15 클라우드에 올리기

노드로 만든 프로그램을 서버 장비에 올려 일반 사용자들에게 서비스할 수 있습니다. 그런데 IDC 서버를 임대하면 비용이 많이 들기 때문에 무료로 제공하는 서버나 상대적으로 가격이 저렴한 클라우드(Cloud) 서비스를 많이 사용합니다. 특히 안드로이드나 아이폰용 앱을 위해 만든 웹 서버나 모바일 서버는 서비스를 시작한 초기에는 많은 수익이 발생하지 않을 확률이 크기 때문에 무료 제공 클라우드에 프로그램을 올려 서비스하는 경우가 많습니다. 또 클라우드를 사용하면 세계 곳곳의 사용자들에게 안정적으로 서비스할 수 있다는 장점이 있습니다. 노드로 만든 프로그램을 올려 서비스할 수 있는 클라우드는 여러 가지가 있습니다. 각각의 클라우드 서비스가 제공하는 기능이나 방식이 조금씩 다르므로 서비스 목적에 맞는 클라우드를 선택해야 합니다.

이 장에서는 여러 클라우드 서비스 중에서 헤로쿠(Heroku), 오픈시프트(OpenShift), 아마존(Amazon) 클라우드 서비스를 사용하는 방법에 대해 알아보겠습니다. 전 세계 사람들이 여러분이 만든 프로젝트를 이용할 수 있도록 클라우드에 올려 보세요.

질문으로 시작하기

질문1. 소스를 업로드하는 방법을 먼저 알아야 할까요? ➡ 15-1. Git 클라이언트 설치하기

질문2. 헤로쿠 클라우드에 올리고 싶나요? ➡ 15-2. 헤로쿠 클라우드에 올리기

질문3. 오픈시프트 클라우드에 올리고 싶나요? ➡ 15-3. 오픈시프트 클라우드에 올리기

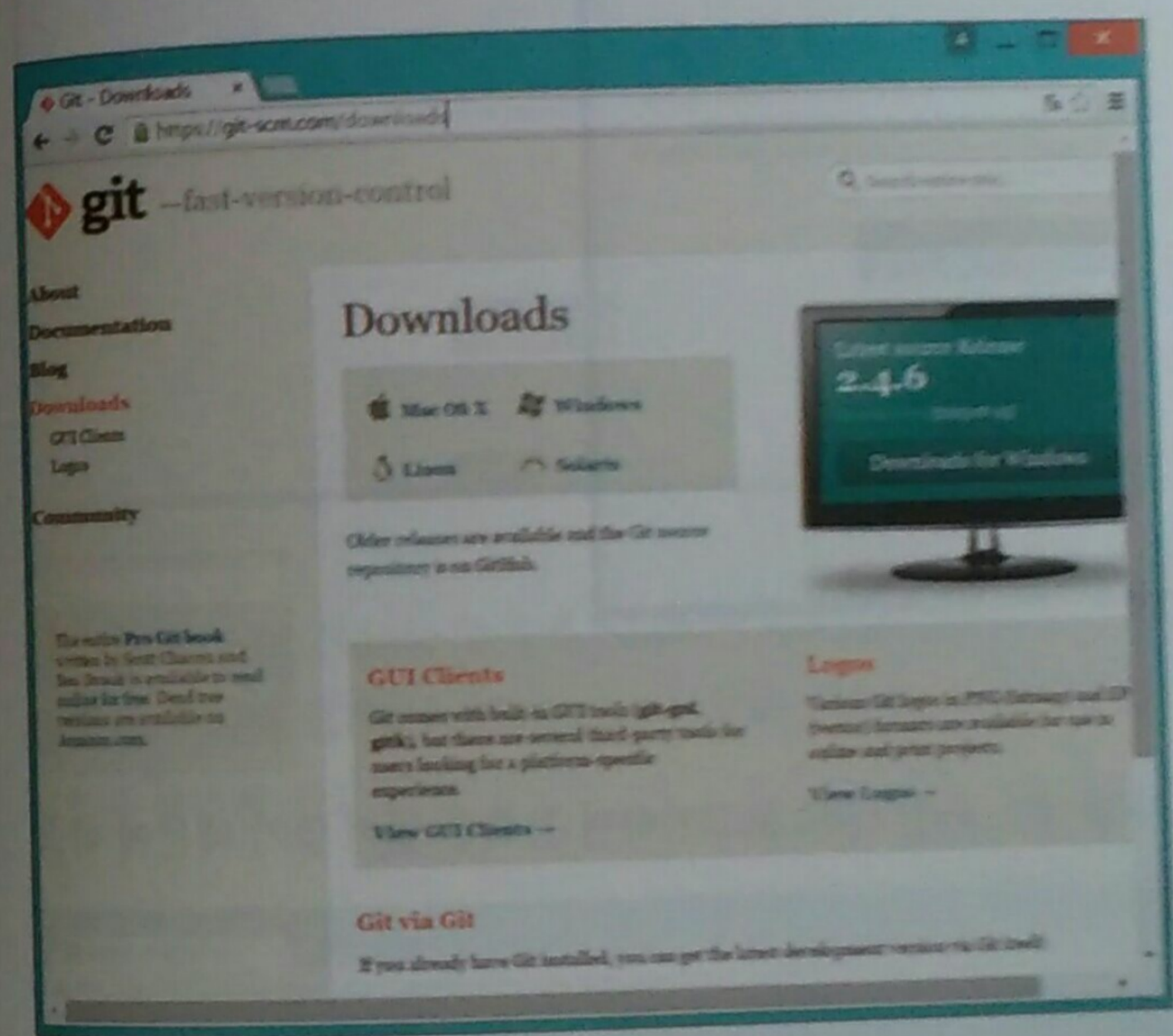
질문4. 아마존 클라우드에 올리고 싶나요? ➡ 15-4. 아마존 클라우드에 올리기

질문5. 전 세계로 배포할 때 필요한 기능을 알고 있나요? ➡ 15-5. 배포에 필요한 기능 알아보기

15-1 | Git 클라이언트 설치하기

대부분의 클라우드 서비스는 소스 파일을 업로드하거나 다운로드할 때 git를 사용합니다. 따라서 클라우드에 올리기 전에 먼저 **git 클라이언트**를 설치해야 합니다. 윈도우용 git 클라이언트 프로그램은 다음 사이트에서 다운로드해서 설치합니다.

▶ https://git-scm.com/downloads



▲ git 클라이언트를 다운로드할 수 있는 사이트

GUI Clients 목록을 눌러 GitHub Desktop 설치 파일을 다운로드한 후 실행하면 어렵지 않게 설치할 수 있습니다. 설치가 끝나면 명령 프롬프트에서 git를 사용할 수 있습니다. 필요하다면 GUI 프로그램을 추가로 설치하여 사용할 수도 있습니다.

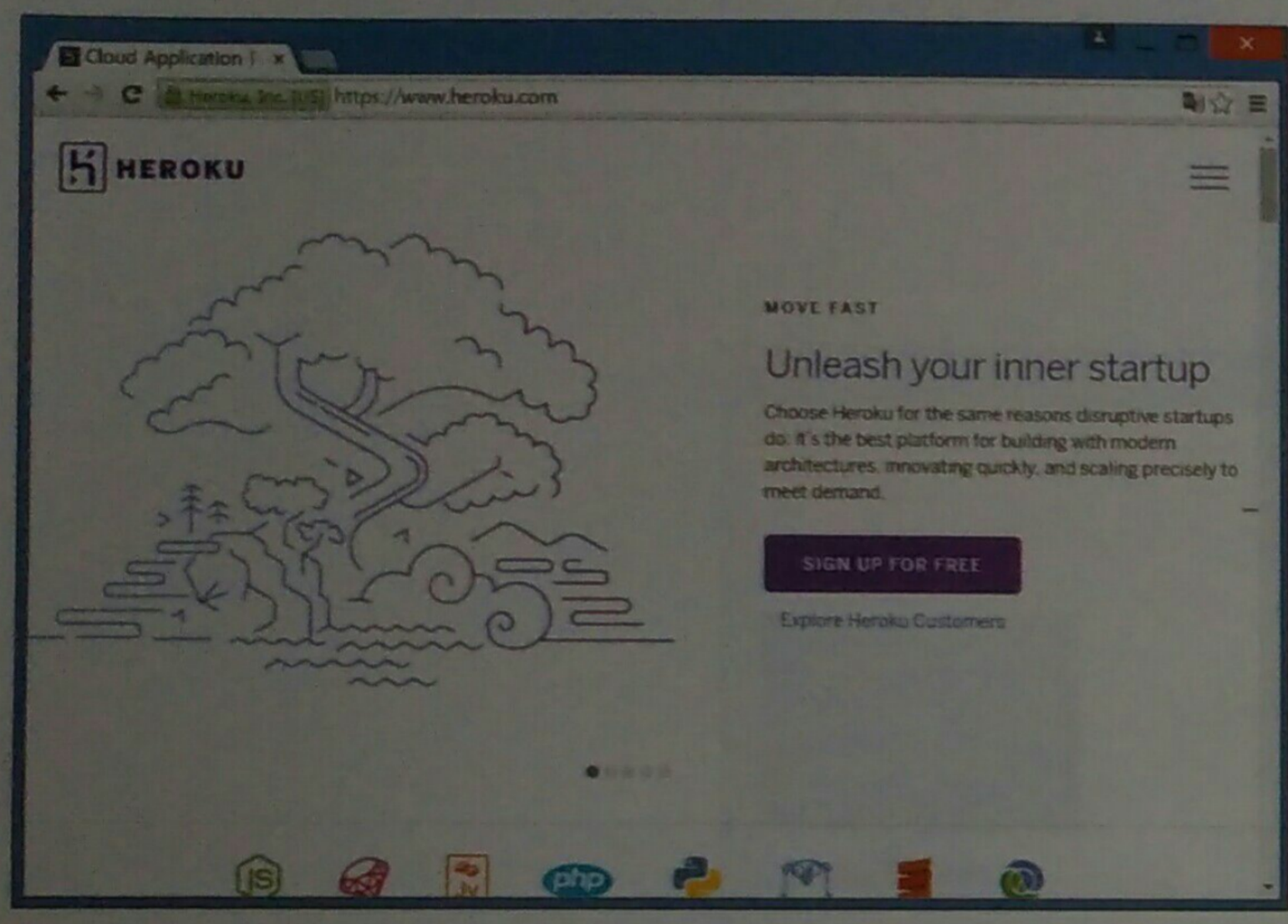
15-2 | 헤로쿠 클라우드에 올리기

헤로쿠(Heroku) 클라우드는 세일즈포스닷컴에서 운영하고 있으며, 데이터 저장소나 데이터 분석, 모니터링 등 다양한 **애드온(Add-on)** 기능을 제공합니다. 이처럼 헤로쿠를 대표적인 클라우드 서비스로 자리잡게 만드는 데 중요한 역할을 한 것이 바로 편리하게 사용할 수 있는 애드온 기능입니다.

헤로쿠 사이트에 회원가입하고 앱 만들기

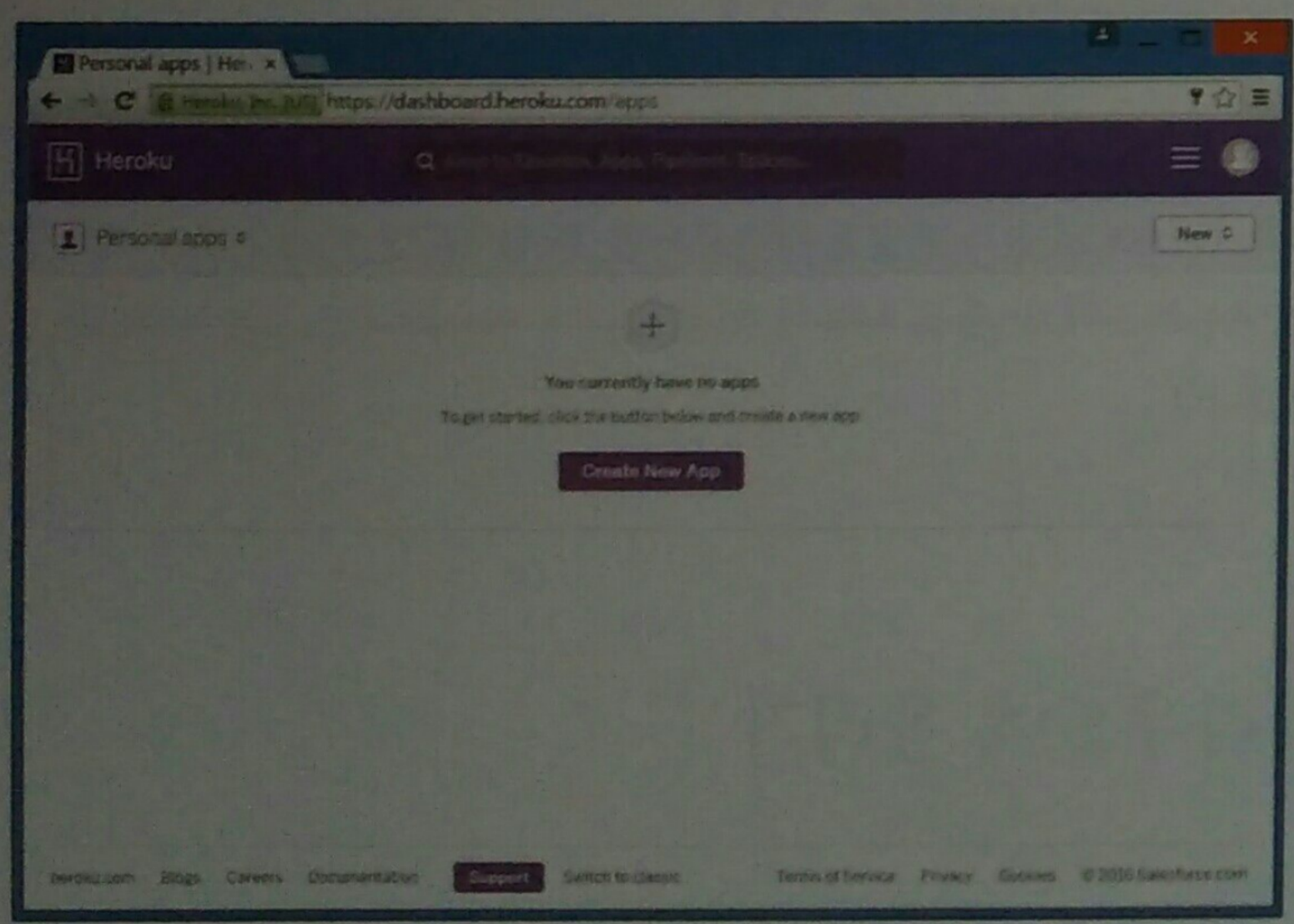
헤로쿠 클라우드를 사용하려면 다음 사이트에 접속한 후 먼저 회원가입을 합니다.

헤로쿠 사이트 ▶ http://heroku.com



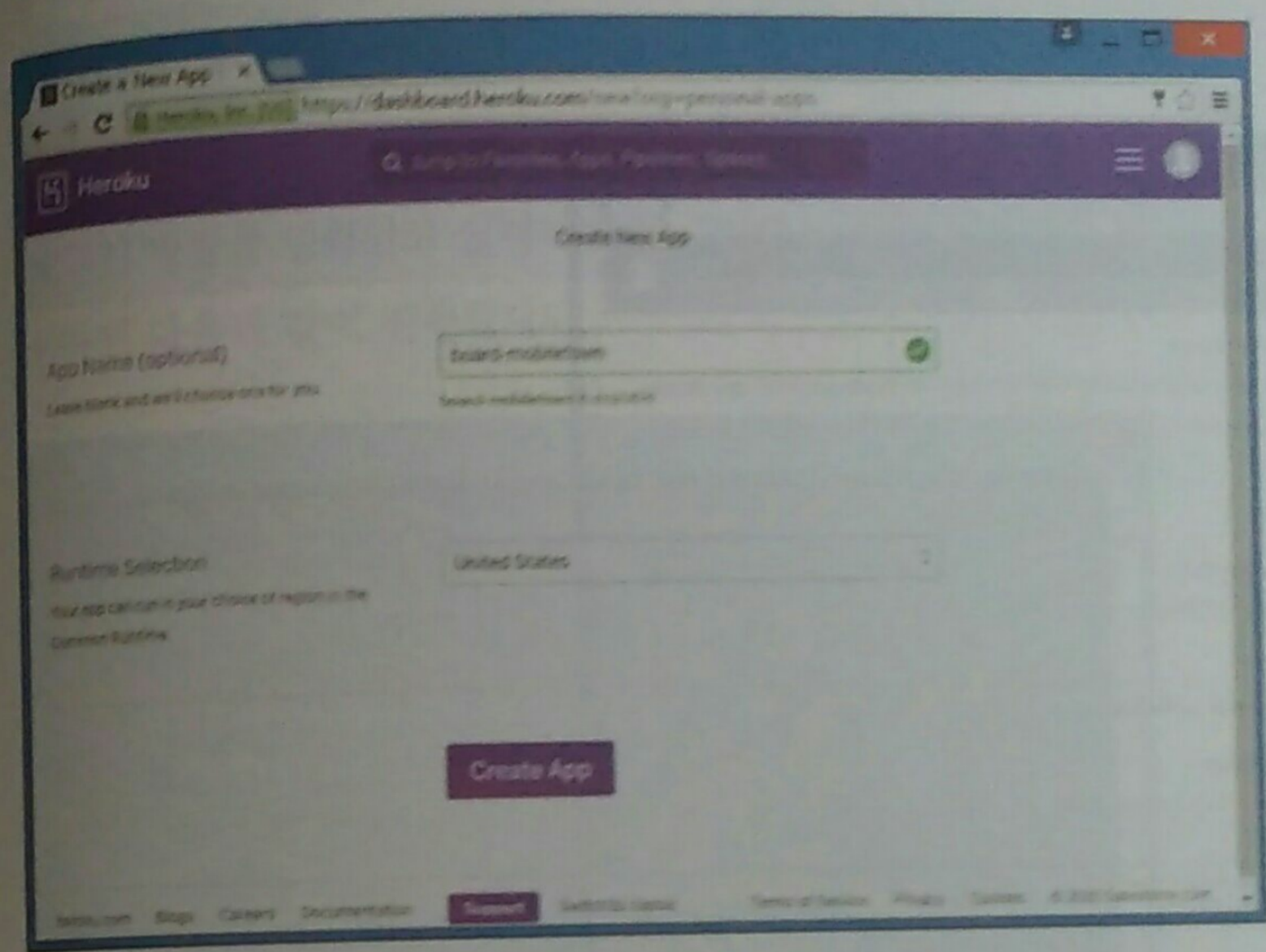
▲ 헤로쿠 사이트

이메일 계정을 ID로 사용해서 회원가입을 합니다. 회원 ID로 로그인하면 등록한 앱이 없기 때문에 다음처럼 새로운 앱을 등록하라는 화면이 나타납니다.



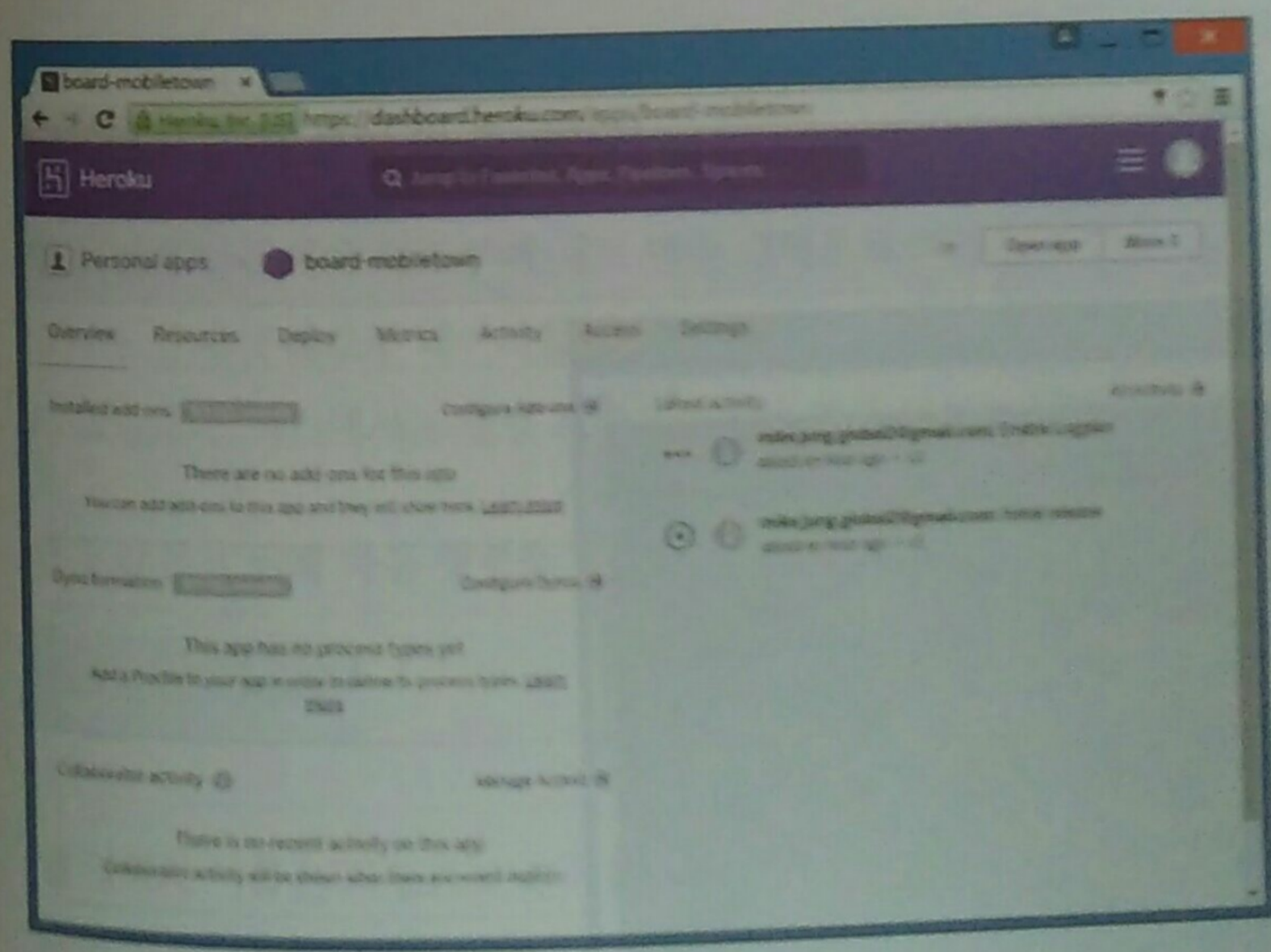
▲ 새로운 앱을 만들도록 안내하는 페이지

화면 중앙에 있는 [Create New App] 버튼을 클릭하면 새로운 앱을 생성하는 화면으로 바뀝니다. App Name 항목에는 이름을 입력하지 않아도 앱 이름이 자동으로 만들어지므로 입력 상자를 비워 놓아도 문제는 없습니다.



◀ 새로운 앱을 만드는 페이지

여기에서는 App Name 항목에 board-OOOOOOO을 입력했습니다. OOOOOOO은 앱의 고유 이름을 만들기 위해 입력하는 문자입니다. 각자 고유한 이름을 입력하면 됩니다. 그러고 나서 [Create App] 버튼을 클릭합니다.

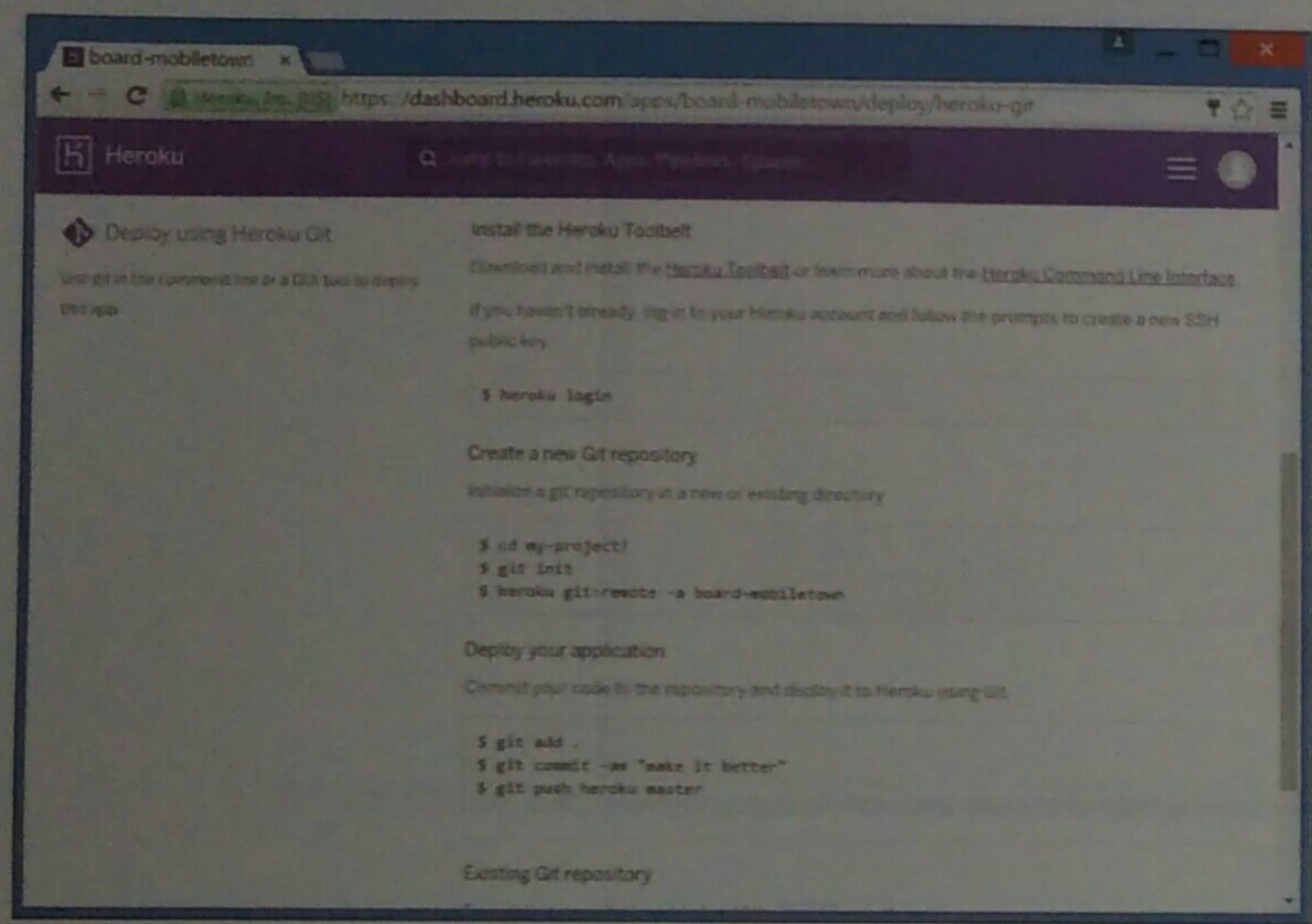


◀ 헤로쿠에 새로운 App을 만든 후 나타나는 메인 화면

화면 위쪽의 타이틀 영역 바로 아래에는 Personal apps 〉 board-mobiletown이라고 표시되어 있습니다. 이것은 사용자가 만든 앱 중에서 현재 board-mobiletown을 선택한 상태이며, 해당 앱 정보를 아래쪽에 표시한다는 의미입니다.

Toolbelt 설치하기

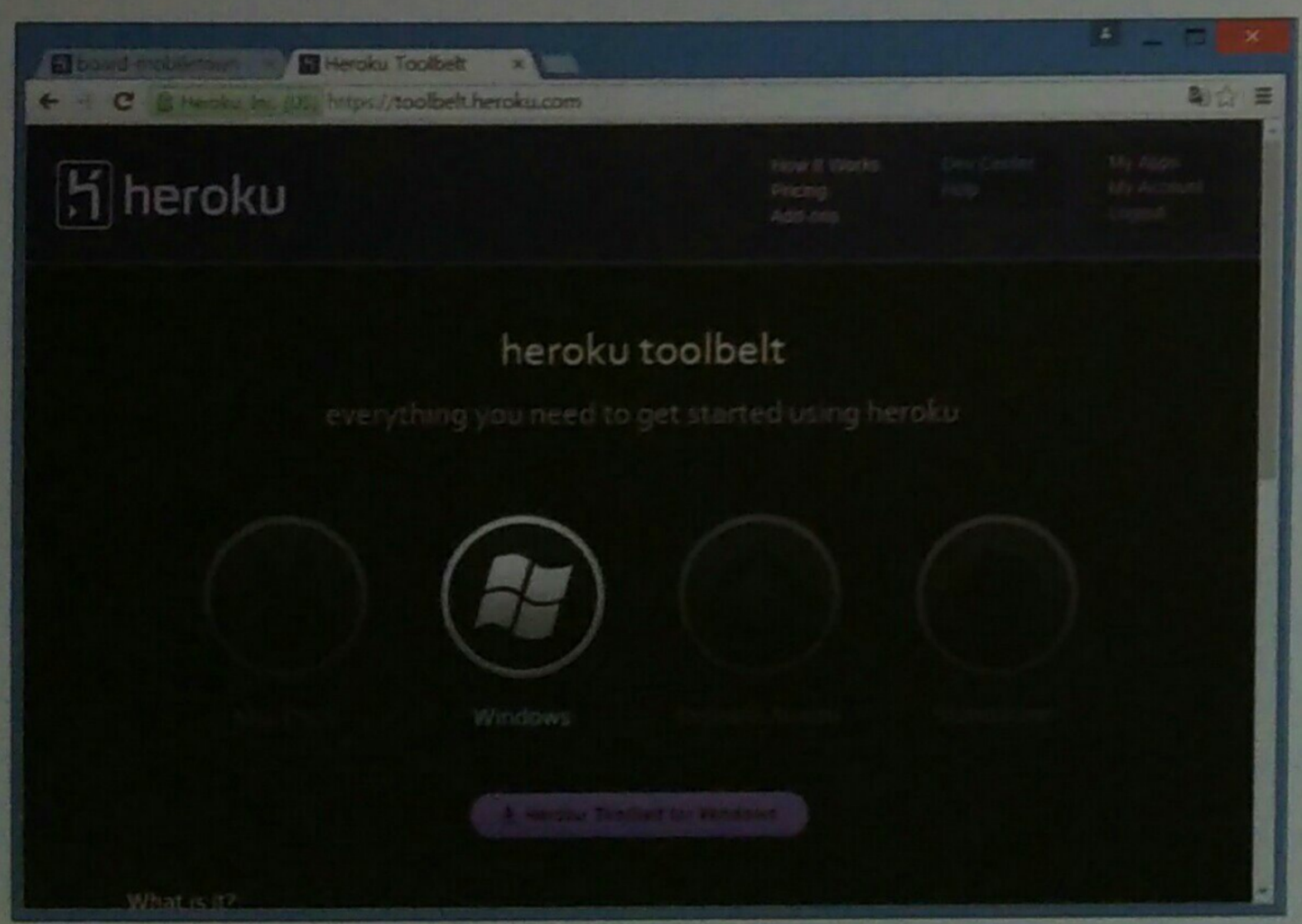
헤로쿠에서는 여러분이 만든 노드 프로그램을 쉽게 배포할 수 있도록 **툴벨트**(Toolbelt)를 제공합니다. 화면의 메뉴 중에서 [Deploy] 메뉴를 선택하고 아래쪽으로 스크롤을 내려 보면 툴벨트의 설치 및 사용 방법이 나와 있습니다.



▲ [Deploy] 메뉴를 선택했을 때 보이는 툴벨트 안내 메시지

그중에 'Heroku Toolbelt' 링크를 눌러 다운로드하여 설치할 수 있습니다. 또는 아래 사이트 주소에 접속하여 툴벨트를 설치하면 됩니다.

툴벨트 사이트 ▶ http://toolbelt.heroku.com

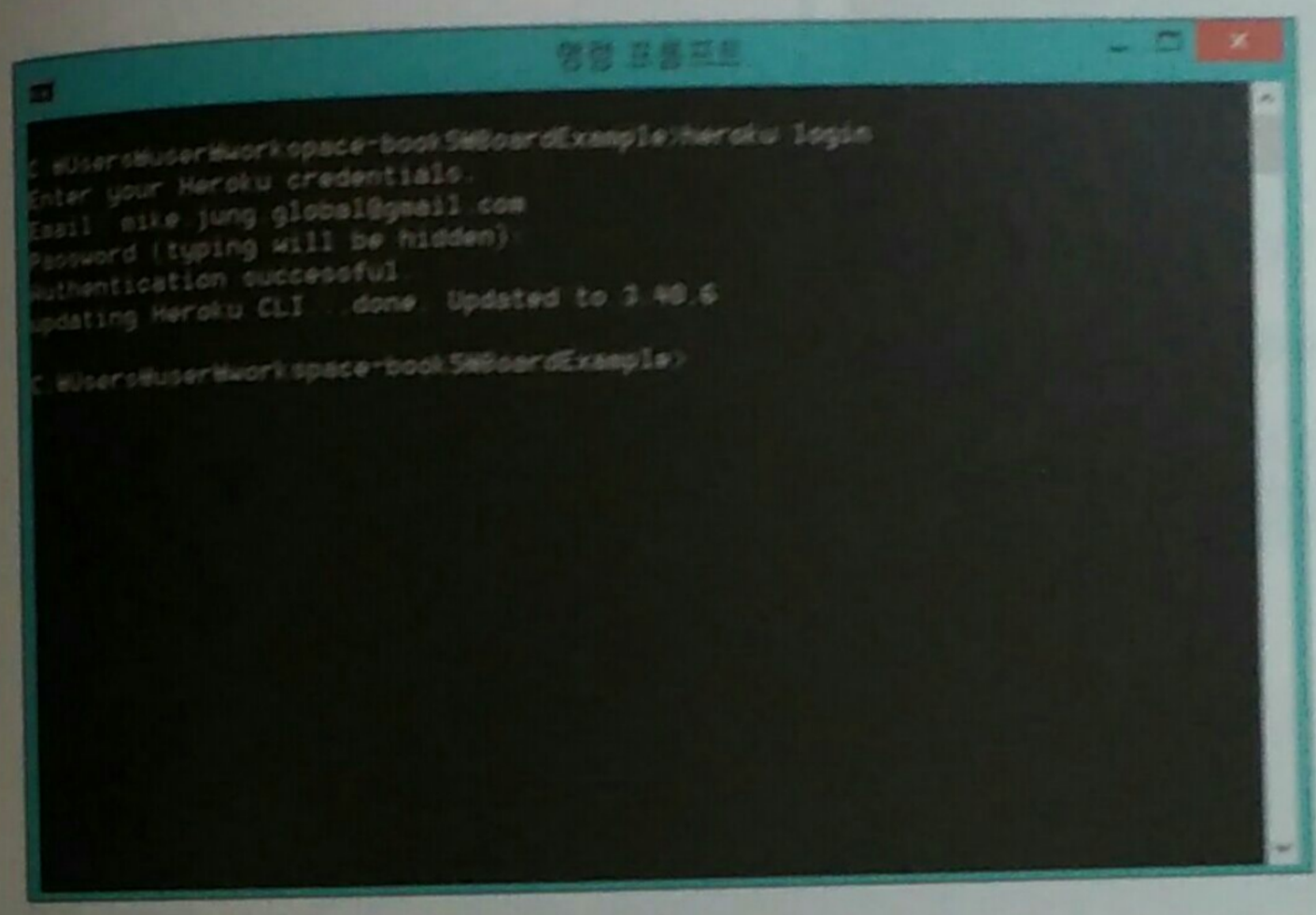


▲ 툴벨트를 다운로드하여 설치할 수 있는 사이트

툴벨트를 사용하려면 먼저 [Heroku Toolbelt for Windows] 버튼을 클릭하여 다운로드한 후 설치합니다. 설치가 끝나면 명령 프롬프트를 열고 앞 장에서 만든 BoardExample 프로젝트 폴더로 이동합니다. 그다음 로그인 명령을 입력합니다.

```
% heroku login
```

로그인 명령을 입력하고 실행하면 로그인에 사용할 이메일과 비밀번호를 물어봅니다. 정상적으로 로그인하면 다음과 같이 진행됩니다.



▲명령 프롬프트에서 로그인을 진행한 화면

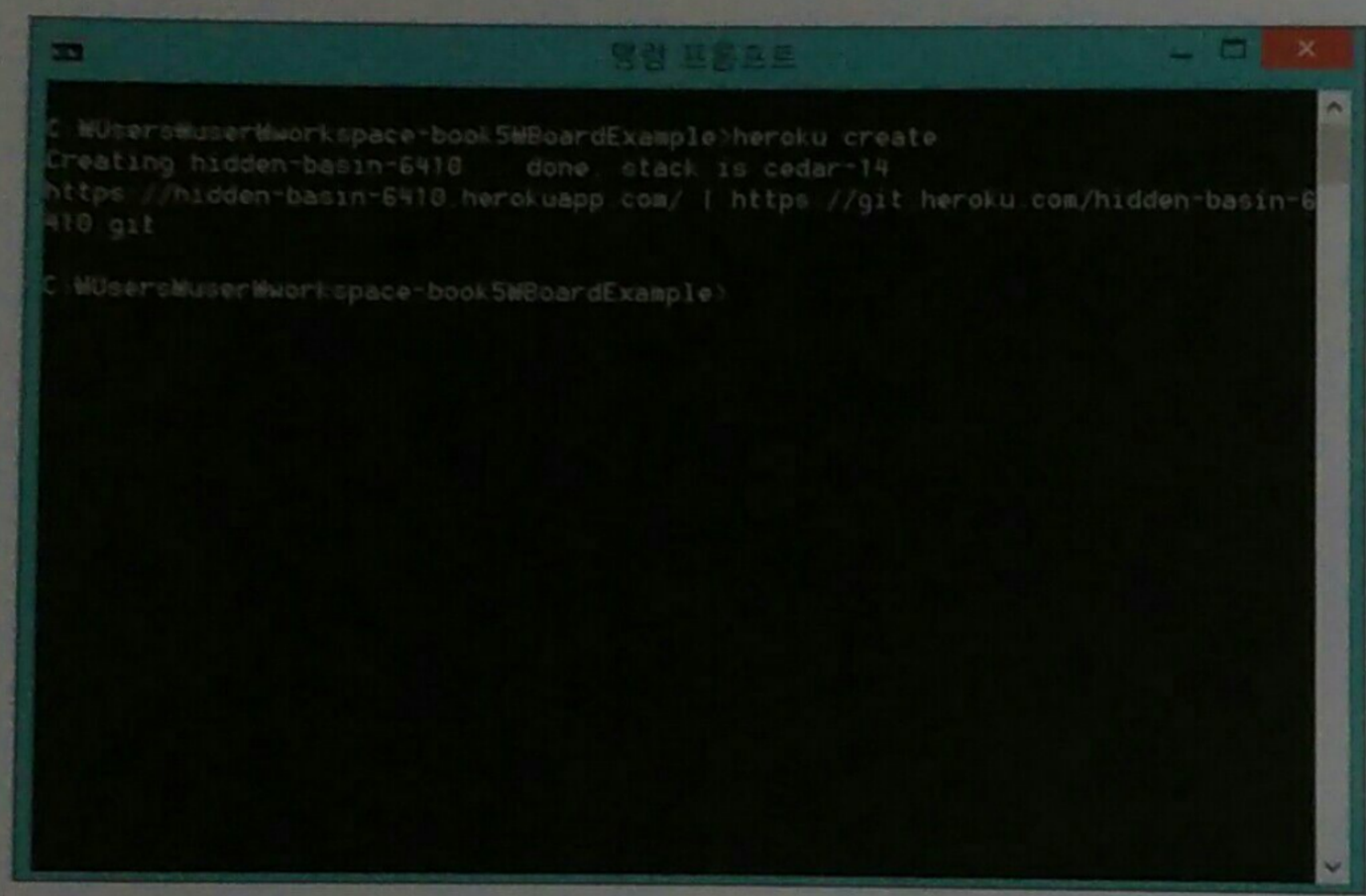
배포하기 위한 설치 및 준비하기

앱을 먼저 만들어야 헤로쿠에 새로 만든 영역으로 여러분이 만든 프로젝트 파일을 올리고 실행할 수 있습니다. 앞에서 헤로쿠 페이지를 열고 board-mobiletown 앱을 만들었지만 명령 프롬프트에서 만들 수도 있습니다. 여기에서는 명령 프롬프트에서 앱을 만드는 방법을 설명합니다.

앱을 만들기 위해 다음 명령을 입력하고 Enter 키를 누릅니다.

```
% heroku create
```

앱 영역의 이름은 자동으로 부여됩니다. 여기에서는 hidden-basin-6410로 만들어졌으며 이 앱 영역은 https://hidden-basin-6410.herokuapp.com으로 접근할 수 있습니다.



◀ 새로운 앱 영역이 만들어진 경우

앱 영역의 이름은 무작위로 만들어지지만 원하는 이름으로 변경할 수도 있습니다. 자동으로 만들어진 hidden-basin-6410이라는 이름을 boardexample로 바꾸고 싶다면 다음과 같이 입력합니다.

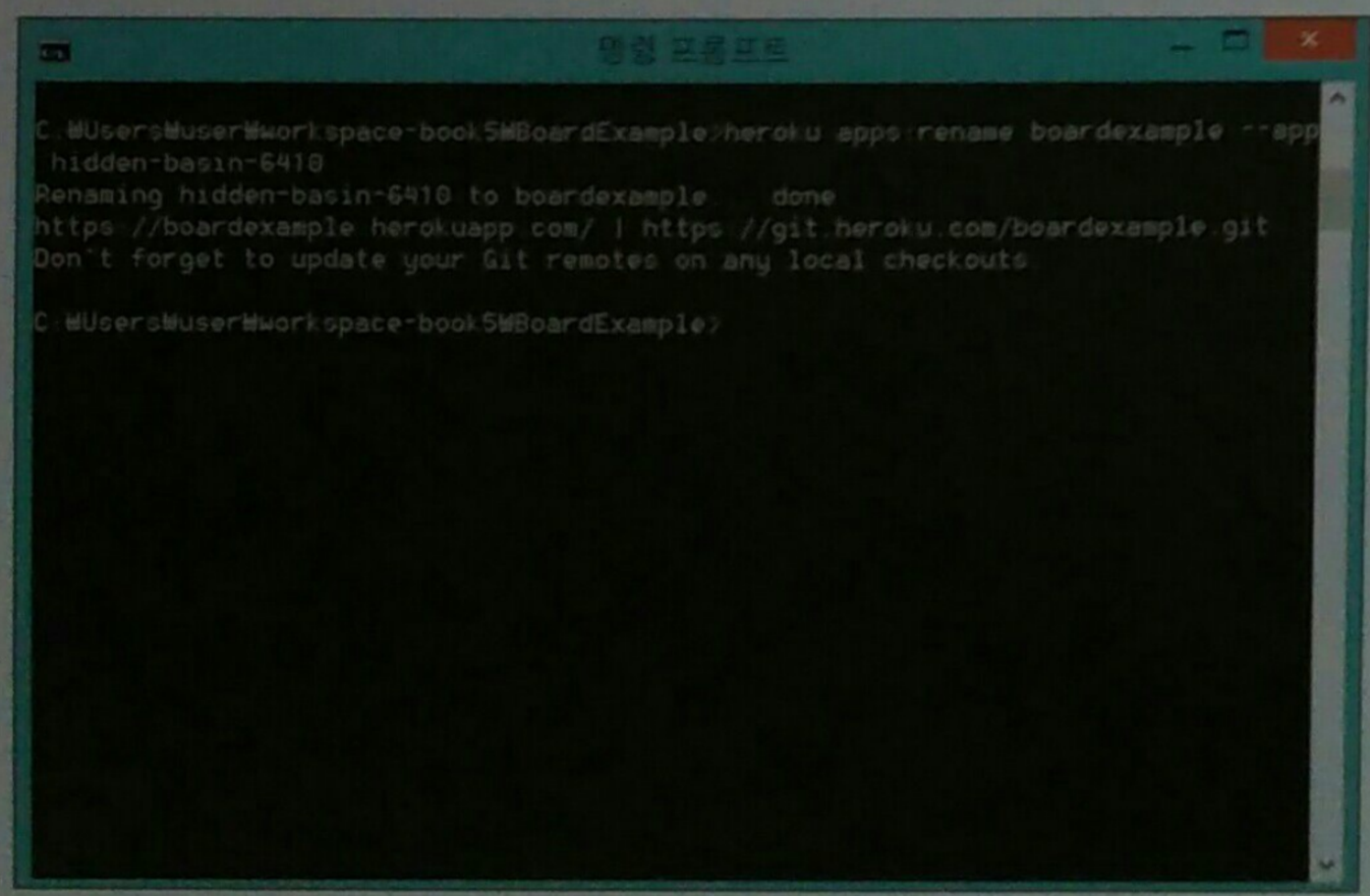
```
% heroku apps:rename boardexample --app hidden-basin-6410
```

정박사의 한마디 앱 이름이 중복되면 안 돼요

앱 이름은 헤로쿠 사이트에서 고유한 값으로 구별할 수 있어야 합니다. 따라서 여러분은 이 책에서 입력한 boardexample이 아닌 다른 이름을 입력해야 합니다.

