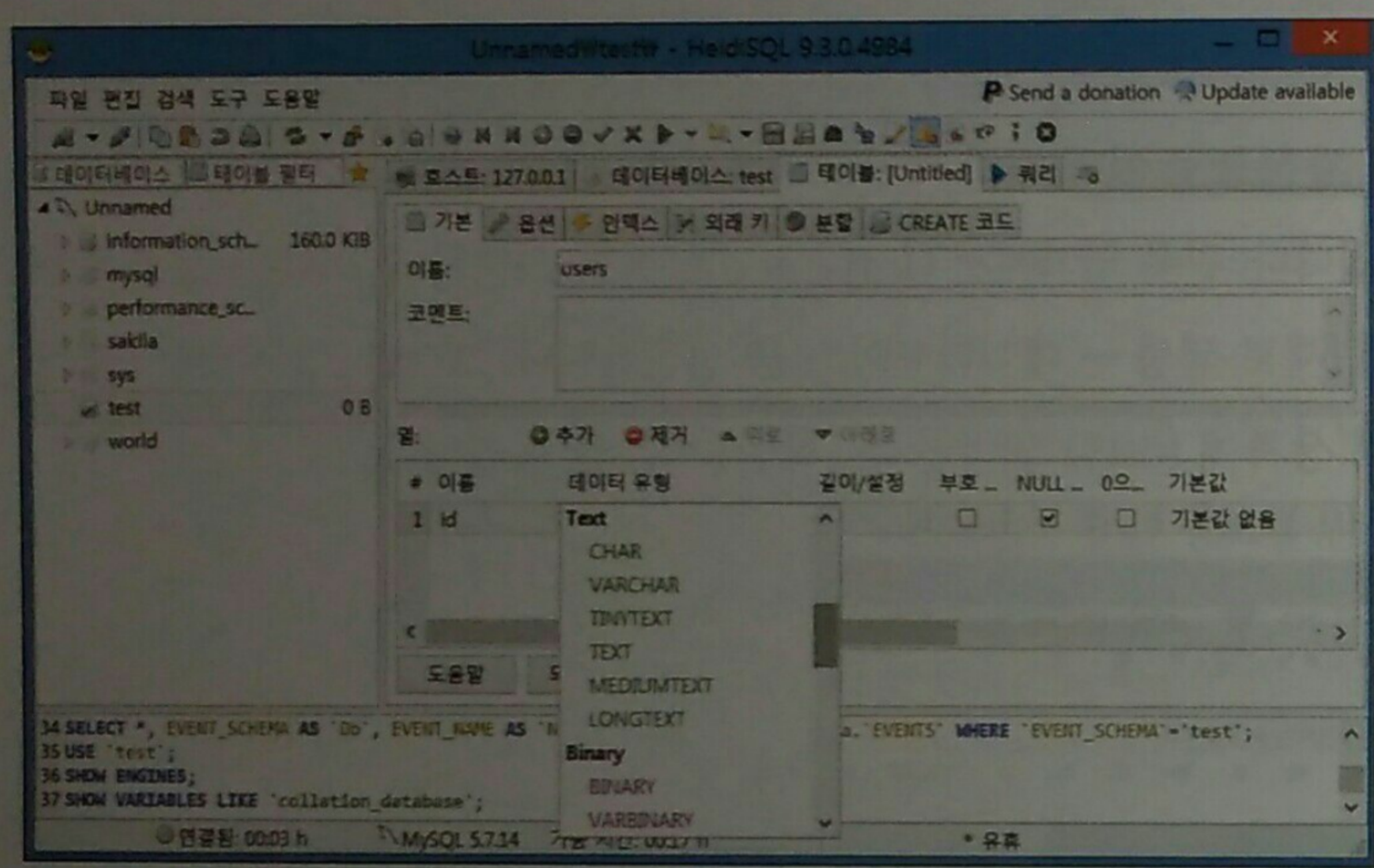

▲ 새로운 test 데이터베이스 생성

❺ 데이터베이스가 새로 만들어지면 왼쪽 목록에 test 항목이 추가됩니다. 간단하게 데이터베이스를 다루는 4단계 중 첫 번째 단계가 완료되었습니다. 이제 두 번째 단계인 테이블 생성을 진행하겠습니다. 왼쪽 목록에서 지금 생성한 test 항목을 선택한 후 마우스 오른쪽 버튼을 누르고 [새로 생성 → 테이블] 메뉴를 선택합니다.



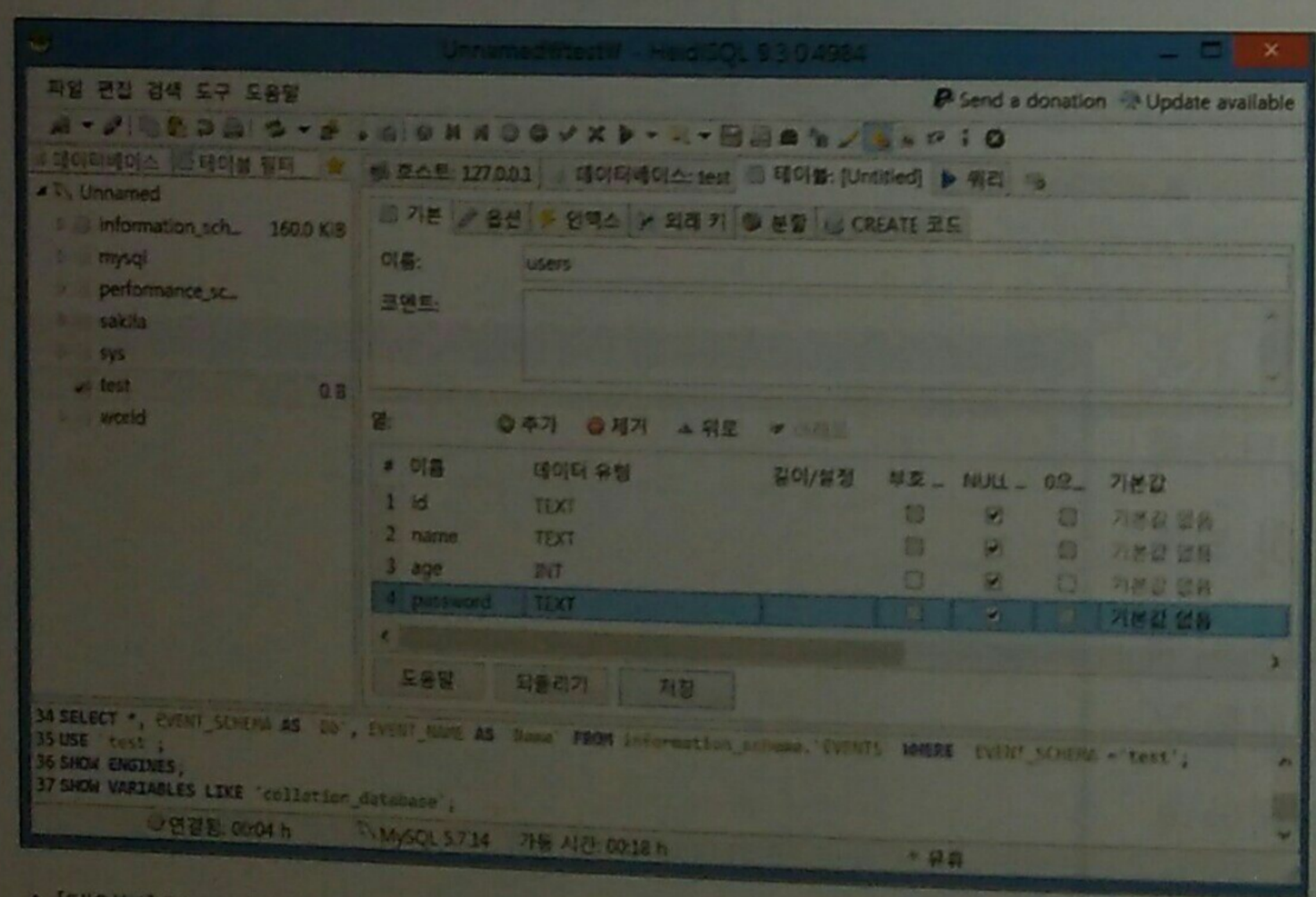
▲ 테이블 생성 메뉴

❻ 오른쪽 화면 위쪽에 새로운 [테이블: [Untitled]] 탭이 만들어집니다. 화면에는 새로운 테이블의 정보를 입력할 수 있습니다. 이름 입력 상자에는 users를 입력하고 그 아래에 있는 열 내용으로 칼럼 정보를 추가합니다. [➕ 추가] 버튼을 누르면 칼럼 정보를 하나씩 추가할 수 있습니다. 테이블에는 네 개의 칼럼을 추가할 것이며 각각 id, name, age, password라는 이름을 부여하겠습니다. 먼저 [추가] 버튼〈아이콘-001.tif〉을 누른 후 이름 입력란에 id를 입력합니다. 그리고 그 옆에 있는 데이터 유형을 클릭해서 데이터 유형을 선택합니다. 이름은 문자열 타입으로 입력할 것이므로 TEXT 항목을 찾아 선택합니다. 하나의 칼럼 정의가 모두 끝났습니다. 다시 [➕ 추가] 버튼을 눌러 새로운 칼럼을 추가합니다. 이번에는 이름 입력란에 name을 입력하고 데이터 유형으로 TEXT를 선택합니다. age와 password 칼럼도 추가한 후 각각 INT와 TEXT 타입으로 데이터 유형을 선택합니다.



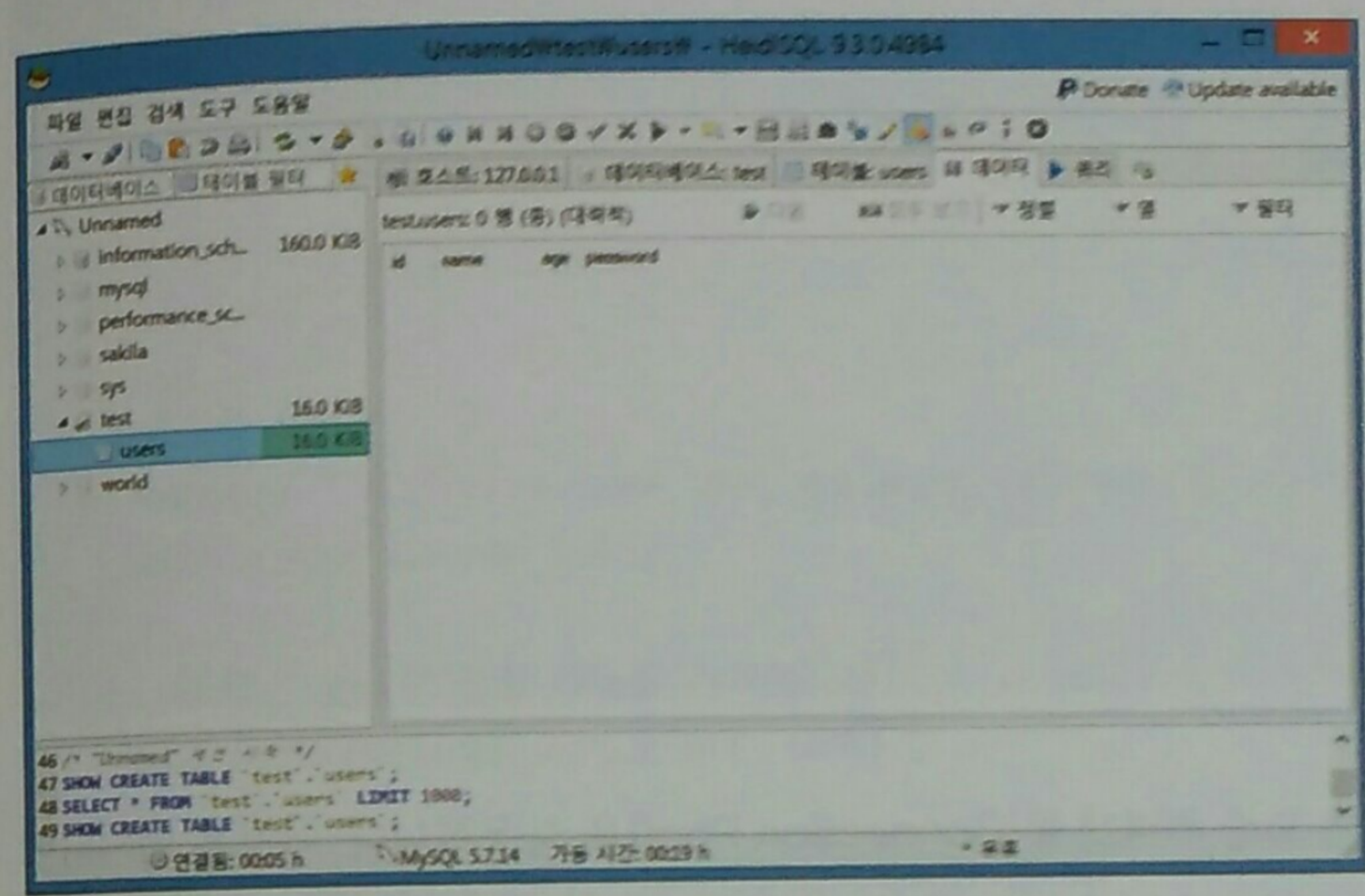

▲ 테이블 생성 정보를 입력하는 화면

❼ 칼럼 정보를 모두 추가했으면 아래에 있는 [저장] 버튼을 눌러 테이블을 저장합니다.




▲ [테이블] 탭 안에 칼럼 정보까지 모두 입력한 화면

❽ 테이블이 생성되면 왼쪽의 test 데이터베이스 항목 아래에 users 테이블이 추가됩니다. 이 테이블 이름을 선택하고 오른쪽에 있는 탭 중에서 [데이터] 탭을 선택하면 데이터를 조회할 수 있습니다. 처음에는 데이터가 없기 때문에 레코드가 하나도 보이지 않지만 나중에 데이터를 넣은 후 조회하면 그 데이터를 볼 수 있고 수정할 수도 있습니다.




▲users 테이블을 만들어서 데이터를 확인한 화면

이제 데이터베이스를 다루는 4가지 단계 중에서 (1) 데이터베이스 생성 단계와 (2) 테이블 생성 단계를 마쳤습니다. 화면이 있는 관리 도구를 사용하니 두 단계는 아주 쉽게 진행됩니다. 만약 여러분이 데이터베이스나 테이블을 직접 만들지 않고 기존 데이터베이스에 연결하여 사용한다면 이 2가지 단계는 모두 생략됩니다.

## MySQL을 사용하는 사용자 추가 기능 만들기

다음 단계인 (3) 데이터 추가와 (4) 데이터 조회는 익스프레스로 만든 웹 서버에서 진행합니다. 노드에서 MySQL 데이터베이스에 연결하려면 mysql 모듈을 사용하면 됩니다. 먼저 명령 프롬프트에서 이 모듈을 설치합니다.

```
% npm install mysql --save
```

몽고디비에 연결할 때 사용한 app.js 파일을 복사해서 app8.js 파일로 만든 후 코드를 일부 수정합니다. 몽고디비를 연결할 때 추가한 코드는 모두 삭제합니다. connectDB 함수 부분은 물론 http.createServer 메소드 안에 들어 있던 connectDB 함수 호출 부분도 모두 삭제합니다. 코드 윗부분에 mysql 모듈을 불러오는 코드를 추가한 후 그 아래에 MySQL 데이터베이스에 연결하는 다음 코드를 추가합니다.

```
……
//===== MySQL 데이터베이스를 사용할 수 있는 mysql 모듈 불러오기 =====//
var mysql = require('mysql');

//===== MySQL 데이터베이스 연결 설정 =====//
var pool = mysql.createPool({
    connectionLimit : 10,
    host : 'localhost',
    user : 'root',
    password : '00000',
    database : 'test',
    debug : false
});
……
```

관계형 데이터베이스에 연결할 때는 보통 **커넥션 풀**(Connection Pool)을 사용합니다. 이것은 데이터베이스 연결 객체가 너무 많이 만들어지는 것을 막고 한번 만든 연결을 다시 사용할 수 있게 합니다. 데이터베이스에 연결하면 메모리 리소스를 많이 차지하므로 한번 만든 연결 객체는 커넥션 풀에 넣어 두고 다음번 요청이 있을 때 다시 사용합니다. 이때 너무 많은 연결이 만들어지지 않도록 커넥션 풀의 최대 크기를 설정합니다. 커넥션 풀은 연결 개수를 제한하므로 연결을 사용한 후에는 반드시 다시 풀에 넣어주어야 하는 제약이 있습니다. 커넥션 풀을 만들려면 mysql 모듈의 **createPool 메소드**를 호출해서 옵션 정보가 있는 객체를 넣어 줍니다. 객체 안에는 connectionLimit, host, port 등의 연결 정보가 속성으로 들어갑니다.

| 속성 | 설명 |
|---|---|
| connectionLimit | 커넥션 풀에서 만들 수 있는 최대 연결 개수를 설정합니다. |
| host | 연결할 호스트 이름을 설정합니다.<br>내 컴퓨터인 경우 localhost 또는 127.0.0.1을 입력할 수 있습니다. |
| port | 데이터베이스가 사용하는 포트 번호를 설정합니다.<br>MySQL의 디폴트 포트는 3306입니다. |
| user | 데이터베이스 사용자 아이디를 설정합니다.<br>MySQL에서 루트 권한을 가진 디폴트 사용자 아이디는 root입니다. |
| password | 데이터베이스 사용자의 비밀번호를 설정합니다. |
| database | 데이터베이스 이름을 설정합니다. |
| debug | 데이터베이스 처리 과정을 로그로 남길 것인지 설정합니다. |

[표] 커넥션 풀을 만들 때 전달하는 객체 속성

createPool 메소드를 호출하면 데이터베이스에 연결해서 커넥션 풀을 만들게 됩니다. 따라서 그다음부터는 pool 객체에서 연결 객체를 가져와 사용할 수 있습니다.

MySQL 데이터베이스에 연결하는 과정이 끝났으니 이제 사용자 추가 기능을 넣어 보겠습니다. addUser 함수는 다음 코드처럼 만듭니다.

**참조 파일** | DatabaseExample>app8.js

```
……

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

    // 커넥션 풀에서 연결 객체를 가져옵니다.
    pool.getConnection(function(err, conn) {
        if(err) {
            conn.release();  // 반드시 해제해야 합니다.
          return;
        }
        console.log('데이터베이스 연결 스레드 아이디 : ' + conn.threadId);

        // 데이터를 객체로 만듭니다.
        var data = {id:id, name:name, age:age, password:password};

        // SQL문을 실행합니다.
        var exec = conn.query('insert into users set ?', data, function(err, result) {
            conn.release();  // 반드시 해제해야 합니다.
            console.log('실행 대상 SQL : ' + exec.sql);

            if(err) {
             console.log('SQL 실행 시 오류 발생함.');
             console.dir(err);

             callback(err, null);

             return;
            }

            callback(null, result);

        });

……
```

코드가 약간 길지만 몽고디비를 사용하던 방식과 코드 구조가 거의 비슷하니 쉽게 이해될 것입니다. addUser 함수에는 다섯 개의 파라미터를 전달합니다. id, name, age, password 파라미터는 웹 브라우저에서 요청할 때 전달한 요청 파라미터이며 callback은 결과를 처리할 콜백 함수입니다. pool 객체의 getConnection 메소드를 호출하면 커넥션 풀에서 연결 객체를 하나 가져올 수 있습니다. 연결 객체를 성공적으로 가져오면 콜백 함수가 호출되면서 conn 파라미터로 연결 객체가 전달됩니다. 연결 객체에는 query 메소드가 있어 SQL문을 실행할 수 있습니다. 여기에서는 데이터를 추가하는 INSERT문을 사용했습니다. SQL문을 만들어 실행할 때는 SQL문 안에 ? 기호를 넣을 수 있는데 이 기호는 query 메소드를 호출할 때 전달하는 추가 파라미터를 사용해 대체한 후 실행됩니다. 즉, data 변수에 {id:id, name:name, age:age, password:password} 객체를 할당한 후 query 메소드를 호출하면서 파라미터로 전달하면 SQL문은 다음과 같은 형태로 만들어진 후 실행됩니다.

```
insert into users set id='test01', name='김준수', age=20, password='123456'
```

SQL문이 실행되면 콜백 함수가 호출되면서 결과가 result 파라미터로 전달됩니다. SQL문을 실행한 후에는 연결 객체의 release 메소드를 호출하여 연결 객체를 커넥션 풀로 반환해야 합니다. 실행 결과는 콜백 함수 쪽에서 처리할 수 있도록 callback(null, result) 코드를 넣어 콜백 함수를 실행합니다.

### 사용자 추가 요청을 처리하는 함수 만들기

사용자를 추가할 때 사용하는 라우팅 함수의 코드도 몽고디비를 사용할 때와 거의 비슷합니다. 다음과 같이 사용자 추가 기능을 처리하는 함수를 추가합니다.

**참조 파일** | DatabaseExample>app8.js

```
.....

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

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

    if(pool) {
        addUser(paramId, paramName, paramAge, paramPassword, function(err, result) {
            if(err) {throw err;}
```

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

         console.log('inserted ' + result.affectedRows + ' rows');

            var insertId = result.insertId;
            console.log('추가한 레코드의 아이디 : ' + insertId);

         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();
}

});

.......
```

POST 방식으로 요청된 /process/adduser 패스를 처리하는 코드는 앞에서 살펴본 것과 거의 같습니다. 먼저 웹 브라우저로부터 id, name, age, password 파라미터를 받아 처리하도록 수정합니다. 그리고 addUser 함수를 실행한 결과를 확인한 후 응답을 보냅니다.

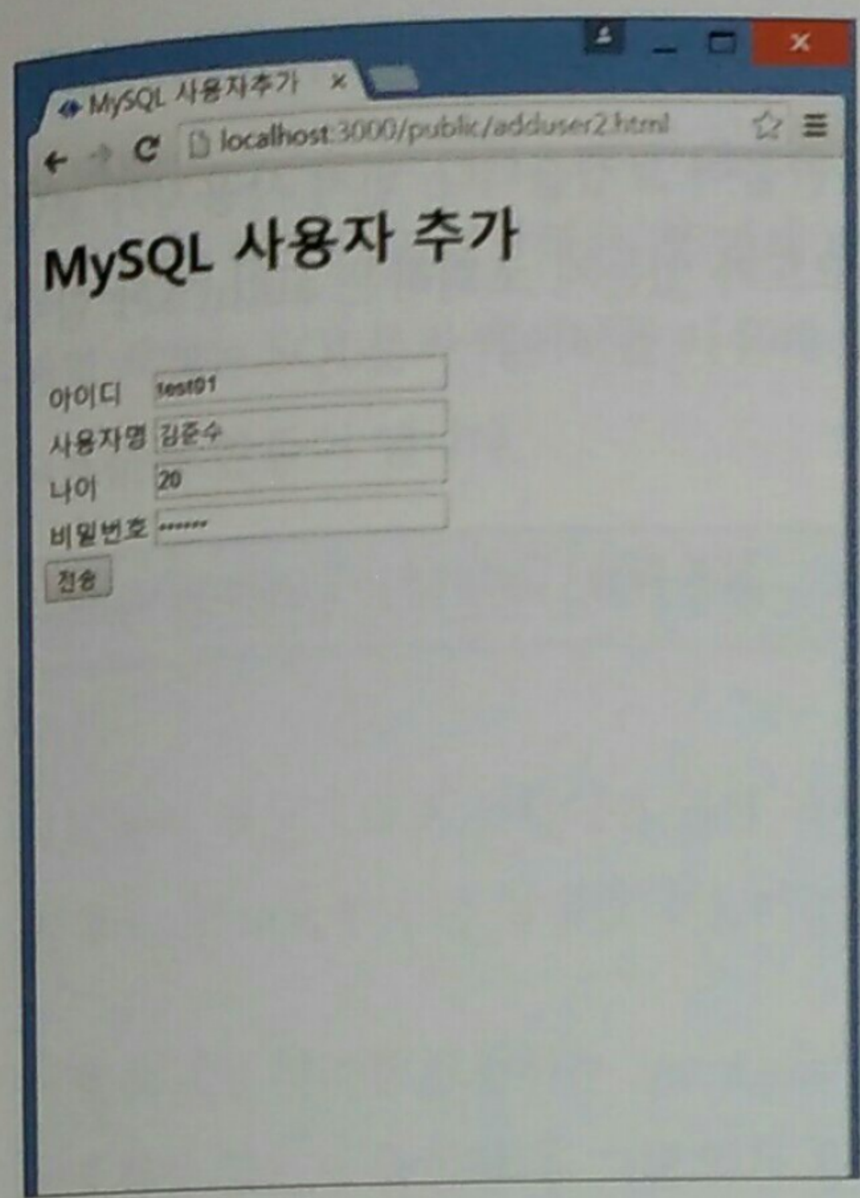
이제 사용자를 등록할 때 [public] 폴더 안에 만들었던 adduser.html 파일을 복사해 adduser2.html 파일을 만들고 다음 코드처럼 〈body〉 안의 태그를 수정합니다.

**참조 파일** | DatabaseExample>/public/adduser2.html

```
……

    <h1>MySQL 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>사용자명</label></td>
                <td><input type="text" name="name" ></td>
            </tr>
            <tr>
                <td><label>나이</label></td>
                <td><input type="text" name="age" ></td>
            </tr>
            <tr>
                <td><label>비밀번호</label></td>
                <td><input type="password" name="password" ></td>
            </tr>
        </table>
        <input type="submit" value="전송" >
    </form>
……
```

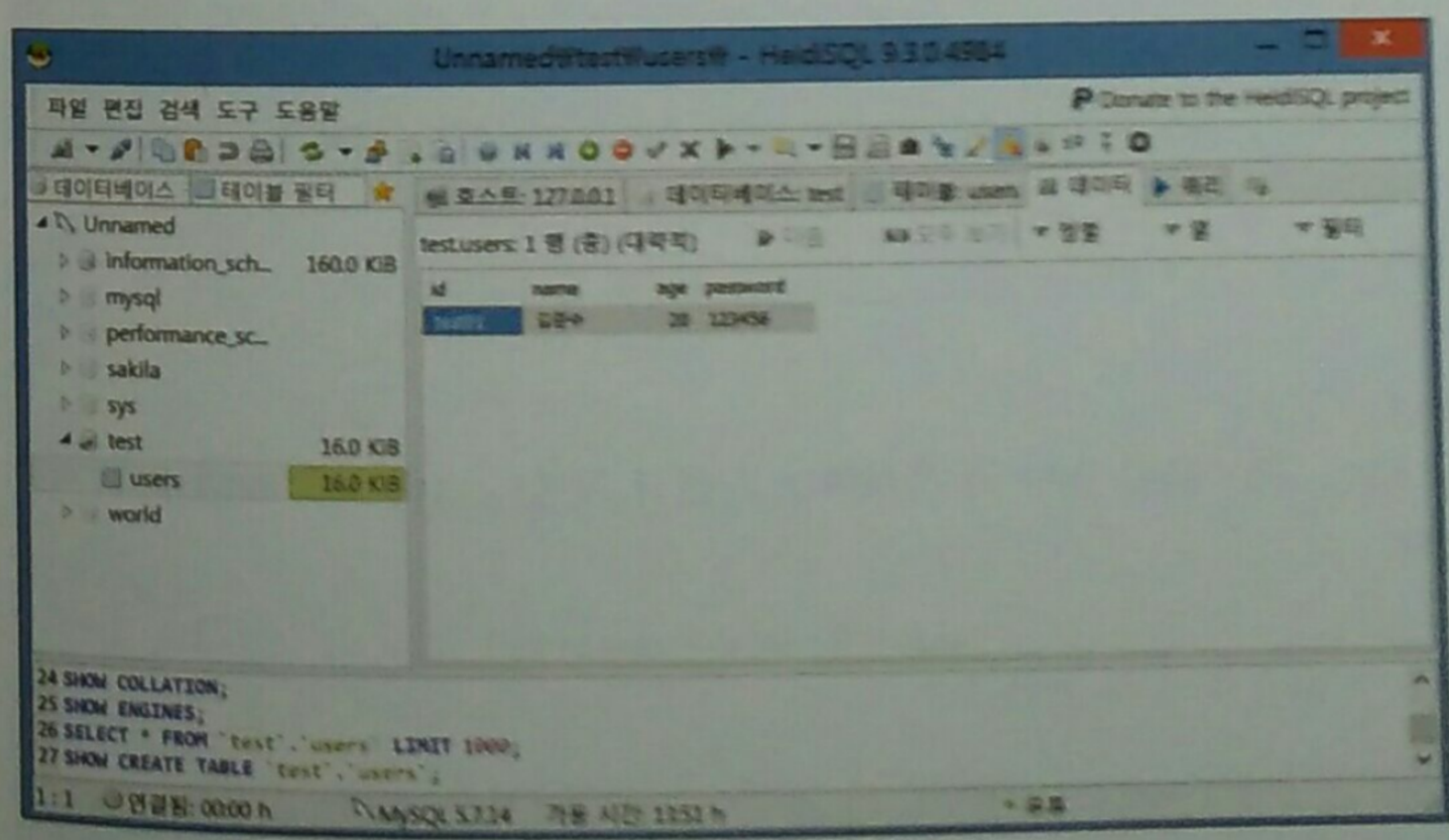
〈input〉 태그가 하나 더 추가되었습니다. 이제 app8.js 파일을 실행한 후 웹 브라우저를 열고 adduser2.html 파일을 엽니다.

▶ http://localhost:3000/public/adduser2.html




◀ MySQL 데이터베이스에 사용자를 추가하려고 만든 웹 페이지

MySQL에 사용자를 추가할 수 있는 화면이 나타나면 아이디 입력란에 test01, 사용자 이름 입력란에 김준수, 나이 입력란에 20, 비밀번호 입력란에 123456을 입력한 후 [전송] 버튼을 누릅니다. 사용자가 정상적으로 추가되면 성공 메시지가 표시됩니다. MySQL에 실제로 데이터가 추가되었는지 확인하기 위해 HeidiSQL 화면에서 테이블 데이터를 조회해 보겠습니다. 왼쪽 목록에서 test 데이터베이스를 찾고 그 안에 들어 있는 users 테이블을 선택합니다. 그리고 오른쪽 영역에서 [데이터] 탭을 클릭하면 users 테이블의 데이터가 조회됩니다.




▲ users 테이블의 데이터를 조회한 화면

데이터가 정상적으로 추가된 것을 확인할 수 있습니다. 데이터베이스에 사용자를 추가하는 기능을 만들어 보았습니다. 이제 노드에서 어떻게 MySQL 데이터베이스를 연결하여 사용하는지 어느 정도 이해했을 것입니다.

## MySQL에 들어 있는 사용자 정보로 로그인하기

이제 MySQL에 들어 있는 사용자 정보로 로그인하는 기능을 만들어 보겠습니다. 먼저 사용자가 보낸 아이디와 비밀번호를 사용해 데이터베이스 안에 일치하는 레코드가 있는지 조회하는 authUser 함수를 만듭니다.

**참조 파일** | DatabaseExample>app8.js

```
……

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

    // 커넥션 풀에서 연결 객체를 가져옵니다.
    pool.getConnection(function(err, conn) {
        if(err) {
            conn.release();  // 반드시 해제해야 합니다.
            return;
        }
        console.log('데이터베이스 연결 스레드 아이디 : ' + conn.threadId);

        var columns = ['id', 'name', 'age'];
        var tablename = 'users';

        // SQL문을 실행합니다.
        var exec = conn.query("select ?? from ?? where id = ? and password = ?",
                              [columns, tablename, id, password], function(err, rows) {
            conn.release();  // 반드시 해제해야 합니다.
            console.log('실행 대상 SQL : ' + exec.sql);

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

클라이언트로부터 전달받는 파라미터는 아이디와 비밀번호입니다. authUser 함수는 이 두 개의 파라미터와 함께 콜백 함수를 전달받습니다. 그리고 그 안에서 getConnection 메소드를 사용해 커넥션 풀에서 연결 객체를 가져옵니다. 그다음에는 연결 객체의 query 메소드로 SQL문을 실행하는데 데이터를 조

회해야 하므로 SELECT문을 사용합니다. SELECT문에는 어떤 칼럼을 어떤 테이블에서 조회할지 그리고 어떤 조건으로 조회할지 등의 정보를 넣어주는데 이 세 가지 정보는 객체나 변수로 만들어 전달합니다. query 메소드로 전달하는 첫 번째 파라미터는 SQL문이고 두 번째 파라미터는 배열 객체입니다. 이 배열 객체는 SQL문과 데이터를 사용해 SQL문 안에 들어 있는 ?? 또는 ? 기호를 대체한 후 다음과 같은 SQL문을 만들어 냅니다.

```
select id, name, age from users where id = 'test01' and password = '123456'
```

SQL문이 정상적으로 실행되면 연결 객체의 release 메소드를 호출하여 연결 객체를 커넥션 풀로 반환한 후 callback 함수를 실행합니다. 이렇게 하면 이 함수를 호출한 쪽에서 rows 객체를 전달받게 됩니다.

