13 모바일 서버 만들기

모바일 서버는 모바일 단말에서 필요한 데이터를 조회하거나 저장할 수 있도록 만든 서버입니다. 그러면 모바일 단말에서 동작하는 앱에서는 웹 브라우저에서 접속하는 웹 서버를 사용할 수 없을까요? 그렇지는 않습니다. 웹 서버를 사용해도 모바일 단말에서 데이터를 요청하고 응답을 받을 수 있어 모바일 서버를 따로 만들지 않아도 됩니다. 하지만 PC가 아닌 모바일 단말을 클라이언트로 사용할 때는 단말의 정보를 관리하거나 단말로 알림 메시지를 보내는 등 모바일 단말에 특화된 기능을 사용할 수 있도록 모바일 서버를 만들어야 합니다. 이 장에서는 안드로이드 단말에서 동작하는 앱으로 웹 서버에 데이터를 요청한 후 응답 받은 결과를 보여 주는 방법에 대해 알아봅니다. 그리고 모바일에서 주로 사용하는 모바일 단말 관리와 푸시 서비스 기능도 함께 살펴보겠습니다.

질문으로 시작하기

질문1. 앱에서 웹 서버의 데이터를 조회하도록 만들 수 있나요?

➡ 13-1. 모바일 단말에서 웹 서버로 요청하기

질문2. 모바일 단말들에 대한 정보를 관리할 수 있나요?

➡ 13-2. 모바일 단말 관리 기능 만들기

질문3. 모바일 단말로 공지사항을 전송하고 싶나요?

➡ 13-3. 모바일 단말로 푸시 메시지 전송하기

13-1 | 모바일 단말에서 웹 서버로 요청하기

요즘에는 많은 사람이 스마트폰에서 동작하는 앱을 만들고 있는데 대부분 인터넷에 연결된 다른 서버에 저장되어 있는 정보를 가져와서 보여 주는 경우가 많습니다. 그리고 사람들끼리 의사소통이 가능한 대화 기능이나 글을 작성해서 올리는 업로드 기능까지 붙여서 사용하기도 합니다. 이런 앱들은 서버에 데이터를 저장하거나 조회할 수 있도록 만들어야 합니다. 예를 들어, 모바일 쇼핑 앱을 만든다고 가정하면 사용자에게 상품 이미지나 가격 정보 등을 보여 주기 위해 먼저 서버에서 데이터를 가져와야 합니다. 또한 사용자가 선택한 상품을 평가할 때 작성한 상품 평가 데이터를 서버로 보내 저장하는 기능도 필요합니다.

이런 모바일 서버를 만들려면 무엇이 필요할까요? 노드를 사용해서 웹 서버나 채팅 서버를 만들 때는 앞에서 익혔던 이클립스 개발 도구를 사용할 수 있습니다. 하지만 모바일 단말에서 동작하는 안드로이드 앱을 만들려면 안드로이드 스튜디오라는 개발 도구를 사용해야 합니다. 따라서 안드로이드 개발 도구인 안드로이드 스튜디오를 설치해서 필요한 데이터를 웹 서버에 요청하는 간단한 앱을 만들어 보겠습니다.

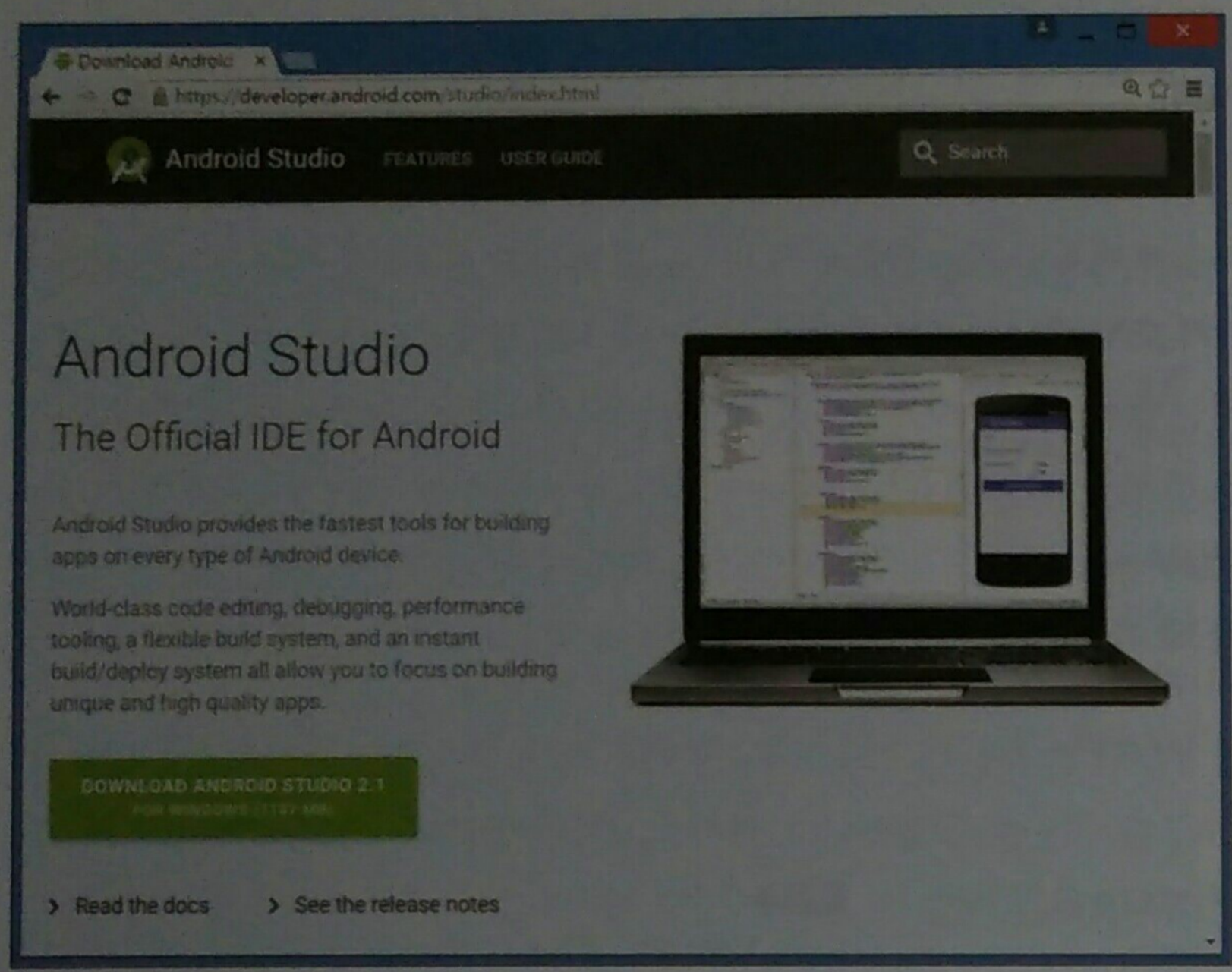
주의 이 책은 노드로 서버를 구성하는 방법을 다루는 책이므로 안드로이드 앱을 만드는 방법은 상세히 설명하지 않습니다. 따라서 이 책에 소개된 앱의 기능을 확인하려면 소스 코드와 함께 제공된 안드로이드 앱 프로젝트를 사용해서 앱을 실행해 보세요. 그리고 안드로이드 개발 도구 설치 방법이나 앱 개발 방법을 좀 더 자세히 알고 싶다면 《Do it! 안드로이드 앱 프로그래밍》과 같은 안드로이드 기본서를 참조하세요.

안드로이드 앱 개발 도구 설치하기

안드로이드 스튜디오(Android Studio)는 안드로이드 앱을 만드는 개발 도구입니다. 이 개발 도구도 자바를 사용해 실행하는데 이미 이클립스를 설치하면서 자바도 함께 설치했으므로 안드로이드 스튜디오만 설치하면 됩니다. 개발 도구 설치가 어렵다면 안드로이드 앱 개발 관련 서적을 참조하기 바랍니다. 먼저 다음 사이트에서 안드로이드 스튜디오의 설치 파일을 다운로드합니다.

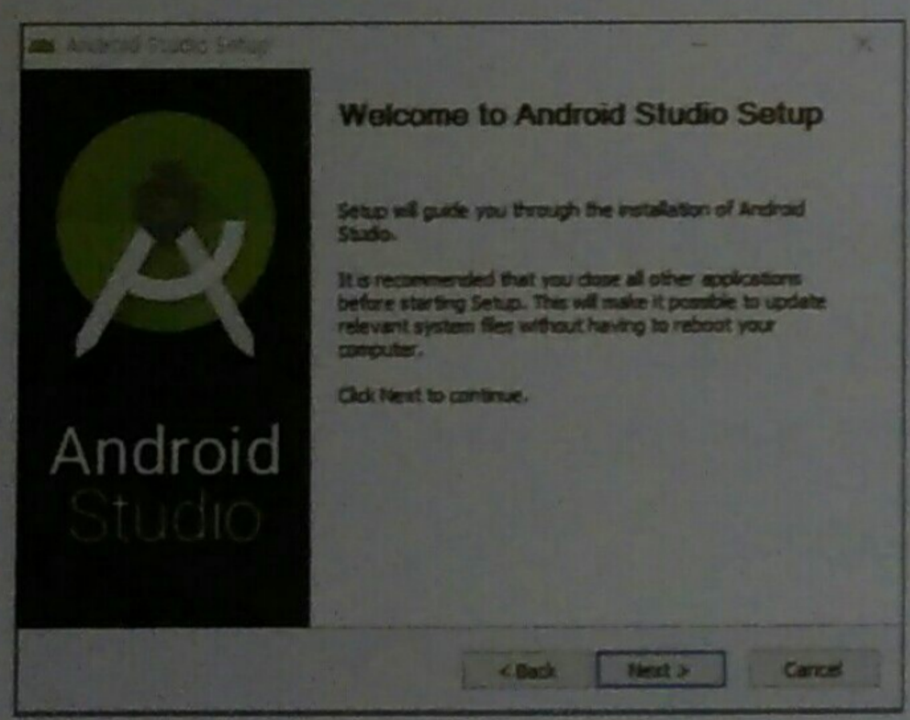
안드로이드 개발자 사이트 ▶ http://developer.android.com/studio/index.html

사이트를 열면 [DOWNLOAD ANDROID STUDIO 2.1] 버튼이 보입니다. 이 버튼을 클릭하여 설치 파일을 다운로드합니다.



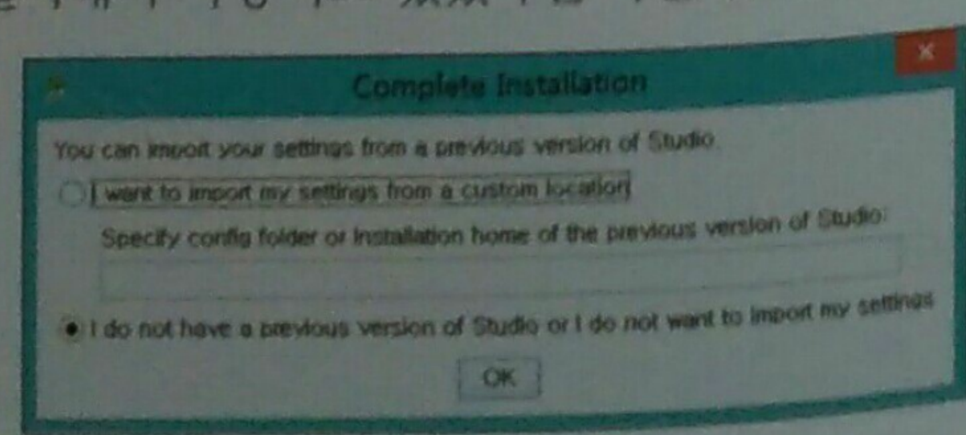
▲ 안드로이드 개발 도구 다운로드하기

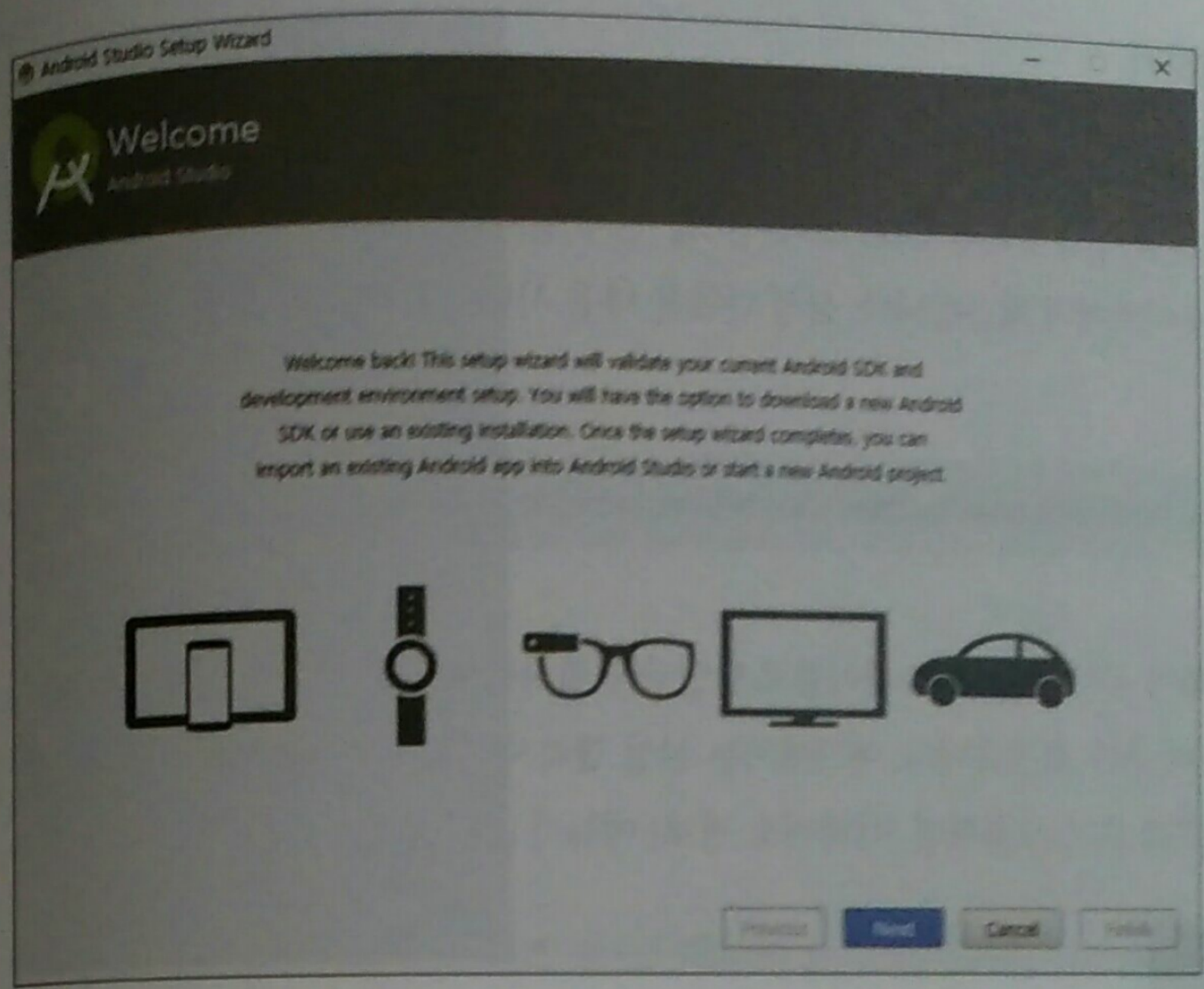
다운로드한 설치 파일을 더블클릭하면 설치 화면이 시작됩니다. [Next›] 버튼을 클릭해서 다음 화면으로 넘어갑니다. [Next›] 버튼을 눌러서 설치 과정을 순서대로 진행하면 어렵지 않게 설치됩니다.



▲ 안드로이드 개발 도구 설치 화면

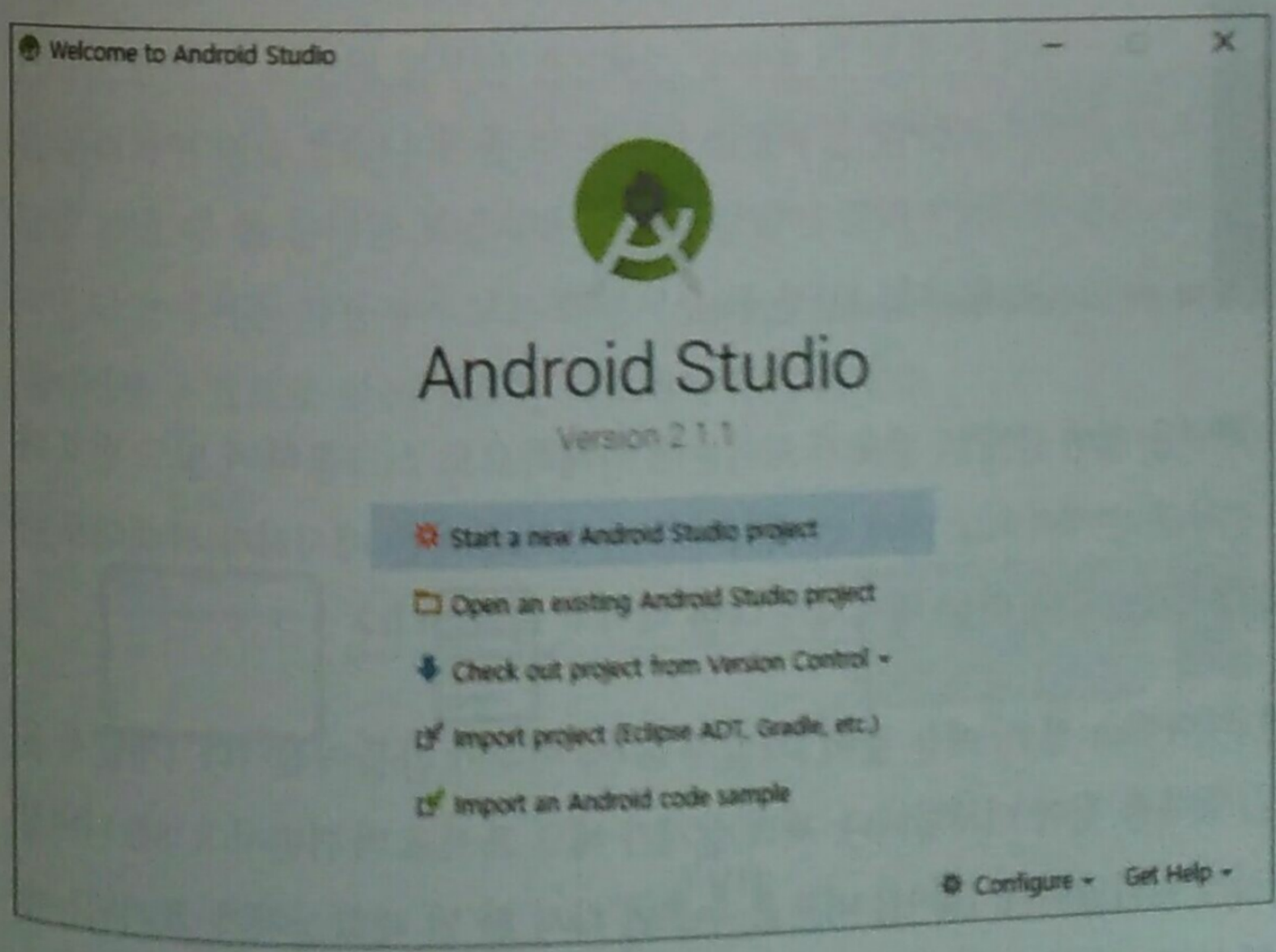
마지막 화면에서 [Finish] 버튼을 클릭하면 안드로이드 스튜디오 프로그램이 실행됩니다. 프로그램을 처음 실행한 경우에는 [Complete Installation] 대화상자가 나타납니다. 이 대화상자는 이전 설정을 가져올 것인지를 물어보는데 전에 안드로이드 스튜디오를 설치해서 사용하고 있었다면 다른 옵션을 선택할 수 있습니다. 처음 설치한 것이라면 [OK] 버튼을 클릭해서 다음 과정으로 넘어갑니다. 프로그램을 처음 실행한 경우라면 또다시 다른 내용을 선택하는 대화상자가 나타납니다.





▲안드로이드 스튜디오를 처음 실행했을 때 보이는 화면

[Next〉] 버튼을 눌러 넘어가면 설치에 필요한 것들을 자동으로 다운로드하여 설치하는 화면이 나타납니다. 이 과정은 몇 분 정도 걸릴 수 있습니다. 자동으로 진행되므로 아래쪽에 [Finish] 버튼이 활성화될 때까지 기다립니다. 추가 설치 과정이 끝난 후 아래쪽의 [Finish] 버튼을 클릭하면 드디어 안드로이드 스튜디오 프로그램의 **시작 화면(Start Window)**이 나타납니다. 이 화면에 나타난 메뉴를 눌러서 필요한 작업을 할 수 있습니다.



▲시작 화면

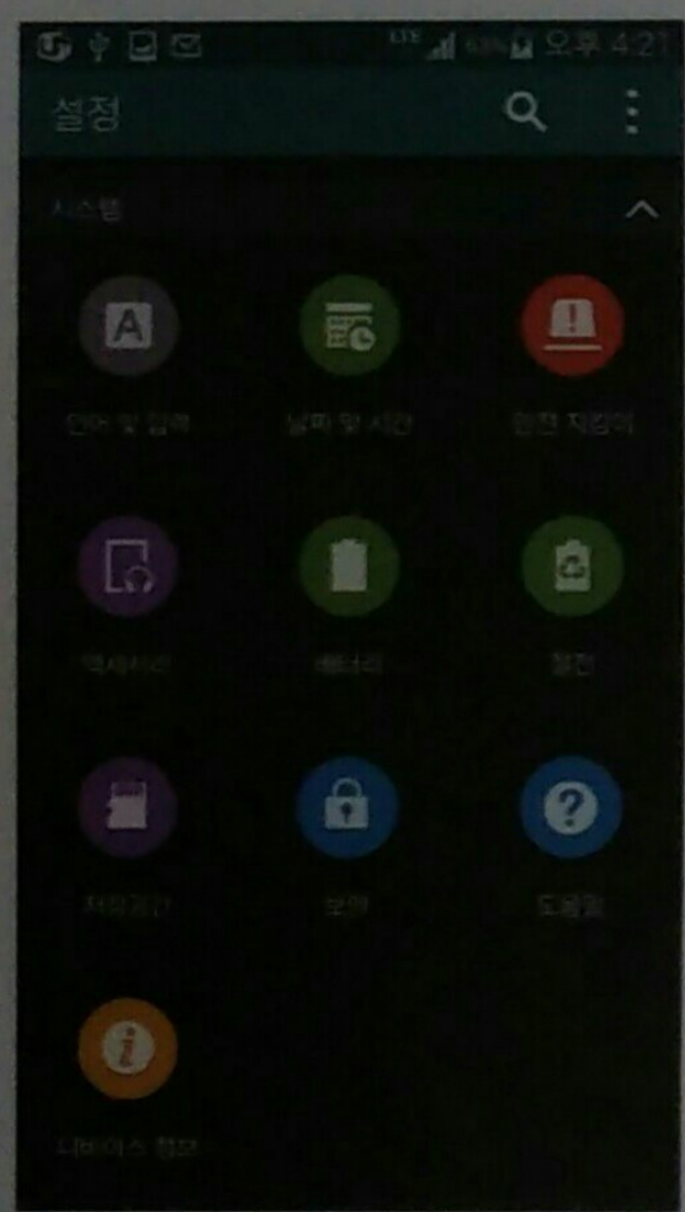
이제 안드로이드 스튜디오 프로그램의 설치가 끝났습니다.

실제 모바일 단말에 연결하기

앱을 만들어 실행할 때는 에뮬레이터로 실행할 수도 있지만 여기에서는 실제 모바일 단말을 연결하여 사용하기를 권합니다. 실제 단말을 연결하려면 USB 드라이버를 먼저 설치해야 합니다. USB 드라이버는 일반적으로 단말기 제조사 사이트에서 제공합니다. 삼성 단말은 다음 사이트에서 USB 드라이버를 다운로드할 수 있습니다.

▶ http://local.sec.samsung.com/comLocal/support/down/kies_main.do?kind=usb

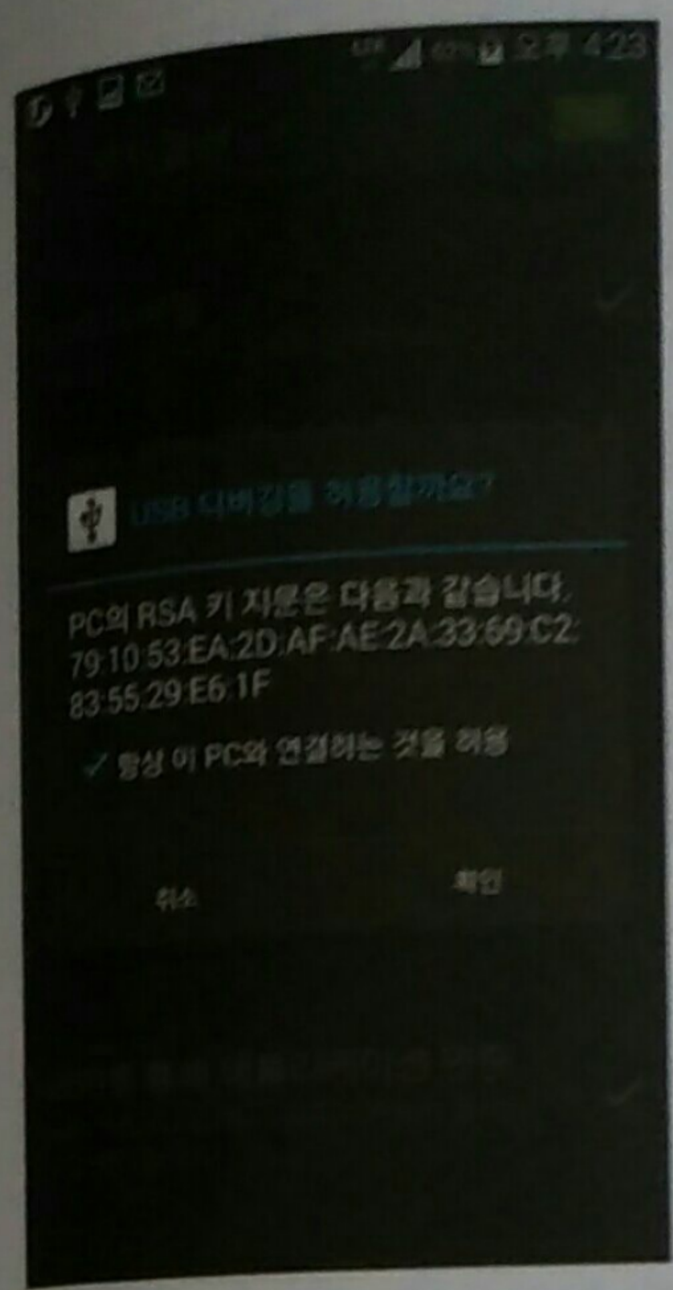
PC에 드라이버를 다운로드해서 설치한 후 USB 케이블로 단말과 연결하면 모바일 장치를 자동으로 인식합니다. 이제 단말의 설정을 바꾸어 보겠습니다. 여기에서는 삼성 갤럭시S5 단말을 예로 설명합니다. 단말의 설정 화면을 열고 아래쪽으로 스크롤하면 [디바이스 정보] 메뉴가 보입니다.



◀ 갤럭시S5 설정 화면의 [디바이스 정보] 메뉴

[디바이스 정보] 메뉴를 눌러 화면을 열면 단말의 정보가 보이는데 아래쪽으로 스크롤하여 빌드 번호 메뉴를 찾습니다. 이 빌드 번호 정보를 손가락으로 여러 번 터치하면 아래쪽에 작은 메시지가 차례대로 나타나는데 '[개발자 모드]를 실행하였습니다.'라는 메시지가 나올 때까지 터치합니다.

이제 설정 화면으로 돌아오면 [디바이스 정보] 메뉴 앞에 [개발자 옵션] 메뉴가 만들어집니다. [개발자 옵션] 메뉴를 눌러 [USB 디버깅] 항목을 찾아 터치합니다. 체크 상자가 체크 표시로 바뀌면서 'USB 디버깅을 허용할까요?'라는 대화상자가 나타납니다. [확인] 버튼을 누르면 다시 한 번 대화상자가 표시되는데 이번에는 현재 연결된 PC와의 USB 디버깅을 허용할 것인지 물어봅니다. 체크 상자를 체크한 후 [확인] 버튼을 누르면 단말 설정 과정이 끝납니다.

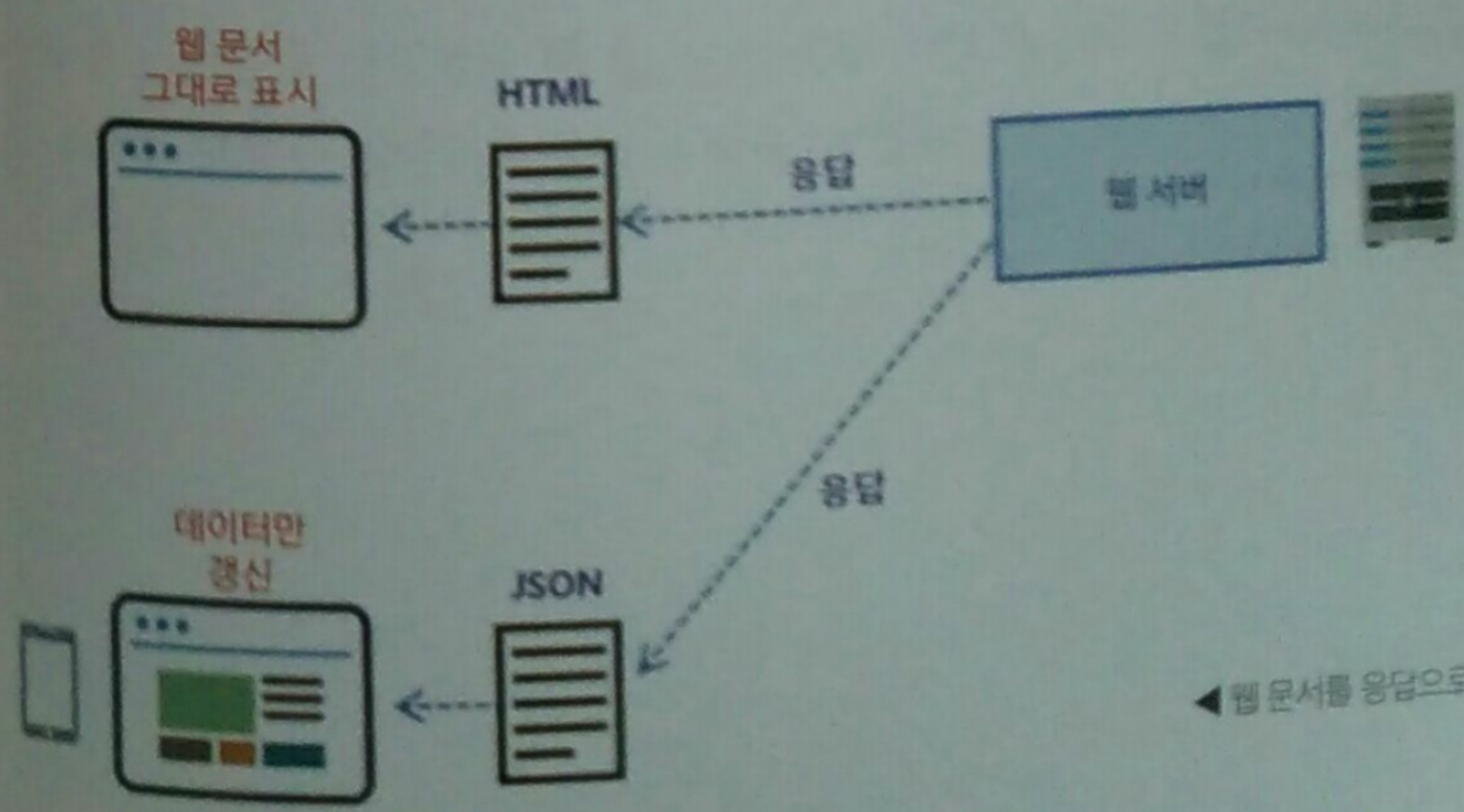


◀ USB 디버깅을 허용할 것인지 묻는 대화상자

이제 앱을 만들어 실제 단말에서 실행할 준비가 되었습니다.