또 헤로쿠 사이트에서 이미 앱을 만들었다면 명령 프롬프트에서 다시 만들지 않도록 이 단계를 건너뛰면 됩니다. 연습 삼아 명령 프롬프트에서 앱을 만들어 보려면 앱 이름을 반드시 다른 것으로 바꾸세요.

앱 영역의 이름이 정상적으로 바뀌었다면 이제부터는 https://boardexample.herokuapp.com/ 주소로 접근할 수 있습니다.



◀ 앱 영역의 이름을 바꾼 경우

이제 헤로쿠에 소스 파일을 올릴 수 있습니다. 여러분이 만든 BoradExample 프로젝트 파일을 올리려면 먼저 두 가지 작업을 진행해야 합니다. 첫 번째로 실행 파일을 추가합니다. 헤로쿠 클라우드를 사용할 때는 Procfile 파일을 프로젝트에 추가로 만들어야 이 파일에 들어 있는 코드로 노드 프로그램을 실행할 수 있습니다.

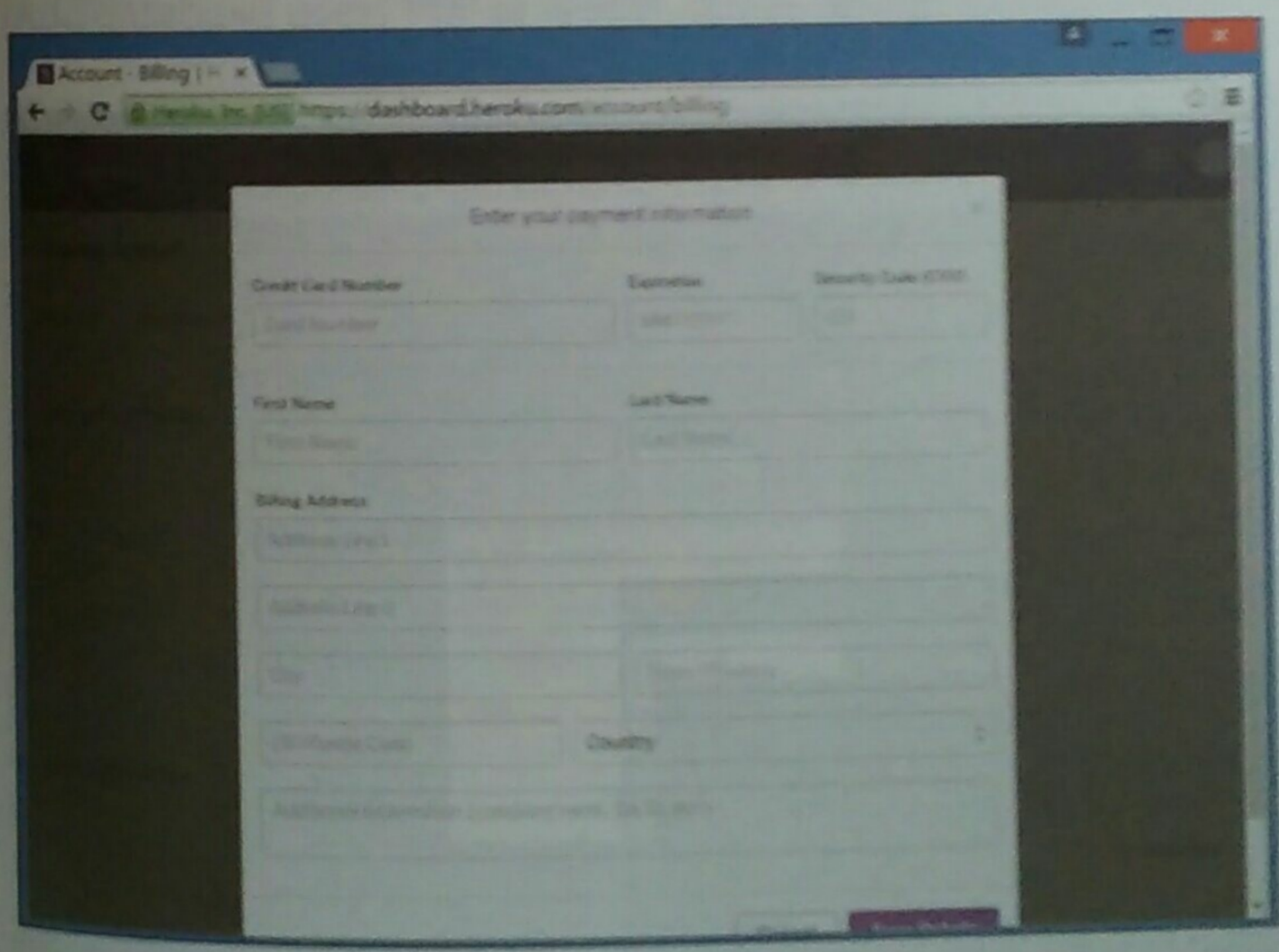
텍스트 편집기를 열어서 확장자가 없는 Procfile 파일을 만들고 다음 코드를 입력합니다. 노드 프로그램을 실행했을 때 app.js 메인 파일을 실행 대상으로 한다는 의미입니다.

참조 파일 | BoardExample>/Procfile

```
web: node app.js
```

두 번째로 몽고디비 데이터베이스를 사용할 수 있도록 mLab MongoDB라는 애드온(add-on)을 설치한 후 설정을 진행합니다. 그런데 애드온 프로그램을 설치하려면 신용카드 정보를 입력해야 합니다. 기본 사용량까지는 무료지만 일정 사용량을 넘으면 유료로 전환해야 하므로 가격 정책을 잘 참조하기 바랍니다.

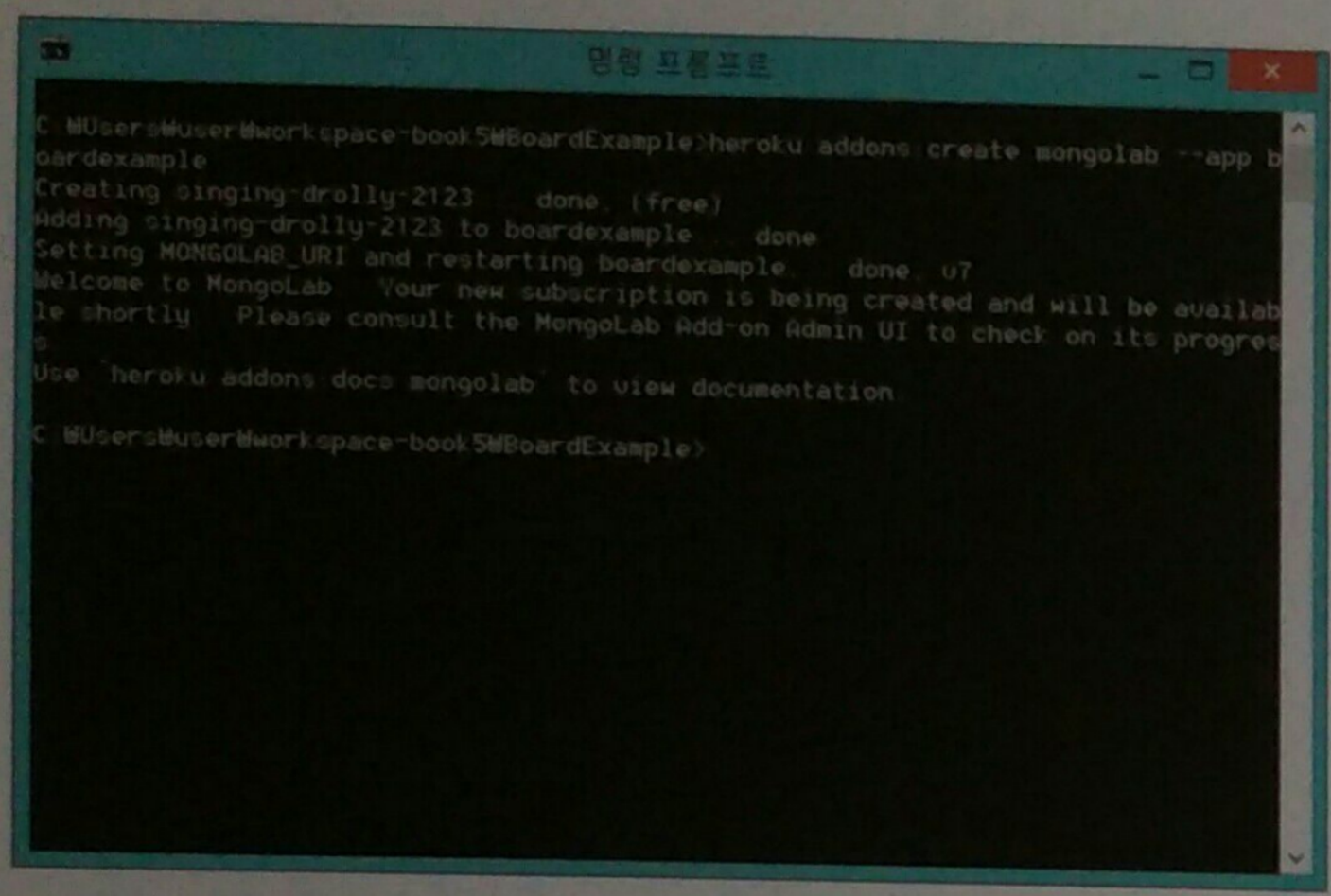
▶ https://dashboard.heroku.com/account/billing



▲ 신용카드 입력 화면

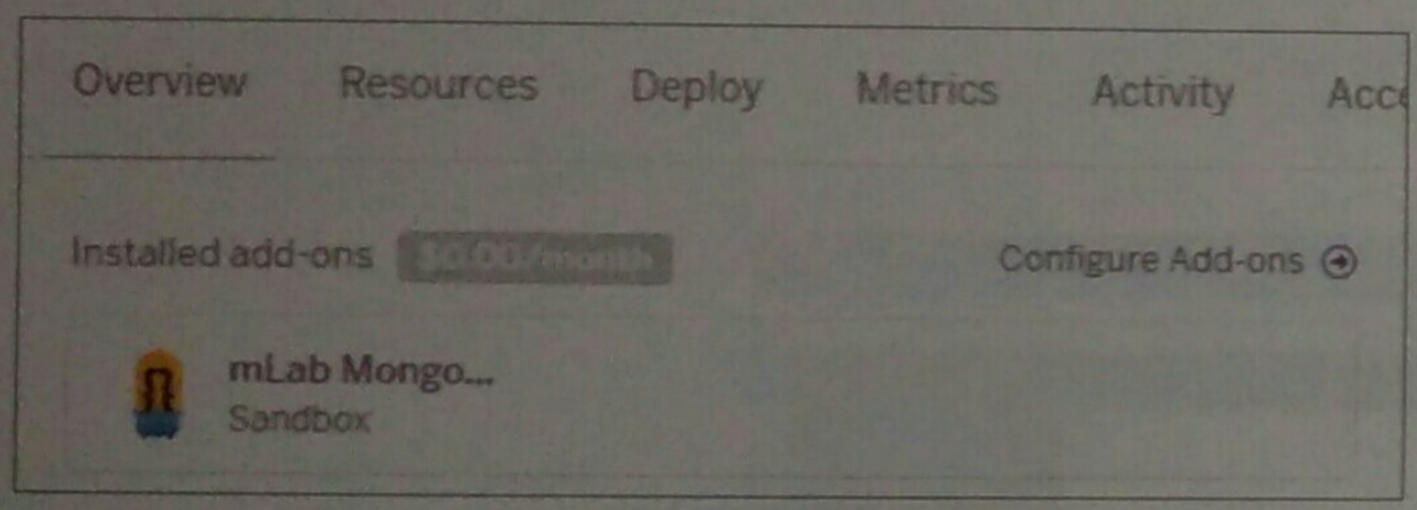
신용카드 정보를 입력했다면 이제 애드온을 설치할 수 있습니다. mLab MongoDB 애드온은 기본 사용량까지는 무료이므로 일단 비용을 걱정할 필요는 없습니다. 다음 명령을 입력하여 애드온을 설치합니다.

```
% heroku addons:create mongolab:sandbox --app boardexample
```



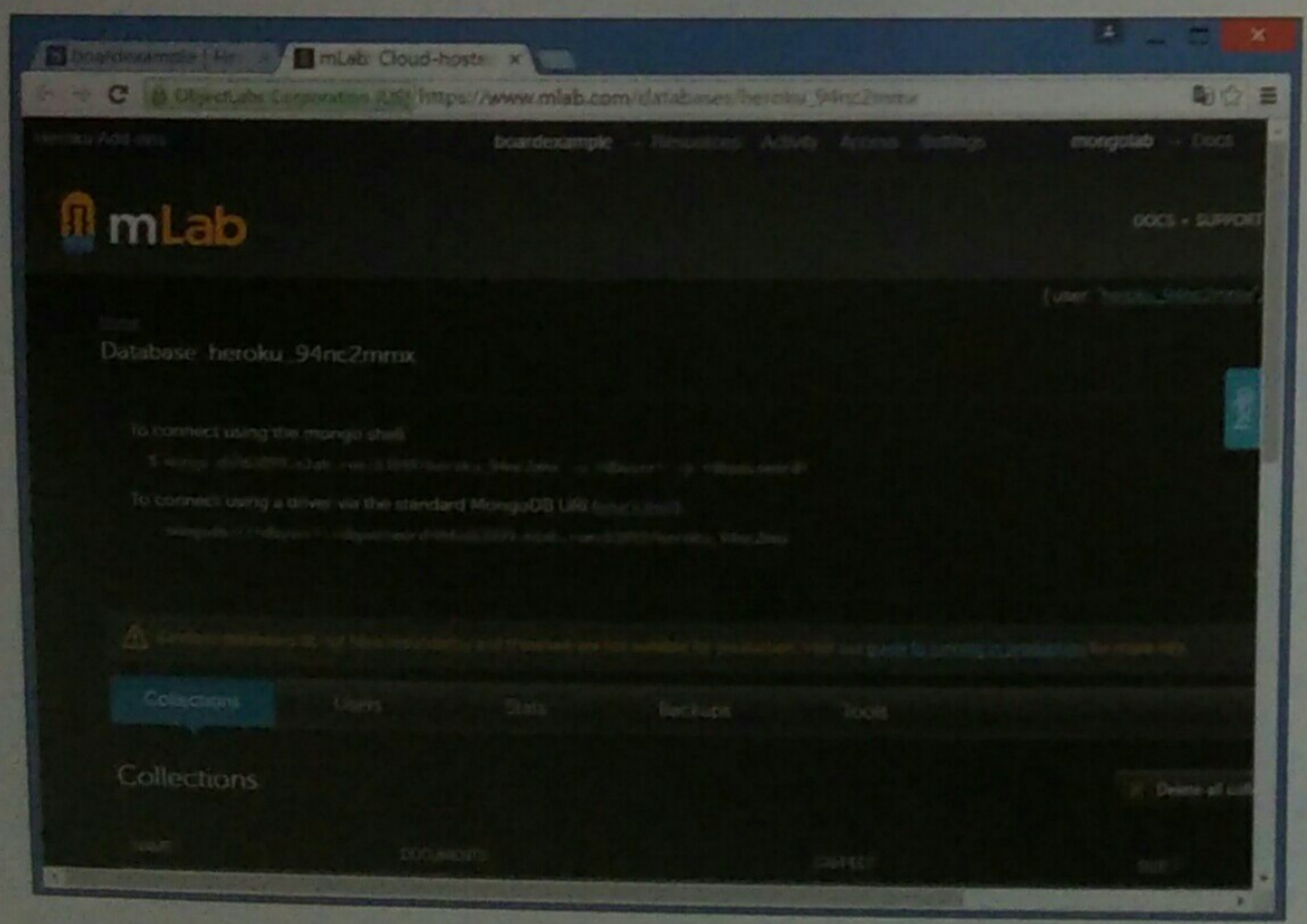
▲ mLab MongoDB 애드온 설치 화면

mLab MongoDB 애드온을 성공적으로 설치했으면 헤로쿠 사이트에서 앱 정보를 확인합니다. 그러면 다음 화면처럼 상세 정보가 보이면서 Installed add-ons 항목에 mLab MongoDB 애드온이 추가된 것을 확인할 수 있습니다.



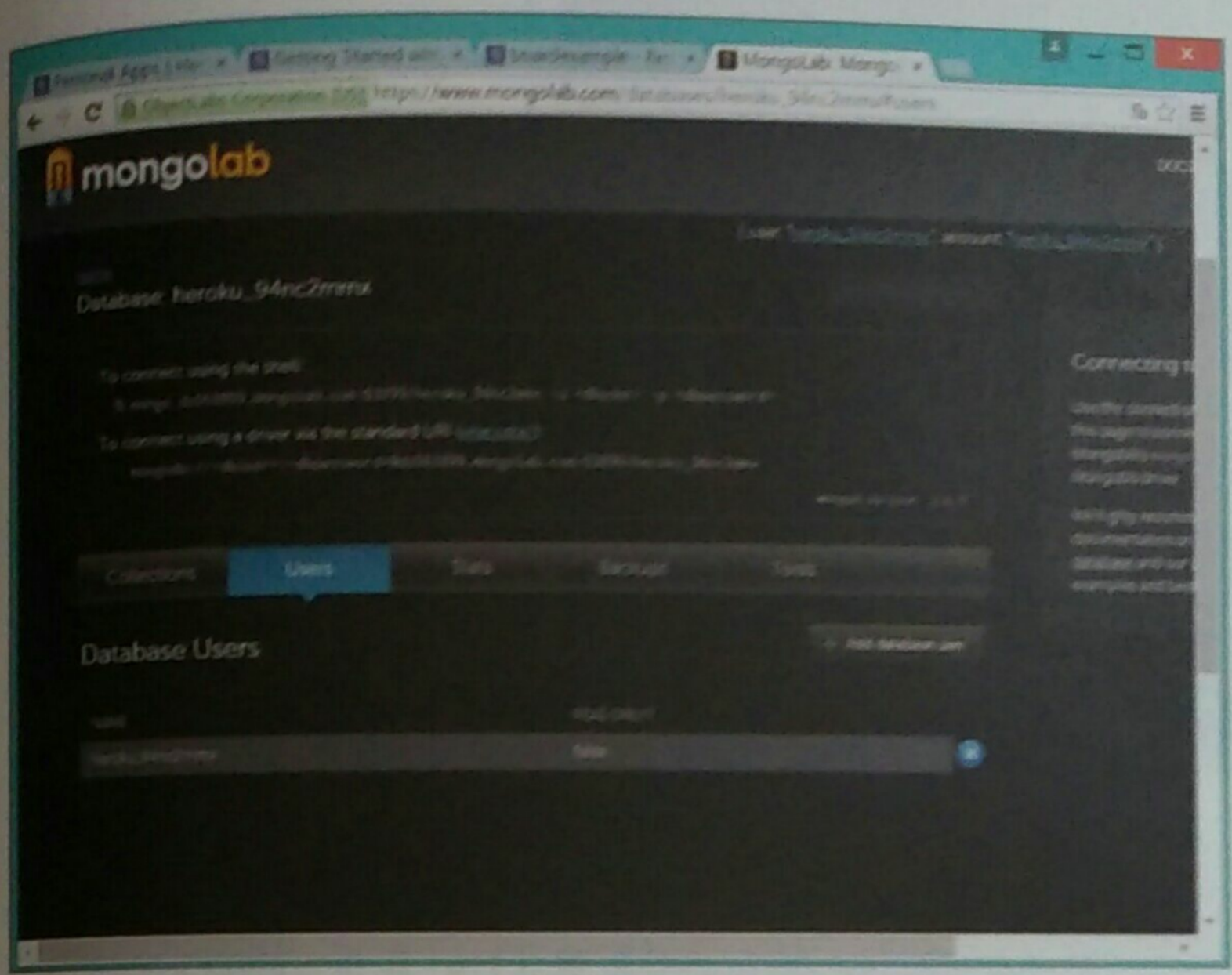
▲ 앱에 추가된 mLab MongoDB 애드온

화면에서 mLab MongoDB 항목을 누르면 MongoDB의 상세 정보 화면이 보입니다.



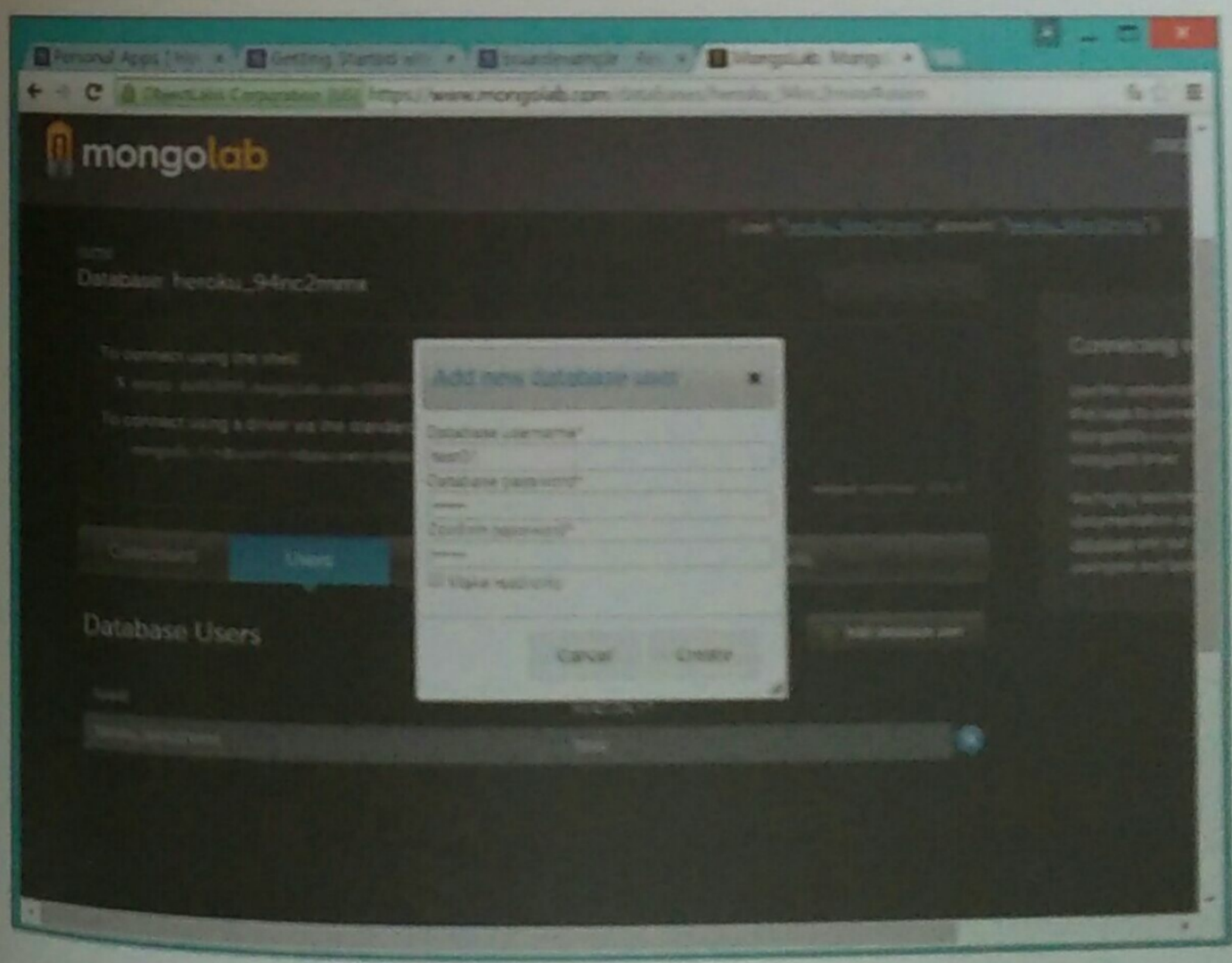
▲ MongoDB의 상세 정보 화면

이 화면에서는 사용자가 만든 컬렉션을 볼 수 있고 사용자 계정을 추가할 수도 있습니다. 아래쪽에 있는 [Users] 탭을 누르면 추가한 계정이 나타납니다.



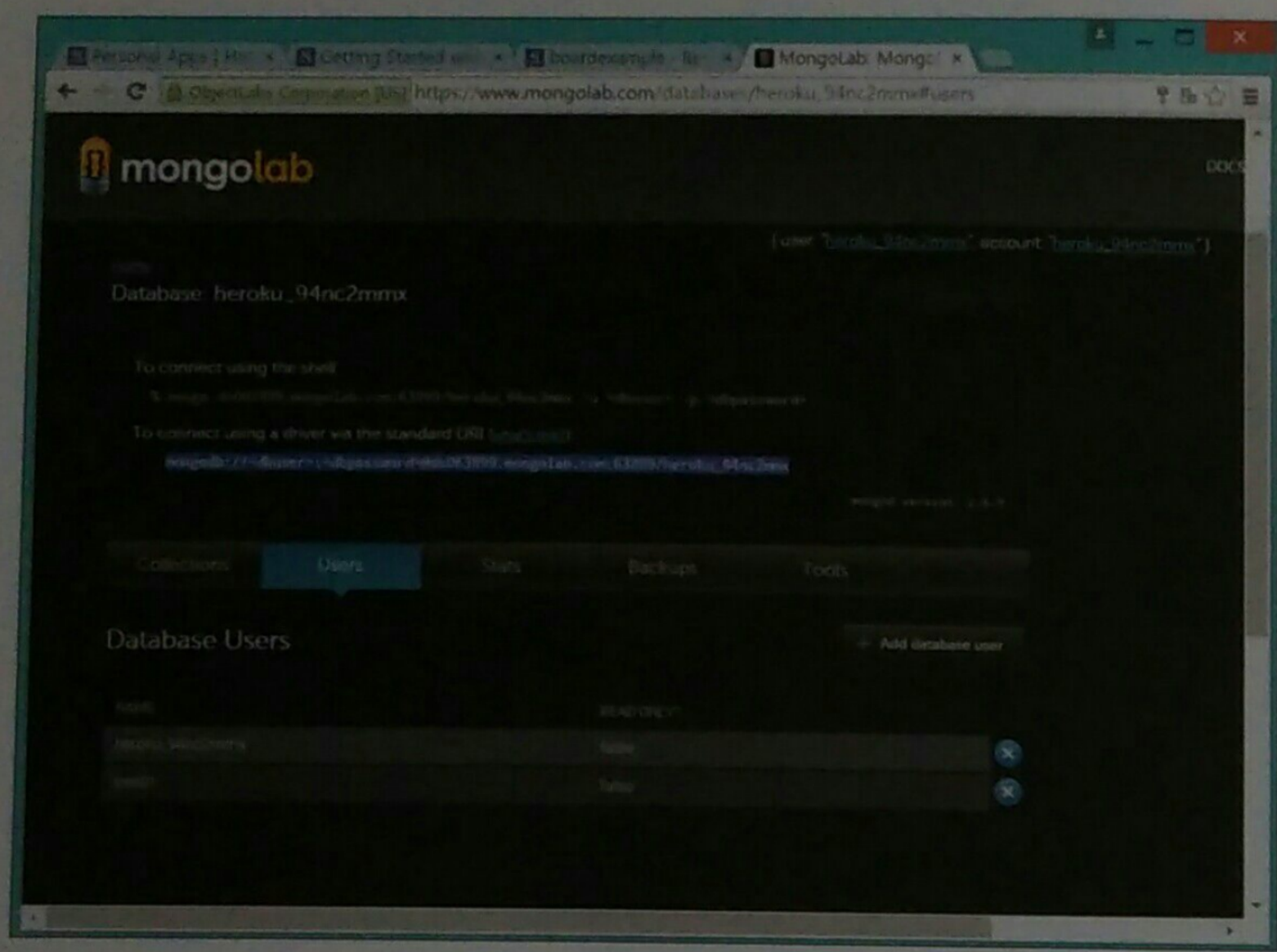
▲[Users] 탭을 선택했을 때의 화면

[Add database user] 버튼을 누르면 새로운 계정을 추가하는 대화상자가 나타납니다.



▲새로운 데이터베이스 계정을 추가하는 화면

대화상자에 필요한 정보를 입력한 후 [Create] 버튼을 누르면 계정이 만들어집니다. 계정이 정상적으로 만들어졌다면 위쪽의 mongodb 접근 URL을 복사하여 PC의 BoardExample 프로젝트에 들어 있는



▲사용자 계정을 추가한 화면

BoardExample 프로젝트의 [config] 폴더에 있는 config.js 파일을 열고 URL을 변경합니다. 기존 파일은 [config.old] 폴더를 새로 만든 후 그곳에 넣어 둡니다. 이렇게 하면 나중에 다시 로컬 PC에서 실행되고 있는 몽고디비 쪽으로 테스트할 때 이 파일을 사용할 수 있습니다.

참조 파일 | BoardExample>/config/config.js

```
module.exports = {

......

  db_url : 'mongodb://test01:0000@ds063899.mongolab.com:63899/heroku_94nc2mmx',

......
```

새로 추가한 계정의 아이디와 비밀번호를 함께 넣어 줍니다. 마지막으로 소스 파일을 업로드하기 전에 app.js 파일에서 포트 번호를 설정하는 코드 부분을 수정합니다. 헤로쿠 클라우드는 포트를 임의로 설정하기 때문에 process.env.PORT **환경 변수를** 사용해 여러분이 지정한 포트를 설정하도록 코드를 수정해야 합니다.

주의! 여기에서는 비밀번호를 OOOO으로 표시했습니다. 여러분이 만든 비밀번호로 바꾸어 주면 됩니다.

BoardExample 프로젝트의 app.js 파일을 열고 다음 코드처럼 수정합니다.

참조 파일 | BoardExample>/app.js

```
……,

var server = http.createServer(app).listen(process.env.PORT || app.get('port'), function() {

……,
```

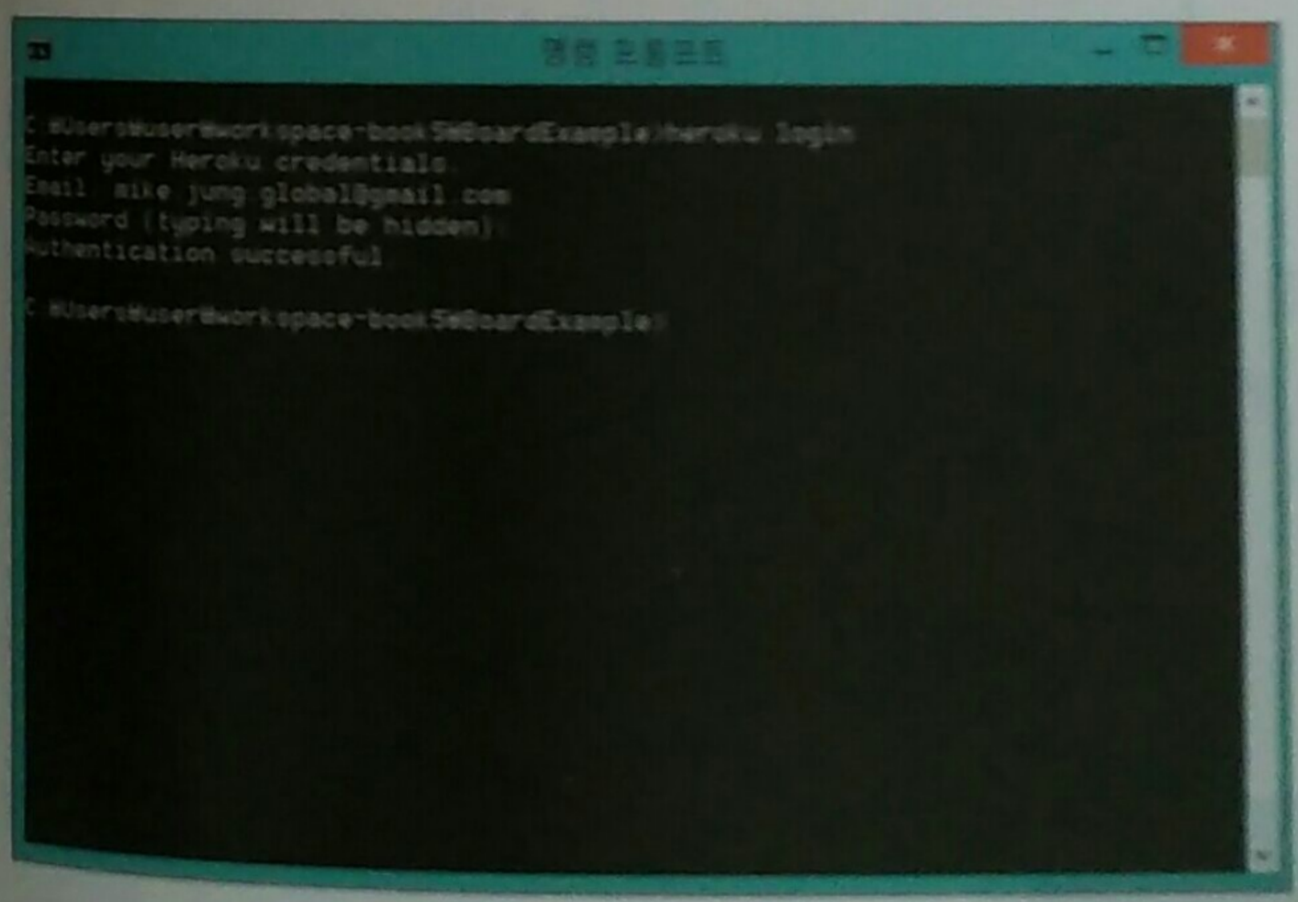
서버를 실행할 때 사용하는 기본 포트를 process.env.PORT로 설정했습니다. 이 값이 없다면 코드에서 설정한 포트를 사용합니다.

소스 업로드하기

이제 프로젝트의 소스 파일들을 업로드할 차례입니다. 명령 프롬프트에서 다음 명령을 입력하여 헤로쿠에 로그인합니다.

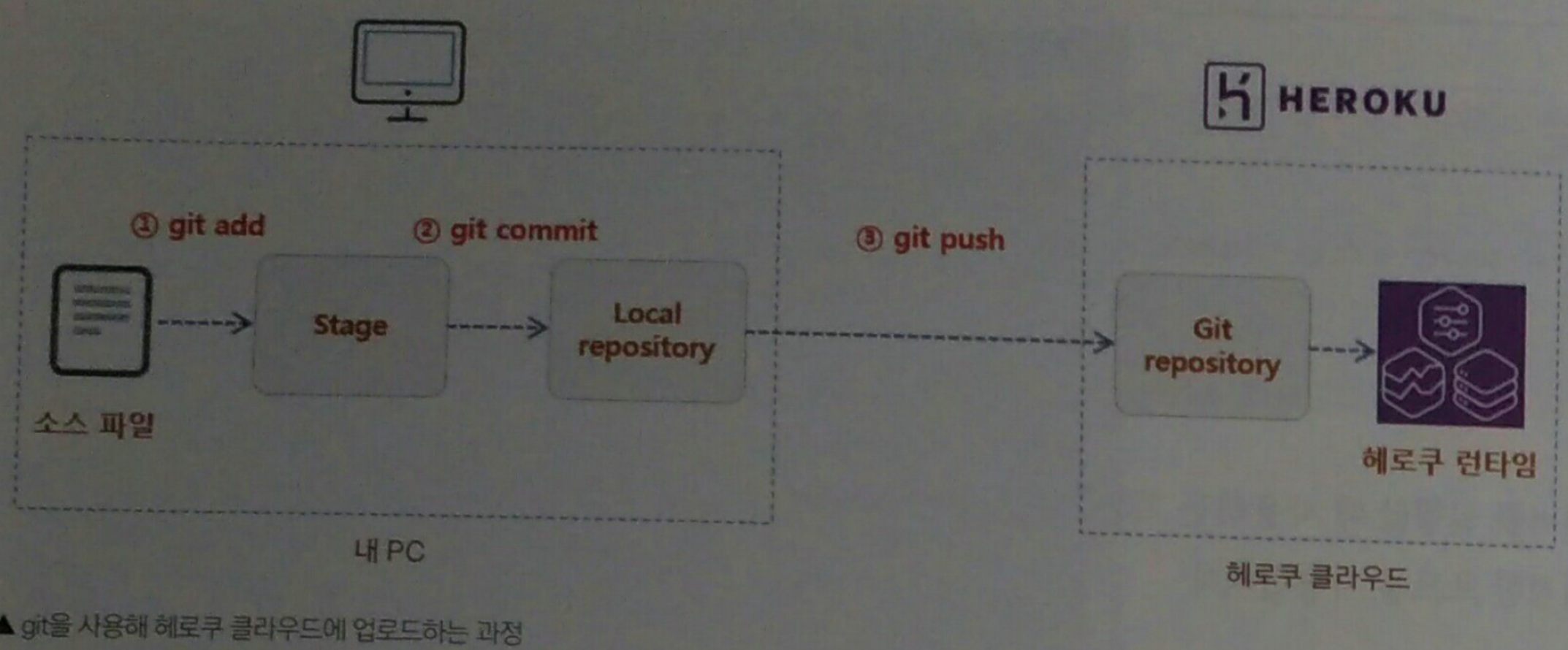
```
% heroku login
```

로그인을 하면 이메일 계정을 물어봅니다. 이메일 계정을 입력하고 비밀번호까지 입력하면 정상적으로 로그인됩니다.



▲헤로쿠에 로그인한 결과 화면

로그인 상태에서 git을 사용해 헤로쿠 클라우드에 소스를 업로드해야 합니다. 업로드 과정은 다음과 같습니다.



▲ git을 사용해 헤로쿠 클라우드에 업로드하는 과정

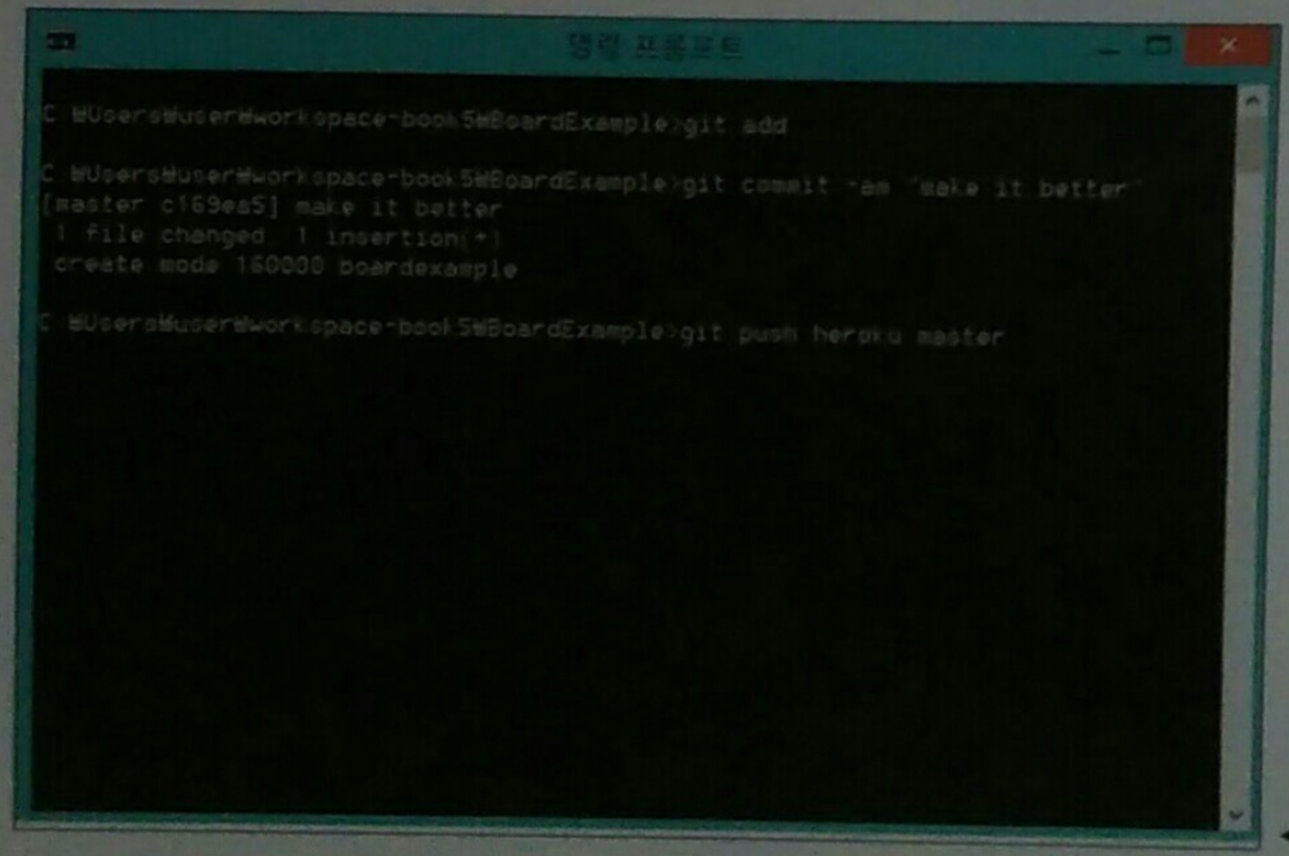
내 PC에서 작성한 소스 코드는 내 PC 안에 만든 **git 로컬 저장소**(Local repository)에 저장합니다. git 로컬 저장소에 저장하려면 먼저 **git add 명령**을 사용해 **스테이징 단계**로 넘겨야 합니다. 스테이징 단계에서는 로컬 저장소에 저장하기 전에 어떤 파일들을 저장할지 체크합니다. 그다음 **git commit 명령**을 사용해 git 로컬 저장소에 저장합니다. 로컬 저장소에 저장된 소스 코드는 헤로쿠 클라우드로 업로드할 수 있으며, 이 과정을 푸시(push)라고 부릅니다. **git push 명령**을 사용해 푸시하면 헤로쿠 클라우드의 git 저장소에 소스 코드가 저장됩니다.

그러면 먼저 BoardExample 프로젝트 폴더에서 git의 add 명령을 실행합니다.

```
% git add .
```

이렇게 하면 현재 [프로젝트] 폴더가 스테이징 단계로 넘어갑니다. 그다음 변경된 파일들을 git commit 명령으로 로컬 저장소에 저장한 후 git push 명령으로 푸시합니다.