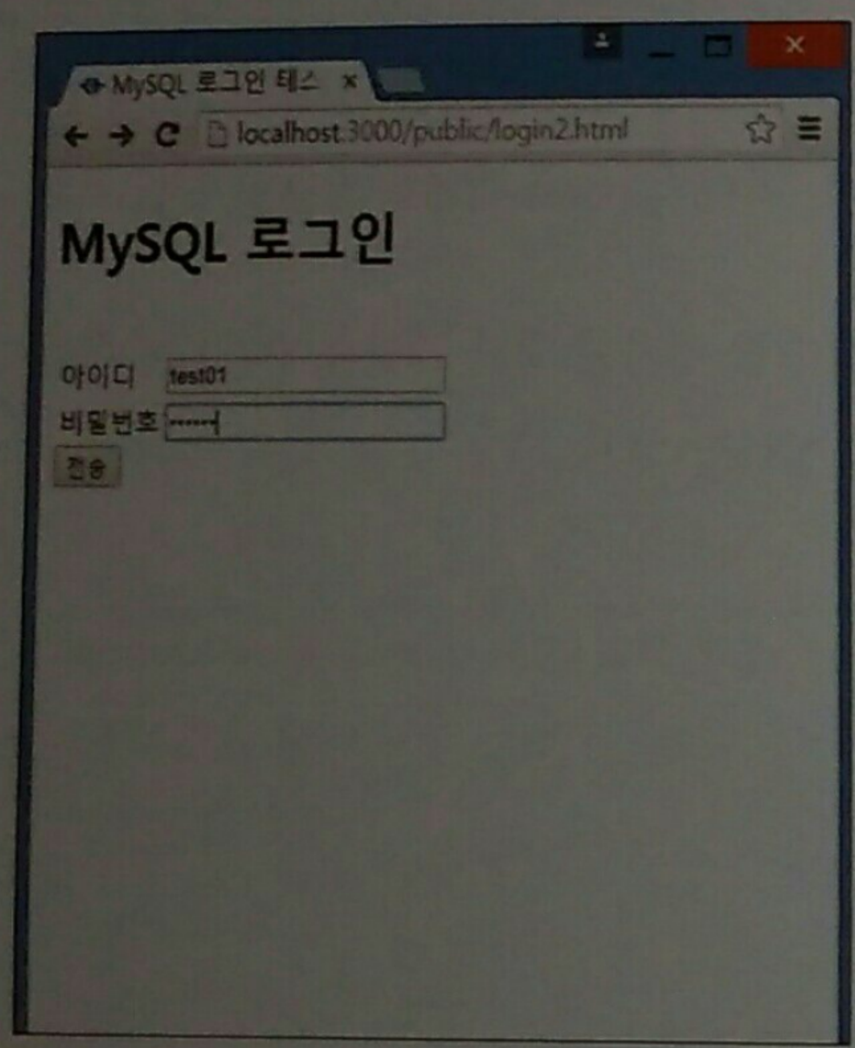
이제 로그인 처리를 요청하는 /process/login 패스에 라우팅 함수를 추가할 때가 되었습니다. 다음처럼 사용자가 있는지 확인한 후 그에 맞는 응답을 보내 주는 코드를 입력합니다.

참조 파일 | DatabaseExample>app8.js

```
.....

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

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

    if(pool) {
        authUser(paramId, paramPassword, function(err, rows) {
            if(err) {throw err;}

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

                var username = rows[0].name;

                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/login2.html'>다시 로그인하기</a>");
                res.end();

.....
```

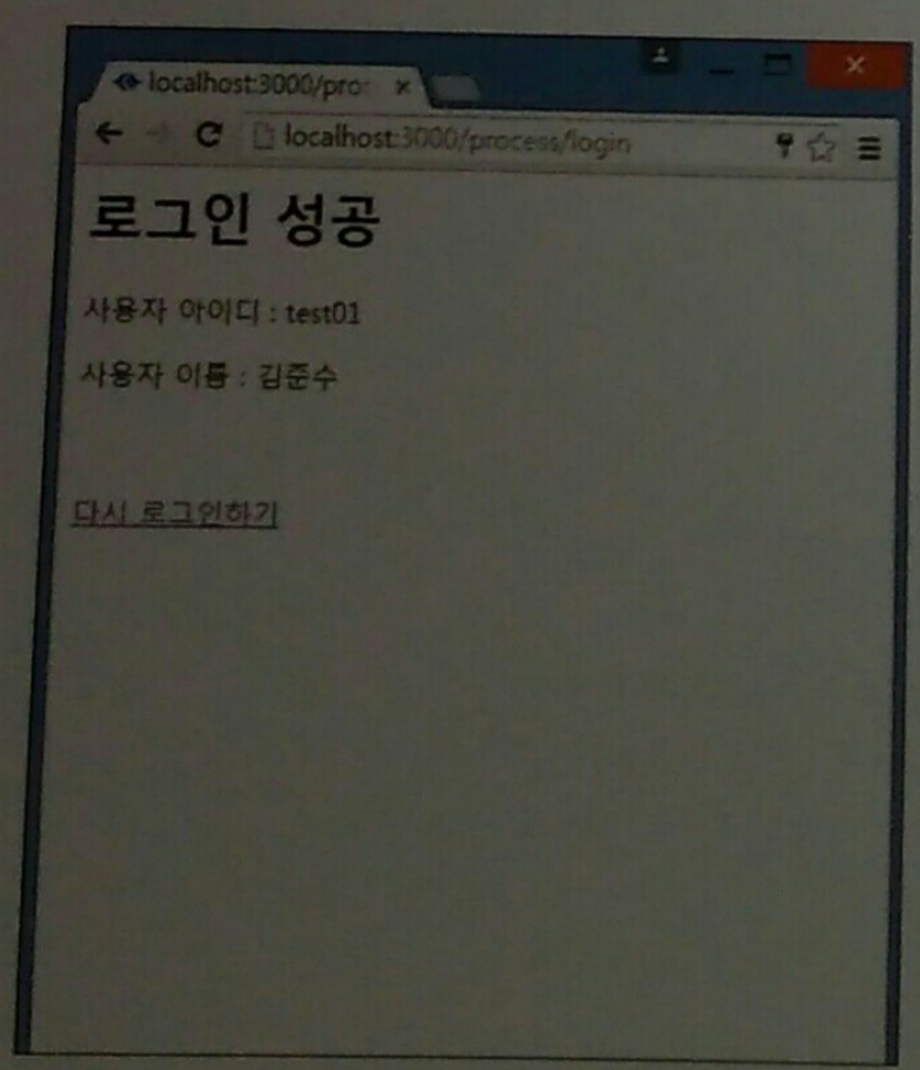
사용자가 요청한 id와 password 파라미터를 확인한 후 authUser( ) 메소드를 호출하면서 파라미터로 전달합니다. MySQL 데이터베이스에서 조회한 사용자 데이터는 rows 파라미터에 들어 있으므로 rows 객체의 첫 번째 배열 요소를 참조한 후 사용자 이름을 확인합니다. 웹 페이지는 login.html 파일을 그대로 사용해도 되지만 제목을 약간 바꾸기 위해 login.html 파일을 복사해서 새로운 login2.html 파일을 만들어 제목을 변경합니다. 이제 app8.js 파일을 실행한 후 웹 브라우저에서 다음 주소로 요청합니다.

```
▶ http://localhost:3000/public/login2.html
```




▲ MySQL의 데이터로 사용자 인증을 요청하는 웹 페이지

사용자 로그인 화면이 나타나면 아이디 입력란에 test01, 비밀번호 입력란에 123456을 입력한 후 [전송] 버튼을 누릅니다. 사용자가 정상적으로 인증되면 성공 메시지와 함께 조회한 사용자 정보가 표시됩니다.



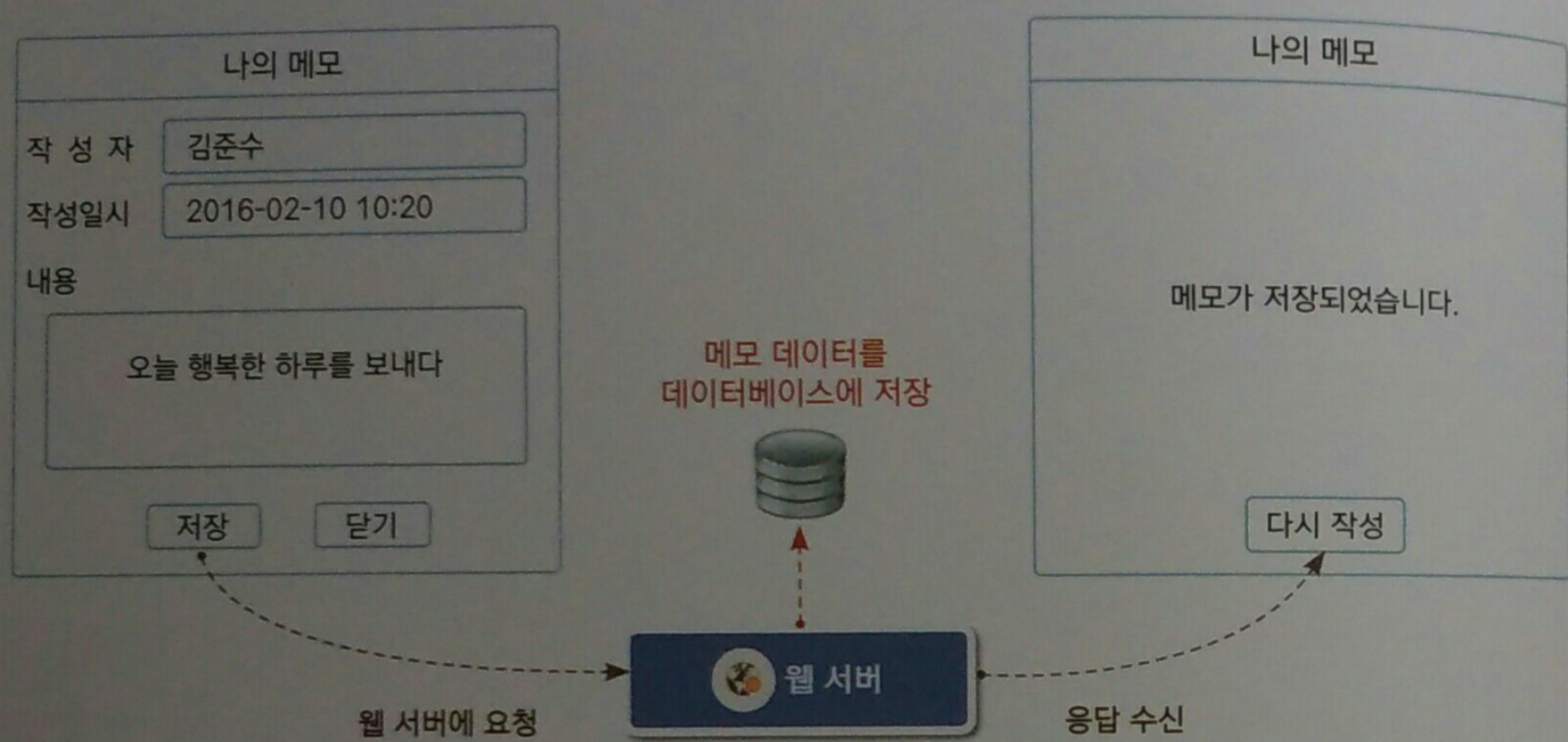

▲ 로그인 후 사용자 정보를 조회한 결과

이제 노드에서 어떻게 MySQL 데이터베이스를 사용할 수 있는지 충분히 이해했을 것입니다. 데이터베이스에 저장한 데이터를 UPDATE SQL문으로 업데이트하는 방법이나 DELETE SQL문으로 삭제하는 방법은 여러분이 직접 해보기 바랍니다.

Do it!
도전 문제

01. 05장에서 만든 Mission-03 예제에 데이터베이스 기능을 추가합니다. 메모를 기록하여 웹 서버로 보내면 메모 내용을 데이터베이스에 저장하는 기능을 만들어 보세요.

프로젝트 소스 Mission05 난이도 초급 ★☆☆☆




Hint!

1 작성 화면의 구성은 Mission-03과 같으므로 해당 소스 파일을 복사하여 새로운 파일을 만듭니다. 코드는 수정할 필요가 없습니다.

2 작성 화면에서 [저장] 버튼을 누르면 웹 서버로 메모 내용을 보냅니다. 웹 서버에서는 메모 내용을 확인한 후 메모 내용을 데이터베이스에 저장합니다. 메모 내용을 저장하기 위해 스키마를 만들고 저장하는 메소스들을 추가합니다.

3 데이터베이스에 저장한 후 정상적으로 저장되었다는 응답 메시지를 클라이언트로 보냅니다.

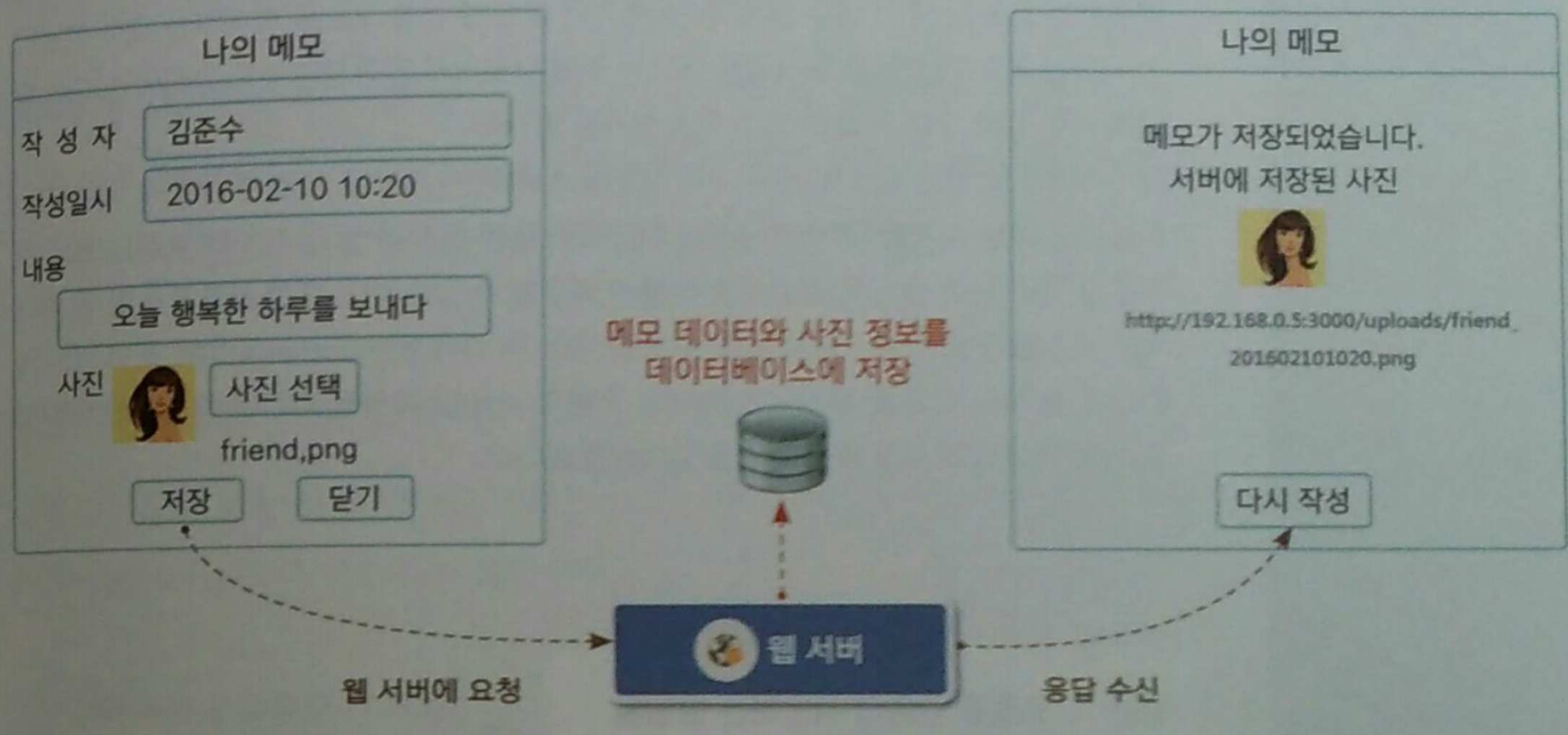
4 클라이언트에서 응답 메시지를 받으면 응답 화면에 메시지를 보여 줍니다. 응답 화면의 구성도 Mission-03과 같으므로 수정할 필요가 없습니다.

MEMO

Do it!
도전 문제

02. 05장에서 만든 Mission-04 예제에 데이터베이스 기능을 추가합니다. 메모를 기록하여 웹 서버로 보내면 메모 내용과 사진 정보를 데이터베이스에 저장하는 기능을 만들어 보세요.

프로젝트 소스 Mission06 난이도 중급 ★★☆☆




Hint!

1 작성 화면의 구성은 Mission-04와 같으므로 해당 소스 파일을 복사하여 새로운 파일을 만듭니다. 코드는 수정할 필요가 없습니다.

2 작성 화면에서 [저장] 버튼을 누르면 웹 서버로 메모 내용을 보내고 사진 정보도 함께 보냅니다. 웹 서버에서는 메모 내용을 확인하고 사진을 업로드한 후 데이터베이스에 메모 내용과 사진 정보를 저장합니다. 사진은 파일로 업로드되었으므로 사진이 있는 파일 패스만 데이터베이스에 저장하면 됩니다.

3 데이터베이스에 저장한 후 정상적으로 저장되었다는 응답 메시지를 클라이언트로 보냅니다.

4 클라이언트에서 응답 메시지를 받으면 응답 화면에 메시지와 사진을 보여 줍니다. 응답 화면의 구성도 Mission-04와 같으므로 수정할 필요가 없습니다.

MEMO

# 07 익스프레스 프로젝트를 모듈화하기

지금까지 익스프레스로 웹 서버를 만들고 사용자 정보를 데이터베이스에 저장하는 방법에 대해 알아보았습니다. 그런데 하나둘씩 따라하다 보면 '기능이 추가되는 것은 좋지만 입력하는 코드의 양이 너무 많네'라는 생각이 들면서 점점 코드를 해석하는 것이 부담되기 시작합니다. 이것은 프로그램을 만드는 것에 익숙하지 않아서일 수도 있지만 app.js 파일에 너무 많은 코드가 들어 있기 때문이기도 합니다.

익스프레스로 만든 웹 서버는 각각의 요청을 처리하는 기능을 계속 추가할 수 있습니다. 이렇게 기능을 추가할 때마다 app.js 파일에는 많은 코드가 쌓입니다. 이 문제는 코드 일부분을 잘라 내어 별도의 파일로 분리해서 해결할 수 있습니다. 또한 사용자가 볼 웹 페이지도 뷰 템플릿으로 분리해서 만들 수 있습니다. 이 장에서는 app.js 파일에 들어 있는 많은 코드를 별도의 모듈 파일로 분리하는 방법과 app.js 파일을 자주 수정하지 않도록 설정 파일을 만들어 사용하는 방법을 알아보겠습니다.

질문으로 시작하기

질문1. 모듈을 만들어 사용하는 방법을 자세히 알고 싶나요?
➡ 07-1. 모듈화 방법 자세히 살펴보기

질문2. 사용자 정보 관련 기능을 모듈로 만들어 볼 수 있나요?
➡ 07-2. 사용자 정보 관련 기능을 모듈화하기

질문3. 설정 정보를 별도 파일에 넣어 두고 싶나요?
➡ 07-3. 설정 파일 만들기

질문4. 사용자를 위해 웹 문서를 예쁘게 만들고 싶나요?
➡ 07-4. UI 라이브러리로 웹 문서 예쁘게 꾸미기

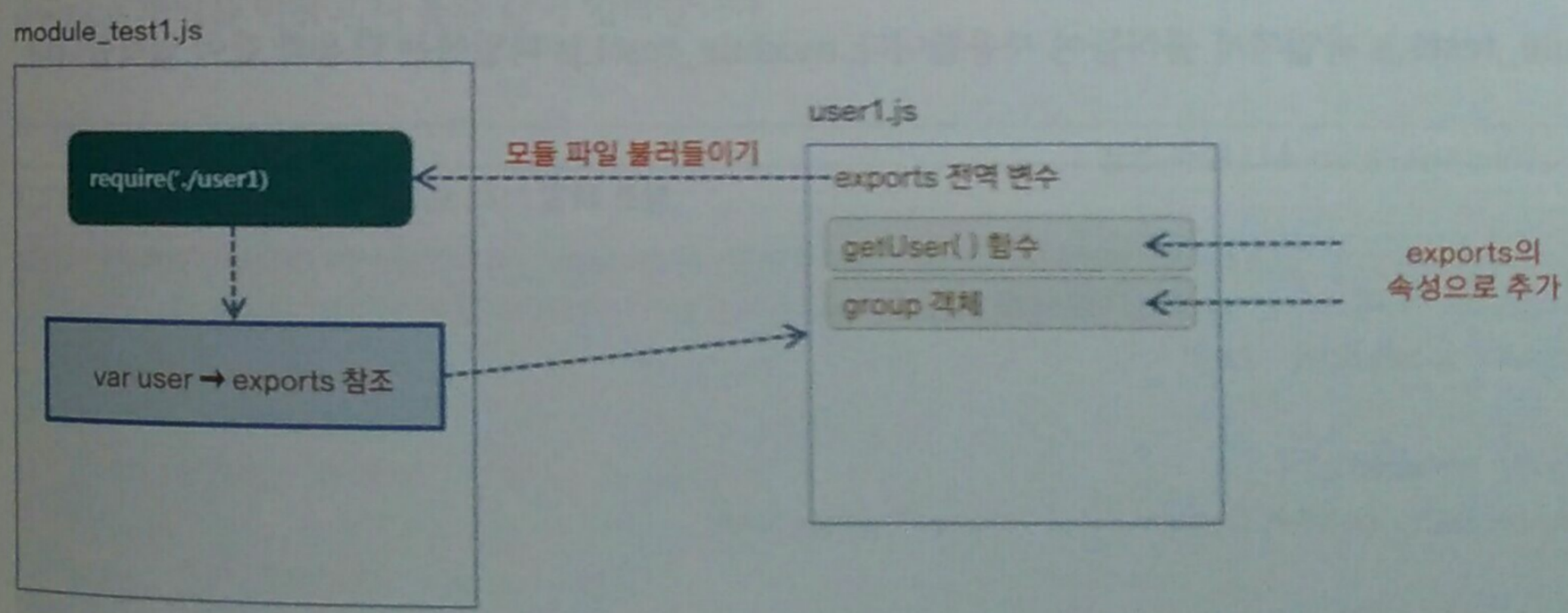
# 07-1 | 모듈화 방법 자세히 살펴보기

지금까지 만든 사용자 기능이나 데이터베이스 저장 기능은 모두 하나의 app.js 파일 안에 들어 있습니다. 따라서 기능이 많아질수록 하나의 파일에 들어가는 코드의 양도 많아지겠죠. 결국 코드를 분석하기 어려워집니다. 이 문제를 해결하려면 기능별로 코드를 구분한 후 독립된 파일로 분리시키는 것이 좋습니다. 이렇게 별도의 파일로 분리한 것을 모듈이라고 합니다.

익스프레스로 웹 서버를 만드는 과정에서 이미 모듈에 대해 알아보았습니다. 이 모듈은 구성 방법에 따라 몇 가지 전형적인 코드 패턴을 갖습니다. 여기에서는 모듈을 더 자세히 알아본 후 DatabaseExample 프로젝트의 app5.js 파일에서 구현한 각 기능을 모듈로 분리하여 구성해 보겠습니다.

## 다양한 방법으로 모듈 만들기

가장 기본적인 모듈 사용 방법은 자바스크립트 파일을 새로 만드는 것부터 출발합니다. 그리고 그 파일 안에서 exports 전역 변수를 사용합니다.




▲ exports 전역 변수를 사용하여 모듈 구성하기

exports 전역 변수는 어디에서나 접근할 수 있게 정의된 것이고 속성도 추가할 수 있습니다. 즉, 새로운 파일을 만들고 그 안에서 exports 전역 변수에 속성을 추가하면 다른 파일에서도 exports 전역 변수의 속성을 참조할 수 있습니다. 예를 들어, 사용자 정보를 확인하려고 만든 기능을 별도의 user1.js 자바스크립트 파일로 분리한 후 모듈로 구성하려면 getUser( ) 함수와 group 객체를 exports의 속성으로 추가해야 합니다. 이렇게 만든 모듈은 module_test1.js처럼 이 모듈을 사용할 파일에서 require( ) 메소드로 불러들일 수 있습니다. require( ) 메소드로 불러들인 모듈을 변수에 할당하면 그 변수에는 exports가 속성으로 할당됩니다. 따라서 속성이 할당된 변수에서는 exports에 추가한 getUser( ) 함수나 group 객체를 참조하여 사용할 수 있습니다.

위와 같은 구조를 만들기 위해 먼저 새로운 익스프레스 프로젝트 ModuleExample을 만듭니다. 프로젝트 안에는 user1.js 파일과 module_test1.js 파일을 새로 만들어 추가합니다. user1.js 파일에 들어가는 코드는 다음과 같습니다.