모바일용 서버 프로젝트 만들기

모바일 단말에서 서버에 데이터를 요청하려면 서버 쪽에 모바일 관련 기능들을 추가해야 합니다. 모바일 단말에서도 HTTP를 사용해 웹 서버에 요청을 보낼 수 있으므로 앞에서 만든 웹 서버 모듈을 그대로 사용할 수 있습니다. 하지만 모바일 단말은 웹 서버에서 웹 문서를 받아 와 화면에 보여 주는 방식보다는 데이터만 받아 와 화면의 일부만 갱신하는 방식을 더 자주 사용합니다. 그래서 PC용 모듈과 모바일용 모듈을 분리하여 만들겠습니다. 좀 더 자세히 설명하면, 앞에서 모듈을 만들 때는 서버에서 클라이언트로 응답을 보낼 때 웹 문서를 보냈지만 모바일 단말이 클라이언트일 때는 데이터만 응답으로 보내 줍니다. 그러면 단말에서는 사용자가 보는 화면에서 데이터만 갱신합니다. 다음은 웹 문서를 응답으로 받는 경우와 데이터를 응답으로 받는 경우를 비교해서 보여 줍니다.



◀ 웹 문서를 응답으로 받는 경우와 데이터를 응답으로 받는 경우

PC에서도 웹 문서는 그대로 유지한 채 데이터만 받아 필요한 부분을 갱신하는 경우가 많이 있습니다. 예를 들어, 게시판의 글 목록에서 다음 페이지로 넘어가기 위해 사용자가 [다음 페이지] 링크를 누르면 예전에는 웹 서버에서 웹 문서 전체를 새로 받아 온 후 화면 전체를 갱신했습니다. 하지만 요즘에는 다음 페이지의 데이터만 받아 와 글 목록을 갱신하는 경우가 많아지고 있습니다. 이렇게 데이터만 갱신하는 경우의 장점은 응답 받는 데이터의 양이 적어 화면에 표시하는 속도가 빨라진다는 것입니다. 모바일에서는 주로 단말 OS에서 제공하는 API를 사용해 앱을 만들기 때문에 클라이언트 화면은 유지한 채 데이터만 받아 와 갱신하는 경우가 대부분입니다.

서버로부터 응답을 받을 때 데이터만 받고 싶다면 JSON 포맷으로 만든 데이터를 받도록 만들 수 있습니다. 예를 들어, 사용자 리스트를 데이터로 받고 싶다면 데이터베이스에서 사용자를 조회한 후 JSON 포맷의 데이터로 만들어 응답을 보낼 수 있습니다.

그러면 사용자 리스트를 조회해서 응답으로 보내 주는 모듈을 서버 쪽에 추가한 후 모바일 단말에서 요청하도록 만들어 보겠습니다. PassportExample2 프로젝트를 복사하여 새로운 프로젝트 MobileExample을 만듭니다. 이 프로젝트에는 패스포트 모듈을 사용해 사용자 로그인과 회원가입을 진행하는 기능이 이미 들어 있습니다. 먼저 사용자 리스트를 응답으로 보내 주는 라우팅 모듈을 추가합니다. [routes] 폴더에 user.js 파일을 만들고 다음처럼 listuser 함수를 추가합니다.

참조 파일 | MobileExample>/routes/user.js

```
var listuser = function(req, res) {
  var database = req.app.get('database');
  if(database) {
    database.UserModel.findAll(function(err, results) {
      if(err) {
        callback(err, null);
        return;
      }

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

        res.writeHead('200', {'Content-Type' : 'application/json; charset = utf8'});
        res.write(JSON.stringify(results));
        res.end();
      } else {

......

module.exports.listuser = listuser;

......
```

프로젝트 안에는 데이터베이스 스키마가 만들어져 있고 UserModel을 사용해 모델 객체에 접근할 수 있으므로 사용자 리스트를 가져오는 라우팅 함수는 어렵지 않게 추가할 수 있습니다.

UserModel의 findAll 함수를 호출하여 사용자 리스트를 받아 오면 res.writeHead, res.write, res.end 함수를 호출하여 클라이언트에게 응답을 보냅니다. writeHead 메소드를 사용해 헤더를 설정할 때는 application/json을 Content-Type 헤더 값으로 입력합니다. write 메소드를 호출할 때는 파라미터로 JSON 타입으로 된 문자열을 넣어 주어야 하므로 JSON.stringify 메소드를 호출하여 결과 객체를 문자열로 변경한 후 파라미터로 전달합니다.

이렇게 만든 라우팅 함수는 [config] 폴더의 config.js 파일에 등록합니다.

참조 파일 | MobileExample>/config/config.js

```
module.exports = {

.....

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

.....
```

이제 웹 브라우저에서 /process/listuser 패스로 요청하면 JSON 포맷으로 된 사용자 리스트를 받을 수 있습니다.

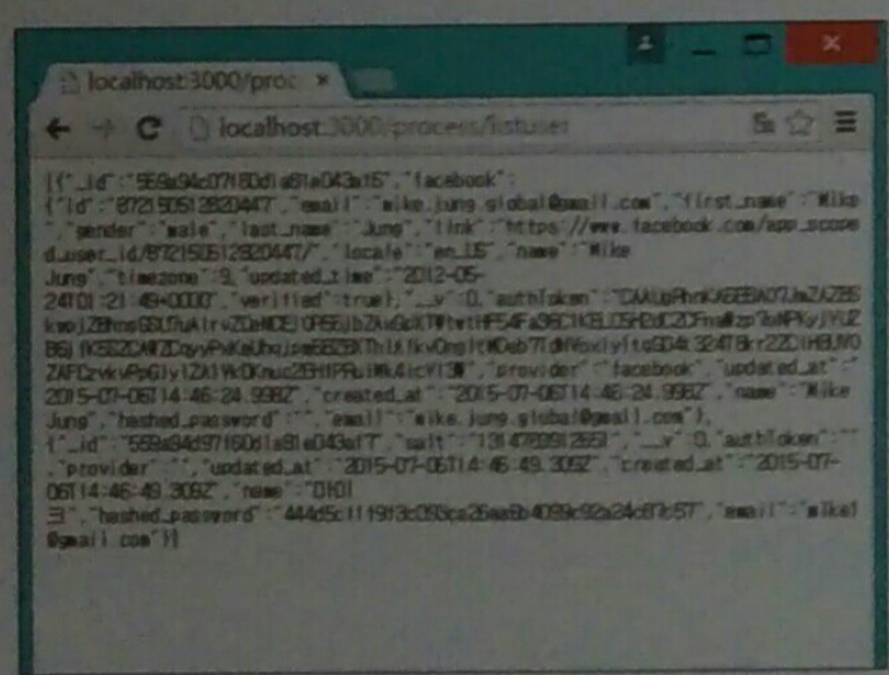
서버 쪽에 모듈을 만들어서 등록했으므로 이 기능을 요청할 클라이언트 웹 문서를 만들어 보겠습니다. ViewExample2 프로젝트의 [public] 폴더 안에 들어 있는 listuser.html 파일과 jQuery, Semantic UI 라이브러리 파일을 MobileExample 프로젝트의 [public] 폴더 안으로 복사합니다. listuser.html 파일에는 버튼을 클릭했을 때 /process/listuser 패스로 요청하는 코드가 들어 있으므로 여러분이 새로 추가한 모듈을 실행하고 응답을 받을 수 있습니다.

```
.....

  <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>

.....
```

app.js 파일을 실행한 후 웹 브라우저를 열고 다음 주소로 요청하면 [조회] 버튼이 들어 있는 웹 문서가 보입니다.

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

[조회] 버튼을 누르면 JSON 포맷으로 된 사용자 리스트의 데이터를 볼 수 있습니다.



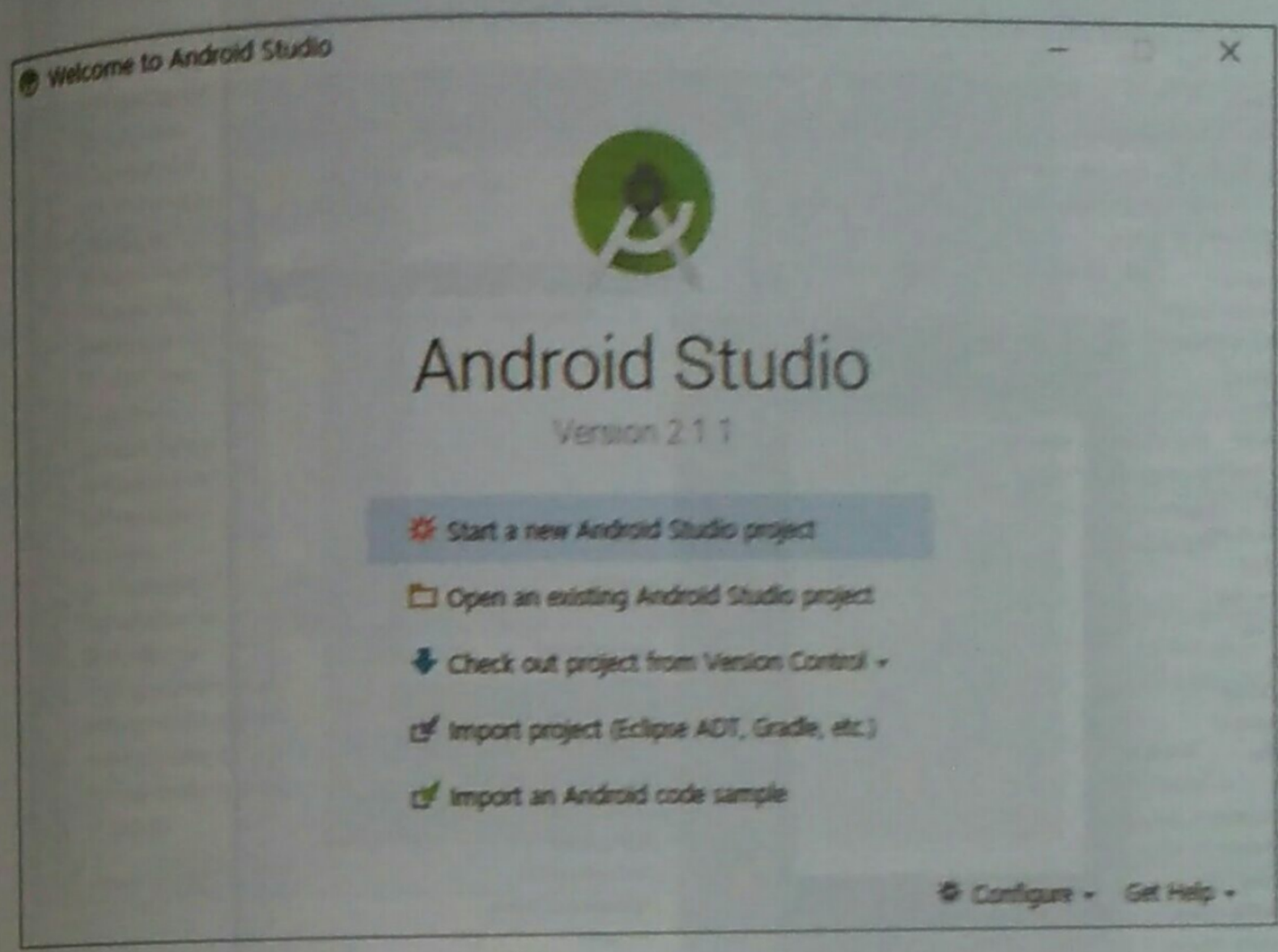
▲ 웹 브라우저에서 사용자 리스트를 요청했을 때 JSON 포맷의 데이터를 받음

응답으로 받은 데이터에는 _id, authToken, hashed_password 칼럼 등이 들어 있는데 이 정보는 제외할 필요가 있습니다. 그러나 여기에서는 테스트 용도로 사용할 것이므로 전체 데이터를 모두 응답받을 것입니다.

이제 안드로이드 앱을 만들어 사용자 리스트 데이터를 응답받을 차례입니다.

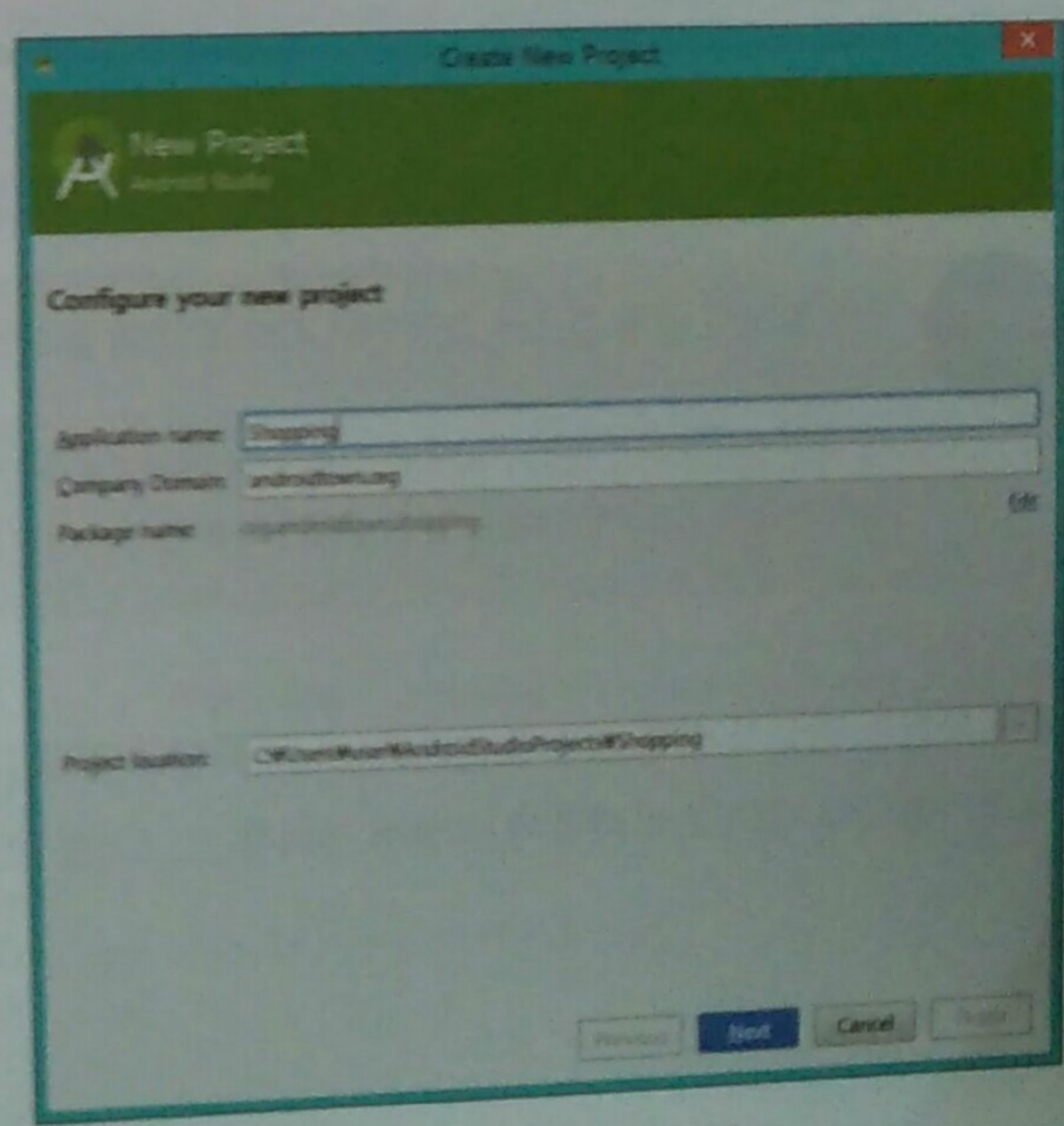
안드로이드 앱 프로젝트 만들기

안드로이드 앱은 앞에서 설치한 안드로이드 스튜디오로 만듭니다. 안드로이드 스튜디오를 실행하면 다음과 같은 시작 화면이 나타납니다.



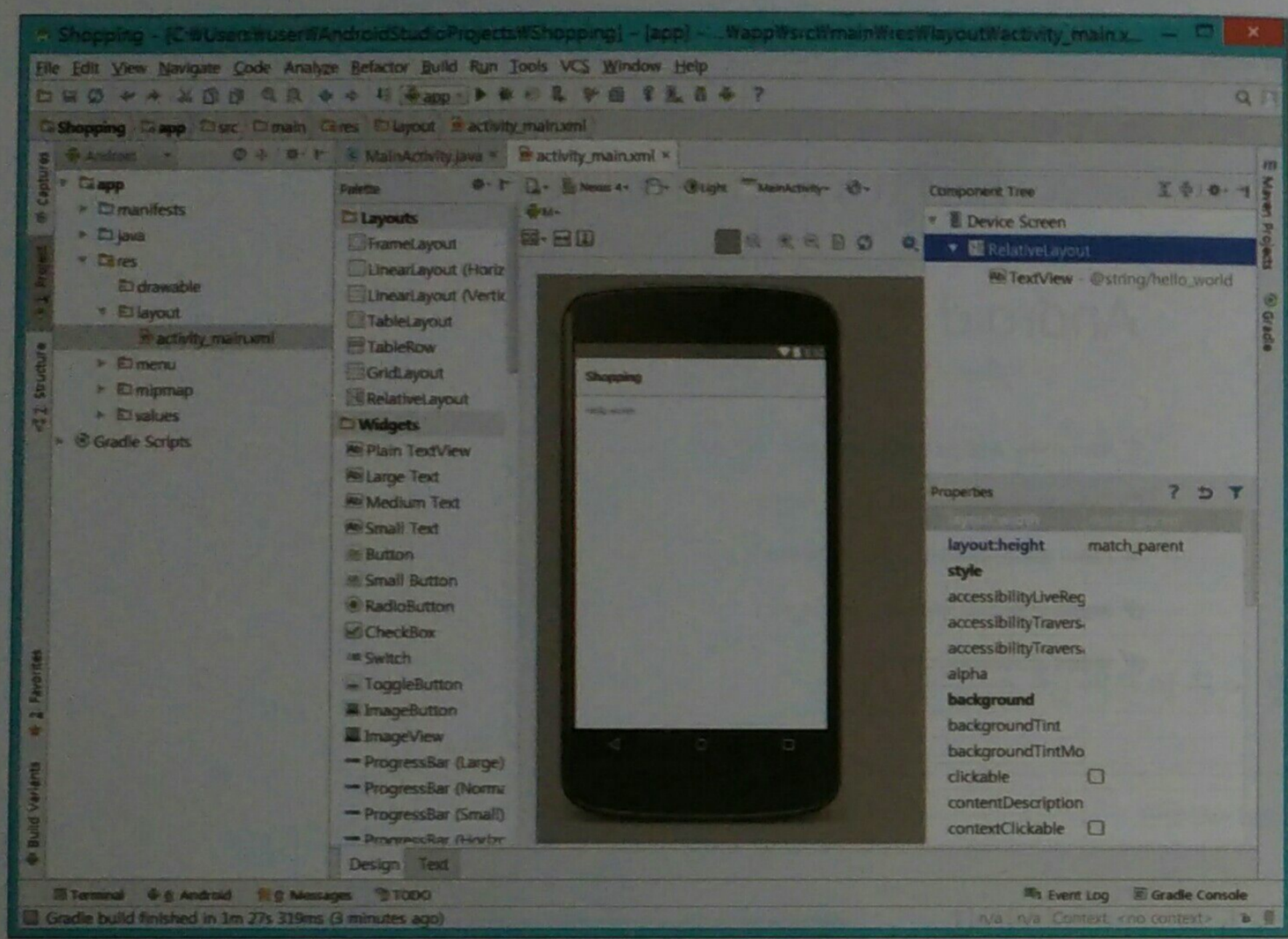
▲안드로이드 스튜디오의 시작 화면

이 화면의 메뉴에서 [Start a new Android Studio project] 메뉴를 눌러 새로운 프로젝트를 만들 수 있는 화면으로 넘어갑니다. 첫 번째 화면에는 앱의 이름을 입력합니다. Application name: 입력 상자에 Shopping을 입력하고 아래쪽에 있는 Company domain: 입력 상자에는 androidtown.org를 입력합니다. 그러면 그 아래에 있는 Package name: 입력 상자에는 자동으로 org.androidtown.shopping이 들어갑니다.



▲새로운 프로젝트를 만들 때 앱 이름을 입력하는 대화상자

[Next] 버튼을 계속 눌러 넘어가다가 마지막 화면에서 [Finish] 버튼을 누르면 자동으로 새로운 프로젝트가 만들어집니다. 지금부터는 안드로이드 앱 화면을 간단하게 바꾸고 여러분이 만든 웹 서버로 요청을 보내는 작업을 진행해 보겠습니다.

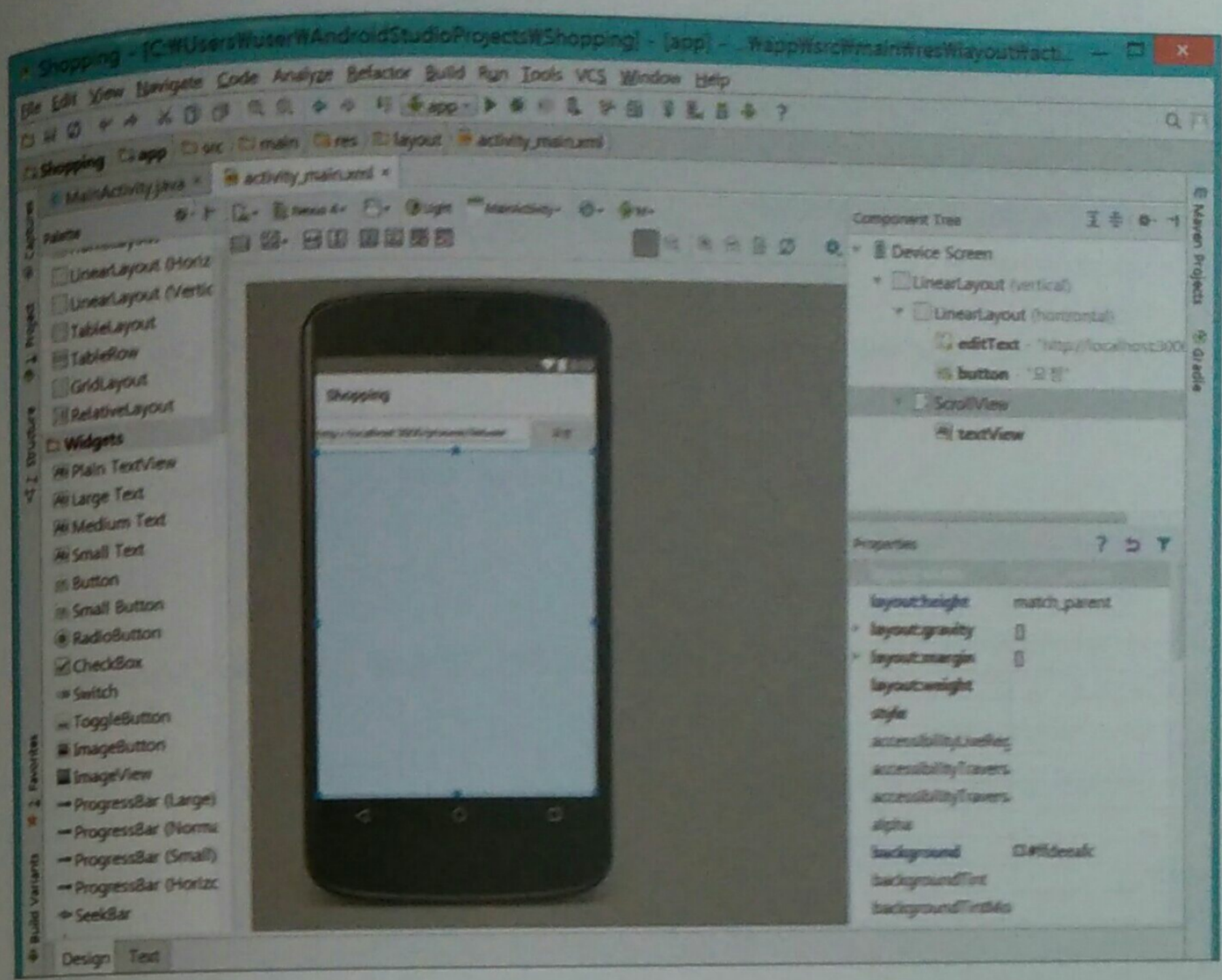


▲ 새로 만든 프로젝트 창

정박사의 한마디 안드로이드 앱을 만드는 과정이나 자바에 대해 잘 모르나요?

이 책은 안드로이드 앱이나 자바를 다룬 책이 아니므로 여기에서는 앱을 만드는 과정 중 핵심만 진행합니다. 따라서 아주 기초적인 내용을 알고 싶다면 다른 관련 서적의 도움을 받으시기 바랍니다.

먼저 화면을 만들기 위해 activity_main.xml 파일을 열고 다음 그림처럼 입력 상자 하나와 버튼을 위쪽에 추가합니다. 그리고 아래쪽에는 진행 상태를 로그처럼 보여 줄 수 있는 텍스트뷰를 하나 추가합니다.



▲ 새로 만드는 앱의 화면 구성

입력 상자에는 어떤 사이트로 요청을 보낼 것인지를 지정하는 서버 URL을 입력해 두었습니다. 그 아래쪽에 있는 **텍스트뷰(TextView)**는 스크롤이 자동으로 생길 수 있도록 **스크롤뷰(ScrollView)**로 감싸 주었는데 여기에 진행 상황을 로그처럼 출력할 것입니다. 버튼을 클릭했을 때 onButton1Clicked 메소드가 실행되도록 onClick 속성 값으로 onButton1Clicked를 입력합니다.

아래쪽에 있는 [Text] 탭을 누르면 원본 XML 코드를 볼 수 있는데, 이 화면의 XML 원본 코드는 다음과 같습니다.

참조 파일 | Shopping>/res/layout/activity_main.xml

```
.....
<LinearLayout xmlns : android = "http://schemas.android.com/apk/res/android
  android : layout_width = "match_parent"
  android : layout_height = "match_parent"
  android : orientation = "vertical" >

  <LinearLayout
    android : orientation = "horizontal"
    android : layout_width = "match_parent"
    android : layout_height = "wrap_content">
```

```
<EditText
  android : layout_width = "match_parent"
  android : layout_height = "wrap_content"
  android : layout_weight = "1"
  android : textSize = "15dp"
  android : text = "http://192.168.0.5:3000/process/listuser"
  android : id = "@+id/editText"/>

<Button
    android : layout_width = "match_parent"
    android : layout_height = "wrap_content"
    android : layout_weight = "3"
    android : text = "요청"
    android : id = "@+id/button"
    android : onClick = "onButton1Clicked"/>

</LinearLayout>

<ScrollView
  android : layout_width = "match_parent"
  android : layout_height = "match_parent"
  android : background = "#ffdeeafc">

  <TextView
    android : layout_width = "match_parent"
    android : layout_height = "match_parent"
    android : text = ""
    android : id = "@+id/textView"/>

</ScrollView>

</LinearLayout>

.....
```

입력 상자에 들어 있는 웹 서버의 주소는 localhost가 아니라 웹 서버가 실행된 실제 IP여야 합니다. 왜냐하면 실제 스마트폰 단말에서 앱을 실행했을 때는 무선 랜으로 서버에 접속해야 하는데 실제 서버의 IP를 입력해야 접속할 수 있기 때문입니다. PC의 IP를 확인하려면 PC에서 무선 랜에 접속한 후 명령 프롬프트 창을 열고 다음 명령을 입력합니다.

```
% ipconfig /all
```

주의! 가장 앞에 있는 '%' 기호는 명령 프롬프트에 표시된 기호를 나타낸 것이므로 명령 프롬프트 창에는 입력하지 말고 그 뒤에 있는 명령만 입력한 후 Enter 키를 누릅니다.

명령 실행 후 표시되는 결과 중에서 IPv4 항목에 IP 정보가 들어 있습니다. 이 IP 정보를 확인한 후 입력 상자에 해당 IP를 포함한 서버 URL을 넣어 줍니다. 예를 들어, 다음 주소를 입력할 수 있습니다.

```
▶ http://192.168.0.5:3000/process/listuser
```

안드로이드 앱을 만드는 방법이 익숙하지 않다면 이 부분은 간단하게 이해만 하고 넘어가도 좋습니다. 왜냐하면 앱을 직접 만들지 못하더라도 이 책에서 제공하는 샘플 앱에서 주소만 바꾼 후 실행하면 여러분이 만든 서버에 접속한 후 새로 추가한 라우팅 함수로 사용자 리스트 데이터를 응답받을 수 있기 때문입니다.

이제 [MainActivity.java] 탭을 열고 버튼을 클릭했을 때 처리하는 코드를 입력합니다.