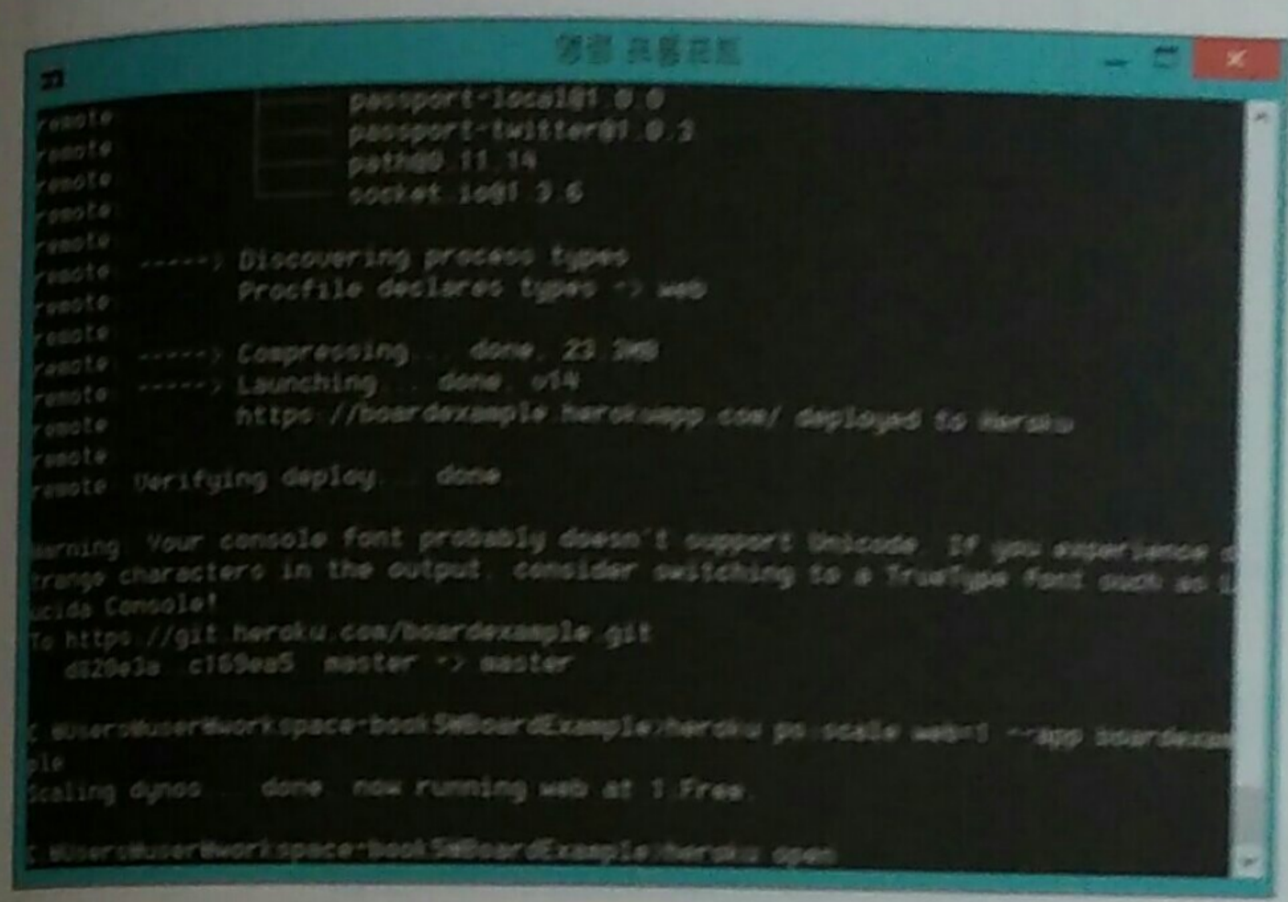
```
% git commit -am "make it better"
% git push heroku master
```



◀ 커밋 후 파일을 푸시하는 화면

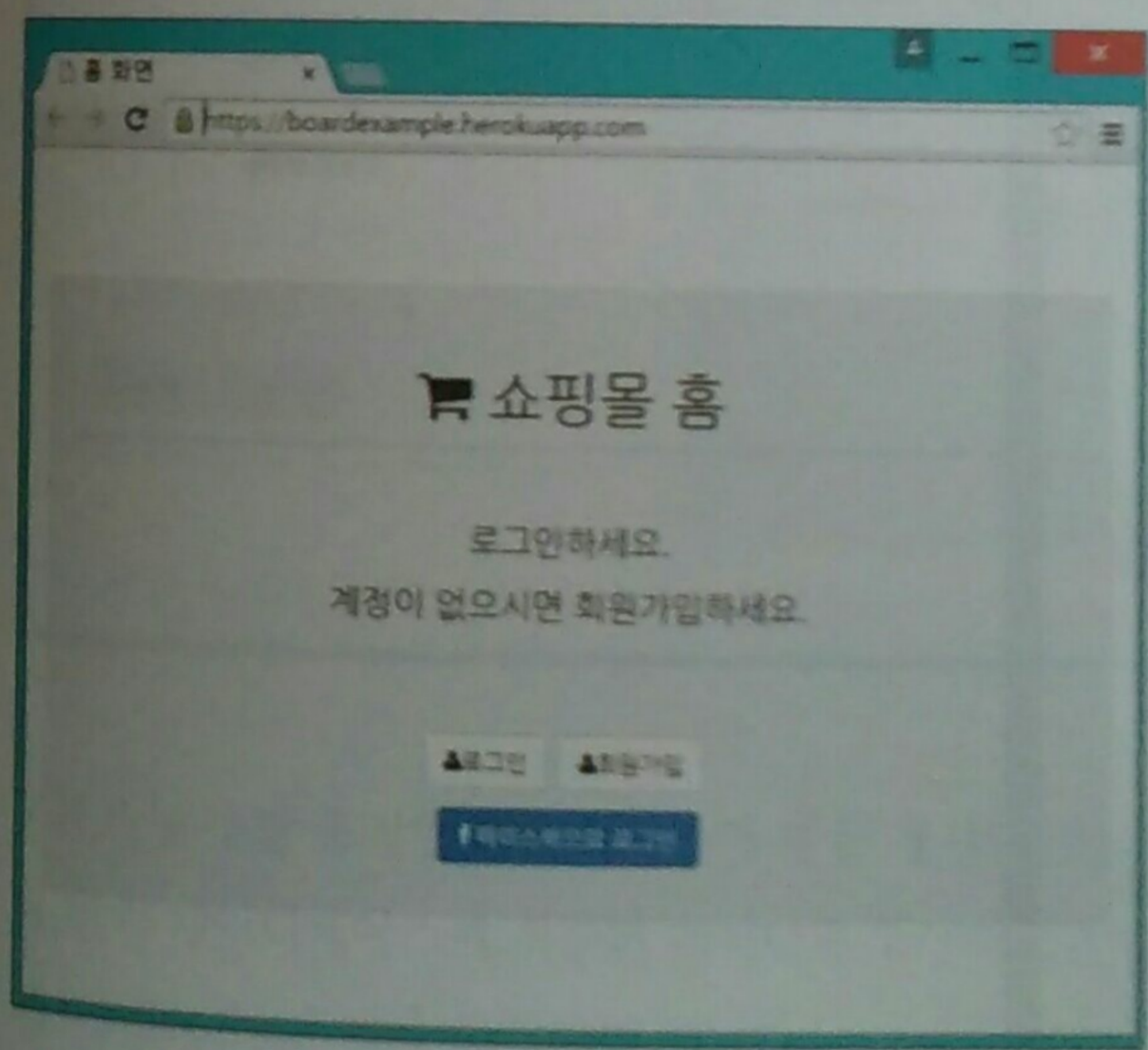
git push 명령을 실행하면 파일이 업로드되면서 여러 가지 메시지가 나타납니다. 업로드가 성공적으로 끝났으면 서버를 실행합니다.

```
% heroku ps:scale web=1 --app boardexample
% heroku open
```



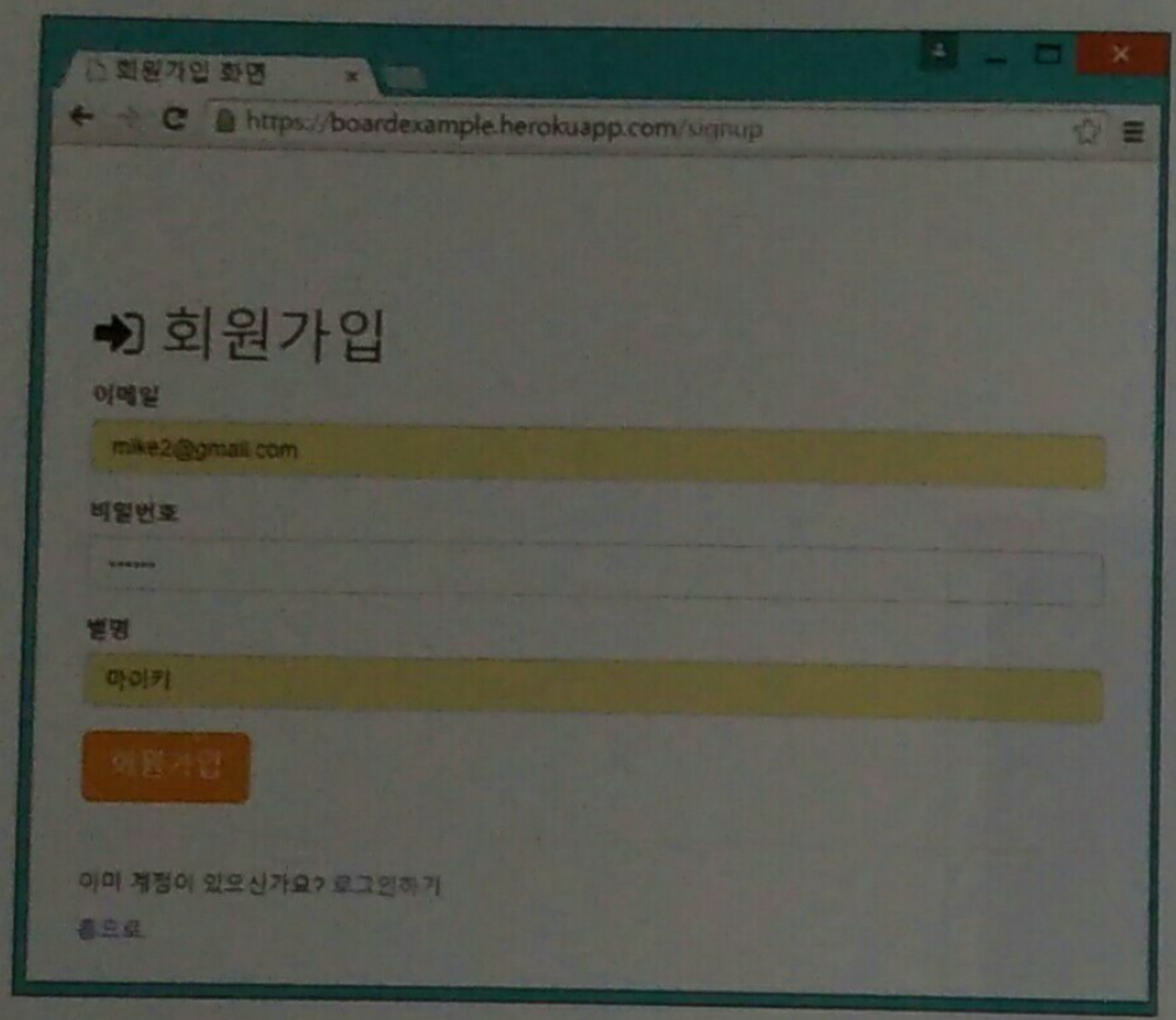
◀ 서버를 실행하는 화면

이제 웹 브라우저가 화면에 나타나고 여러분이 올린 클라우드의 홈 화면이 표시됩니다.



▲ 클라우드에서 보이는 홈 화면

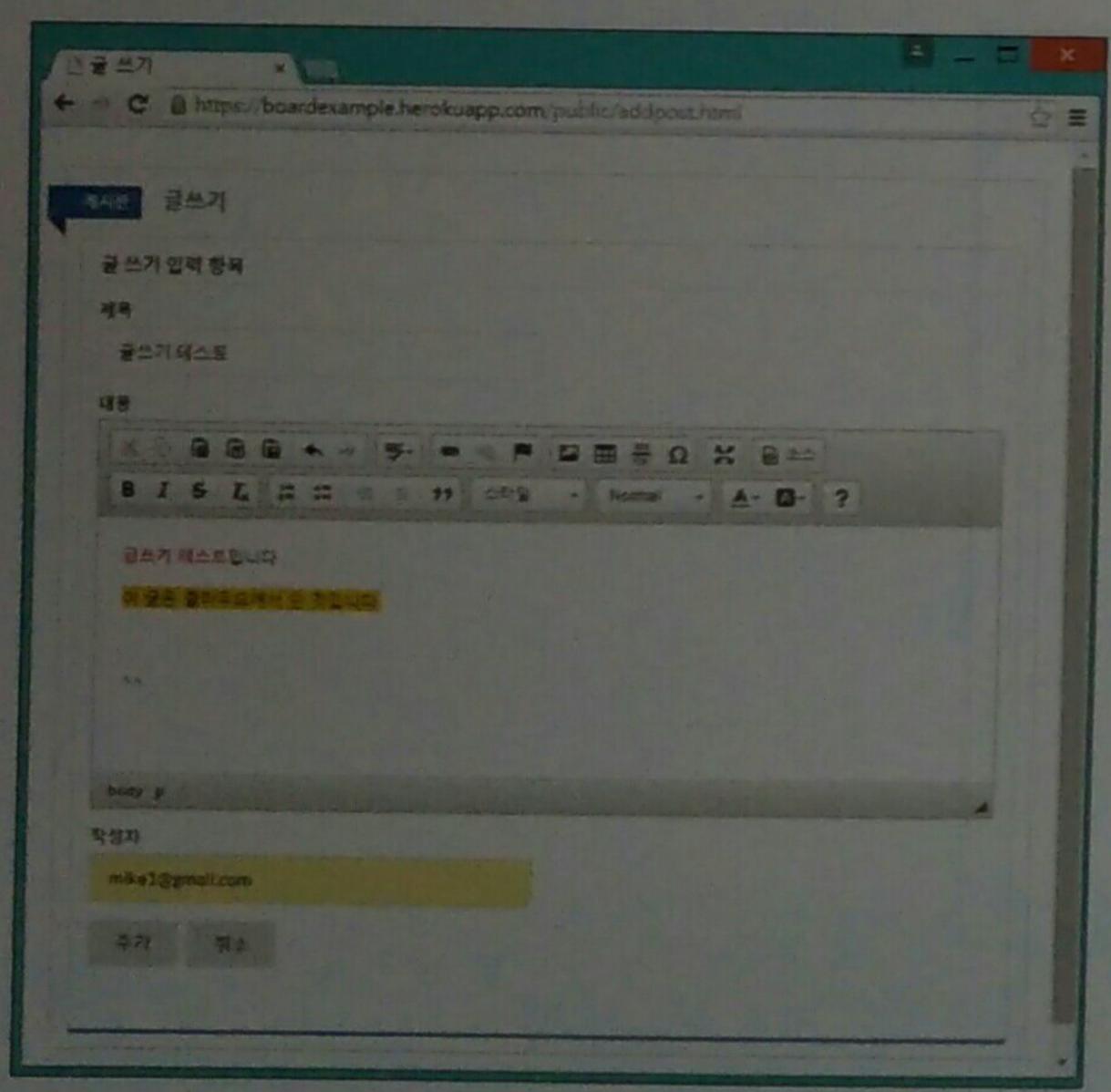
드디어 여러분이 만든 서버를 사용해 홈 화면을 보여 줄 수 있게 되었습니다. 이 홈 화면은 전 세계 어디에서나 접속할 수 있습니다. 여러분이 직접 만든 기능을 전 세계 사람들에게 서비스할 수 있게 되었다니 감격스럽지 않나요? 기능이 정상적으로 동작하는지 확인하기 위해 먼저 [회원가입] 버튼을 눌러 사용자를 등록합니다.



▲ 클라우드에서 보이는 회원가입 화면

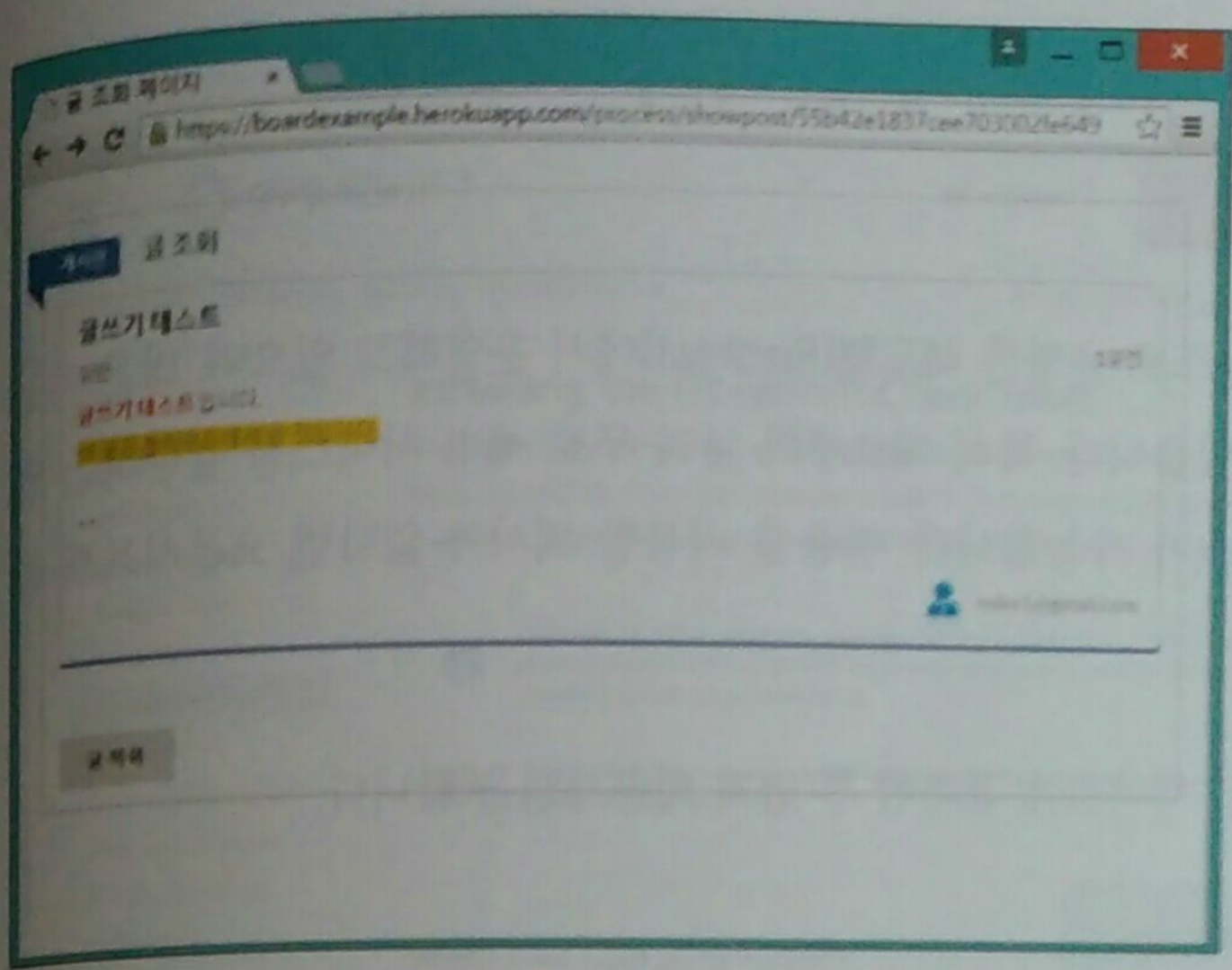
회원가입을 했으면 다음 주소를 입력하여 글쓰기 화면으로 이동합니다.

▶ https://boardexample.herokuapp.com/public/addpost.html



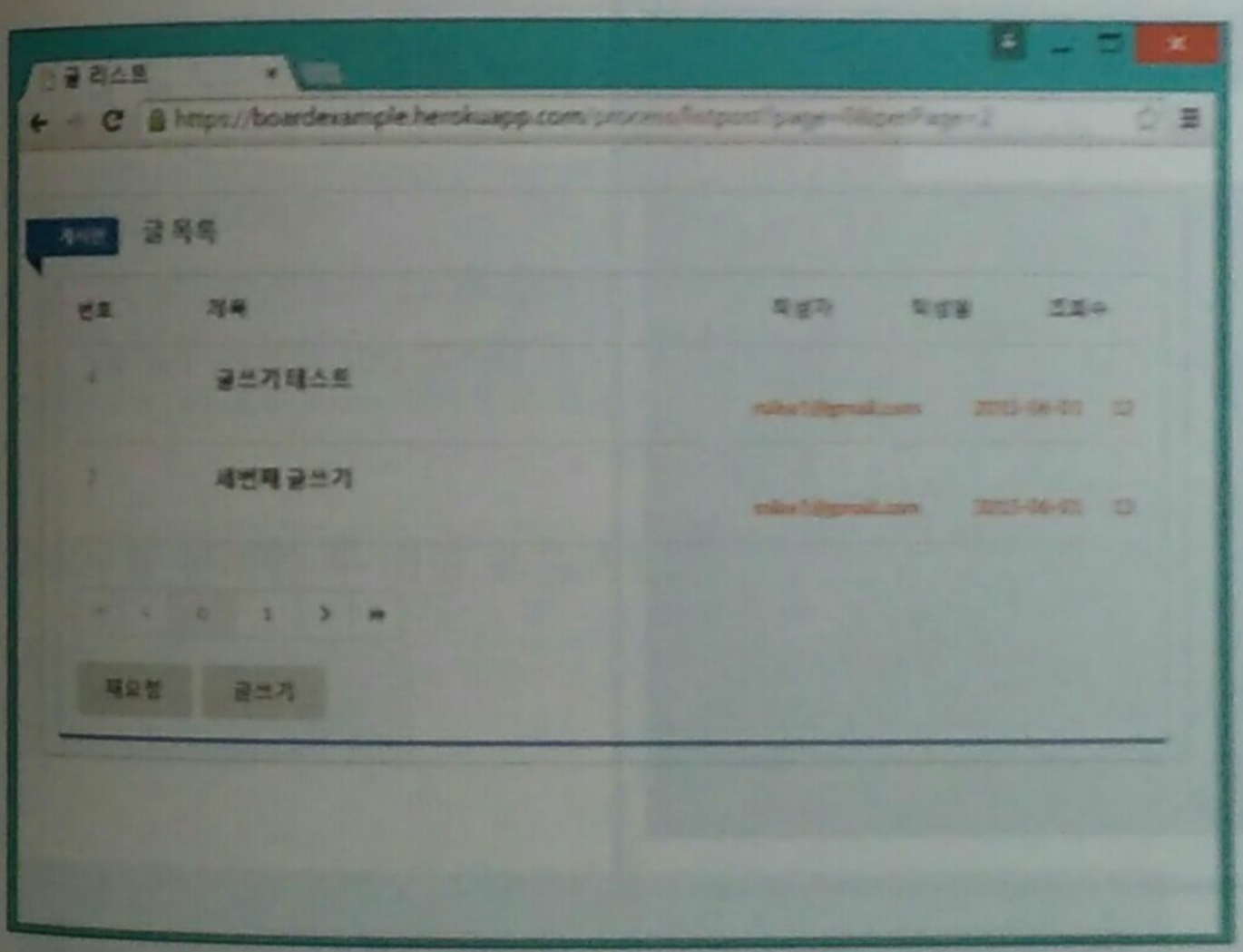
▲ 클라우드에서 보이는 글쓰기 화면

제목, 내용, 작성자를 입력한 후 [추가] 버튼을 누르면 글이 정상적으로 저장된 후 글의 내용이 조회됩니다.



◀ 클라우드에서 보이는 글 조회 화면

글 조회 화면에서 [글 목록] 버튼을 누르면 글 목록 화면으로 전환됩니다.



◀ 클라우드에서 보이는 글 목록 화면

모든 기능이 잘 동작한다면 여러분이 만든 게시판 기능은 정상적으로 헤로쿠 클라우드에 올라간 것입니다. 만약 데이터베이스에 저장된 데이터를 확인하고 싶다면 헤로쿠 사이트에서 mLab MongoDB 페이지의 [Collection] 탭을 확인하면 됩니다.

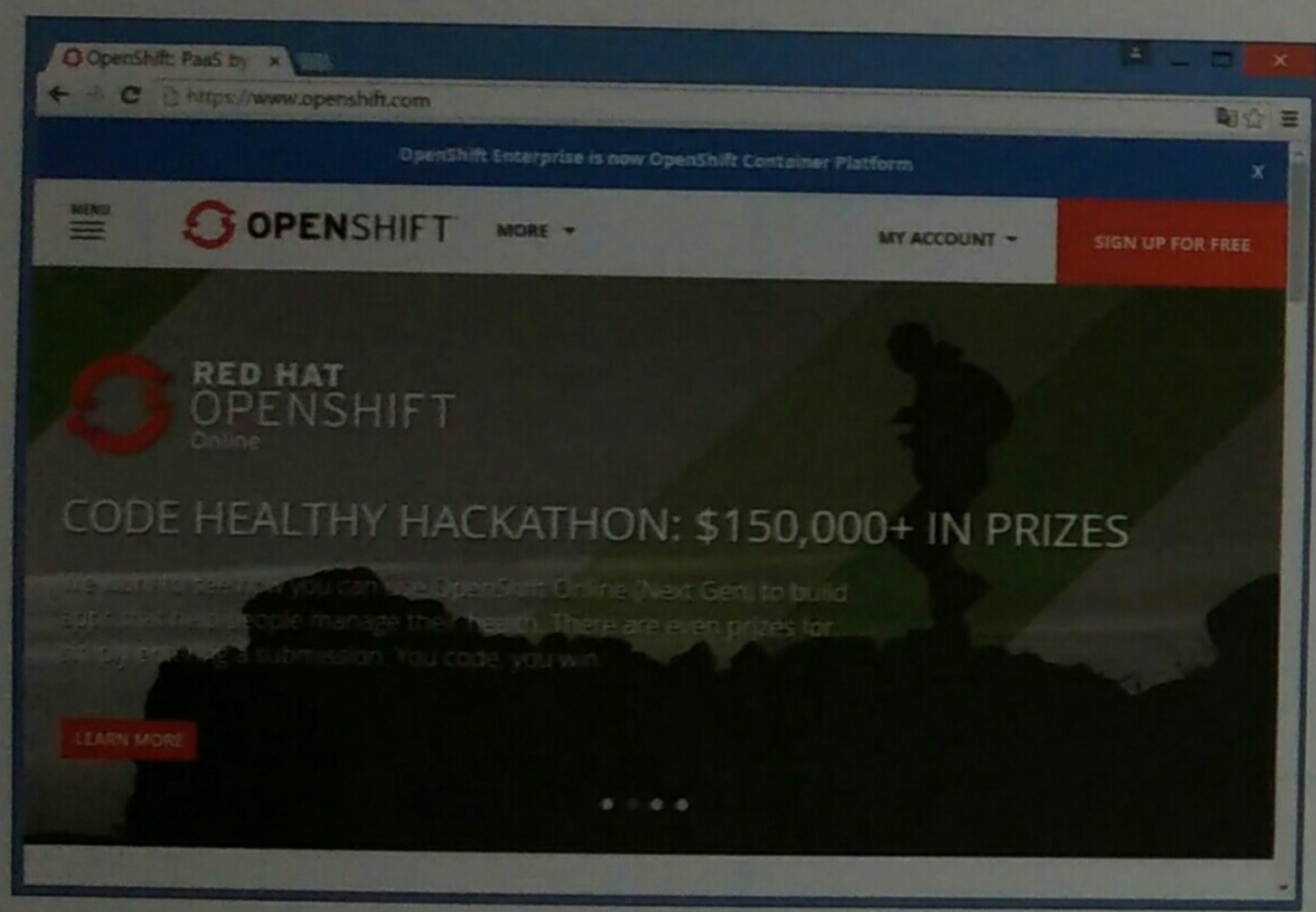
지금까지 여러분이 만든 노드 서버를 헤로쿠 클라우드에 올리는 방법에 대해 알아보았습니다. 헤로쿠는 많은 웹 개발자 또는 앱 개발자가 사용하는 사이트이지만 무료 계정을 사용한다면 하루 24시간 중 일부 시간 동안 서비스가 중단됩니다. 게다가 많은 애드온 패키지들이 유료로 제공되기 때문에 안정적인 서비스를 위해서는 유료로 전환해야 합니다. 그럼에도 불구하고 헤로쿠 클라우드는 사용이 간편하고 보유하고 있는 다양한 애드온 패키지들이 매력적이라서 약간의 비용을 지불할 의사가 있다면 헤로쿠 클라우드를 사용하길 권합니다.

15-3 | 오픈시프트 클라우드에 올리기

오픈시프트(OpenShift) 클라우드는 리눅스로 유명한 **레드햇(RedHat)**에서 운영하고 있으며, 많은 기능을 개방적으로 제공하는 정책을 취하고 있습니다. 특히 헤로쿠와 달리 무료 계정 서비스를 중단하는 일이 없어 무료 계정임에도 안정적인 서비스가 가능합니다. 비용을 지불할 의사가 없다면 오픈시프트 클라우드 사용을 권합니다.

오픈시프트 클라우드를 사용하려면 다음 사이트에 접속한 후 먼저 회원가입을 합니다.

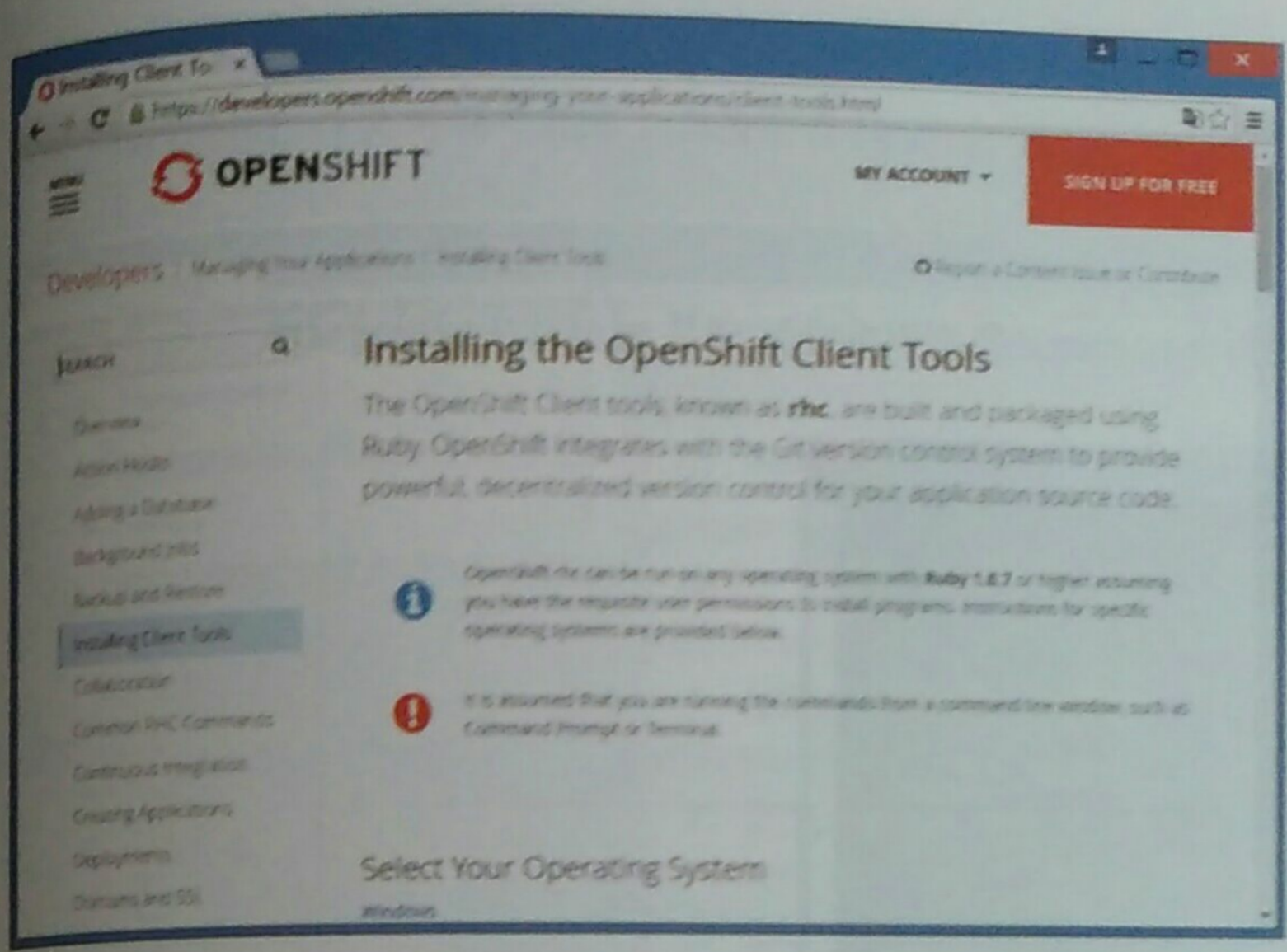
오픈시프트 사이트 ▶ http://www.openshift.com



▲ 오픈시프트 사이트

회원가입을 한 후 오픈시프트 클라우드 접속 도구인 rhc를 다운로드합니다. 다음 사이트를 참고하여 rhc를 설치합니다.

rhc 설치 안내 사이트 ▶ https://developers.openshift.com/en/managing-client-tools.html



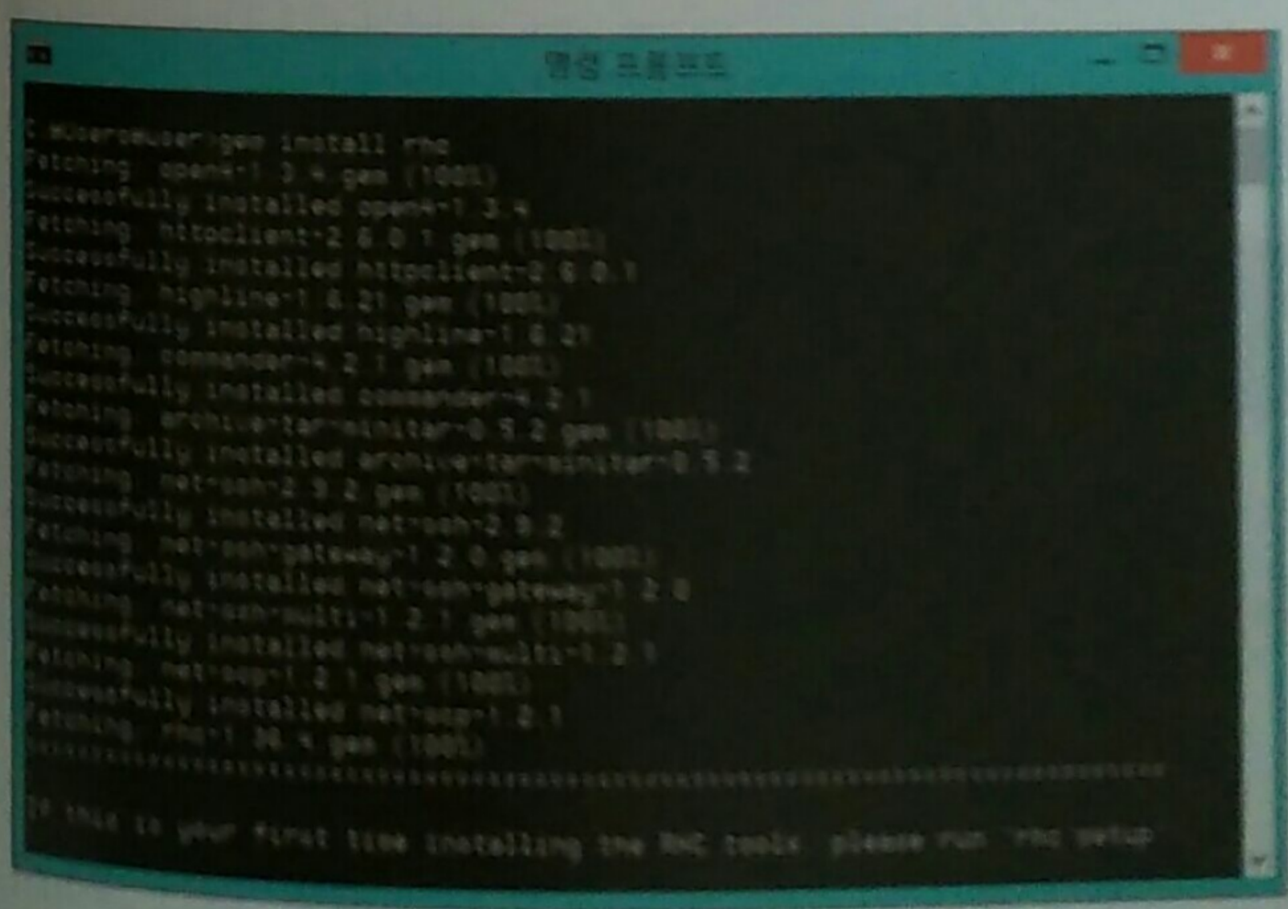
▲rhc 설치 안내 사이트

rhc를 설치하려면 다음 세 가지 단계를 거쳐야하니 잘 보고 설치를 진행합니다.

1. Ruby 설치하기
2. Git 설치하기(이미 Git을 설치한 경우는 제외)
3. Gem을 사용해 rhc 설치하기

Ruby를 설치한 후 명령 프롬프트를 열고 다음 명령을 입력하면 rhc가 설치됩니다.

```
% gem install rhc
```

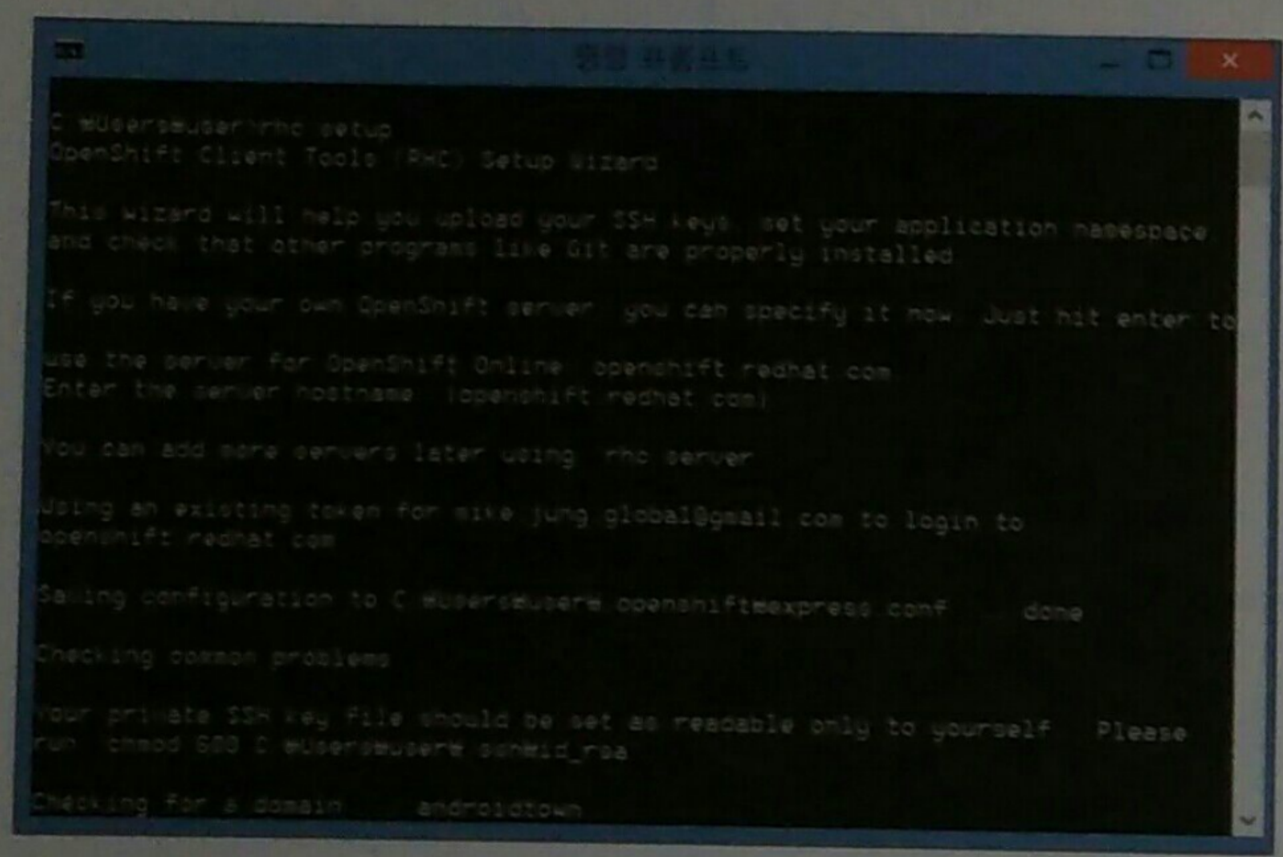


▲gem 명령으로 rhc를 설치하는 화면

rhc를 사용하기 위해 다음 명령을 실행하여 net-ssh를 설치한 후 rhc setup을 진행합니다.

```
% gem install net-ssh
% rhc setup
```

rhc setup 명령을 실행하면 서버의 hostname을 입력하라는 영문 메시지가 나타납니다.

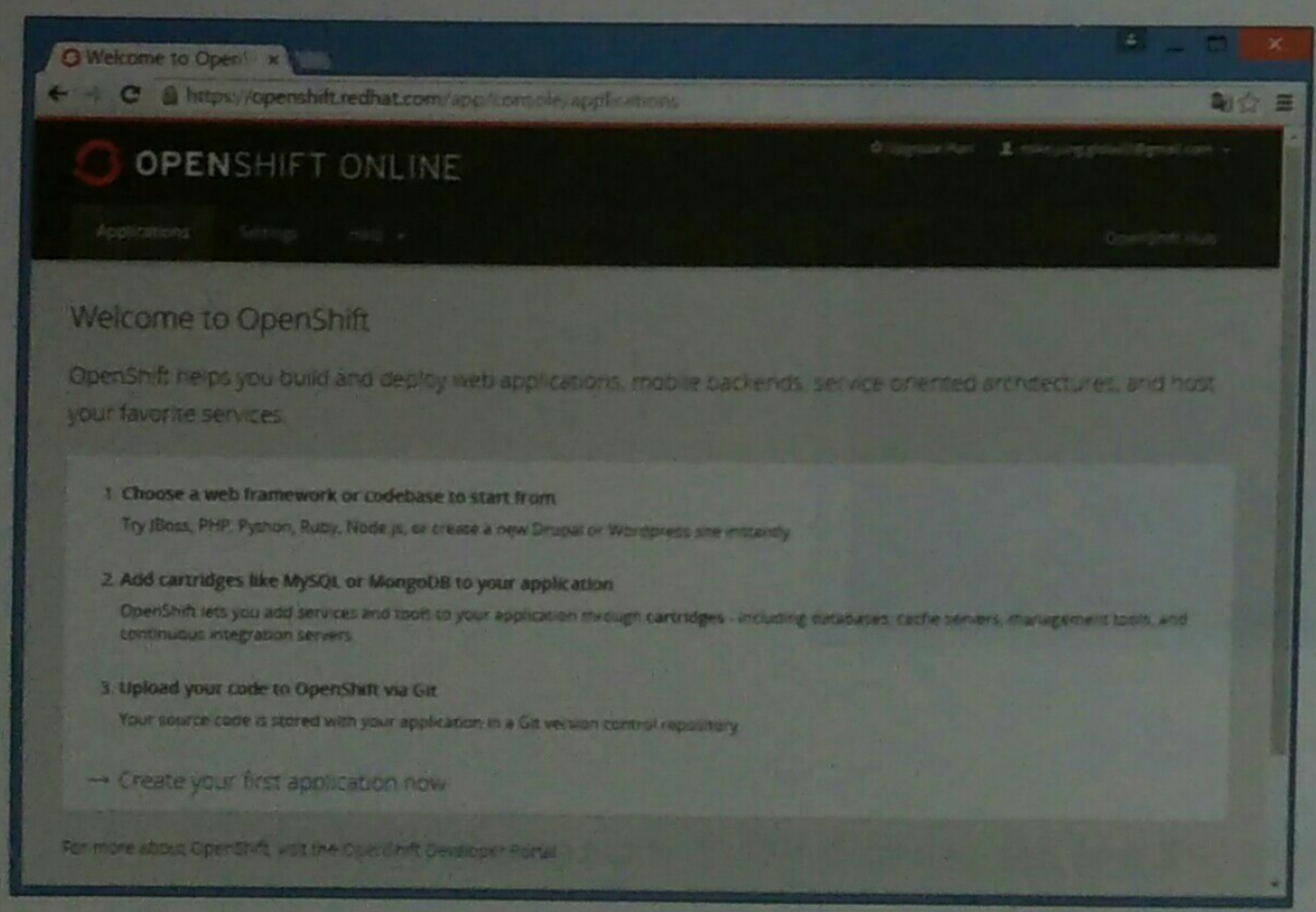


▲ rhc setup 명령을 입력했을 때 명령 프롬프트 창에 출력되는 내용

아무것도 입력하지 않고 Enter 키를 누르면 로그인 정보를 요구합니다. 오픈시프트 사이트에서 회원 가입을 할 때 사용한 이메일 주소와 비밀번호를 차례대로 입력하면 설정이 끝납니다.