**참조 파일** | ModuleExample>user1.js

```
// exports 객체 속성으로 함수 추가
exports.getUser = function() {
  return {id : 'test01', name : '소녀시대'};
}

// exports 객체 속성으로 객체 추가
exports.group = {id : 'group01', name : '친구'};
```

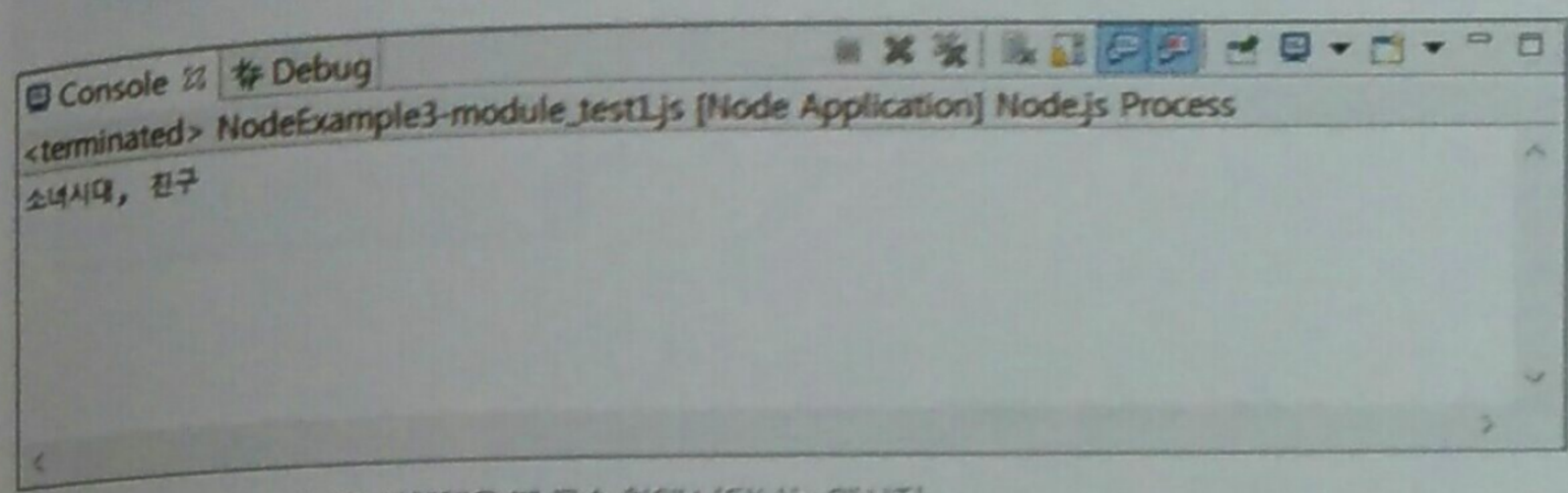
exports에 속성 두 개가 추가되었습니다. 첫 번째 속성은 getUser 속성이며 속성의 값으로 함수가 할당되었습니다. 이 함수는 id와 name 속성을 가진 객체를 반환하는데, 사용자 정보를 객체로 반환받아 확인하려고 만든 함수입니다. 두 번째 속성은 group 속성이며 id와 name 속성을 가진 객체가 할당되었습니다. 이것은 그룹의 아이디와 이름을 알 수 있도록 만든 객체입니다. 이렇게 만든 모듈 파일은 module_test1.js 파일에서 불러들여 사용합니다. module_test1.js 파일에는 다음과 같이 입력합니다.

**참조 파일** | ModuleExample>module_test1.js

```
// require( ) 메소드는 exports 객체를 반환함
var user1 = require('./user1');

function showUser() {
  return user1.getUser().name + ', ' + user1.group.name;
}

console.log('사용자 정보 : %s', showUser());
```

user1.js 파일로 만든 모듈을 사용하기 위해 require( ) 메소드를 호출합니다. require( ) 메소드로 불러들인 모듈은 user1 변수에 할당됩니다. require( ) 메소드는 exports 객체를 반환하기 때문에 user1 변수는 exports 객체를 참조합니다. 이제 user1 변수는 exports 객체와 같다고 볼 수 있으므로 user1 변수에 들어 있는 getUser( ) 함수를 호출할 수 있습니다. showUser( ) 함수는 콘솔 창에 로그를 출력하는 간단한 기능을 수행합니다. showUser( ) 함수 안에 들어 있는 코드는 user 변수의 getUser( ) 메

소드를 호출하여 반환받은 객체의 name 속성을 참조하여 사용자의 이름을 알아냅니다. 그리고 group 객체의 name 속성을 참조하여 그룹 이름을 알아낸 후 두 개의 문자열을 나란히 붙여서 반환합니다. module_test1.js 파일을 실행하면 콘솔 창에서 다음과 같은 결과를 볼 수 있습니다.




▲module_test1.js 파일을 실행했을 때 콘솔 창에 나타나는 메시지

이 결과는 user1 변수가 exports 객체를 참조하여 출력한 것입니다.

### exports에 객체 지정하기

user1.js 파일에서 exports 객체의 속성으로 getUser( ) 함수와 group 객체를 추가했으니 exports에 객체를 만들어 할당할 수도 있지 않을까요? exports에 객체를 할당할 수 있는지 확인해 보기 위해 user2.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | ModuleExample>user2js

```
// Reason : exports는 속성으로, exports에 속성을 추가하면 모듈에서 접근하지만
//          exports에 객체를 지정하면 자바스크립트에서 새로운 변수로 처리함
exports = {
  getUser : function() {
    return {id : 'test01', name : '소녀시대'};
  },
  group: {id : 'group01', name : '친구'}
}
```

이렇게 코드를 입력하면 getUser( ) 함수와 group 객체를 속성으로 가진 객체를 만들어 exports에 할당하게 됩니다. 따라서 exports 속성으로 getUser( ) 함수와 group 객체를 추가한 것과 같은 효과를 냅니다. 일반적인 자바스크립트 코드에서는 user1.js 파일의 코드와 user2.js 파일의 코드가 같은 것처럼 보이는 것이죠. 그럼 이 코드가 정상으로 동작하는지 확인하기 위해 module_test2.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | ModuleExample>module_test2.js

```
// Reason : user2.js 파일에서 exports에 객체를 할당하였으므로,
//          require( )를 호출할 때 자바스크립트에서 새로운 변수로 처리함
//          결국 아무 속성도 없는 { } 객체가 반환됨
var user = require('./user2');

console.dir(user);

function showUser() {
  return user.getUser().name + ', ' + user.group.name;
}

console.log(showUser());
```

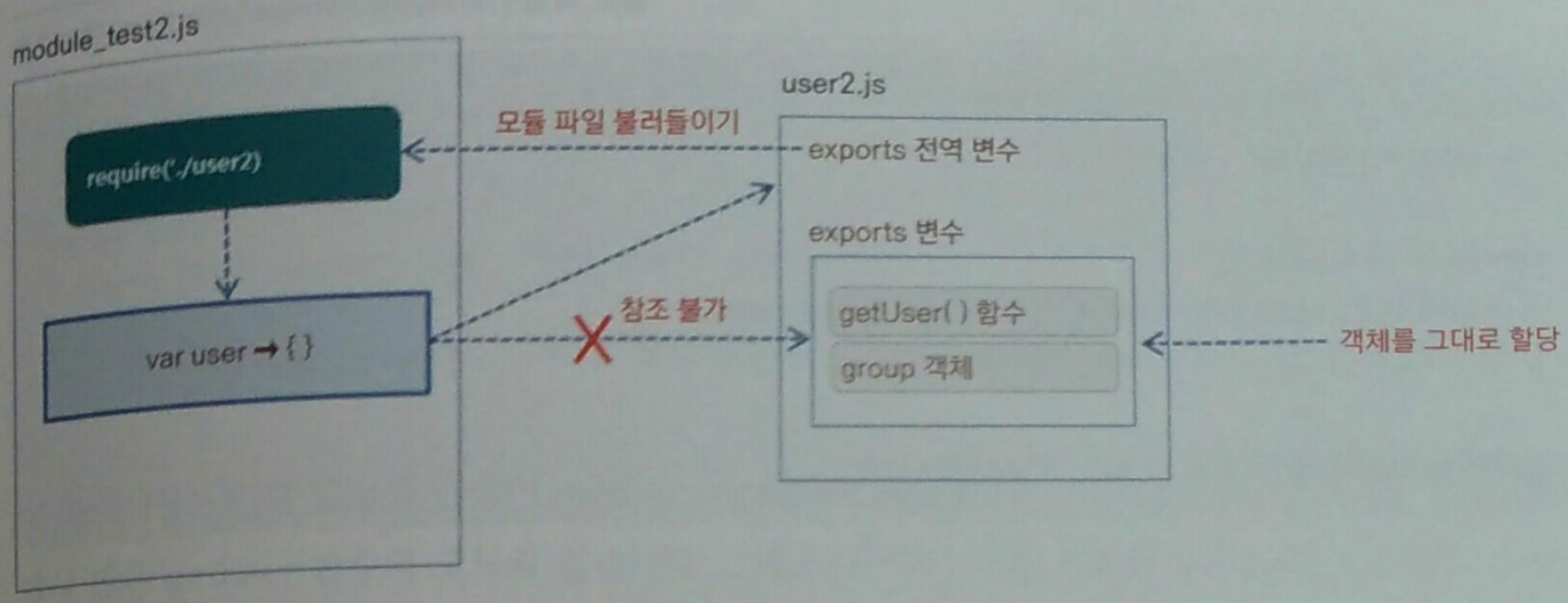
module_test1.js 파일에 입력한 코드와 같지만 require( ) 메소드로 불러들이는 모듈 파일의 이름을 ./user2로 하였습니다. 그리고 user 변수에 할당한 객체의 속성 값을 확인하기 위해 console.dir( ) 메소드를 호출하였습니다. 그런데 module_test2.js 파일을 실행해 보면 다음과 같이 오류가 표시되면서 정상적으로 실행되지 않습니다.

```
Console  Debug
<terminated> NodeExample3-module_test2.js [Node Application] Node.js Process
{}
C:\Users\user\workspace-book5\NodeExample3\module_test2.js:17
        return user.getUser().name + ', ' + user.group.name;
                    ^
TypeError: undefined is not a function
    at showUser (C:\Users\user\workspace-book5\NodeExample3\module_test2.js:
```

▲ 모듈에서 exports에 객체를 지정했을 때 나타나는 오류

잘 보면 user 변수에는 객체가 할당되었지만 어떤 속성도 들어 있지 않습니다. 이 때문에 user 변수의 getUser( ) 메소드를 호출할 때 오류가 발생한 것입니다. 왜 모듈을 불러와도 모듈 파일 안에서 exports에 할당한 객체가 보이지 않는 것일까요?

노드는 모듈을 처리할 때 exports를 속성으로 인식합니다. 이 속성에 함수나 객체를 속성으로 추가하면 모듈을 불러들인 쪽에서 exports에 추가된 속성들을 참조할 수 있습니다. 그러나 exports에 객체를 할당하면 모듈 파일 안에서 선언한 exports는 모듈 시스템에서 처리할 수 있는 전역 변수가 아닌 단순 변수로 인식됩니다. 이 때문에 모듈을 불러들인 쪽에서 exports를 참조할 수 없게 되고 결과적으로 모듈 파일을 불러들일 때 그 결과 객체에는 아무것도 들어 있지 않게 됩니다.




▲exports에 객체를 그대로 할당한 경우

결국 user2.js 파일에서 exports에 객체를 할당하면 전역 변수 exports와 무관한 모듈 파일 안의 변수가 되고 이 모듈을 불러들인 쪽에서는 비어 있는 exports 전역 변수만 참조하게 됩니다.

### module.exports를 사용해서 객체를 그대로 할당하기

자바스크립트 코드를 만들 때 객체 안에 속성을 넣어 두는 경우가 많습니다. 따라서 모듈 파일에서 이 모듈을 불러들인 쪽으로 객체를 전달하는 것이 훨씬 편할 수 있습니다. 그렇다면 객체를 그대로 할당하는 방법은 없을까요? exports가 아니라 module.exports를 사용하면 객체를 그대로 할당할 수 있습니다. module.exports를 사용해 보기 위해 user3.js 파일을 만들고 다음과 같이 코드를 입력합니다.

참조 파일 | ModuleExample>user3.js

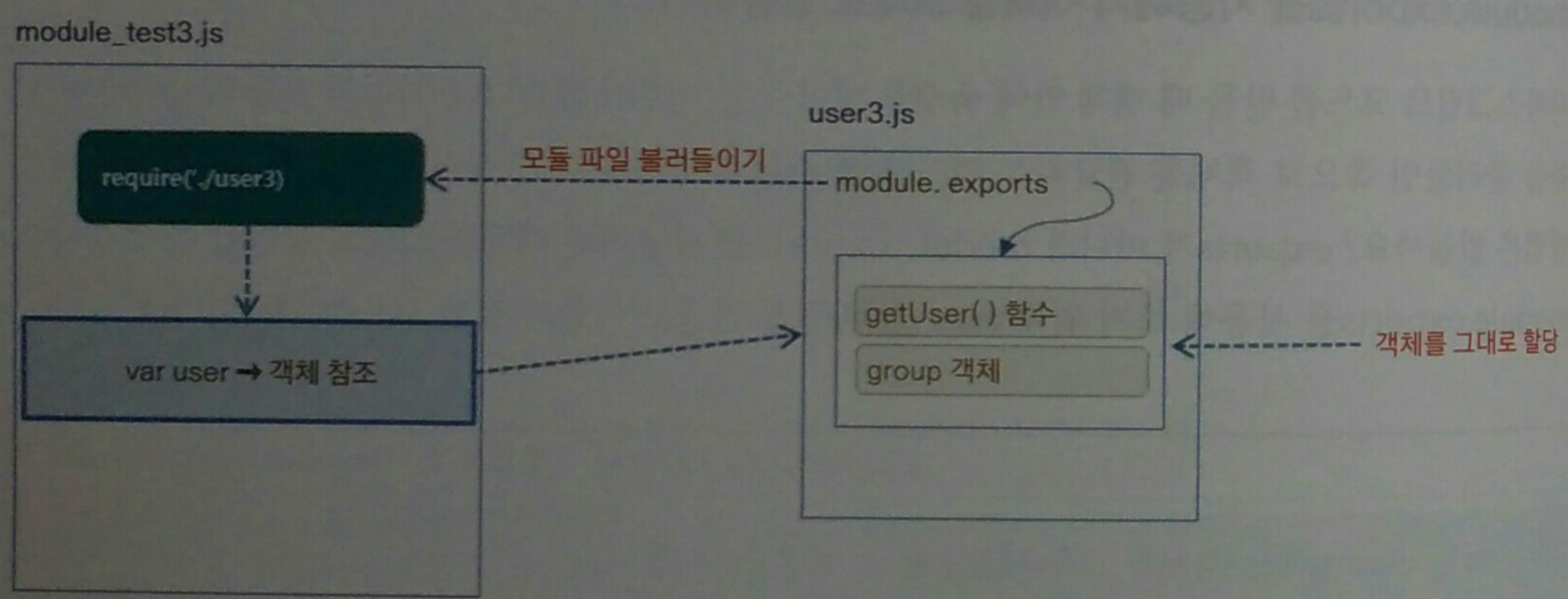
```
// module.exports에는 객체를 그대로 할당할 수 있음
var user = {
  getUser : function() {
    return {id : 'test01', name : '소녀시대'};
  },
  group : {id : 'group01', name : '친구'}
}

module.exports = user;
```

getUser( ) 메소드와 group 객체를 속성으로 갖는 객체를 새로 만들고 user 변수에 할당합니다. 이 user 변수는 다시 module.exports에 그대로 할당되었습니다. 이제 user3.js 모듈을 사용하는 module_test3.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | ModuleExample>module_test3.js

```
// require( ) 메소드는 객체를 반환함
var user = require('./user3');

function showUser() {
  return user.getUser().name + ', ' + user.group.name;
}

console.log('사용자 정보 : %s', showUser());
```

module_test3.js 파일의 코드는 module_test1.js 파일의 코드와 같으며, require( ) 메소드에 전달되는 파일 이름만 ./user3으로 변경되었습니다. 이 파일을 실행하면 module_test1.js 파일을 실행했을 때와 똑같은 결과가 콘솔 창에 표시됩니다. 모듈을 만들 때 module.exports에 객체를 그대로 할당하는 방식으로 만들었다는 점과 이 모듈을 require( ) 메소드로 불러들이면 그 객체가 그대로 반환된다는 점을 잘 이해해 두세요.




▲ module.exports에 객체를 그대로 할당한 경우

exports에 객체를 그대로 할당하면 모듈 파일을 불러들인 쪽에서 그 객체를 참조할 수 없지만, module.exports에 객체를 그대로 할당하면 모듈 파일 안에서 할당한 객체를 참조할 수 있습니다. 이 때문에 아무런 오류가 발생하지 않습니다.

### module.exports에 함수만 할당하기

module.exports에는 자바스크립트 객체만 할당할 수 있을까요? 함수도 객체이므로 함수만 할당할 수도 있습니다. user4.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | ModuleExample>user4.js

```
// 인터페이스(함수 객체)를 그대로 할당할 수 있음

module.exports = function() {
  return {id : 'test01', name : '소녀시대'};
};
```

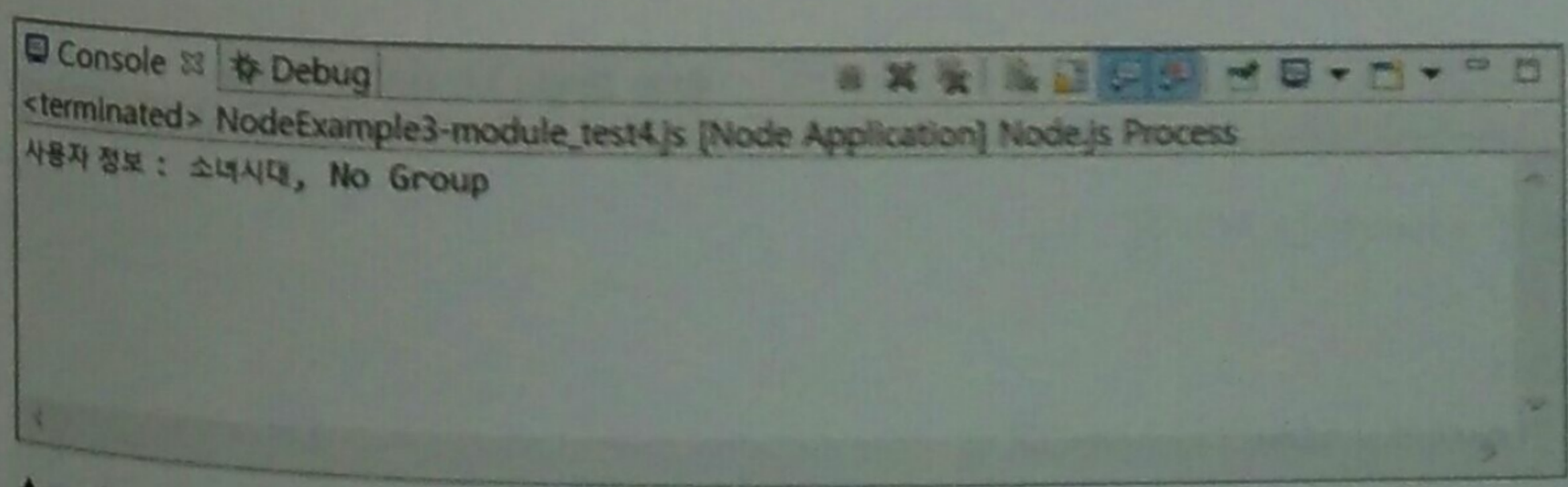
이 파일에서는 익명 함수를 만들어 module.exports에 할당합니다. 익명 함수에 들어 있는 코드는 앞에서 만든 getUser 함수의 코드와 같습니다. 그러고 나서 이 모듈 파일을 불러들여 사용하는 기능을 module_test4.js 파일에 입력합니다.

참조 파일 | ModuleExample>module_test4.js

```
// require( ) 메소드는 함수를 반환함
var user = require('./user4');

function showUser() {
  return user().name + ', ' + 'No Group';
}

console.log('사용자 정보 : %s', showUser());
```

require( ) 메소드를 호출하면 모듈 파일 안에서 module.exports에 할당한 익명 함수가 반환됩니다. 따라서 이 익명 함수를 user 변수에 할당했다면 user( )와 같이 소괄호를 붙여 함수를 실행할 수 있습니다. showUser( ) 함수 안에서 user 변수에 할당한 함수를 실행하면 id와 name 속성을 가진 객체가 반환되므로 그 객체의 name 속성을 참조할 수 있습니다.

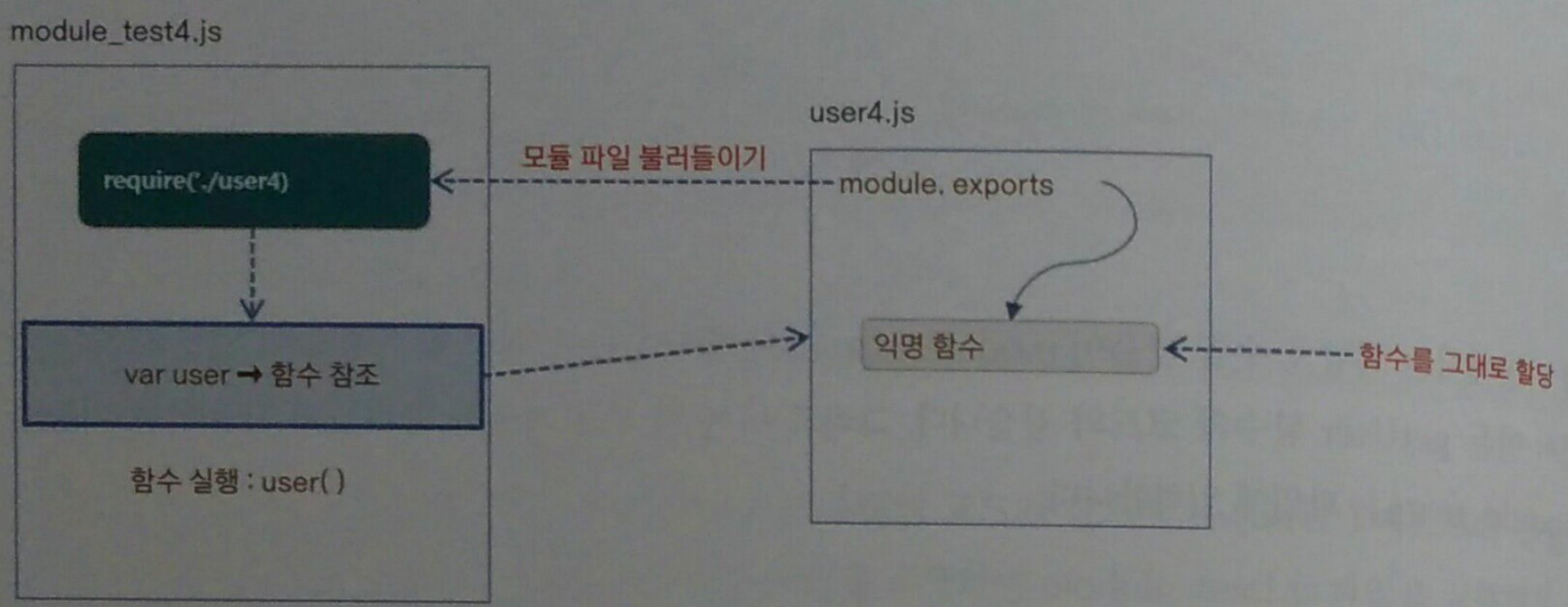
module_test4.js 파일을 실행하면 다음과 같은 결과가 나타납니다.




▲module.exports에 함수를 할당하여 실행한 경우

콘솔에서 확인할 수 있는 것처럼, 모듈 파일 안에서 module.exports에 함수를 할당하면 모듈을 불러들인 쪽에서 그 함수를 그대로 참조할 수 있습니다. 따라서 반환된 함수에 소괄호만 붙여 주면 함수를 그대

로 실행할 수 있습니다. user 변수에 할당된 함수를 실행하면 사용자 정보가 들어 있는 객체가 반환되므로 그 객체의 name 속성을 사용할 수 있습니다.




▲ module.exports에 함수를 할당한 경우

## exports와 module.exports를 함께 사용하기

exports와 module.exports를 함께 사용하면 어떻게 될까요? 이 두 가지를 함께 사용하면 module.exports가 우선으로 적용됩니다. 즉 모듈을 불러오는 쪽에서는 module.exports에 할당된 객체나 속성을 참조할 수 있으며 exports 전역 변수는 무시됩니다.

**정박사의 한마디** module.exports를 사용하는 것을 권장해요

exports를 사용하면 모듈을 불러들인 쪽으로 객체를 전달할 수도 없을뿐더러 module.exports에 의해 무시되는 상황이 발생할 수 있다는 점에 주의하세요. 따라서 실무에서는 module.exports 사용을 권장합니다.

이 내용을 확인하기 위해 user5.js 파일을 만들고 다음처럼 입력합니다.

**참조 파일** | ModuleExample>user5.js

```
// module.exports가 사용되면 exports는 무시됨
module.exports = {
        getUser: function() {
             return {id : 'test01', name : '소녀시대'};
        },
        group : {id : 'group01', name : '친구'}
}

exports.group = {id : 'group02', name : '가족'};
```

module.exports에 객체를 그대로 할당했는데 그 객체에는 group 속성이 들어 있습니다. exports에도 group 속성을 추가하고 그룹의 이름에 '가족'이라는 정보를 넣었습니다.

이제 이 모듈 파일을 사용하는 module_test5.js 파일을 다음과 같이 만듭니다.

참조 파일 | ModuleExample>module_test5.js

```
// require( ) 메소드는 export가 아닌 module.exports로 설정된 속성을 반환함
var user = require('./user5');

function showUser() {
  return user.getUser().name + ', ' + user.group.name;
}

console.log('사용자 정보 : %s', showUser());
```

이 파일을 실행하면 그룹의 name 속성에 '친구'라는 글자가 들어 있고 모듈 파일 안에서 exports.group으로 할당한 '가족'이라는 그룹 이름은 적용되지 않았음을 확인할 수 있습니다.

이제 require( ) 메소드를 호출했을 때 모듈을 어떻게 불러오고 불러온 모듈을 어떻게 사용할 수 있는지 어느 정도 이해가 되었나요? 그렇다면 require( ) 메소드를 호출했을 때 모듈 파일에서 설정한 객체나 함수를 그대로 반환받을 수 있다는 것도 이해할 수 있을 것입니다.

### require( ) 메소드의 동작 방식 이해하기

require( ) 메소드는 다른 모듈 파일에서 만든 객체를 참조할 수 있습니다. 이 require( ) 메소드를 더 쉽게 이해하려면 가상의 함수를 만들어 보는 것도 좋은 방법입니다. module_test6.js 파일을 만들고 다음과 같이 코드를 입력합니다.