참조 파일 | Shopping>/java/org.androidtown.shopping/MainActivity.java

```
.....

public class MainActivity extends AppCompatActivity {
  EditText editText;
  TextView textView;

  Handler handler = new Handler();

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    editText = (EditText) findViewById(R.id.editText);
    textView = (TextView) findViewById(R.id.textView);
  }

  public void onButton1Clicked(View v) {
    String url = editText.getText().toString();
  }

  public void println(final String data) {
    handler.post(new Runnable() {
      public void run() {
        textView.append(data + '\n');
```

```
        }
    });
    }
……
```

onCreate( ) 메소드 안에서는 XML 레이아웃 파일로 만든 화면의 입력 상자와 텍스트뷰를 findViewById( ) 메소드를 호출하여 찾아내고, id와 같은 이름으로 만든 변수에 할당합니다. println( ) 메소드는 진행 상태를 화면에 출력하기 위해 만든 것으로 텍스트뷰에 한 줄의 글자를 추가하여 보여 줍니다. 버튼을 클릭했을 때는 onButton1Clicked( ) 메소드가 호출되므로 그 메소드 안에서는 사용자가 입력 상자에 입력한 접속 주소 정보를 문자열로 가져옵니다.

이제 웹 서버에 요청하는 코드를 입력해야 합니다. 여기에서는 구글에서 만든 Volley 라이브러리를 사용합니다. 안드로이드 앱에서 HTTP 프로토콜로 웹 서버에 요청할 때는 HttpURLConnection을 사용하거나 okhttp 라이브러리를 사용하기도 합니다. Volley도 많이 사용하는 라이브러리이며 이 라이브러리를 사용하면 코드의 양을 줄일 수 있습니다. Volley 라이브러리와 함께 gson 라이브러리도 함께 사용할 수 있는데, gson 라이브러리는 JSON 포맷의 문자열 데이터를 자바 객체로 만들어 주거나 자바 객체를 JSON 포맷의 문자열로 만들어 줄 수 있습니다.

이 두 가지 라이브러리를 사용하려면 gradle 빌드 설정 파일에 두 줄의 정보를 추가해야 합니다. 안드로이드 스튜디오 화면의 왼쪽 프로젝트 영역에서 [Gradle Scripts] 폴더 안에 있는 build.gradle(Module:app) 파일을 더블클릭해서 연 다음 가장 아래쪽에 있는 dependencies 안에 다음처럼 두 줄의 코드를 추가합니다.

참조 파일 | Shopping>/Gradle Scripts/build.gradle (Module : app)

```
……

dependencies {
……

    compile 'com.android.volley:volley:1.0.0'
    compile 'com.google.code.gson:gson:2.3.1'}
……
```

두 줄의 가장 오른쪽에 보이는 숫자는 버전을 의미하는데 이 버전 정보는 라이브러리를 사용하는 시점에 따라 달라질 수 있습니다. 만약 여러분이 사용하는 시점에 위와 다른 최신 버전이 출시되었다면 이 숫자들만 바꾸어 주면 됩니다.

이렇게 두 줄을 추가한 후 안드로이드 스튜디오 화면 위쪽 아이콘 중에서 오른쪽에서부터 다섯 번째에 있는 [Sync Project...] 아이콘( )을 누릅니다. 그러면 안드로이드 스튜디오가 변경된 정보를 자동으로 인식합니다.

이제 다시 [MainActivity.java] 탭을 누른 후 onButton1Clicked 메소드 안에 다음 코드를 입력합니다.

참조 파일 | Shopping>/java/org.androidtown.shopping/MainActivity.java

```
.....
public void onButton1Clicked(View v) {
  String url = editText.getText().toString();

  StringRequest request = new StringRequest(Request.Method.POST, url,
    new Response.Listener<String>() {
      @Override        public void onResponse(String response) {
        try {
          println("onResponse() 호출됨 : " + response);
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    },
    new Response.ErrorListener() {
      @Override
      public void onErrorResponse(VolleyError error) {
        error.printStackTrace();
      }
    }
  ) {
    @Override
    protected Map<String, String> getParams() {
      Map<String, String>  params = new HashMap<>();

      return params;
    }
  };

  Volley.newRequestQueue(this).add(request);
  println("웹 서버에 요청함 : " + url);

  }

.....
```

서버에 요청한 후 문자열 타입으로 된 응답을 받기 위해 StringRequest 객체를 만들고 Volley 라이브러리의 RequestQueue에 넣어 줍니다. 웹 서버에 요청할 때 POST 방식을 사용하려면 Request.Method.POST 상수를 파라미터로 전달합니다.

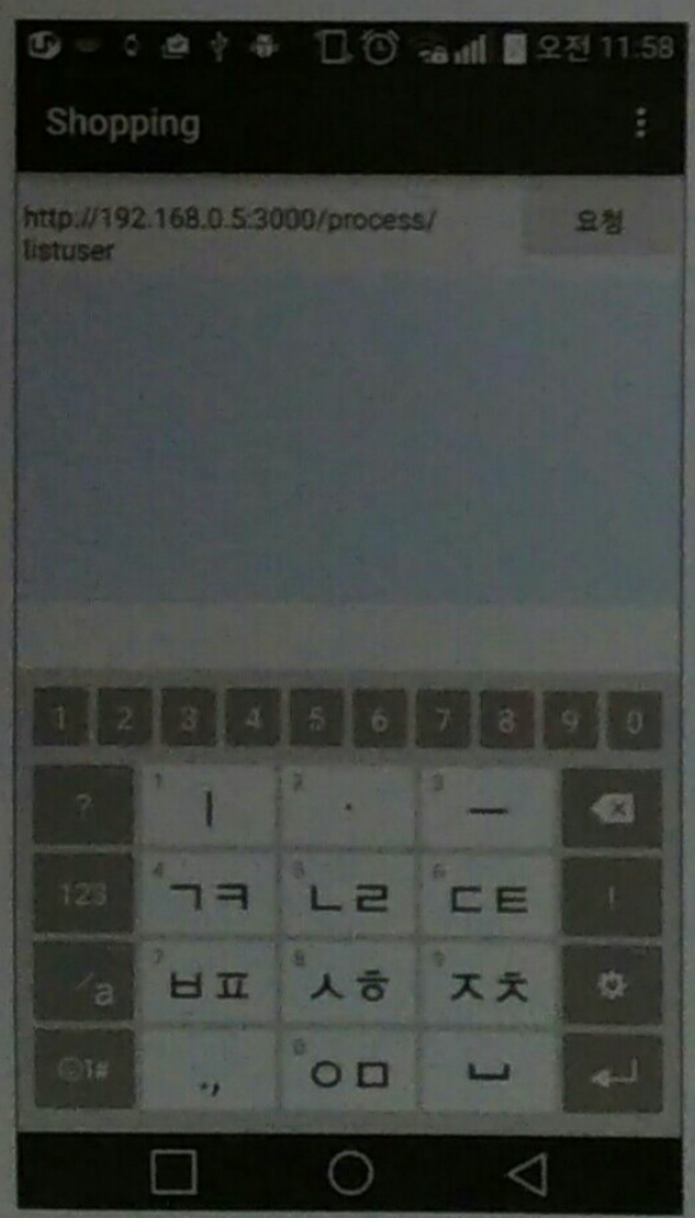
웹 서버로부터 응답을 받으면 Response.Listener 객체의 onResponse( ) 메소드가 호출되는데 그 안에서 응답 받은 문자열 데이터를 확인할 수 있습니다. 이렇게 전달 받은 데이터는 println( ) 메소드를 호출하여 화면에 표시합니다.

안드로이드 앱은 네트워킹 기능을 사용할 때 INTERNET 권한이 있어야 하므로 [manifests] 폴더 안에 있는 AndroidManifest.xml 파일을 열고 INTERNET 권한을 추가합니다.

참조 파일 | Shopping>/manifests/AndroidManifest.xml

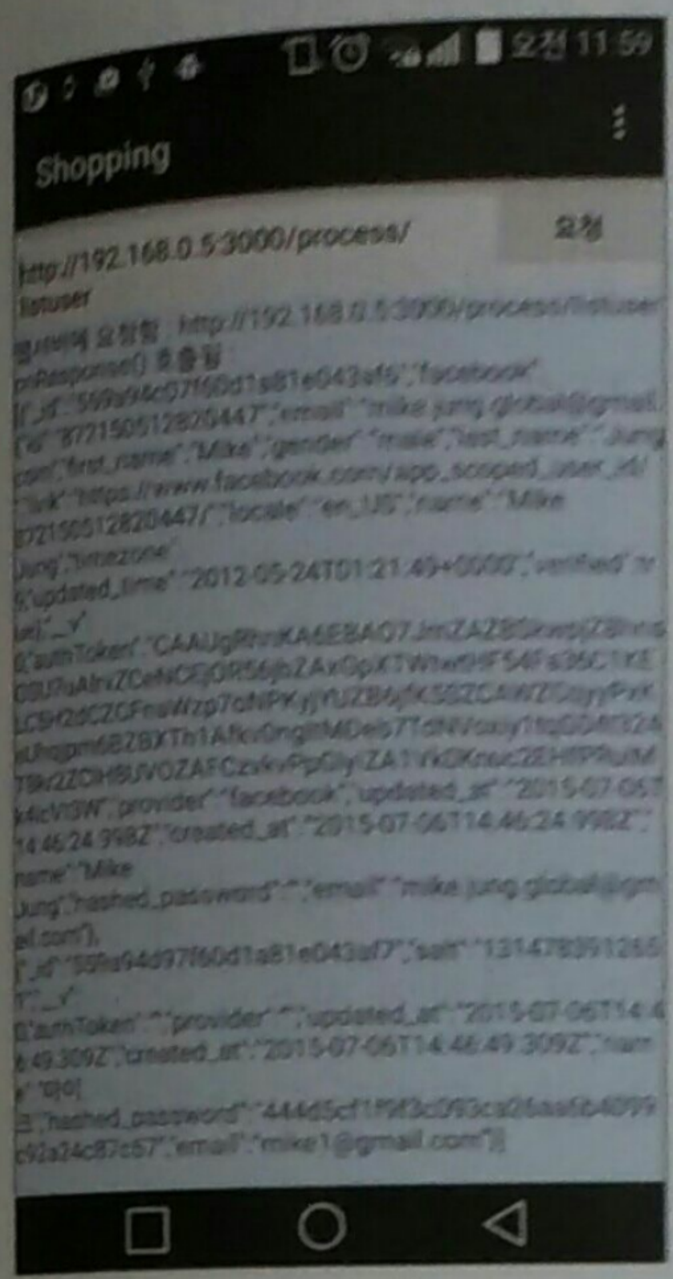
```
.....

<uses-permission android : name = "android.permission.INTERNET"/>

.....
```

앱을 실행할 때는 안드로이드 스튜디오 위쪽에 있는 실행 아이콘(▶)을 클릭합니다. 앱은 실제 단말도 실행하여 테스트해 보세요. PC를 연결한 무선 랜과 똑같은 무선 랜에 단말을 연결한 후 앱을 실행합니다.



▲ 실제 단말에서 앱을 실행한 화면

[요청] 버튼을 클릭하면 단말과 연결된 무선 랜을 통해 웹 서버에 접속한 후 결과를 가져와 화면에 나타냅니다.

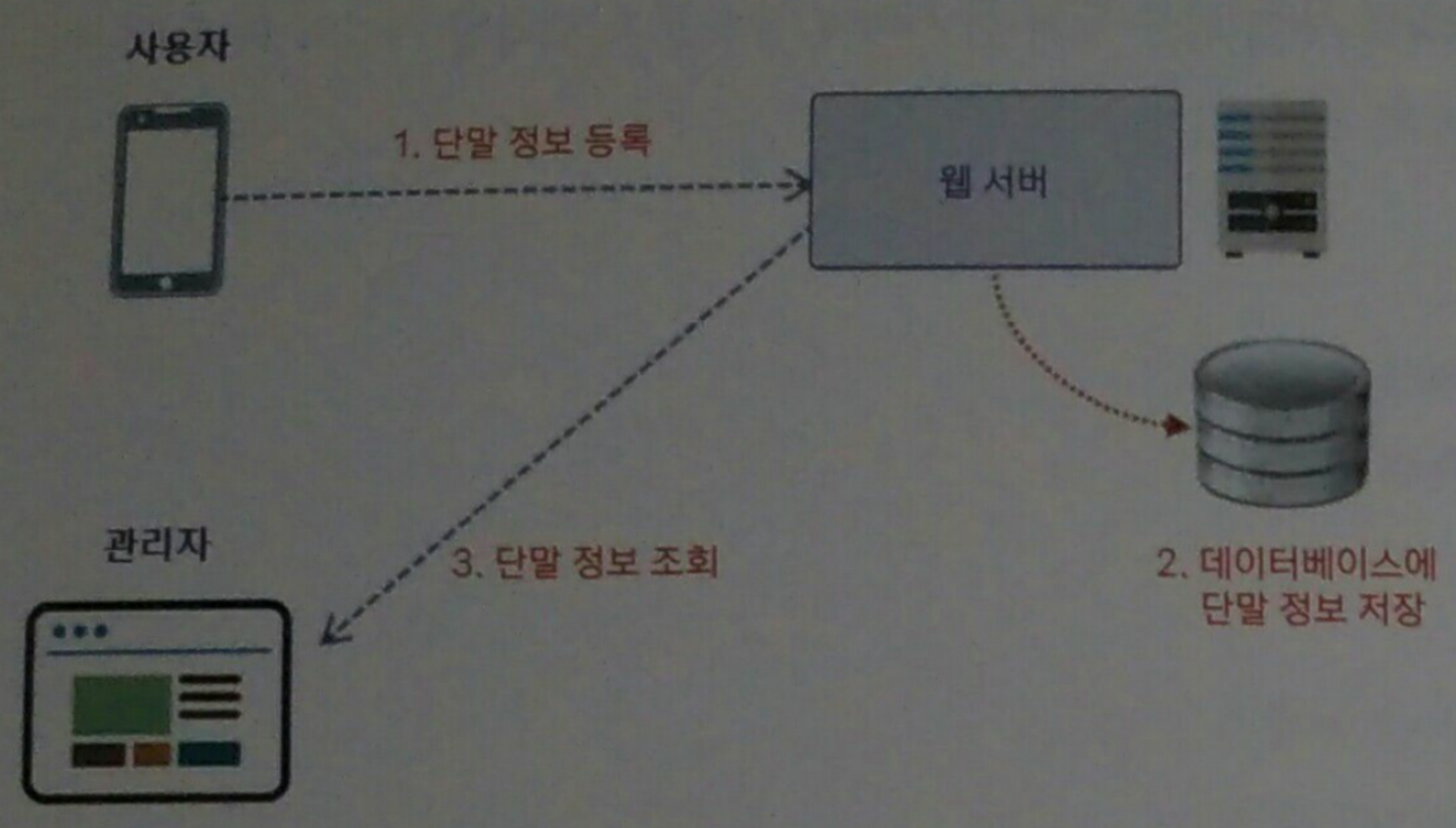


▲웹 서버에 요청한 후 응답을 받아 표시한 화면

웹 브라우저에서 본 JSON 포맷의 데이터가 똑같이 보이는 것을 알 수 있습니다. 이렇게 받은 데이터 중에서 필요한 사용자 정보만 뽑아서 화면에 예쁘게 보여 주는 작업은 여러분이 직접 해 보기 바랍니다.

13-2 | 모바일 단말 관리 기능 만들기

웹 서버에 라우팅 함수를 만들고 모바일 단말에서 데이터를 요청하는 방법에 대해 알아보았습니다. 이제 본격적으로 **모바일 단말 관리 기능**을 만들어 보겠습니다. 모바일 단말 관리 기능은 사용자의 단말을 서버에 등록해 두고 서버에 접근이 허용된 또 다른 단말만 접속을 허용하거나, 현재 사용하고 있는 단말의 상태를 한눈에 모니터링할 때 사용합니다. 따라서 단말 관리를 위해 만든 클라이언트 앱을 먼저 모바일 단말에 설치하고 그 앱으로 단말 정보를 확인하여 웹 서버에 등록하거나 갱신하도록 만들어야 합니다. 이렇게 등록한 단말 정보는 관리자가 웹 브라우저에서 조회하도록 만들 수 있습니다.



▲ 모바일 단말 관리 기능의 처리 과정

모바일 단말 관리 기능을 순서대로 살펴보겠습니다. (1) 사용자 모바일 단말에서 단말 정보를 웹 서버로 보내서 등록하면 (2) 그 정보를 웹 서버의 데이터베이스에 저장합니다. 이렇게 저장된 정보는 필요할 때 (3) 관리자가 웹 브라우저에서 단말 정보를 조회할 수 있습니다. 여기에서는 먼저 모바일 단말 관리 기능을 서버에 추가합니다.

데이터베이스 스키마 추가하기

먼저 웹 서버에서 단말 정보를 저장할 수 있도록 데이터베이스에 스키마를 추가합니다. [database] 폴더 안에 device_schema.js 파일을 만들고 다음 코드를 입력합니다.

참조 파일 | MobileExample>/database/device_schema.js

```
var Schema = { };

Schema.createSchema = function(mongoose) {

  // 스키마 정의
  var DeviceSchema = mongoose.Schema({
    mobile : {type : String, index : 'hashed', 'default' : ''}
    , osVersion : {type : String, 'default' : ''}
    , model : {type : String, 'default' : ''}
    , display : {type : String, 'default' : ''}
    , manufacturer : {type : String, 'default' : ''}
    , macAddress : {type : String, 'default' : ''}
    , registrationId : {type : String, 'default' : ''}
    , created_at : {type : Date, index : {unique : false}, 'default' : Date.now}
    , updated_at : {type : Date, index : {unique : false}, 'default' : Date.now}
  });
……
```

단말 정보를 저장할 때 키(Key)가 되는 칼럼의 이름은 mobile로 설정하고 이 칼럼에는 단말 전화번호를 저장합니다. 그 밖의 칼럼은 단말의 OS 버전, 모델, 디스플레이, 제조사, 맥 주소(MacAddress) 데이터를 저장하는 것입니다.

용어! 맥 주소(Mac Address)는 네트워크 어댑터(NIC)에 붙인 고유 식별자를 말합니다. 다시 말해, 모바일 단말이 인터넷을 사용하려면 네트워크 어댑터가 필요한데 이때 네트워크 어댑터에 고유한 값을 부여합니다. 따라서 서로 다른 단말을 구별할 때 사용하는 경우가 많습니다.

푸시 메시지를 보내는 데 사용하는 registrationId는 다음 단락에서 자세히 설명하도록 하고 여기에서 미리 추가합니다. DeviceSchema에 다음과 같이 몇 가지 메소드를 추가합니다.

참조 파일 | MobileExample>/database/device_schema.js

```
……

// 입력된 칼럼의 값이 있는지 확인
DeviceSchema.path('mobile').validate(function(mobile) {
  return mobile.length;
}, 'mobile 칼럼의 값이 없습니다.');

// 스키마에 static 메소드 추가
DeviceSchema.static('findByMobile', function(mobile, callback) {
  return this.find({mobile : mobile}, callback);
});

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

DeviceSchema.static('load', function(options, callback) {
  options.select = options.select || 'mobile';
  this.findOne(options.criteria)
  .select(options.select)
  .exec(callback);
});

// 모델에 스키마 등록
mongoose.model('Device', DeviceSchema);
console.log('DeviceSchema 정의함.');
return DeviceSchema;
};

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

DeviceSchema 객체의 path 메소드를 호출하여 mobile 칼럼 값이 있는지 확인하는 코드를 추가합니다. 그 아래에는 mobile 칼럼을 사용해 데이터를 조회하는 메소드와 모든 데이터를 조회하는 메소드를 추가합니다. 마지막으로 module.exports에 Schema 객체를 할당합니다.

이렇게 만든 스키마는 config.js 파일에 등록합니다.

참조 파일 | MobileExample>/config/config.js

```
module.exports = {

  ......

  db_schemas : [

  ......

    ,{file : './device_schema', collection : 'devices'
                        , schemaName : 'DeviceSchema', modelName : 'DeviceModel'}
  ],

  ......
```

db_schemas 배열에 추가한 객체를 보면 file, collection, schemaName, modelName 속성이 들어 있고 각각의 값으로 ./device_schema, devices, DeviceSchema, DeviceModel이 들어 있습니다. 따라서 device_schema.js 파일에서 만든 객체는 DeviceSchema 스키마 객체와 DeviceModel 모델 객체로 등록합니다. 데이터베이스에는 devices 컬렉션이 만들어집니다.

라우팅 함수 추가하기

데이터베이스 스키마를 추가했으니 두 개의 라우팅 함수를 추가합니다. 그중 하나는 모바일 단말의 정보를 클라이언트로부터 전송받아 데이터베이스에 추가하는 addDevice 함수입니다. 이 함수는 모바일 단말에서 요청할 때 실행되도록 추가한 것입니다. 또 다른 하나는 웹 브라우저에서 등록된 단말을 조회할 때 사용하는 listDevice 함수입니다.

[routes] 폴더에 device.js 파일을 만들고 다음과 같이 addDevice 함수를 추가합니다.

```
var adddevice = function(req, res) {
  console.log('device 모듈 안에 있는 adddevice 호출됨.');

  var database = req.app.get('database');
  var paramMobile = req.param('mobile');
  var paramOsVersion = req.param('osVersion');
  var paramModel = req.param('model');
  var paramDisplay = req.param('display');
  var paramManufacturer = req.param('manufacturer');
  var paramMacAddress = req.param('macAddress');

  if (database) {
    // DeviceModel 인스턴스 생성
    var device = new database.DeviceModel({"mobile" : paramMobile
                    , "osVersion" : paramOsVersion
                    , "model" : paramModel
                    , "display" : paramDisplay
                    , "manufacturer" : paramManufacturer
                    , "macAddress" : paramMacAddress
                  });

    // save( )로 저장
    device.save(function(err) {
      if (err) {throw err;}
      console.log("단말 데이터 추가함.");
      console.dir(device);
      res.writeHead('200', {'Content-Type' : 'application/json;charset=utf8'});
      res.write("{code : '200', 'message' : '단말 데이터 추가 성공'}");
      res.end();
    });

……
```

addDevice 함수에서 클라이언트가 보낸 데이터를 확인한 후 DeviceModel 객체의 인스턴스를 만들고 save 메소드를 호출하여 단말 정보를 데이터베이스에 저장합니다. 데이터를 성공적으로 저장했다면 성공 메시지를 JSON 객체로 만든 후 클라이언트에 응답으로 보냅니다.

addDevice 함수 아래에는 다음과 같이 listDevice 함수도 추가합니다.

```
……

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

  var database = req.app.get('database');
    if (database) {
      // 1. 모든 사용자 검색
      database.DeviceModel.findAll(function(err, results) {
        if (err) {
          callback(err, null);
          return;
        }

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

          var context = {
            title : '단말 목록',
            devices : results
          };
          req.app.render('listdevice', context, function(err, html) {
            res.end(html);
          });

……

module.exports.adddevice = adddevice;
module.exports.listdevice = listdevice;
```

단말 목록을 조회할 때는 DeviceModel 객체에 정의되어 있는 findAll 메소드를 호출합니다. 조회한 단말 목록은 listdevice.ejs 뷰 템플릿을 사용해 웹 문서로 만든 후 클라이언트에 응답으로 보냅니다. device.js 파일의 마지막 부분에는 module.exports의 속성으로 adddevice와 listdevice 함수를 추가하는 코드가 들어 있습니다.

라우팅 함수를 추가했으니 config.js 파일에 새로 추가한 함수들을 등록합니다.

참조 파일 | MobileExample>/config/config.js

```
module.exports = {

……,

  route_info : [

……,

    ,{file : './device', path : '/process/adddevice', method : 'adddevice', type : 'post'}
    ,{file : './device', path : '/process/listdevice', method : 'listdevice', type : 'post'}
  ],

……,
```

device.js파일에 추가한 adddevice 함수는 POST 방식으로 요청할 수 있으며 요청 패스는 /process/adddevice입니다. listdevice 함수도 POST 방식으로 요청할 수 있으며 요청 패스는 /process/listdevice입니다.

사용자에게 응답을 보낼 때 사용할 뷰 템플릿은 [views] 폴더에 listdevice.ejs라는 이름으로 만듭니다. listdevice.ejs 파일의 내용은 다음 코드와 같습니다.

참조 파일 | MobileExample>/views/listdevice.ejs

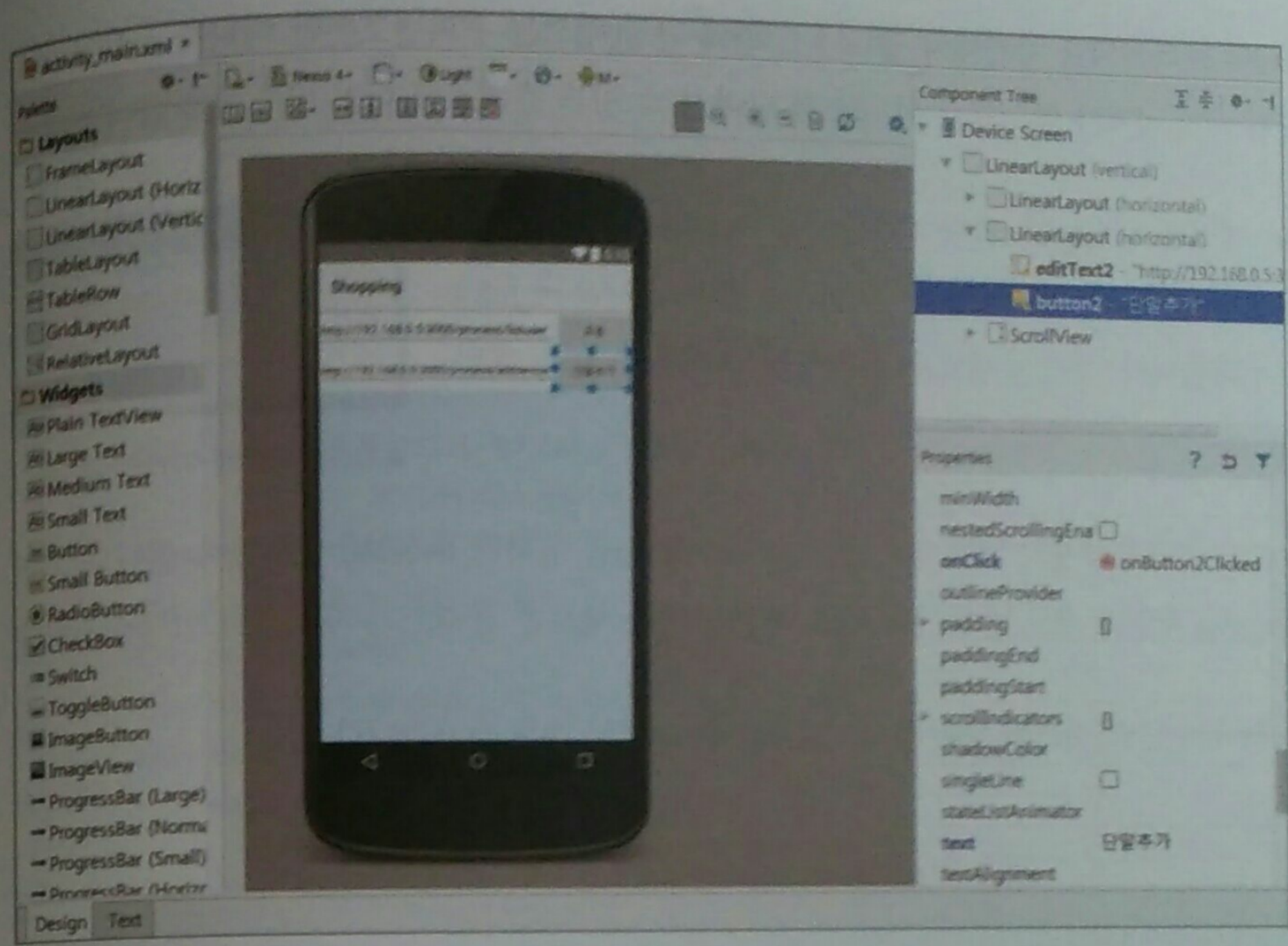
```
……,

  <div class="ui very relaxed selection celled list">
    <%
      for (var i = 0; i < devices.length; i++) {
        var curMobile = devices[i]._doc.mobile;
        var curOsVersion = devices[i]._doc.osVersion;
        var curModel = devices[i]._doc.model;
        var curDisplay = devices[i]._doc.display;
        var curManufacturer = devices[i]._doc.manufacturer;
        var curMacAddress = devices[i]._doc.macAddress;
        var curNo = devices.length - i;
    %>
    <div class = "item">
      <div class = "ui grid">
        <div class = "two wide column"><% = curNo %></div>
        <div class = "fourteen wide column"
```

```
        onclick = "javascript : window.location = '/process/listdevice">
    <div class = "ui header">
      <h4 class = "ui left aligned header">
        <% = curMobile %>
      </h4>
      <h5 class = "ui right aligned orange header">
        <% = curOsVersion %>
        <% = curModel %>
        <% = curDisplay %>
        <% = curManufacturer %>
        <% = curMacAddress %>
      </h5>
    </div>
    </div>
  </div>
  <% } %>

</div>

.....
```

이 뷰 템플릿은 단말 목록을 보여 주는 웹 문서를 만들 때 사용하며, for문으로 devices 변수에 들어 있는 단말 정보를 하나씩 확인하여 웹 문서를 만들어 줍니다.

단말 정보 추가를 요청하는 앱 만들기

서버 쪽에 필요한 기능을 추가했으니 이제 안드로이드 앱을 만들어야 합니다. 먼저 앞에서 만든 안드로이드 스튜디오 창을 열고 서버에 단말 정보 추가를 요청하기 위해 필요한 화면을 만들어 보겠습니다. activity_main.xml 파일을 열고 다음과 같이 입력 상자와 버튼을 하나씩 더 추가합니다. 새로 추가한 버튼에는 [단말 추가]라는 글자를 표시하고 버튼을 클릭했을 때 호출되는 메소드의 이름은 onClick 속성 값인 onButton2Clicked로 입력합니다.



▲ 앱의 화면에 입력 상자 하나와 버튼 하나를 더 추가한 경우

화면에 버튼을 추가했으니 MainActivity.java 파일을 열고 버튼을 클릭했을 때 호출될 onButton2Clicked 메소드를 추가합니다. 버튼을 클릭했을 때는 입력 상자에 입력된 URL로 단말 정보를 추가할 수 있도록 POST 요청을 보낼 것입니다.

참조 파일 | Shopping2)/java/org.androidtown.shopping/MainActivity.java

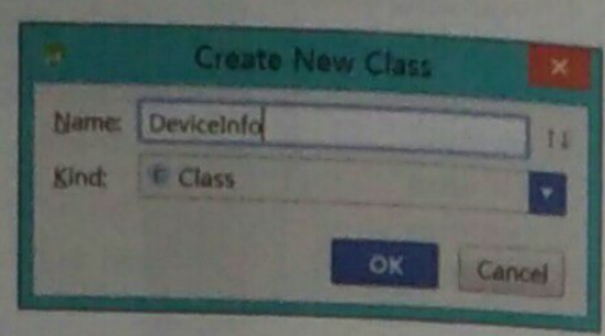
```
……
public void onButton2Clicked(View v) {
  String url = editText2.getText().toString();
  StringRequest request = new StringRequest(Request.Method.POST, url,