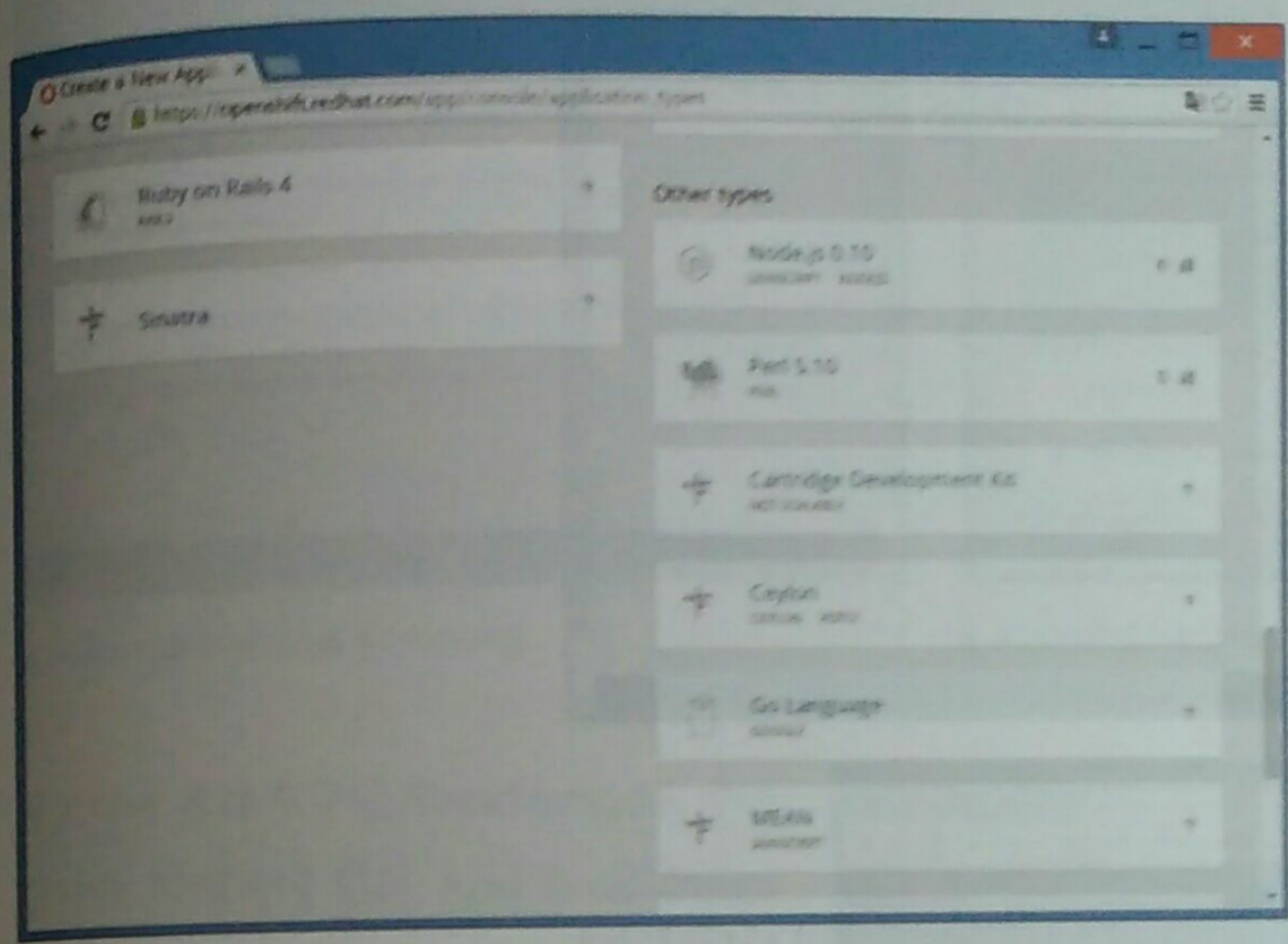
이제 rhc를 사용할 준비가 되었습니다. 먼저 오픈시프트 온라인 사이트에서 새로운 앱 영역을 만들어 보겠습니다. 로그인 후에 다음 사이트에 접속합니다.

▶ https://openshift.redhat.com/app/console/applications



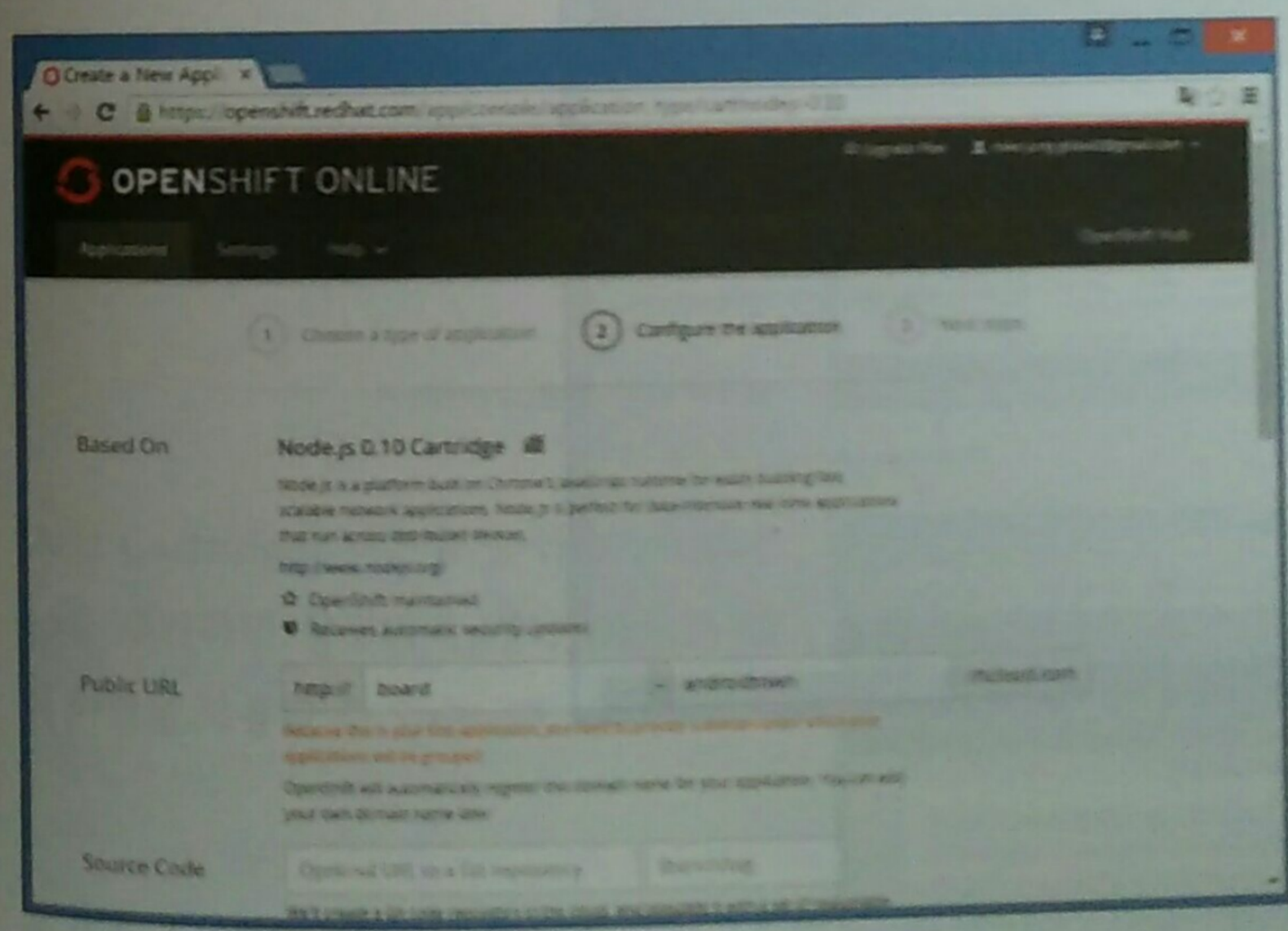
◀ 오픈시프트 콘솔 페이지

아래쪽에 있는 [Create your first application now] 링크를 누르면 애플리케이션 유형을 선택하는 화면이 나타납니다. 가장 아래쪽으로 스크롤하면 Node.js 항목이 보입니다.



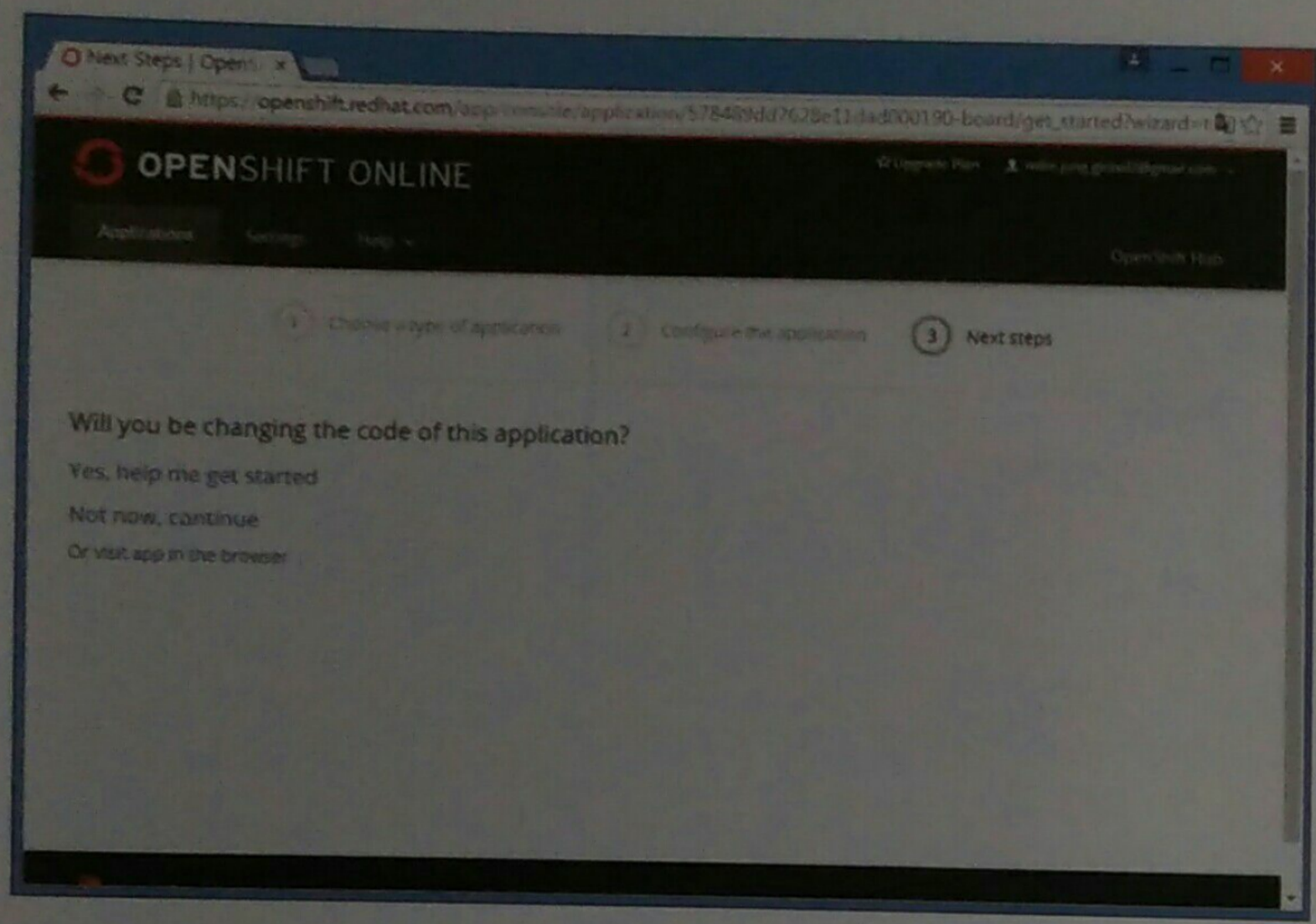
▲애플리케이션 유형을 선택하는 화면

Node.js 항목을 누르면 애플리케이션을 설정하는 화면으로 넘어갑니다.



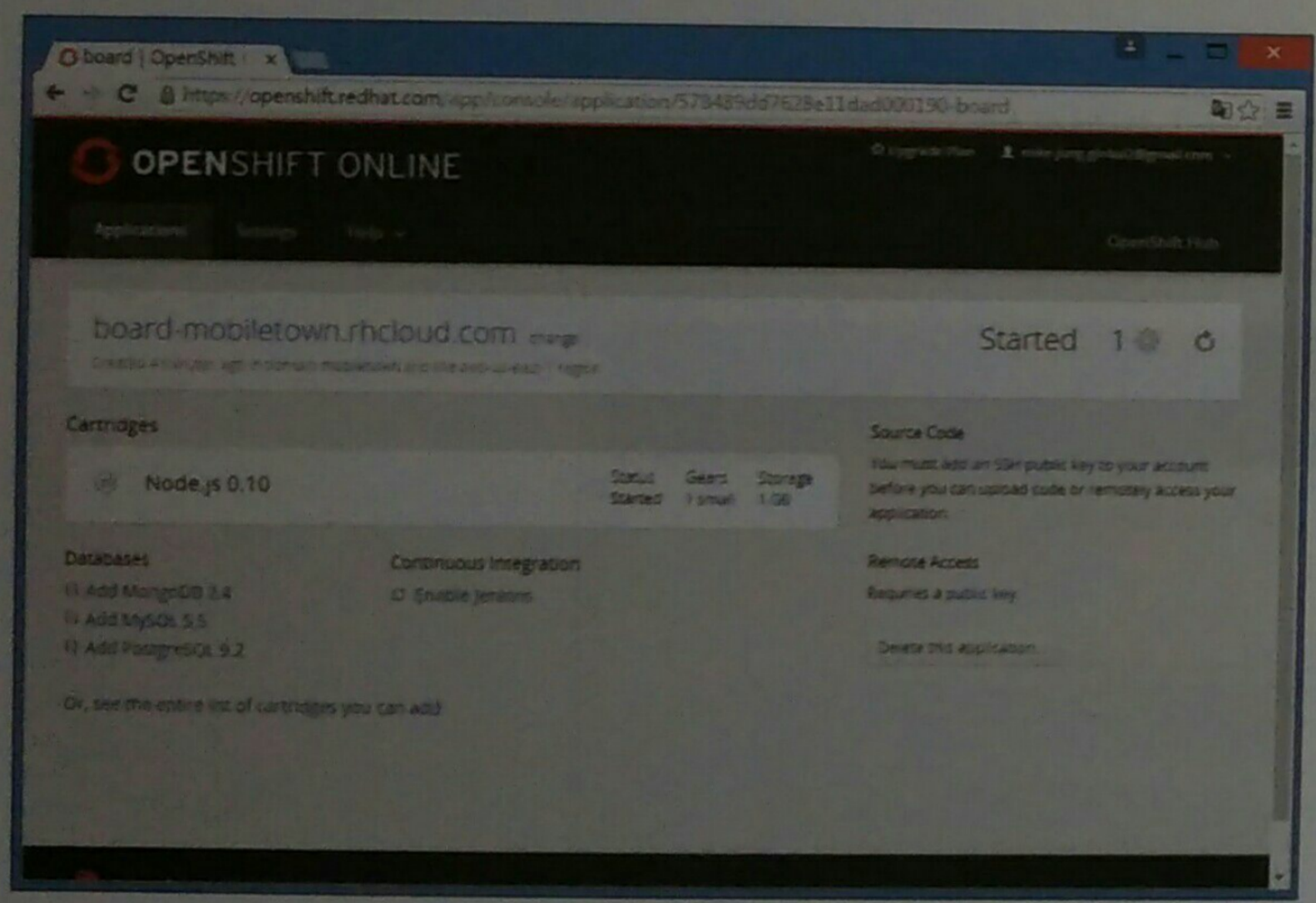
▲애플리케이션을 설정하는 화면

Public URL 항목에는 웹 사이트 접속을 위해 사용할 주소를 입력합니다. 여기에서는 board-android town을 두 개의 입력 상자에 나누어 입력했습니다. 여러분은 이와 다른 이름(board-OOOOOOO와 같은 형식으로 된 이름)을 입력해야 합니다. 아래쪽에 있는 [Create Application] 버튼을 누르고 조금 기다리면 애플리케이션이 만들어집니다.



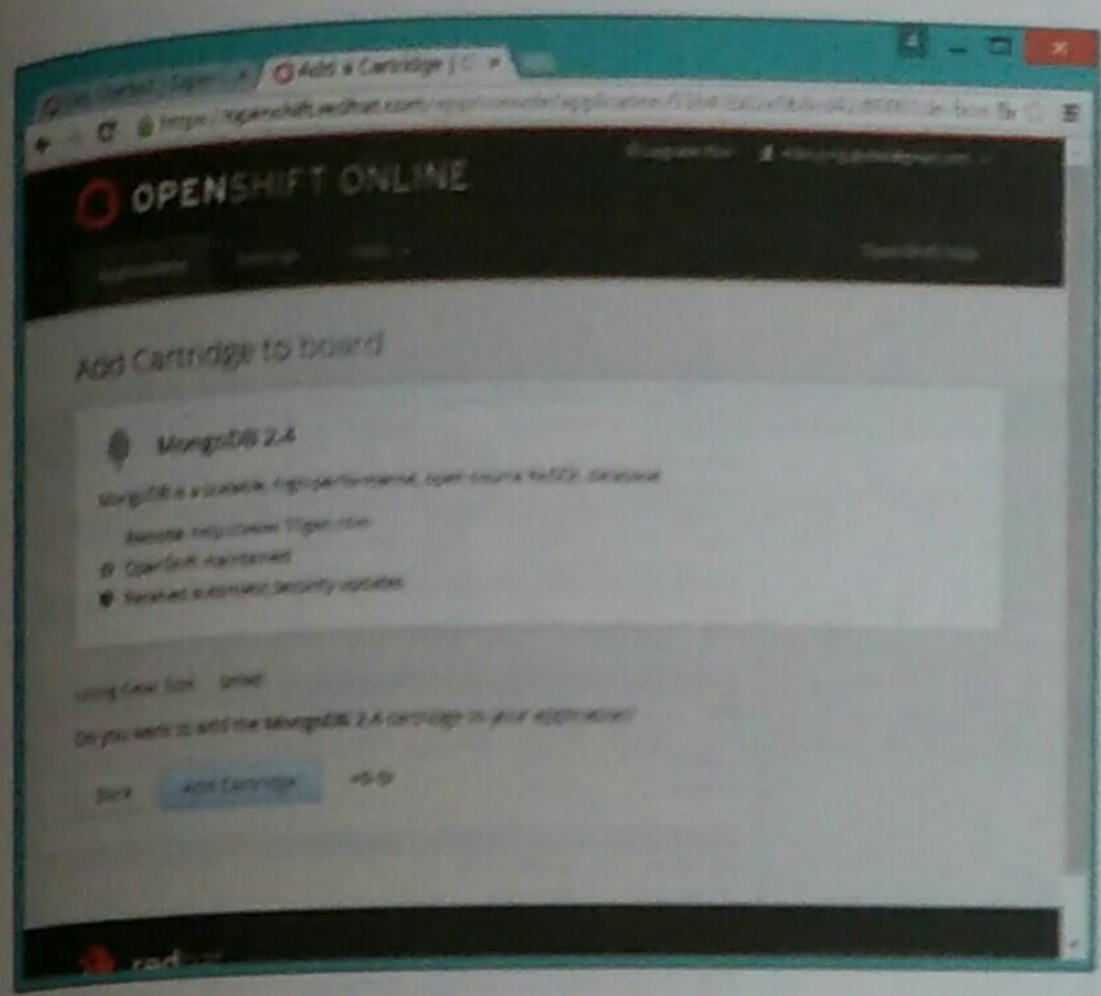
▲ 애플리케이션이 만들어진 후의 결과 화면

'Not now, continue' 링크를 클릭하면 새로 만들어진 앱이 나타납니다.

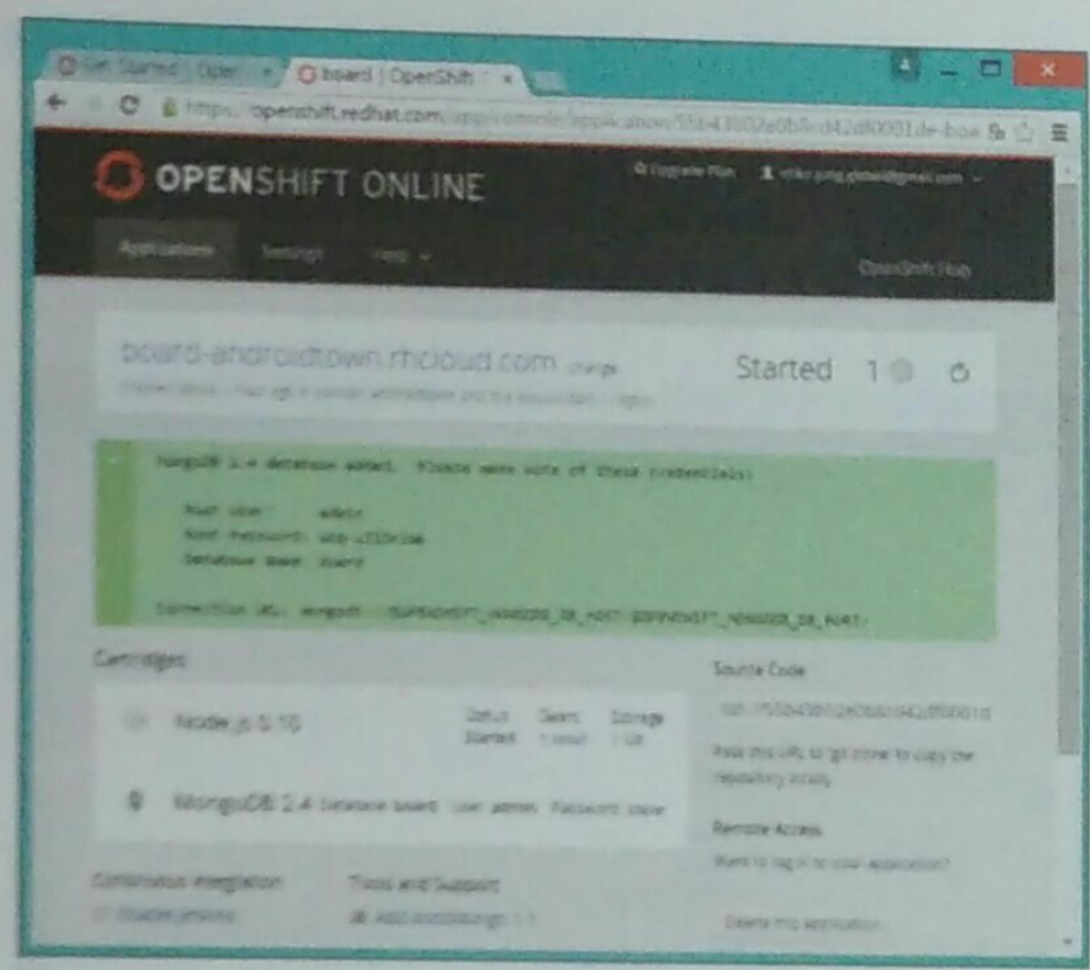


▲ 오픈시프트 온라인 화면에 보이는 추가된 애플리케이션

이제 몽고디비 데이터베이스를 추가합니다. 'Add MongoDB 2.4' 링크를 클릭해서 데이터베이스 추가 화면으로 넘어갑니다. [Add Cartridge] 버튼을 클릭하여 데이터베이스를 추가합니다. 그러면 추가된 몽고디비 데이터베이스에 접속할 수 있는 URL과 계정 정보가 표시됩니다.

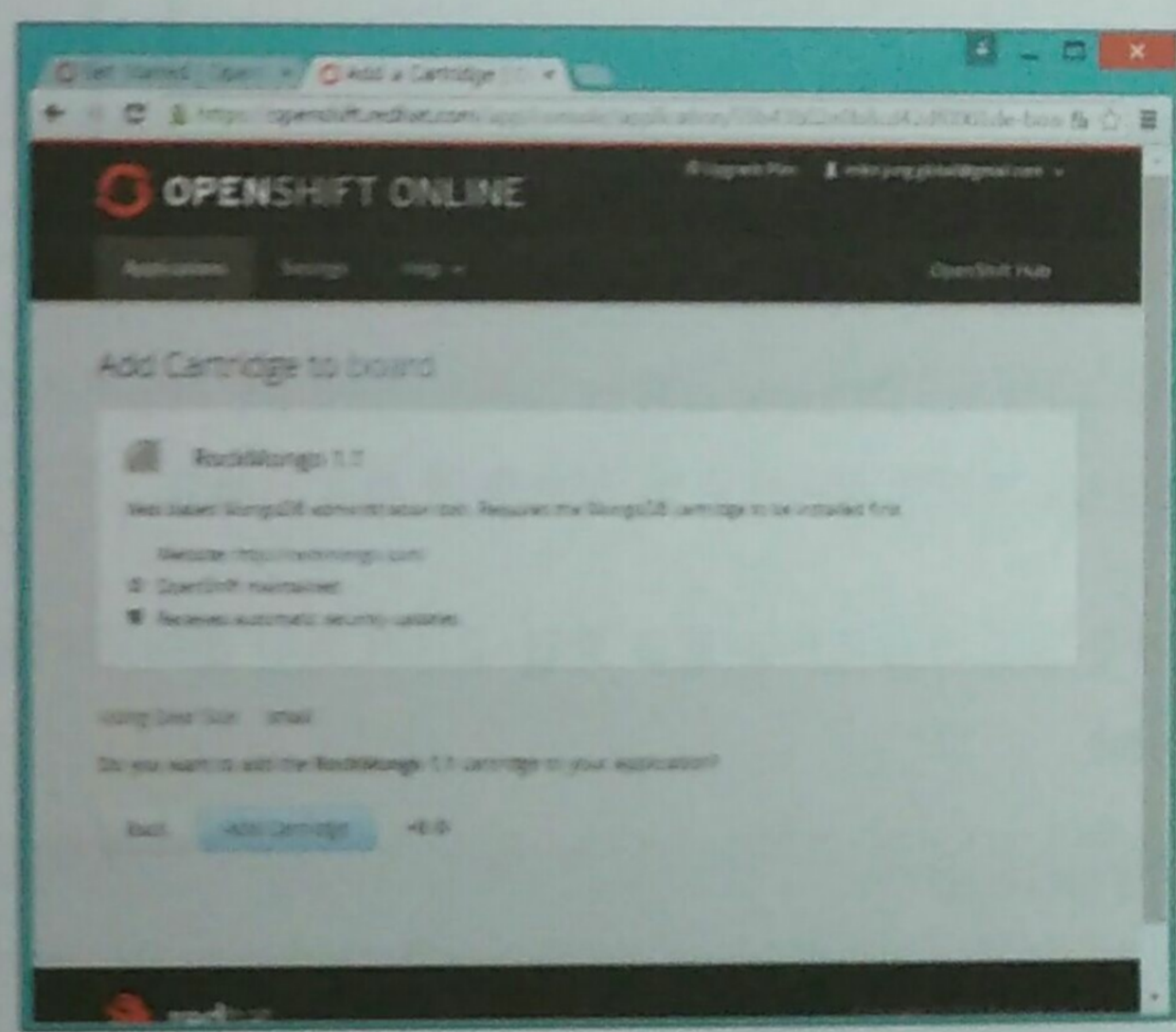


▲몽고디비 데이터베이스를 추가하는 화면



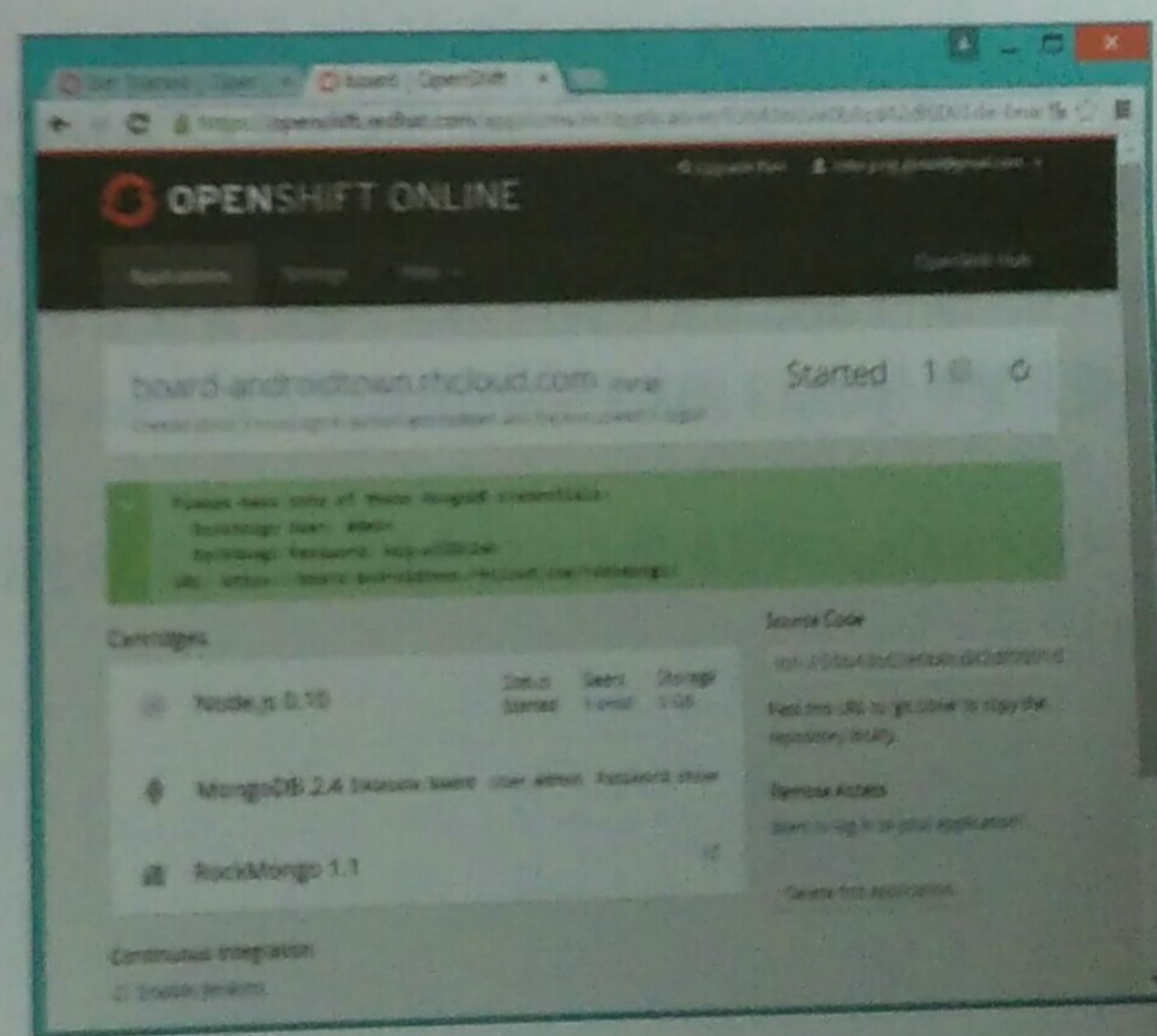
▲몽고디비 데이터베이스를 추가한 후의 화면

몽고디비 관리 도구인 RockMongo도 추가합니다. 아래쪽에 있는 'Add RockMongo 1.1' 링크를 클릭하여 추가합니다.



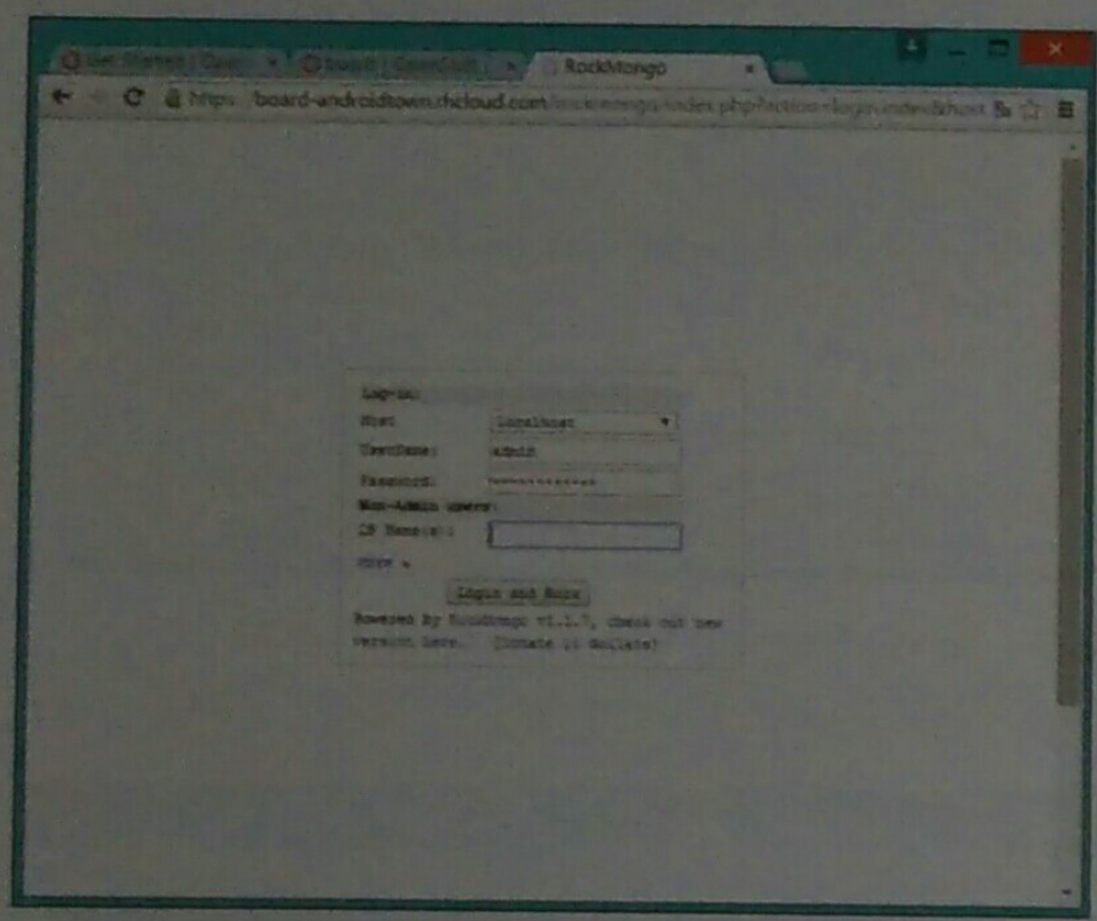
RockMongo를 추가하는 화면▶

[Add Cartridge] 버튼을 눌러 RockMongo를 추가하면 몽고디비 데이터베이스 관리 화면을 사용할 수 있습니다.

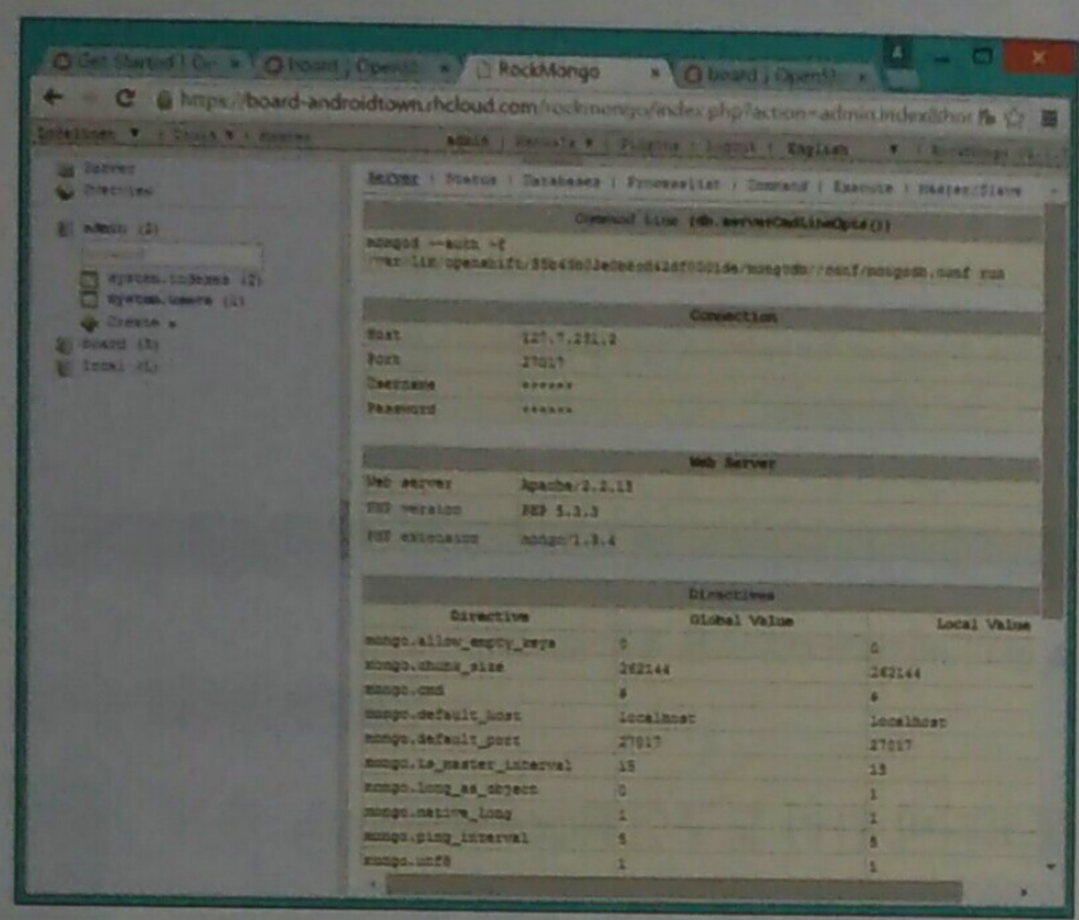


RockMongo를 추가한 후의 화면▶

데이터베이스 관리 화면을 조회하는 URL은 화면에 RockMongo를 추가하면 확인할 수 있습니다. 데이터베이스 관리 화면에 접속하면 데이터베이스에 접속할 수 있는 Host와 Port 정보가 표시됩니다. 데이터베이스 관리 화면에 로그인해서 Host와 Port를 확인합니다.



▲ 데이터베이스 관리 화면에 로그인하는 화면



▲ 데이터베이스 관리 화면에 표시된 Host와 Port

여기에서 확인한 Host와 Port 그리고 아이디와 비밀번호를 사용해 config.js 파일의 데이터베이스 접속 URL을 변경할 것이므로 이 정보를 잘 적어 두세요.

이제 명령 프롬프트를 열고 작업 공간이 있는 폴더로 이동한 후 다음과 같이 git clone 명령을 실행합니다.

```
% git clone
 ssh://55b43b02e0b8cd42df0001de@board-androidtown.rhcloud.com/~/git/board.git/
% cd board
```

여기에서 ssh:// 뒤에 나오는 주소는 애플리케이션이 만들어졌을 때 표시되는 설명글에 있습니다. 그 설명글을 보고 잘 입력하기 바랍니다. 이제 새로운 board 프로젝트 폴더가 만들어집니다. 이 프로젝트는 아무런 기능이 없는 간단한 노드 프로그램만 들어 있습니다. 여러분이 만든 [BoardExample] 폴더의 파일을 모두 [board] 폴더로 복사합니다.

그다음 config.js 파일을 열고 데이터베이스 URL을 변경합니다. server_port 정보는 8080으로 변경합니다.

참조 파일 | board>/config/config.js

```
……,

module.exports = {
  server_port : 8080,
  db_url : 'mongodb://admin:Wcq-u3lDr1mk@127.7.231.2:27017/board',

……,
```

app.js 파일을 열고 서버를 시작할 때 전달하는 파라미터를 변경합니다.

참조 파일 | board>/app.js

```
……,

var server = http.createServer(app).listen(process.env.OPENSHIFT_NODEJS_PORT ||
             process.env.OPENSHIFT_INTERNAL_PORT ||
             app.get('port'),
             process.env.OPENSHIFT_NODEJS_IP ||
             process.env.OPENSHIFT_INTERNAL_IP ||
             '127.0.0.1',
             function() {

……,
```

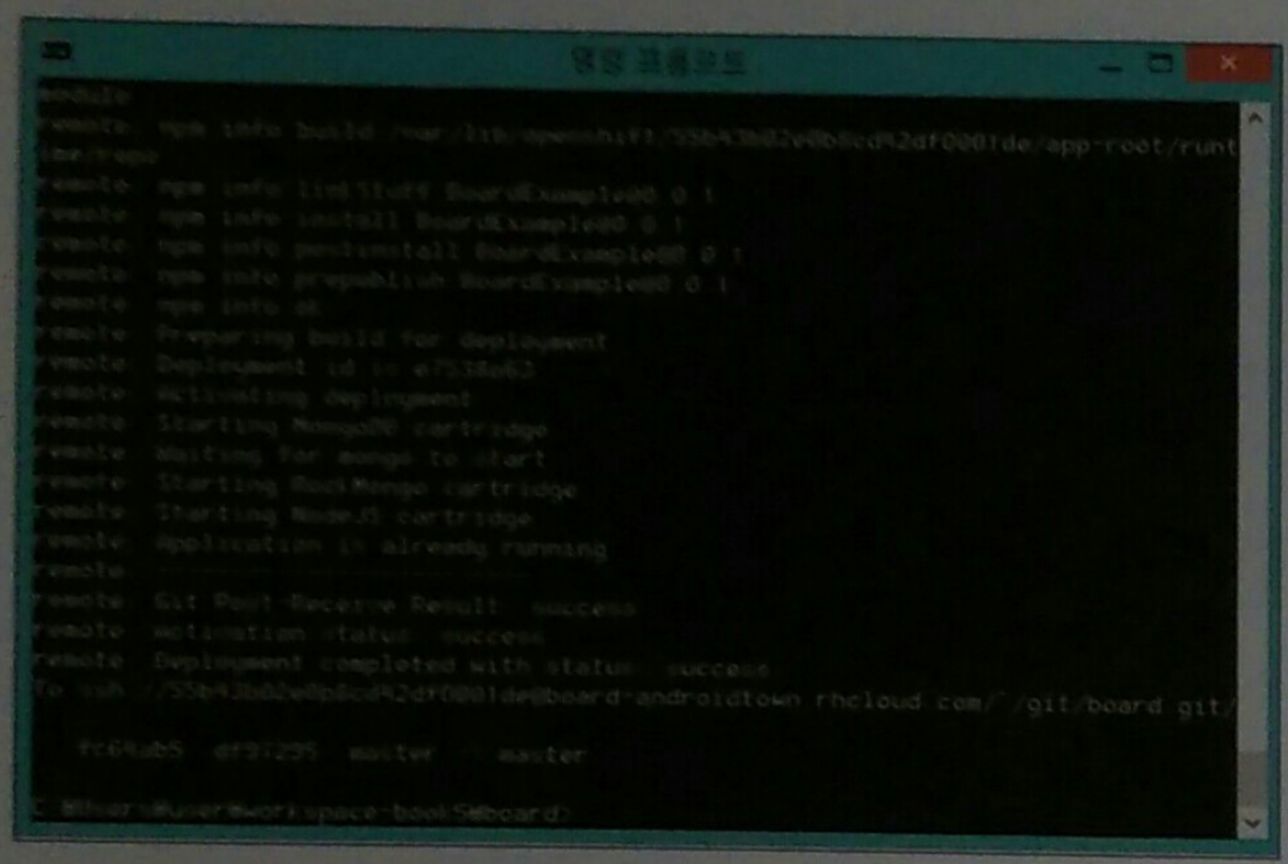
이 파일에서는 환경 변수를 사용해 호스트와 포트 정보를 모두 파라미터로 전달하도록 코드를 변경했습니다. 이제 명령 프롬프트를 열고 데이터베이스에 접속할 수 있도록 포트포워딩 설정을 합니다.

```
% rhc port-forward board
```

드디어 파일을 업로드할 수 있게 되었습니다. [board] 폴더 안에서 다음 명령을 실행하면 파일이 업로드됩니다.