참조 파일 | ModuleExample>module_test6.js

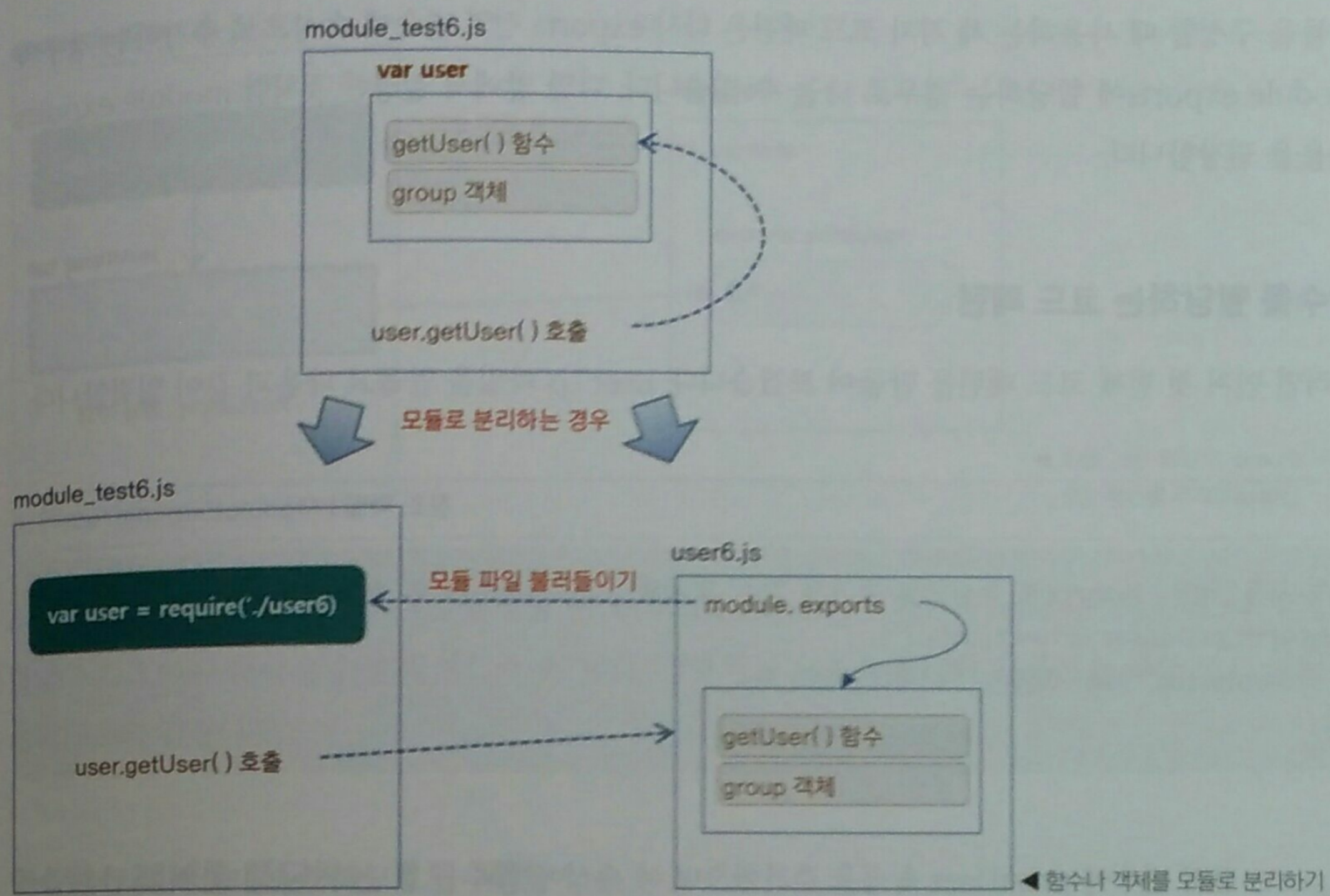
```
// 가상으로 require( ) 함수를 정의해 보면 require( ) 함수가 내부적으로 처리되는 방식을 이해할
수 있음
var require = function(path) {
  var exports = {
    getUser : function() {
      return {id : 'test01', name : '소녀시대'};
    },
    group : {id : 'group01', name : '친구'}
  }

  return exports;
}

var user = require('...');

function showUser() {
  return user.getUser().name + ', ' + user.group.name;
}

console.log('사용자 정보 : %s', showUser());
```

여기에서는 모듈을 불러올 때 사용한 require( ) 함수가 아닌, 여러분이 직접 require( ) 함수를 새로 만들었습니다. 이 require( ) 함수는 한 개의 파라미터를 입력받으며, 이 파라미터를 모듈 파일의 이름처럼 전달합니다. require( ) 함수 안에서는 exports 변수를 만들어 객체를 할당했는데, 함수의 마지막 부분에서 exports 변수를 반환하도록 만들면 require( ) 메소드를 호출했을 때 이 객체를 반환받게 됩니다.

이제 그 아랫부분에서 require( ) 메소드를 호출한 후 반환받은 객체의 메소드를 호출하거나 속성을 참조하도록 만듭니다. 이 부분은 모듈 파일을 불러올 때 사용한 코드와 똑같게 구성합니다. 따라서 이 module_test6.js 파일을 실행하면 모듈 파일을 불러와 사용했을 때와 똑같은 결과를 얻을 수 있습니다.

이렇게 함수를 직접 만들어 실행해 보면 모듈 파일을 불러올 때 사용한 require( ) 함수가 어떻게 동작하는지 이해할 수 있습니다. 그리고 어떤 함수나 객체가 있을 때 이것을 별도의 모듈 파일로 분리시키는 방법이 아주 자연스러운 자바스크립트의 코드 사용 패턴 중 하나라는 것을 알게 될 것입니다.

다음 그림은 모듈 파일을 만들어 불러들이는 경우와 조금 전 직접 만든 코드를 비교해서 보여 줍니다.




별도의 모듈 파일로 분리하면 module.exports에 객체를 할당해야 하고 이 모듈 파일을 require( ) 메소드로 불러와서 변수에 할당하는 과정이 추가됩니다. 그러나 exports를 사용하지 않고 module.exports에 객체를 할당하는 방식으로만 모듈을 구성한다면, 모듈로 분리하는 과정이 생각보다 복잡하지 않습니다.

### 모듈을 분리할 때 사용하는 전형적인 코드 패턴

이제 모듈로 분리하여 구성할 때 사용하는 전형적인 코드 패턴 세 가지를 살펴보겠습니다. 첫 번째 패턴은 이미 살펴본 것처럼 모듈 파일 안에서 함수를 할당합니다. 그리고 두 번째 패턴은 모듈 파일 안에서 인스턴스 객체를 할당하며, 세 번째 패턴은 모듈 파일 안에서 프로토타입 객체를 할당합니다.

| 코드 패턴 | 설명 |
|---|---|
| 함수를 할당하는 경우 | 모듈 안에서 함수를 만들어 할당합니다.<br>모듈을 불러온 후 소괄호를 붙여 모듈을 실행합니다. |
| 인스턴스 객체를 할당하는 경우 | 모듈 안에서 인스턴스 객체를 만들어 할당합니다.<br>모듈을 불러온 후 해당 객체의 메소드를 호출하거나 속성을 사용할 수 있습니다. |
| 프로토타입 객체를 할당하는 경우 | 모듈 안에서 프로토타입 객체를 만들어 할당합니다.<br>모듈을 불러온 후 new 연산자로 인스턴스 객체를 만들어 사용할 수 있습니다. |

[표] 모듈을 만들어 사용하는 전형적인 코드 패턴 세 가지

모듈을 구성할 때 사용하는 세 가지 코드 패턴은 다시 exports 전역 변수에 속성으로 추가하는 경우와 module.exports에 할당하는 경우로 나눌 수 있습니다. 다만 앞에서 설명한 것처럼 module.exports 사용을 권장합니다.

### 함수를 할당하는 코드 패턴

그러면 먼저 첫 번째 코드 패턴을 만들어 보겠습니다. user7.js 파일을 만들고 다음과 같이 입력합니다.

**참조 파일** | ModuleExample>user7.js

```
// 사용 패턴: exports에 속성으로 추가된 함수 객체를 그대로 참조한 후 호출함
exports.printUser = function() {
  console.log('user 이름은 소녀시대입니다.');
};
```

exports 전역 변수에 printUser 속성을 추가했으며 이 속성에 함수를 할당하였습니다. 이 패턴은 앞에서도 만들어 보았으므로 쉽게 이해될 것입니다.

이 모듈 파일을 사용하는 module_test7.js 파일에 다음과 같이 입력합니다.

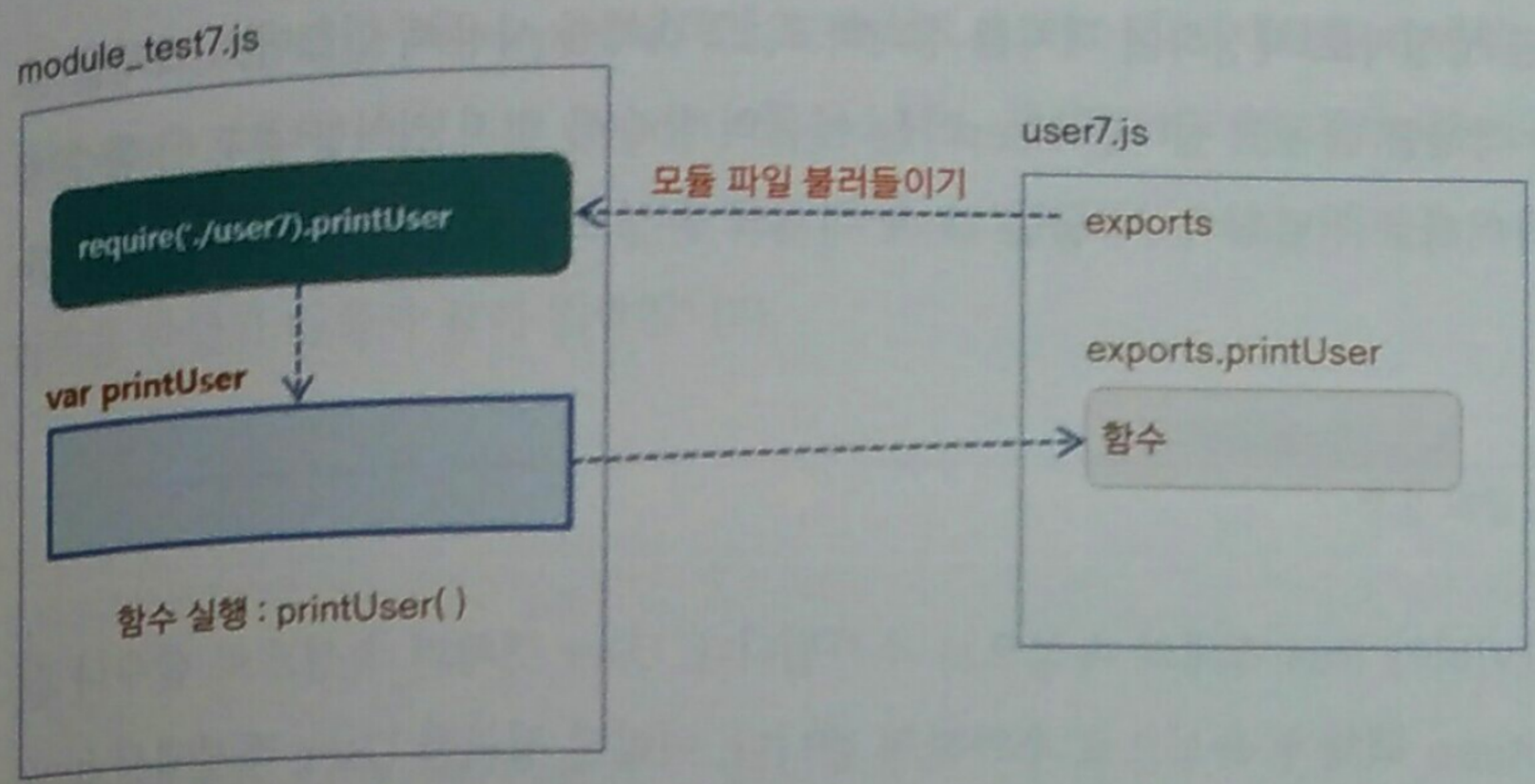
**참조 파일** | ModuleExample>module_test7.js

```
// 사용 패턴: exports에 속성으로 추가된 함수 객체를 그대로 참조한 후 호출함
var printUser = require('./user7').printUser;

printUser();
```

require( ) 메소드로 모듈을 불러들이면 exports 자체가 참조됩니다. exports의 printUser 속성이 함수이므로 모듈을 불러들이면서 그 속성을 바로 참조하면 다음과 같은 코드를 구성할 수 있습니다.

```
require('./user7').printUser;
```

exports가 아닌 exports의 printUser 속성을 변수에 할당했습니다. 여기에서 printUser 속성이 함수이므로 함수를 실행하려면 printUser( )처럼 변수 이름 뒤에 소괄호를 붙여 줍니다.




◀ 모듈 사용 패턴 1: exports 속성으로 함수를 추가하는 패턴

module.exports에는 객체를 직접 할당할 수 있으므로 exports가 아니라 module.exports에 객체를 할당하고 그 안에 다시 함수를 속성으로 추가하는 경우에도 모듈을 불러와 사용하는 쪽의 코드 패턴은 위의 구조와 같습니다.

### 인스턴스 객체를 할당하는 코드 패턴

두 번째 패턴은 모듈을 위해 만든 파일 안에서 인스턴스 객체를 만들어 할당합니다. 먼저 인스턴스 객체를 만들어 module.exports에 직접 할당하는 예를 만들어 보겠습니다. user8.js 파일을 만들고 다음과 같이 입력합니다.

**참조 파일** | ModuleExample>user8.js

```
// 사용 패턴: module.exports에 인스턴스 객체를 만들어 할당함

// 생성자 함수
function User(id, name) {
  this.id = id;
  this.name = name;
}

User.prototype.getUser = function() {
  return {id:this.id, name : this.name};
}

User.prototype.group = {id : 'group1', name : '친구'};

User.prototype.printUser = function() {
  console.log('user 이름 : %s, group 이름 : %s', this.name, this.group.name);
}
module.exports = new User('test01', '소녀시대');
```

자바스크립트에서는 함수를 생성자로 지정하여 객체를 정의하고, 그 객체를 사용해 인스턴스 객체를 만들 수 있습니다. 만약 User 객체를 만들고 싶다면 User라는 이름의 함수를 먼저 정의합니다. 이 함수에는 id와 name 속성이 파라미터로 전달되며, 이 둘을 User 객체의 속성으로 만들기 위해 다음 코드를 사용합니다.

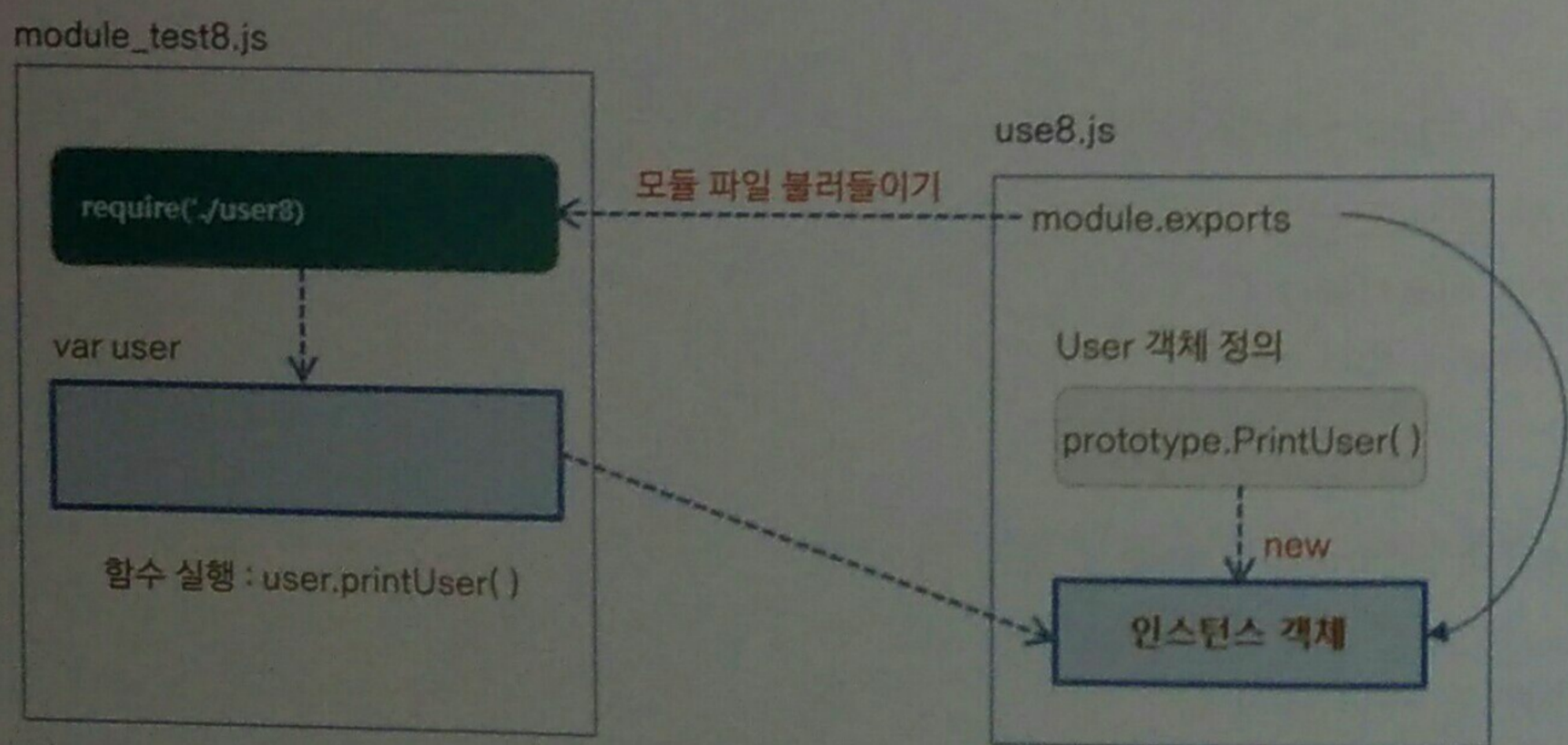
```
this.id = id;
```

이렇게 하면 전달받은 파라미터가 this 객체의 속성으로 추가됩니다. User 객체의 속성으로 함수나 값을 추가하려면 User.prototype 객체에 속성으로 추가하면 됩니다. 이렇게 정의한 User 객체에서 new 연산자를 사용하여 새로운 인스턴스를 객체를 만든 후 module.exports에 직접 할당합니다. 이렇게 하면 모듈을 불러오는 쪽에서 인스턴스 객체를 바로 참조할 수 있습니다. 다음과 같이 module_test8.js 파일을 만듭니다.

**참조 파일** | ModuleExample>module_test8.js

```
// 사용 패턴: new 연산자로 만든 인스턴스 객체를 module.exports에 할당한 후 그 인스턴스 객체의
함수를 호출함

var user = require('./user8');

user.printUser();
```

require( ) 메소드를 사용하여 user8.js 파일에 정의된 모듈을 불러오면 new User( )로 만든 인스턴스 객체가 반환됩니다. User의 인스턴스 객체에는 printUser( ) 메소드가 추가되어 있으므로 user.printUser( )와 같은 형태로 메소드를 호출할 수 있습니다.




▲모듈 사용 패턴 2: module.exports에 인스턴스 객체를 할당하는 패턴

그런데 모듈을 만들 때 exports 객체의 속성으로 인스턴스 객체를 추가했다면 이 객체를 어떻게 사용할까요? 모듈의 속성으로 exports에 인스턴스 객체를 추가한 경우에는 모듈을 불러들이는 쪽에서 require( ) 메소드를 호출한 후 반환되는 객체 속성으로 인스턴스 객체를 참조할 수 있습니다. user9.js 파일을 만들고 다음과 같이 입력합니다.

**참조 파일** | ModuleExample>user9.js

```
.....

exports.user = new User('test01', '소녀시대');
```

User 객체를 정의하는 코드는 user8.js 파일과 똑같습니다. 다만 마지막에 있는 module.exports에 인스턴스 객체를 직접 할당했던 방식을 바꾸어 여기에서는 exports.user 속성으로 인스턴스 객체를 추가합니다. module_test9.js 파일을 만들고 다음과 같이 코드를 입력합니다.

**참조 파일** | ModuleExample>module_test9.js

```
// 사용 패턴: exports의 속성 이름을 추가하되 인스턴스 객체를 만들어 할당함

var user = require('./user9').user;

user.printUser();
```

require( ) 메소드를 사용해서 user9.js에 정의한 모듈을 불러들이고, 곧바로 반환된 객체의 user 속성을 참조합니다. user 속성은 User에서 만든 인스턴스 객체를 의미하므로 user.printUser( )와 같은 코드를 사용하면 User 안에 정의된 함수를 호출할 수 있습니다.

### 프로토타입 객체를 할당하는 코드 패턴

세 번째 패턴은 User 객체를 새로 정의한 후 module.exports에 직접 할당합니다. 다시 말해, 앞에서 만든 User 객체로 인스턴스 객체를 만든 후 module.exports에 할당하는 것이 아니라 User 객체 자체를 module.export에 할당합니다. 이렇게 하면 User 객체를 정의하는 부분만 별도의 모듈 파일로 분리할 수 있으니 다른 파일에서 필요할 때마다 직접 인스턴스 객체를 만들어 사용할 수 있다는 장점이 있습니다. user10.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | ModuleExample>user10.js

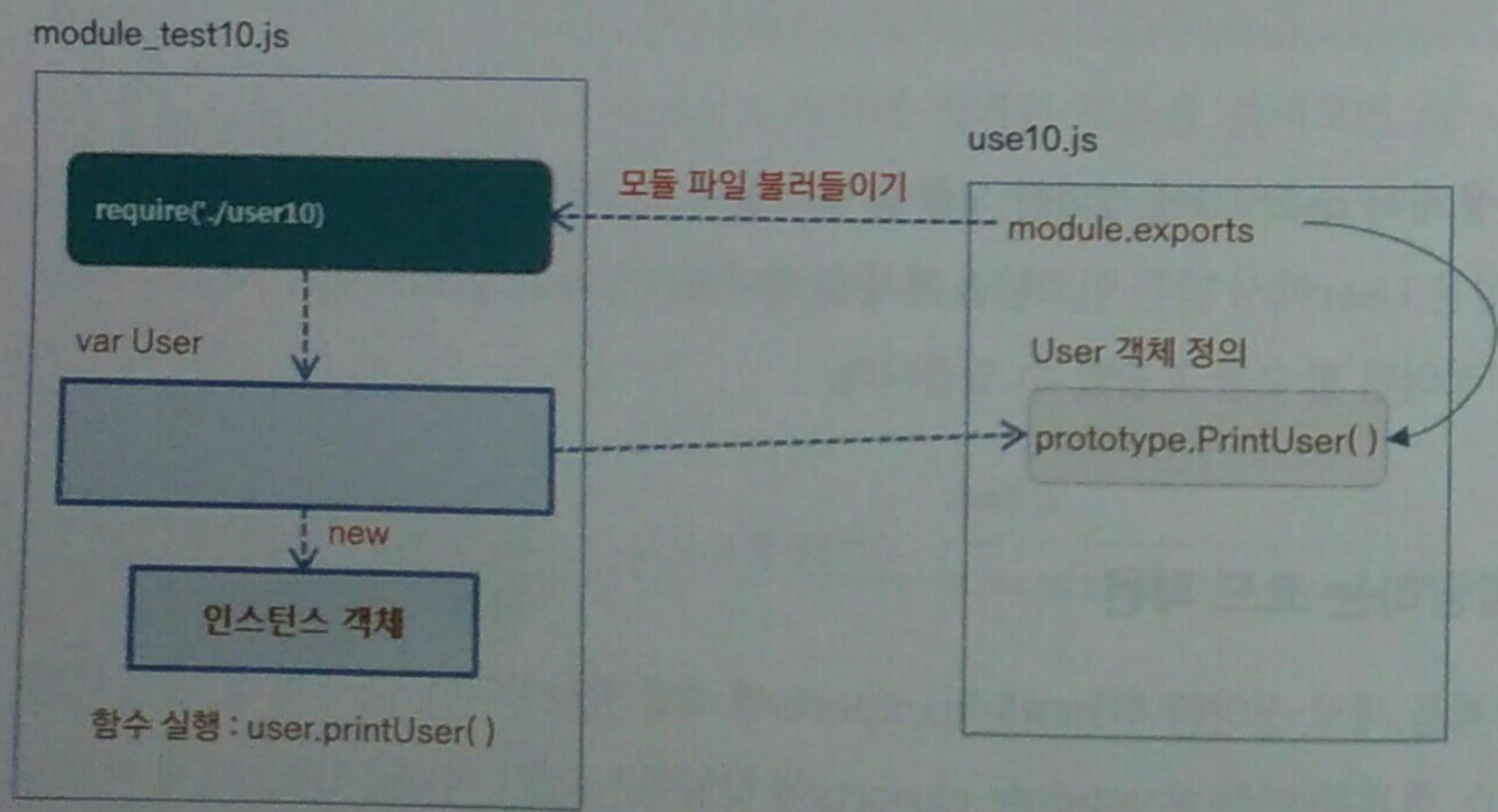
```
.....

module.exports = User;
```

모듈 파일의 나머지 부분은 user9.js 파일과 같으며, 마지막 부분에서만 module.exports에 User 객체를 직접 할당합니다. module_test10.js 파일의 코드는 다음과 같습니다.

참조 파일 | ModuleExample>module_test10.js

```
// 사용 패턴: module.exports에 프로토타입 객체를 정의한 후 할당함

var User = require('./user10');
var user = new User('test01', '소녀시대');

user.printUser();
```

require( ) 메소드를 호출하여 모듈을 불러들이면 User 객체가 반환됩니다. 프로토타입 객체로 만든 User 객체를 참조했으므로 new 연산자를 사용해 인스턴스 객체를 만든 후 그 안에 정의된 함수를 호출할 수 있습니다.




▲ 모듈 사용 패턴 3: module.exports에 프로토타입 객체를 할당하는 패턴

이 코드 패턴도 모듈 파일 안에서 module.exports에 객체를 할당하는 방식 대신에 exports 객체에 속성을 추가한 후 프로토타입 객체를 할당하는 방식을 사용할 수 있습니다. user11.js 파일을 만들고 다음과 같이 exports에 User 속성을 추가한 후 프로토타입 객체를 지정하는 코드를 입력합니다.

참조 파일 | ModuleExample>user11.js

```
……,

exports.User = User;
```

이렇게 모듈을 정의했다면 module_test11.js 파일에는 다음과 같이 입력합니다.

참조 파일 | ModuleExample>module_test11.js

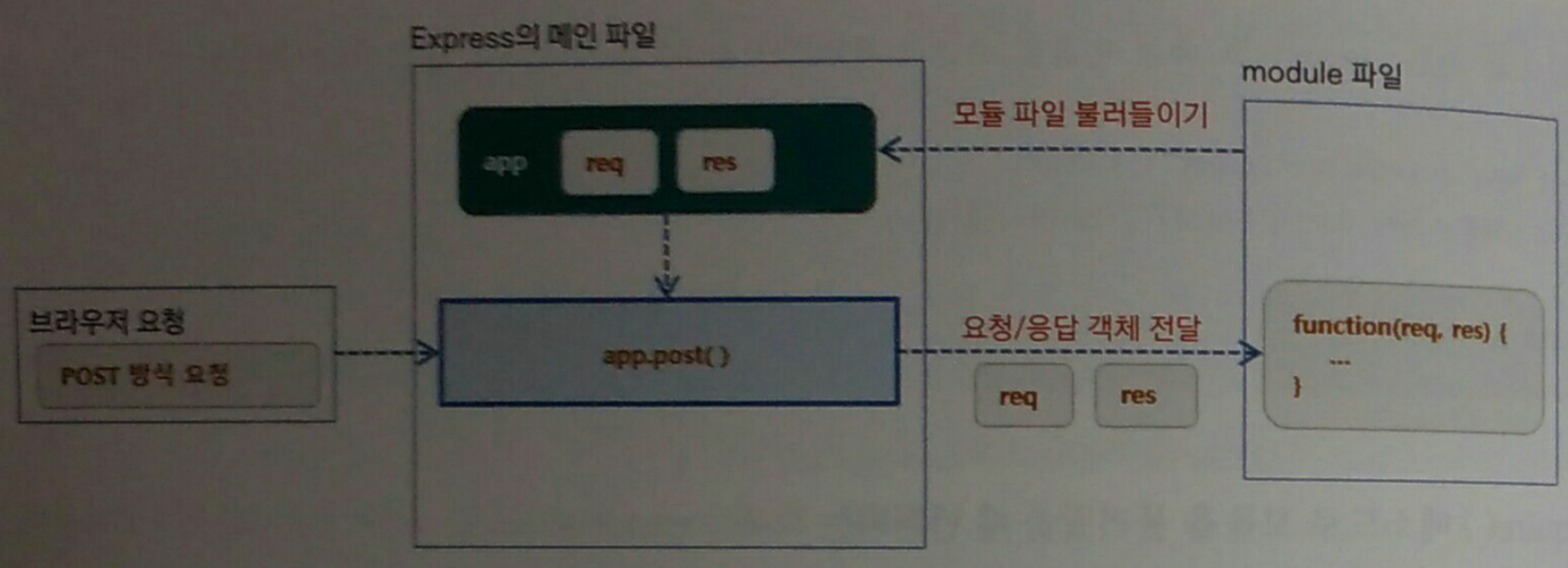
```
// 사용 패턴: exports의 속성 이름을 주면서 추가하되 프로토타입 객체를 정의한 후 할당함

var User = require('./user11').User;
var user = new User('test01', '소녀시대');

user.printUser();
```

require( ) 메소드로 모듈을 불러왔을 때 반환되는 객체는 exports이므로 그 안에 추가한 User 객체를 참조한 후 new 연산자로 인스턴스 객체를 만들 수 있습니다.

지금까지 모듈을 정의하는 방법과 대표적인 코드 사용 패턴 몇 가지를 알아보았습니다. 코드 패턴이 다양해 보일 수 있지만 어떤 경우이든 별도의 모듈 파일에서 module.exports에 객체를 할당하면 된다고 생각하면 이해하기 쉽습니다. 그리고 module.exports에 할당한 객체나 exports에 속성으로 추가한 객체는 모듈을 불러왔을 때 그대로 참조할 수 있다는 것만 알아두면 됩니다. 그러면 다른 사람이 만든 모듈이더라도 쉽게 이해하고 사용할 수 있습니다.

# 07-2 | 사용자 정보 관련 기능을 모듈화하기

앞 장에서 만든 app5.js 파일에는 사용자 등록, 사용자 로그인, 사용자 리스트 조회 등의 기능이 들어 있습니다. 이제 이 기능을 변경하여 별도의 파일로 분리해 보겠습니다. 여러분은 이미 익스프레스로 웹 서버를 만들면 라우팅 미들웨어를 사용할 수 있다는 것과 이 미들웨어는 각각의 기능을 별도의 모듈 파일로 분리할 수 있다는 것을 알고 있습니다. 여기에서 사용할 라우팅 미들웨어는 등록된 모듈 쪽으로 요청 객체와 응답 객체를 전달합니다.