……
```

onButton2Clicked 메소드 안에 들어가는 코드는 onButton1Clicked 메소드 안에 들어가는 코드와 같으니 복사하여 붙여넣기를 한 후 수정합니다. 수정할 곳은 두 군데인데 그중 첫 번째는 사용자가 입력한 URL 정보를 가져오는 부분입니다. editText 입력 상자가 아니라 editText2 입력 상자에 입력한 글자를 가져오도록 수정합니다.

두 번째는 웹 서버로 요청할 때 넣어 줄 데이터들인데 단말의 전화번호 등의 정보를 가져와서 파라미터로 추가해야 합니다. 일단 단말 정보를 가져온 후 Volley 라이브러리로 새로운 요청을 보낼 때 사용

하는 Map 객체에 파라미터를 추가해야 합니다. 그런데 단말 정보는 전화번호뿐만 아니라 osVersion, macAddress 등 여러 가지가 있으므로 이 정보를 담아둘 수 있는 클래스를 먼저 만들어야 합니다.

왼쪽 프로젝트 영역에서 [java] 폴더 안에 있는 [org.androidtown.shopping] 폴더를 선택한 후 마우스 오른쪽 버튼을 눌러 [New → Java Class] 메뉴를 선택합니다. 그러면 새로운 클래스를 추가할 수 있는 대화상자가 나타납니다.



▲ 새로운 클래스를 추가하는 대화상자

[Create New Class] 대화상자의 입력 상자에는 DeviceInfo를 입력한 후 [OK] 버튼을 누릅니다. 그러면 DeviceInfo.java 파일이 새로 만들어지고 그 파일 안에는 새로운 클래스가 자동으로 만들어집니다.

이 클래스에는 하나의 단말 정보를 넣어 둘 수 있도록 변수 여섯 개를 추가합니다.

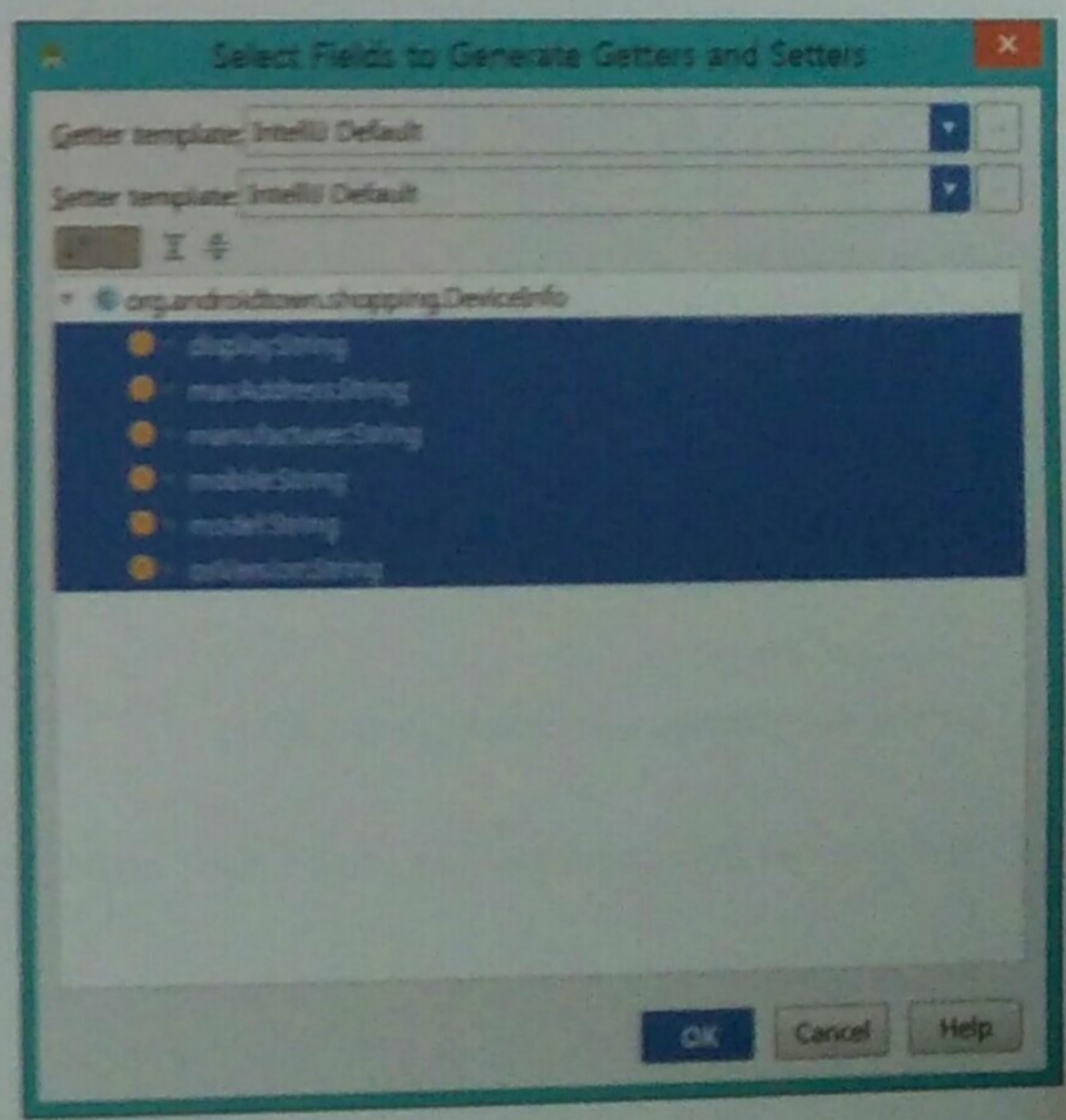
참조 파일 | Shopping2>/java/org.androidtown.shopping/DeviceInfo.java

```
.....
  public class DeviceInfo {

    String mobile;
    String osVersion;
    String model;
    String display;
    String manufacturer;
    String macAddress;

.....
```

그다음에는 새로운 인스턴스 객체를 만들 때 사용하도록 DeviceInfo 클래스로 생성자를 추가합니다. 여섯 개의 변수에 들어갈 데이터를 모두 파라미터로 전달할 수 있도록 생성자 메소드를 다음과 같이 만듭니다.

```
……

public class DeviceInfo {

……

    public DeviceInfo(String mobile, String osVersion, String model, String display,
                      String manufacturer, String macAddress) {
        this.mobile = mobile;
        this.osVersion = osVersion;
        this.model = model;
        this.display = display;
        this.manufacturer = manufacturer;
        this.macAddress = macAddress;
    }
……
```

이제 DeviceInfo 클래스로 새로운 인스턴스 객체를 만들 수 있습니다. 하지만 변수에 접근할 때 Get 메소드와 Set 메소드를 사용하도록 만드는 것이 좋기 때문에 Get, Set 메소드를 자동으로 추가합니다. 클래스를 만들 중괄호 안에 커서를 두고 마우스 오른쪽 버튼을 눌러 [Generate...] 버튼을 찾아 누릅니다. 다른 하위 메뉴들이 보이면 그중에서 [Getter and Setter] 메뉴를 누릅니다. 그러면 다음과 같이 Get과 Set 메소드를 만들 변수를 선택하는 대화상자가 나타납니다.



▲ Get과 Set 메소드를 추가하는 대화상자

모든 변수를 선택한 후 [OK] 버튼을 누르면 자동으로 메소드들이 추가됩니다. 이제 단말 정보를 넣어 둘 수 있는 DeviceInfo 클래스를 만들었습니다. 그러면 이 클래스를 사용해 인스턴스 객체를 만든 다음 그 인스턴스 객체에 단말 정보를 채워 주는 getDeviceInfo 메소드를 MainActivity.java 파일에 추가합니다.

MainActivity.java 파일을 열고 다음 코드를 입력합니다.

참조 파일 | Shopping2>/java/org.androidtown.shopping/MainActivity.java

```
.....

public DeviceInfo getDeviceInfo() {
  DeviceInfo device = null;

  // 1. mobile
  String mobile = null;
  TelephonyManager telephonyManager =
              (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
  if(telephonyManager.getLine1Number() != null ) {
    mobile = telephonyManager.getLine1Number();
  }

  // 2. osVersion
  String osVersion = Build.VERSION.RELEASE;

  // 3. model
  String model = Build.MODEL;

  // 4. display
  String display = getDisplay(this);

  // 5. manufacturer
  String manufacturer = Build.MANUFACTURER;

  // 6. macAddress
  String macAddress = getMacAddress(this);

  device = new DeviceInfo(mobile, osVersion, model, display, manufacturer, macAddress);

  return device;
}

.....
```

단말 정보를 가져오는 코드는 그리 복잡하지 않습니다. 단말 전화번호는 TelephonyManager를 사용해서 가져올 수 있으며, osVersion이나 model 정보는 Build 클래스의 클래스 변수에 저장되어 있습니다. 단말의 해상도를 알아내는 기능과 맥 주소를 알아내는 기능은 몇 줄의 코드가 더 필요합니다. 따라서 이 두 가지 정보를 알아내는 기능은 별도의 메소드로 정의한 후 호출하여 사용합니다.

단말의 해상도를 알아내려면 다음처럼 getDisplay 메소드를 만듭니다. 단말의 해상도를 알아낼 때는 WindowManager를 사용합니다.

참조 파일 | Shopping2>/java/org.androidtown.shopping/MainActivity.java

```
……

private static String getDisplay(Context context) {
  DisplayMetrics displayMetrics = new DisplayMetrics();
  WindowManager windowManager = (WindowManager)
                    context.getSystemService(Context.WINDOW_SERVICE);
  windowManager.getDefaultDisplay().getMetrics(displayMetrics);
  int deviceWidth = displayMetrics.widthPixels;
  int deviceHeight = displayMetrics.heightPixels;

  return deviceWidth + "x" + deviceHeight;
}

……
```

맥 주소를 알아내려면 getMacAddress 메소드를 다음처럼 만듭니다. 단말의 맥 주소를 알아낼 때는 WifiManager를 사용합니다.

참조 파일 | Shopping2>/java/org.androidtown.shopping/MainActivity.java

```
……

private static String getMacAddress(Context context) {
  WifiManager wifiManager = (WifiManager)
                    context.getSystemService(Context.WIFI_SERVICE);
  WifiInfo info = wifiManager.getConnectionInfo();

  return info.getMacAddress();
}

……
```

이렇게 알아낸 단말 정보는 getDeviceInfo 메소드를 호출했을 때 DeviceInfo 객체에 넣은 후 반환됩니다. 그렇기 때문에 Volley 라이브러리를 사용해 웹 서버로 단말 추가를 요청할 때는 DeviceInfo 객체에 들어 있는 데이터를 사용하면 됩니다.

참조 파일 | Shopping2>/java/org.androidtown.shopping/MainActivity.java

```
……

public void onButton2Clicked(View v) {
  String url = editText2.getText().toString();

  StringRequest request = new StringRequest(Request.Method.POST, url,

……
        @Override
        protected Map<String, String> getParams() {
            Map<String, String>  params = new HashMap<>();

            DeviceInfo device = getDeviceInfo();
            params.put("mobile", device.getMobile());
            params.put("osVersion", device.getOsVersion());
            params.put("model", device.getModel());
            params.put("display", device.getDisplay());
            params.put("manufacturer", device.getManufacturer());
            params.put("macAddress", device.getMacAddress());

            return params;
        }
    };

    Volley.newRequestQueue(this).add(request);
    println("웹 서버에 요청함 : " + url);
}

……
```

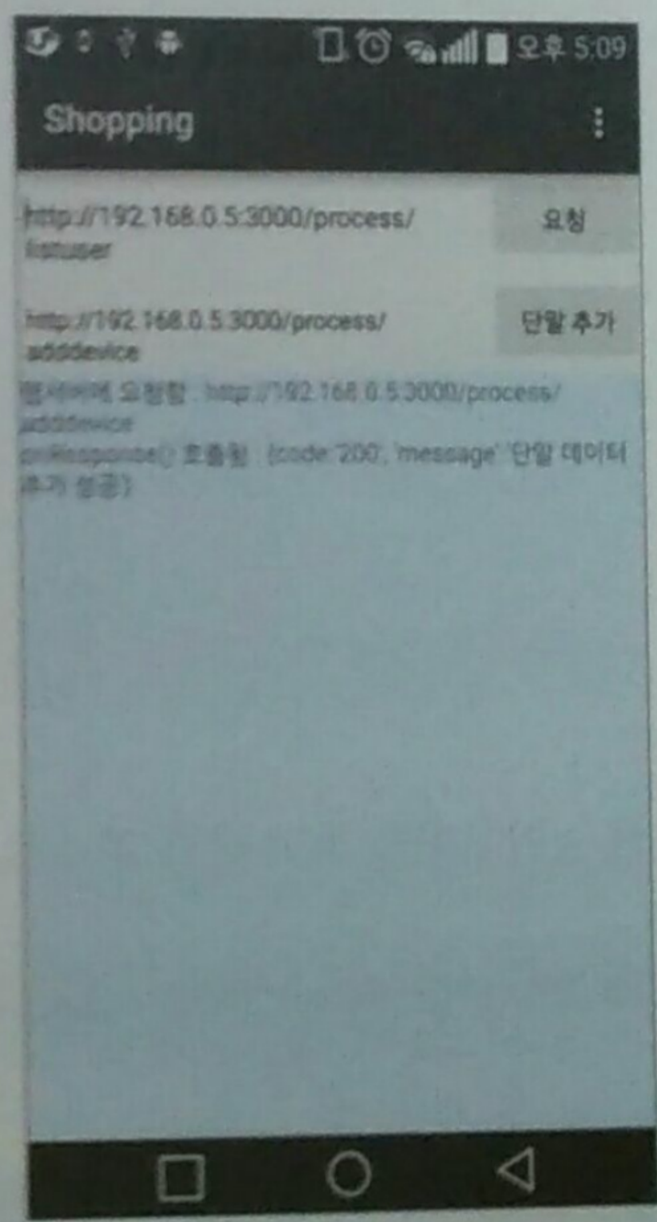
두 번째 버튼을 클릭했을 때 실행되는 Volley 요청에는 getParams 메소드가 정의되어 있는데 이 메소드 안에서 getDeviceInfo 메소드를 호출합니다. getDeviceInfo 메소드를 호출하여 알아낸 단말 정보는 params 변수에 추가됩니다.

이제 마지막으로 [manifests] 폴더의 AndroidManifest.xml 파일을 열고 다음 두 가지 권한을 추가합니다.

참조 파일 | Shopping2>/manifests/AndroidManifest.xml

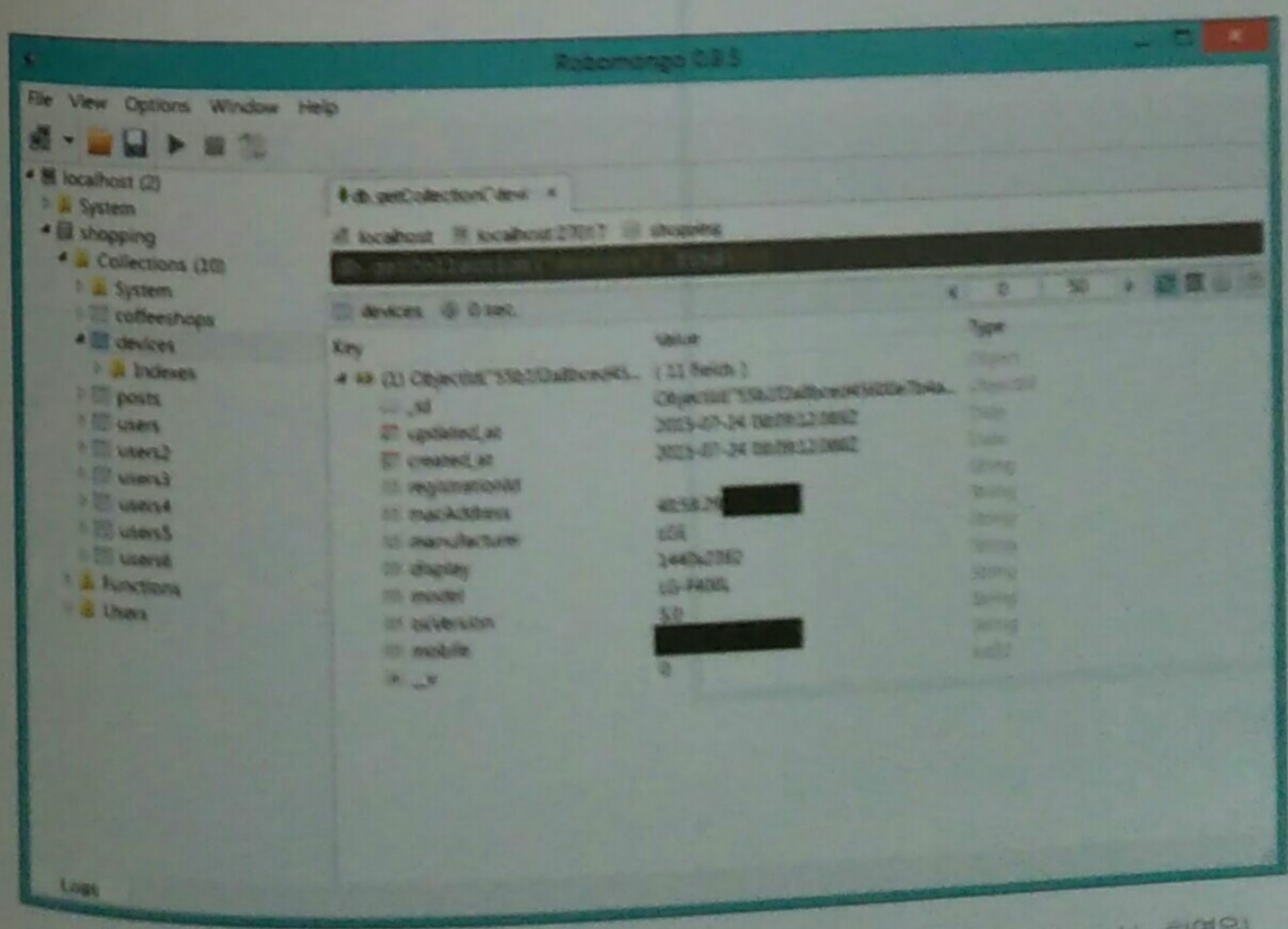
```
……,

<uses-permission android : name = "android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android : name = "android.permission.READ_PHONE_STATE"/>

……,
```

이제 단말이 무선 랜에 접속된 상태로 앱을 실행한 후 [단말 추가] 버튼을 클릭합니다. 그러면 웹 서버에 데이터를 보내서 단말 정보를 추가하며, 앱의 화면에는 정상적으로 처리되었다는 응답 메시지가 나타납니다.



[단말 추가] 버튼을 누른 후 정상적인 응답을 받은 화면▶

웹 서버에서 정상적으로 처리했다는 응답을 받았으므로 Robomongo를 사용해 shopping 데이터베이스의 devices 컬렉션을 조회하면 새로 추가된 데이터를 확인할 수 있습니다.



▲ [단말 추가] 버튼을 누른 후 데이터베이스에 저장된 데이터(전화번호는 검은색으로 masking하였음)

이제 모바일 단말에서 동작하는 앱에서 웹 서버에 단말 정보를 보내 서버에 데이터를 저장할 수 있게 되었습니다.

웹 브라우저에서 단말 리스트 조회하기

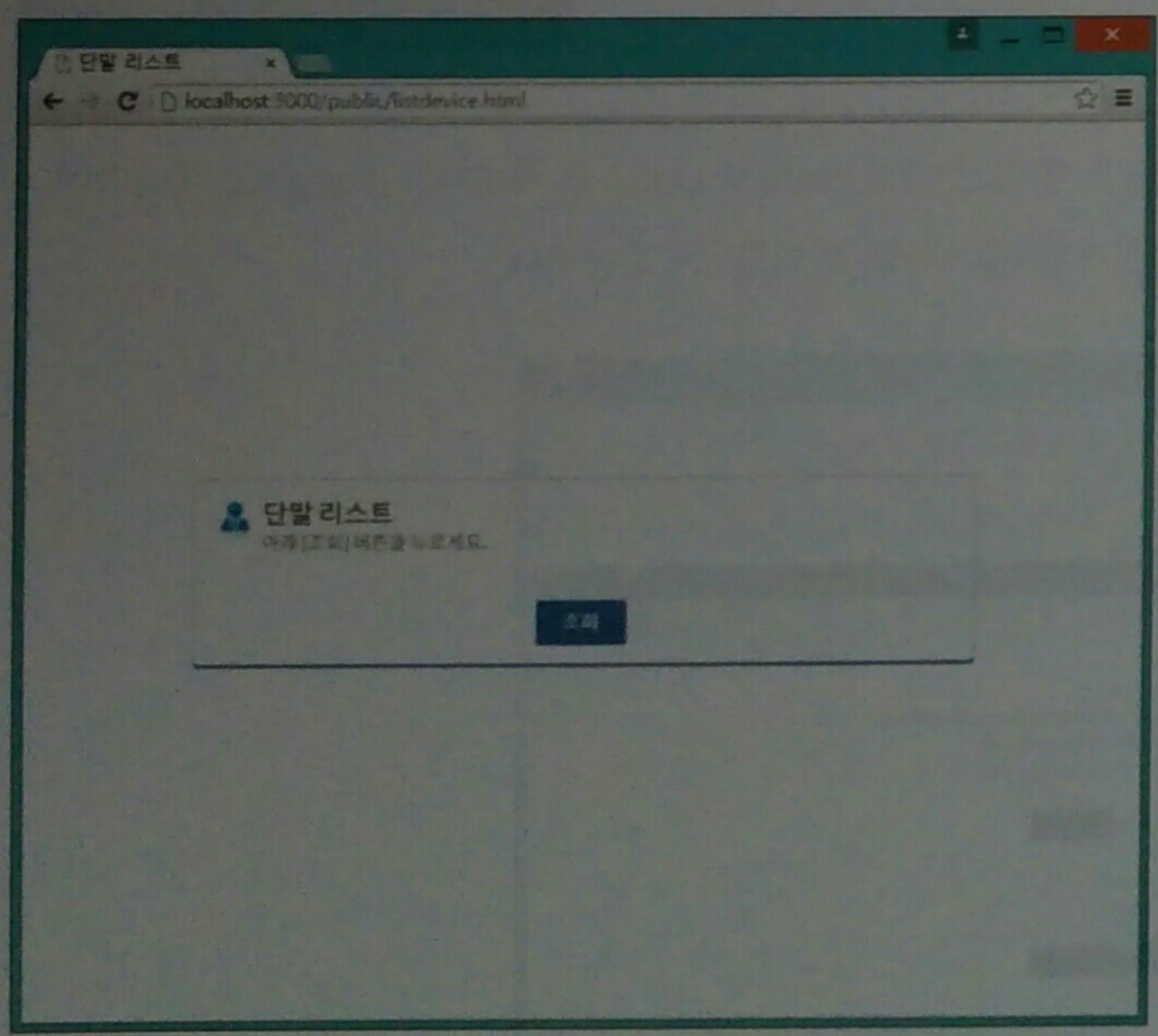
데이터베이스에 저장된 단말 정보 데이터는 웹 브라우저에서 조회할 수 있습니다. 단말 데이터 조회를 요청하기 위해 먼저 [public] 폴더에 있는 listuser.html 파일을 복사하여 listdevice.html 파일을 만듭니다. 이 웹 문서는 단순히 [조회] 버튼만 보이도록 만든 것으로, 타이틀 글자와 버튼을 클릭했을 때 요청하는 URL만 변경합니다.

참조 파일 | MobileExample>/public/listdevice.html

```
……
  <form id = "form1" method = "post" action = "/process/listdevice">

……
```

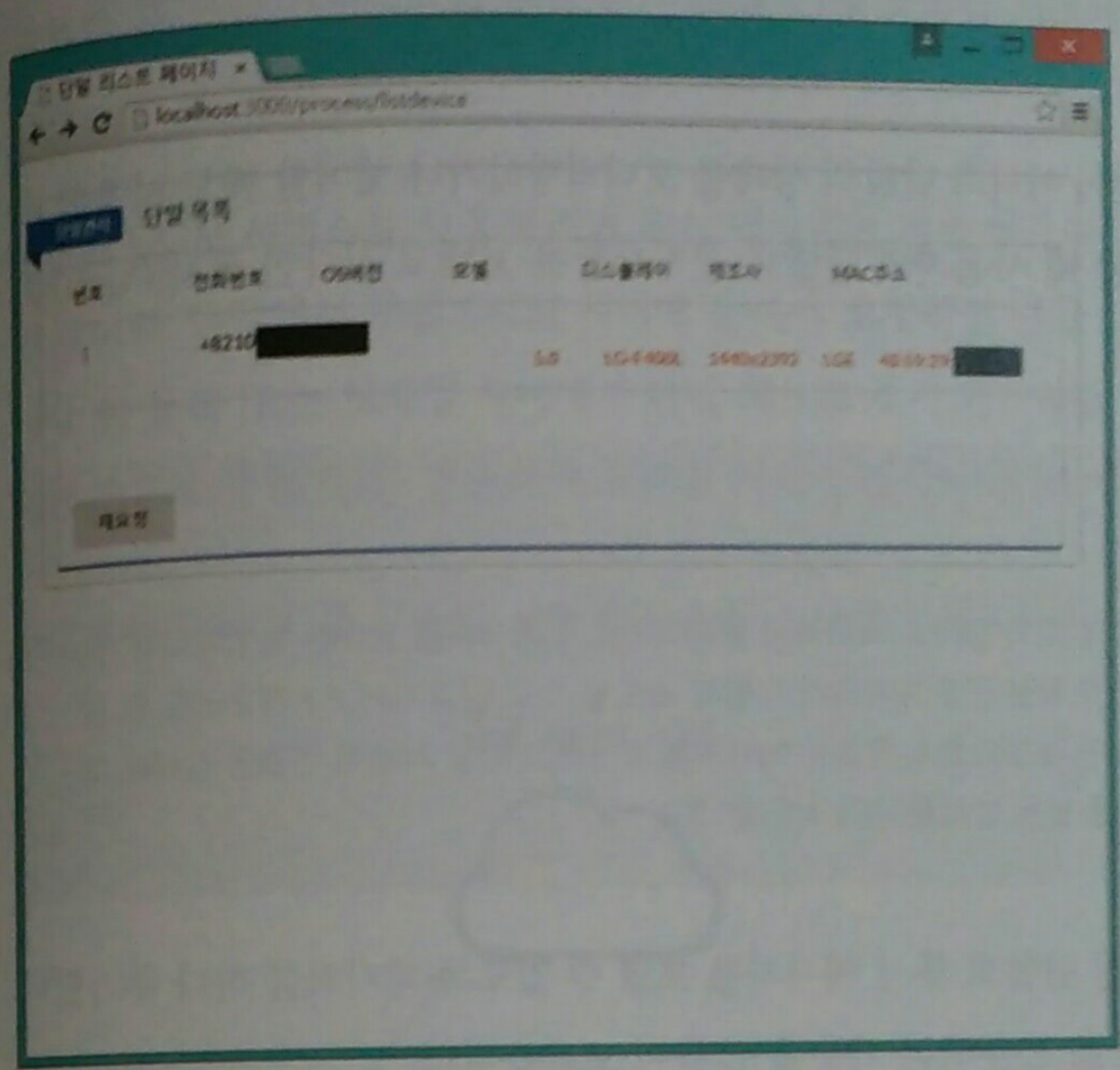
웹 브라우저를 열고 다음 주소로 접속하면 [조회] 버튼을 볼 수 있습니다.

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



▲ 단말 리스트 조회를 요청할 때 사용하는 웹 문서

[조회] 버튼을 클릭하면 단말 목록이 화면에 표시됩니다. 지금까지 추가한 단말이 모두 나타납니다.



▲ 단말 목록을 조회한 화면(실제 단말의 전화번호는 검은색으로 masking하였음)

사용자의 모바일 단말에서 단말 정보를 추가하고, 관리자가 웹 브라우저에서 등록된 단말을 조회하는 기능까지 만들어 보았습니다. 이 과정을 통해 웹 브라우저는 물론 앱에서도 여러분이 만든 웹 서버 기능을 그대로 사용할 수 있다는 것을 이해했을 것입니다.

13-3 | 모바일 단말로 푸시 메시지 전송하기

모바일 서버는 단말 정보만 관리하는 것이 아니라 단말의 상태를 모니터링하거나 제어할 때도 사용할 수 있습니다. 또 관리자나 다른 사람이 원할 때 사용자의 단말로 공지사항을 보내고 싶다면 **푸시(Push) 기능**을 사용하면 됩니다.

정박사의 한마디 푸시가 무엇인지 아세요?

푸시란 클라이언트가 서버에 연결되어 있지 않아도 서버에서 클라이언트에 메시지를 보내는 방법입니다. 클라이언트가 서버에 연결되어 있으면 모바일 단말의 배터리가 빨리 소모될 수도 있고 CPU나 메모리도 더 많이 소모할 수도 있습니다. 이 때문에 언제 도착할지 모르는 메시지를 받으려고 항상 서버에 연결된 상태로 대기하기보다는 푸시 방식으로 메시지를 받는 방식을 많이 사용합니다.

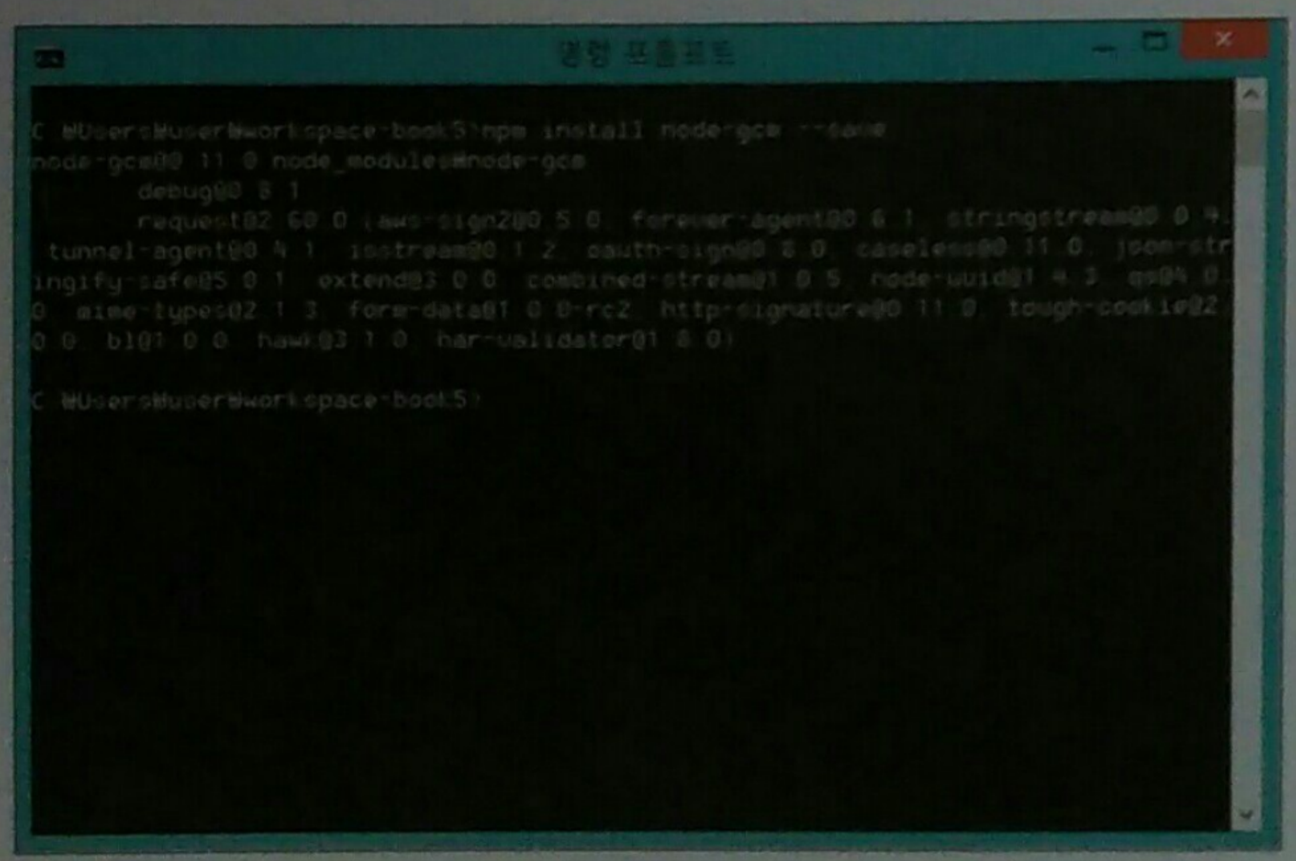
여기에서는 노드로 만든 서버에서 모바일 단말로 푸시 메시지를 보낼 수 있도록 패키지를 하나 추가합니다. 그러고 나서 모바일 단말로 공지사항을 전송하는 기능을 만들어 보겠습니다.

단말의 등록 ID를 확인하여 모바일 서버로 전송하기

노드로 만든 서버에서 안드로이드 단말로 푸시 메시지를 보내려면 구글에서 만든 GCM(Google Cloud Messaging) **푸시 전송 방식**을 따라야 합니다. 노드의 node-gcm **패키지**를 사용하면 구글의 푸시 전송 방식으로 푸시 메시지를 보낼 수 있습니다.

먼저 node-gcm 패키지를 설치하기 위해 명령 프롬프트를 열고 다음 명령을 실행합니다.