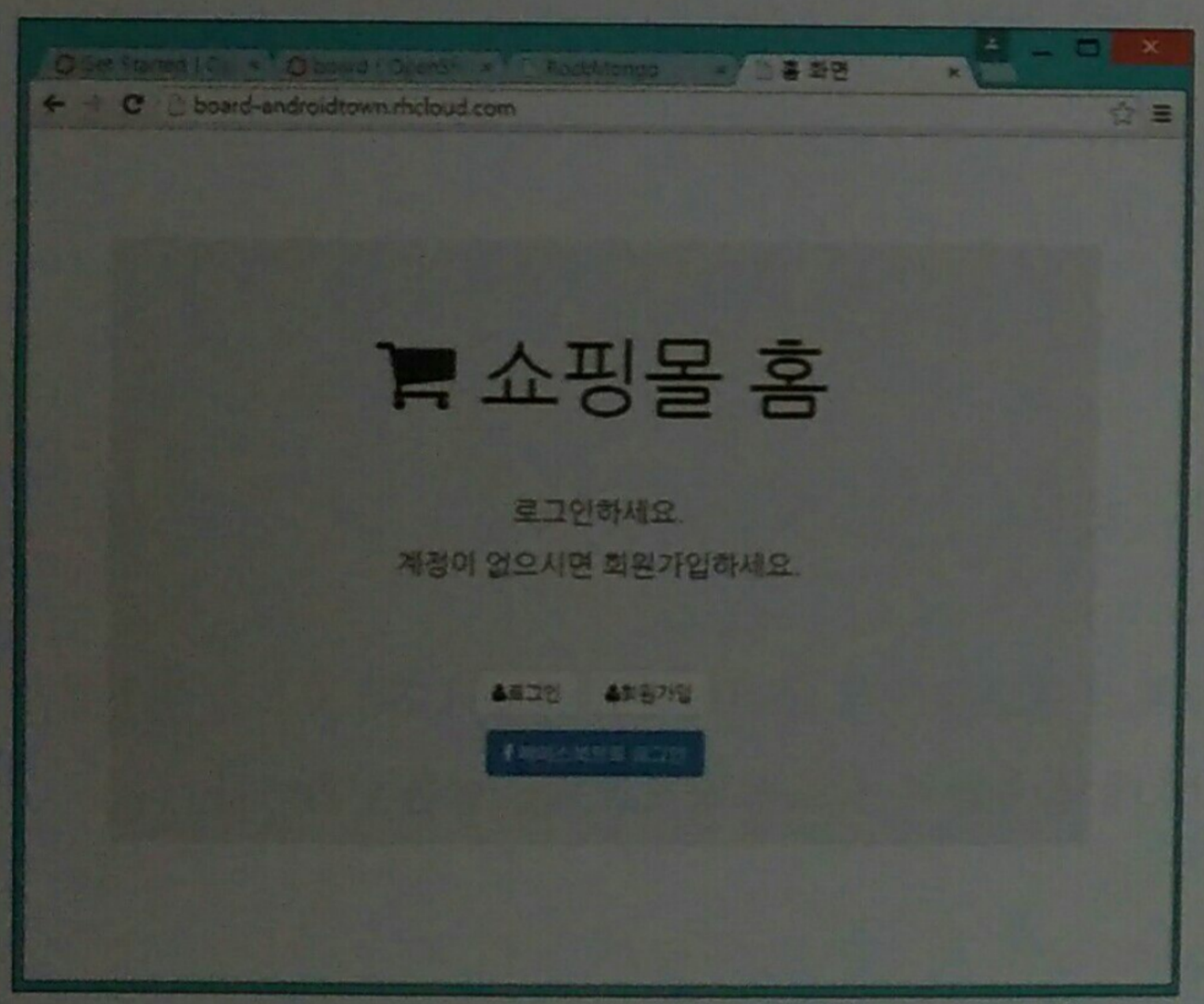
```
% git add .
% git commit -m "My changes"
% git push
```

오픈시프트 클라우드는 파일을 업로드하면 자동으로 서버를 만들고 재시작합니다.



▲ 프로젝트 파일을 업로드한 후 서버를 자동으로 시작한 화면

마지막에 입력한 git push 명령을 실행했을 때 서버가 자동으로 시작됩니다. 드디어 여러분이 만든 서버를 오픈시프트 클라우드에서 접속할 수 있습니다.



▲ 오픈시프트 클라우드에 올린 게시판 프로젝트를 조회한 화면

이제 오픈시프트 클라우드를 사용해서 여러분이 만든 프로그램을 전 세계 사용자들이 조회할 수 있게 되었습니다. 서버를 실행했을 때 오류가 발생한다면 로그를 확인해 봅니다. 로그를 확인하려면 명령 프롬프트를 열고 다음 명령을 실행합니다.

```
% rhc ssh -a board
```

이 명령은 서버의 셸(Shell)로 들어가도록 만들어 줍니다. 로그 폴더로 이동한 후 vim 명령으로 파일을 열어 봅니다.

```
> cd $OPENSHIFT_LOG_DIR
> vim nodejs.log
```

편집기를 끝내려면 다음 명령을 입력합니다.

```
:q!
```

vim은 리눅스나 유닉스에서 사용하는 편집기이므로 편집기를 다루는 명령에 익숙하지 않다면 조금 불편할 수 있습니다. 하지만 몇 가지 간단한 사용 방법은 이내 익숙해질 것입니다.

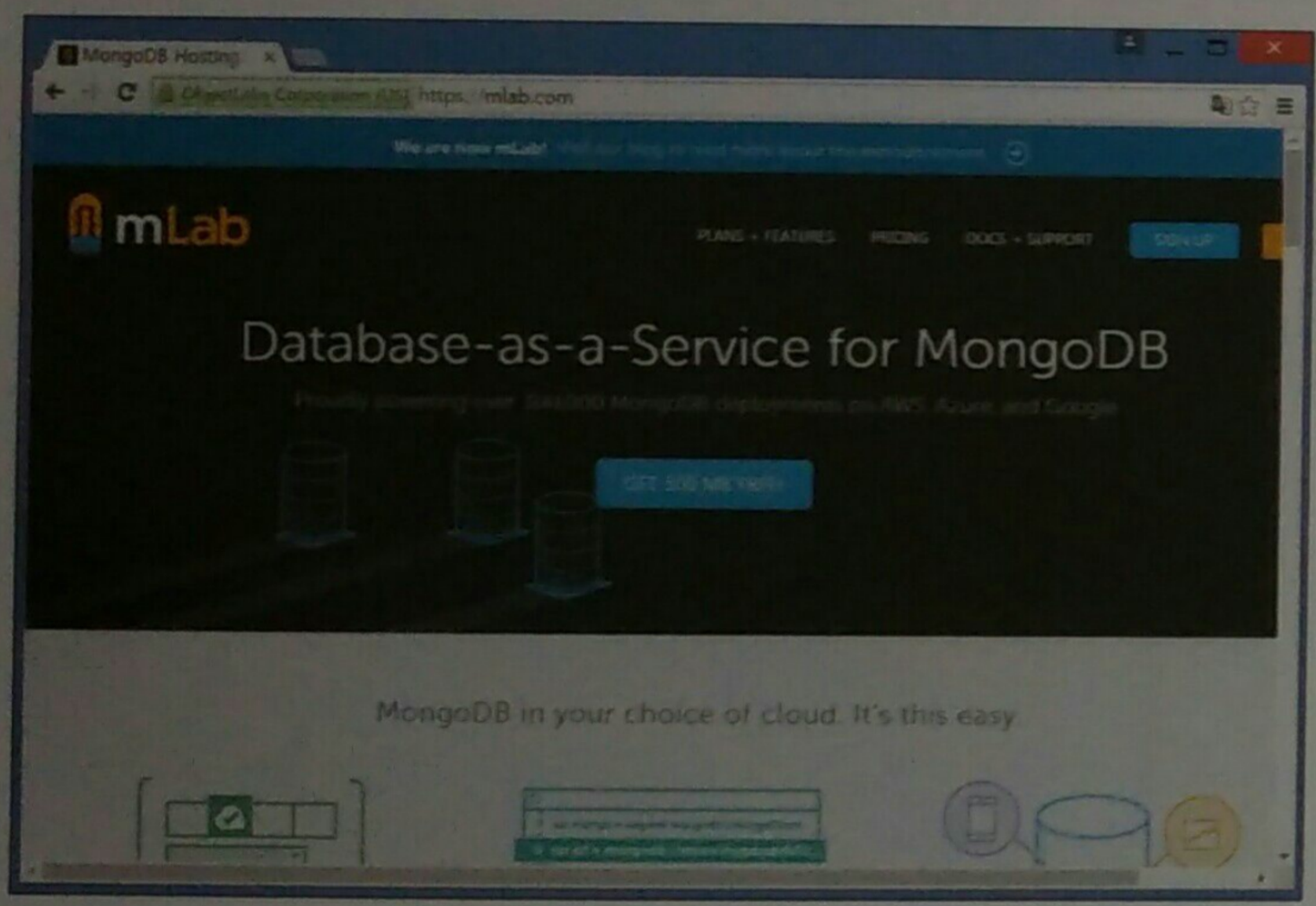
지금까지 오픈시프트 클라우드에 여러분이 만든 프로젝트를 올리는 과정을 살펴보았습니다. 오픈시프트 클라우드를 익히면서 헤로쿠 클라우드와는 조금 다른 느낌을 받았을 것입니다. 오픈시프트 클라우드는 무료 계정을 사용해도 헤로쿠 클라우드처럼 서비스를 중단하지 않고 아마존 클라우드처럼 사용 기한이 정해져 있지도 않습니다. 또 한국에서 사이트에 접속할 때 조금 더 빠른 경우가 많아 모바일 앱 서버를 만들어 클라우드에 올릴 때 유리합니다.

15-4 | 아마존 클라우드에 올리기

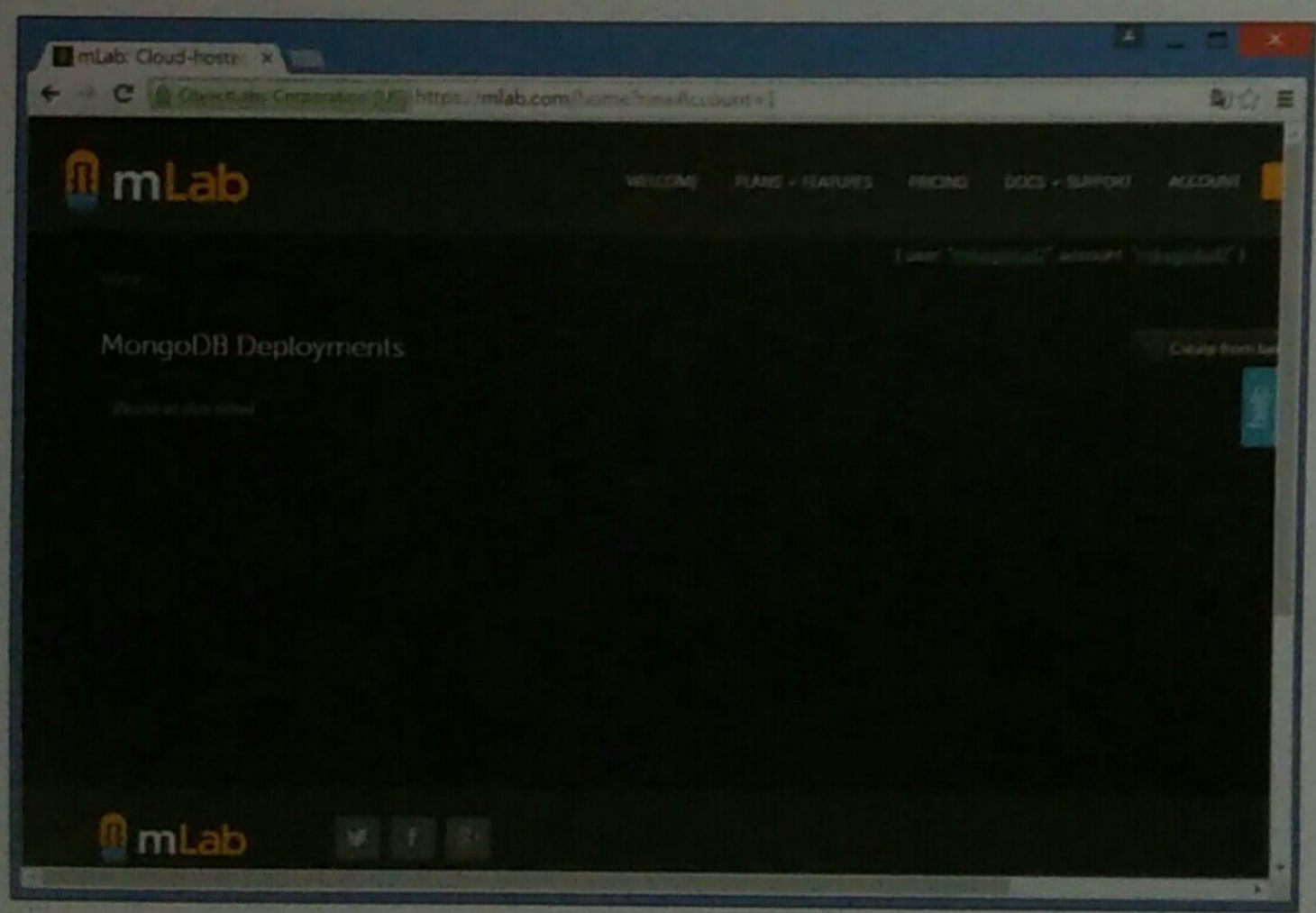
아마존(Amazon) 클라우드는 많은 사용자가 있는 대표적인 클라우드입니다. 특히 안정적인 서비스를 할 수 있어 실무에서도 많이 사용하고 있습니다. 아마존 클라우드는 서버 운영에 필요한 다양한 기능을 제공하기 때문에 여러분이 만든 노드 프로그램을 올리려면 조금 복잡한 과정을 거쳐야 합니다. 그러나 쉽고 간편하게 올릴 수 있는 방법도 제공합니다. 여기에서는 가장 간단한 방법으로 클라우드에 소스 파일을 올린 후 설정하는 방법을 알아보겠습니다.

먼저 mlab MongoDB 사이트에 회원가입을 한 다음 오른쪽 위의 [SIGN UP] 링크를 눌러 회원가입을 합니다.

▶ http://mlab.com

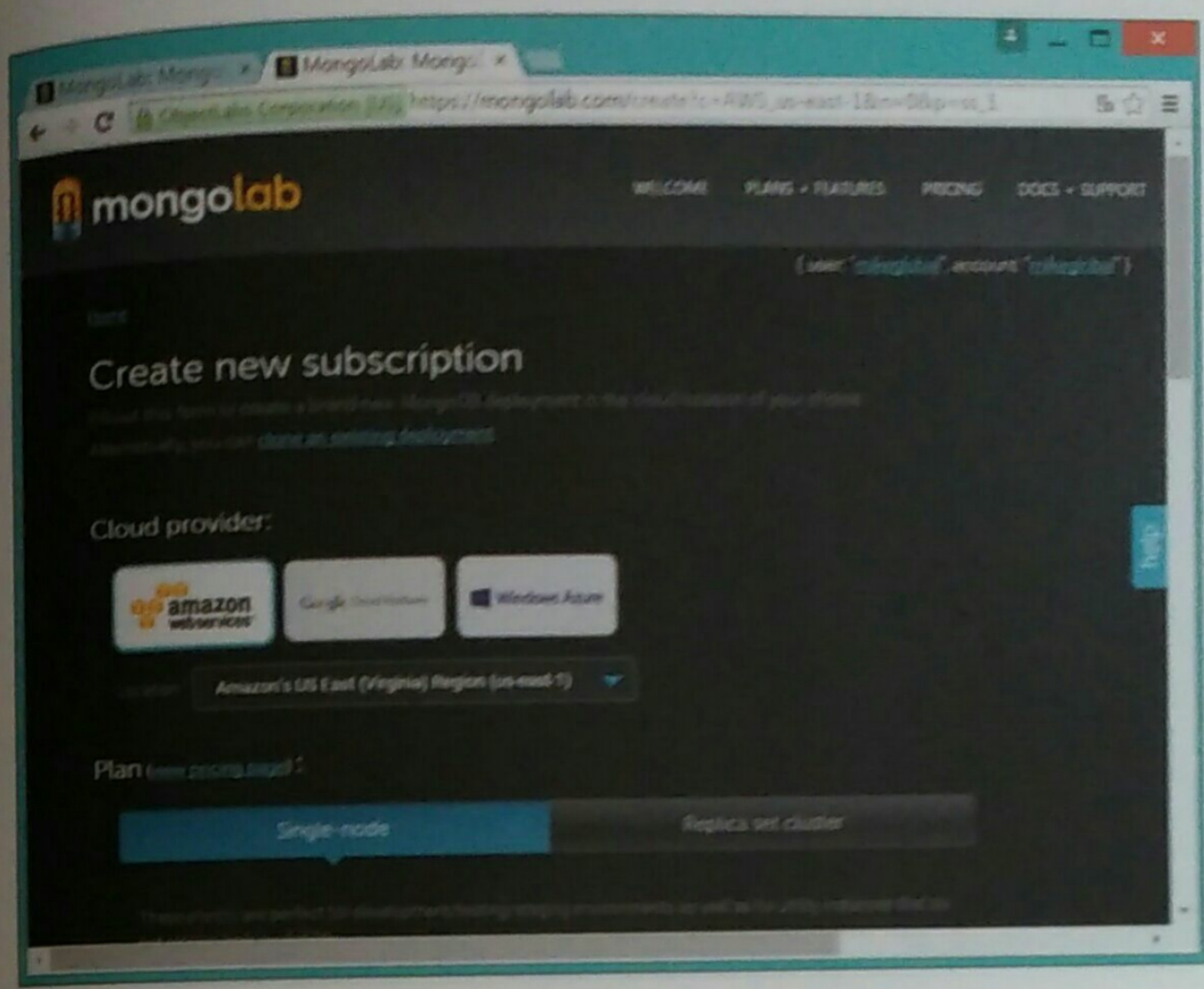


◀ mLab MongoDB 사이트



◀ 로그인 후 MongoLab의 홈 화면

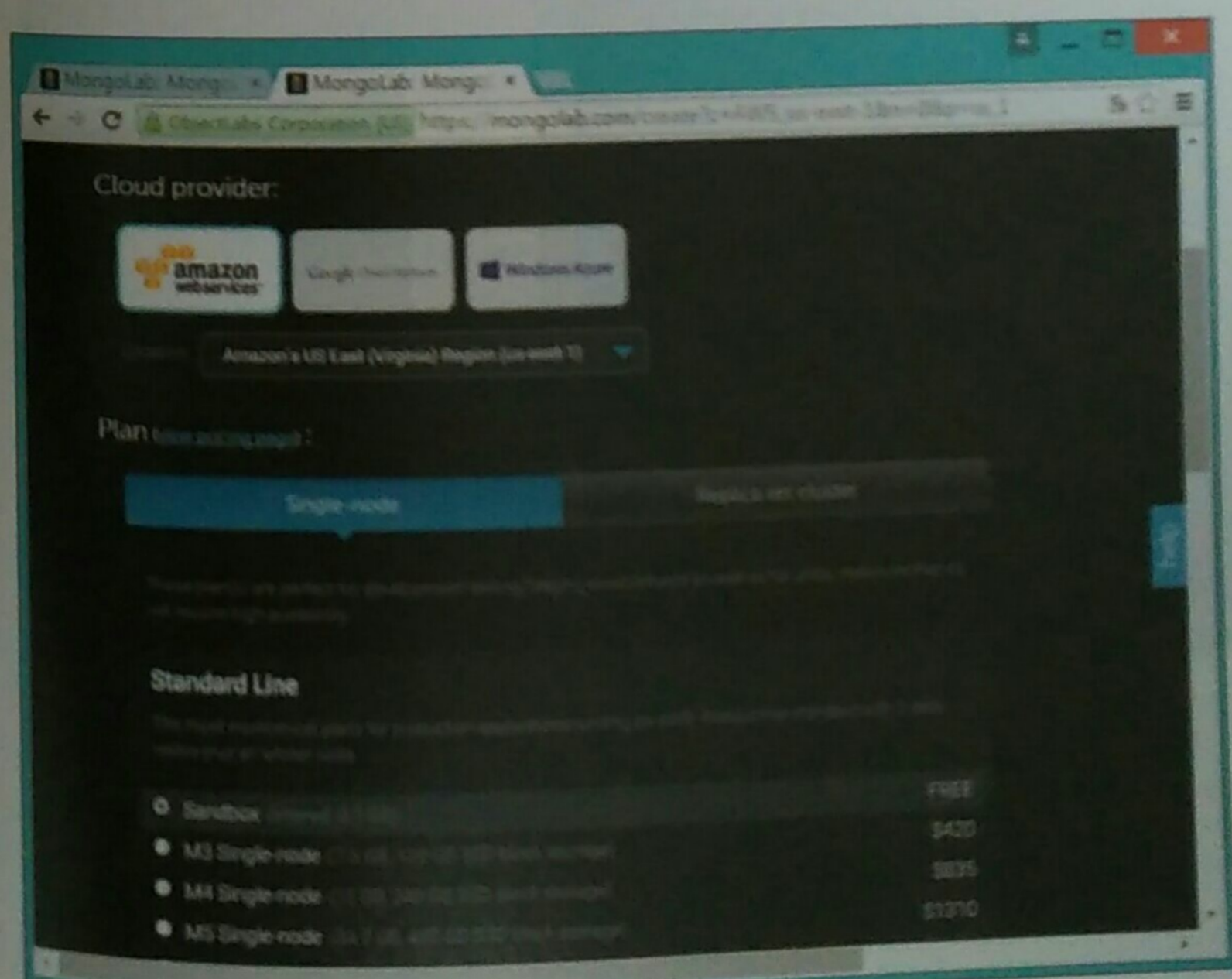
오른쪽 위의 [Create new] 버튼을 눌러 새로운 몽고디비 인스턴스를 만듭니다. 그러면 Create new subscription 페이지가 나타납니다. 이 페이지에서 몽고디비 인스턴스를 만들 때 필요한 설정을 합니다.



▲몽고디비 인스턴스를 만드는 설정 화면

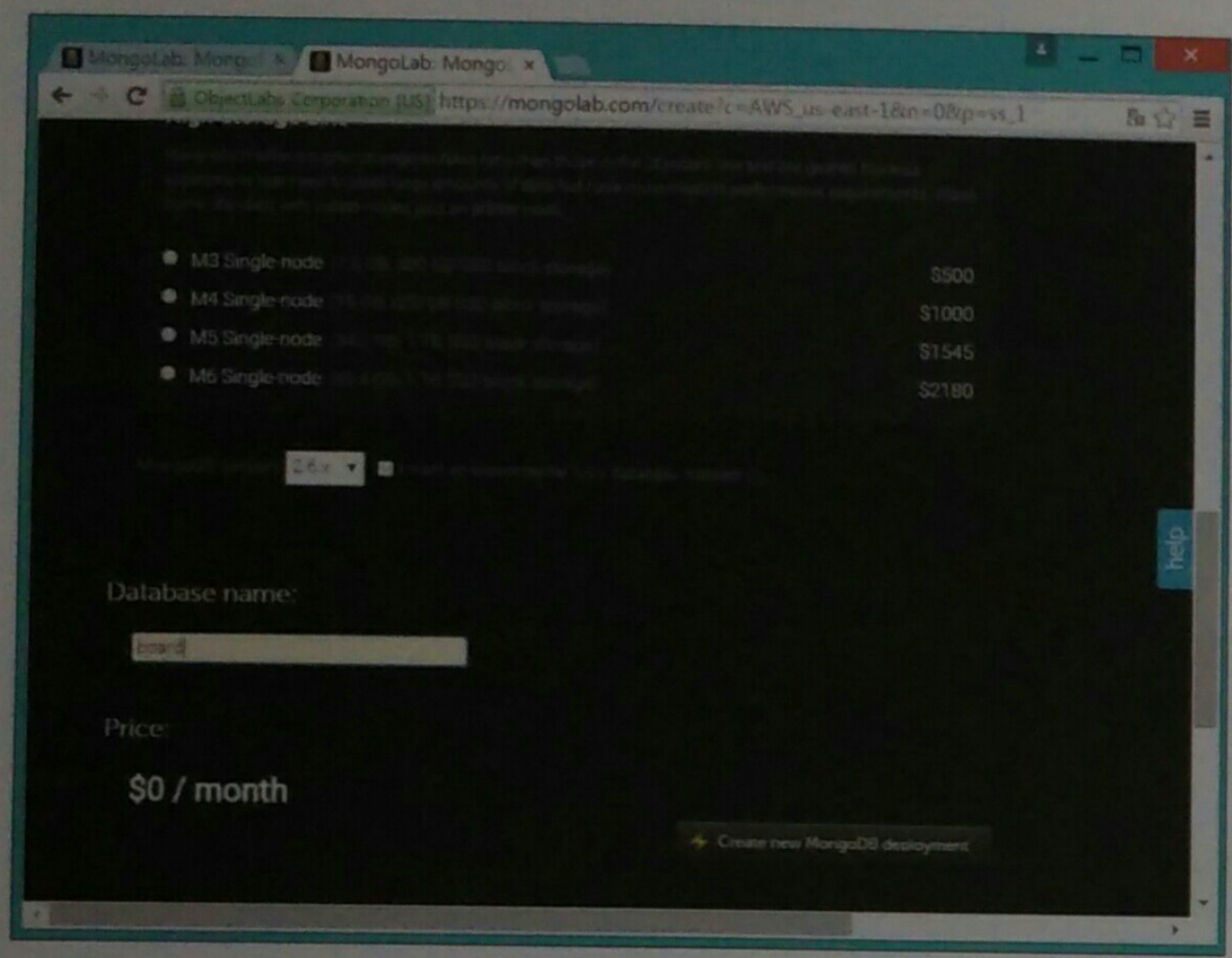
Cloud provider: 항목은 amazon web services를 선택합니다. 이 항목은 기본 값으로 설정되어 있습니다. 그 아래의 Location: 항목은 아시아 지역에서 사용하기 위해 Singapore을 선택합니다. 어떤 노드를 선택할 것인지를 물어보는 Plan 항목은 무료 노드를 사용할 것이므로 Single-node를 선택합니다. 그리고 Sandbox 항목을 선택합니다.

주의 만약 Single-node 항목이 보이지 않는다면 view pricing page 링크를 눌러 나타난 페이지에서 [Free] 항목을 선택합니다. 그런 다음 다시 이전 페이지로 돌아오면 Single-node가 나타납니다.



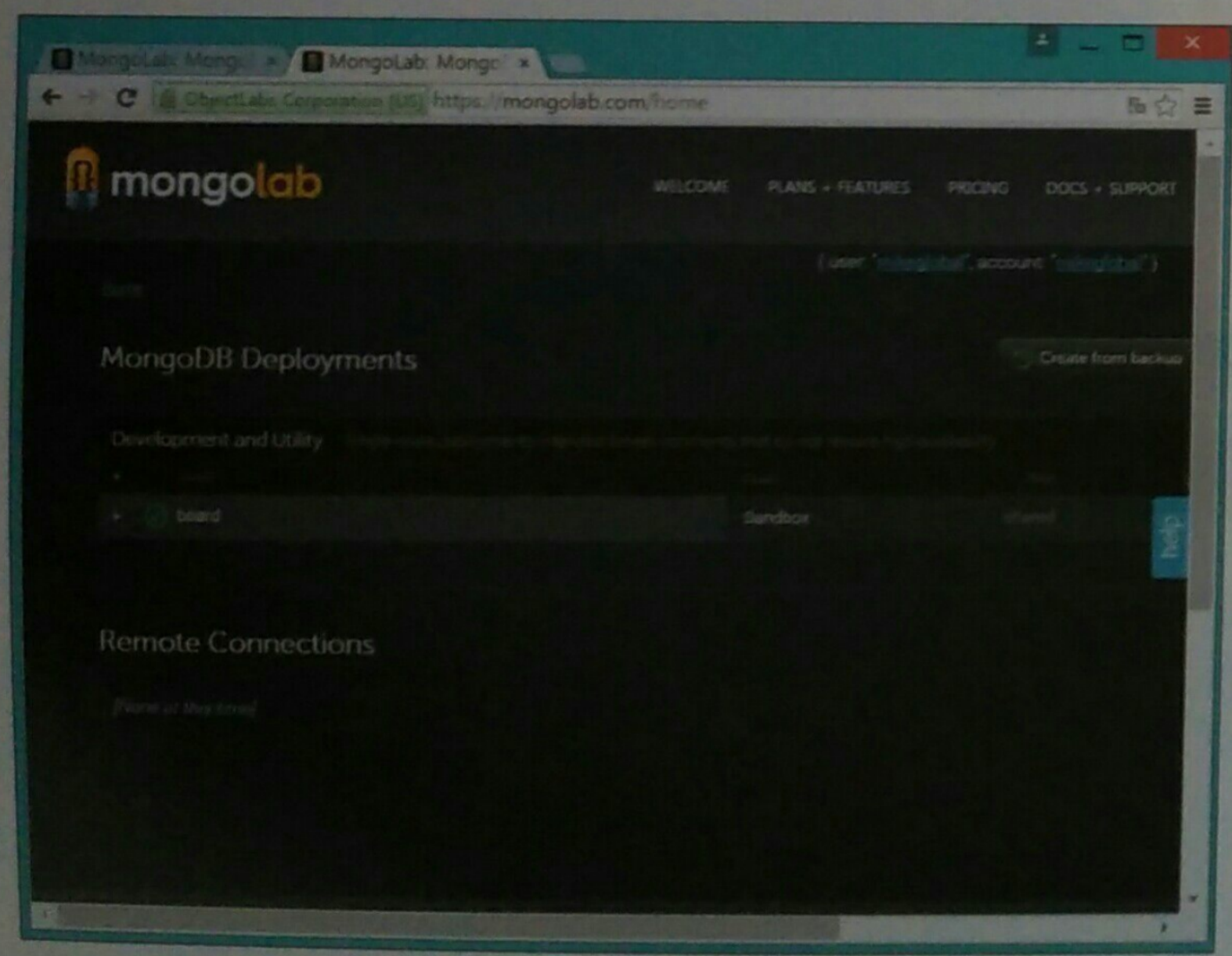
◀ Cloud provider와 Plan 항목을 선택한 화면

아래쪽의 데이터베이스 이름은 board로 입력합니다. Price 항목이 $0/month로 표시된 것을 확인한 후 [Create new MongoDB deployment] 버튼을 누릅니다.



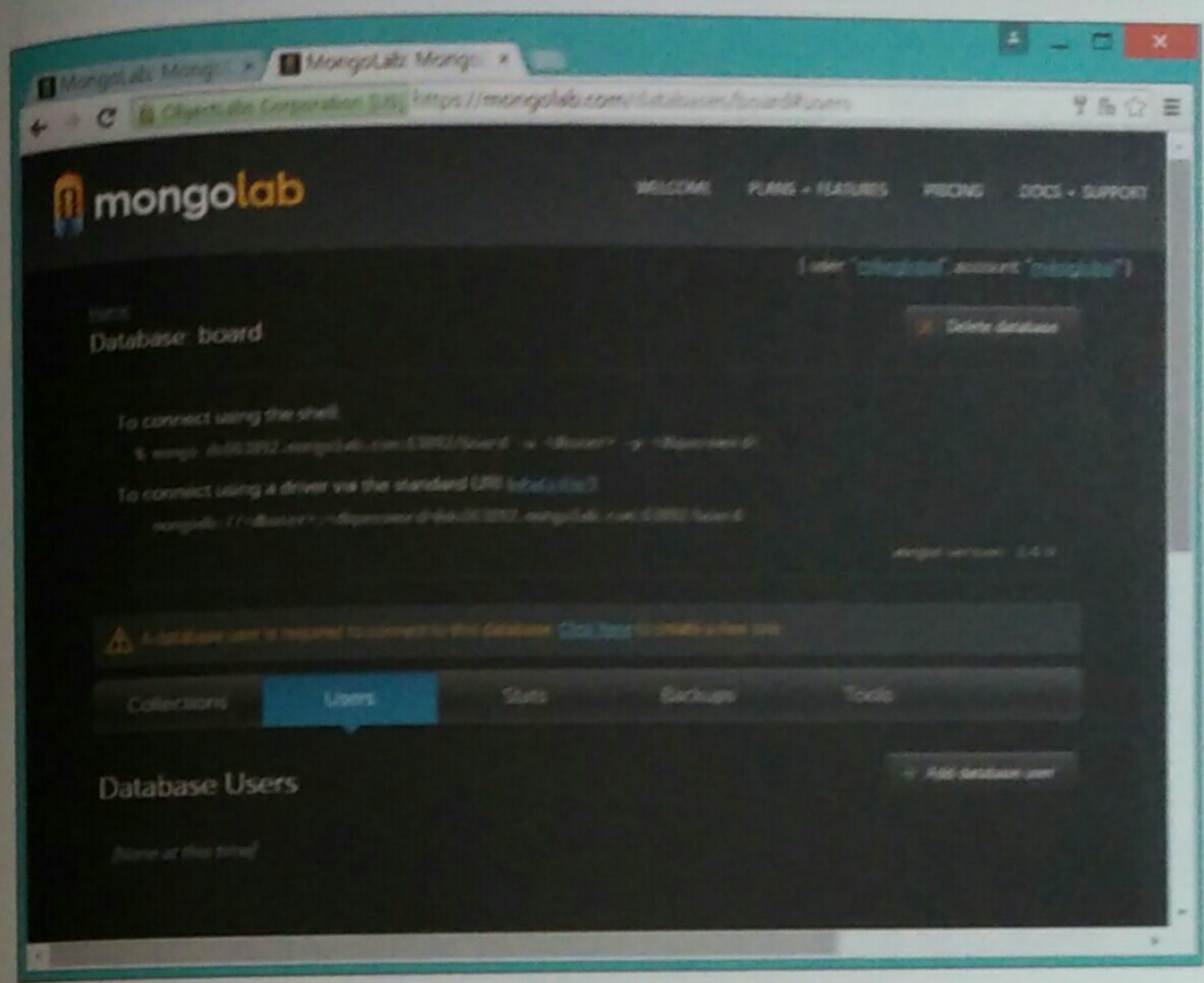
▲ 데이터베이스 이름 입력 상자와 가격이 표시되어 있는 화면

몽고디비 대시보드 화면으로 돌아오면 새로운 board 데이터베이스가 만들어져 있습니다.



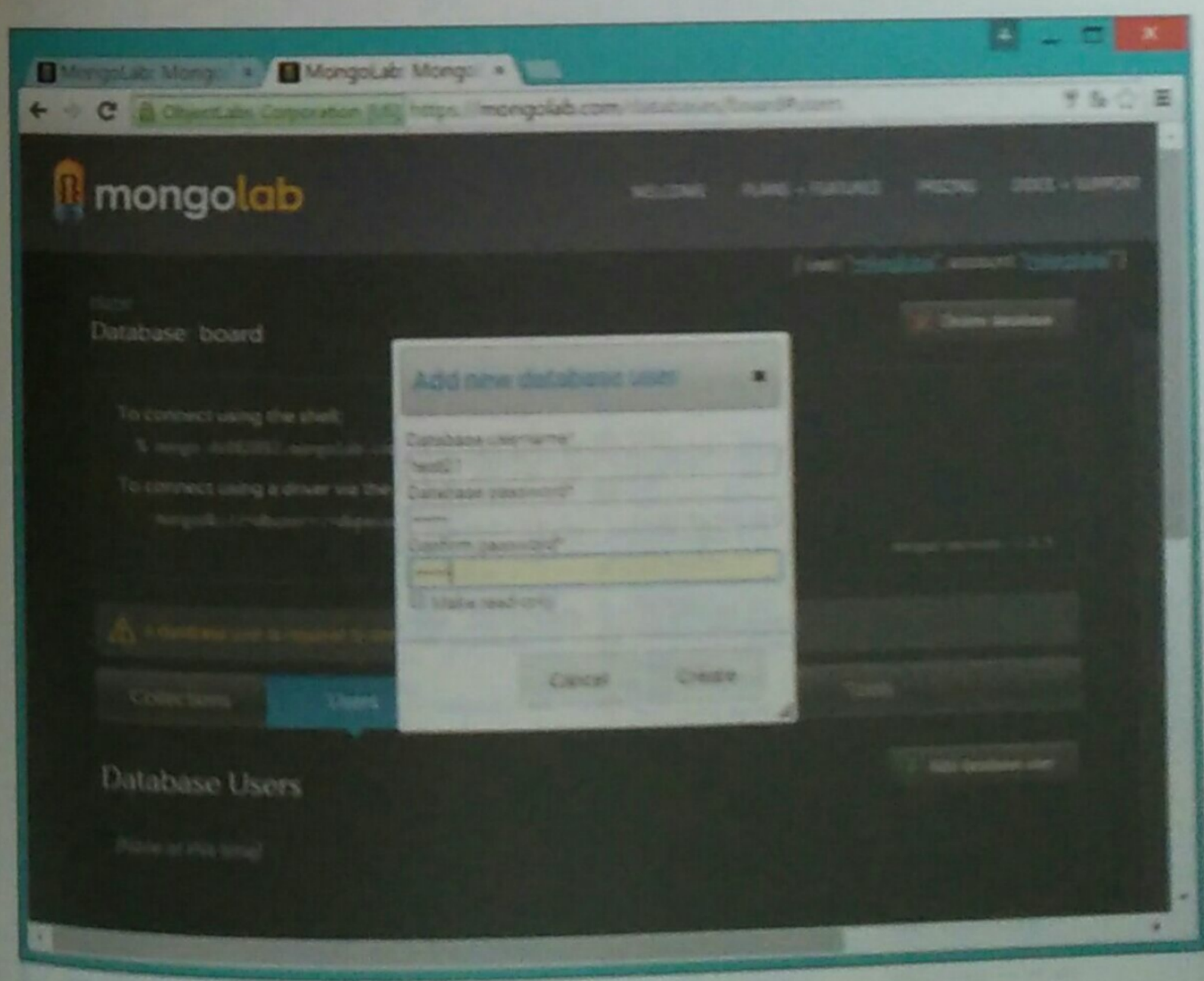
▲ 새로 만들어진 board 데이터베이스

새로 만든 board 데이터베이스를 클릭하면 상세 화면으로 이동합니다. 상세 화면에서 [Users] 탭을 클릭한 후 [Add database user] 버튼을 클릭하면 사용자를 추가할 수 있습니다.



▲ 데이터베이스 상세 화면에서 [Users] 탭을 클릭한 경우

여기에서는 test01로 사용자 이름을 만들겠습니다.



▲ 데이터베이스에 사용자 계정을 추가하는 화면

이제 데이터베이스에 연결할 수 있는 URL 정보를 만들 수 있습니다.

mongodb://test01:123456@ds063892.mongolab.com:63892/board

앞에서 만든 [board] 폴더를 파일 탐색기로 찾아 복사한 다음 [board2] 폴더를 만듭니다. 그리고 config.js 파일에서 server_port와 db_url 속성 값을 위의 값으로 변경합니다.

주의! server_port 속성으로 설정하는 포트가 맞지 않는다면 나중에 애플리케이션을 실행했을 때 오류가 발생할 수 있습니다.

참조 파일 | board2>/config/config.js

```
……

module.exports = {
  server_port : 8081,
  db_url : 'mongodb://test01:123456@ds063892.mongolab.com:63892/board',

……
```

app.js 파일을 열고 서버를 실행하기 위해 포트 설정 부분을 수정합니다.

참조 파일 | board2>/app.js

```
……

var server = http.createServer(app).listen(process.env.PORT || app.get('port'), function() {

……
```

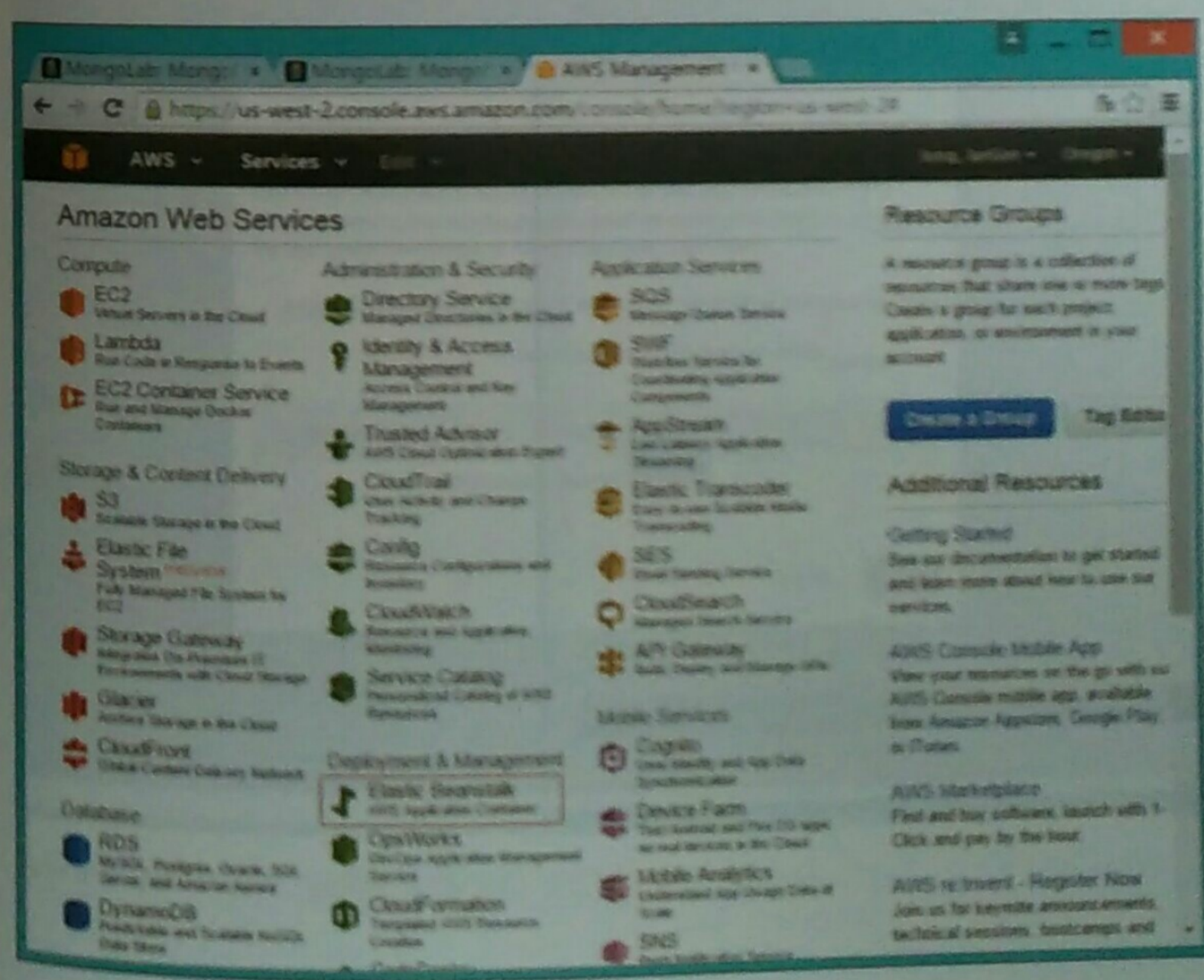
[board2] 폴더는 폴더 안에 있는 [node_modules] 폴더를 삭제한 후 압축 파일로 만들어야 클라우드에 올릴 수 있습니다. 파일 탐색기를 열고 프로젝트 폴더 안에 있는 모든 파일을 선택해서 board2.zip 파일을 만듭니다. 이제 클라우드에 올릴 수 있는 파일을 준비했으니 아마존 웹 서비스(AWS)에 계정을 만듭니다.

▶ http://aws.amazon.com/

▲ 아마존 웹 서비스(AWS) 사이트

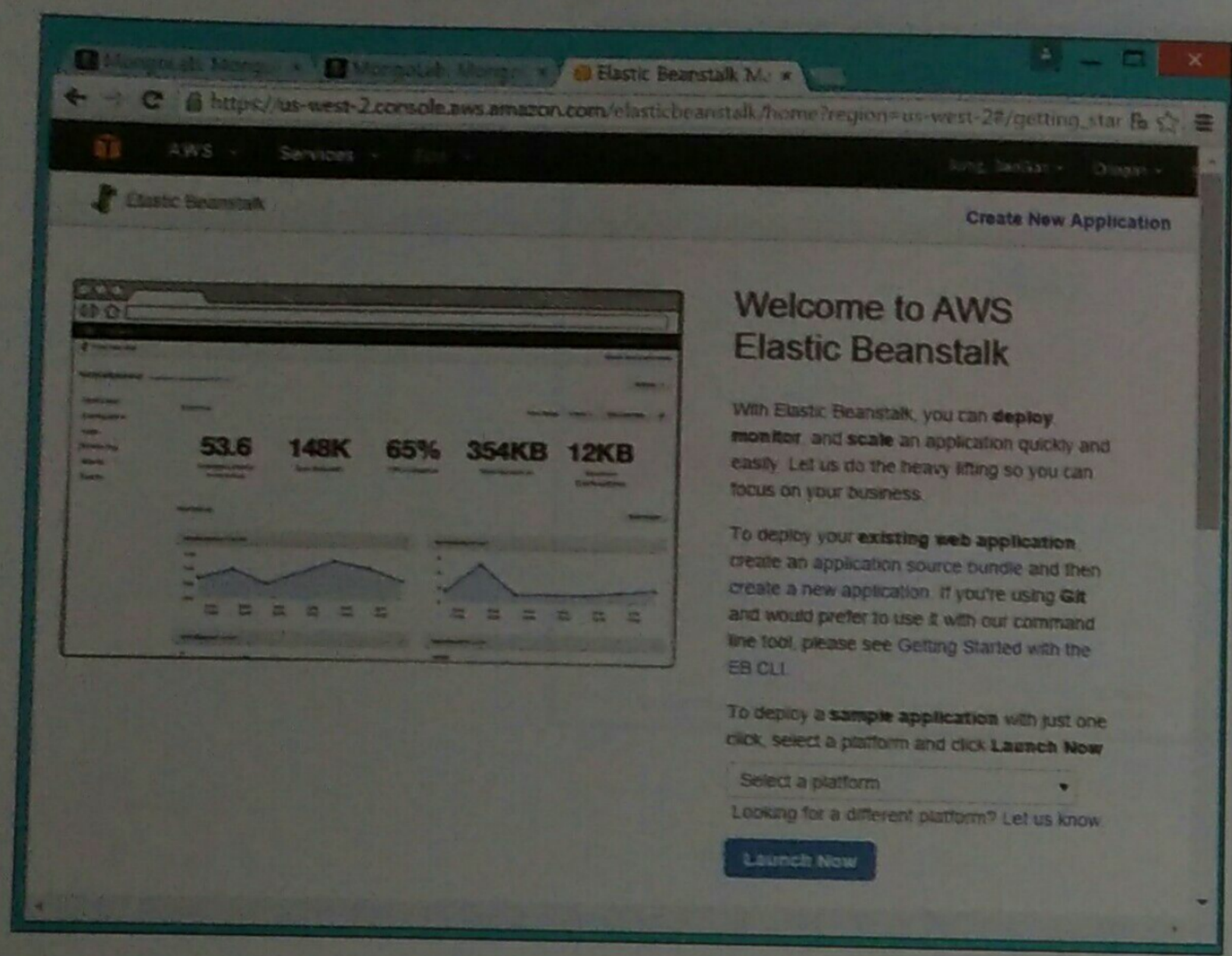
[무료 계정 생성] 버튼을 눌러 계정을 만듭니다. 계정을 만드는 과정은 안내 메시지만 따르면 쉽게 진행할 수 있습니다. 계정을 만드는 과정에서 신용카드 정보를 넣는 화면이 나오는데, 신용카드 정보를 넣더라도 무료 제공 기간에는 결제되지 않습니다. 또 모든 작업이 끝난 후 아마존 사이트(http://www.amazon.com)에서 [Manage payment options] 메뉴를 선택하면 신용카드 정보를 삭제할 수도 있습니다.