▲ 라우팅 미들웨어에서 모듈로 전달하는 요청 객체와 응답 객체

그렇다면 왜 요청 객체와 응답 객체를 전달하는 것일까요? 웹 서버는 클라이언트의 요청에 응답하는 것이 가장 중요한 작업 중 하나이므로, 라우팅 미들웨어에서 모듈 쪽으로 요청 객체와 응답 객체를 전달할 수 있다면 많은 작업을 분리하여 처리할 수 있기 때문입니다.

그런데 라우팅 미들웨어에서 사용하는 모듈들은 대부분 데이터베이스 기능을 사용합니다. 따라서 데이터베이스 기능도 각 기능을 독립적으로 분리하는 것이 좋습니다. 그러면 우선 사용자 기능을 분리하기 전에 데이터베이스 관련 기능 중에서 스키마를 정의하는 부분을 분리해 보겠습니다.

### 스키마 파일을 별도의 모듈 파일로 분리하기

스키마는 컬렉션의 구조를 결정하는 것으로 데이터베이스와 분리해서 독립적으로 정의할 수 있습니다. 먼저 DatabaseExample 프로젝트에 들어 있는 app5.js 파일의 내용을 복사하여 ModuleExample 프로젝트 안에 app.js 파일을 만듭니다.

ModuleExample 프로젝트 안에 [database] 폴더를 만든 후 그 폴더 안에 user_schema.js 파일을 만듭니다. 새로 만든 파일 안에는 UserSchema 객체를 만들 때 사용한 코드를 넣어 줍니다.

DatabaseExample 프로젝트의 app5.js 파일에는 이미 UserSchema 객체를 만드는 코드가 들어 있으니 해당 부분을 복사하여 넣으면 됩니다.

user_schema.js 파일의 코드는 다음과 같습니다.

**참조 파일** | ModuleExample>/database/user_schema.js

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

var Schema = { };

Schema.createSchema = function(mongoose) {

  // 스키마 정의
  var 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}
  });

.......

  console.log('UserSchema 정의함.');

  return UserSchema;
};

// module.exports에 UserSchema 객체 직접 할당
module.exports = Schema;
```

스키마를 만들려면 mongoose 모듈 객체를 참조해야 합니다. 이 모듈 객체는 app.js 파일에서 불러들인 후 변수에 할당한 것이므로 user_schema.js 파일에서는 이 객체를 파라미터로 전달받아야 사용할 수 있습니다. 물론 이 user_schema.js 파일 안에서 require( ) 메소드로 mongoose 모듈을 직접 불러올 수도 있습니다. 하지만 app.js 파일에서 mongoose 모듈에 설정한 정보를 똑같이 사용하려면 파라미터로 전달받아야 합니다. 이 때문에 createSchema라는 함수를 새로 만들고 mongoose 객체를 파라미터로 전달받게 합니다.

module.exports에 객체를 할당하는 방식을 사용하기 위해 먼저 Schema 객체를 정의한 후 그 안에 createSchema 속성을 추가합니다. 이렇게 하면 이 모듈을 불러오는 쪽에서 createSchema( ) 함수를 호출하면서 mongoose 객체를 파라미터로 전달할 수 있습니다. createSchema( ) 함수 안의 코드는 app.js 파일에서 사용자 스키마를 만들기 위해 사용한 코드와 똑같습니다. 스키마를 만들고 비밀번호를 저장할 때 암호화 모듈을 사용했으므로 user_schema.js 파일 안에서 require( ) 메소드로 crypto 모듈을 불러옵니다. 코드의 마지막 부분에서는 module.exports에 Schema 객체를 할당합니다.

이제 app.js 파일을 열고 데이터베이스 스키마를 만들어야 하는 코드 부분에 createSchema( ) 함수를 호출하도록 다음과 같이 입력합니다.

**참조 파일** | ModuleExample>app.js

```
.....

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

  // user_schema.js 모듈 불러오기
  UserSchema = require('./database/user_schema').createSchema(mongoose);

  // UserModel 모델 정의
  UserModel = mongoose.model("users3", UserSchema);
  console.log('UserModel 정의함.');

.....
```

스키마를 만드는 코드는 createUserSchema( ) 함수 안에 있었습니다. 그런데 이 부분의 코드를 user_schema.js 파일로 분리했으므로 require( ) 메소드를 호출하여 ./database/user_schema.js 파일을 불러오도록 수정합니다.

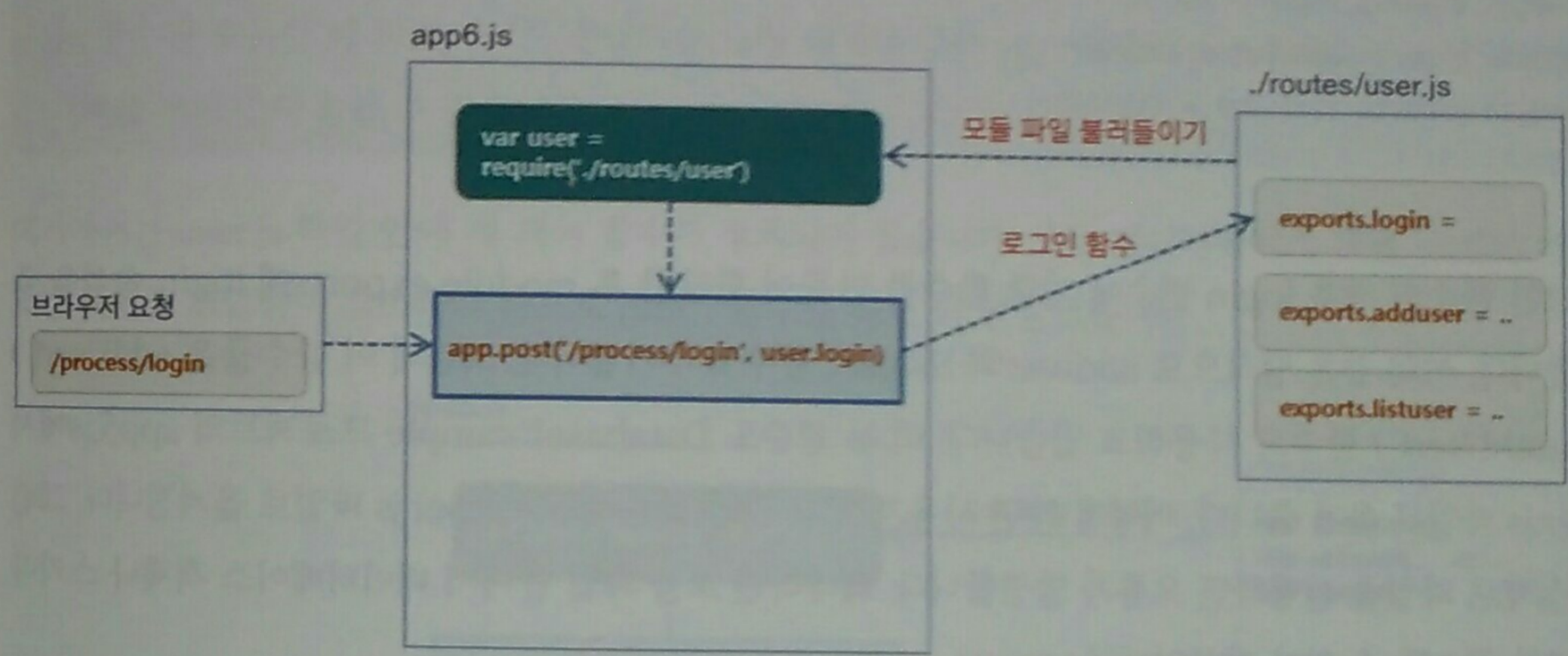
require( ) 메소드를 호출했을 때 반환되는 객체는 모듈 파일에서 module.exports에 할당한 Schema 객체이므로 그 객체 안에 들어 있는 createSchema( ) 함수를 호출할 수 있습니다. createSchema( ) 함수를 호출하면서 mongoose 객체를 파라미터로 전달하면 새로 만들어진 사용자 스키마 객체가 반환됩니다. 그 아랫부분에서는 이 스키마 객체를 사용해 모델 객체를 만들고 UserModel 변수에 할당합니다. 이렇게 수정한 후 app.js 파일을 실행하면 DatabaseExample 프로젝트 안에서 app5.js 파일을 실행했을 때와 똑같이 동작합니다.

## 사용자 처리 함수를 별도의 모듈 파일로 분리해 보기

스키마 파일을 별도 파일로 분리했으니 이제 사용자 처리 관련 함수들을 별도의 모듈 파일로 분리해 보겠습니다. 사용자 관련 함수들은 익스프레스의 라우팅 미들웨어를 사용하지만 모듈로 분리하는 방법은 일반적인 모듈을 만들어 사용할 때와 같습니다. 클라이언트에서 요청이 들어오면 라우팅하는 부분은 다음과 같이 세 가지가 있습니다.

```
app.post('/process/login', function(...) {...});
app.post('/process/adduser', function(...) {...});
app.post('/process/listuser', function(...) {...});
```

app.post( ) 함수의 두 번째 파라미터로 전달되는 것이 함수이므로 이 함수를 별도의 모듈로 분리할 수 있습니다. 사용자 관련 기능을 모두 하나의 모듈 파일에 넣어 주면 더 구분하기 좋으므로 다음과 같이 구성합니다.




▲사용자 관련 기능을 모듈로 분리하는 구성

[routes] 폴더에 들어 있는 user.js 파일을 수정하여 세 개의 함수로 만들고 DatabaseExample 프로젝트의 app5.js 파일에 들어 있던 세 개의 함수를 각각 옮겨 넣습니다. 그다음 module.exports에 login, adduser, listuser 속성을 추가합니다. 이렇게 하면 ./routes/user.js 파일을 불러온 후 user.login으로 모듈 파일의 login 함수를 지정할 수 있습니다.

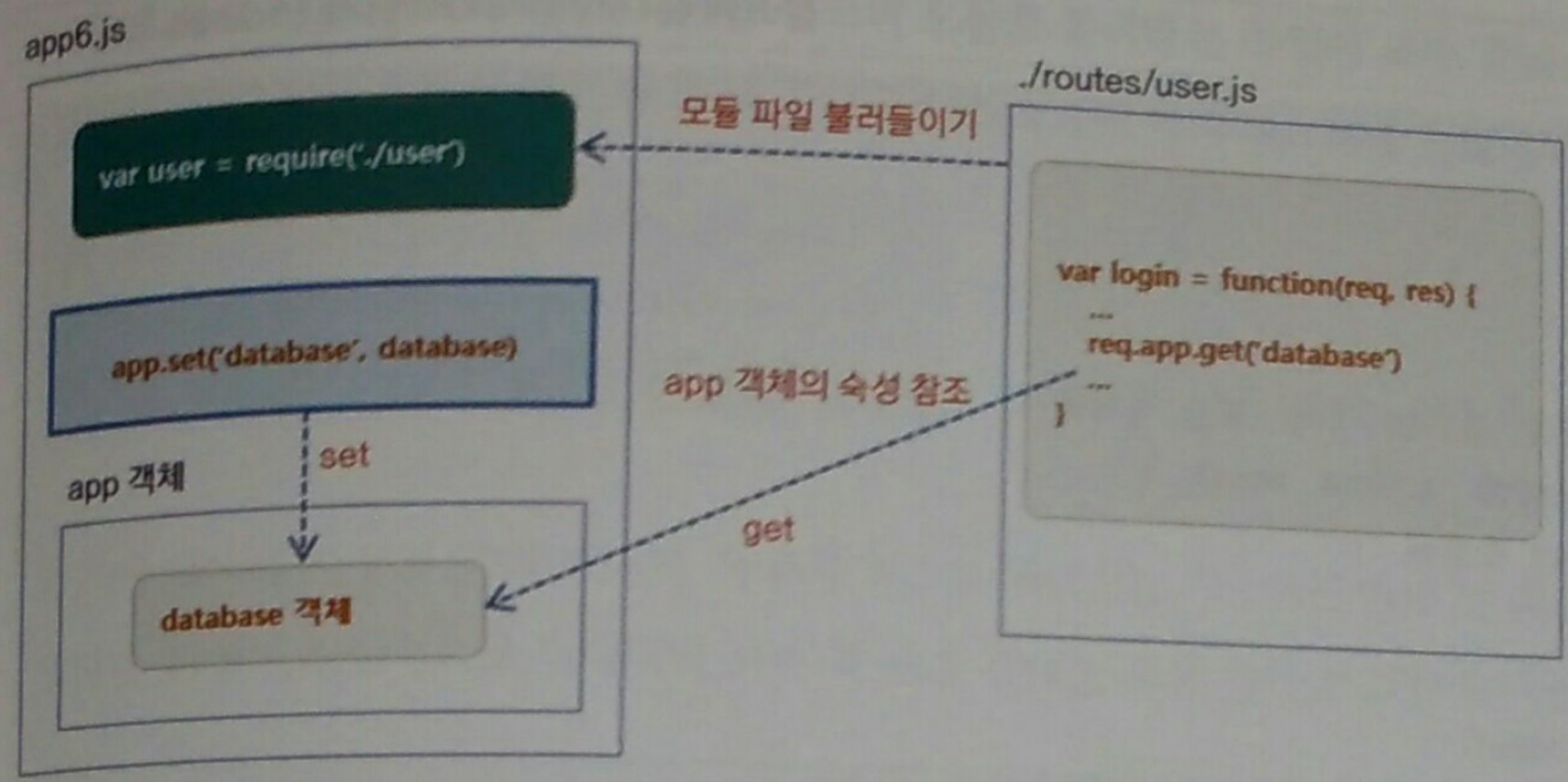
다음은 user.js 파일에 추가된 코드의 주요 부분입니다.

```
……

var login = function(req, res) {
    console.log('user 모듈 안에 있는 login 호출됨.');

……
var adduser = function(req, res) {
    console.log('user 모듈 안에 있는 adduser 호출됨.');

……
var listuser = function(req, res) {
    console.log('user 모듈 안에 있는 listuser 호출됨.');

……

module.exports.login = login;
module.exports.adduser = adduser;
module.exports.listuser = listuser;
```

로그인 함수의 경우 login 변수에 익명 함수를 만들어 할당한 후 module.exports에 login 속성을 추가합니다. 이와 같은 방식으로 adduser와 listuser 함수도 추가합니다. 그런데 이 함수들은 addUser( )와 authUser( ) 함수도 사용하고 있습니다. 이 두 함수도 DatabaseExample 프로젝트의 app5.js에서 user.js 파일로 옮겨 줍니다. 이렇게 하면 사용자 처리 관련 코드는 모두 user.js 파일로 옮겨집니다. 그런데 실제로 파일을 실행하면 오류가 발생합니다. 왜냐하면 모듈 파일 안에서 데이터베이스 객체나 스키마 객체를 참조할 수 없기 때문입니다.

이처럼 모듈 파일 안에서 이 모듈을 불러들여 사용하는 쪽의 변수를 참조해야 할 때가 많습니다. 그중에서도 데이터베이스 관련 객체(데이터베이스 객체, 스키마 객체, 모델 객체)를 별도의 다른 모듈에서 만들거나 app.js 같은 메인 파일에서 만들 때 모듈 파일 쪽으로 이 객체들을 전달할 수 있어야 합니다.

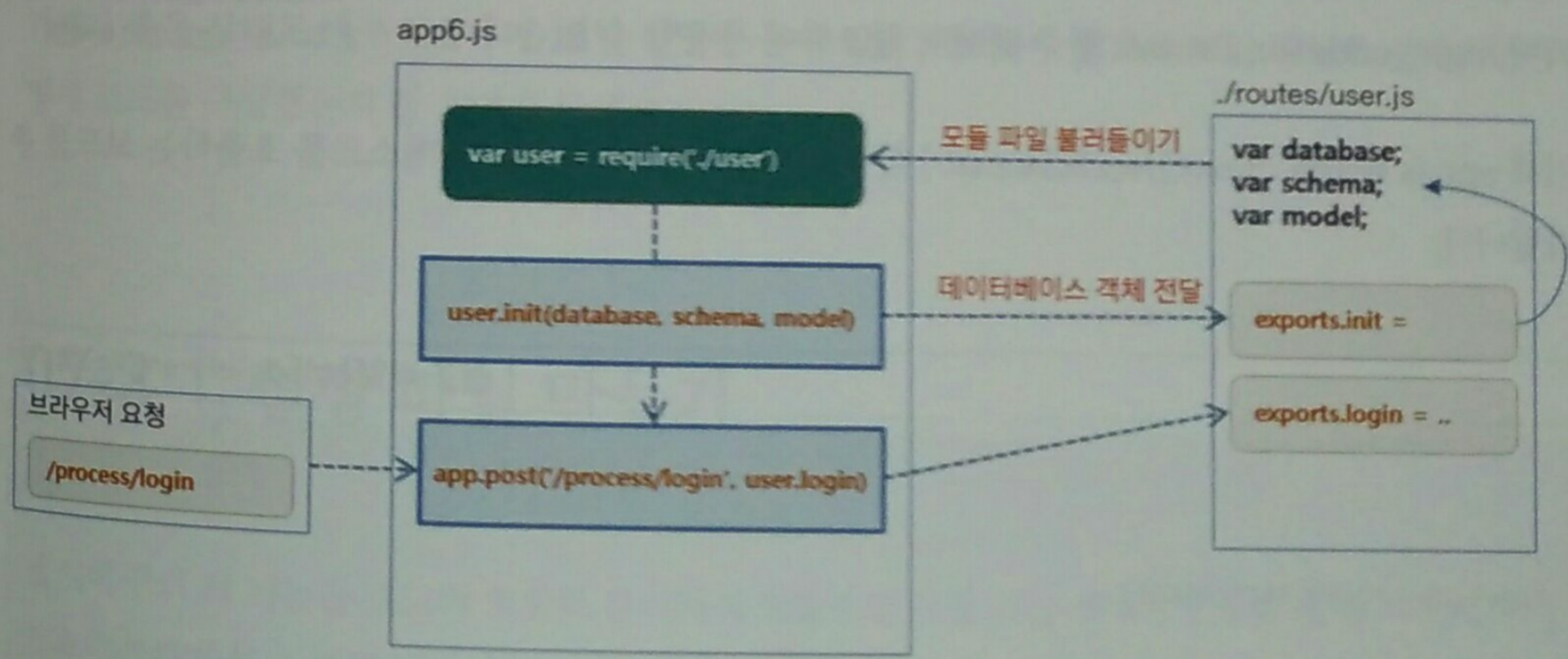
익스프레스는 요청 객체가 app 객체를 속성으로 가지고 있기 때문에 메인 파일에서 모듈 파일로 객체를 전달할 때 app 객체에 속성으로 추가한 후 전달할 수 있습니다. 이때는 app 객체의 set( )과 get( ) 메소드를 사용하는 것이 좋습니다.




▲app 객체의 set( )과 get( ) 메소드를 사용해 메인 파일에서 모듈 파일로 객체 전달하기

메인 파일에서는 데이터베이스 객체를 사용할 수 있는 시점이 되면 app 객체의 set( ) 메소드를 호출하여 데이터베이스 객체를 database 속성으로 추가합니다. 이렇게 하면 모듈 파일인 user.js 안에 정의한 login 함수를 호출할 때 파라미터로 전달되는 요청 객체에 있는 app 객체의 get( ) 메소드로 데이터베이스 객체를 꺼내어 사용할 수 있습니다.

여기에서는 user.js 파일 안에 세 개의 함수가 정의되어 있습니다. 모두 데이터베이스 객체, 스키마 객체, 모델 객체를 전달받아 참조해야 하므로 init( ) 함수를 하나 더 만들었습니다.




▲모듈 파일에 init( ) 함수를 만들어 데이터베이스 관련 객체를 모듈로 전달하는 방법

데이터베이스 객체들을 사용할 수 있는 시점이 되면 user 모듈의 init( ) 메소드를 호출하면서 데이터베이스 객체, 스키마 객체, 모델 객체를 파라미터로 전달합니다. user.js 파일에 추가된 코드는 다음과 같습니다.

참조 파일 | ModuleExample>/routes/users.js

```
var database;
var UserSchema;
var UserModel;

// 데이터베이스 객체, 스키마 객체, 모델 객체를 이 모듈에서 사용할 수 있도록 전달함
var init = function(db, schema, model) {
  console.log('init 호출됨.');

  database = db;
  UserSchema = schema;
  UserModel = model;
}

.....

module.exports.init = init;
module.exports.login = login;
module.exports.adduser = adduser;
module.exports.listuser = listuser;
```

init( ) 함수를 호출하면서 파라미터로 전달된 객체들은 모두 변수에 할당되며 login, adduser, listuser 함수 안에서 참조하여 사용할 수 있습니다.

**주의!** 코드의 가장 아랫부분에서 module.exports에 init 속성으로 추가하는 것도 잊지 마세요.

이제 app.js 파일의 createUserSchema( ) 함수 안에 user 모듈의 init( ) 메소드를 호출하는 코드를 추가합니다.

참조 파일 | ModuleExample>app.js

```
.....

function createUserSchema() {

.....

  // init 호출
  user.init(database, UserSchema, UserModel);

}

.....
```

이렇게 해서 사용자 처리 관련 기능을 별도의 모듈로 분리하는 작업이 모두 끝났습니다. app.js 파일을 실행한 후 웹 브라우저를 열고 다음 세 곳의 사이트에 접속해서 테스트를 진행합니다.

- http://localhost:3000/public/login.html
- http://localhost:3000/public/adduser.html
- http://localhost:3000/public/listuser.html

로그인 페이지, 사용자 등록 페이지, 사용자 리스트 페이지는 지금까지 여러 번 테스트해 보았던 웹 문서입니다. 파일이 정상적으로 실행되면 서버 쪽 콘솔 창에는 오류 없이 로그만 출력됩니다.

```
Console  Debug
NodeExample3-app6.js [Node Application] Node.js Process
서버가 시작되었습니다. 포트 : 3000
데이터베이스에 연결되었습니다. : mongodb://localhost:27017/shopping
UserSchema 정의함.
users3 정의함.
init 호출됨.
user 모듈 안에 있는 login 호출됨.
authUser 호출됨.
아이디 [test01]로 사용자 검색결과
[ { '$__':
    { strictMode: true,
```

▲app6.js 테스트가 정상 진행되었을 때의 콘솔 로그

실행했을 때의 결과는 같지만 별도의 모듈 파일로 분리했기 때문에 코드를 수정하거나 새로운 기능을 추가하기 좋은 상태로 바뀌었습니다. 메인 파일에 들어 있던 코드들이 별도의 모듈 파일로 분리되면서 어떻게 코드를 구성했는지 잘 기억해 두세요.

## 07-3 | 설정 파일 만들기

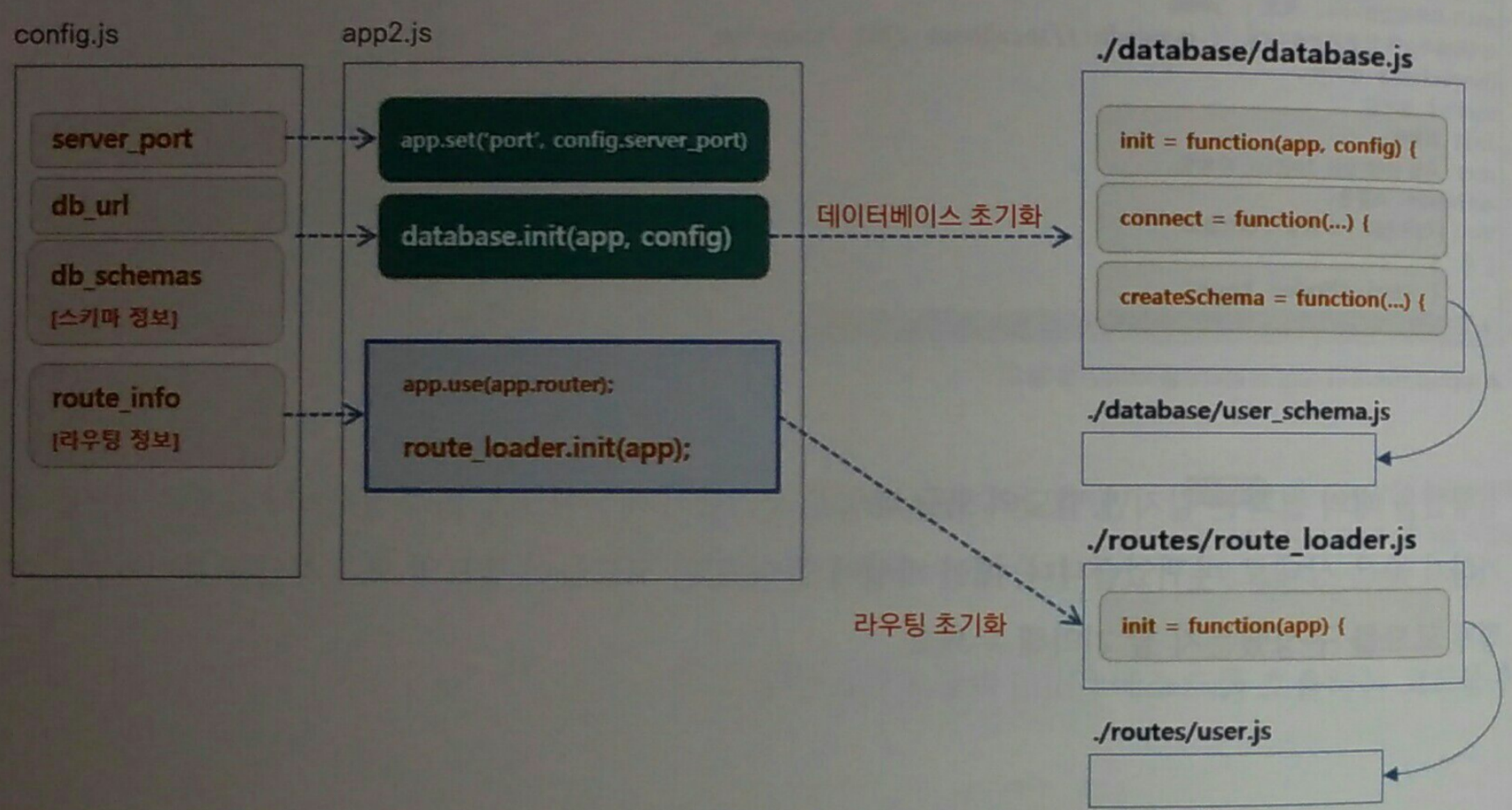
웹 서버 안의 각 기능을 별도의 파일로 분리하여 모듈로 만들면 기능 수정이 필요할 때 웹 서버의 메인 파일을 수정하지 않고 모듈 부분만 수정해도 됩니다. 또 새로운 모듈을 추가할 때도 메인 파일을 수정하지 않아도 되기 때문에 서버의 유지관리에 아주 좋은 구성이 됩니다.

그런데 새로운 모듈을 추가했을 때 메인 파일을 수정하지 않고도 어떤 모듈이 추가되었는지 어떻게 알 수 있을까요? 설정 파일을 만들고 메인 파일이 설정 파일을 불러오도록 하면 됩니다. 왜냐하면 새로운 모듈을 추가했을 때 설정 파일만 수정해도 메인 파일에서 변경된 내용을 알 수 있기 때문입니다. 즉, 새로운 설정 파일의 이름이 config.js라면 이 파일을 모듈 파일로 만들고 이 파일 안에 설정 정보를 넣어 둔 후 서버가 실행될 때 메인 파일에서 이 설정 파일에 있는 설정 정보를 읽어 들이도록 만들면 됩니다.

config.js 파일 안에 들어 있는 설정 정보에는 서버를 실행하는 데 필요한 정보와 데이터베이스 정보 등을 담아 둡니다. 서버 정보는 서버를 실행할 때 필요한 포트 등의 정보를 말하는 것으로 환경 변수로 설정하여 읽어 들일 수 있으며, config.js 파일에 넣어 둘 수도 있습니다. 데이터베이스 정보는 데이터베이스 연결에 필요한 URL이나 인증 정보를 말합니다. mongoose를 사용하는 데이터베이스는 스키마를 생성한 후에 데이터베이스에 접근하므로 스키마 생성 부분을 별도 파일로 분리해 두고 스키마 파일의 로딩 정보를 config.js 파일에 넣어 둘 수 있습니다.

## 설정 파일 분리하기

이제 설정 정보를 이용하여 최종 모듈 구조를 만들어 볼 차례입니다.




▲ 설정 정보 이용하여 최종 모듈 구조 만들기

config.js 파일 안에는 서버를 실행할 때 필요한 포트 정보와 데이터베이스 연결에 필요한 데이터베이스 URL 정보를 넣어 둡니다.

데이터베이스 연결에 필요한 데이터베이스 스키마 정보는 db_schemas 배열 객체에 넣어 둡니다. 그리고 사용자가 요청한 패스 정보로 익스프레스에서 라우팅하던 정보는 route_info 배열 객체에 넣어 둡니다. app.js 파일을 복사하여 app2.js 파일로 만든 후 위와 같이 넣어 둔 설정 정보를 app2.js 파일에서 불러들여 보겠습니다.

먼저 프로젝트 안에 config.js 파일을 만들고 포트 정보와 데이터베이스 URL 정보를 코드로 입력합니다.

참조 파일 | ModuleExample>/config.js

```
……,

module.exports = {

  server_port : 3000,
  db_url : 'mongodb://localhost:27017/shopping',

……,

}
```

module.exports에는 중괄호로 자바스크립트 객체를 만들어 바로 할당하는데, 그 안에 server_port와 db_url 속성이 추가되어 있습니다. 서버 포트는 3000번으로 지정되어 있으며, 데이터베이스 URL은 이전에 사용했던 URL과 똑같은 mongodb://localhost:27017/shopping으로 되어 있습니다. 이렇게 만든 config.js 파일은 메인 파일인 app2.js 파일에서 불러와 사용합니다. 서버 포트 정보를 지정하는 코드는 다음과 같습니다.

참조 파일 | ModuleExample>/app2.js

```
……,

var config = require('./config');

……,

//===== 서버 변수 설정 및 static으로 [public] 폴더 설정 =====//
console.log('config.server_port : %d', config.server_port);
app.set('port', process.env.PORT || 3000);

……,
```

require( ) 메소드를 호출하여 config.js 모듈 파일을 불러온 후 서버 포트 정보를 사용하기 위해 config.server_port 속성을 참조합니다.

### 설정 파일에 데이터베이스 스키마 정보 넣기

이제 데이터베이스 스키마 정보를 설정하여 사용하는 방법을 알아보겠습니다. 먼저 config.js 파일 안의 db_schemas 배열 객체에 들어가는 정보는 다음과 같습니다.

참조 파일 | ModuleExample>/config.js

```
……

module.exports = {

……

  db_schemas: [
    {file:'./user_schema', collection:'users3', schemaName:'UserSchema',
     modelName:'UserModel'}
  ],

……

}
```

db_schemas 속성의 값은 배열이며 배열의 각 원소는 스키마 파일의 정보가 됩니다. 각 속성의 이름과 설명은 다음과 같습니다.

| 속성 이름 | 설명 |
|---|---|
| file | 스키마 파일을 지정합니다. |
| collection | 데이터베이스의 컬렉션 이름을 지정합니다. |
| schemaName | 스키마 파일을 불러들인 후 반환된 객체를 어떤 속성 이름으로 할 것인지 지정합니다. |
| modelName | 스키마에서 모델 객체를 만든 후 어떤 속성 이름으로 할 것인지 지정합니다. |

[표] 스키마 파일을 정의할 때 사용하는 속성들

데이터베이스에 들어가는 스키마 정보를 별도 파일에 넣어 두었으니 그 파일에서 사용자 관련 스키마를 로딩해야 합니다. 그런데 새로운 스키마를 추가하면 로딩 작업을 위한 코드도 변경해야 하므로 로딩 작업에 필요한 파일을 또다시 분리시키는 것이 좋습니다. [database] 폴더 안에 database.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | ModuleExample>/database/database.js

```
var mongodb = require('mongodb');
var mongoose = require('mongoose');

// database 객체에 db, schema, model 모두 추가
var database = {};
```

```
database.init = function(app, config) {
  console.log('init() 호출됨.');

  connect(app, config);
}

// 데이터베이스에 연결하고 응답 객체의 속성으로 db 객체 추가
function connect(app, config) {
  console.log('connect() 호출됨.');

  ......