```
% npm install node-gcm --save
```

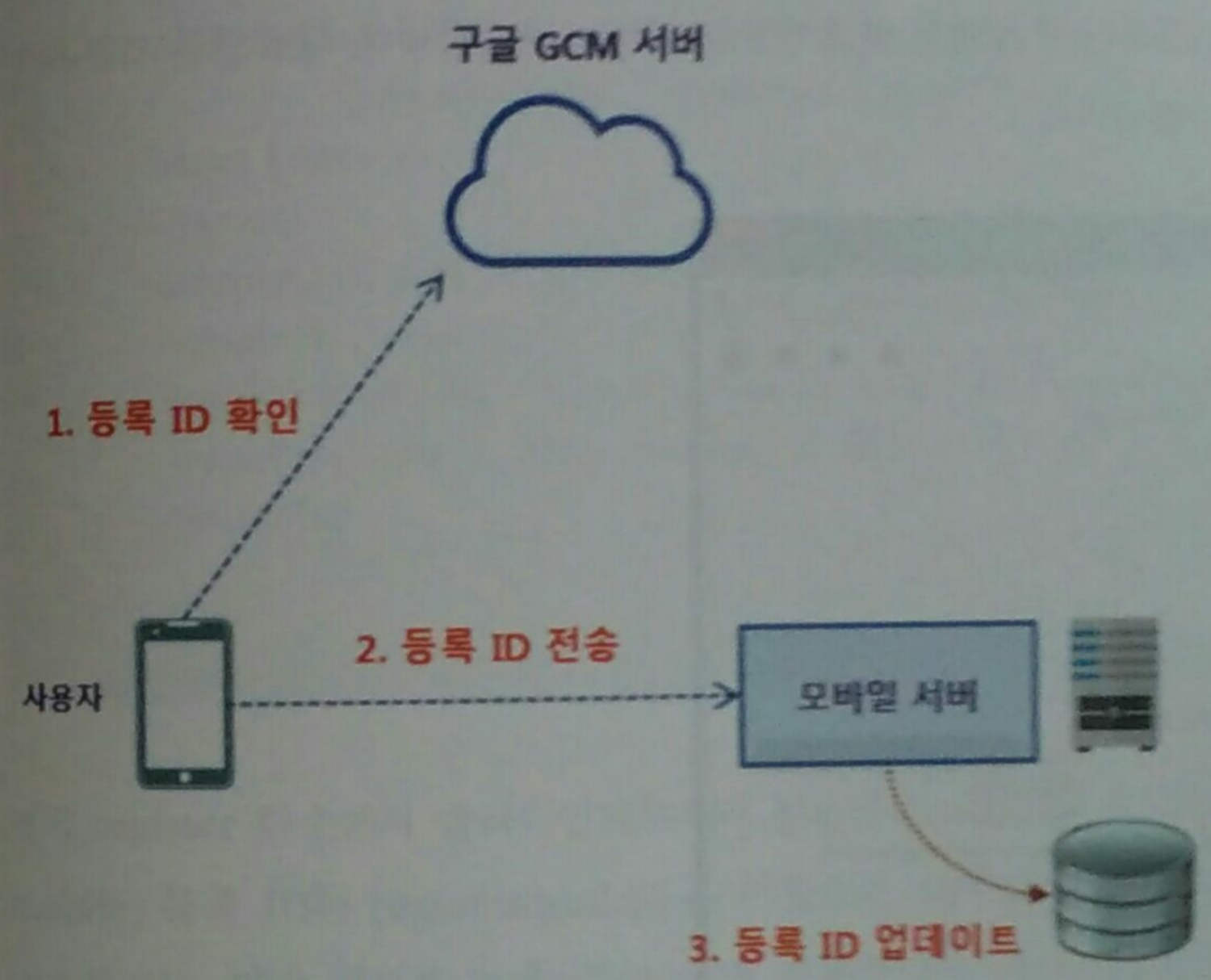


▲ 명령 프롬프트에서 node-gcm 패키지를 설치한 화면

이제 node-gcm 패키지를 사용할 수 있게 되었습니다. 그러면 단말에서 구글 서버에 사용자의 계정을 등록한 후 구글 서버로부터 전달 받은 **등록 ID**를 모바일 서버에 전송하는 기능을 먼저 만들어 보겠습니다.

구글의 GCM 서비스를 사용하려면 푸시 메시지를 전송 받으려는 단말에서 먼저 등록하는 과정을 거쳐야 합니다. 단말에서 구글 GCM 서버에 등록을 요청하면 구글 GCM 서버는 등록 ID를 단말로 보내 줍니다. 이 등록 ID는 모바일 서버에서 단말 쪽으로 푸시 메시지를 전송할 때 사용하므로 단말에서 모바일 서버 쪽으로 등록 ID를 전송하여 모바일 서버의 데이터베이스에 저장해두어야 합니다.

단말에서 등록 ID를 모바일 서버에 전송하는 과정을 정리하면 다음과 같습니다.



▲단말에서 등록 ID를 모바일 서버에 전송하는 과정

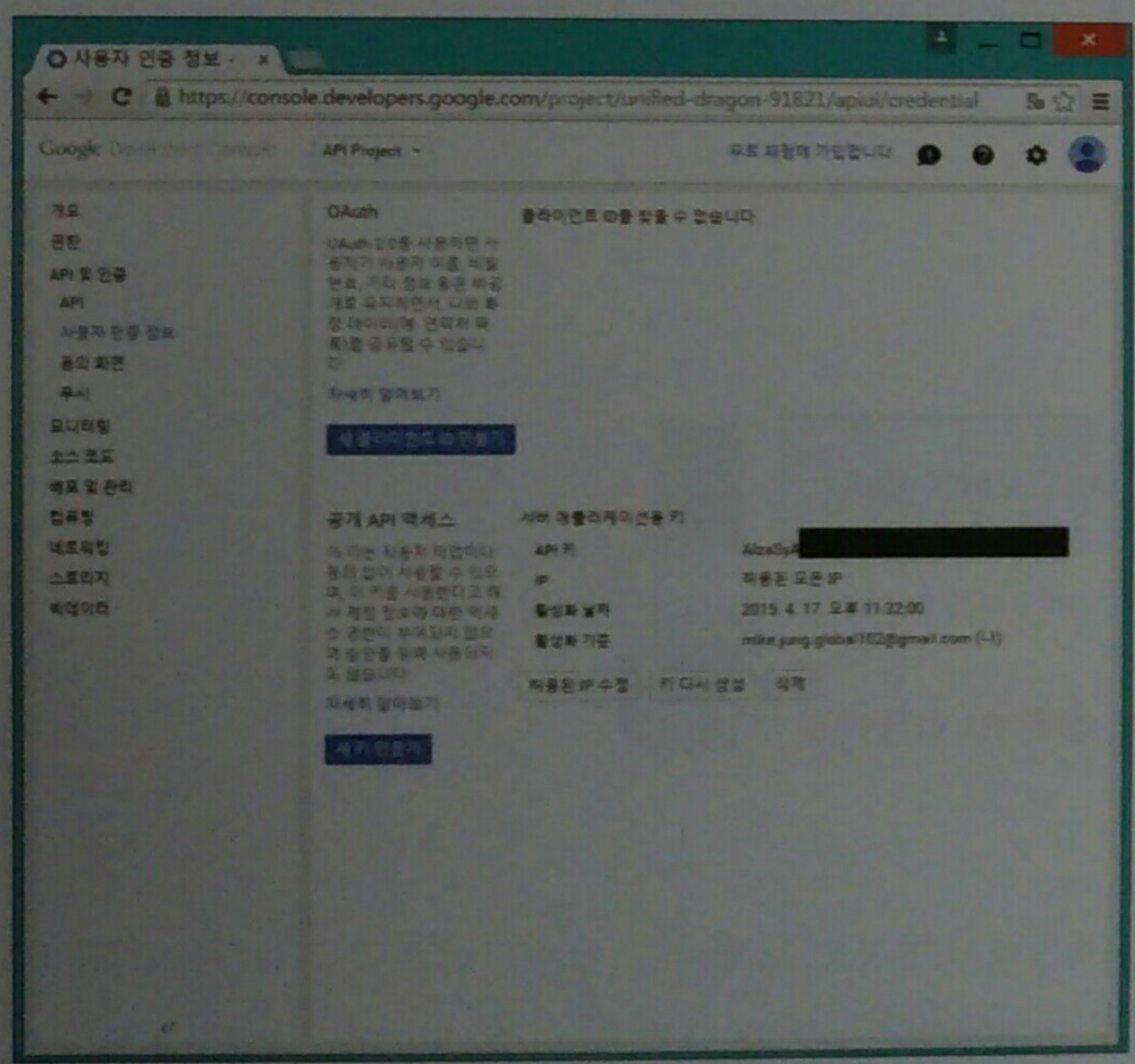
단말의 등록 ID를 서버로 보내 저장하는 과정에서 가장 먼저 해야 할 일은 단말을 구글 GCM 서버에 등록하는 것입니다. 단말 등록이 정상적으로 이루어지면 등록 ID를 받을 수 있으며, 이 등록 ID는 모바일 서버 쪽으로 전송됩니다. 모바일 서버로 전송할 때 단말의 전화번호와 등록 ID를 함께 보내면 모바일 서버에서는 데이터베이스에 있는 devices 컬렉션에서 전화번호를 사용해 객체를 검색한 후 등록 ID를 업데이트합니다. 이렇게 저장한 등록 ID는 공지사항을 전송할 때 사용합니다.

그런데 이 과정에서 구글의 GCM 서비스를 사용하려면 먼저 허가받은 개발자가 만든 앱이 푸시 서비스를 사용하고 있음을 증명할 수 있도록 구글 사이트에서 키(Key)를 발급받아야 합니다. 키는 구글 개발자 콘솔 사이트에서 발급받을 수 있습니다.

▶ http://developer.google.com/console

만약 구글 계정이 없으면 구글 계정을 새로 만들고 로그인합니다. 그다음에는 구글 개발자 콘솔 사이트에서 새로운 프로젝트를 만들 수 있으며, 새로운 프로젝트를 만들면 프로젝트의 고유한 ID가 만들어집니다. '프로젝트 번호'라고 표시되는 숫자는 '93511021974'처럼 여러 개의 숫자로 구성되어 있으니 잘 적어두었다가 사용합니다. 프로젝트의 상세 정보로 들어가면 왼쪽 메뉴 중에서 [API] 메뉴가 있습니다. 이 메뉴를 눌러 들어간 후 [API 사용 설정] 버튼을 클릭하여 Google Cloud Messaging for Android 서비스를 사용할 수 있는 상태로 바꾸어야 합니다. 왜냐하면 구글의 다양한 서비스 중에서 사용자가 선택적으로 서비스를 사용할 수 있기 때문입니다.

Google Cloud Messaging for Android 서비스를 사용하겠다고 바꾼 다음에는 [사용자 인증 정보] 메뉴로 들어가 새로운 서버 키를 발급받습니다. 키를 만드는 과정은 아주 간단합니다. [새 키 만들기] 버튼을 누른 후 대화상자에서 [서버 키]를 누르면 새로운 키를 만드는 화면이 나타납니다. 새로 만든 키는 API 키인데 이 값도 잘 적어두었다가 사용합니다.



▲구글 개발자 콘솔 창에서 서버 키를 만든 경우(실제 키 값은 검은색으로 masking이었음)

주의! 이 책은 안드로이드 앱을 만들 때 설정하는 부분을 간단히 설명하므로 좀 더 자세한 설정 과정을 알고 싶다면 안드로이드 기본서에서 푸시 사용 부분을 참고하세요.

이제 구글 GCM 서비스를 사용할 수 있게 되었으니 모바일 서버에 라우팅 함수를 만들어야 합니다. [routes] 폴더의 device.js 파일을 열고 라우팅 함수인 register 함수를 추가한 다음 데이터베이스에 등록 ID를 저장하는 코드를 입력합니다.

참조 파일 | MobileExample>/routes/device.js

```
.....
var register = function(req, res) {
  console.log('device 모듈 안에 있는 register 호출됨.');

  var database = req.app.get('database');
  var paramMobile = req.param('mobile');
  var paramRegistrationId = req.param('registrationId');

  if(database) {

    // 업데이트
    database.DeviceModel.findOneAndUpdate({mobile : paramMobile},
                {registrationId : paramRegistrationId}, {multi : true}, function(err, result) {
      if(err) {throw err;}
      if(result) {
        console.log("등록 ID 업데이트함.");
        console.dir(result);
        res.writeHead('200', {'Content-Type' : 'application/json;charset=utf8'});
        res.write("{code : '200', 'message' : '등록 ID 업데이트 성공'}");
        res.end();

.....
```

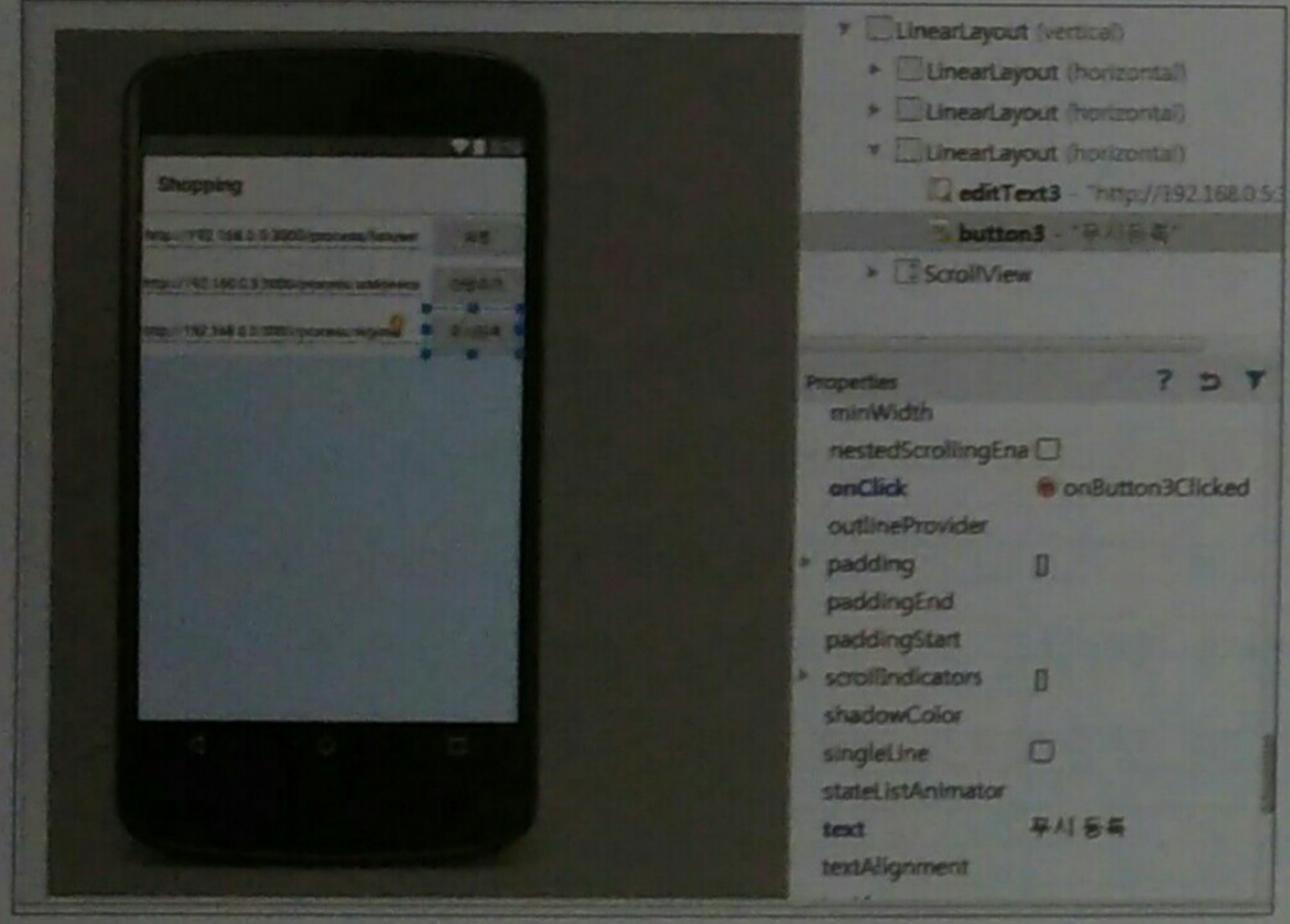
먼저 register 함수에서 클라이언트로부터 전달받은 전화번호와 등록 ID를 확인합니다. 전화번호는 mobile, 등록 ID는 registrationId라는 이름으로 되어 있습니다. 이 두 개의 파라미터는 데이터베이스에 있는 단말 정보를 업데이트하는 데 사용합니다. 데이터베이스에 정의한 devices 스키마에는 registrationId 칼럼이 있는데 이 칼럼에는 등록 ID를 넣어야 합니다. 따라서 DeviceModel 객체에 정의된 findOneAndUpdate 메소드를 호출하여 전달받은 전화번호와 같은 객체를 찾은 후 그 객체에 들어 있는 registrationId 칼럼을 업데이트합니다. 성공적으로 업데이트되면 성공했다는 메시지를 JSON 타입으로 만들어 클라이언트에 보냅니다.

새로 추가한 라우팅 함수의 정보는 [config] 폴더의 config.js 파일에 등록합니다.

참조 파일 | MobileExample>/config/config.js

```
……
module.exports = {

……

  route_info : [

……

    ,{file : './device', path : '/process/register', method : 'register', type : 'post'}
  ],

……
```

이제 안드로이드 앱에서 등록 ID를 보내어 데이터베이스에 있는 단말 정보를 업데이트할 수 있게 되었습니다. 안드로이드 앱 쪽에는 새로운 입력 상자와 버튼을 추가하고 버튼을 클릭했을 때 구글 GCM 서버에 단말을 등록합니다. 그리고 응답 받은 등록 ID는 모바일 서버로 전송합니다. 그러면 먼저 activity_main.xml 파일을 열고 화면을 수정해 보겠습니다.

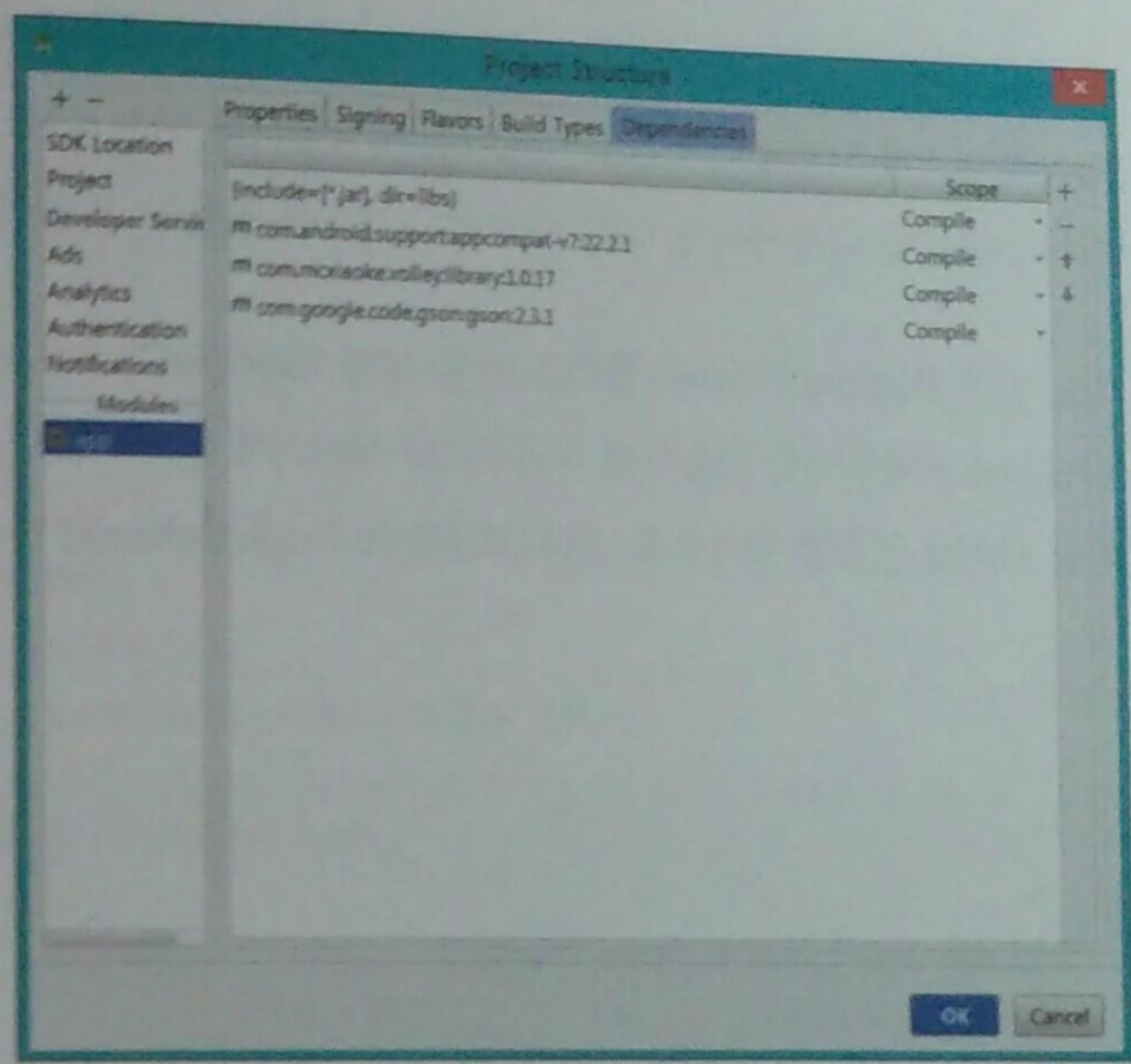


▲ 안드로이드 앱의 화면에 입력 상자 하나와 [푸시 등록] 버튼 추가

입력 상자에는 접속할 URL 정보를 입력하고 버튼에는 '푸시 등록'이라는 글자가 나타나도록 입력합니다. 버튼을 클릭하면 onButton3Clicked 메소드를 호출합니다.

그다음에는 MainActivity.java 파일의 코드를 수정해야 합니다. 코드에서 Google Play Services 라이브러리를 사용하므로 이 라이브러리를 dependencies 항목에 추가합니다. 위쪽 메뉴에서 [File

→ Project Structure] 메뉴를 눌러 나타난 대화 상자의 왼쪽 메뉴에서 [app] 항목을 선택합니다. 그런 다음 오른쪽에 보이는 탭 중에서 [Dependencies] 탭을 선택합니다.



▲ 프로젝트의 라이브러리 Dependency를 설정하는 화면

오른쪽 위의 [+] 버튼을 눌러 나타난 메뉴에서 [Library dependency] 메뉴를 선택하면 라이브러리 중에서 하나를 선택할 수 있습니다. play-services라는 이름이 포함되어 있는 항목을 선택한 후 [OK]를 누릅니다. 이 라이브러리는 구글 GCM 서비스를 사용할 수 있도록 만들어 줍니다.

MainActivity.java 파일을 열고 onButton3Clicked 메소드를 추가합니다. 이 메소드는 구글 GCM 서비스에 접속하여 등록 ID를 받아 오도록 만든 스레드를 시작합니다.

참조 파일 | Shopping3)/java/org.androidtown.shopping/MainActivity.java

```
.....

public void onButton3Clicked(View v) {
  registerUrl = editText3.getText().toString();

  // 단말 등록하고 등록 ID 받기
  registerDevice();
}

private void registerDevice() {
  RegisterThread registerObj = new RegisterThread();
  registerObj.start();
}

.....
```

단말 등록에 필요한 URL 정보는 입력 상자에서 가져와 변수에 저장합니다. 그리고 RegisterThread 클래스의 인스턴스 객체를 만들고 스레드를 시작합니다. 안드로이드에서는 네트워킹 기능을 사용할 때 반드시 스레드를 사용해야 합니다. 따라서 스레드를 만들고 그 안에서 요청을 보내야 합니다.

새로 만든 RegisterThread 클래스 안을 보면, GoogleCloudMessaging 클래스의 인스턴스를 참조한 후 register 메소드를 호출하여 구글 GCM 서비스에 등록하고 있습니다. 이 함수가 반환한 등록 ID는 모바일 서버로 전송할 것이므로 일단 GCMInfo.RegistrationId 변수에 저장합니다.

참조 파일 | Shopping3>/java/org.androidtown.shopping/MainActivity.java

```
……

class RegisterThread extends Thread {
  public void run() {
    try {
      GoogleCloudMessaging gcm =
                        GoogleCloudMessaging.getInstance(getApplicationContext());
      String regId = gcm.register(GCMInfo.PROJECT_ID);
      println("푸시 서비스를 위해 단말을 등록했습니다.");
      println("등록 ID : " + regId);

      // 등록 ID를 모바일 서버에 전송
      GCMInfo.RegistrationId = regId;
      sendToMobileServer();
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }

……
```

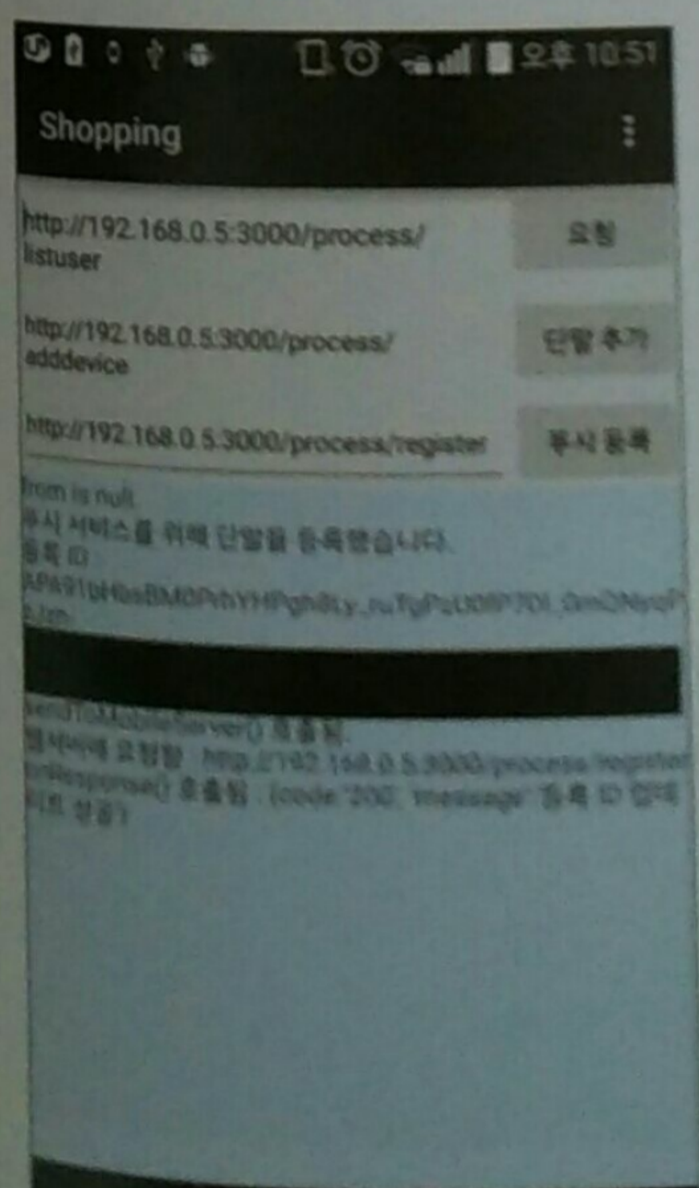
sendToMobileServer 메소드를 호출하면 Volley 라이브러리를 사용해 모바일 서버 쪽에 등록 ID 업데이트를 요청합니다. 화면의 세 번째 입력 상자에서 가져온 URL로 서버에 요청하며 전화번호와 등록 ID를 파라미터로 전달합니다.

참조 파일 | Shopping3>/java/org.androidtown.shopping/MainActivity.java