계정이 정상적으로 만들어졌다면 새로 로그인합니다. 관리 콘솔이 나타날 것입니다. Deployment and Managed 항목 아래에 있는 Elastic Beanstalk를 사용합니다.



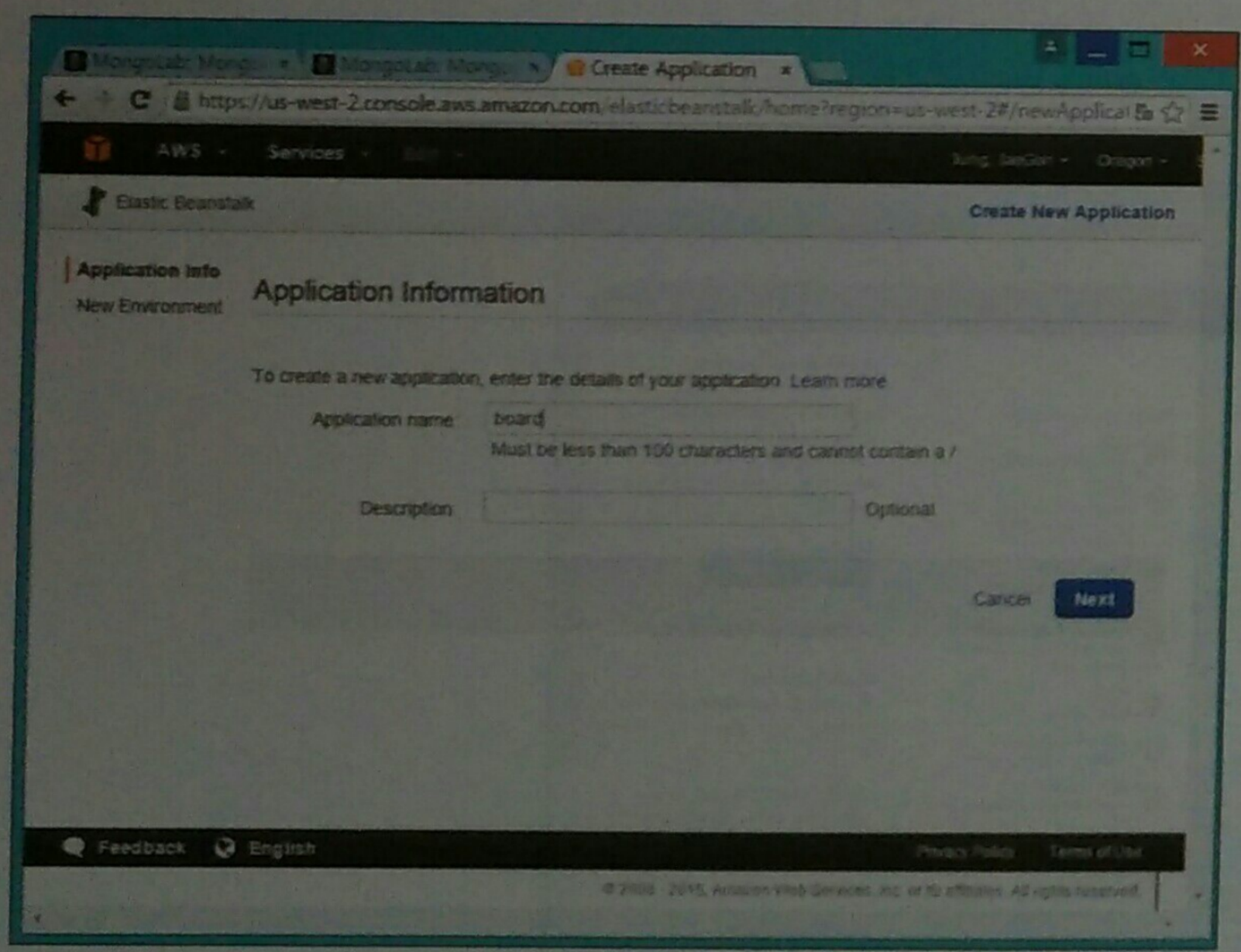
▲ AWS 관리 콘솔

Elastic Beanstalk 항목을 클릭하면 Elastic Beanstalk 화면이 나타납니다.



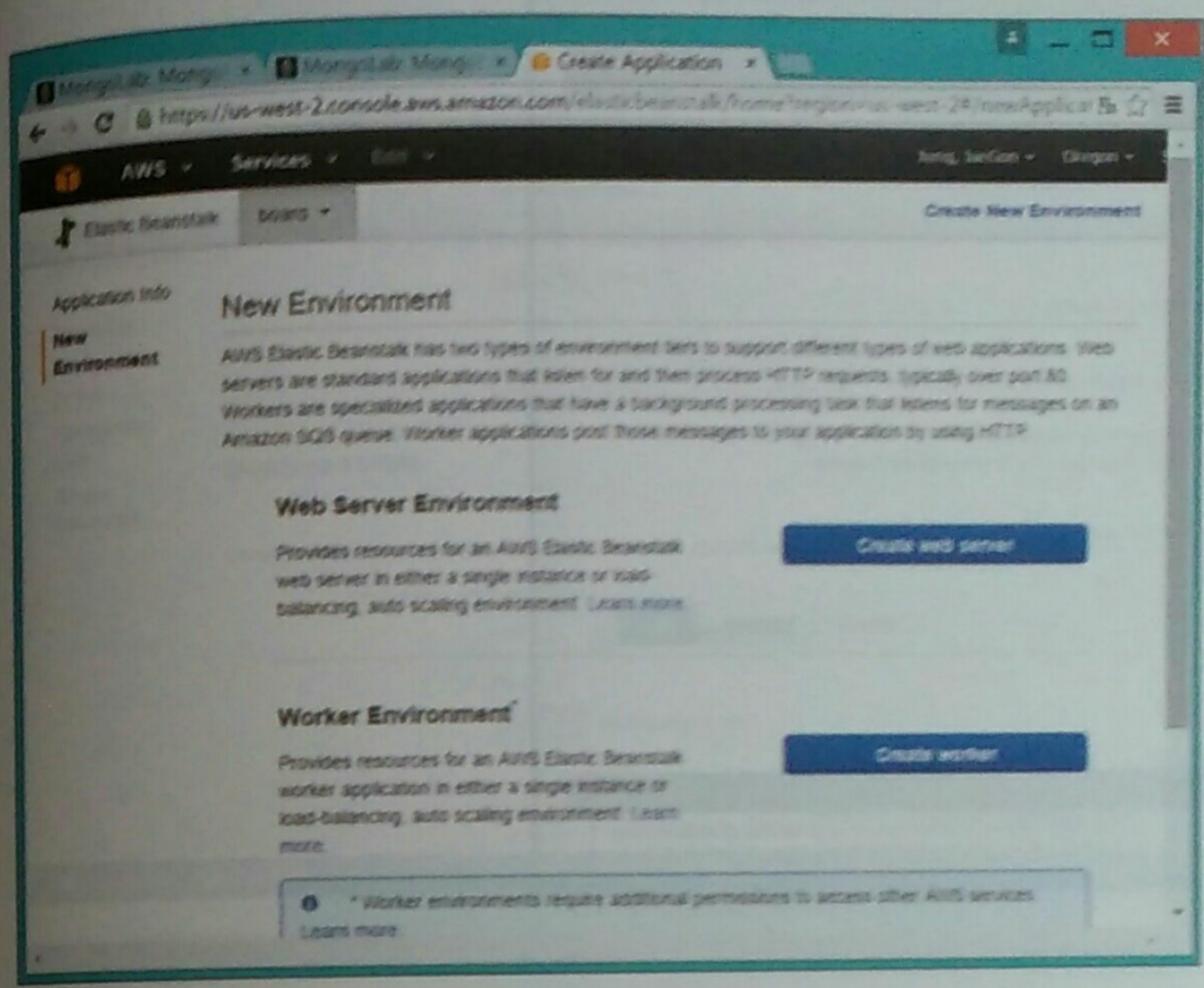
▲ Elastic Beanstalk 화면

오른쪽 위에 있는 'Create New Application' 링크를 눌러 새로운 애플리케이션을 만들 수 있는 화면으로 넘어갑니다. 첫 화면은 애플리케이션 정보를 입력하는 화면입니다.



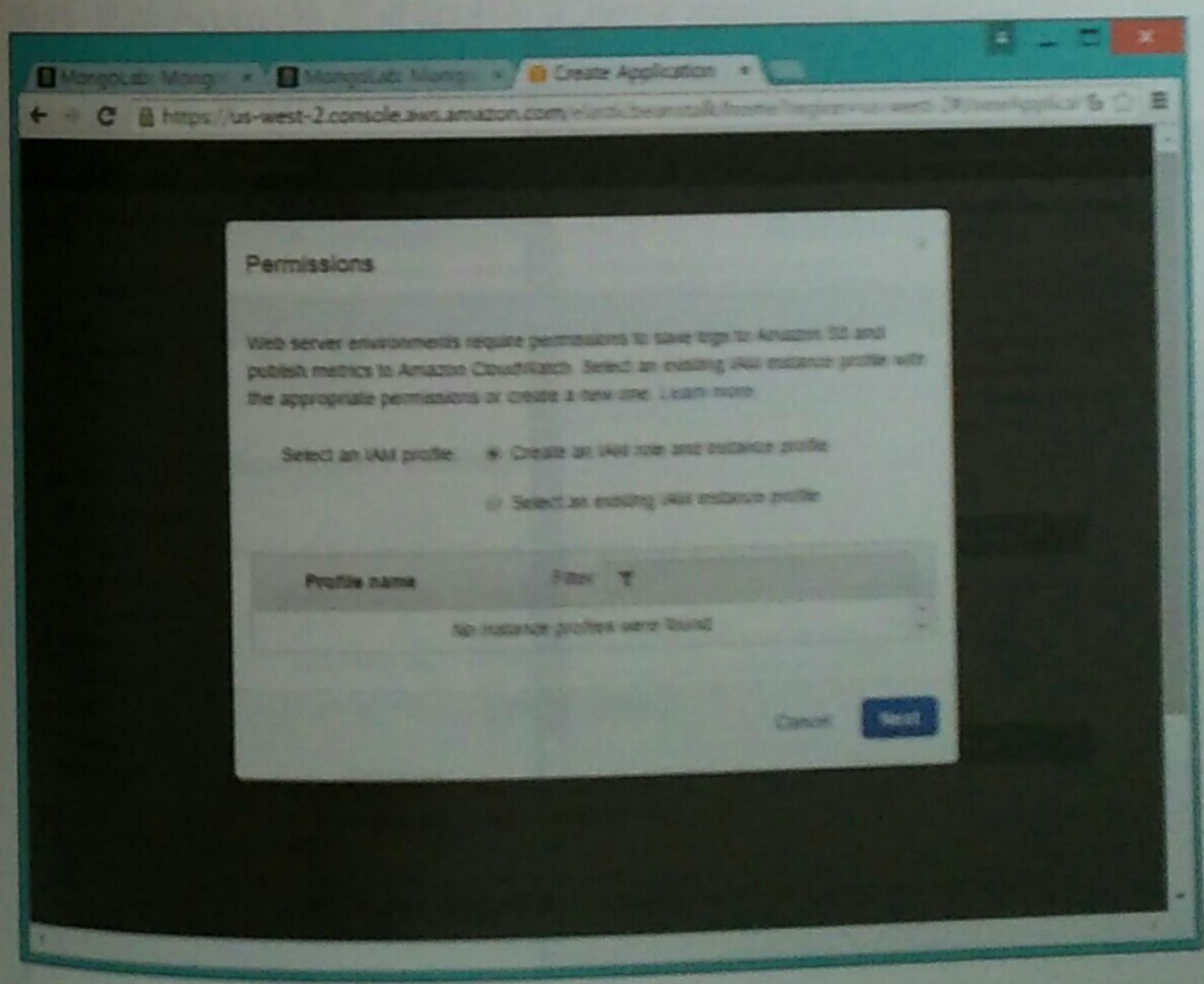
▲ 애플리케이션 정보를 입력하는 화면

Application name 항목에 board를 입력한 후 [Next] 버튼을 누르면 새로운 환경을 선택하는 화면이 나타납니다.



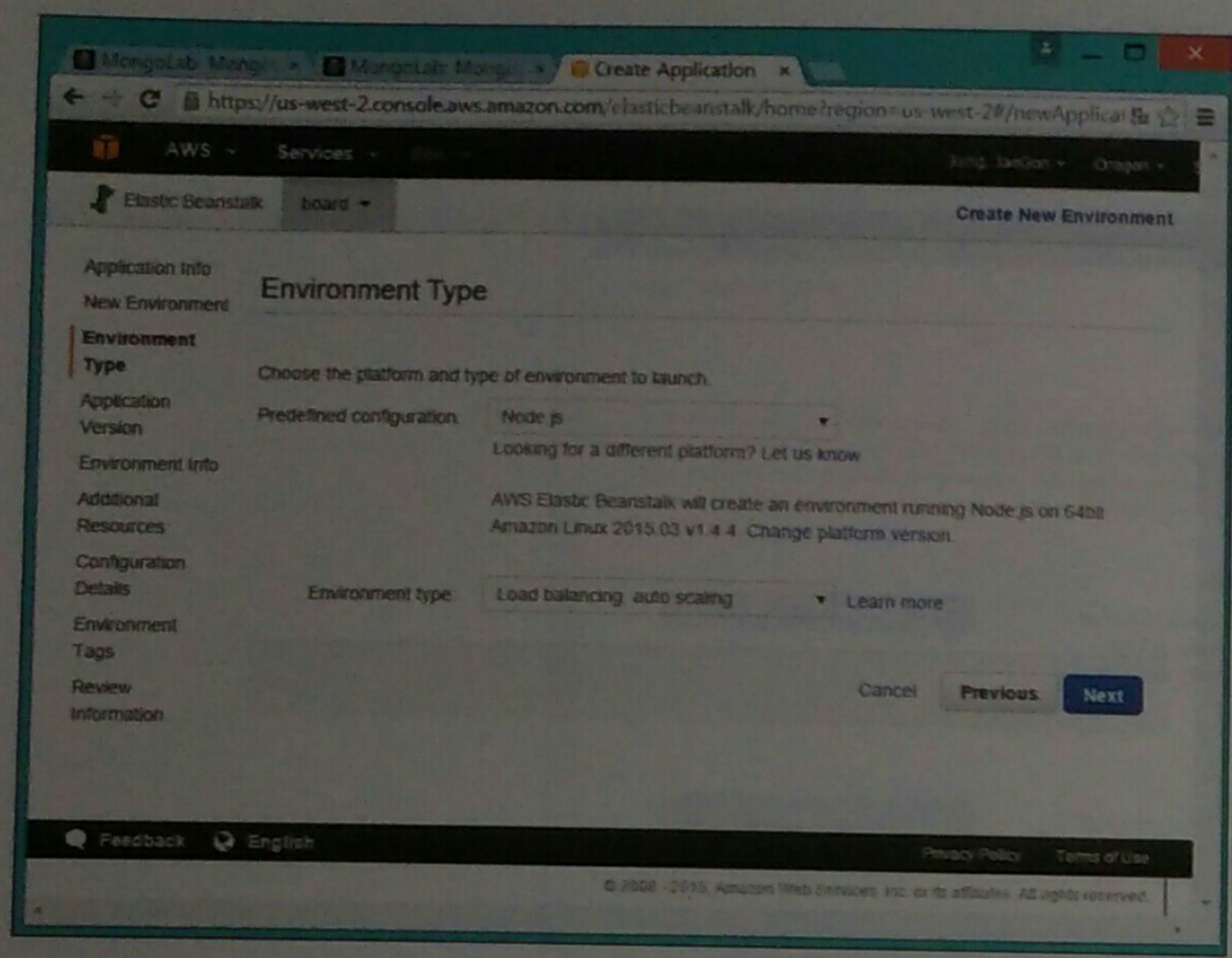
▲ 새로운 환경을 선택하는 화면

[Create web server] 버튼을 클릭하면 권한을 요청하는 대화상자가 나타납니다.



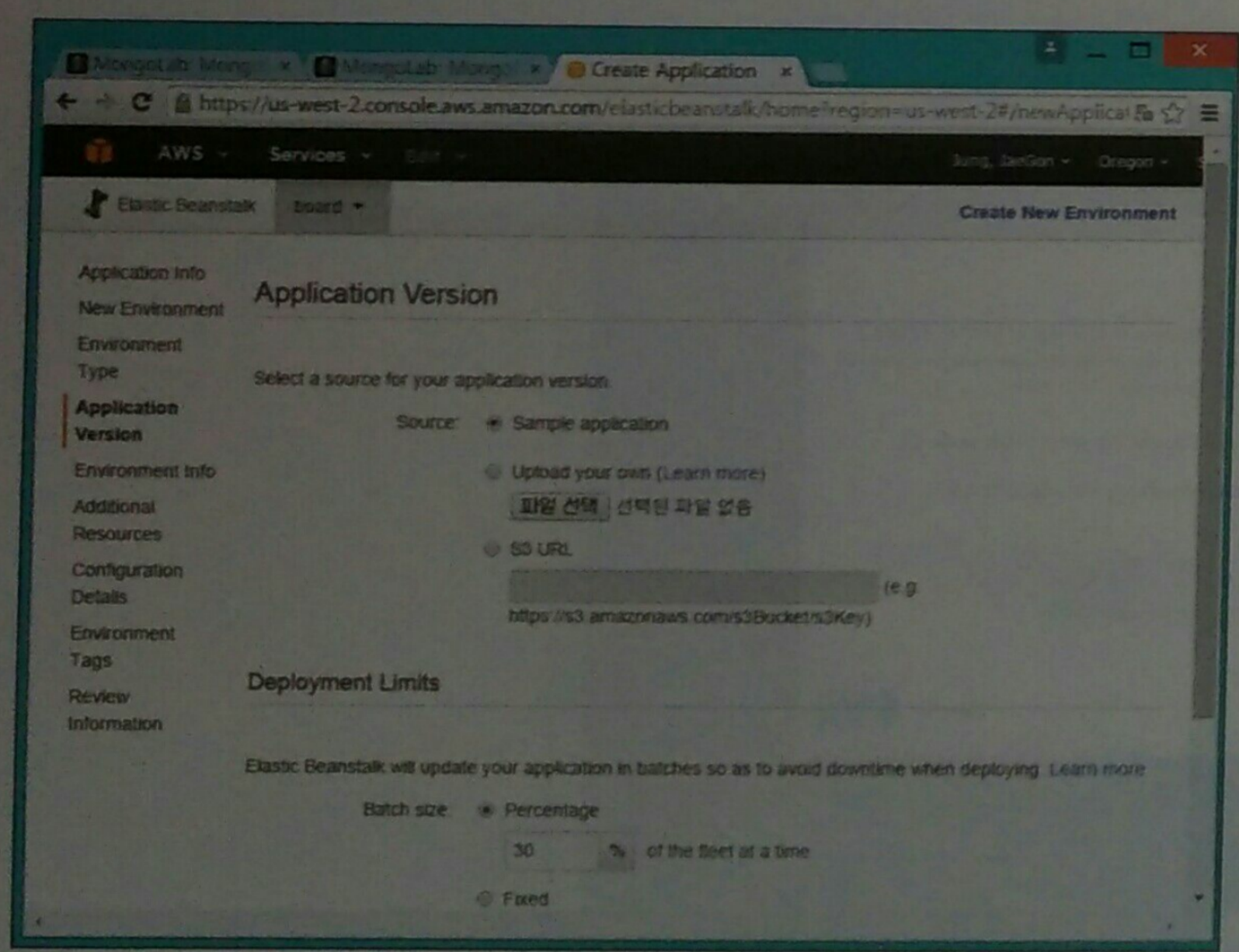
▲ Web server의 권한을 설정하는 화면

[Next] 버튼을 누르면 환경 유형(Environment Type)을 선택하는 화면이 나타납니다.



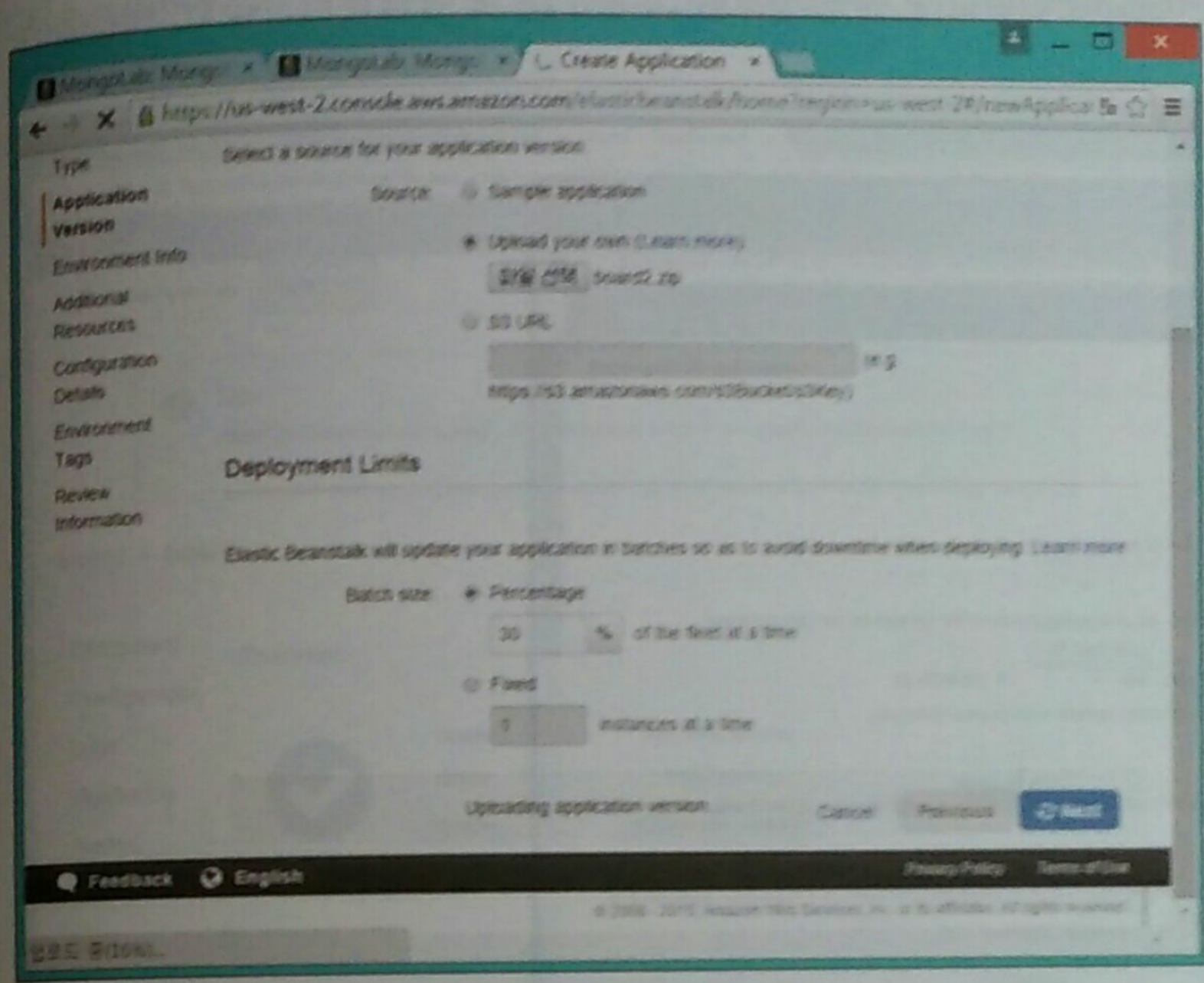
▲ 환경 유형을 선택하는 화면

Node.js를 선택한 후 [Next] 버튼을 누르면 애플리케이션 버전을 지정하는 화면으로 전환됩니다. 이 화면에서 소스를 업로드할 수 있습니다.



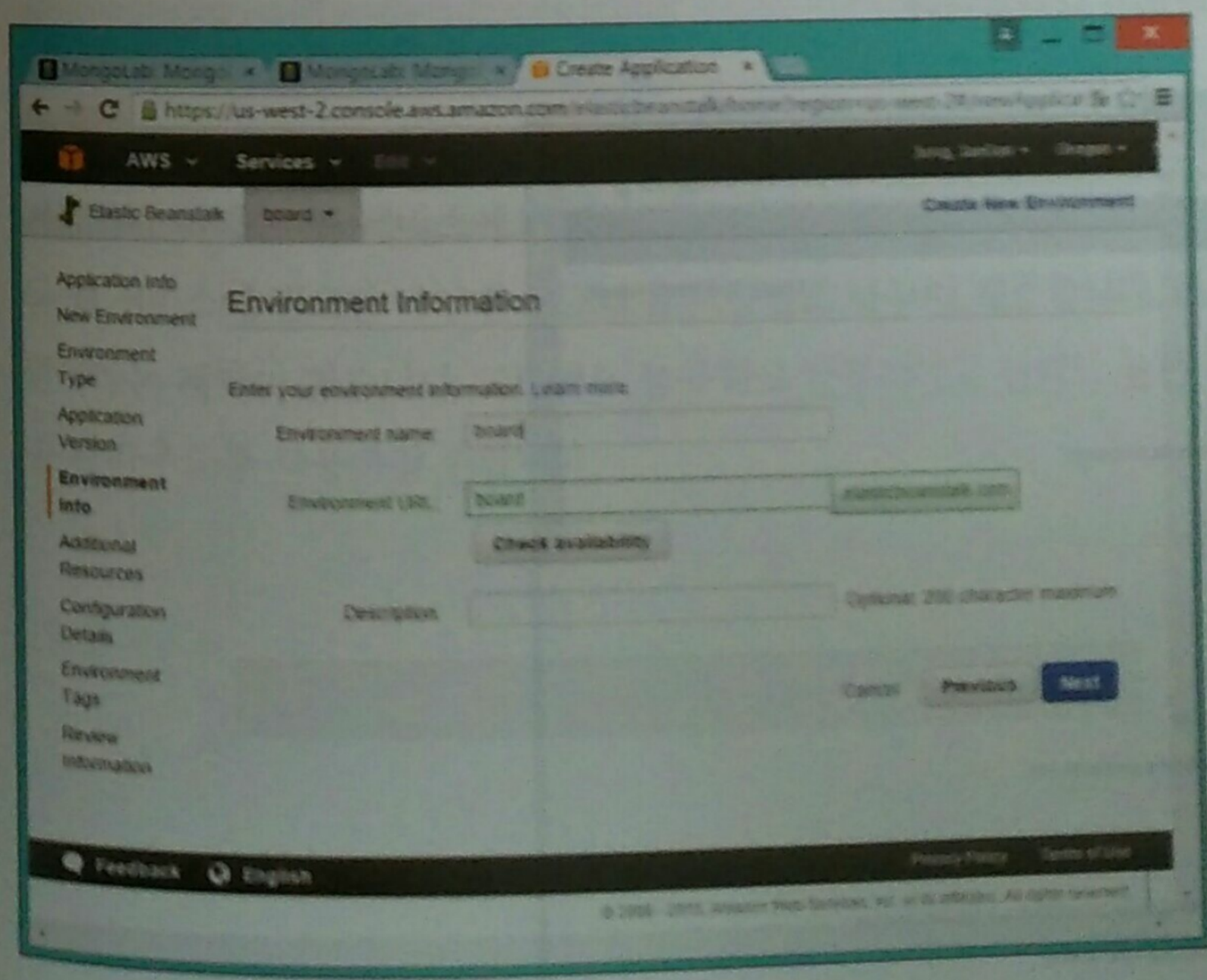
▲ 애플리케이션 버전을 지정하는 화면

Source 항목은 Upload your own을 선택한 후 앞에서 압축해 둔 board2.zip 파일을 파일 탐색기로 찾아 선택합니다. 그런 다음 [Next] 버튼을 클릭하면 파일 업로드가 시작됩니다.



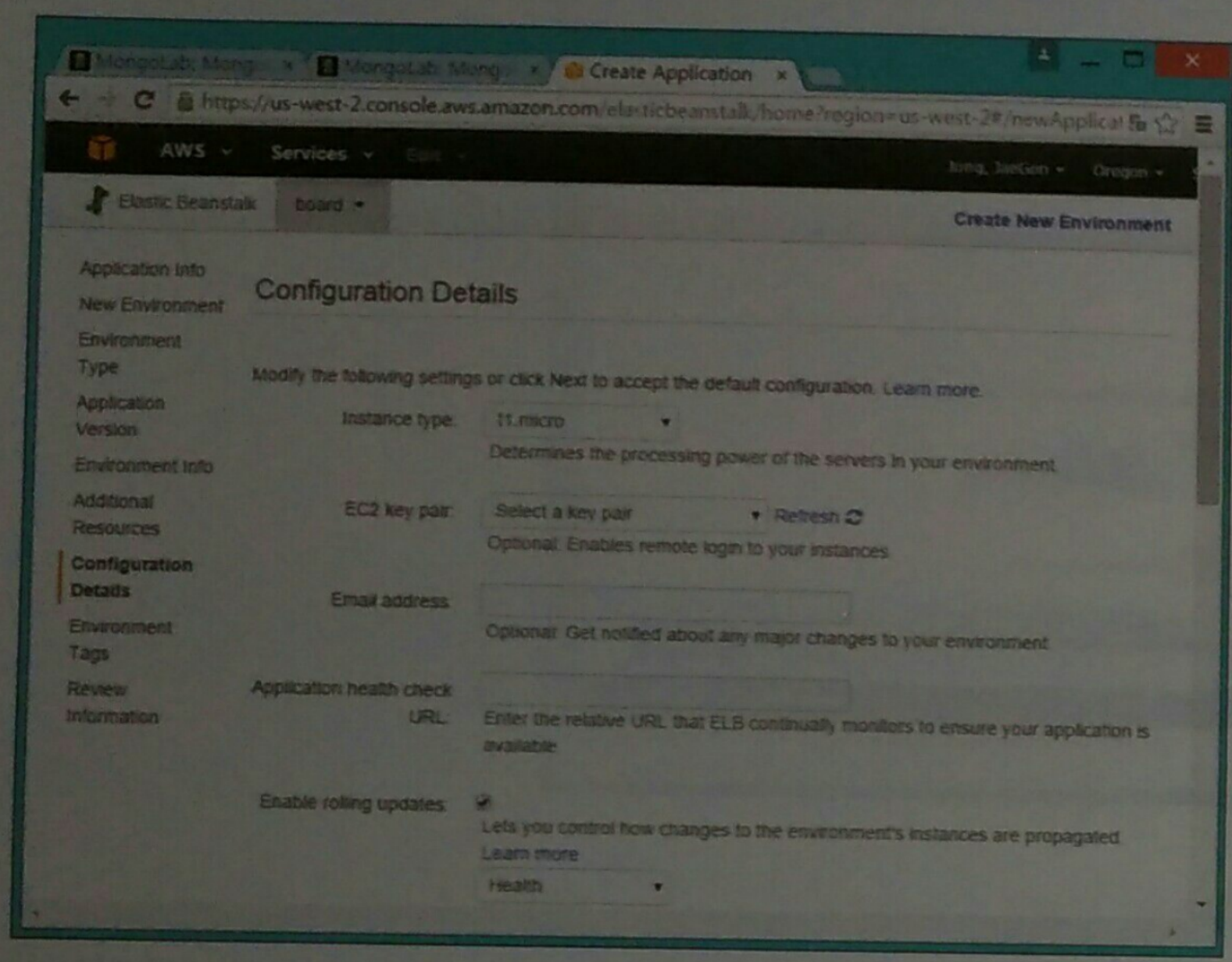
▲ 파일 업로드가 진행 중인 화면

업로드가 끝나면 환경 정보(Environment Information)를 입력하는 화면이 나타납니다. 두 개의 입력 상자에 모두 board를 입력합니다. [Check availability] 버튼을 눌러 입력한 정보를 사용할 수 있는지 확인합니다. 만약 입력한 정보를 사용할 수 없다면 다른 이름으로 입력합니다.



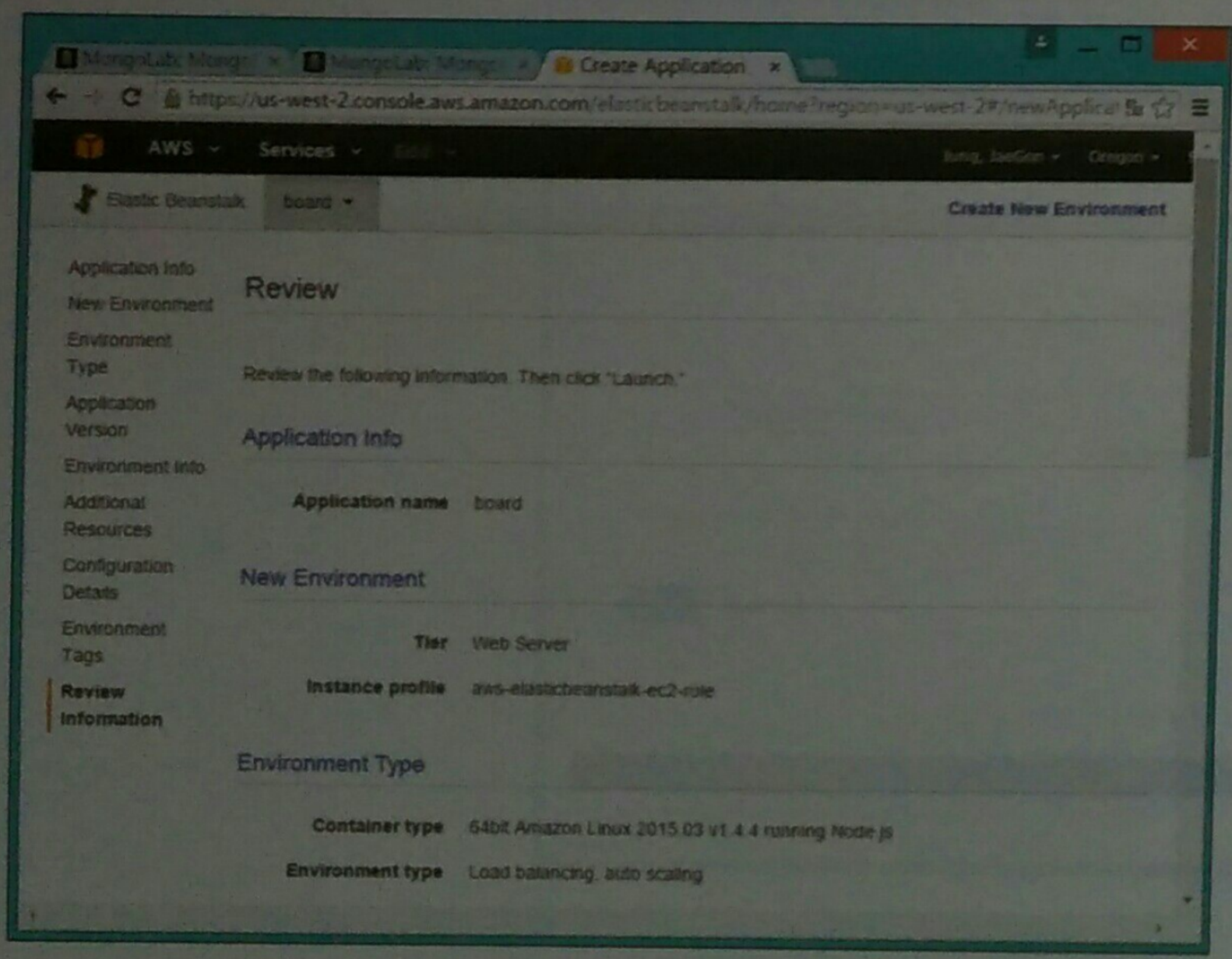
▲ 환경 정보 입력 화면

[Next] 버튼을 누르면 **추가 리소스(Additional Resources)**를 묻는 화면이 나타나는데 [Next] 버튼을 눌러 넘어갑니다. 그다음에는 **설정 상세(Configuration Details)** 화면이 나타나는데 여기에서는 Instance type이 t1.micro로 선택되어 있습니다. 설정 정보를 그대로 둔 채 아래쪽의 [Next] 버튼을 누릅니다.



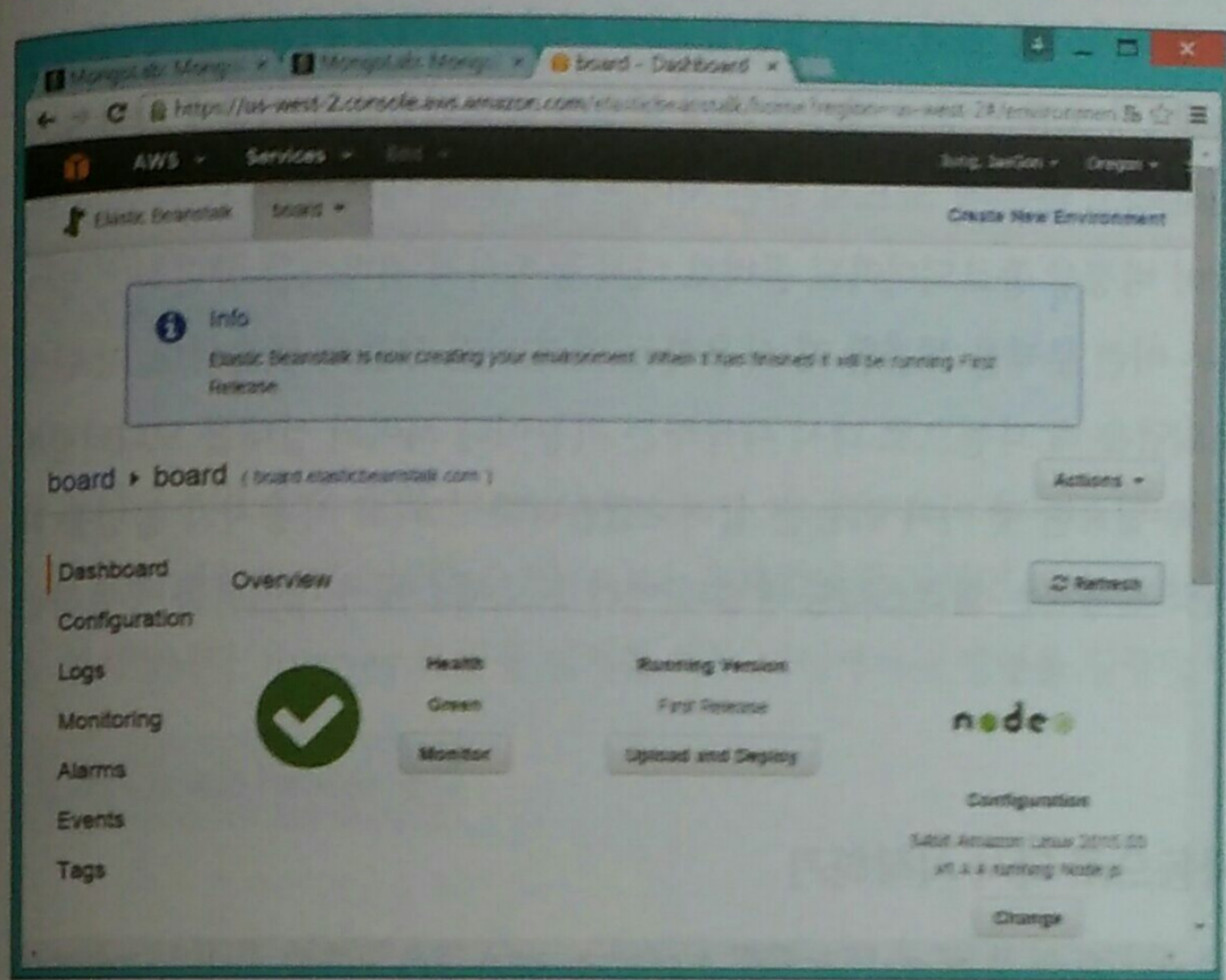
▲ 설정 상세 화면

다음 화면에 나타난 Environment Tags 정보도 그대로 둔 채 [Next] 버튼을 누르면 마지막으로 설정 정보를 확인할 수 있는 Review 화면이 나타납니다.



▲ 마지막 Review 화면

아래쪽에 있는 [Launch] 버튼을 클릭하면 애플리케이션이 시작됩니다. 이 화면으로 넘어가는 데 몇 분 정도 걸릴 수 있습니다. 만약 오류가 생겼거나 프로젝트 파일을 업데이트하고 싶다면 AWS 콘솔 화면에서 board 애플리케이션을 선택한 후 [Upload and Deploy] 버튼을 누르면 됩니다.



▲ 애플리케이션을 실행한 후의 AWS 콘솔 화면

이제 전 세계 사람들이 다음 주소로 접근할 수 있도록 아마존 클라우드에 여러분이 만든 서비스를 올렸습니다.

▶ http://board.elasticbeanstalk.com

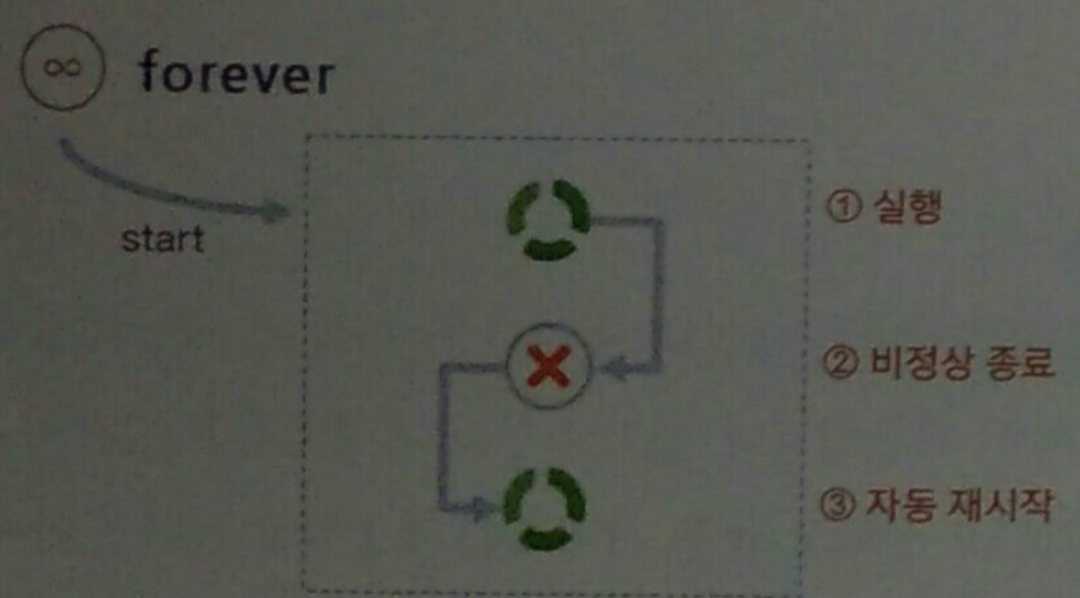
지금까지 아마존 웹 서비스에 배포하는 방법을 알아보았습니다. 아마존 클라우드는 웹 서비스, 저장소 등 웹 서비스에 필요한 다양한 인프라를 기능별로 나누어 제공합니다. 하지만 무료 계정의 유효 기간이 1년이라는 제약이 있습니다. 그럼에도 불구하고 안정적이고 신뢰도 높은 서비스를 제공하고 싶다면 아마존 클라우드를 사용하길 권합니다.