}

// config에 정의한 스키마 및 모델 객체 생성
function createSchema(app, config) {
  var schemaLen = config.db_schemas.length;
  console.log('설정에 정의된 스키마의 수 : %d', schemaLen);

  for (var i = 0; i < schemaLen; i++) {
    var curItem = config.db_schemas[i];

    // 모듈 파일에서 모듈 불러온 후 createSchema( ) 함수 호출하기
    var curSchema = require(curItem.file).createSchema(mongoose);
    console.log('%s 모듈을 불러들인 후 스키마 정의함.', curItem.file);

    // User 모델 정의
    var curModel = mongoose.model(curItem.collection, curSchema);
    console.log('%s 컬렉션을 위해 모델 정의함.', curItem.collection);

    // database 객체에 속성으로 추가
    database[curItem.schemaName] = curSchema;
    database[curItem.modelName] = curModel;
    console.log('스키마 이름 [%s], 모델 이름 [%s]이 database 객체의 속성으로 추가됨.',
                curItem.schemaName, curItem.modelName);
  }

  app.set('database', database);
  console.log('database 객체가 app 객체의 속성으로 추가됨.');
}

// database 객체를 module.exports에 할당
module.exports = database;
```

database.js 파일 안에서는 자바스크립트 객체를 하나 만들어 database 변수에 할당합니다. 이 객체는 다시 module.exports에 지정되어 이 모듈 파일을 불러들이는 쪽에서 사용할 수 있습니다. 이 객체에 추가되어 있는 메소드는 init( ), connect( ), createSchema( ) 세 개입니다.

init( ) 메소드는 단순히 connect( ) 메소드를 호출하는데 connect( ) 메소드 안에 들어가는 코드는 앞에서 데이터베이스에 연결하는 데 사용한 코드입니다. 따라서 init( ) 메소드를 호출하면 데이터베이스에 연결됩니다. createSchema( ) 메소드는 config.js 파일의 db_schemas 속성에 들어 있는 스키마 정보로 스키마를 만듭니다. 다만 이 코드 부분에서 직접 스키마를 만드는 것은 아니며 각 용도별로 데이터베이스 스키마를 정의한 모듈 파일을 다시 로딩합니다. 예를 들어, 사용자 정보를 저장하기 위해 정의한 스키마는 [database] 폴더의 user_schema.js 파일에서 만드는데 이 database.js 파일에서는 user_schema.js 파일을 로딩하여 실행하는 역할만 합니다.

코드를 좀 더 자세히 살펴보겠습니다. config.js 파일에 넣어 둔 스키마 파일 데이터를 사용해 각각의 스키마를 담고 있는 모듈 파일을 로딩한 후 스키마와 모델 객체를 database 객체의 속성으로 추가합니다. 이렇게 하면 config.js 파일에서 지정한 스키마 이름과 모델 이름이 database 객체의 속성 이름이 됩니다. database 객체에 속성들을 모두 추가했다면 app 객체의 set( ) 메소드를 호출하여 database라는 속성 이름으로 추가합니다. 이렇게 하면 사용자가 요청했을 때 이 함수로 라우팅되어 실행되며 req.app.get('database') 코드로 database 객체를 참조할 수 있습니다.

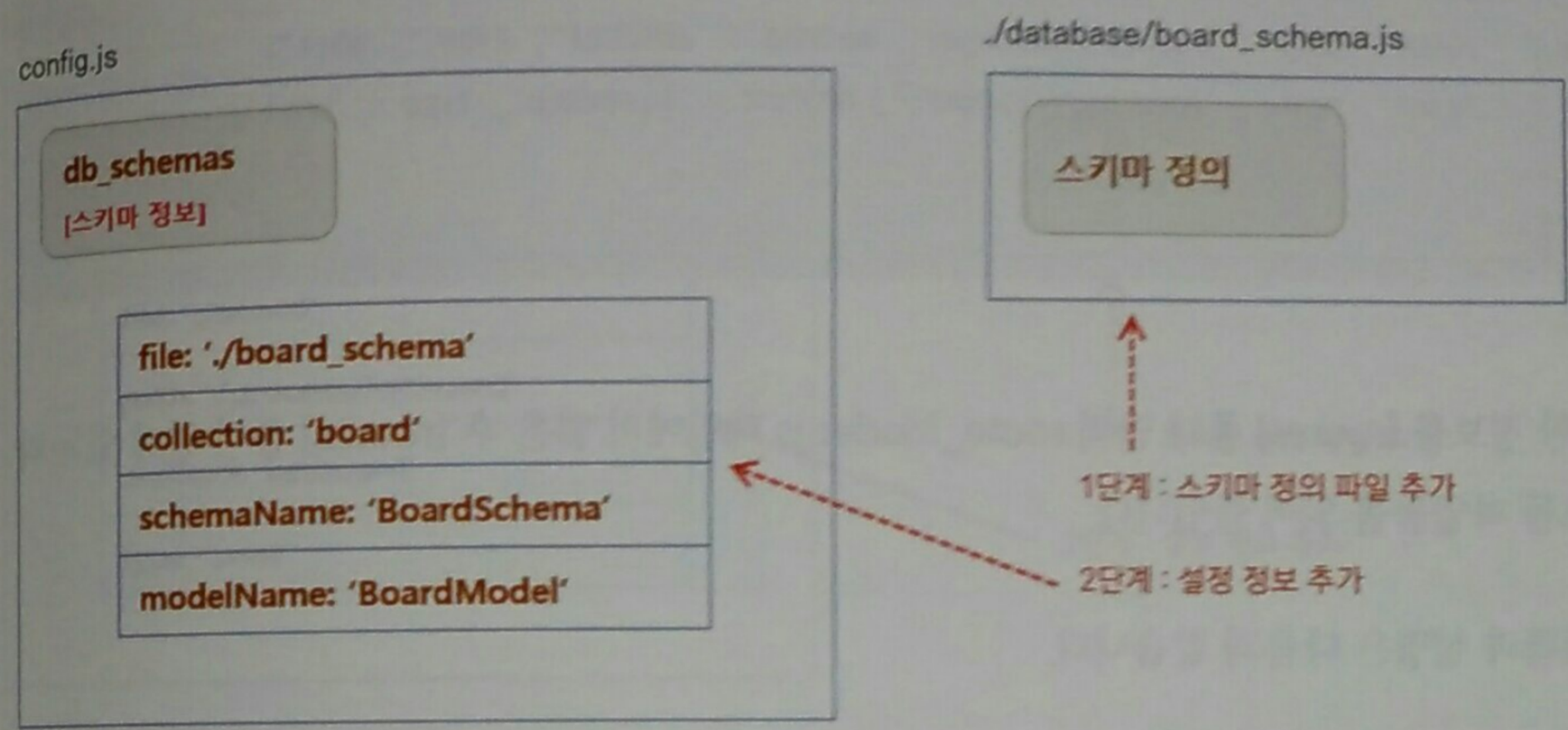
user_schema.js 파일의 내용은 앞에서 사용자 정보를 저장하려고 만든 스키마와 같습니다. 이제 user_schema.js 파일에 넣어 준 코드를 살펴보겠습니다.

**참조 파일** | ModuleExample>/database/user_schema.js

```
……

var crypto = require('crypto');

var Schema = { };

Schema.createSchema = function(mongoose) {

……

  console.log('UserSchema 정의함.');

  return UserSchema;
};

// module.exports에 UserSchema 객체 직접 할당
module.exports = Schema;
```

user_schema.js 파일 안에서는 Schema 자바스크립트 객체를 만들고 createSchema라는 속성 이름으로 함수를 추가합니다. 함수의 마지막 부분에서는 새로 만들어진 스키마 객체를 반환합니다.

이제 데이터베이스와 관련된 기능은 모두 메인 파일에서 분리하였으므로 하나의 파일로 되어 있던 코드가 설정 파일인 config.js 파일과 [database] 폴더 안에 들어 있는 파일들로 나뉩니다. 이제부터는 데이터베이스 관련 기능을 추가하고 싶을 때 [database] 폴더 안에 필요한 파일을 만든 후 config.js 파일의 db_schemas 속성에 정보를 추가하면 됩니다.




▲ 데이터베이스 스키마를 새로 정의하는 작업 과정

데이터베이스 스키마를 새로 정의하려면 다음 두 단계를 거칩니다.

1단계 스키마 정의 파일을 추가합니다. 만약 게시판을 추가하고 싶다면 먼저 게시판 스키마를 board_schema.js 파일에 정의합니다.

2단계 새로 만든 스키마 정보를 config.js 설정 파일 안에 추가합니다. 1단계에서 게시판 스키마를 만들었다면 이 스키마 정보를 config.js 파일 안에 들어 있는 db_schemas 배열 객체 안에 추가합니다. 이렇게 하면 해당 정보가 추가되어 어떤 스키마를 만들었는지 알 수 있습니다.

## 설정 파일에 라우팅 정보 넣기

데이터베이스 기능을 별도의 파일로 분리하고 해당 정보를 config.js 파일에 추가하듯이 라우팅 함수도 모듈로 분리한 후 설정 정보에 추가할 수 있습니다. 사용자를 조회하거나 추가하는 기능을 라우팅하는 코드는 원래 메인 파일에 들어 있었지만 이제 이 코드는 [routes] 폴더 안에 만든 user.js 파일에 들어 있습니다. 라우팅 정보를 config.js 파일에 등록해 두면 별도의 모듈 파일에 분리되어 있는 사용자 기능들을 사용자 요청으로 라우팅하여 사용할 수 있습니다.

참조 파일 | ModuleExample>/config.js

```
var user = require('./routes/user2');

module.exports = {

……

  route_info : [
    {file : './user', path : '/process/login', method : 'login', type : 'post'}
   , {file : './user', path : '/process/adduser', method : 'adduser', type : 'post'}
   , {file : './user', path : '/process/listuser', method : 'listuser', type : 'post'}
  ]
}
```

이렇게 설정한 정보를 [routes] 폴더 안의 route_loader.js 파일에서 읽은 후 [routes] 폴더 안에 있는 라우팅 관련 모듈 파일들을 읽어 들입니다.

각 속성의 이름과 설명은 다음과 같습니다.

| 속성 이름 | 설명 |
|---|---|
| file | 라우팅 파일을 지정합니다. |
| path | 클라이언트로부터 받은 요청 패스를 지정합니다. |
| method | 라우팅 파일 안에 만들어 놓은 객체의 함수 이름을 지정합니다. |
| type | get이나 post와 같은 요청 방식을 지정합니다. |

[표] 라우팅 함수 정의 속성들

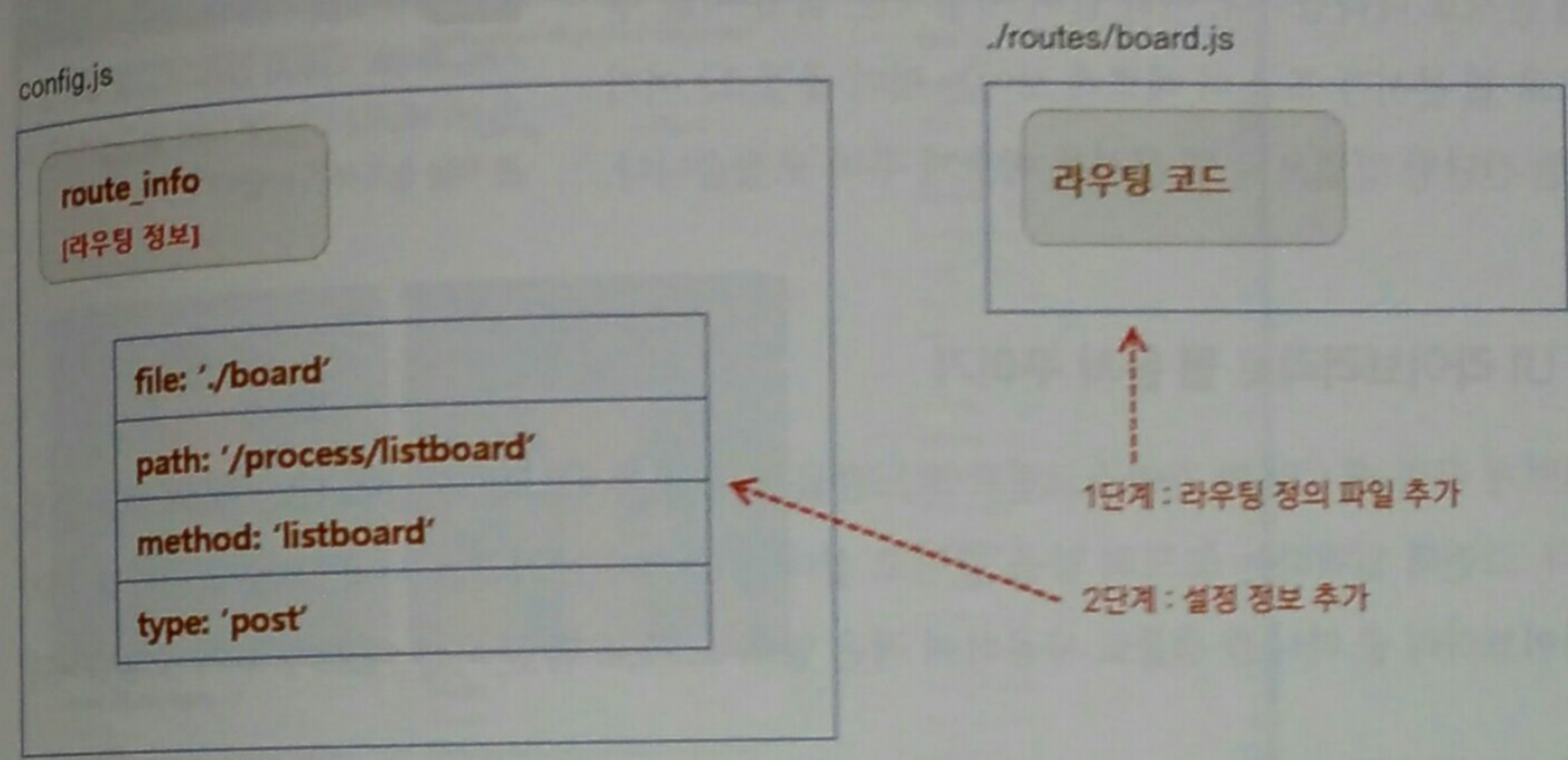
[routes] 폴더의 route_loader.js 파일은 어떤 라우팅 모듈들이 있는지 확인한 후 해당 모듈 파일들을 읽어서 실행합니다. 따라서 메인 파일 app2.js에서는 단순히 route_loader.js 모듈에 정의된 init( ) 함수를 호출하기만 하면 됩니다.

참조 파일 | ModuleExample>app2.js

```
……

var route_loader = require('./routes/route_loader');

……
```

```
// 라우팅 정보를 읽어 들여 라우팅 설정
route_loader.init(app);

.....
```

이제 app2.js 파일에는 사용자의 요청을 처리하는 코드가 하나도 남아 있지 않습니다.




▲라우팅 함수를 새로 정의하는 작업 과정

라우팅 함수를 새로 추가할 때는 다음 두 단계를 거칩니다.

1단계 라우팅 코드가 들어 있는 모듈 파일을 추가합니다. 만약 게시판 기능을 추가하고 싶다면 [routes] 폴더 안에 board.js 파일을 추가하고 그 안에 게시판 기능을 만들어 줍니다.

2단계 새로 만든 라우팅 모듈의 정보를 config.js 설정 파일 안에 추가합니다. route_info 배열 객체 안에 정보를 추가하면 됩니다.

app2.js 파일을 실행한 후 웹 브라우저에서 로그인, 사용자 추가, 사용자 리스트 기능을 다시 테스트합니다. config.js 파일로 설정 정보를 잘 분리했다면 실행 결과는 이전 프로젝트를 실행했을 때와 똑같이 나타납니다.

이렇게 해서 새로운 기능을 추가할 때 메인 파일을 건드리지 않아도 되는 구조를 만들었습니다.

앞으로는 이렇게 만든 모듈 구조를 사용해 볼 것입니다.

# 07-4 | UI 라이브러리로 웹 문서 예쁘게 꾸미기

지금까지 만든 프로젝트의 app2.js 파일을 노드로 실행하면 웹 서버가 실행됩니다. 이제 사용자는 필요한 정보를 요청해서 받아볼 수 있습니다. 사용자가 웹 브라우저를 열고 웹 서버로 데이터 조회를 요청하고 싶다면 login.html 같은 웹 문서를 먼저 띄워보게 됩니다. 웹 서버로부터 응답을 받았을 때도 응답 결과가 웹 문서로 나타납니다. 이와 같이 사용자는 웹 문서를 주로 보게 되므로 웹 문서가 조금 더 예쁘게 보이는 것이 좋겠죠? 여기에서는 최대한 간단한 방법으로 웹 문서를 예쁘게 꾸며 보겠습니다.

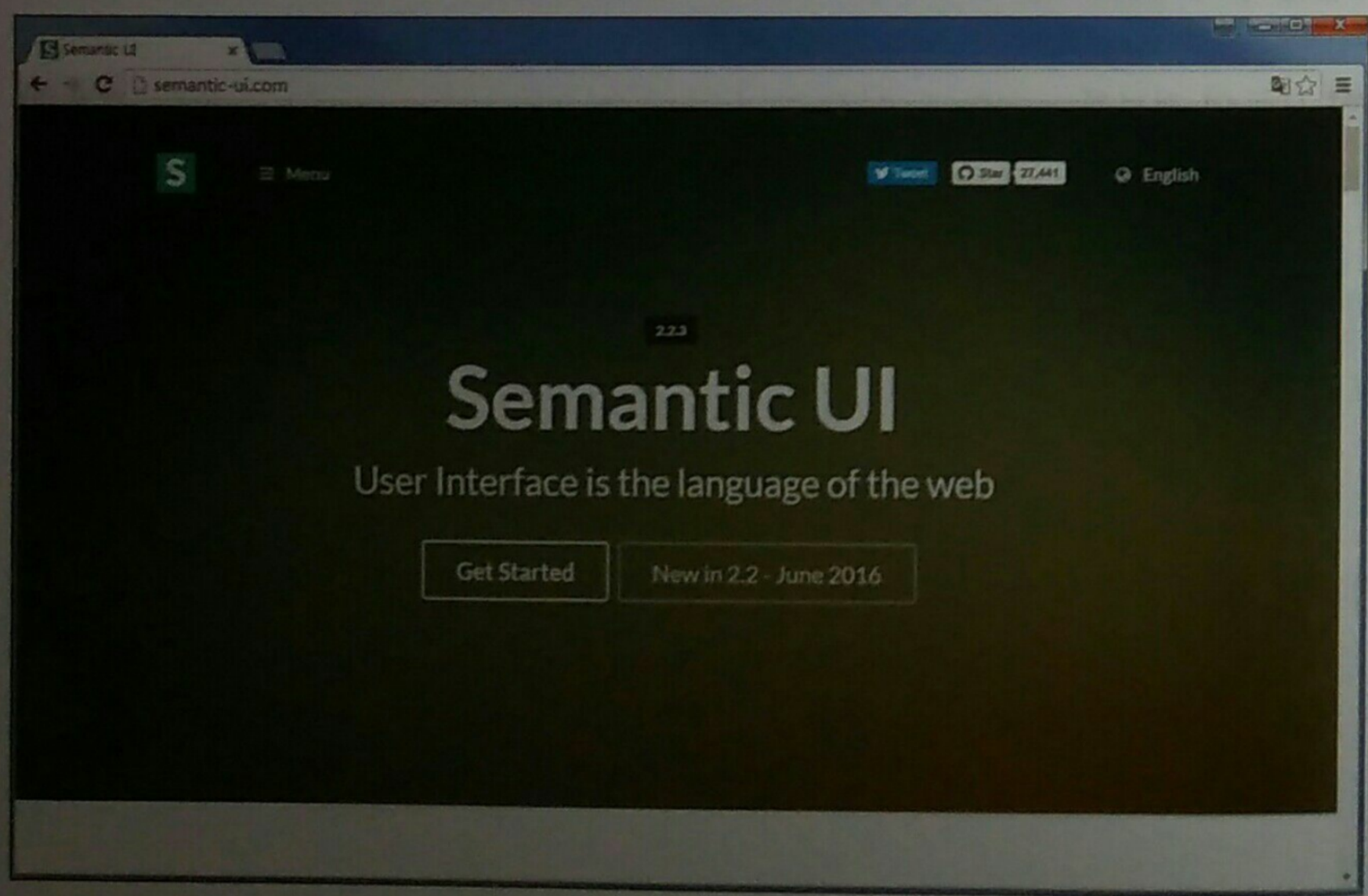
**주의!** 이 책에서 HTML5나 CSS에 대해서는 자세히 다루지 않습니다. 만약 기초적인 HTML5나 CSS에 대해 알고 싶다면 다른 책을 참조하기 바랍니다.

## Semantic UI 라이브러리로 웹 문서 꾸미기

웹 문서를 예쁘게 꾸밀 때 CSS와 자바스크립트 코드만을 사용해야 한다면 상당히 많은 양의 코드를 입력해야 합니다. 그런데 입력하는 코드의 양을 줄이고 싶다면 UI 라이브러리를 사용할 수 있습니다. 여기에서는 UI 라이브러리 중 하나를 샘플로 사용하여 적은 양의 코드로 웹 문서를 예쁘게 꾸며 보겠습니다.

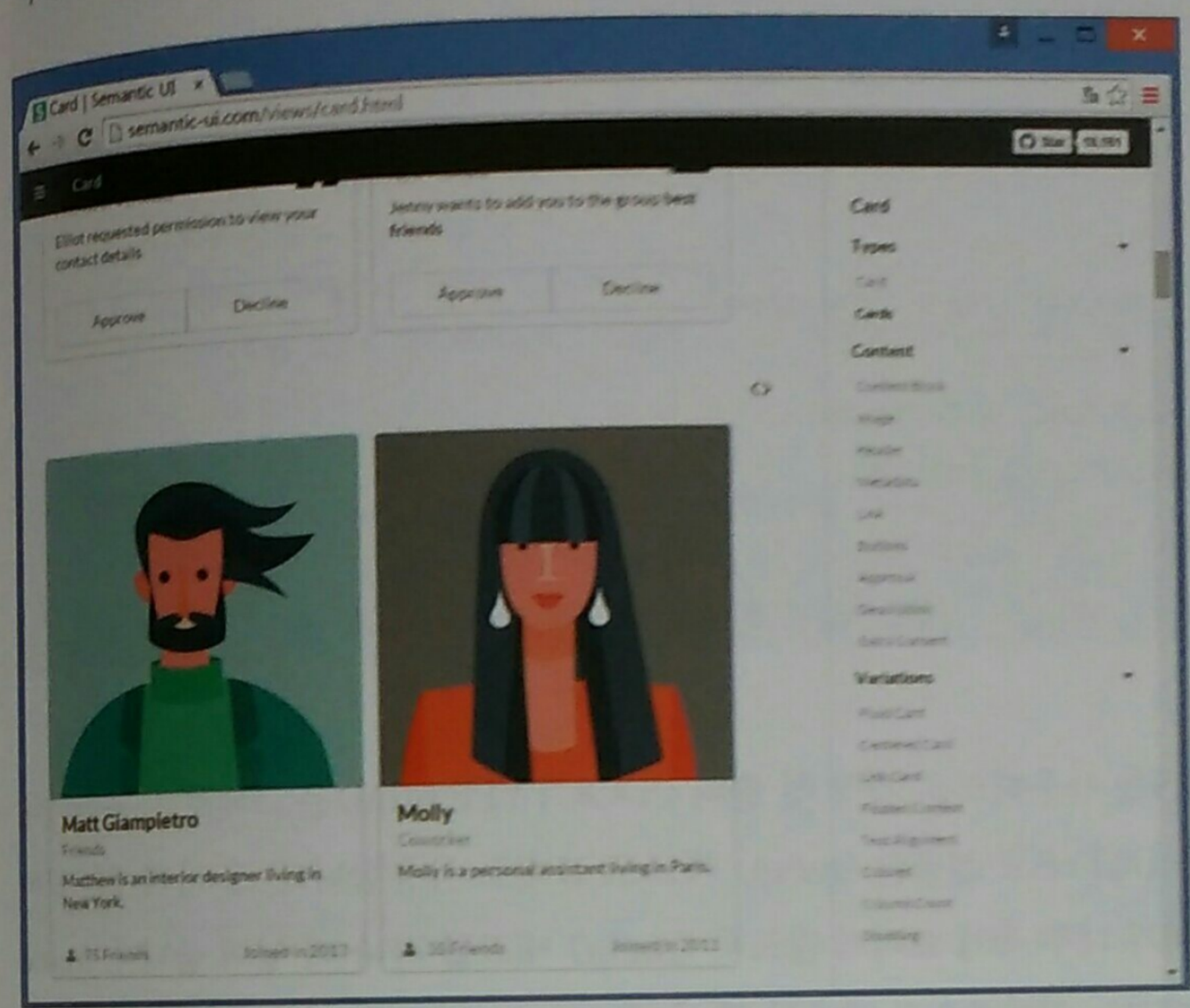
다양한 UI 라이브러리 중에서 Semantic UI 라이브러리는 태그만으로도 버튼이나 입력 상자와 같은 웬만한 UI 구성 요소를 만들 수 있습니다. 다음 웹 사이트를 열어보면 Semantic UI 라이브러리에서 제공하는 UI 구성 요소 샘플을 확인할 수 있습니다.

▶ http://semantic-ui.com




▲ Semantic UI 사이트

Semantic UI 라이브러리는 심플한 UI 스타일을 사용하며, 모바일 단말에서도 제대로 동작합니다. 먼저 왼쪽 위의 [Menu] 버튼을 눌러 보면 버튼이나 입력 상자와 같은 구성 요소를 어떻게 만들 수 있는지 확인할 수 있습니다. 그중에서 Card 컴포넌트를 눌러 보면 다음과 같이 하나의 정보를 박스 안에 넣어서 보여 주는 카드 방식의 UI를 볼 수 있습니다.




▲Semantic UI의 Card 컴포넌트

이 외에도 다양한 UI 구성 요소가 있습니다. 그러면 앞에서 만든 프로젝트의 웹 문서를 Semantic UI의 버튼, 입력 상자, Card 컴포넌트 등을 사용해서 수정해 보겠습니다.

CSS 파일 하나와 자바스크립트 파일 하나를 웹 문서에 링크하면 Semantic UI를 사용할 수 있습니다. npm 프로그램으로 semantic-ui 패키지를 설치하면, 그 안에 사용할 수 있는 라이브러리 파일들이 있습니다. 명령 프롬프트를 열고 다음과 같이 semantic-ui 패키지를 설치합니다.