```
……
public void sendToMobileServer() {

……

  @Override
  protected Map<String, String> getParams() {
    Map<String, String> params = new HashMap<>();

    params.put("mobile", getMobile());
    params.put("registrationId", GCMInfo.RegistrationId);

    return params;
  }
};……

……
```

GCM 기능을 제대로 사용하려면 이 밖에도 추가해야 할 자바 소스 파일이 여러 개 있으며, 매니페스트 파일에는 권한과 함께 몇 가지 정보를 넣어 주어야 합니다. 이 내용은 책과 함께 제공되는 샘플 프로젝트를 참조하세요.

이제 단말이 무선 랜에 연결된 상태에서 샘플 프로젝트 앱을 실행한 후 [푸시 등록] 버튼을 누릅니다. 그러면 단말에서는 먼저 구글 GCM 서버에 단말을 등록한 후 등록 ID를 받아 옵니다. 그리고 이 등록 ID를 모바일 서버에 전송합니다.



◀ 단말에서 [푸시 등록] 버튼을 눌러 등록 ID를 받아 온 후 모바일 서버에 전송한 결과 화면(실제 등록 ID 값

웹 브라우저에서 푸시 메시지 전송하기

이렇게 전송한 등록 ID는 모바일 서버에서 데이터베이스에 저장하므로 이제부터 푸시 메시지 전송에 사용할 수 있습니다. 가장 먼저 할 일은 모바일 서버 쪽에 푸시 전송 기능을 추가하는 것입니다.

관리자가 웹 브라우저에서 푸시 메시지를 전송하기 위해 요청할 라우팅 함수를 device.js 파일에 추가합니다.

참조 파일 | MobileExample>/routes/device.js

```
……

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

  var database = req.app.get('database');
  var paramData = req.param('data');

  if(database) {
    // 1. 모든 단말 검색
    database.DeviceModel.findAll(function(err, results) {
      if(err) {throw err;}
      if(results) {
        console.dir(results);

        // 등록 ID만 추출
        var regIds = [];
        for(var i = 0; i < results.length; i++) {
          var curId = results[i]._doc.registrationId;
          console.log('등록 ID #' + i + ' : ' + regIds.length);
          regIds.push(curId);
        }
        console.log('전송 대상 단말 수 : ' + regIds.length);

        // node-gcm을 사용해 전송
        var message = new gcm.Message();
        message.addData('command', 'show');
        message.addData('type', 'text/plain');
        message.addData('data', paramData);

        var sender = new gcm.Sender(config.gcm_api_key);
        sender.send(message, regIds, function (err, result) {
          if(err) {throw err;}
          console.dir(result);
```

```
    res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
    res.write('<h2>푸시 메시지 전송 성공</h2>');
    res.end();
  });

……,
```

sendall 함수는 웹 브라우저에서 보내 온 data 파라미터를 확인한 후 단말로 이 데이터를 푸시합니다. 단말로 푸시 메시지를 전송하려면 수신 단말들의 등록 ID를 배열 객체로 만들어야 합니다. 따라서 데이터베이스에서 모든 객체를 조회한 후 등록 ID만 따로 빼내어 배열 객체에 추가합니다. 이렇게 만든 배열 객체는 send 메소드를 호출할 때 사용합니다. send 메소드는 구글 GCM을 사용할 수 있는 키 값을 파라미터로 전달받아 만듭니다. 이 키 값은 config.js 파일에 gcm_api_key 속성으로 추가되어 있습니다. send 메소드로 전달되는 파라미터에는 Message 객체도 있습니다. 이 객체에는 전송할 데이터를 추가할 수 있는데 command, type, data 값을 추가했습니다.

라우팅 함수를 만들었으니 이 함수를 config.js 파일에 등록합니다.

참조 파일 | MobileExample>/config/config.js

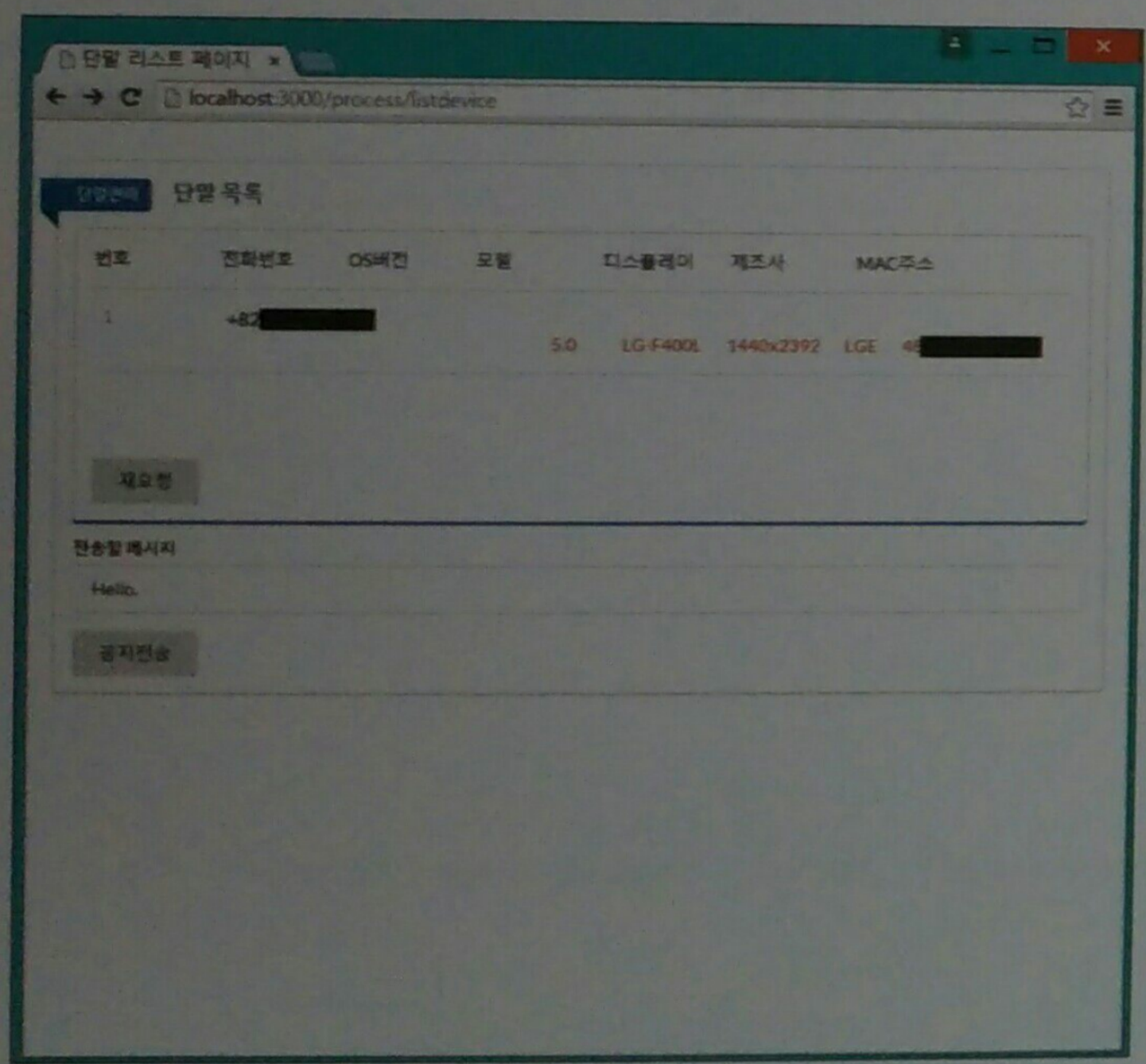
```
……,
module.exports = {

……,

  route_info : [

……,

    ,{file : './device', path : '/process/sendall', method : 'sendall', type : 'post'}
  ],

……,
```

관리자는 단말 리스트 화면에서 공지사항을 전송할 것이므로 [views] 폴더의 listdevice.ejs 파일을 열고 아래쪽에 공지사항을 전송하는 〈form〉 태그를 추가합니다.

참조 파일 | MobileExample>/views/listdevice.ejs

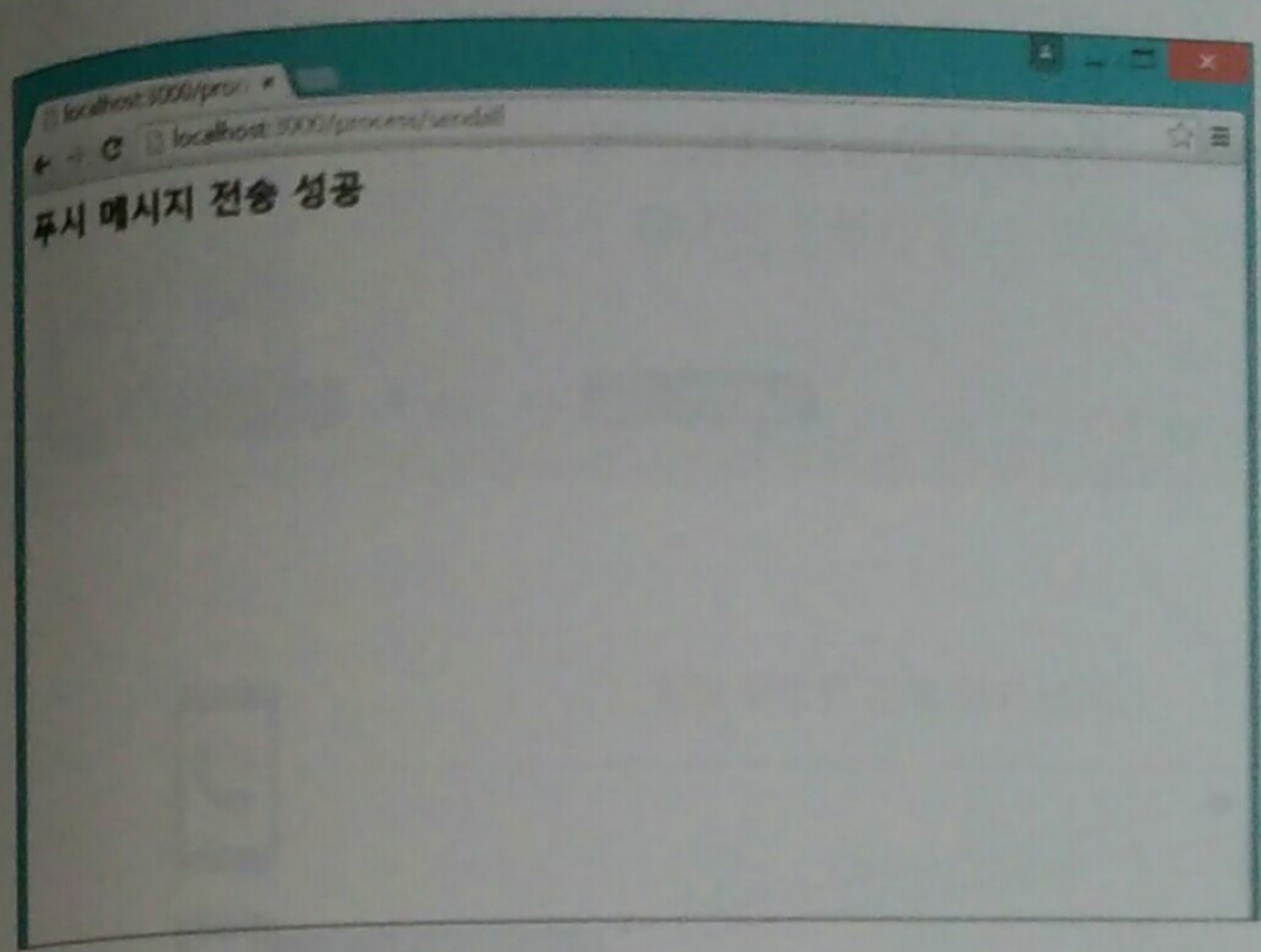
```
……
        <div>
    <form action = "/process/sendall" method = "post" class = "content ui form">
      <div class = "field">
        <label>전송할 메시지</label>
        <input type = "text" class = "ui input" name = "data">
      </div>

      <button type = "submit" class = "ui button">공지전송</button>
    </form>
  </div>

……
```

이제 app.js 파일을 실행한 후 웹 브라우저를 열고 단말 목록을 조회합니다.



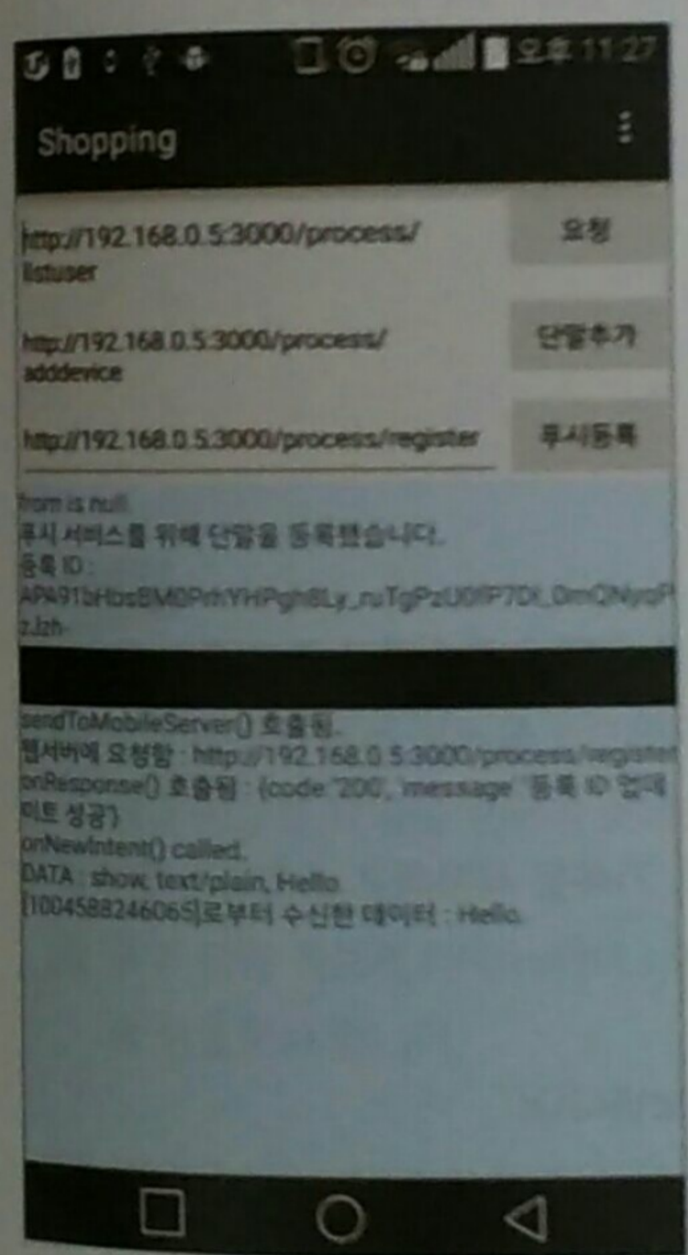
▲ 웹 브라우저에서 단말 목록을 조회한 화면

화면의 아래쪽에 있는 입력 상자에 'Hello.'라고 입력하고 [공지 전송] 버튼을 누르면 /process/sendall 패스로 푸시 메시지 전송을 요청합니다. 모바일 서버에서 성공적으로 푸시 메시지를 전송했다면 응답 화면은 다음과 같이 출력됩니다.



▲모바일 서버로부터 응답 받은 푸시 전송 성공 메시지

푸시 메시지가 성공적으로 전송되면 모바일 단말에서 전송받은 메시지를 볼 수 있습니다.



◀ 모바일 단말에서 푸시 메시지를 받아 화면에 뿌려 준 경우(실제 등록 ID 값은 검은색으로 masking 하였음)

관리자가 사무실의 PC에서 웹 브라우저를 열고 단말 목록을 볼 수 있으며 그 단말들로 푸시 메시지를 전송할 수 있도록 만들어 보았습니다. 모바일 단말에서 동작하는 앱과 여러분이 만든 서버가 서로 데이터를 주고받을 수 있고 서버에서 모바일 단말로 푸시를 보낼 수도 있다는 것이 신기하지 않나요? 모바일 앱에 대해 잘 모르더라도 이 책과 함께 제공되는 앱을 사용하면 여러분도 서버 쪽 함수들만 간단히 재구성해서 푸시 메시지 전송 기능을 만들 수 있습니다.

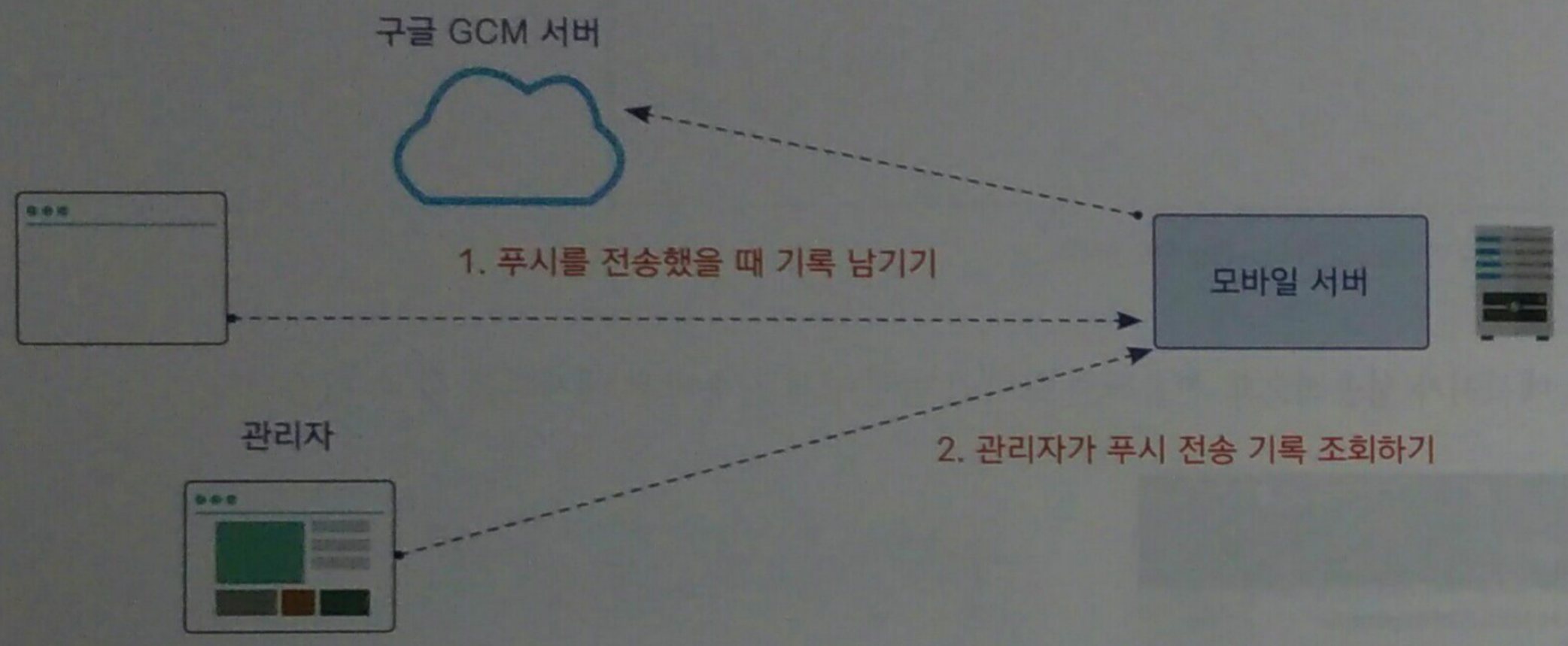
다음 장에서는 게시판 기능을 만든 후 여러분이 직접 만든 게시판을 클라우드에 올려 볼 테니 기대하세요!

Do it!

도전 문제

01. 푸시 메시지를 전송할 때마다 그 기록을 서버 데이터베이스에 남기고 필요할 때 전송 기록을 조회하는 기능을 만들어 보세요.

프로젝트 소스 Mission19 난이도 초급 ★☆☆☆



Hint!

1. PC의 웹 브라우저 화면에서 푸시 메시지를 전송하면 모바일 서버에서 푸시 메시지를 단말로 전송하고 그 기록을 데이터베이스에 남깁니다.

2. 푸시 전송 기록을 데이터베이스에 남기도록 새로운 데이터베이스 스키마를 만들고, 기록을 저장하고 조회하는 함수를 만들어야 합니다.

3. 데이터베이스에 저장한 후 정상적으로 저장되었다는 응답 메시지를 클라이언트로 보냅니다.

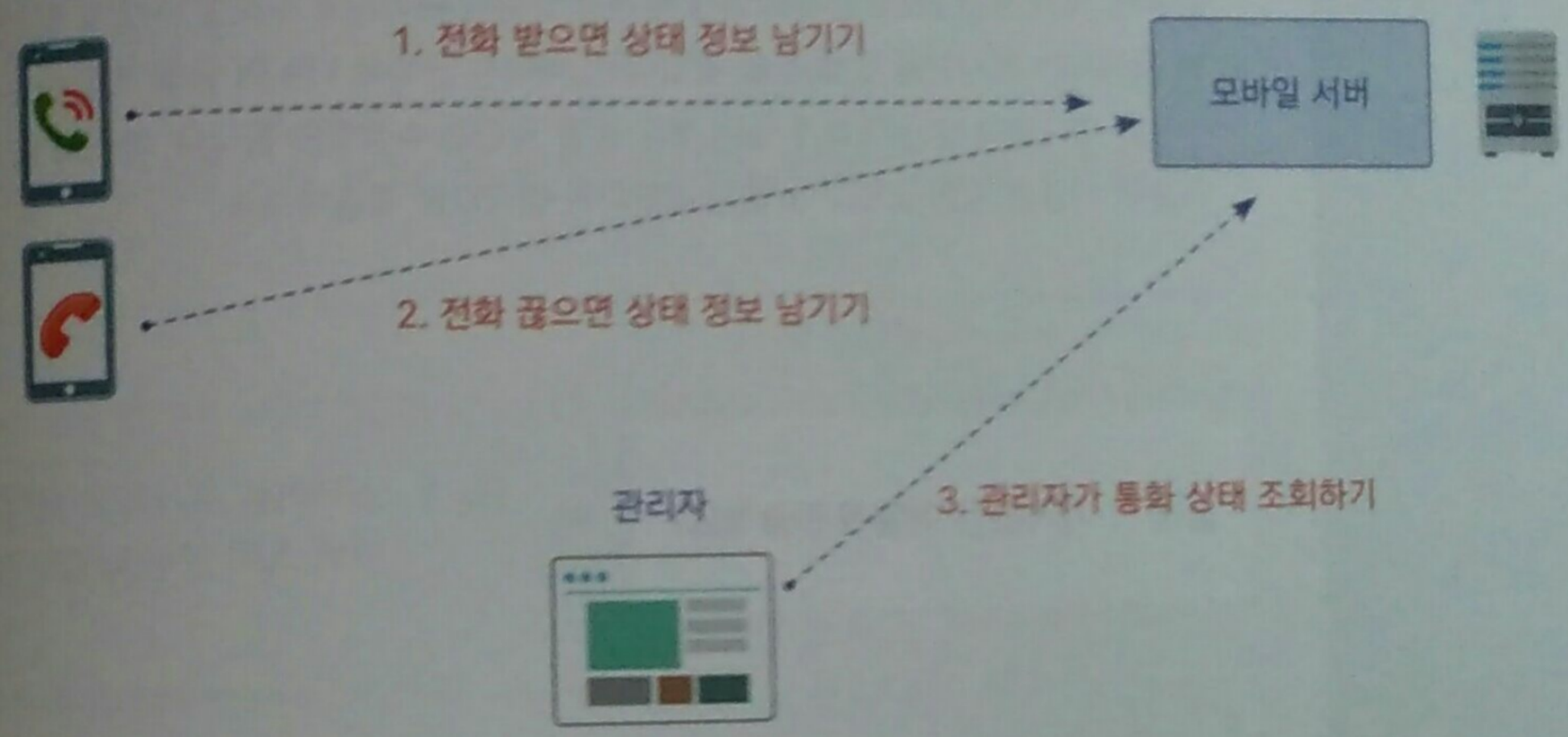
4. 관리자가 푸시 전송 기록을 조회할 수 있도록 웹 문서를 만들고, 모바일 서버로 조회를 요청했을 때 푸시 전송 기록을 조회할 수 있도록 응답을 보내 줍니다.

MEMO

DO it!
도전 문제

02. 모바일 단말에서 상태 정보를 서버로 보내 모니터링할 수 있는 기능을 만들어 봅니다. 모바일 단말에서 전화를 받으면 통화 중이라는 상태 정보를 서버로 보내고 통화가 끝나면 통화가 끝난 상태를 서버로 보내도록 만들어 보세요.

프로젝트 소스 Mission20 난이도 중급 ★★☆☆



Hint!

1 모바일 단말에서 전화를 받으면 전화를 받았다는 상태 정보를 모바일 서버에 보냅니다. 전화를 끊었을 때 전화를 끊었다는 상태 정보를 모바일 서버에 보내도록 안드로이드 앱을 만듭니다.

2 통화 상태 정보를 데이터베이스에 남기도록 새로운 데이터베이스 스키마를 만들고, 통화 상태를 저장하고 조회하는 함수를 만들어야 합니다.

3 데이터베이스에 저장한 후 정상적으로 저장되었다는 응답 메시지를 클라이언트로 보냅니다.

4 관리자가 통화 상태를 조회할 수 있도록 웹 문서를 만들고, 모바일 서버로 조회를 요청했을 때 통화 상태 기록을 조회할 수 있도록 응답을 보내 줍니다.

MEMO

14 게시판 만들기

지금까지 단계별로 하나씩 만들어 본 서버 기능은 어느 정도 실무에서 사용할 수 있을 만큼 실용적인 내용이 포함되어 있습니다. 그런데 '게시판은 어떻게 만드나요?'라는 질문이 아직 남아 있습니다. 웹 서비스를 할 때 가장 기본이 되는 기능 중 하나가 블로그에 글을 쓰고 글 목록을 조회하는 게시판입니다. 게시판에는 사용자가 작성한 글이 계속 누적되기 때문에 10개 또는 20개 단위로 글을 묶어서 하나의 페이지로 조회할 수 있도록 만듭니다. 이 장에서는 게시판을 만들어 볼 것입니다. 페이지 단위로 나누어 글을 조회하는 게시판 기능은 여러분이 앞 장에서 만들어 본 기능을 약간만 수정하면 됩니다. 따라서 이 장은 여러분이 지금껏 익힌 노드 기능을 정리한다는 생각으로 학습하세요.

질문으로 시작하기

질문1. 게시판은 어떻게 만들까요? ➡ 14-1. 스키마를 추가하고 페이지 단위 조회 방식 이해하기

질문2. 게시판에 글 쓰는 기능은 어떻게 만들까요? ➡ 14-2. 글쓰기와 글 조회 기능 만들기

질문3. 페이지 단위로 게시판 글을 보고 싶나요? ➡ 14-3. 페이지 단위로 글 리스트 조회하기

14-1 스키마를 추가하고 페이지 단위 조회 방식 이해하기

게시판 프로젝트는 앞에서 만든 PassportExample2 프로젝트를 복사하여 새로운 BoardExample 프로젝트로 만듭니다. 사용자가 쓴 글을 게시판에 저장하기 위해서는 가장 먼저 데이터베이스에 스키마를 추가해야 합니다.

[database] 폴더에 post_schema.js 파일을 만들고 다음 코드를 입력합니다.

참조 파일 | BoardExample>/database/post_schema.js

```
.....

var SchemaObj = { };

SchemaObj.createSchema = function(mongoose) {
  // 글 스키마 정의
  var PostSchema = mongoose.Schema({
    title : {type : String, trim : true, 'default' : ''},        // 글 제목
    contents : {type : String, trim :true, 'default' : ''},      // 글 내용
    writer : {type : mongoose.Schema.ObjectId, ref : 'users'},  // 글 쓴 사람
    tags : {type : [ ], 'default' : ''},
    created_at : {type : Date, index : {unique : false}, 'default' : Date.now},
    updated_at : {type : Date, index : {unique : false}, 'default' : Date.now},
                                                                 // 댓글
    comments : [{
      contents : {type : String, trim : true, 'default' : ''},  // 댓글 내용
      writer : {type : mongoose.Schema.ObjectId, ref : 'users'},
      created_at : {type : Date, 'default' : Date.now}
    }]
  });

.....
```

게시판이나 블로그에 쓴 글을 저장하는 스키마에는 제목과 내용 그리고 글쓴이 등의 칼럼이 필요합니다. 여기에서는 다음과 같은 일곱 가지 칼럼을 넣어 두었습니다.

칼럼 이름	설명
title	글 제목
contents	글 내용
writer	글 쓴 사람 글쓴이 정보를 참조할 수 있도록 ObjectId가 저장됨
tags	글에 붙이는 태그 정보
created_at	글을 쓴 일시
updated_at	글을 수정한 일시
comments	댓글 저장

[표] 게시판에 쓴 글을 저장하는 스키마의 기본 칼럼

이 칼럼은 누군가 글을 썼을 때 그 정보를 저장하는 데 사용하므로 이름만 보아도 어떤 데이터를 저장하려는지 쉽게 알 수 있습니다. 하지만 여기서 두 가지 특이한 점이 보입니다.

첫 번째 특이점은 writer 칼럼에 저장되는 데이터 타입이 mongoose.Schema.ObjectId라는 것입니다. 이것은 사용자 정보가 users 컬렉션에 저장되어 있는데 이 컬렉션에 있는 문서 객체 중 ObjectId 속성 값이 이 writer 칼럼에 저장된다는 의미입니다. 이것을 다른 컬렉션을 가리키는 **레퍼런스(Reference)**라고 하며, 몽고디비의 여러 컬렉션 사이에서 데이터를 연결해 주는 키(Key) 역할을 합니다.

두 번째 특이점은 comments 칼럼 안에 다시 객체가 정의되어 있다는 것입니다. comments는 댓글을 저장하는데 댓글은 별도의 컬렉션에 저장할 수도 있지만 여기에서는 하나의 글과 함께 저장하는 방식을 사용합니다. 하나의 글에 댓글이 여러 개 달릴 수 있으므로 배열 타입으로 댓글의 내용, 글쓴이, 글 쓴 날짜를 저장하도록 만들었습니다.

스키마를 만들고 칼럼을 정의했으니 이 칼럼 중에서 제목과 내용 칼럼을 필수로 입력하도록 다음 속성을 추가합니다.

참조 파일 | BoardExample>/database/post_schema.js

```
……

// 필수 속성의 'required' validation
PostSchema.path('title').required(true, '글 제목을 입력하셔야 합니다.');
PostSchema.path('contents').required(true, '글 내용을 입력하셔야 합니다.');

……
```

스키마 객체에 정의된 path 메소드를 호출하여 칼럼을 지정한 후 required 메소드를 호출하여 필수 입력 칼럼으로 만듭니다. 그리고 글 제목을 입력하지 않았을 때는 '글 제목을 입력하셔야 합니다.'라는 메시지를 오류 메시지로 전달합니다.

그다음 스키마에 인스턴스 메소드를 추가합니다. 이미 알고 있는 것처럼 인스턴스 메소드를 추가하면 모델 인스턴스 객체에서 호출할 수 있습니다. 새로 추가하는 인스턴스 메소드로는 savePost, addComment, removeComment입니다.

참조 파일 | BoardExample>/database/post_schema.js