15-5 | 배포에 필요한 기능 알아보기

여러분이 만든 노드 프로그램을 클라우드에 올렸으니 이제 전 세계 사용자들에게 서비스할 수 있습니다. 그런데 만약 실행 중이던 웹 서버에 오류가 생겨 예기치 않게 비정상 종료되었다면 어떻게 해야 할까요? 이런 상황을 보통 **장애(Failure)**라고 하는데, 노드로 만든 서버 프로그램은 대체로 24시간 서비스하는 경우가 많아서 만약 프로그램이 비정상 종료되더라도 곧바로 다시 동작시켜 서비스를 중단하는 일이 없도록 해주어야 합니다. 노드에는 이런 문제를 해결할 때 사용하는 여러 가지 모듈이 있습니다. 이 단락은 노드 프로그램이 비정상 종료되었을 때 자동으로 다시 시작하는 기능이나 서버의 상태를 모니터링하는 기능 등 전 세계로 배포할 때 꼭 필요한 몇 가지 기능을 살펴보겠습니다. 그리고 사용자가 늘었을 때 서버를 여러 개의 프로세스로 띄워 실행하는 **클러스터링** 방법과 여러 프로세스에서 데이터를 공유하는 방법도 알아보겠습니다.

비정상으로 종료되었을 때 자동으로 다시 시작하기

실행 중이던 노드 프로그램은 예기치 못한 상황에서 비정상 종료될 수 있습니다. 이때는 서비스가 중단되므로 어떤 클라이언트의 요청도 받을 수 없습니다. 노드는 이런 장애를 해결하는 몇 가지 모듈이 있는데 그중에서 가장 많이 사용하는 것이 forever 모듈입니다. 다음 그림을 보면 forever 모듈의 역할을 쉽게 이해할 수 있습니다.



◀ forever 모듈의 역할

forever 모듈로 노드 프로그램을 실행하면 비정상 종료되었을 때 자동으로 다시 시작합니다. 이 기능은 forever 모듈이 실행된 프로그램의 상태를 계속 감시하기 때문에 가능합니다. forever 모듈로 노드 프로그램을 실행하는 방법을 알아보기 위해 먼저 DeployExample 프로젝트를 만듭니다. [File → New → Node.js Project] 메뉴를 누르고 새로운 대화상자가 보이면 프로젝트 이름을 작성하는 입력 상자에 DeployExample을 입력합니다. 새로운 프로젝트가 만들어지면 5장에서 만든 간단한 익스프레스 서버 소스를 복사하여 사용합니다. ExpressExample 프로젝트 안에 있던 app8.js 파일을 복사해서 붙여 넣은 파일 이름을 app.js로 변경합니다. [public] 폴더도 새로 만들고 ExpressExample 프로젝트의

[public] 폴더 안에 있던 login2.html 파일을 복사해서 옮겨 놓습니다. 그리고 이름을 login.html로 변경합니다.

이렇게 복사한 새 파일이 정상으로 실행되는지 확인하기 위해 명령 프롬프트를 열고 다음과 같이 두 가지 모듈을 설치한 후 app.js 파일을 실행합니다.

```
% npm install express --save
% npm install body-parser --save
% node app.js
```

웹 브라우저의 주소 창에 http://localhost:3000/public/login.html을 입력하여 복사한 코드가 정상으로 동작하는지 테스트할 수 있습니다. 문제없이 동작하면 제대로 복사한 것입니다. 이제 forever 모듈을 설치합니다. forever 모듈을 설치할 때는 어디에서나 명령을 실행할 수 있도록 -g 옵션을 추가합니다.

```
% npm install forever -g
```

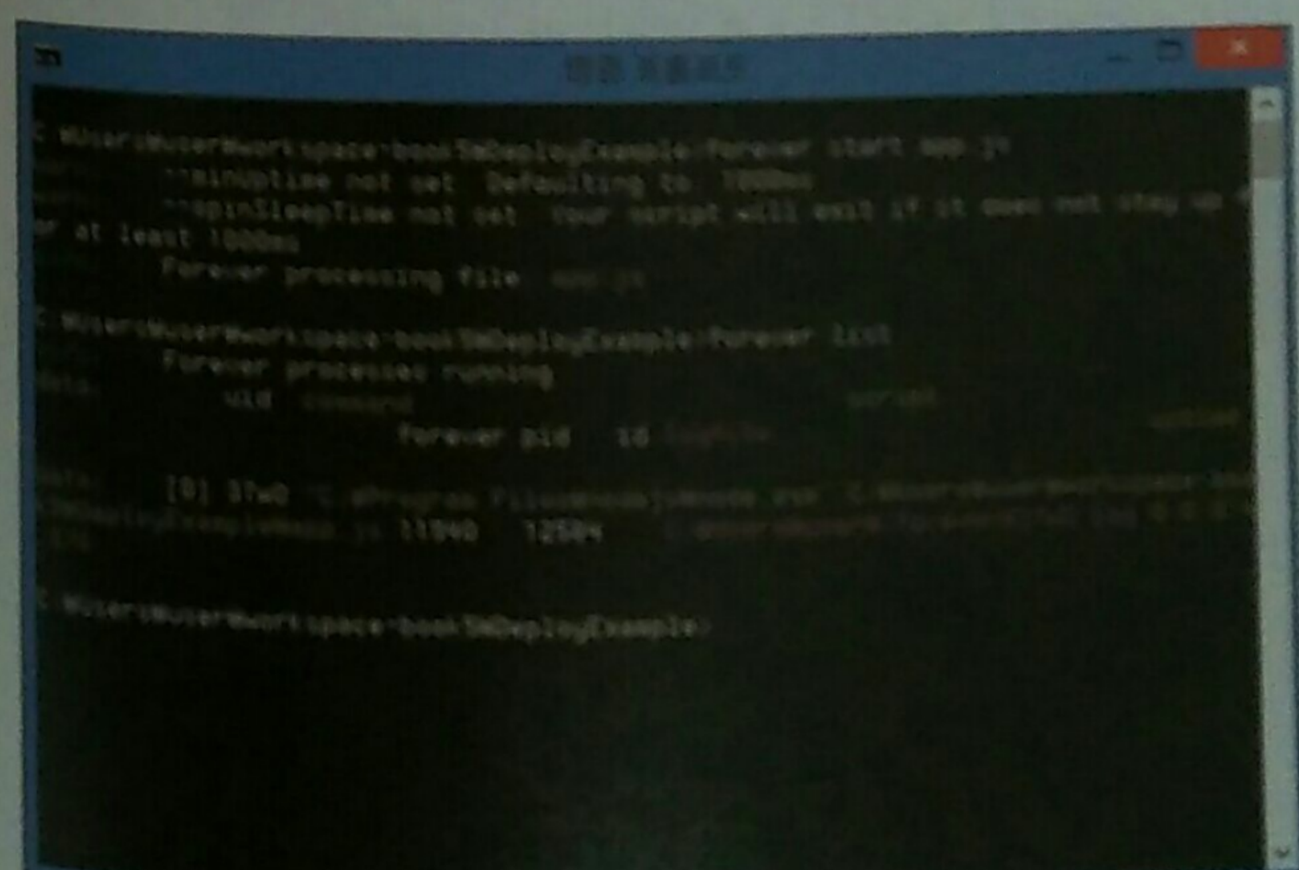
forever 모듈을 설치했으면 forever start 명령으로 노드 프로그램을 실행할 수 있습니다.

```
% forever start app.js
```

forever 모듈로 노드 프로그램을 실행하면 간단한 메시지만 출력됩니다. 이때 프로그램 내부에서는 app.js 파일을 하나의 프로세스로 실행합니다. 프로그램의 실행 상태를 보여주지는 않기 때문에 어떤 프로그램이 실행되었는지 확인하려면 forever list 명령을 사용합니다.

```
% forever list
```

그러면 다음과 같이 실행 프로세스 목록이 표시됩니다.



◀ forever 모듈로 노드 프로그램을 실행하고 그 목록을 확인한 것은

하나의 프로세스가 실행 중인 것을 확인할 수 있습니다. 웹 브라우저에서 login.html 파일을 열고 테스트해 보면 웹 서비스도 정상인 것을 알 수 있습니다. node 명령이 아니라 forever 명령으로 실행했으므로 지금부터는 app.js 프로그램이 비정상 종료되더라도 자동으로 다시 시작됩니다. 만약 소스 코드를 수정했다면 어떻게 해야 할까요? 소스를 수정했을 때는 forever restart 명령으로 다시 시작하면 됩니다. forever restart로 노드 프로그램을 다시 시작하면 소스 코드에서 수정한 기능이 반영됩니다.

```
% forever restart app.js
```

정박사의 한마디 supervisor 모듈을 사용하면 소스가 바뀔 때마다 자동으로 다시 시작하던데요?

노드에서 사용할 수 있는 모듈 중에서 supervisor 모듈을 사용하면 소스가 변경되었을 때 자동으로 다시 시작하도록 만들 수 있어 편리하다고 생각할 수 있습니다. 하지만 소스 파일을 잘못 바꿨을 때 실행 중인 프로그램도 오동작하기 때문에 restart 명령을 직접 입력하는 것이 안전합니다. 따라서 배포된 프로그램이 실행 중일 때 업데이트하려면 restart 명령을 사용하는 것이 좋습니다. 물론 개발자가 주의해서 사용한다면 supervisor 모듈도 아주 좋은 배포 도구입니다.

프로그램을 중지하고 싶다면 forever stop 명령을 사용합니다.

```
% forever stop app.js
```

대표적인 forever 명령을 정리하면 다음과 같습니다.

명령	설명
forever start	노드 프로그램을 실행합니다. 예) forever start app.js
forever list	실행 중인 노드 프로그램 목록을 보여 줍니다. 예) forever list
forever restart	실행 중인 노드 프로그램을 다시 시작합니다. 변경된 소스를 반영할 때 사용합니다. 예) forever restart app.js
forever stop	실행 중인 노드 프로그램을 중지합니다. 예) forever stop app.js

[표] 대표적인 forever 명령

클러스터링 사용하기

클러스터링은 하나의 프로그램을 여러 개의 프로세스로 실행하는 방법입니다. pm2 모듈을 사용하면 원하는 개수만큼 프로세스를 실행할 수 있고, 필요한 경우에 메모리 같은 시스템 리소스를 얼마나 사용하는지 간단하게 모니터링할 수도 있습니다. 먼저 pm2 모듈을 설치합니다.

```
% npm install pm2 -g
```

pm2를 사용하는 방법은 forever와 거의 같습니다. 대표적인 pm2 명령은 다음과 같습니다.

명령	설명
pm2 start	노드 프로그램을 실행합니다. pm2는 프로그램을 이름으로 구별할 수 있어서 --name 옵션을 사용해 프로그램에 이름을 부여할 수도 있습니다. --name 옵션을 주지 않으면 확장자를 제외한 파일 이름이 프로그램 이름이 됩니다. 예) pm2 start app.js --name web1
pm2 list	실행 중인 노드 프로그램의 목록을 보여 줍니다. 예) pm2 list
pm2 show	실행 중인 노드 프로그램의 id나 name을 지정하여 해당 정보를 보여 줍니다. 예) pm2 show web1
pm2 restart	실행 중인 노드 프로그램을 다시 시작합니다. 변경된 소스를 반영할 때 사용합니다. 예) pm2 restart web1
pm2 stop	실행 중인 노드 프로그램을 중지합니다. 예) pm2 stop web1
pm2 delete	노드 프로그램을 목록에서 삭제합니다. 만약 프로그램이 실행 중이라면 프로그램을 중지한 후 삭제합니다. 예) pm2 delete web1

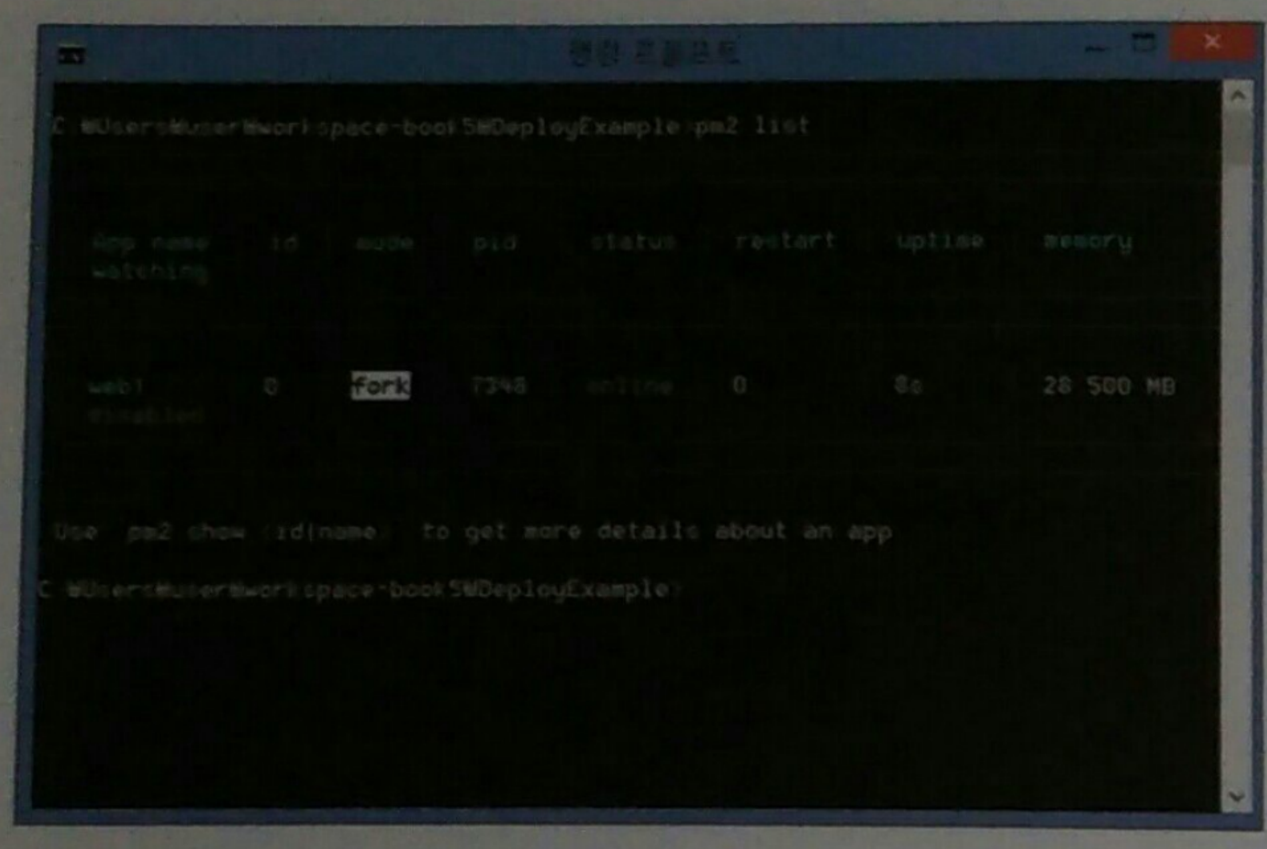
[표] 대표적인 pm2 명령

그러면 pm2 start 명령으로 app.js 파일을 실행해 보겠습니다.

```
% pm2 start app.js --name web1
```

app.js 파일을 실행하면서 web1이라는 프로그램 이름을 부여했습니다. pm2 list 명령을 실행하면 다음처럼 프로그램 목록이 표시됩니다.

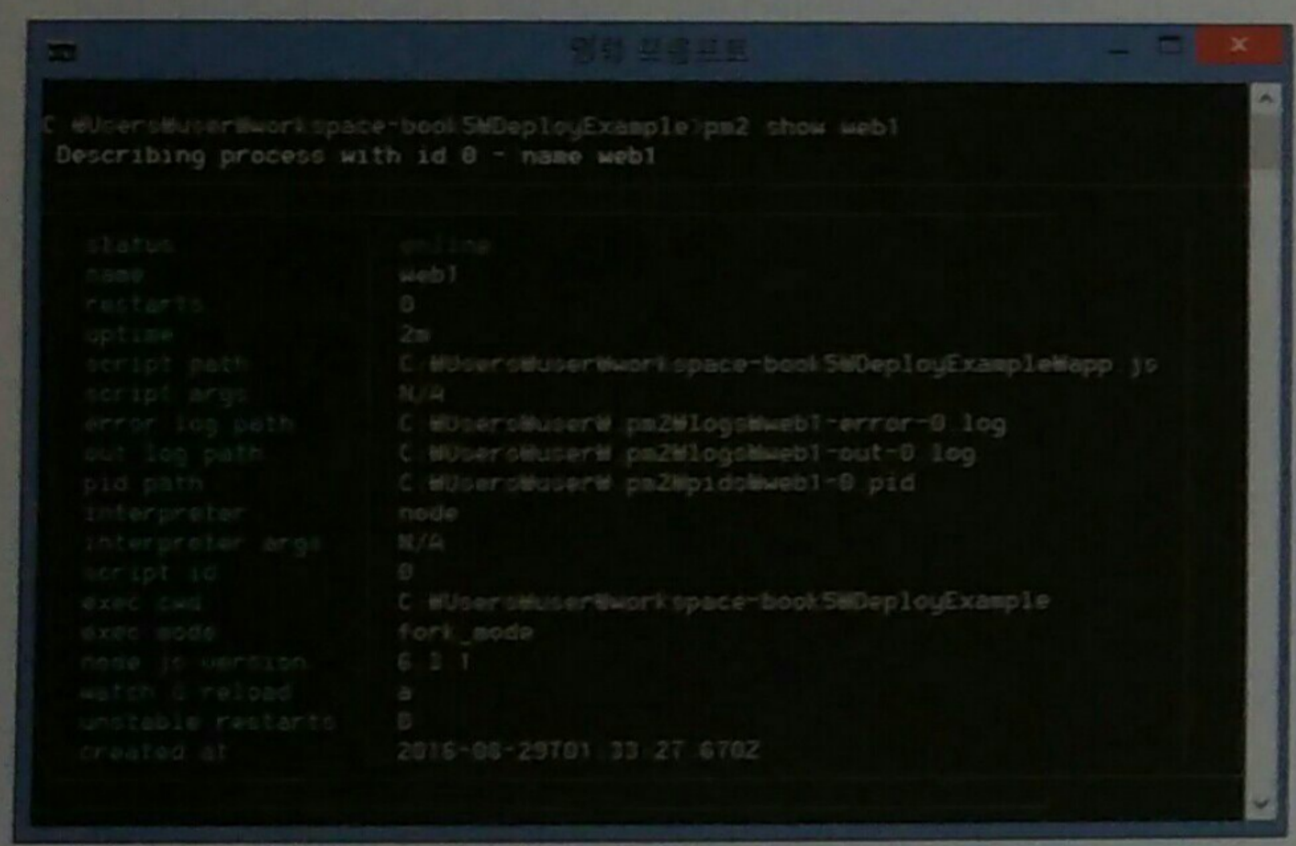
```
% pm2 list
```



▲ pm2로 프로그램 목록을 확인한 경우

pm2 show 명령을 사용하면서 프로그램 이름을 지정하면 좀 더 상세한 정보를 확인할 수 있습니다.

```
% pm2 show web1
```



▲ pm2 show 명령으로 프로그램의 상세 정보를 확인한 경우

표시된 정보가 forever보다 더 깔끔하지 않나요? 하지만 큰 차이는 느껴지지 않습니다. 우리는 이런 일반적인 기능보다 pm2의 클러스터링(Clustering) 기능에 더 관심이 많습니다. 왜냐하면 클러스터링 기능은 성능을 끌어올리거나 클라이언트의 요청을 분산시키는 데 큰 효과가 있기 때문입니다. 예를 들어, 여러분이 사용하는 노트북에 쿼드코어(Quad-core) CPU가 탑재되어 있더라도 노드 프로그램을 실행시키면 그중 하나의 CPU만 사용하는데, 각각의 CPU마다 여러분의 프로그램을 실행할 수 있다면 성능은 훨씬 좋아집니다. pm2로 노드 프로그램을 실행할 때 -i 옵션을 붙이면 클러스터 모드로 실행됩니다. 그리고 -i 뒤에 숫자를 넣으면 클러스터 모드에서 실행할 인스턴스의 개수가 결정됩니다. 만약 0으로 지정했다면 현재 사용하고 있는 장비에서 사용할 수 있는 CPU 개수만큼 인스턴스를 실행합니다.

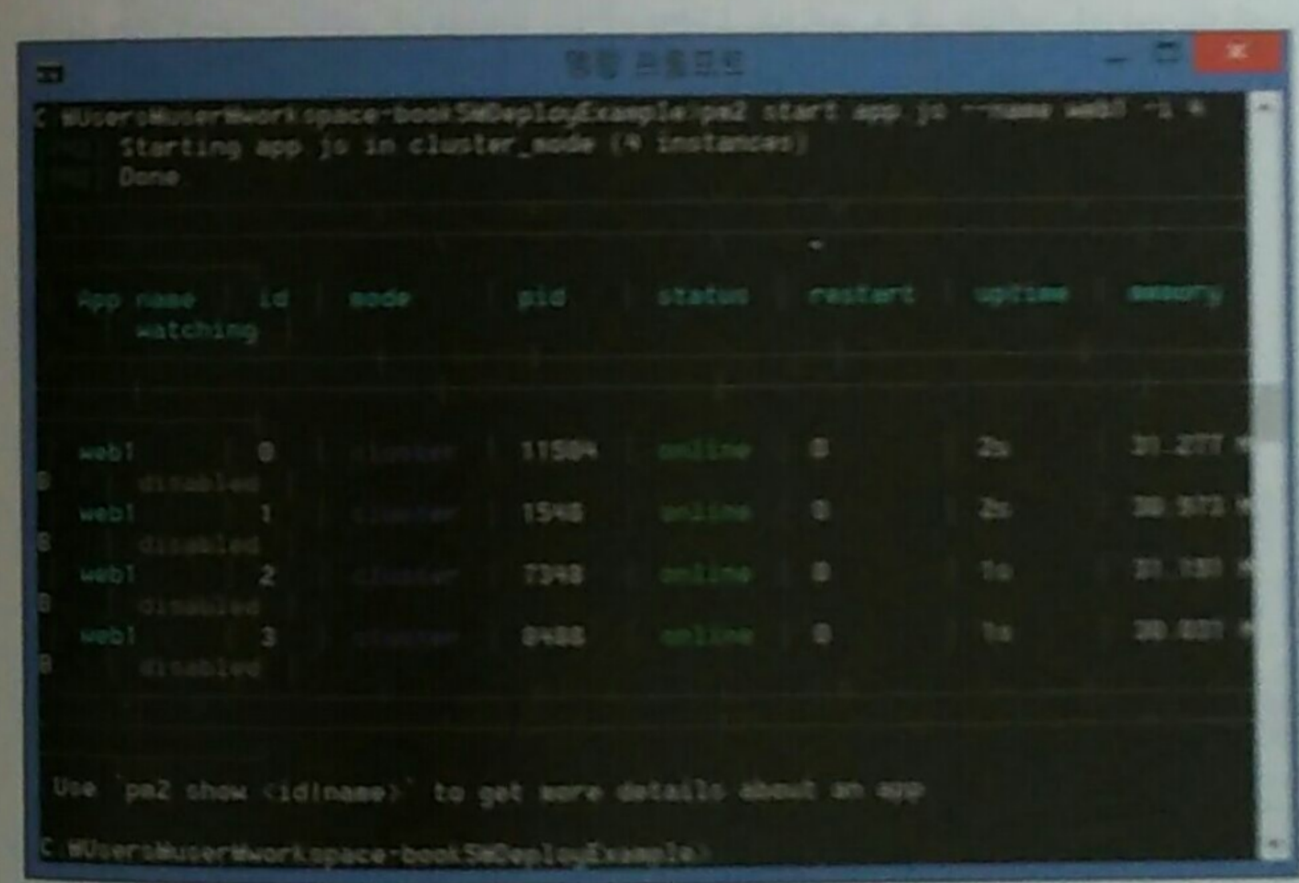
정박사의 한마디 프로세스와 인스턴스는 무엇이 다를까요?

프로세스(Process)는 프로그램이 실행될 때의 단위라고 이해하면 쉽습니다. 예를 들어, 프로그램을 두 번 실행하면 두 개의 프로세스가 실행됩니다. 인스턴스(Instance)는 클러스터 모드에서 하나의 프로그램을 실행하기 위해 생성한 하나의 프로세스를 의미하는 단어입니다. 따라서 여기에서 사용하고 있는 인스턴스는 프로세스와 같은 말입니다.

pm2 delete 명령을 사용하면 기존에 실행했던 프로그램을 pm2 목록에서 삭제할 수 있습니다. pm2 delete 명령으로 기존 프로세스를 삭제한 후 클러스터 모드에서 4개의 인스턴스를 실행하도록 다음 명령을 입력합니다.

```
% pm2 delete web1
% pm2 start app.js -i 4 --name web1
```

인스턴스마다 명령 프롬프트 창이 만들어지면서 실행됩니다. 따라서 네 개의 명령 프롬프트 창이 뜨면서 실행되고 pm2 목록에는 mode 칼럼의 값이 cluster로 표시됩니다.



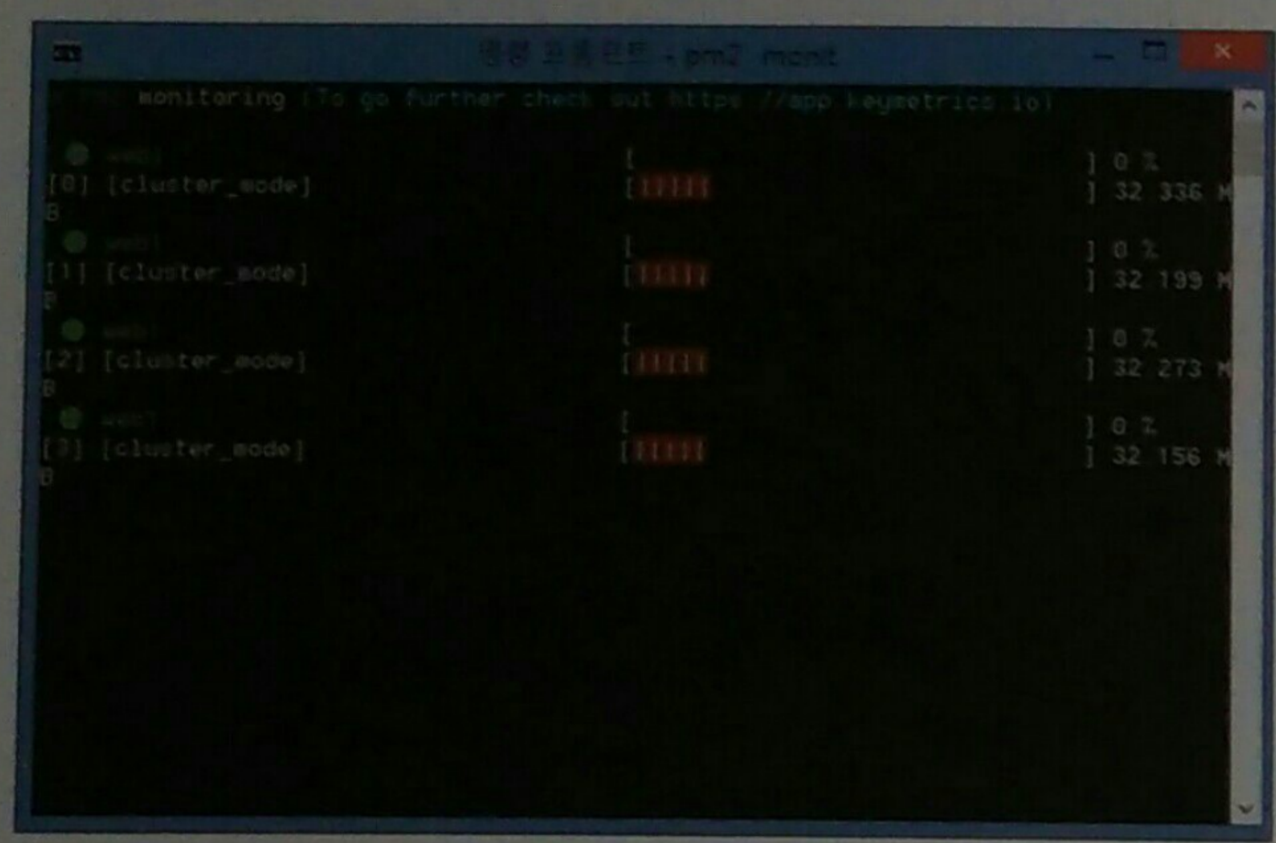
▲ 클러스터 모드로 실행하여 4개의 인스턴스가 만들어지도록 한 경우

정박사의 한마디 클러스터 모드를 사용할 때 서버를 다시 시작하려면 pm2 reload 명령을 쓰세요

클러스터 모드를 사용할 때는 pm2 restart 명령 대신 pm2 reload 명령을 사용하는 것이 좋습니다. 왜냐하면 pm2 reload 명령은 기존 서버가 종료되고 새로 시작될 때까지의 시간 차를 없애 다운타임(down-time) 없이 서버를 다시 시작할 수 있기 때문입니다.

pm2에는 각 프로세스별로 얼마만큼의 CPU와 메모리를 사용하는지 확인할 수 있는 명령도 있습니다. 다음과 같이 pm2 monit 명령을 실행합니다. 그러면 인스턴스마다 CPU와 메모리 사용량이 표시됩니다.

```
% pm2 monit
```

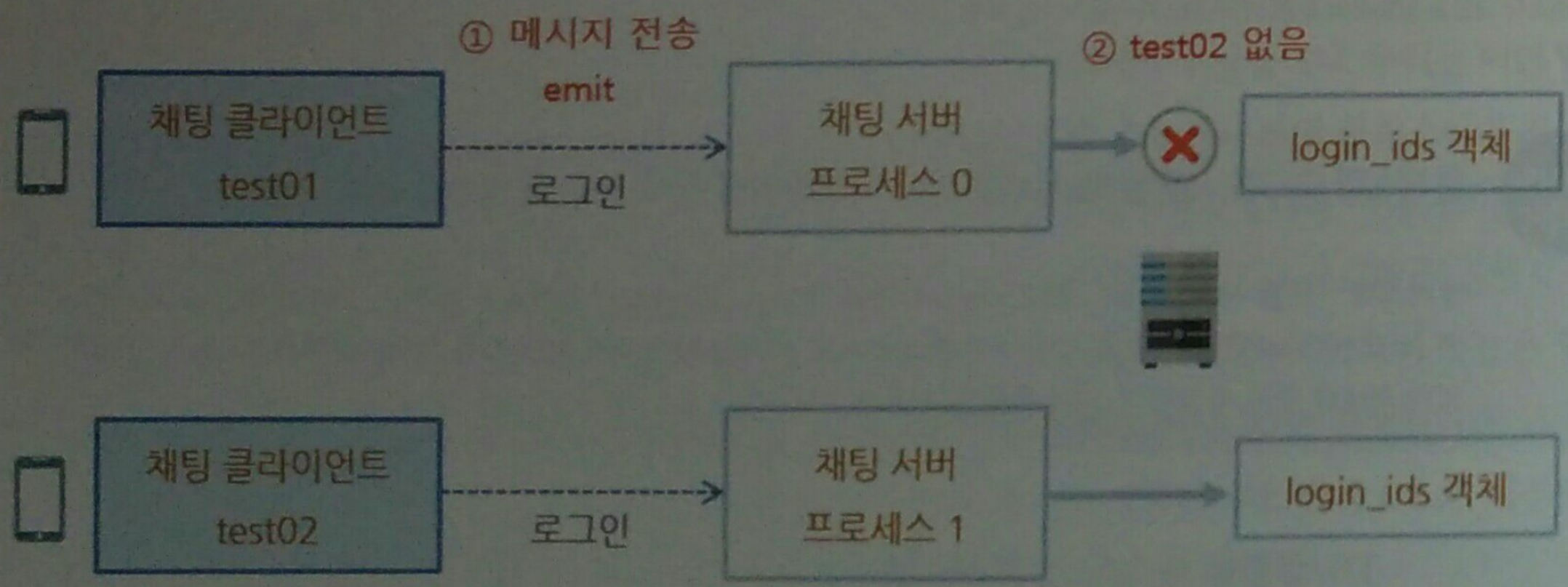


▲ pm2 monit 명령으로 CPU와 메모리 사용량을 확인한 경우

지금까지 클러스터링 기능으로 하나의 프로그램을 여러 프로세스에서 실행하는 방법을 알아보았습니다.

레디스 사용하기

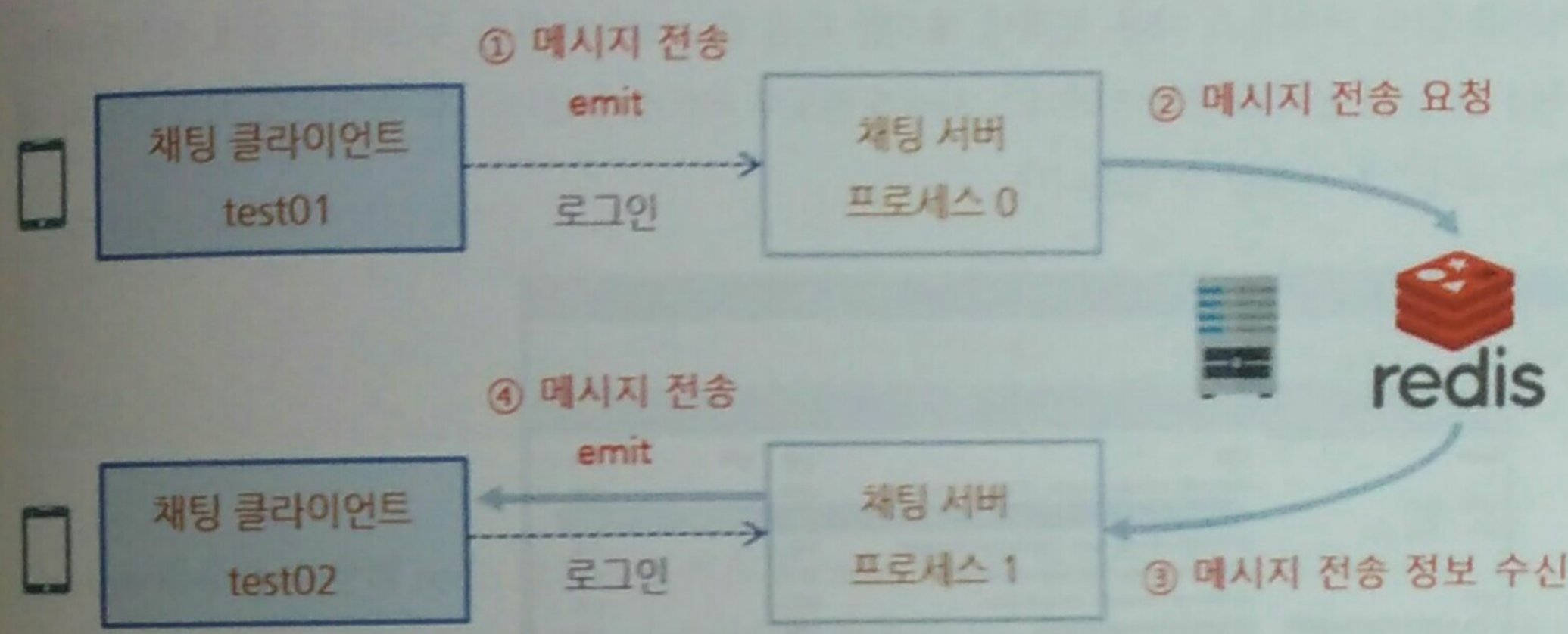
하나의 프로그램을 여러 개의 프로세스로 동시에 실행하면 문제가 생기기도 합니다. 이런 문제는 채팅 서버처럼 클라이언트의 상태를 공유해야 하는 경우에 발생합니다. 예를 들어, 채팅 클라이언트 A는 프로세스 0번으로 접속하고 채팅 클라이언트 B는 프로세스 1번으로 접속했을 때 두 개의 채팅 클라이언트는 서로 채팅 메시지를 전송할 수 없습니다. 왜냐하면 프로세스는 독립적으로 실행되어 변수의 값을 서로 공유하지 못하기 때문입니다.



▲ 채팅 서버에 로그인한 정보를 프로세스끼리 서로 공유하지 못하는 문제

10장에서 채팅 서버를 구성할 때 로그인한 사용자 정보는 login_ids 객체에 넣어 두었습니다. 그런데 이 객체는 각각의 프로세스가 실행될 때마다 해당 프로세스 안에만 존재합니다. 따라서 첫 번째 채팅 클라이언트에서 메시지 전송을 요청했을 때 프로세스 0으로 실행된 채팅 서버에서는 두 번째 채팅 클라이언트의 정보를 찾을 수 없습니다.

이럴 때 통신이 가능한 메모리 데이터베이스인 **레디스(Redis)**를 사용합니다. 어떤 프로그램이 레디스 서버에 연결하여 데이터를 넣어 두면 다른 프로그램이 레디스 서버에 연결하여 그 데이터를 가져올 수 있습니다. 또한 어떤 프로그램이 레디스 서버 쪽에 데이터를 넘겨주면 그 데이터를 받겠다고 미리 등록한 다른 프로그램 쪽에 데이터가 자동으로 넘어가도록 만들 수도 있습니다.



▲ 레디스를 이용한 프로세스 간 데이터 전송

레디스로 이런 데이터 전송 구조를 만들어 사용하는 것이 복잡할 수도 있습니다. 따라서 socket.io 모듈에는 레디스 연동 기능을 미리 만들어 놓았습니다. 다시 말해, 10장에서 만든 채팅 서버의 코드를 일부만 수정하면 레디스를 사용하는 방식으로 변경할 수 있습니다. 그럼 레디스를 어떻게 사용하는지 간단하게 알아본 후 채팅 서버의 코드를 레디스와 연동하는 샘플 코드를 살펴보겠습니다.

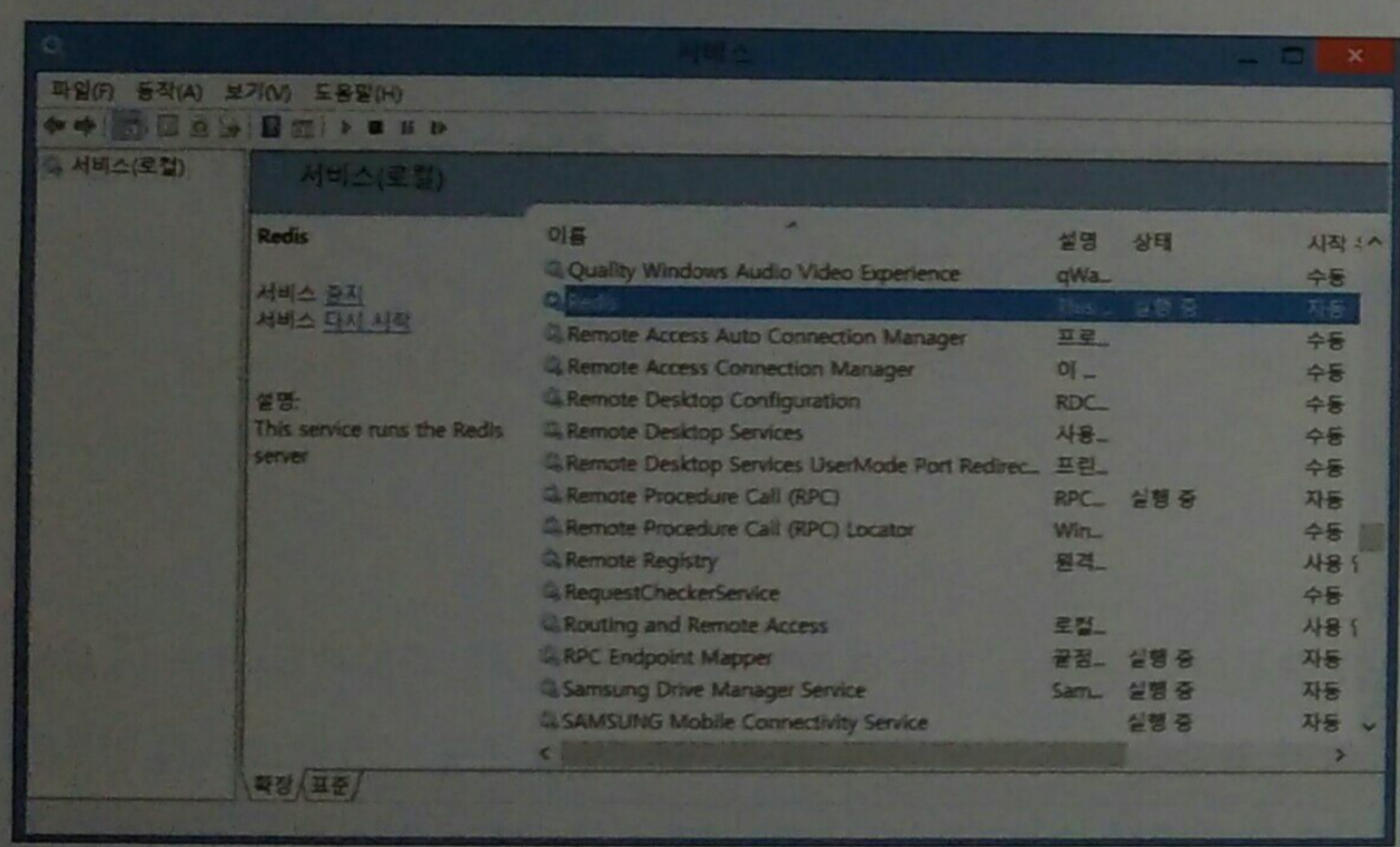
레디스는 레디스 서버 프로그램이 동작하고 있어야 다른 프로그램들이 레디스 서버로 연결하여 사용할 수 있습니다. 따라서 레디스를 사용하려면 먼저 프로그램을 설치하고 레디스 서버를 실행해야 합니다. 레디스 사이트는 http://redis.io 주소로 접속하면 되지만 이 사이트에서는 윈도우 버전을 제공하지 않습니다. 윈도우 버전을 다운로드할 수 있도록 별도로 만든 다음 사이트에서 프로그램을 다운로드하여 설치합니다.

▶ https://github.com/MSOpenTech/redis/releases

해당 사이트의 Downloads 항목에서 msi 확장자를 가진 레디스 설치 파일을 다운로드한 후 실행하면 다음과 같은 설치 시작 화면이 나타납니다.

◀ 레디스 윈도우 버전의 설치 시작 화면

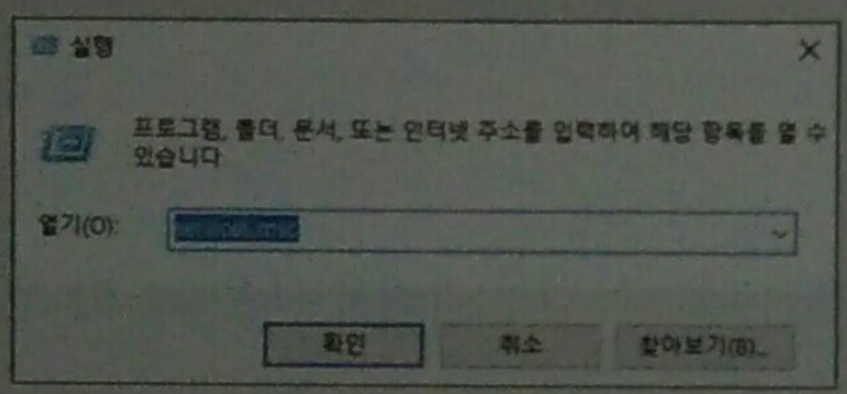
[Next] 버튼을 눌러 라이선스 사용 동의에 체크한 다음 [Install] 버튼을 누르면 손쉽게 설치됩니다. [Finish] 버튼을 눌러 설치를 끝내면 레디스 서버가 **윈도우 서비스**에 설치된 후 자동으로 시작되었기 때문에 곧바로 레디스를 사용할 수 있습니다.