```
% npm install semantic-ui --save
```

앞에서 만든 ModuleExample 프로젝트를 복사하여 DefaultExample 프로젝트를 만듭니다. 그다음 새로 설치한 모듈이 들어 있는 [node_modules/semantic-ui] 폴더 안에서 semantic.min.css 파일과 semantic.min.js 파일을 DefaultExample 프로젝트의 [/public] 폴더로 복사합니다.

먼저 로그인 웹 문서에서 Semantic UI 라이브러리를 사용하도록 수정해 보겠습니다. login.html 파일을 열고 다음과 같이 입력합니다.

참조 파일 | DefaultExample>/public/login.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset = "UTF-8">
    <meta name = "viewport" content = "width = device-width, height = device-height,
                                       initial-scale = 1">

    <title>로그인</title>

    <link href = "./semantic.min.css" rel = "stylesheet">

    <script src = "./jquery-2.1.4.min.js"></script>
    <script src = "./semantic.min.js"></script>

  ……
```

우리가 만들 웹 문서는 HTML5를 사용하므로 가장 첫 줄의 〈!DOCTYPE html〉은 그대로 사용합니다. 〈head〉 태그 안에서 인코딩을 UTF-8로 설정한 〈meta〉 태그도 그대로 사용합니다. 그 아래에는 모바일 단말에서 조회할 때 웹 문서가 너무 작게 보이지 않도록 〈meta〉 태그를 하나 더 넣어 줍니다. name 속성 값이 viewport인 메타 태그는 일반적으로 모바일 웹 사이트에서 사용하는 태그이며, 모바일 단말의 가로 크기와 세로 크기에 맞춰 웹 문서의 크기를 정해 줍니다. 따라서 PC 화면에 맞춰진 크기 때문에 웹 문서가 너무 작게 보이는 문제를 없애 줍니다.

〈title〉 태그 아래에는 semantic-ui 라이브러리에 필요한 CSS 파일을 사용하도록 〈link〉 태그를 추가합니다. 그 아래에는 제이쿼리(Jquery)를 사용하기 위한 파일과 semantic-ui 라이브러리에 필요한 자바스크립트 파일을 사용하도록 〈script〉 태그를 추가합니다.

**주의!** 만약 제이쿼리 사용에 필요한 자바스크립트 파일이 없다면 제이쿼리 사이트(http://jquery.com)에서 자바스크립트 파일 하나만 다운로드하면 됩니다.

## Card 컴포넌트 추가하기

이제 semantic-ui 라이브러리를 사용할 준비가 되었습니다. 〈body〉 태그 안에 Card 컴포넌트를 사용하기 위한 태그를 추가합니다.

```
.....

<body>
  <div class = "container">
    <div id = "cardbox" class = "ui blue fluid card">
      <div class="content">
        <div class = "left floated author">
          <img id = "iconImage" class = "ui avatar image" src = "./images/author.png">
        </div>
        <div>
          <div id = "titleText" class = "header">로그인</div>
          <div id = "contentsText" class = "description">
            아이디와 비밀번호를 입력하고 로그인하세요.
          </div>
        </div>
      </div>

.....
```

 태그는 웹 문서 안에서 일정 공간을 지정받는 용도로 사용되기 때문에 아주 흔하게 볼 수 있습니다. 여기에서도 로그인에 필요한 컴포넌트들을 가운데로 보내기 위해 class 속성 값이 'container'인  태그를 먼저 추가했습니다. 이  태그 안에 Card 컴포넌트를 추가했는데,  태그를 추가한 후 class 속성 값에 'ui blue fluid card'만 넣어 주면 아래쪽에 파란색 줄이 생긴 카드 상자가 만들어집니다. 그 안에 다시  태그를 추가하고 class 속성 값을 'content'로 작성합니다. 작은 이미지를 보여 주고 싶다면 'left floated author' 값을 가진 class 속성을 추가하고 그 안에 class 속성 값이 'ui avatar image'인  태그를 추가합니다. 이렇게 추가한 아바타(사람 얼굴) 이미지는 원형으로 나타납니다. 인터넷에서 아무 아이콘 이미지나 다운로드해서 author.png라는 파일 이름으로 [images] 폴더 안에 넣으면 이미지가 보입니다.

그 아래에는 두 개의  태그를 추가하고 제목과 설명글을 넣어 줍니다.

## <table> 태그로 로그인 입력 상자와 버튼 추가하기

이제 <table> 태그로 아이디와 비밀번호를 입력할 수 있는 입력 상자, 그리고 [로그인] 버튼을 넣어 보겠습니다.

```
.....

<form id = "form1" method = "post" action = "/process/login">
  <table>
    <tr class = "row">
      <td class = "col1"><label id = "contentsText">아이디</label></td>
      <td class = "col2" colspan = "2">
        <div class = "ui input">
          <input class = "inputbox" type = "text" name = "id">
        </div>
      </td>
      <td></td>
    </tr>
    <tr class = "row">
      <td class = "col1"><label id = "contentsText">비밀번호</label></td>
      <td class = "col2" colspan = "2">
        <div class = "ui input">
          <input class = "inputbox" type = "password" name = "password">
        </div>
      </td>
      <td></td>
    </tr>
    <tr valign = "baseline">
      <td></td>
      <td>
        <div class = "ui toggle checkbox">
          <input type = "checkbox" name = "saveOption">
          <label>아이디 저장</label>
        </div>
      </td>
      <td id = "buttonContainer" align = "right">
        <input id = "submitButton" class = "ui primary button" type = "submit"
               value = "로그인" name = "">
      </td>
    </tr>
  </table>
</form>

.....
```

태그를 많이 추가했지만 결과물은 단순히 세 줄로 만든 아이디 입력 상자, 비밀번호 입력 상자, [로그인] 버튼입니다. [로그인] 버튼 앞에는 아이디를 저장할 것인지를 묻는 체크 상자가 있습니다.

## 〈style〉 태그로 전체 화면 모양 만들기

마지막으로 〈head〉 태그 안에 〈style〉 태그를 추가하여 전체 화면 모양을 결정합니다.

**참조 파일** | DefaultExample>/public/login.html

```
……

<style>
  * {
    padding : 0;
    margin : 0;
    box-sizing : border-box;
  }

  html {
    width : 100%;
    height : 100%;
  }

  body {
    width : 100%;
    height : 100%;
    color : #000;
    background-color : #fff;
  }

  .container {
    width : 100%;
    height : 100%;
    display : flex;
    flex-flow : column wrap;
    align-items : center;
    justify-content : center;
  }

……
```

웹 브라우저마다 margin과 padding 값이 조금씩 다릅니다. 먼저 margin과 padding 값을 0으로 초기화하는 CSS 속성을 추가합니다. * 기호를 넣으면 웹 문서 안의 모든 태그에 CSS 속성을 적용할 수 있습니다. 그리고 문서 안에 들어간 요소의 크기보다 테두리가 더 커지지 않도록 box-sizing 속성의 값을 border-box로 넣어 줍니다.

그 밑에는 〈html〉과 〈body〉 태그의 가로와 세로 크기를 100%로 만들어 주고 배경색도 조절합니다. 이 CSS 속성은 모바일 단말의 가로와 세로 크기에 맞추는 데 자주 사용됩니다. 〈body〉 태그 안에서 가장 먼저 넣은 〈div〉 태그의 class 속성 값이 container였는데, 이 문서 요소는 display 속성을 적용하고 그 값을 flex로 설정합니다. 이 속성으로 flexbox를 사용하여 레이아웃을 구성합니다. align-items와 justify-content 속성 값을 모두 center로 설정하면 로그인에 필요한 Card 컴포넌트가 화면 가운데로 정렬됩니다.

그 아래에는 다음과 같이 각 문서 요소에 대해 필요한 CSS 속성을 추가합니다.

**참조 파일** | DefaultExample>/public/login.html

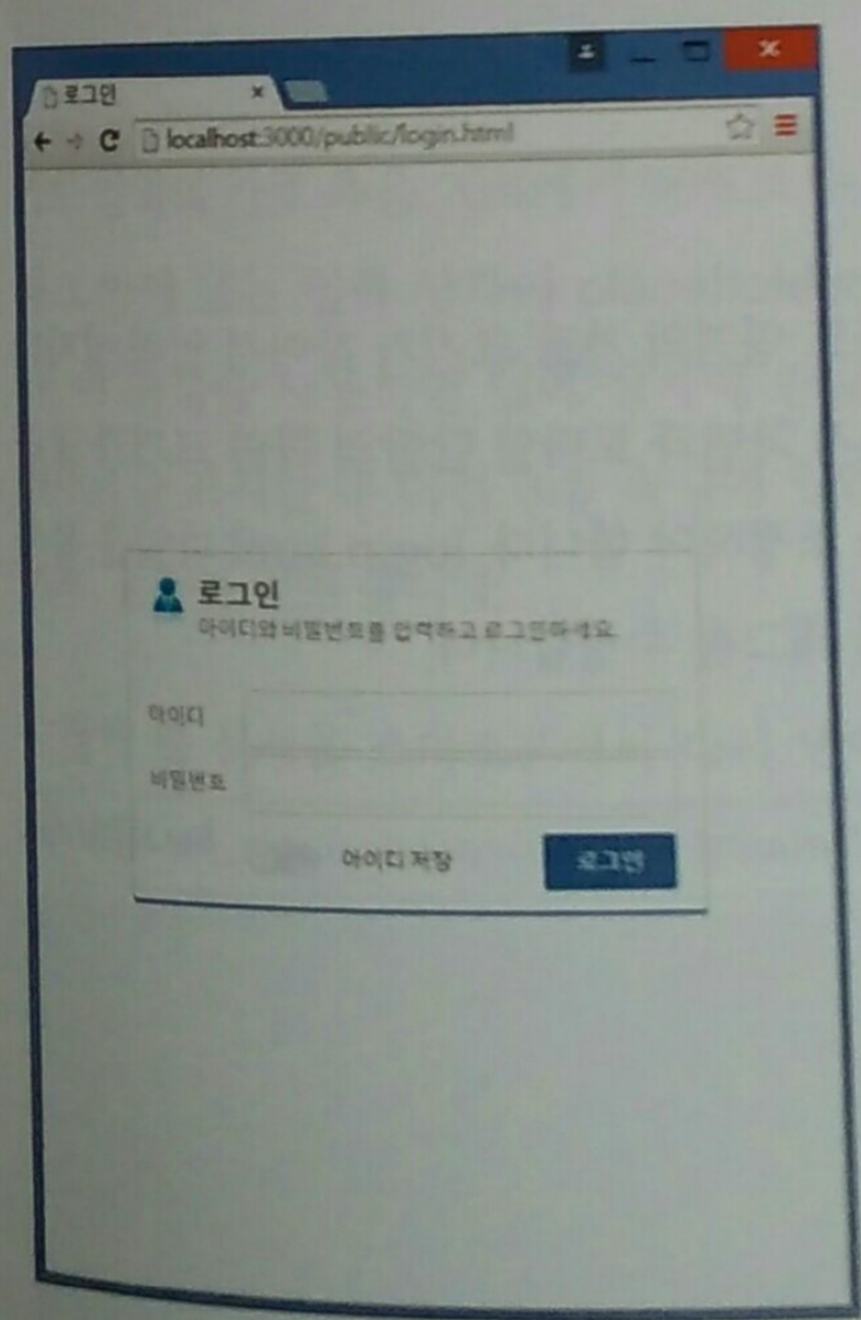
```
.....

  #cardbox {
    width : 70%;
  }

  #iconImage {
    display : inline;
  }

  #titleText {
    font-size : 1.4em;
    font-weight : bold;
    color : #777;
  }

  #contentsText {
    color : #999;
  }

  #form1 {
    padding : 1em;
  }

  .row {
    height : 3em;
  }

  .col1 {
    width : 5em;
  }
```

```
.inputbox {
  width : 20em;
}

#buttonContainer {
  padding-top : 0.6em;
  text-align : right;
}
</style>

……
```

필요에 따라 글자 크기, 가로 크기, 세로 크기 등을 font-size, width, height 속성으로 설정합니다. 이 책에서는 태그 구성이나 CSS에 대한 자세한 설명은 생략합니다.

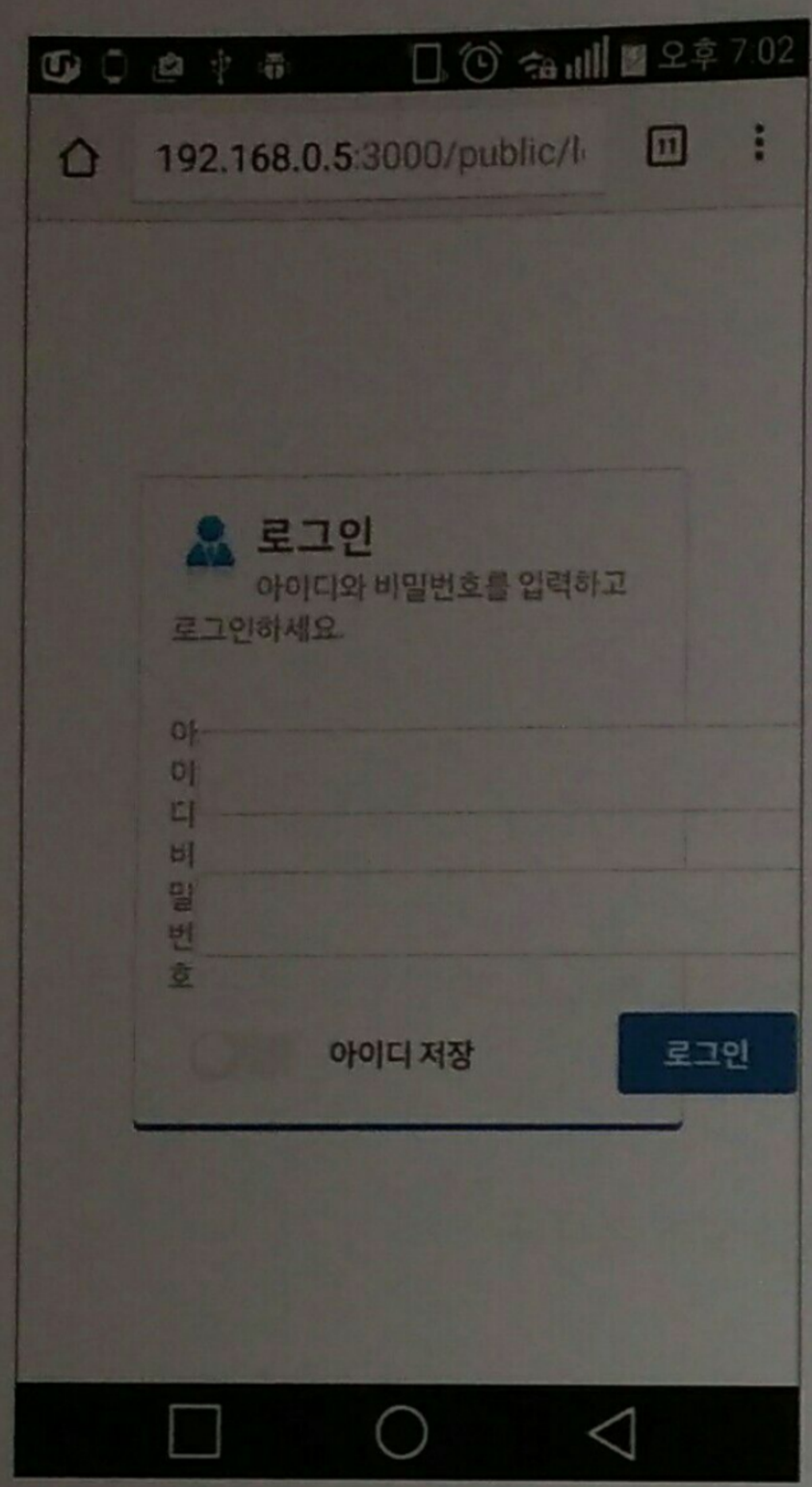
이제 웹 브라우저를 열고 app.js 파일을 실행한 후 다음 사이트를 열어 봅니다.

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




◀ Semantic UI로 예쁘게 만든 로그인 페이지

이제 단순하지만 예쁜 모양의 로그인 화면이 만들어졌습니다. [아이디 저장] 체크 상자는 이전에 만든 프로젝트에는 없었지만 로그인 화면에서 자주 사용되므로 화면 구성을 위해 넣어 두었습니다. 이제 스마트폰 단말기가 PC와 같은 Wi-Fi로 연결되었다면 PC의 IP를 입력하여 모바일 단말의 웹 브라우저로 조회해 볼 수 있습니다.




◀ 스마트폰 웹 브라우저의 로그인 화면

## 로그인 카드를 모바일 화면에 맞도록 CSS 조정하기

그런데 스마트폰 웹 브라우저에서 로그인 화면을 조회하면 카드 영역의 가로 크기가 작아서 입력 상자나 버튼이 카드 밖으로 벗어나 있습니다. 이것은 PC의 웹 브라우저 화면과 모바일 단말의 화면 크기가 달라서 발생하는 문제입니다. 따라서 모바일 화면에 맞도록 CSS를 조정해야 합니다. login.html 파일을 복사해서 login_mobile.html 파일을 만들고 다음과 같이 CSS와 태그를 수정합니다.

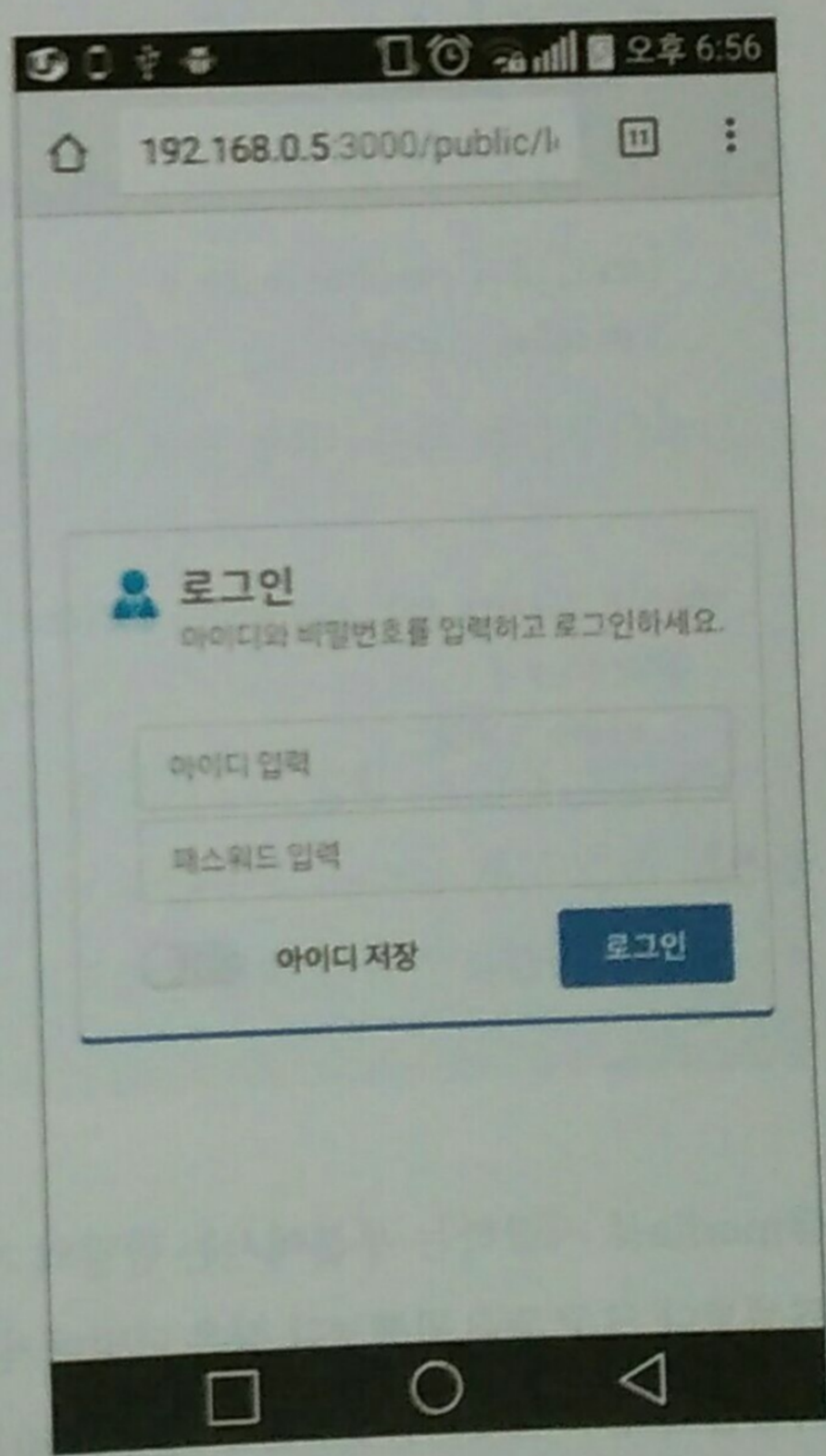
**참조 파일** | DefaultExample>/public/login_mobile.html

```
……

  <style>

……

    #cardbox {
      width : 90%;
    }

……
```

```
</style>

……

<body>

……

    <div class = "ui input">
      <input class = "inputbox" type = "text" placeholder = "아이디 입력"
            name = "id">
    </div>

……

    <div class = "ui input">
      <input class = "inputbox" type = "password" placeholder = "패스워드 입력"
            name = "password">
    </div>

……
```

카드 영역의 가로 폭을 70%에서 90%로 수정합니다. 〈table〉 태그 안에 있는 입력 상자에 placeholder 속성을 추가합니다. 이 속성을 사용하면 입력 상자에 안내글이 나타납니다. 안내글을 표시한 후 이전 〈td〉 속성에 있던 '아이디'와 '비밀번호' 글자는 없애 줍니다.

수정한 웹 문서를 스마트폰에서 다시 열어 보면 다음과 같이 보입니다.




코드를 수정한 후 스마트폰에 나타난 웹 브라우저▶

PC의 웹 브라우저에서 로그인 화면을 요청하면 login.html 문서를 보여 주고, 모바일 단말에서 요청하면 login_mobile.html 문서를 보여주도록 만듭니다. 이렇게 하면 PC와 모바일 단말에서 모두 정상적으로 로그인 화면이 나타납니다.

### 반응형 웹으로 웹 문서를 구별해서 보여 주기

만약 하나의 파일에서 모바일 단말에서 요청한 것인지 아니면 PC에서 요청한 것인지를 구별해서 보여 주려면 **반응형 웹**(Responsive Web)을 사용하면 됩니다. 반응형 웹은 단말의 종류에 따라 다르게 보이는 웹 문서를 상황에 맞게 설정할 수 있습니다. 보통 **미디어 쿼리**(Media Query)를 사용해서 단말의 가로 크기나 출력 단말의 종류 등을 구별한 후 CSS를 조정합니다.

login.html 파일을 복사하여 login_responsive.html 파일을 만든 후 〈style〉 태그 안에 들어 있는 코드 마지막 부분에 다음 코드를 추가합니다.

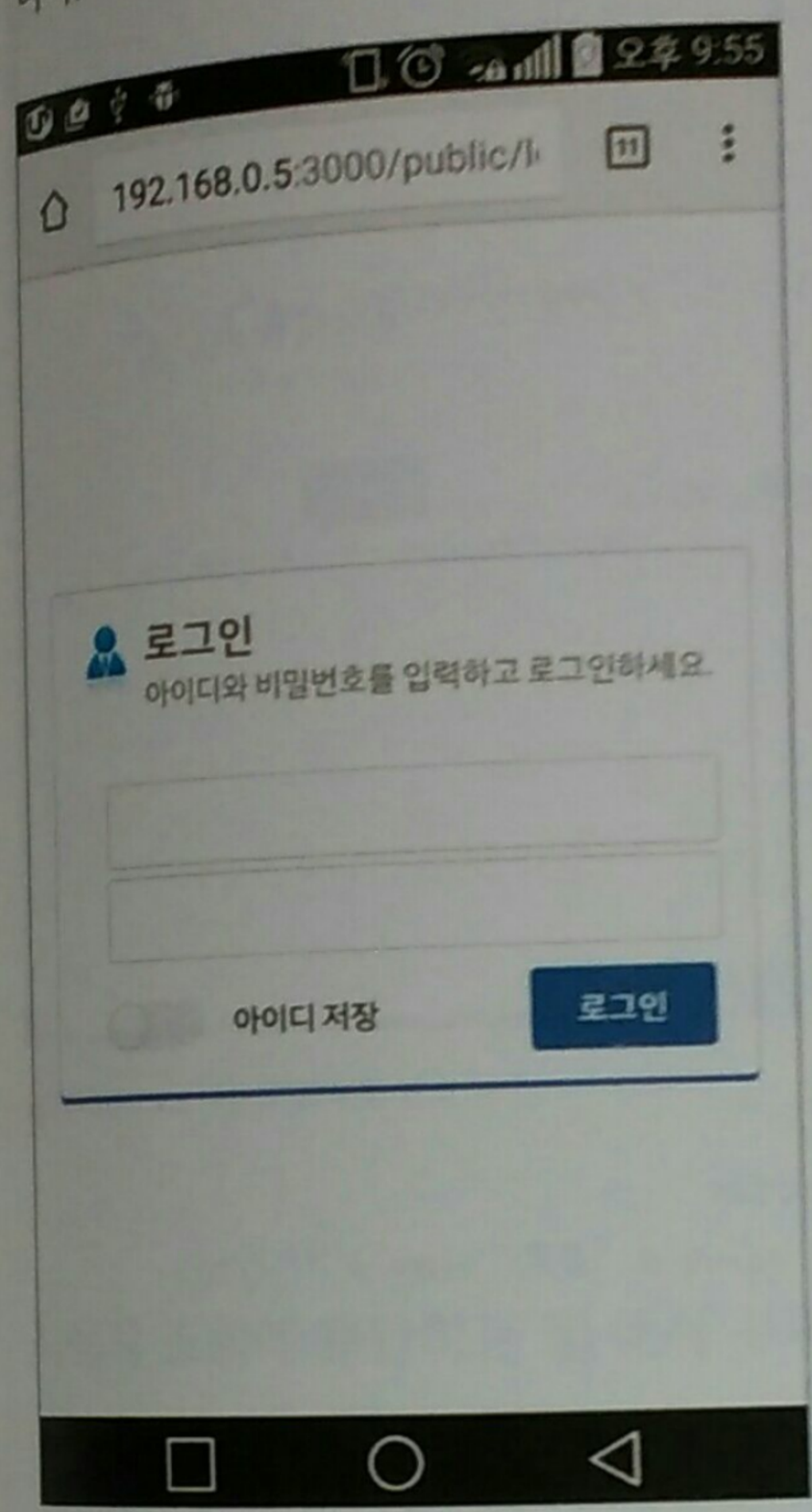
**참조 파일** | DefaultExample>/public/login_responsive.html

```
.....

@media screen and (min-width : 320px) and (max-width : 768px) {
  #cardbox {
    width : 90%;
  }

  label[id = contentsText] {
    display : none;
  }
}

@media screen and (min-width : 768px) and (max-width : 979px) {
  #cardbox {
    width : 80%;
  }
}

.....
```

@media로 시작하는 부분에서는 단말의 가로 크기를 최솟값과 최댓값 사이의 범위로 지정합니다. 320 픽셀보다 크고 768 픽셀보다 작은 단말에서는 id 속성 값이 cardbox인 카드 영역의 가로 크기를 90%로 지정합니다. 그리고 '아이디'와 '비밀번호'라는 글자가 들어 있는 〈label〉 태그를 보이지 않도록 만듭니다.

만약 768 픽셀보다 크고 979 픽셀보다 작다면 카드 영역의 가로 크기를 80%로 지정합니다.

이렇게 만든 login_responsive.html 파일을 열어 보면 PC와 모바일 단말에서 모두 이상 없이 출력됩니다.




▲login_responsive.html 파일을 열어 본 경우

입력 상자에 placeholder 속성이 없어서 안내글은 표시되지 않지만 화면 레이아웃은 깨지지 않습니다.