```
……

PostSchema.methods = {
  savePost : function(callback) {              // 글 저장
    var self = this;

    this.validate(function(err) {
      if(err) return callback(err);

      self.save(callback);
    });
  },

  addComment : function(user, comment, callback) {     // 댓글 추가
    this.comment.push({
      contents : comment.contents,
      writer : user._id
    });
    this.save(callback);
  },

  removeComment : function(id, callback) {             // 댓글 삭제
    var index = utils.indexOf(this.comments, {id : id});

    if(~index) {
      this.comments.splice(index, 1);
    } else {
      return callback('ID [' + id + ']를 가진 댓글 객체를 찾을 수 없습니다.');
    }
    this.save(callback);
  }
}
```

savePost는 글을 저장할 때 사용합니다. 이 함수 안에는 save 메소드를 호출하여 데이터베이스에 글을 저장하는 코드가 들어 있습니다. addComment는 댓글을 추가할 때 사용합니다. 댓글을 추가하는 칼럼은 배열 객체로 정의되어 있으므로 push 메소드를 사용해 댓글을 추가합니다.

removeComment는 댓글을 삭제할 때 사용합니다. 댓글을 삭제할 때는 댓글의 id 값을 전달하며 id를 사용해 인덱스 값을 찾아냅니다.

id 값으로 인덱스를 찾는 코드는 [utils] 폴더를 만든 후 그 안에 utils.js 파일을 만들어 추가합니다.

참조 파일 | BoardExample>/utils/utils.js

```
exports.indexOf = function(arr, obj) {
  var index = -1;
  var keys = Object.keys(obj);

  var result = arr.filter(function(doc, idx) {
    var matched = 0;

    for(var i = keys.length - 1; i >= 0; i--) {
      if(doc[keys[i]] == obj[keys[i]]) {
        matched++;

        if(matched == keys.length) {
          index = idx;
          return idx;
        }
      }
    }
  });

  return index;
}

......
```

utils.js 파일 안에 입력한 indexOf 함수는 배열 안에 들어 있는 객체 중에서 조건에 맞는 객체를 찾아냅니다.

이렇게 모듈로 만든 utils.js 파일 안에 정의한 함수를 post_schema.js 파일 안에서 사용하므로 post_schema.js 파일의 위쪽에는 utils.js 모듈을 사용할 수 있는 코드를 추가해야 합니다.

참조 파일 | BoardExample>/database/post_schema.js

```
var utils = require('../utils/utils');

......
```

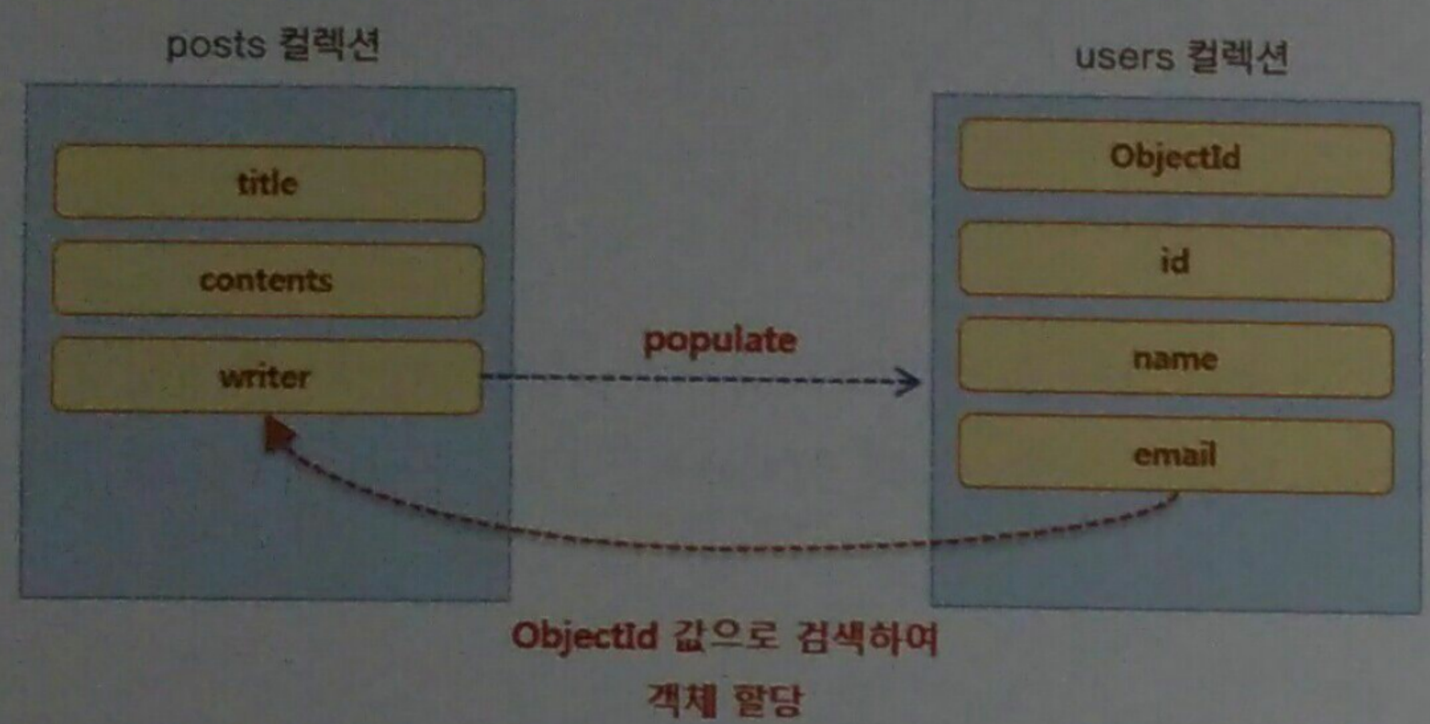
배열 안에 들어 있는 댓글을 삭제할 때는 splice 메소드를 사용합니다.

이제 모델 객체에서 호출할 수 있는 메소드들을 추가한 후 스키마 객체를 module.exports에 할당합니다. static 메소드를 추가하면 모델 객체에서 사용할 수 있습니다. 여기에서는 load와 list 메소드를 추가합니다.

참조 파일 | BoardExample>/database/post_schema.js

```
......

PostSchema.statics = {
  // ID로 글 찾기
  load : function(id, callback) {
    this.findOne({_id : id})
        .populate('writer', 'id name email')
        .populate('comments.writer')
        .exec(callback);
  },
  list : function(options, callback) {
    var criteria = options.criteria || { };

    this.find(criteria)
        .populate('writer', 'id name email')
        .sort({'created_at' : -1})
        .limit(options.perPage)
        .skip(options.perPage * options.page)
        .exec(callback);
  }
}

console.log('PostSchema 정의함.');

return PostSchema;
};

// module.exports에 PostSchema 객체를 직접 할당
module.exports = SchemaObj;
```

첫 번째 메소드는 id 값으로 객체를 찾는 load 메소드입니다. 이 메소드는 findOne 메소드를 호출하는데 load 함수 안에서 populate 메소드를 함께 호출하고 있습니다. populate 메소드는 ObjectId로 참조한 다른 컬렉션의 객체를 가져와 데이터를 채워 줍니다.

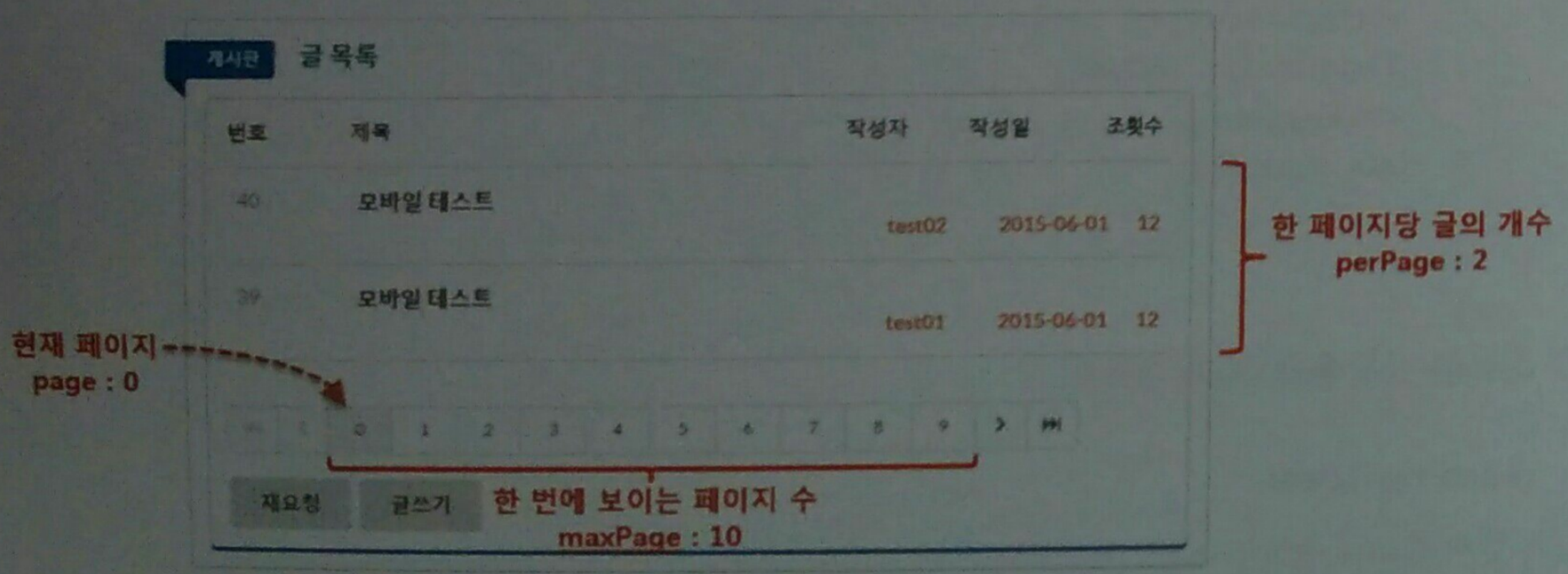


▲ populate 메소드의 역할

게시판의 글을 저장하고 있는 posts 컬렉션에는 writer 칼럼이 들어 있고 writer 칼럼의 값으로 ObjectId 값이 저장됩니다. 이 ObjectId 값은 users 컬렉션에 들어 있는 사용자 데이터 객체 중 하나와 같으므로 populate 메소드가 호출되면 ObjectId 값으로 users 컬렉션에서 검색한 후 그 데이터를 이 객체에 할당합니다.

페이지 단위로 조회하기

list 메소드는 모든 데이터를 조회할 때 사용합니다. 검색된 데이터가 너무 많아 문제가 생기지 않도록 조건에 따라 일부 데이터만 조회합니다. 이때 pagination을 사용해서 페이지 단위로 조회하도록 만들 수 있습니다. 다음은 페이지 단위로 조회할 때 필요한 정보를 정리한 것입니다.



전체 글의 개수 ··· count : 25

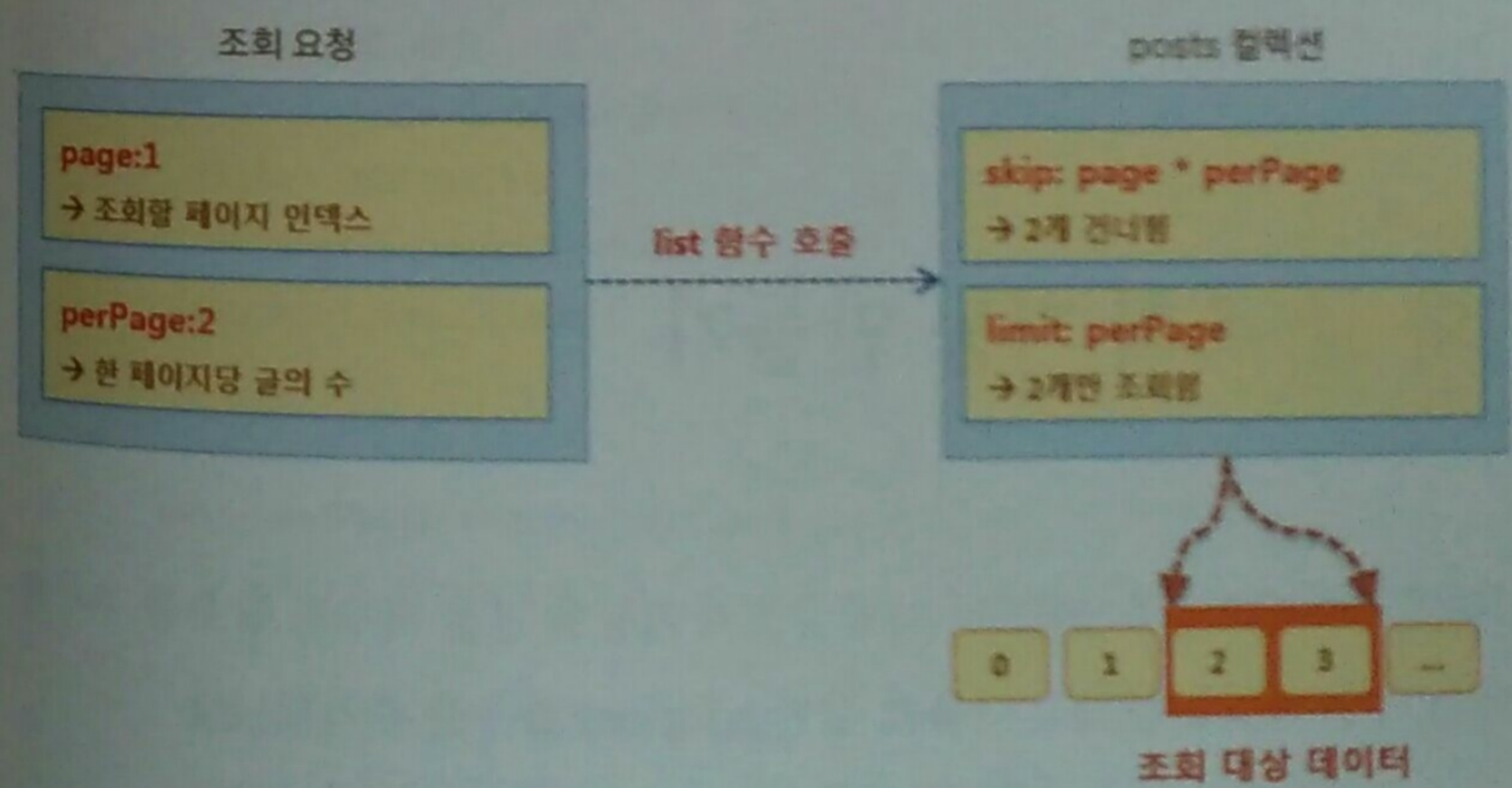
전체 페이지 수 ··· pageCount : count / perPage (25/2+1) = 13

▲ 페이지 단위로 조회할 때 필요한 정보

앞에서 살펴본 화면은 글 목록을 보여 주는 것으로 이 장에서 만들 화면입니다. 지금 보고 있는 페이지는 0번 인덱스에 해당하는 첫 번째 페이지이며 화면 아래쪽에 나열된 열 개의 페이지 중 하나를 선택할 수 있습니다. 이것은 한 번에 볼 수 있는 글 목록 페이지 수를 열 개로 제한한 결과입니다. 만약 열 개의 글 목록 페이지 다음 페이지를 보려면 왼쪽이나 오른쪽의 화살표를 눌러 이동하면 됩니다. 첫 페이지나 끝 페이지로 바로 이동하려면 좌우 가장자리에 있는 버튼을 누릅니다. 이 내용은 maxPage 변수에서 정의할 수 있습니다.

전체 페이지 수를 계산할 때는 전체 글의 수를 한 페이지에 보이는 글 목록의 수로 나눕니다. 예를 들어, 전체 글의 개수가 25개이고 한 페이지당 보이는 글 목록의 개수가 2라면, 페이지 수는 25를 2로 나눈 몫인 12에 1을 더한 13이 됩니다. 전체 글의 개수를 count 변수에 담아 두고 한 페이지당 보이는 글 목록의 개수를 perPage 변수에 담아 두면 이 변수들을 사용해서 전체 페이지 개수를 계산할 수 있습니다. 계산한 값은 pageCount 변수에 담아 둘 수 있습니다. 현재 조회하고 있는 페이지의 인덱스 값은 page 변수에 담아 둘 수 있는데, 웹 브라우저에서 조회 요청을 할 때는 page와 perPage 값만 전달해도 서버에서 전체 글의 개수나 전체 페이지 개수를 계산하여 결과물을 만들 수 있습니다.

이렇게 페이지 단위 조회가 필요하므로 list 메소드에서는 한 페이지에서 볼 수 있는 글의 개수만큼만 조회하도록 limit 메소드를 사용하였고, 현재 페이지의 시작 글 이전으로 건너뛴 후 조회하도록 skip 메소드를 사용하였습니다. list 메소드를 호출할 때 전달하는 options 객체에는 perPage와 page 속성이 들어 있어 사용자가 한 페이지당 몇 개의 글을 조회하고 싶어 하는지, 그리고 현재 보고 있는 페이지의 인덱스 값이 무엇인지를 알 수 있습니다.



▲list 메소드를 호출할 때 조회 대상 데이터를 결정하는 방식

만약 사용자가 두 번째 페이지를 조회하려면 page 속성 값은 1, perPage 속성 값은 2를 전달할 수 있습니다. 이 값들을 파라미터로 하여 list 메소드를 호출하면, page와 perPage의 값을 곱한 값이 2가 되므

로 skip 옵션을 사용해 두 개의 데이터를 건너뛰게 됩니다. 또 perPage의 값이 2이므로 limit 옵션을 사용해 두 개의 데이터만 조회합니다. 따라서 조회 대상이 되는 데이터는 인덱스 값이 2와 3인 객체입니다.

이제 페이지 단위 조회 방식에 대해 어느 정도가 이해가 되었을 것입니다. 게시판을 만들기 위한 스키마를 추가했으니 이 스키마 정보를 config.js 파일에 등록합니다.

참조 파일 | BoardExample>/config/config.js

```
module.exports = {

.......

  db_schemas : [

.......

    ,{file : './post_schema', collection : 'post',
                  schemaName : 'PostSchema', modelName : 'PostModel'}
  ],

.......
```

post_schema.js 파일에 정의한 스키마는 post 컬렉션에 저장하고 스키마 객체 이름은 PostSchema, 모델 객체 이름은 PostModel로 만들었습니다.

14-2 | 글쓰기와 글 조회 기능 만들기

페이지 단위 조회 방식을 이해했고 스키마도 추가했으니 이제 글쓰기 기능을 만들 라우팅 함수를 추가해 보겠습니다. [routes] 폴더에 post.js 파일을 만들고 다음과 같이 addpost 함수를 추가합니다.

```
.....

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

  var paramTitle = req.param('title');
  var paramContents = req.param('contents');
  var paramWriter = req.param('writer');
  var database = req.app.get('database');

  if(database.db) {

    // 1. 아이디를 사용해 사용자 검색
    database.UserModel.findById(paramWriter, function(err, results) {
      if(err) {throw err;}
      if(results == undefined || results.length < 1) {
        res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
        res.write('<h2>사용자 [' + paramWriter + ']를 찾을 수 없습니다.</h2>');
        res.end();
        return;
      }

      var userObjectId = results[0]._doc._id;
      console.log('사용자 ObjectId : ' + paramWriter +' -> ' + userObjectId);

      // save( )로 저장
      var post = new database.PostModel({
        title : paramTitle,
        contents : paramContents,
        writer : userObjectId
      });

      post.savePost(function(err, result) {
        if(err) { throw err; }

        console.log("글 데이터 추가함.");
        console.log('글 작성', '포스팅 글을 생성했습니다. : ' + post._id);

        return res.redirect('/process/showpost/' + post._id);
      });

    });

.....
```

addpost 메소드를 호출하면 데이터베이스에 글이 추가되고 클라이언트로부터 제목, 내용, 글쓴이 정보를 전달받도록 만들었습니다. 그런데 글쓴이는 post 컬렉션에 데이터를 직접 저장하지 않고 user 컬렉션에 있는 사용자 데이터를 참조하여 ObjectId만 저장해야 합니다. 따라서 먼저 글쓴이의 이메일 주소를 사용해 user 컬렉션에서 객체를 조회한 후 글쓴이의 객체가 조회되면 그 객체의 ObjectId를 저장합니다.

데이터를 저장했으면 저장한 글을 사용자에게 응답으로 보여 주기 위해 redirect 메소드를 호출합니다. 이때 파라미터로 글을 조회할 때 사용하는 패스를 전달합니다. 글을 조회할 때 사용하는 패스는 /process/showpost/ 문자열에 저장된 글의 ObjectId 값을 붙여서 만듭니다.

이제 글을 조회하는 함수를 추가해 보겠습니다. addpost 함수 아래쪽에 다음과 같이 showpost 함수를 추가합니다.

참조 파일 | BoardExample>/routes/post.js

```
.....

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

  var paramId = req.param('id');              // 글 ID
  console.log('전달 받은 id : ' + paramId);

  var database = req.app.get('database');

  if(database.db) {
    // 1. 글 조회
    database.PostModel.load(paramId, function(err, results) {
      if(err) {throw err;}
      if(results) {
        console.dir(results);
        res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});

        // 뷰 템플릿을 사용하여 렌더링한 후 전송
        var context = {
          title : '글 조회 ',
          posts : results,
          Entities : Entities
        };
```

```
    req.app.render('showpost', context, function(err, html) {
      if(err) {throw err;}
      console.log('응답 웹 문서 : ' + html);
      res.end(html);
    });

……
```

showpost 함수를 호출할 때는 저장된 글의 ObjectId 값이 id 파라미터로 전달되므로 이 값을 사용해 글 객체를 조회할 수 있습니다. 글 객체를 조회하면 showpost 뷰 템플릿을 사용해 자동으로 웹 문서를 만든 후 클라이언트에 응답으로 보냅니다.

파일의 마지막 부분에는 module.exports의 속성으로 다음 두 함수를 추가합니다.

참조 파일 | BoardExample>/routes/post.js

```
……

module.exports.addpost = addpost;
module.exports.showpost = showpost;
```

저장된 글을 응답으로 보내기 위해 showpost 뷰 템플릿을 사용하도록 코드를 작성하였습니다. [views] 폴더에 showpost.ejs 파일을 추가하고 응답으로 보낼 웹 문서의 템플릿을 작성합니다.

참조 파일 | BoardExample>/views/showpost.ejs

```
……

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

……

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

……
```

```
<div class = "ui blue fluid card">
  <div class = "content">
    <%
      var curTitle = posts._doc.title;
      var curContents = posts._doc.contents;
      var curWriter = posts._doc.writer.id;

      // html-entities module is required in post.js
      var entities = new Entities();
      var decodedContents = entities.decode(curContents);
    %>

    <div id = "titleOutput" class = "header">< % = curTitle %></div>
    <div class = "meta">
      <span class = "right floated time">1일전</span>
      <span class = "category">일반</span>
    </div>
    <div id = "contentsOutput" class = "description">
      <% = decodedContents %>
    </div>
  </div>
  <div class = "extra content">
    <div class = "right floated author">
      <img class = "ui avatar image" src = "/public/author.png"> <% = curWriter %>
    </div>
  </div>
</div>

.....
```

데이터베이스에 저장된 글을 보여 줄 때는 Semantic UI로 예쁘게 꾸며 줍니다. 따라서 Semantic UI 라이브러리 파일들(js 파일과 css 파일)을 [public] 폴더 안에 복사하여 넣어 줍니다. 그런데 요즈음에는 웹에서 글을 작성할 때 단순 텍스트뿐 아니라 입력한 글자의 색상이나 글자 크기를 바꾸는 경우가 많습니다. 이렇게 웹 문서에 입력한 글자의 속성을 바꿀 수 있게 하려면 저장된 글을 HTML 코드로 만들어야 합니다. 그리고 HTML로 저장된 글을 웹 문서에 다시 보여 줄 때도 HTML 태그를 해석하여 보여 주도록 만들어야 합니다.

HTML 태그를 해석하여 웹 문서 형태로 보여 주는 다양한 jQuery 라이브러리가 나와 있습니다. 여기에서는 여러 jQuery 라이브러리 중에서 cleditor를 사용합니다. 자세한 내용은 다음 사이트를 참고하세요. 사이트에서 필요한 파일을 다운로드할 수 있으며, [public] 폴더 안에 [cleditor] 폴더를 만들어 그 안에 필요한 파일을 추가합니다.

▶ http://premiumsoftware.net/cleditor/

글을 보여 줄 때뿐만 아니라 글을 쓸 때도 HTML 편집기를 사용해야 웹 문서 형태로 글을 만들어 저장할 수 있습니다. 글쓰기 기능을 만들 때는 ckeditor를 사용합니다. 자세한 내용은 다음 사이트를 참고하세요. 사이트에서 필요한 파일을 다운로드할 수 있으며, [public] 폴더 안에 [ckeditor] 폴더를 만들어 그 안에 필요한 파일을 추가합니다.

▶ http://ckeditor.com/

글쓰기 기능을 만들기 위해 [public] 폴더의 addpost.html 파일을 만든 후 웹 문서를 구성하는 다음 코드를 입력합니다.

참조 파일 | BoardExample>/public/addpost.html

```
.....

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

  <script>
    $(document).ready(function() {
      CKEDITOR.config.extraPlugins = 'colorbutton';
      CKEDITOR.env.isCompatible = true;
      CKEDITOR.replace('contents');
    });
  </script>

.....

  <form class = "content ui form" method = "post" action = "/process/addpost">
    <h4 class = "ui dividing header">글 쓰기 입력 항목</h4>

    <div class = "two fields">
      <div class = "field">
        <label>제목</label>
        <input type = "text" name = "title" placeholder = "제목 입력">
      </div>
      <div class = "field"></div>
    </div>
```

```
<div class = "field">
  <label>내용</label>
  <textarea id = "contents" name = "contents" rows = "10" cols = "80"></textarea>
</div>

<div class = "two fields">
  <div class = "field">
    <label>작성자</label>
    <input type = "text" name = "writer" placeholder = "작성자 아이디 입력">
  </div>
  <div class = "field"></div>
</div>

<input class = "ui submit button" type = "submit" value = "추가" name = "">
<input class = "ui button" type = "button" value = "취소" name = ""
    onclick = "javascript : history.back()">
</form>

.....
```

ckeditor 라이브러리를 사용하면 단순한 입력 상자인 〈textarea〉 태그가 글자 속성을 바꿀 수 있는 입력 상자로 바뀝니다. 즉, 글자를 예쁘게 꾸밀 수 있도록 만들어 줍니다. 앞에서 살펴본 코드에는 〈textarea〉 태그의 id 속성 값을 contents로 입력하였는데 이 id를 사용해 자바스크립트에서 ckeditor 라이브러리를 적용했습니다. 이렇게 하면 해당 〈textarea〉 태그 부분이 글자를 예쁘게 꾸밀 수 있는 영역으로 바뀝니다.

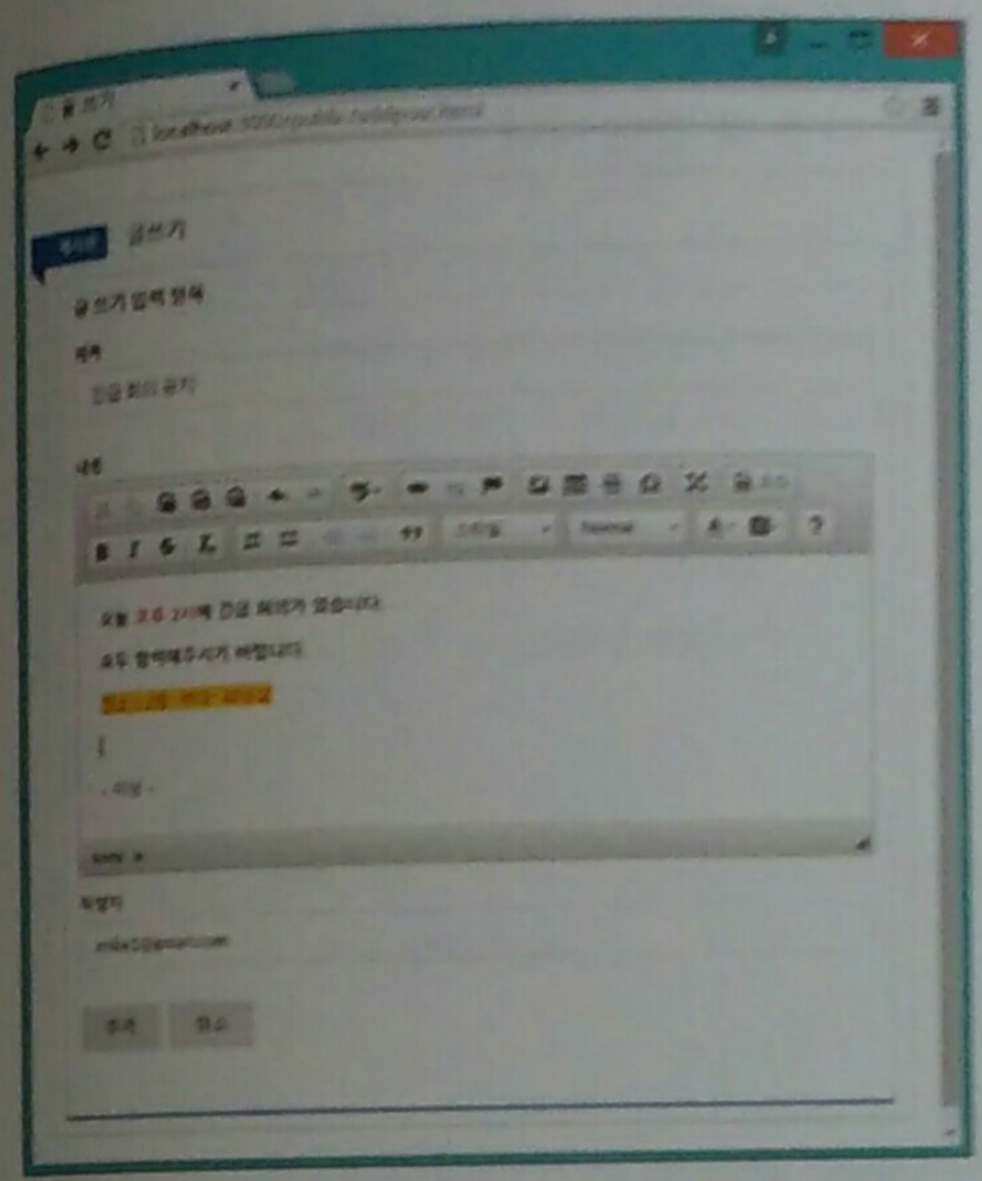
마지막으로 클라이언트에서 보낸 요청을 처리할 라우팅 함수를 config.js 파일에 등록합니다.

참조 파일 | BoardExample>/config/config.js