▲ 레디스 서버가 시작된 상태를 알 수 있는 윈도우 서비스 창

정박사의 한마디 윈도우 서비스 창을 열어 자동으로 시작되었는지 확인해 보세요

레디스를 설치한 후 실행되었는지 확인하고 싶다면 윈도우 서비스 창을 열어 보면 됩니다. [윈도우 시작] 버튼을 마우스 오른쪽 버튼으로 클릭해서 [실행(R)] 메뉴를 선택하세요. [실행] 대화상자가 나타나면 열기(O) 항목에 services.msc를 입력하고 [확인] 버튼을 클릭합니다. 그럼 윈도우 서비스 창을 불러올 수 있습니다. 여기에서 스크롤을 내려 설치한 레디스를 찾으면 됩니다.



노드에서 레디스에 연결하려면 redis 모듈이 있어야 합니다. 다음 명령을 사용해서 redis 모듈을 설치합니다.

```
% npm install redis --save
```

서버 코드는 app.js 코드를 수정해서 사용할 것입니다. DeployExample 프로젝트에 있는 app.js 파일을 복사하여 app2.js 파일을 만든 후 다음처럼 코드를 추가합니다.

참조 파일 | DeployExample>app2.js

```
.....

// Redis 사용하기
var redis = require("redis");
var store = redis.createClient();

.....
```

require 함수를 사용해 redis 모듈을 불러온 후 redis 변수에 할당합니다. 그리고 createClient 함수를 호출하면 레디스 클라이언트 객체가 만들어진 후 반환됩니다. 이 클라이언트 객체는 store 변수에 할당합니다. 레디스 클라이언트 객체에는 여러 가지 함수가 정의되어 있습니다. 그중에서 가장 쉽게 사용할 수 있는 두 가지는 hset과 hget입니다. 이 함수들은 해시 테이블에 데이터를 넣거나 빼는 것과 비슷하므로 사용하기 쉽습니다.

명령	설명
hset(key, field, value, print)	key를 기준으로 속성을 설정합니다. 만약 기존 속성이 없다면 새로 만들어지고, 기존 속성이 있다면 기존 속성을 덮어씁니다. 예) hset('user', user_id, user_name, redis.print)
hget(key, field, callback)	key를 기준으로 속성을 가져옵니다. 조회한 속성은 콜백 함수로 전달합니다. 예) hget('user', user_id, function(err, user_name) { ... });

[표] hset과 hget 함수

hset과 hget 기능을 사용하기 위해 웹 페이지를 두 개 만듭니다. 하나는 사용자 아이디와 이름을 서버로 보내 레디스에 추가하고 또 다른 하나는 사용자 아이디를 보내 레디스에서 사용자 이름을 조회할 것입니다. 웹 페이지를 만들기 전에 먼저 app2.js 파일에서 기존 /process/login 라우팅 함수 부분을 삭제하고 다음 코드처럼 /process/setname 라우팅 함수를 추가합니다.

참조 파일 | DeployExample>app2.js

```
……

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

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

    store.hset("user", paramId, paramName, redis.print);
    console.log('redis에 사용자를 등록했습니다. : ' + paramId + ' -> ' + paramName);

    res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
    res.write('<h1>서버에서 응답한 결과입니다.</h1>');
    res.write('<div><p>redis에 사용자를 등록했습니다. : ' + paramId + ' -> ' + paramName
            + '</p></div>');
    res.write("<br><br><a href='/public/setname.html'>처음으로 돌아가기</a>");
    res.end();
});

……
```

클라이언트에서 /process/setname 요청 패스로 사용자 정보를 요청하면 사용자 아이디와 이름이 파라미터로 전달됩니다. 이 아이디와 이름은 레디스 클라이언트 객체의 hset 함수를 호출하면서 파라미터로 전달됩니다. hset 함수를 호출할 때 첫 번째 파라미터에 키 값이 전달되어야 하는데 그 값으로 user를 넣었으므로 hget 함수로 조회할 때도 첫 번째 파라미터는 user가 되어야 합니다. 레디스에 사용자 정보를 설정한 후에는 클라이언트로 응답을 보냅니다.

이 라우팅 함수 아래에는 다음 코드처럼 /process/getname 라우팅 함수를 추가합니다.

참조 파일 | DeployExample>app2.js

```
……,

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

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

    store.hget("user", paramId, function(err, username) {
        if(err) { throw err; }

        if(username) {
            console.log('redis에서 사용자를 찾았습니다. : ' + paramId + ' -> ' + username);

            res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
            res.write('<h1>서버에서 응답한 결과입니다.</h1>');
            res.write('<div><p>redis에서 사용자를 찾았습니다. : ' + paramId + ' -> ' + username
                    + '</p></div>');
            res.write("<br><br><a href='/public/getname.html'>처음으로 돌아가기</a>");
            res.end();
        } else {
            res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
            res.write('<h1>서버에서 응답한 결과입니다.</h1>');
            res.write('<div><p>redis에서 사용자를 찾지 못했습니다. : ' + paramId + '</p></div>');
            res.write("<br><br><a href='/public/getname.html'>처음으로 돌아가기</a>");
            res.end();
        }
    });

});
……
```

클라이언트에서 /process/getname 요청 패스로 사용자 정보를 요청하면 사용자 아이디가 파라미터로 전달됩니다. 이 아이디는 레디스 클라이언트 객체의 hget 함수를 호출할 때 파라미터로 전달됩니다. hget 함수의 첫 번째 파라미터는 user로 설정하고, 두 번째 파라미터에 사용자 아이디가 들어 있는 변수를 넣어 줍니다. 그러면 레디스 클라이언트가 사용자 아이디를 기준으로 이름을 조회한 후 콜백 함수로 그 결과 값을 전달합니다.

서버 쪽 기능을 모두 추가했습니다. 이제 [public] 폴더 안에 setname.html 파일을 만들고 웹 서버의 /process/setname 요청 패스로 사용자 정보를 요청해 보겠습니다.

참조 파일 | DeployExample>/public/setname.html

```
.....

    <h1>Redis 설정</h1>
    <br>
    <form method="post" action="/process/setname">
        <table>
            <tr>
                <td><label>아이디</label></td>
                <td><input type="text" name="id" ></td>
            </tr>
            <tr>
                <td><label>이름</label></td>
                <td><input type="text" name="name" ></td>
            </tr>
        </table>
        <input type="submit" value="전송">
    </form>

.....
```

getname.html 파일도 만든 다음 /process/getname 패스로 사용자 정보를 요청합니다.

참조 파일 | DeployExample>/public/getname.html

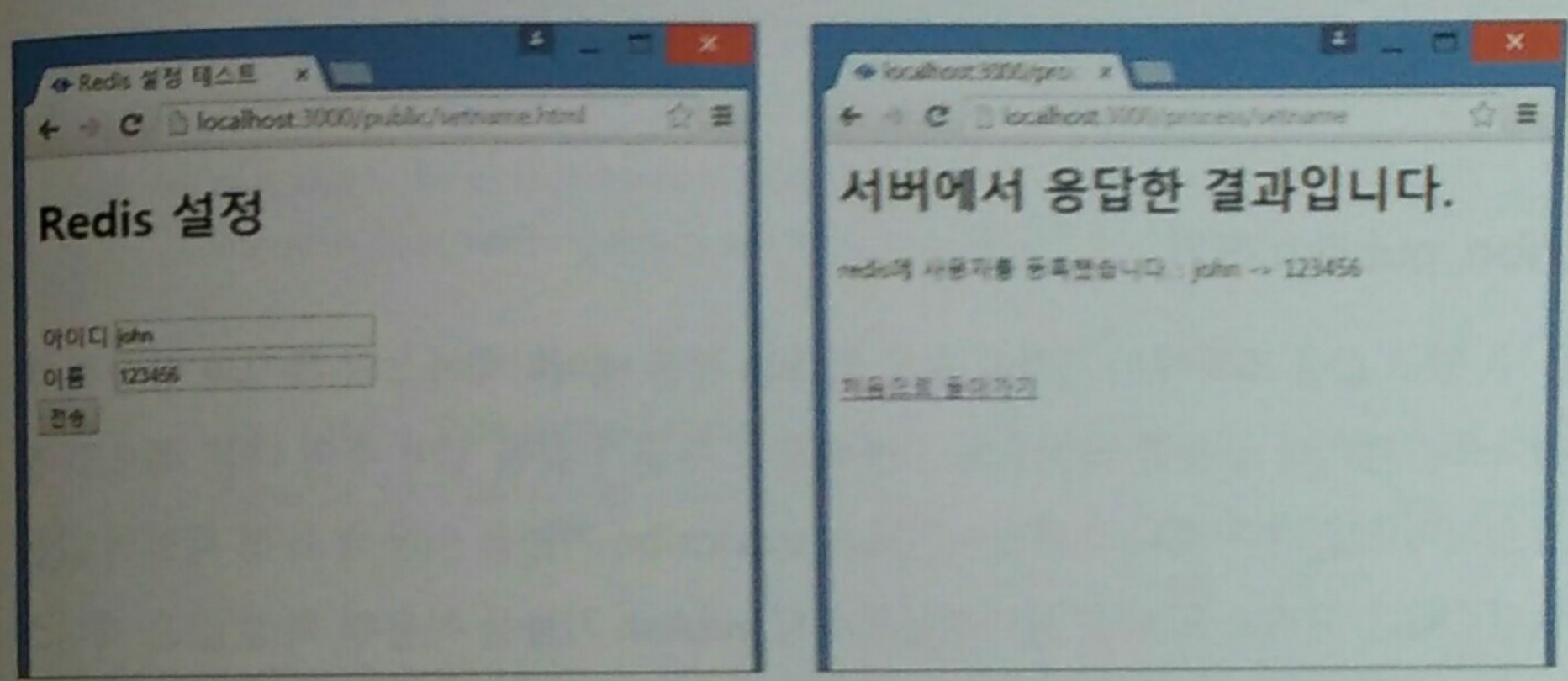
```
.....

    <h1>Redis 조회</h1>
    <br>
    <form method="post" action="/process/getname">
        <table>
            <tr>
                <td><label>아이디</label></td>
                <td><input type="text" name="id" ></td>
            </tr>
        </table>
        <input type="submit" value="전송">
    </form>

.....
```

서버 기능과 클라이언트 웹 페이지를 모두 만들었으니 pm2를 사용해 app2.js 파일을 실행합니다. 다음 명령을 실행하면 클러스터 모드로 전화되어 네 개의 프로세스가 실행될 것입니다.

```
% pm2 start app2.js -i 4 --name web1
```

그리고 웹 브라우저에서 다음 사이트를 엽니다. 사용자 아이디와 이름을 입력한 후 [전송] 버튼을 누르면 redis에 사용자를 등록했다는 응답을 받을 수 있습니다.

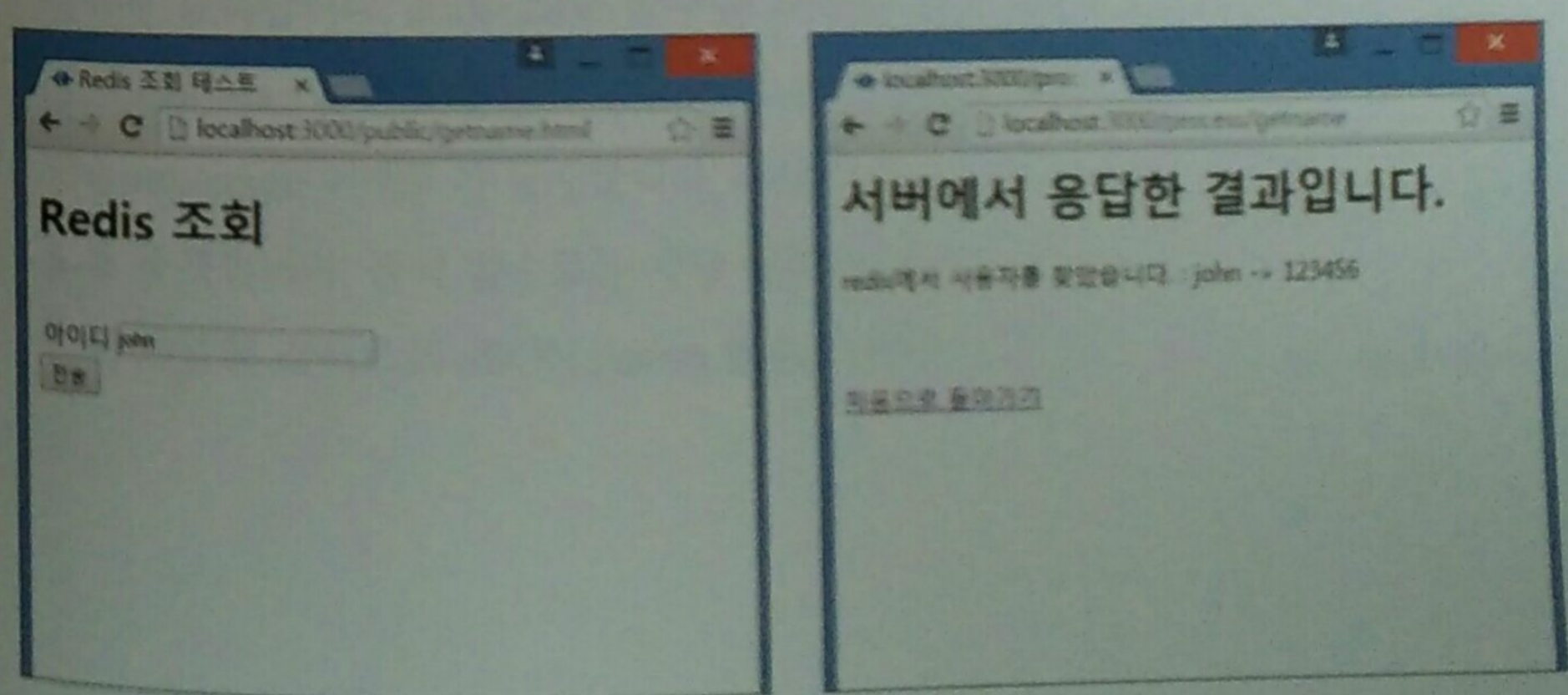
```
▶ http://localhost:3000/public/setname.html
```



▲ 레디스에 이름을 추가하려고 웹 페이지에서 아이디와 이름을 전송한 후 받은 응답 결과

이번에는 웹 브라우저에서 getname.html 페이지를 엽니다. 사용자 아이디를 입력한 후 [전송] 버튼을 누르면 redis에서 조회한 사용자 이름이 표시됩니다.

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



▲ 이름을 조회하려고 웹 페이지에서 아이디를 전송한 후 받은 응답 결과

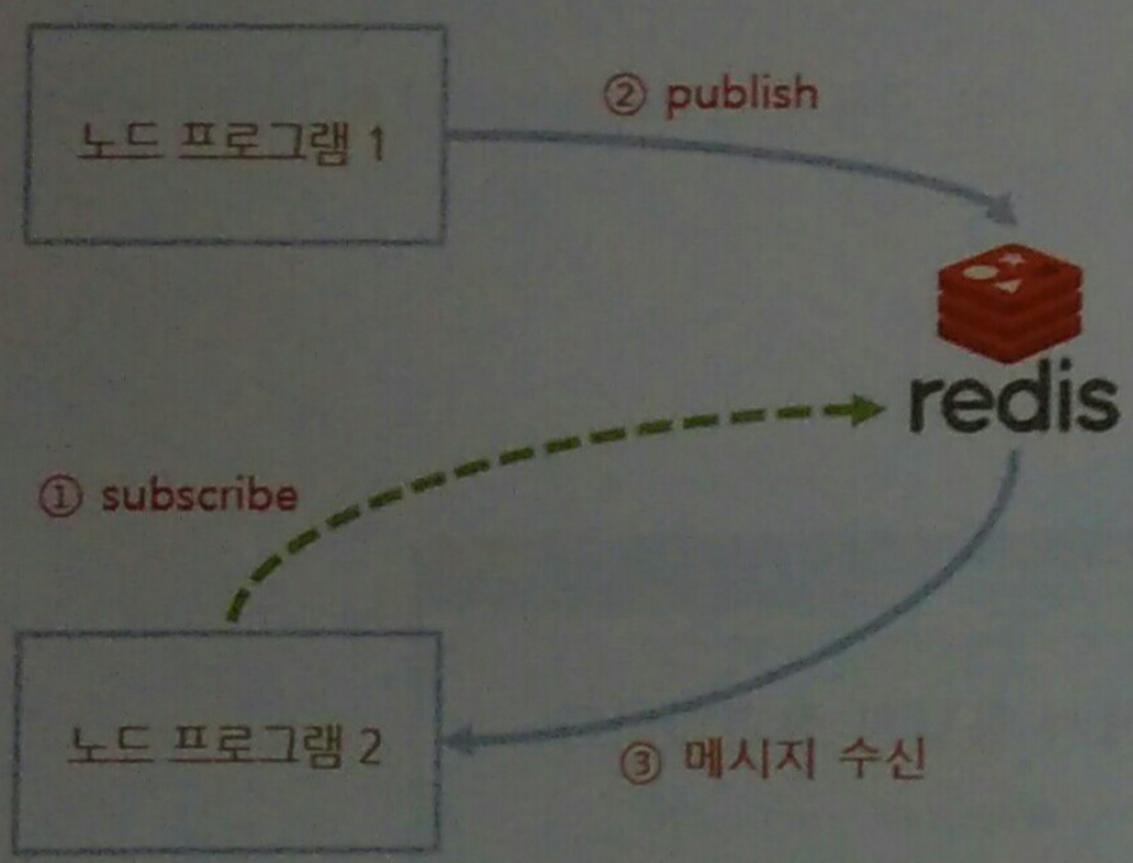
아주 단순한 기능처럼 보이지만 레디스에 설정했던 사용자 정보를 조회했다는 점은 매우 중요합니다. 서버는 클러스터 모드로 실행되었기 때문에 어떤 프로세스에서 데이터를 요청받을지 알 수 없지만 어떤 프로세스에서든 레디스에 넣어 둔 데이터를 조회할 수 있으므로 응답 결과는 항상 같습니다.

정박사의 한마디 데이터베이스에 저장하는 것과 무엇이 다를까요?

강의를 하다 보면 "서버를 데이터베이스에 연결한 후 데이터를 저장했다가 조회하는 것과 무엇이 다른가요?"라는 질문을 받기도 합니다. 레디스는 메모리에 데이터베이스가 있다고 생각해도 됩니다. 따라서 데이터를 넣었다가 가져올 수 있다는 점은 데이터베이스와 다를 바가 없습니다. 하지만 데이터 저장이 목적이 아니라 여러 프로세스 간 데이터 공유가 목적인데 이런 데이터를 레디스는 메모리에서 처리합니다. 따라서 데이터베이스보다 빠르고 간결합니다. 물론 메모리 데이터베이스보다도 빠르고 간결하게 사용할 수 있습니다.

레디스로 subscribe, publish 하기

레디스에 값을 넣을 수 있고 값을 조회해서 가져올 수도 있다는 점은 데이터베이스의 특징과 상당히 비슷합니다. 하지만 레디스는 데이터 공유를 목적으로 사용하므로 누군가 값을 넣어 주면 다른 쪽으로 전달해 주는 기능도 포함되어 있습니다. 하나의 클라이언트는 subscribe 기능을 이용해 특정 채널의 값을 받겠다고 등록할 수 있습니다. 그리고 또 다른 클라이언트에서 publish 기능을 이용해 특정 값을 레디스 서버로 보내면 subscribe로 등록한 클라이언트에 전달됩니다.



▲ subscribe와 publish로 메시지를 전달받는 과정

app2.js 파일을 복사하여 app_pubsub.js 파일을 만듭니다. 그리고 다음 코드처럼 redis 모듈을 사용하는 코드를 수정합니다.

참조 파일 | DeployExample>app_pubsub.js

```
……,

//Redis 사용하기
var redis = require("redis");
var store = redis.createClient();
var pub = redis.createClient();
var sub = redis.createClient();

var channel_name = 'chat';
sub.on('message', function(channel, dataStr) {
        console.log('Redis subscriber received message on channel ' + channel);

        // process JSON formatted string input
        var data = JSON.parse(dataStr);
        console.log('DATA : %j', data);
});

sub.subscribe(channel_name);
console.log('redis에 subscribe하였습니다. : ' + channel_name);

……,
```

redis 객체의 createClient 함수를 호출하여 레디스 클라이언트를 만들 때 세 개의 객체를 만들어 변수에 할당합니다. 그리고 sub 변수에 할당한 클라이언트 객체의 subscribe 함수를 호출하여 레디스 서버에 등록합니다. subscribe 함수를 호출할 때는 채널 이름을 파라미터로 전달합니다. 채널 이름은 channel_name 변수에 넣어 두는데 그 값은 chat으로 설정합니다. 이렇게 등록했을 때 메시지가 수신되면 message 이벤트가 발생합니다. 따라서 sub 객체의 on 함수를 사용해 message 이벤트를 처리하도록 등록합니다. 콜백 함수로는 채널 이름과 데이터가 전달됩니다. 데이터는 JSON 형식의 문자열로 만들어 전달할 것이므로 JSON.parse 함수로 자바스크립트 객체를 만들어 사용할 수 있습니다.

이제 기존 라우팅 함수를 삭제하고 새로운 라우팅 함수를 추가합니다.

참조 파일 | DeployExample>app_pubsub.js

```
.....

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

    var paramSender = req.body.sender;
    var paramReceiver = req.body.receiver;
    var paramContents = req.body.contents;

    var data = {sender:paramSender, receiver:paramReceiver, contents:paramContents};
    var dataStr = JSON.stringify(data);
    pub.publish('chat', dataStr);

    res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
    res.write('<h1>서버에서 응답한 결과입니다.</h1>');
    res.write('<div><p>redis로 publish했습니다. 서버 로그를 확인하세요. : ' + dataStr + '</p></
div>');
    res.write("<br><br><a href='/public/publish.html'>처음으로 돌아가기</a>");
    res.end();
});

.....
```

클라이언트로부터 /process/publish 패스로 메시지 전달 정보를 요청받으면 세 개의 파라미터가 전달됩니다. 이 파라미터들을 자바스크립트 객체로 만든 후 JSON 포맷의 문자열로 변환합니다. JSON 문자열로 변환할 때는 JSON.stringify 함수를 사용합니다. 그다음 pub 객체의 publish 함수를 호출하여 레디스 서버로 보냅니다. 이때 채널의 이름은 chat으로 설정합니다. 이제 요청을 보낼 클라이언트 웹 페이지를 [public] 폴더 안에 만듭니다.

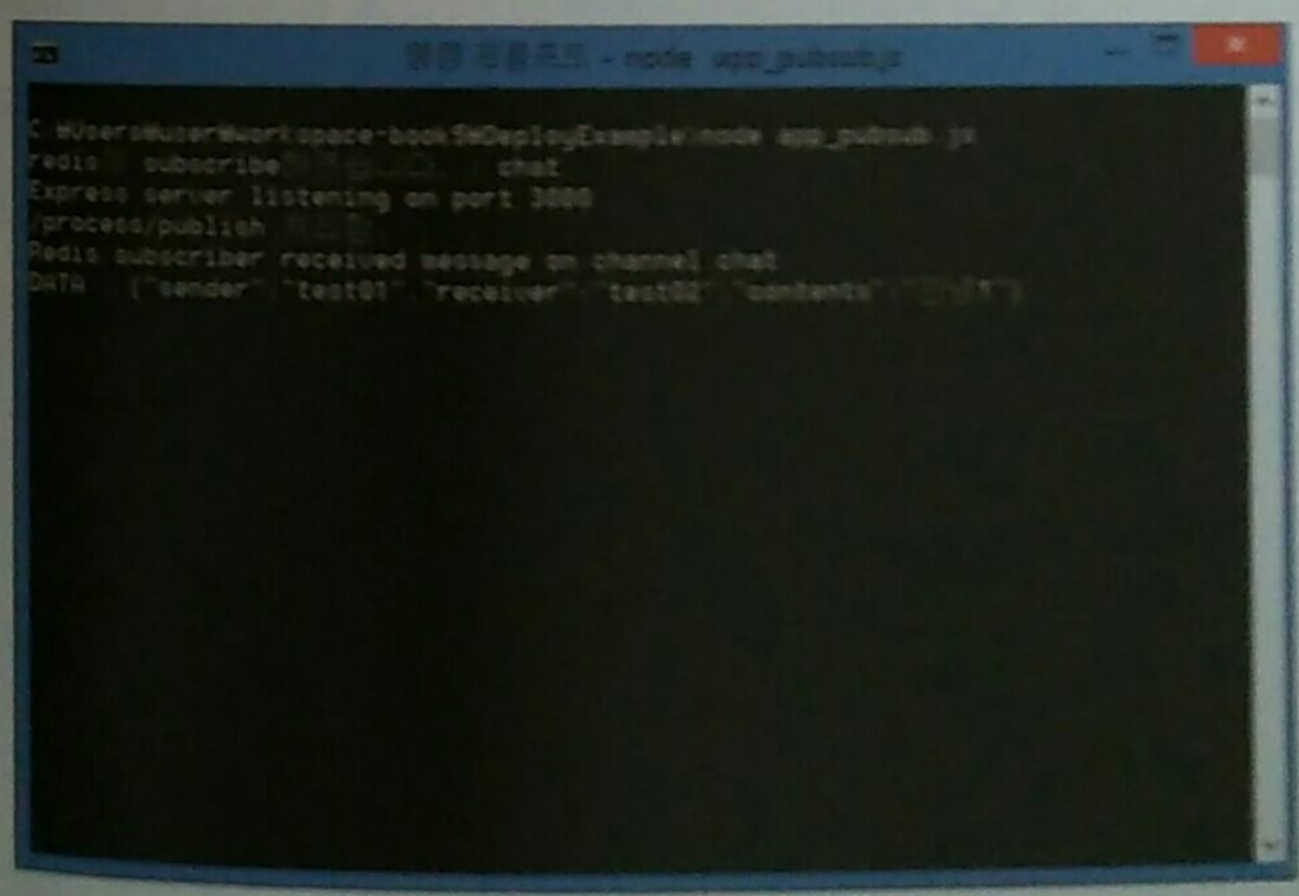
참조 파일 | DeployExample>/public/publish.html

```
.....

<h1>Redis publish</h1>
<br>
<form method="post" action="/process/publish">
    <table>
        <tr>
```

```
        <td><label>보내는사람</label></td>
        <td><input type="text" name="sender" /></td>
    </tr>
    <tr>
        <td><label>받는사람</label></td>
        <td><input type="text" name="receiver" /></td>
    </tr>
    <tr>
        <td><label>내용</label></td>
        <td><input type="text" name="contents" /></td>
    </tr>
  </table>
  <input type="submit" value="전송">
</form>

.....
```

publish.html 파일에서는 보내는 사람, 받는 사람, 내용을 웹 서버로 전송합니다. app_pubsub.js 파일을 실행한 후 웹 브라우저에서 publish.html 파일을 엽니다. 입력 상자에 값을 입력한 후 [전송] 버튼을 누르면 서버 쪽에 로그가 출력됩니다.

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



▲ 레디스로부터 전달받은 메시지를 출력한 경우

지금까지 살펴본 subscribe/publish 기능은 채팅 서버처럼 상태를 유지해야 하는 경우에 유용하게 사용할 수 있습니다.

영덕서의 한마디 상태 유지와 관련된 Stateful과 Stateless라는 단어를 들어 봤나요?

Stateful은 상태를 유지한다는 의미로 사용되고 Stateless는 상태를 유지하지 않는다는 의미로 사용됩니다. 여기에서 상태를 유지한다는 것은 데이터를 유지한다는 의미로 생각하면 쉽습니다. 다시 말해, 채팅 서버에서 로그인한 사용자의 데이터를 보관하고 유지한다는 의미로 '상태를 유지한다' 또는 'Stateful'이라고 이야기합니다.

채팅 서버에서 레디스를 사용하는 샘플

채팅 서버가 클러스터 모드로 실행될 경우에 연결된 소켓 정보를 레디스에서 관리할 필요가 있습니다. socket.io 모듈에는 이 기능이 이미 있기 때문에 여러분은 레디스만 설정하면 바로 사용할 수 있습니다. 다음 코드는 socket.io에서 레디스를 사용하기 위해 추가하는 코드 샘플입니다.

참조 파일 | DeployExample>app_chat_redis.js

```
.....

// socket.io 모듈 불러오기
var socketio = require('socket.io');

// redis 모듈 불러온 후 레디스 클라이언트 객체 만들기
var redis = require("redis");
var pub = redis.createClient(6379,'127.0.0.1');
var sub = redis.createClient(6379,'127.0.0.1');
var store = redis.createClient(6379,'127.0.0.1');

.....

var app = express();

.....

// 익스프레스 서버 시작
var server = http.createServer(app).listen(3000, function() {
    console.log('서버가 시작되었습니다. 포트 : ' + app.get('port'));
});

//socket.io 서버 시작
var io = socketio.listen(server);
console.log('socket.io 요청을 받아들일 준비가 되었습니다.');
```

```
// socket.io 서버에 redis 설정
io.set('store',new socketio.RedisStore({
    redis: redis
    ,redisPub: pub
    ,redisSub: sub
    ,redisClient: store
}));

.....
```

redis 모듈을 불러온 후 레디스 클라이언트 객체를 세 개 만듭니다. createClient 함수를 사용해서 레디스 클라이언트 객체를 만들 때는 레디스 서버의 IP와 포트 번호를 지정할 수 있습니다. socket.io 서버를 시작한 후 반환받은 객체에는 set 함수가 있는데 이 함수를 호출하면서 store라는 이름으로 객체 하나를 설정합니다. 이 객체는 socketio.RedisStore 프로토타입으로 만들 수 있는데 생성자 함수로 전달하는 객체 안에는 레디스 관련 객체들을 속성으로 넣어 줍니다. 이렇게 설정만 하면 socket.io 모듈은 클라이언트가 연결했을 때의 소켓 정보를 레디스와 자동으로 주고받게 됩니다. 따라서 여러 개의 프로세스가 실행되어도 모든 프로세스끼리 채팅 메시지를 주고받을 수 있습니다. 물론 클라이언트가 로그인했을 때 그 정보를 넣어두기 위해 10장에서 사용한 login_ids 변수도 레디스에 로그인 정보를 넣었다가 조회하는 방식으로 변경해야 합니다. 이 부분은 여러분이 직접 수정해보기 바랍니다.

지금까지 배포에 필요한 내용을 살펴보았습니다. 배포에 필요한 내용으로 로드 밸런싱(Load Balancing)과 같은 것이 더 있지만 여기에서 다룬 내용만으로도 여러분은 노드로 만든 서비스를 충분히 안정적으로 운영할 수 있습니다.

A~D

access token 값 404
Access Token 397
action 속성 194
add 변수 86
addDevice 함수 568
addToDiscussion 함수 461
AES 객체 494
AES 표준 알고리즘 493
Ajax 15
Anonymous function 86
API 13
Application 13
Architecture 34
argv 속성 64
authenticate 메소드 253
authToken 속성 404
authUser 함수 215
Binding 36
Blocking IO 31
body-parser 미들웨어 167
Boolean 83
Bootstrap 333
border-box 321
broadcast 객체 431
bson 모듈 218
B-Tree 인덱싱 23
Buffer 객체 123
calc 객체 70
Callback Function 32
Card 컴포넌트 317
cd 명령 55
chunk 변수 36
Client 14
Closure 104
cmd 65
command 속성 426
Comment 128
CommonJs 표준 스펙 36
conn 파라미터 272
connect.sid 쿠키 189
connectDB 함수 214
connect-flash 모듈 368
Connection Pool 270
connection 이벤트 137, 418
console 57
console 객체 57
Constructo 105
constructor 객체 107
Content-Type 헤더 값 143
Controller 337
cookie-parser 미들웨어 180
cookies 객체 181
coords 변수 544
cors 모듈 413
count 변수 103
createPool 메소드 270
createSchema 속성 501
Cross-Origin Resource Sharing 413
crypto 모듈 251
CSS 속성 322
CSS 코드 459
Current 43
data 이벤트 148
database 변수 215
Date 타입 237
db_schemas 배열 객체 308
Declaration 86
DELETE SQL문 279
delete 키워드 96
dependencies 정보 210
Description 243
dest 속성 192
Device 13
disconnected 이벤트 229
docs 문서 객체 217
Document 202

E~H

EADDRINUSE 139
Echo 기능 425
echo 핸들러 471
ejs 뷰 엔진 349
ejs 템플릿 파일 345
ejs 파일 341
email 칼럼 374
Encryption 243
enctype 속성 194
endPoint 속성 473
Enide Studio 38
error 이벤트 229
Ethernet Card 14
Event driven IO 34
EventEmitter 112
events 모듈 116
exports 57
exports 전역 객체 68
Express 17
express-error-handler 미들웨어 176
Expression 86
express-session 모듈 185
file 객체 33
findAll 함수 238
findById 함수 238
findCircle 함수 524
flexbox 461
forEach문 489
forever list 명령 661
forever start 명령 661
forever 모듈 660
for문 93
fs 모듈 119, 128
function 키워드 86
Function 59
GCM 서비스 583
GCM 푸시 전송 방식 582
geometry 속성 501
GeoSpatial Indexing 499

GET 방식 147
getDeviceInfo 메소드 575
git add 명령 634
git clone 명령 644
git commit 명령 634
git push 명령 634
git 로컬 저장소 634
git 클라이언트 623
Global Object 57
Google Cloud Messaging 582
Google Play Services
라이브러리 586
google.maps.Rectangle
프로토타입 544
GPS 좌표 516
gradle 빌드 562
Group Chatting 438
GUI 프로그램 623
Handler 31
HeidiSQL 265
Hello World 53
history 객체 102
host 변수 137
HTTP 15
http 객체 36
http 라이브러리 35
http 모듈 135
HTTP 프로토콜 35
HTTP 헤더 143

I~N

include 키워드 348
indexOf 함수 600
int 82
Interpreter 34
io 변수 418
IOT, Internet Of Things 16
isLoggedIn 미들웨어 380
Item 90
jade 템플릿 355
jade 포맷 350
Java EE 39
Java Standard Edition 39
JAVA_HOME 환경 변수 66
jQuery 파일 420
JSON 데이터 161
JSON 포맷 58
JSON-RPC 모듈 467
JSON-RPC 서버 20
LBS, Location Based Service 498
length 속성 434
limits 객체 193
LineString 위치 데이터 500
listDevice 함수 568
listpost 함수 344, 612
localhost 18
Log Level 131
Logger 130
LTS 43
MacAddress 567
Map 객체 574
margin 321
Marker 531
Media Query 326
Method 59
MIME Type 144
mobile 속성 85
Model-View-Controller 패턴 337
Module 17
moment 모듈 131
mongo 쉘 206
mongodb 모듈 213
MongoDB 17
mongoose 226
multer 미들웨어 190
multipart 190
MVC 패턴 337
MySQL 201, 258
name 속성 92
nconf 73
nconf 패키지 74
Network Card 14
new 연산자 105
next 함수 객체 160
Node Package Manager 73
Node 퍼스펙티브 49
node.exe 54
Node.js 12, 30
Nodeclipse 54
node-gcm 패키지 582
Non-Blocking IO 31
Non-Blocking Programming 100
NoSQL 데이터베이스 201
Not Only SQL 201
npm -v 명령 45
npm 패키지 73
npm 프로그램 37
null 83
Number 83

O~R

OAuth 인증 367
Object 83
Object Mapper 227
Object oriented 104
onLoad 함수 529
Open Standard Protocol 396
open 이벤트 229
OpenAPI 413
os 모듈 77
Package 37
padding 321
pagination 602
param 메소드 164
params 배열 객체 481
passport 모듈 368
Passport 17
passport-facebook 모듈 404
passport-local 모듈 368
password 칼럼 374
password 파라미터 217

path 모듈 78
perPage 변수 603
Person 변수 63
Person 프로토타입 객체 105
Perspective 49
placeholder 속성 325
pm2 모듈 663
Point 위치 데이터 500
Polygon 위치 데이터 500
port 속성 154
POST 방식 147
process 객체 64
Protocol 15
prototype 속성 106
provider 속성 404
putSingleFilesInArray 속성 193
query string 164
query 속성 112
querystring 모듈 111
R*Tree 23
radius 속성 524
recepient 속성 426
Record 202
Rectangle 객체 544
redis 모듈 679
Redis 667
Reference 598
regexp 표현식 213
RegisterThread 클래스 588
registrationId 567
Relational Database 201
Remote Procedure Call 20, 467
render 메소드 535
req 요청 객체 380
request 이벤트 137
res 객체 36
Responsive Web 326
return 키워드 101
rhc setup 명령 640
Robomongo 224
room 이벤트 440
route_info 배열 객체 308
router 171
Routing 155
rows 객체 277
RPC 20
RPC 467
R-Tree 23
Ryan Dahl 30

S~Z

salt 속성 257
–save 옵션 76
Schema Type 229
Schema 227
ScrollView 559
SELECT문 277
Semantic UI 라이브러리 316
sender 속성 426
server 변수 416
Server 14
session 객체 188
Shell 205
showpost 뷰 템플릿 607
Socket 35
socket.io 모듈 413
Spatial Data 23
Spatial Indexing 23
splice 메소드 601
static 미들웨어 166
Strategy 367
String 82
Stub 468
subscribe 기능 674
success 속성 474
supervisor 모듈 662
system variables 67
TelephonyManager 576
Template 17
TextView 559
this 객체 296
this 키워드 105
Titon 333
Token 179
Toolbelt 626
Type 82
Type based language 82
type 속성 426
Uber 545
UI 도구 로보몽고 224
UI 라이브러리 316
undefined 83
UPDATE SQL문 279
URL 객체 109
url 모듈 109
url 변수 110
URL 정보 308
URL 파라미터 174
USB 드라이버 552
USB 디버깅 552
user variables 67
user 세션 185
User.prototype 객체 296
Users 배열 91
UTF-8 61
V8 자바스크립트 엔진 34
var 키워드 82
view engine 속성 154
View Engine 337
View file 154
View Template 17
vim 명령 647
virtual 메소드 257
WifiManager 577
WindowManager 577
winston 모듈 129
Workspace 49
XHR 413
XMLHttpRequest 413

ㄱ~ㅁ

개방형 표준 프로토콜 396
객체 지향 104
객체 59
객체의 원형 104
경위도 좌표 516
계산기 객체 115
공간 데이터 23
공간 인덱싱 23, 236
관계형 데이터베이스 201
구글맵 라이브러리 530
구글맵 528
그룹 채팅 20, 438
내장 모듈 77
네이티브 코드 34
네트워크 인터페이스 정보 78
네트워크 카드 14
노드 30, 39
노드 사이트 43
노드 실행 파일 56
노드 프로젝트 55
노드제이에스 12
노드클립스 54
논블로킹 입출력 31
단말 기기 13
단방향 암호화 243
더하기 기능 함수 69
동기 입출력 31
동기식 IO 117
디바이스 정보 552
디폴트 메시지 175
라우터 150
라우터 미들웨어 171
라우팅 함수 216
라우팅 155
라이언 달 30
레디스 666
레코드 202
레퍼런스 598
로거 130
로그 129
로그 수준 131
로드 밸런싱 679
로보몽고 224
로컬 인증 367
로컬 인증 방식 375
로컬 호스트 18
마커 객체 531
매개변수 57
맥 주소 567
멀티 파트 포맷 190
메소드 59
메시지 문자열 435
명령 문자열 435
명령 프롬프트 55
모델 인스턴스 234
모듈 17
모바일 단말 25
모바일 단말 관리 기능 565
모바일 서버 24, 549
모바일 웹 25
몽고디비 17, 202
몽구스 226
문서 객체 202
미들웨어 150
미들웨어 함수 158
미디어 쿼리 326

ㅂ~ㅇ

바이너리 데이터 123
바인딩 36
반응형 웹 326
배열 90
배열 객체 94
배열 요소 96
변수 82
변수 상자 84
복호화 243, 491
부트스트랩 333
뷰 엔진 154, 337
뷰 템플릿 17
뷰 파일 154
뷰 포트 457
블로킹 31
비관계형 데이터베이스 201
비동기 입출력 31
비동기 프로그래밍 100
비동기식 IO 117
사물인터넷 16
사용자 인증 413
사용자 정의 이벤트 425
사용자 정의 환경 변수 67
상대 패스 151
생성자 105
서버 개발 도구 30
서버 포트 309
서버 14
선언문 86
설명글 128
세션 183
세션 정보 189
셸 205
소켓 ID 452
소켓 서버 133
소켓 클라이언트 133
소켓 35
스레드 풀 35
스크롤뷰 559
스키마 타입 229
스키마 227
스텁 468
스트림 객체 127
스트림 단위 125
시스템 PATH 204
시스템 구조 34
시스템 환경 변수 67
실행되지 않는 설명글 136
아마존 클라우드 648
아키텍처 34
안드로이드 스튜디오 549
암호화 라이브러리 492

암호화 243
애플리케이션 13
에이잭스 방식 22
오브젝터 맵퍼 227
오픈시프트 클라우드 638
외장 모듈 73
요소 90
요청 파라미터 111, 164
우버 545
워크스페이스 49
원소 90
웹 문서의 원형 17
웹 서버 17
웹 프레임워크 17
위치 기반 서비스 서버 22, 498
위치 정보 22
윈도우 서비스 668
유효성 검사 371
이니드 스튜디오 플러그인 39
이니드 스튜디오 38
이더넷(Ethernet) 카드 136
이벤트 112
이벤트 기반 입출력 모델 34
이벤트 루프 35
이진 84
이클립스 사이트 41
익명 함수 86
익스프레스 서버 객체 152, 342
익스프레스 17
인덱스 235
인덱스 값 93
인스턴스 객체 106, 665
인증 토큰 397
인터프리터 34
일대일 채팅 20

ㅈ~ㅎ

자료형 82
자바 SE 39
자바스크립트 33
자바스크립트의 자료형 83
전역 객체 자료형 57
전역 객체 57
점 연산자 85
제이슨 20
주석 128
주소 문자열 110
채팅 서버 19
초기화 함수 107
커넥션 풀 270
컨트롤러 337
컬렉션의 구조 300
컬렉션의 집합 202
코드 값 435
콜백 함수 32, 100
쿠키 180
쿠키 객체 181
쿠키 정보 181
크롬 브라우저 53
클라이언트 14
클래스 104
클러스터 모드 678
클러스터링 663
클로저 104
타이톤 333
타입 기반의 언어 82
텍스트뷰 559
토큰 179
툴벨트 626
파라미터 58
패스포트 17, 367
패키지 37
퍼스펙티브 49
포트 14
포트 정보 211
포트포워딩 설정 645
표준 데이터 포맷 20
표준 라이브러리 35
표현식 86
푸시 메시지 24
프로토콜 15
프로토타입 객체 107, 116
하이브리드 앱 25
한글 인코딩 61
함수 59
핸들러 22, 31
헤로쿠 클라우드 623

기타

/login 패스 176, 376
/profile 패스 376
/signup 패스 376
_doc 속성 239
 태그 319
<form> 태그 194
<head> 태그 348
<input> 태그 194
<table> 태그 319
<textarea> 태그 472
2dsphere 타입 502
2차원 데이터 23

세상의 속도를
따라잡고 싶다면

Do it!

Node.js 프로그래밍

Do it!
Node.js Programming

서버 프로그래밍에 첫 입문하여 실무 수준까지 한 권으로 끝!

저자 직강 동영상 강의 + 실무에 바로 쓰는 소스 파일 제공 + 궁금하면 물어 보자! 저자 Q&A

이 책에서 다루는 5가지 서버

웹 서버를 만들며 입문하고 채팅 서버, 모바일 서버, JSON-RPC 서버, 위치 기반 서비스 서버까지 만들며 노드 전문가가 된다!

이 책을 보면 우리 회사에 필요한 서버를 당장 만들 수 있는 능력이 생긴다!

웹 서버 | 웹 디자인, 웹 퍼블리셔, 웹 개발자 필수!
홈페이지 · 쇼핑몰 사이트 · 기업 업무 페이지

채팅 서버 | 웹 개발자, 앱 개발자, 윈도우 프로그램 개발자 필수!
채팅 앱 개발 프로젝트 · 공유 솔루션 개발 프로젝트

모바일 서버 | 앱 개발자 필수!
앱 개발 프로젝트 · 단말 관리 프로젝트

JSON-RPC 서버 | 앱 개발자, 윈도우 프로그램 개발자 필수!
기업 업무 개발 프로젝트 · 앱 개발 프로젝트

위치 기반 서비스 서버 | 앱 개발자, 웹 개발자 필수!
매장 찾기 프로젝트 · 친구 찾기 프로젝트

값35,000원
13000