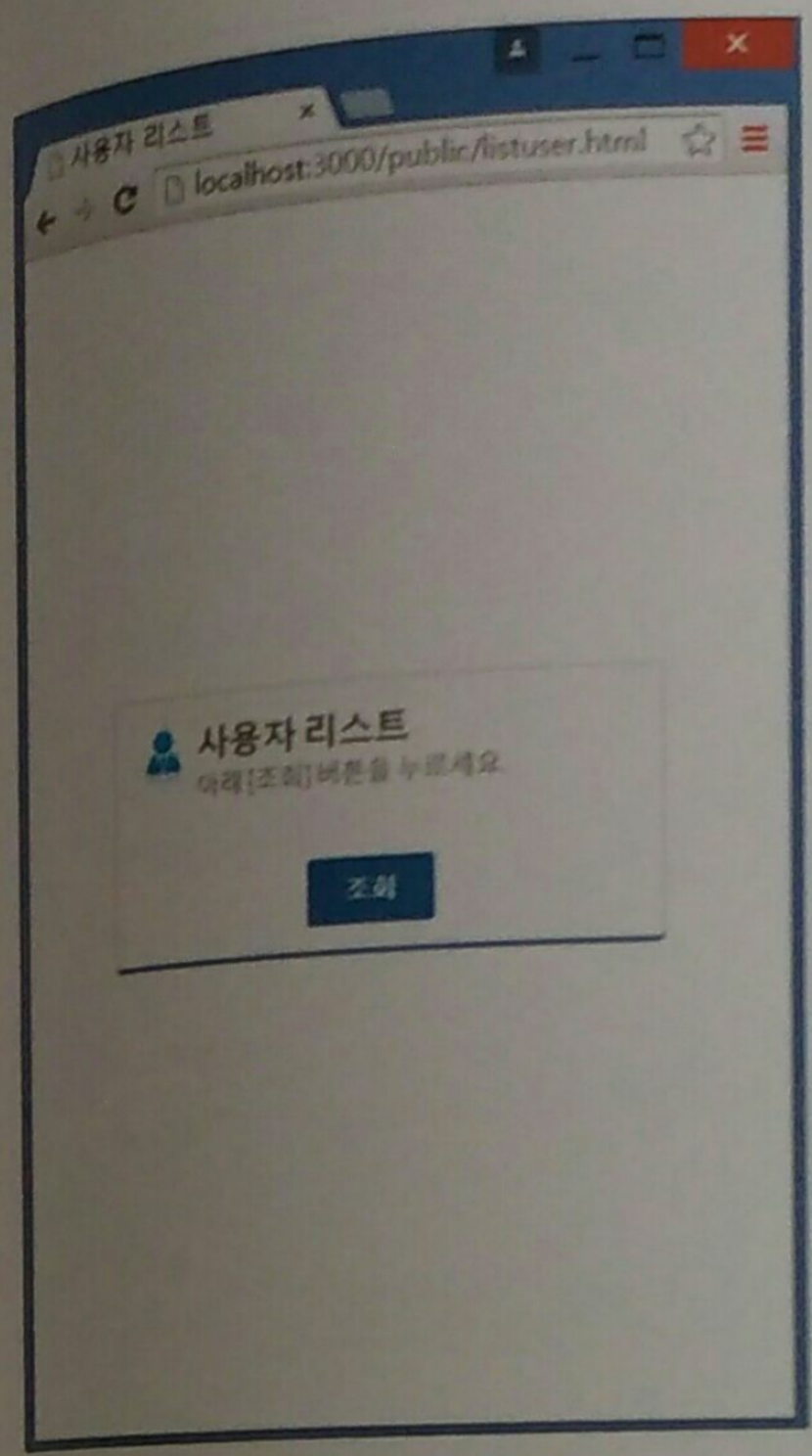
## 사용자 리스트 웹 문서 수정하기

지금까지 실습한 내용을 모두 이해했다면 이제 사용자 리스트 웹 문서도 수정할 수 있을 것입니다. 사용자 리스트를 요청하는 웹 문서는 listuser.html 파일입니다. 이 파일에서 〈head〉 태그 안의 내용을 login.html 파일과 똑같이 수정하고 〈body〉 태그 안에 있는 코드는 다음과 같이 수정합니다.

```
.....

  <div class = "container">

    <div id = "cardbox" class = "ui blue fluid card">
      <div class = "content">
        <div class = "left floated author">
          <img id = "iconImage" class = "ui avatar image" src = "./images/author.png">
        </div>
        <div>
          <div id = "titleText" class = "header">사용자 리스트</div>
          <div id = "contentsText" class = "description">
            아래 [조회] 버튼을 누르세요.
          </div>
        </div>
      </div>

      <form id = "form1" method = "post" action = "/process/listuser">
        <table>
          <tr class = "row" align = "center" valign = "middle">
            <td id = "buttonContainer"><input id = "submitButton"
                class = "ui primary button" type = "submit" value = "조회" name = ""></td>
          </tr>
        </table>
      </form>
    </div>
  </div>

.....
```

카드 영역을 만드는  태그는 login.html 파일에서 사용한 것처럼 글자만 바꾸었습니다. <table> 태그 안에는 [조회] 버튼만 배치합니다.

app.js 파일을 실행하고 listuser.html 파일을 조회하면 다음과 같은 결과가 나타납니다.




▲ listuser.html 파일을 열어 본 결과 화면

## 사용자 조회에 응답하는 웹 문서 꾸미기

아주 간단한 화면이 만들어졌습니다. 그런데 사용자 리스트는 위와 같은 요청 화면보다 응답 화면을 예쁘게 만드는 것이 중요합니다. 응답 코드는 자바스크립트로 되어 있으므로 여러분이 직접 코드를 넣어 보면 됩니다. 여기에서는 별도의 샘플 웹 문서를 하나 만들어 화면 레이아웃만 구성해 보겠습니다.

listuser.html 파일을 복사하여 listuser.html 파일을 만들고 태그 부분을 다음과 같이 수정합니다.

**참조 파일** | DefaultExample>/public/listuser.html

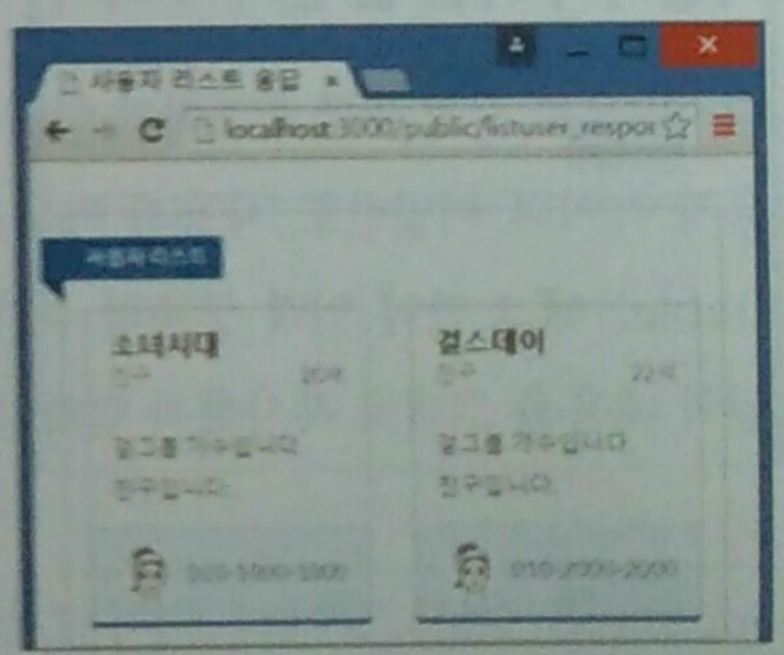
```
.....

<div class = "container">

  <br>
```

```
<div class = "ui raised segment">
  <a class = "ui blue ribbon label">사용자 리스트</a>
  <br><br>
  <div class = "ui two column grid">
    <div class = "column">
      <div class = "ui blue fluid card">
        <div class = "content">
          <div class = "header">소녀시대</div>
          <div class = "meta">
            <span class = "right floated time">20세</span>
            <span class = "category">친구</span>
          </div>
          <div class = "description">
            <br>
            <p>걸그룹 가수입니다.</p>
            <p>친구입니다.</p>
          </div>
        </div>
        <div class = "bottomArea extra content">
          <div class = "right floated author">
            <img class = "ui avatar image"
                 src = "./images/woman.png"> 010-1000-1000
          </div>
        </div>
      </div>
    </div>

.....
```

〈div〉 태그가 여러 번 중복되어 복잡하게 보이지만 하나씩 살펴보면 각각의 내용을 이해할 수 있을 것입니다. 가장 위쪽에 있는 〈div〉 태그는 화면 전체 영역을 차지하도록 만듭니다. 그 아래에 있는 〈div〉 태그에 'ui raised segment'로 class 속성 값을 추가하면 리본을 추가할 수 있습니다. 리본은 왼쪽이나 오른쪽에 만들어서 글자를 눈에 띄게 만들 수 있습니다. 그 아래에 있는 〈div〉 태그의 class 속성 값은 'ui blue ribbon label'로 넣어 줍니다. 그 아래에는 class 속성 값이 'ui two column grid'인 〈div〉 태그가 있는데 이것은 2분할된 격자를 만듭니다. 각각의 격자는 column 속성 값을 가지고 있어야 하며, 그 안에 Card 컴포넌트를 넣습니다. 카드 안에는 각 사용자 정보를 표시한 후 가장 아래쪽에 있는 영역은 CSS로 색상 값을 줍니다.

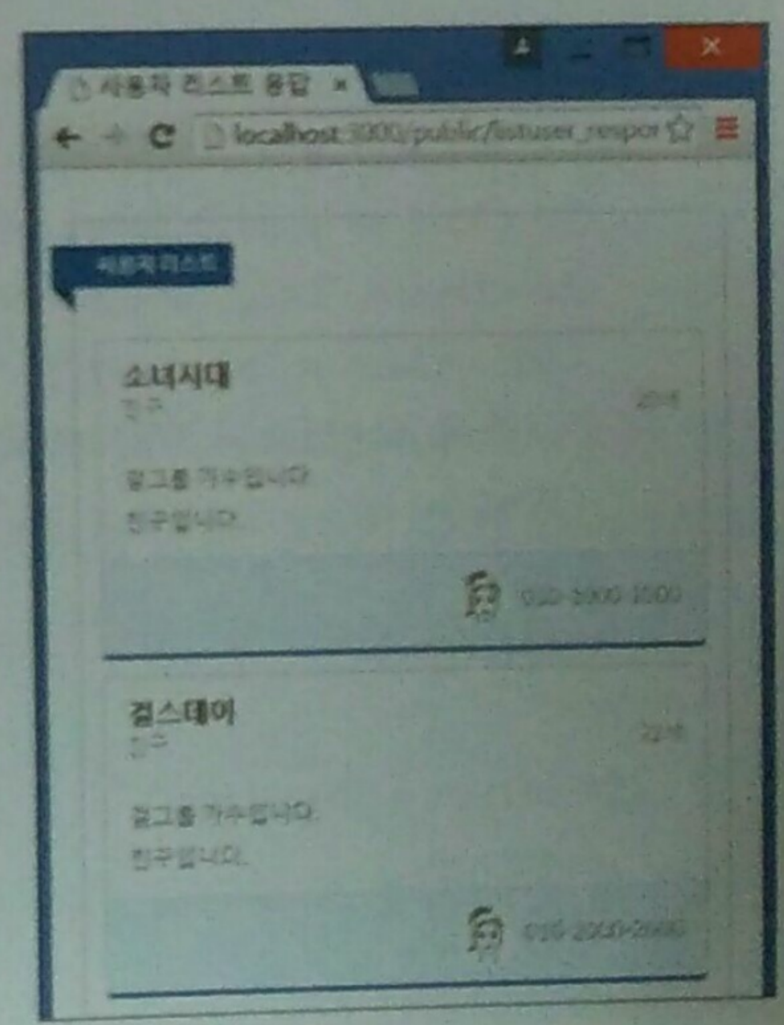
```
……,

  #cardbox {
    width : 70%;
  }

  #iconImage {
    display : inline;
    width : 2em;
    height : 2em;
  }

  .bottomArea {
    background-color : rgba(242, 248, 255, 255) !important;
  }
</style>

……,
```

listuser_response.html 파일을 조회하면 다음처럼 변경된 레이아웃을 확인할 수 있습니다.



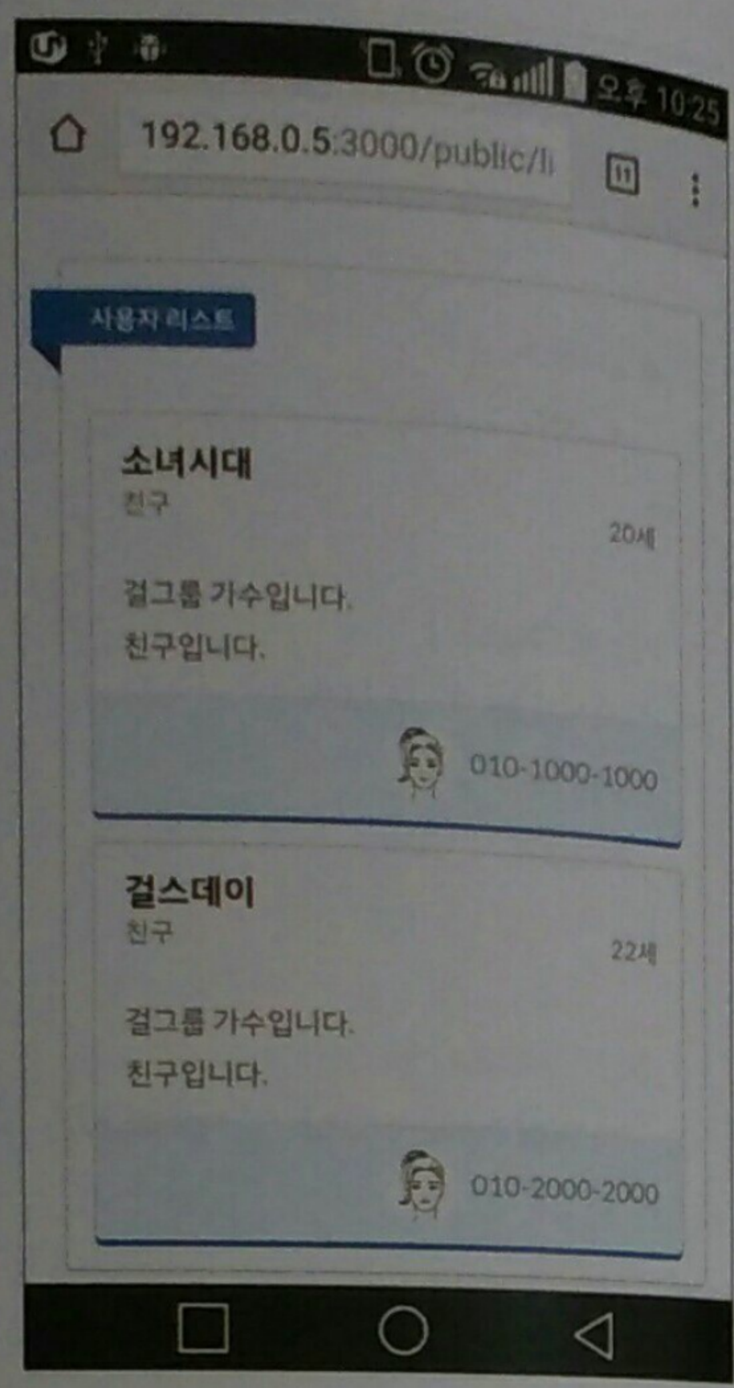

listuser_response.html 파일을 열어 본 결과 화면 ▶

화면의 왼쪽 위에 '사용자 리스트'가 적힌 파란색 리본이 달린 형태로 영역이 구성되었습니다. 각 카드 영역의 아래쪽에는 아이콘과 전화번호가 보입니다. 만약 병렬로 배치된 사용자 리스트를 세로로 재구성하려면  태그 중에서 class 속성 값이 'ui two column grid'인 태그와 'column' 태그를 제거하면 됩니다. 다음은 사용자 리스트가 세로로 배치되도록 수정한 레이아웃입니다.




listuser_response.html 파일을 수정한 화면 구성 ▶

Wi-Fi로 연결된 모바일 단말에서 조회했을 때도 다음처럼 이상 없이 출력됩니다.




listuser_response.html 파일을 모바일 단말에서 조회한 결과 화면 ▶

## 사용자 추가용 웹 문서 꾸미기

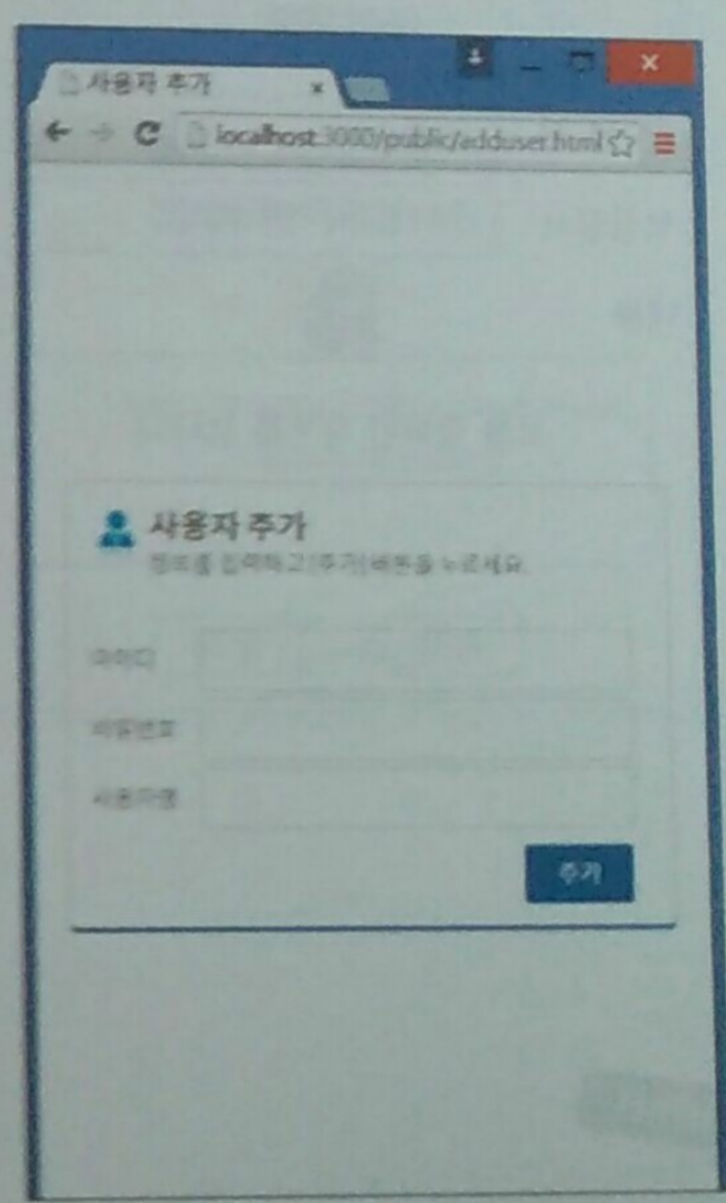
마지막으로 사용자를 추가하기 위해서 adduser.html 파일을 수정합니다. adduser.html 파일에서는 사용자 아이디, 비밀번호, 사용자 이름을 입력하는 입력 상자가 있어야 합니다. 이 입력 상자를 추가하려면 〈table〉 태그 안에 입력 상자의 수만 더 늘리면 됩니다. semantic-ui를 사용하기 위해 login.html 파일의 내용을 그대로 복사해서 입력한 후 사용자 이름을 입력할 수 있는 입력 상자를 하나 추가합니다.

**참조 파일** | DefaultExample>/public/adduser.html

```
……
  <tr class = "row">
    <td class = "col1"><label id = "contentsText">사용자명</label></td>
    <td class = "col2" colspan = "2">
      <div class = "ui input">
        <input class = "inputbox" type = "text" name = "name">
      </div>
    </td>
    <td></td>
  </tr>
  <tr valign = "baseline">
    <td></td>
    <td></td>
```

```
    <td id = "buttonContainer" align = "right">
      <input id = "submitButton" class = "ui primary button" type = "submit"
             value = "추가" name = "">

    </td>
  </tr>
 </table>
</form>

……
```

앞에서 몇 번 반복해서 작성한 코드가 조금씩 수정되었다는 것을 알 수 있습니다. 하나씩 수정해 보면 어렵지 않게 직접 구성할 수 있습니다. 수정을 마친 후 adduser.html 파일을 조회하면 다음 화면이 나타납니다.




사용자 추가 화면인 adduser.html 파일을 조회한 결과 ▶

지금까지 UI 라이브러리로 웹 문서를 예쁘게 꾸미는 방법을 알아보았습니다. UI 라이브러리는 Semantic UI 외에 다른 것도 많으니 **부트스트랩(Bootstrap)**이나 **타이톤(Titon)** 같은 다른 UI 라이브러리도 함께 사용해 보길 권합니다. 어떤 라이브러리를 사용하느냐가 중요한 것이 아니라 여러분의 목적에 맞는 레이아웃과 위젯이 들어간 결과물을 만들 수 있느냐가 중요합니다. 따라서 라이브러리는 필요에 맞게 선별해서 사용하세요.

이 장에서 만든 DefaultExample 프로젝트는 각 기능이 어느 정도는 별도의 파일로 분리되도록 만들었습니다. 그러므로 지금부터는 새로운 프로젝트를 만들 때 어떤 구조로 만들 것인지 알려 주는 기준이 될 것입니다. 또한 UI 라이브러리 중 하나인 Semantic UI로 만든 웹 문서는 앞으로도 화면을 구성할 때 자주 사용할 것입니다.

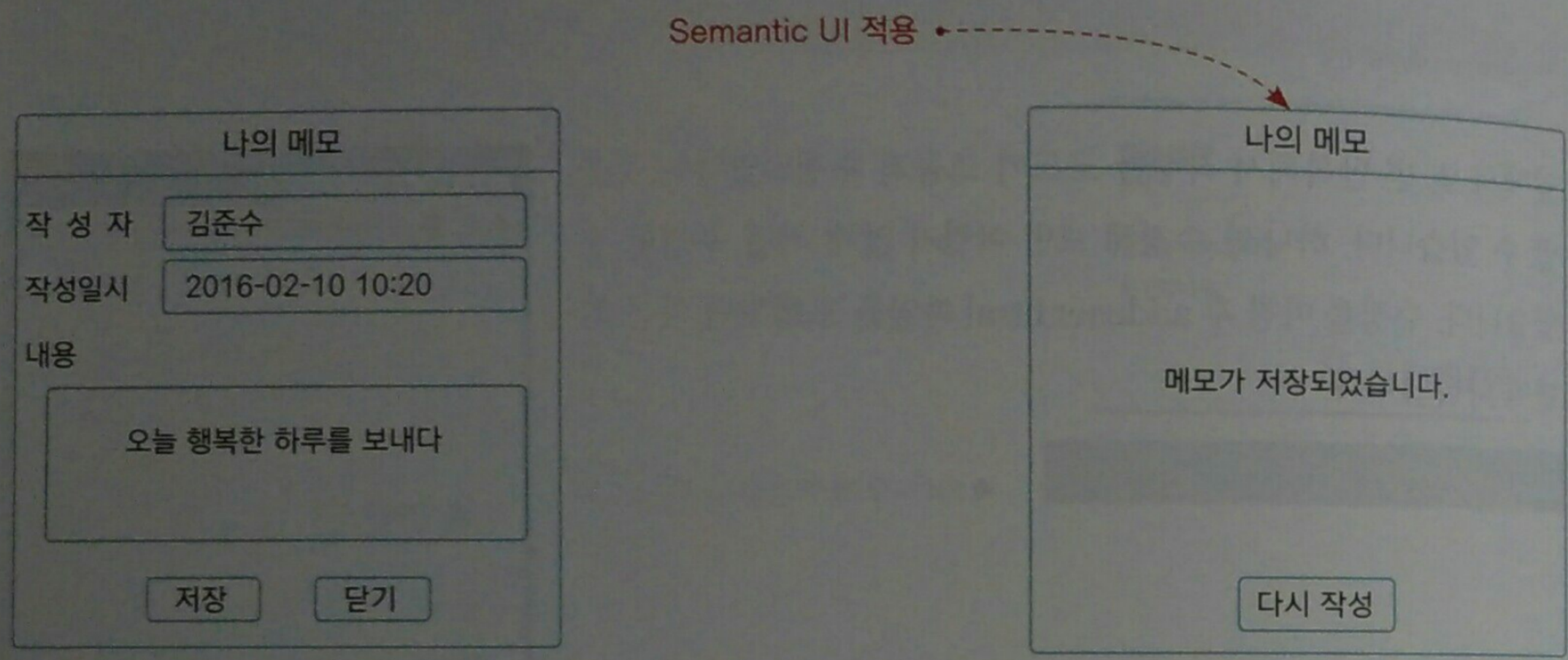
다음 장에서는 별도의 파일로 저장되어 있는 템플릿 파일에서 웹 문서를 읽어 온 후 응답 보낼 웹 문서를 자동으로 생성하여 클라이언트로 보내는 방법을 알아보겠습니다.

Do it!

도전 문제

01. 06장에서 만들었던 Mission-05의 메모 응답 화면에 Semantic UI를 적용해 보세요.

프로젝트 소스 Mission07 난이도 초급 ★☆☆☆




Hint!

1 응답 화면의 구성은 Mission-05의 것과 똑같으니 해당 소스 파일을 복사하여 새로운 파일을 만듭니다. 코드는 수정할 필요 없습니다.

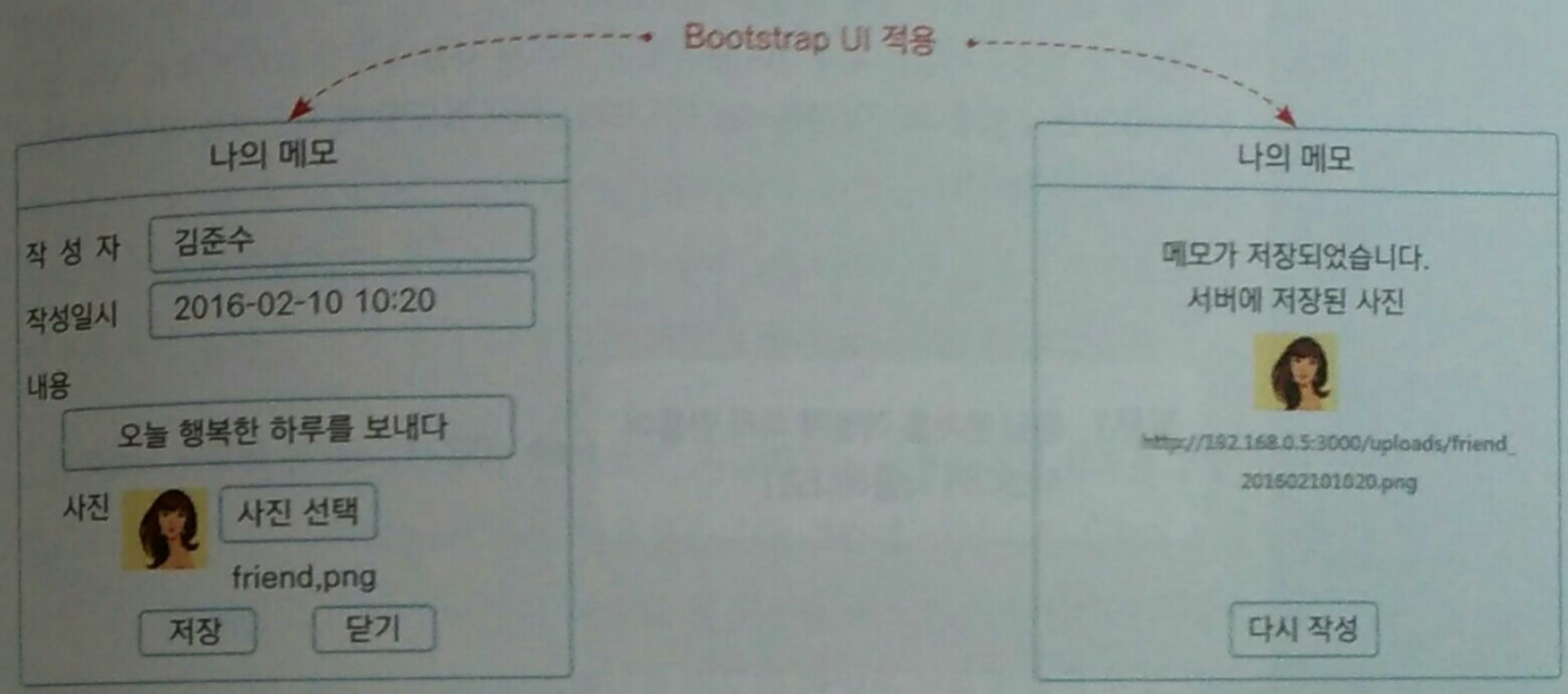
2 Semantic UI 라이브러리에서 적절한 위젯을 찾아 응답 화면에 들어가는 버튼이나 전체 구성을 적용합니다.

MEMO

DO it!
도전 문제

02. 06장에서 만들었던 Mission-06의 코드를 모듈화하고 메모 작성 화면과 응답 화면에 Bootstrap UI를 적용해 보세요.

프로젝트 소스 Mission08 난이도 중급 ★★☆☆




Hint!

1 작성 화면과 응답 화면의 구성은 Mission-06의 것과 같으므로 해당 소스 파일을 복사하여 새로운 파일을 만듭니다.

2 서버에 만들어져 있는 코드를 모듈화하여 기능별로 구분하기 쉽게 수정합니다.

3 Bootstrap UI 라이브러리에서 적절한 위젯을 찾아 작성 화면과 응답 화면에 있는 버튼과 전체 구성을 적용합니다.

MEMO