```
.....

module.exports = {

.....

  route_info : [
    {file : './post', path : '/process/addpost', method : 'addpost', type : 'post'}
    ,{file : './post', path : '/process/showpost/ : id', method : 'showpost', type : 'get'}
  ],

.....
```

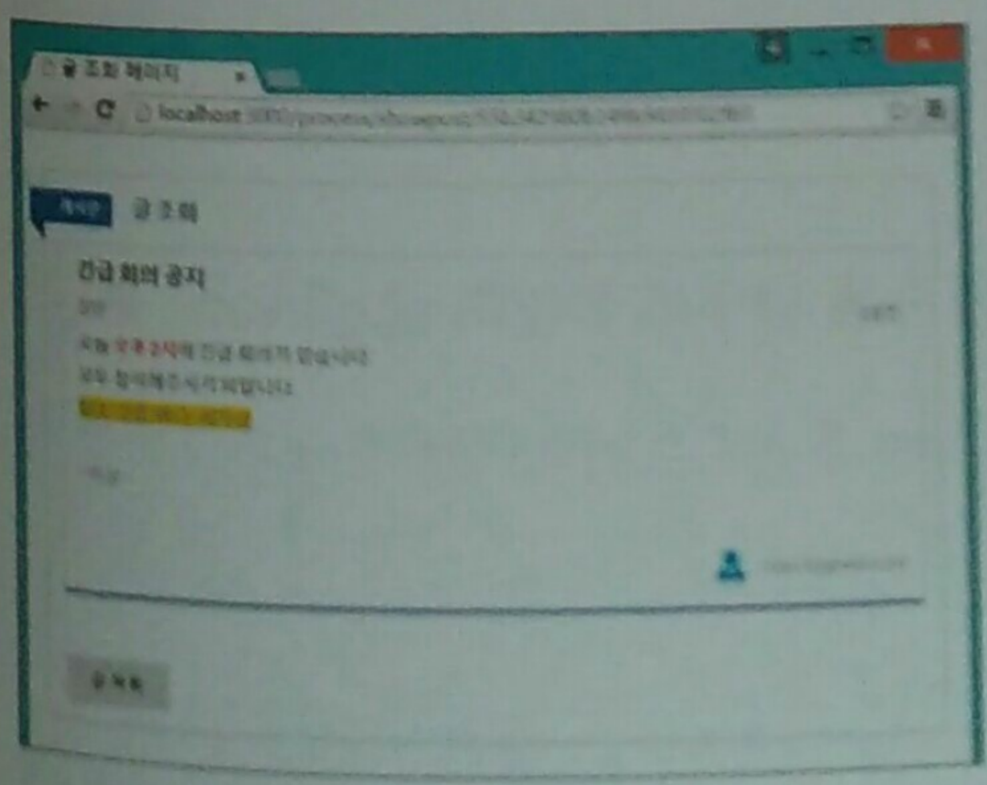
이제 app.js 파일을 실행한 후 웹 브라우저에서 글쓰기 페이지를 열어 봅니다.

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



▲웹 브라우저에서 글쓰기 페이지를 열었을 때의 화면

글의 내용을 입력하는 부분에는 글의 속성도 변경할 수 있는 아이콘들이 나타납니다. 이렇게 웹 문서 편집기가 보이면 글자를 입력한 후 글자색이나 배경색을 바꿀 수 있습니다. 작성자를 입력할 수 있도록 만든 입력 상자에는 글쓴이 ID를 입력하는데, 이 ID는 users 컬렉션에 저장된 사용자의 데이터를 가져와 사용하겠다는 의미입니다. 따라서 users 컬렉션에 저장된 이메일 주소 중 하나를 입력해야 합니다. 예전에 추가해 둔 사용자 이메일 중의 하나를 입력한 후 [추가] 버튼을 누르면 글이 추가된 후 글 내용을 보여 주는 화면으로 전환됩니다.



▲글 저장 후 글 조회 화면으로 전환된 경우

지금까지 글을 작성한 후 작성한 글을 조회하는 기능까지 만들어 보았습니다. 화면에 보이는 내용이나 모양은

14-3 | 페이지 단위로 글 목록 조회하기

이제 앞에서 설명한 페이지 단위 조회 방식으로 게시판 글 목록을 조회하는 기능을 만들어 보겠습니다. 이미 알고 있겠지만 글 목록을 조회하는 기능을 만드는 순서는 다음과 같습니다.

(1) 라우팅 함수 만들기
(2) config.js 파일에 라우팅 함수 등록하기
(3) 응답 웹 문서를 구성할 뷰 템플릿 만들기
(4) 글 목록 조회를 요청하는 웹 페이지 만들기

이 단계에 따라 하나씩 만들어보겠습니다.

1단계 - 라우팅 함수 만들기

우선 첫 번째 단계로 [routes] 폴더의 post.js 파일을 열고 listpost 함수를 추가합니다.

참조 파일 | BoardExample>/routes/post.js

```
……

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

  var paramPage = req.param('page');
  var paramPerPage = req.param('perPage');
  var database = req.app.get('database');

  if(database.db) {
    // 1. 글 목록
    var options = {
      page : paramPage,
      perPage : paramPerPage
    }

    database.PostModel.list(options, function(err, results) {
      if(err) { throw err; }
      if(results) {
        console.dir(results);
```

```
    // 전체 문서 객체 수 확인
    database.PostModel.count().exec(function(err, count) {
      res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});

      // 뷰 템플릿을 사용하여 렌더링한 후 전송
      var context = {
        title : '글 목록',
        posts : results,
        page : parseInt(paramPage),
        pageCount : Math.ceil(count / paramPerPage),
        perPage : paramPerPage,
        totalRecords : count,
        size : paramPerPage
      };

      req.app.render('listpost', context, function(err, html) {
        res.end(html);
      });
    });

……

module.exports.listpost = listpost;
```

클라이언트로부터 전달받은 데이터는 각각 현재 페이지의 인덱스와 한 페이지에 보여 줄 글의 개수를 의미하는 page와 perPage로 되어 있습니다. 이 두 개의 값을 사용해 데이터베이스에서 해당 데이터를 조회합니다. 사용자에게 보여 줄 응답 웹 문서를 만들려면 전체 글의 개수를 알아야 하므로 모델 객체의 count 메소드를 호출하여 전체 글의 개수를 확인합니다. 전체 글의 개수로 계산하여 페이지 개수를 알아낸 다음 뷰 템플릿을 사용해 응답 웹 문서를 만들 때 page와 perPage를 모두 사용합니다.

2단계 - config.js 파일에 라우팅 함수 등록하기

두 번째 단계로 추가한 listpost 함수를 config.js 파일에 등록합니다.

참조 파일 | BoardExample>/config/config.js

```
……

module.exports = {
……
```

```
route_info : [
......
    ,{file : './post', path : '/process/listpost', method : 'listpost', type : 'post'}
    ,{file : './post', path : '/process/listpost', method : 'listpost', type : 'get'}
  ],

......
```

GET 방식이나 POST 방식을 모두 사용해서 listpost 함수를 호출할 수 있게 만들려면 똑같은 내용의 코드 두 줄을 추가할 때 type 속성 값만 바꿔 주면 됩니다.

3단계 - 응답 웹 문서를 구성할 뷰 템플릿 만들기

세 번째 단계로 넘어가서 응답 웹 문서를 구성하는 뷰 템플릿을 만들어 보겠습니다. [views] 폴더에 listpost.ejs 파일을 만든 후 템플릿을 구성하는 다음 코드를 입력합니다.

참조 파일 | BoardExample>/views/listpost.ejs

```
......

  <div class = "ui grid">
    <div class = "two wide column">번호</div>
    <div class = "eight wide column">제목</div>
    <div class = "two wide column">작성자</div>
    <div class = "two wide column">작성일</div>
    <div class = "two wide column">조회수</div>
  </div>

  <div class = "ui very relaxed selection celled list">
    <%  var noStart = (pageCount - page) * perPage;

      for(var i = 0; i < posts.length; i++) {
        var curTitle = posts[i]._doc.title;
        var curContents = posts[i]._doc.contents;
        var curWriter = posts[i]._doc.writer.email;
        var curNo = noStart - i;
    %>
    <div class = "item">
      <div class = "ui grid">
        <div class = "two wide column">><% = curNo %></div>
        <div class = "fourteen wide column"
```

```
onclick = "javascript : window.location = '/process/showpost/<% = posts[i]._id %>'">
<div class = "ui header">
  <h4 class = "ui left aligned header">
    <% = curTitle %>
  </h4>
  <h5 class = "ui right aligned orange header">
    <% = curWriter%>   
    2015-06-01   
    12   
  </h5>
</div>
</div>
</div>
</div>
<% } %>


</div>

.....
```

뷰 템플릿 파일에는 글의 제목과 작성자, 작성일 등을 목록 형태로 보여 줍니다. posts 변수에 들어 있는 데이터는 for문을 사용해 보여 줄 수 있으며, 각각의 글 앞에 붙는 번호는 전체 글의 개수 값을 사용해 계산한 후 화면에 뿌려 줍니다.

글 목록을 표시했으니 이제 글 목록 아래쪽에 있는 페이지 선택 버튼을 추가합니다.

참조 파일 | BoardExample>/views/listpost.ejs

```
.....

<div class = "tiny ui basic buttons">
  <%
    if(page > 0) {
  %>
    <div class = "ui icon button" onclick = "listpost(0, <% = perPage %>)">
      <i class = "fast backward icon"></i>
    </div>
    <div class = "ui icon button" onclick = "listpost(<% = (page - 1) %>, <% = perPage %>)">
      <i class = "left chevron icon"></i>
    </div>
  <%
```

```
  } else {
%>
  <div class = "ui disabled icon button">
    <i class = "fast backward icon"></i>
  </div>
  <div class = "ui disabled icon button">
    <i class = "left chevron icon"></i>
  </div>
<%
  }
%>
<%
  var initial = Math.floor(page / 10) * 10;
  console.log('initial : ' + initial);

  var max = pageCount - initial;
  if(max > 10) {
    max = initial + 10;
  } else {
    max = initial + max;
  }
  console.log('max : ' + max);

  for(var i = initial; i < max; i++) {
    if(i != page) {
%>
      <div class = "ui button" onclick = "listpost(<% = i %>, <% = perPage %>)">
        <% =  i %>
      </div>
<%
    } else {
%>
      <div class = "ui active basic button">
        <% =  i %>
      </div>
<%
    }
  }
%>
<%
  if((page+1) < pageCount) {
%>
  <div class = "ui icon button" onclick = "listpost(<% = (page + 1) %>, <% = perPage %>)">
    <i class = "right chevron icon"></i>
```

```
</div>
<div class = "ui icon button" onclick = "listpost(<% = (pageCount - 1) %>,
            <% = perPage %>)">
  <i class = "fast forward icon"></i>
</div>
<%
  } else {
%>
<div class = "ui disabled icon button">
  <i class = "right chevron icon"></i>
</div>
<div class = "ui disabled icon button">
  <i class = "fast forward icon"></i>
</div>
<%
  }
%>

</div>

.....
```

코드가 상당히 긴데 코드가 길어진 이유는 다음 두 가지로 설명할 수 있습니다. 현재 보고 있는 페이지 번호의 색상을 다른 번호와 다르게 하거나 선택할 수 없도록 만들어야 합니다. 그리고 첫 페이지 또는 끝 페이지로 가기 버튼도 해당 페이지가 열린 상태에서는 비활성화되어야 합니다. 이런 버튼이 모두 앞에서 설명한 페이지 단위 조회 방식에서 사용하는 변수로 만든 것임을 이해하면 코드가 그리 복잡하다고 느끼진 않을 것입니다.

주의! 페이지 단위 처리가 복잡하다고 느껴진다면 노드에서 제공하는 pagination 패키지를 사용하는 방법도 있습니다. 하지만 페이지 단위 처리 방식은 기본적으로 이해하고 있어야 합니다.

4단계 - 글 목록 조회를 요청하는 웹 페이지 만들기

네 번째는 [public] 폴더에 listpost.html 페이지를 만드는 단계입니다. 이 페이지에는 단순히 조회 요청에 필요한 버튼만 넣을 것입니다.

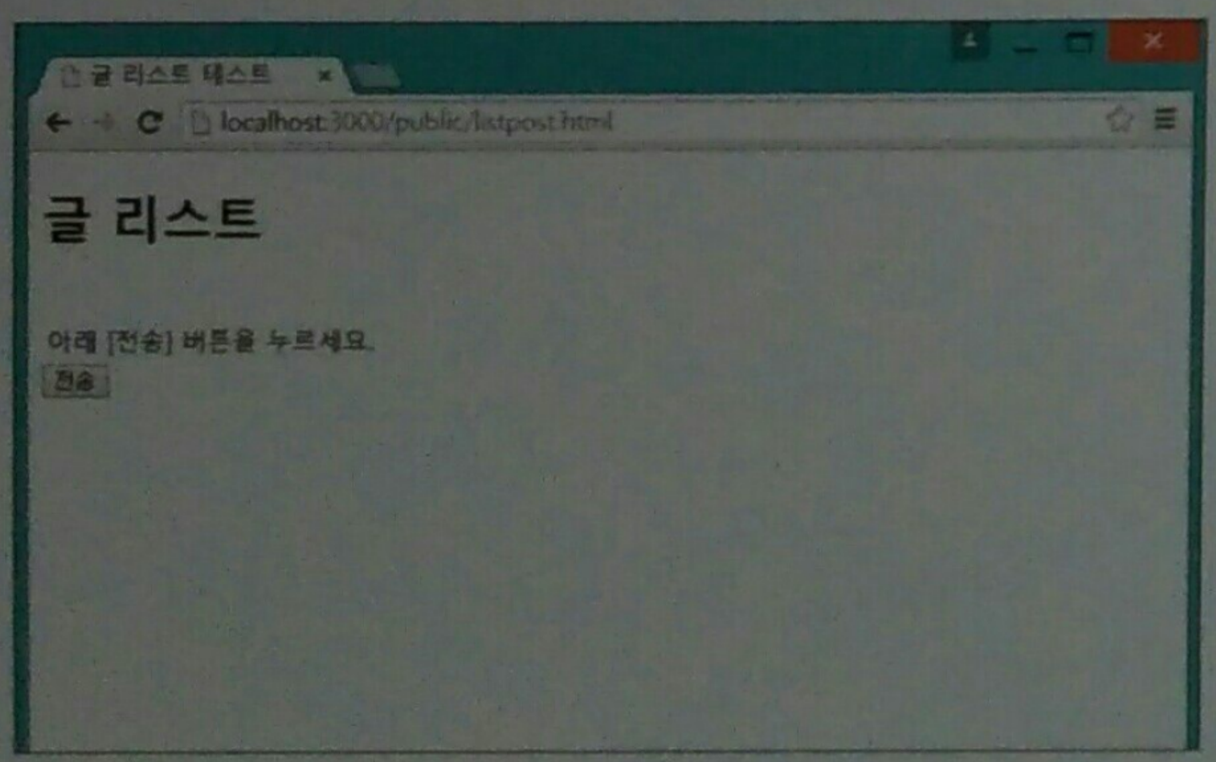
참조 파일 | BoardExample>/public/listpost.html

```
.....

<form method = "post" action = "/process/listpost">
  <table>
    <tr>
      <td><label>아래 [전송] 버튼을 누르세요.</label></td>
    </tr>
  </table>
  <input type = "hidden" name = "page" value = "0">
  <input type = "hidden" name = "perPage" value = "2">
  <input type = "submit" value = "전송" name = "">
</form>

.....
```

게시판 목록 조회를 요청하는 기능은 보통 메인 페이지나 로그인 페이지를 사용합니다. 그렇기 때문에 웹 문서 화면에 버튼 하나만 만드는 경우는 거의 없습니다. 이 장에서는 테스트를 위해 간단하게 버튼 하나만 만들어서 페이지를 꾸미고 있지만 여러분은 필요한 내용을 모두 넣어서 만들어 보세요. [전송] 버튼을 누르면 POST 방식으로 /process/listpost 패스를 요청합니다.

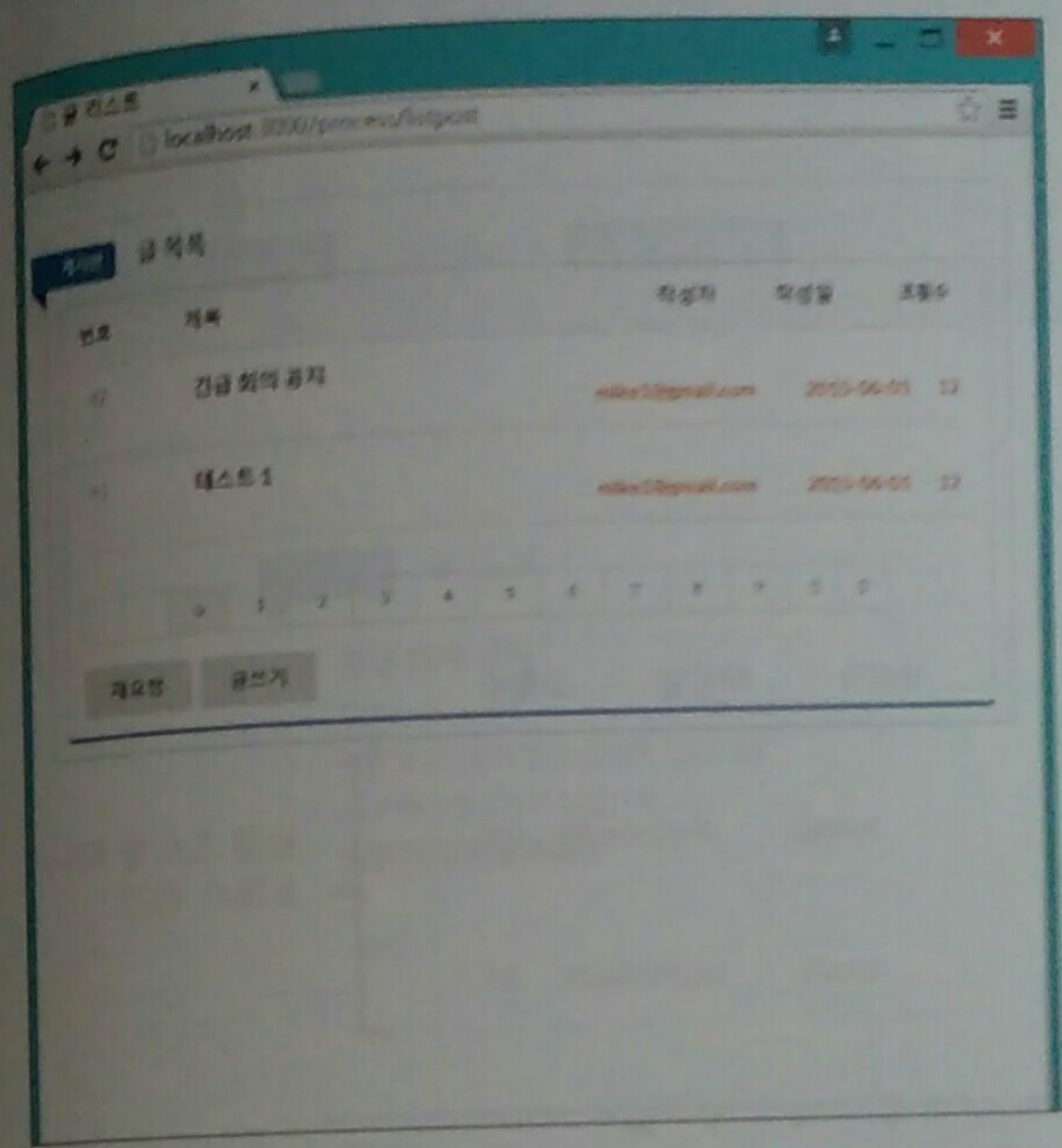
이제 app.js 파일을 실행한 후 웹 브라우저를 열고 다음 사이트에 접속합니다.

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



▲ 게시판의 글 목록을 요청하는 웹 문서를 열어 본 화면

[전송] 버튼을 클릭하면 게시판의 글 목록이 조회됩니다.



▲ 게시판의 글 목록 조회 화면

게시판 목록의 각 항목을 클릭하면 글 조회 화면으로 넘어갑니다. 아래쪽에 있는 [글쓰기] 버튼을 누르면 글쓰기 화면으로 넘어가는 것도 확인할 수 있습니다. 만약 한 페이지에 보이는 글의 수를 5개나 10개로 하고 싶다면 글 목록을 요청할 때 perPage 파라미터 값을 5나 10으로 바꾸면 됩니다.

지금까지 게시판에 글을 쓰거나 작성한 글 목록을 조회하는 기능을 만들어 보았습니다. 이 과정에서 페이지 단위로 처리하는 방법은 반드시 이해하고 넘어가기 바랍니다. 이 방법을 이해했다면 게시판이 아닌 다른 어떤 데이터도 그 양에 관계없이 조회할 수 있는 기능을 만들 수 있을 것입니다.

여기서 만든 게시판 기능은 실무에 바로 적용하기에는 충분하지 않습니다. 예를 들어, 댓글을 쓰거나 댓글을 표시하는 기능은 넣지 않았습니다. 하지만 데이터베이스 스키마와 모델 객체 기능을 이미 넣어 두었으므로 댓글 기능을 추가하는 과정은 그리 어렵지 않습니다. 여러분이 직접 만들어 보기 바랍니다.

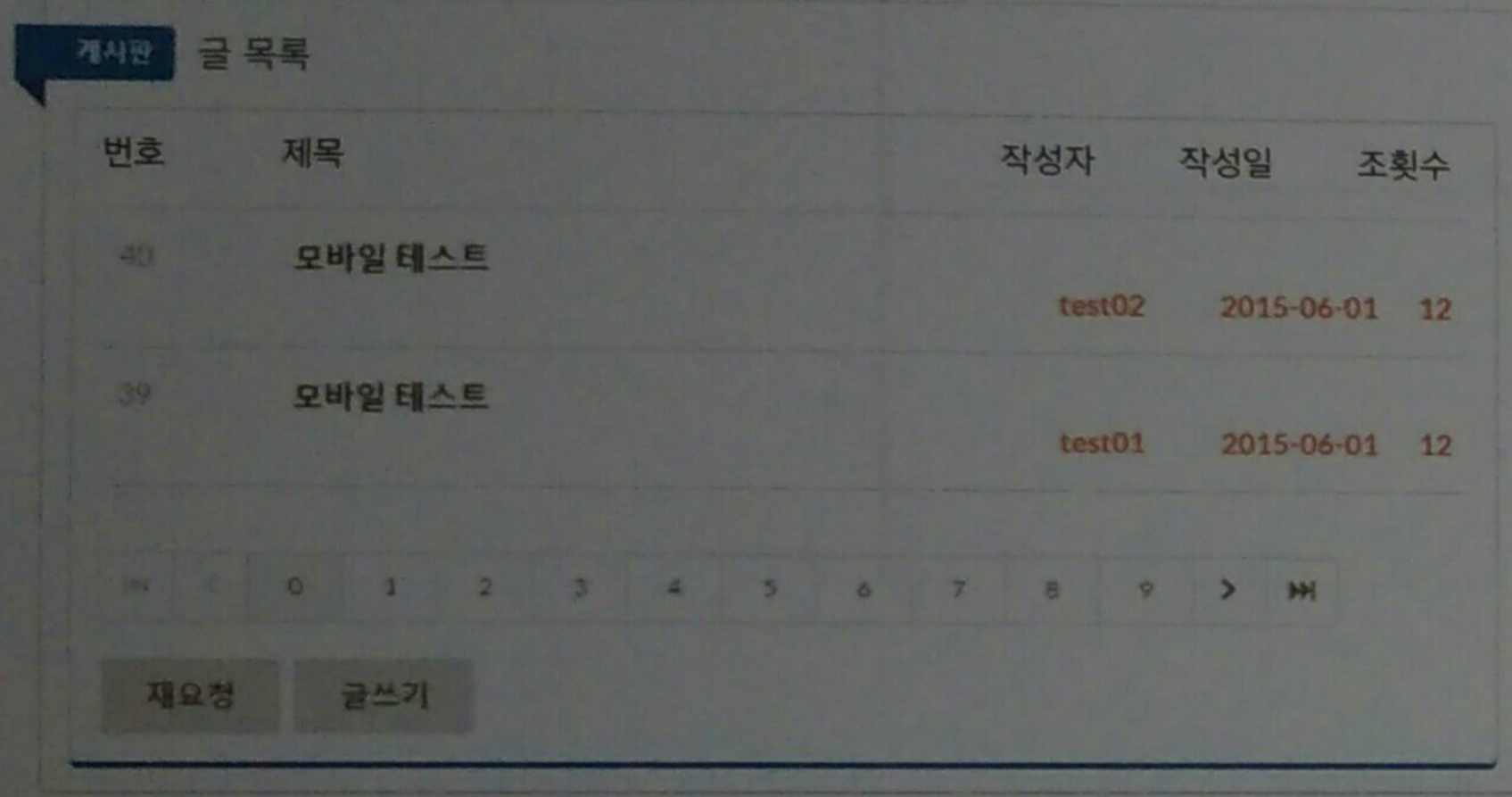
게시판 기능을 모두 만들었으니 다음 장에서는 여러분이 만든 게시판을 클라우드에 올려 보겠습니다.

Do it!

도전 문제

01. 글을 조회할 때마다 조횟수가 올라가도록 게시판 기능을 수정해 보세요.

프로젝트 소스 Mission21 난이도 초급 ★☆☆☆



Hint!

1 BoardExample 프로젝트에서는 글을 조회해도 조횟수가 증가하지 않습니다. 왜냐하면 조횟수를 임의의 숫자로 고정시켜 놓았습니다.

2 글을 조회할 때마다 데이터베이스에 저장된 조횟수를 증가시켜 글 목록 화면에 실제 조횟수가 표시되도록 만듭니다.

3 글 목록을 요청하면 증가된 조횟수를 서버의 데이터베이스에서 조회한 후 글 목록 웹 문서를 만들어 응답으로 보내 줍니다.

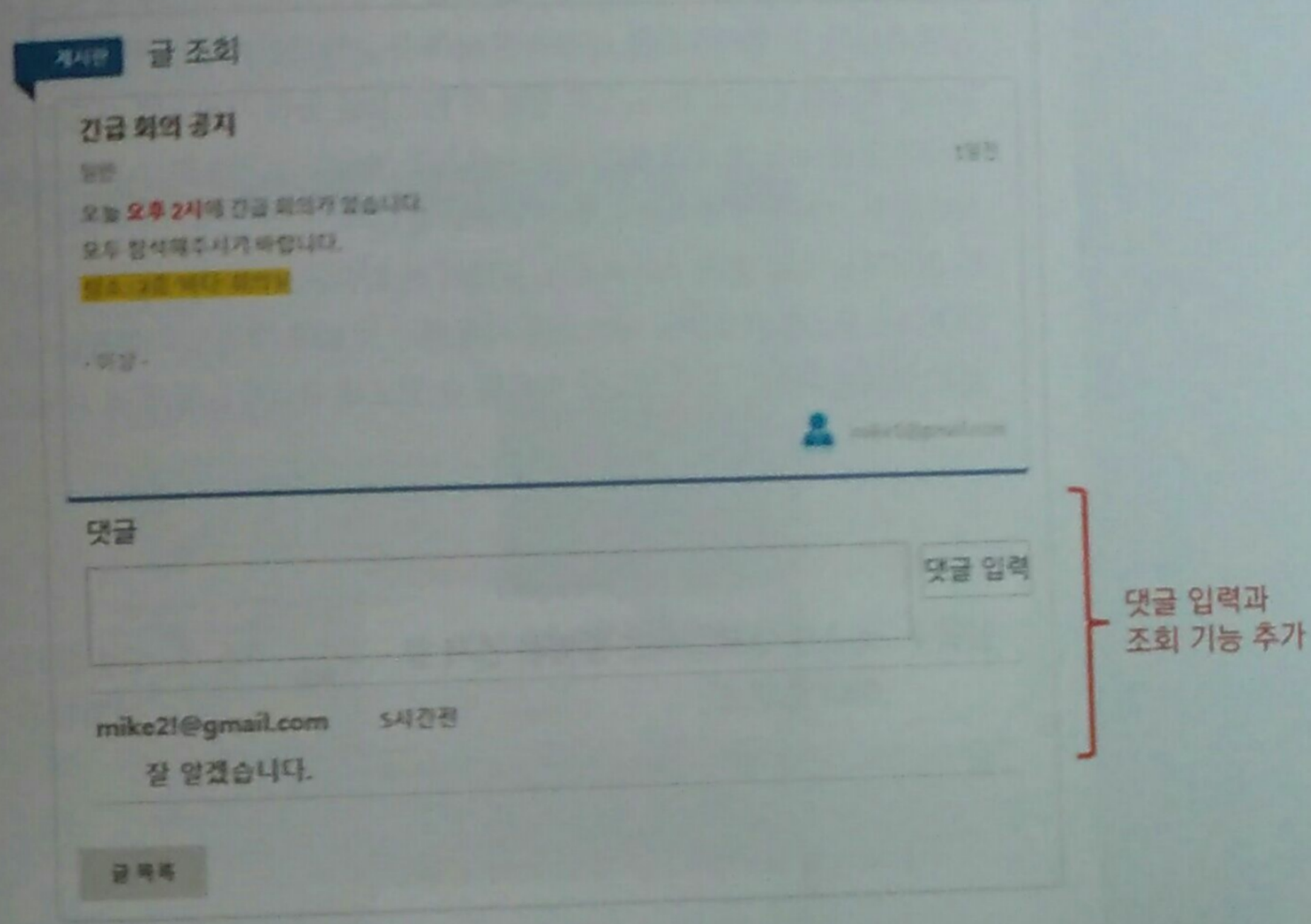
MEMO

DO it!

도전 문제

02. 댓글을 쓰면 서버의 데이터베이스에 댓글을 저장하고, 게시판을 조회할 때 댓글도 같이 조회되도록 만들어 보세요.

프로젝트 소스 Mission22 난이도 중급 ★★☆☆



Hint!

1 BoardExample 프로젝트에는 댓글 입력이나 조회 기능이 들어 있지 않습니다. 다만 데이터베이스 스키마에는 댓글을 저장하는 comment 칼럼이 추가되어 있습니다.

2 게시판 글 조회 화면 아래쪽에 댓글을 입력할 수 있는 기능과 댓글 목록을 보여 주는 기능을 추가합니다.

3 댓글은 게시판 글 목록에 있는 하나의 글을 선택했을 때 글 아래쪽에 나타나게 만들고, 필요할 때 댓글을 더 입력할 수 있도록 입력 상자와 [댓글 저장] 버튼을 추가합니다.

MEMO