09 패스포트로 사용자 인증하기

웹 사이트의 여러 기능을 사용하려면 사용자 로그인을 요구하는 경우가 많습니다. 그리고 만약 사용자가 웹 사이트에 등록이 안 되어 있다면 회원가입을 유도합니다. 요즘에는 회원가입을 번거로운 일로 생각하는 사람들이 많아 '페이스북으로 로그인', '구글로 로그인' 등의 기능을 사용해 번거로움을 줄여주기도 합니다. 그런데 이런 사용자 인증 기능을 만들려면 많은 코드를 입력해야 하고, 페이스북이나 구글 사이트와 어떻게 연동해야 하는지도 잘 알고 있어야 합니다. 하지만 노드의 패스포트(Passport) 모듈을 사용하면 아주 간단한 작업만으로도 로그인과 회원가입 기능을 만들 수 있습니다.

지금까지는 함수를 사용해서 직접 로그인 기능이나 사용자 추가 기능을 만들어 보았는데 이 장에서는 로컬 사용자 인증, 페이스북으로 로그인 등 사용자 인증 기능을 패스포트 모듈로 만들어 볼 것입니다.

질문으로 시작하기

질문1. 어떻게 패스포트를 이용해 로그인 하도록 만드나요? ➡ 09-1. 패스포트로 로그인하기

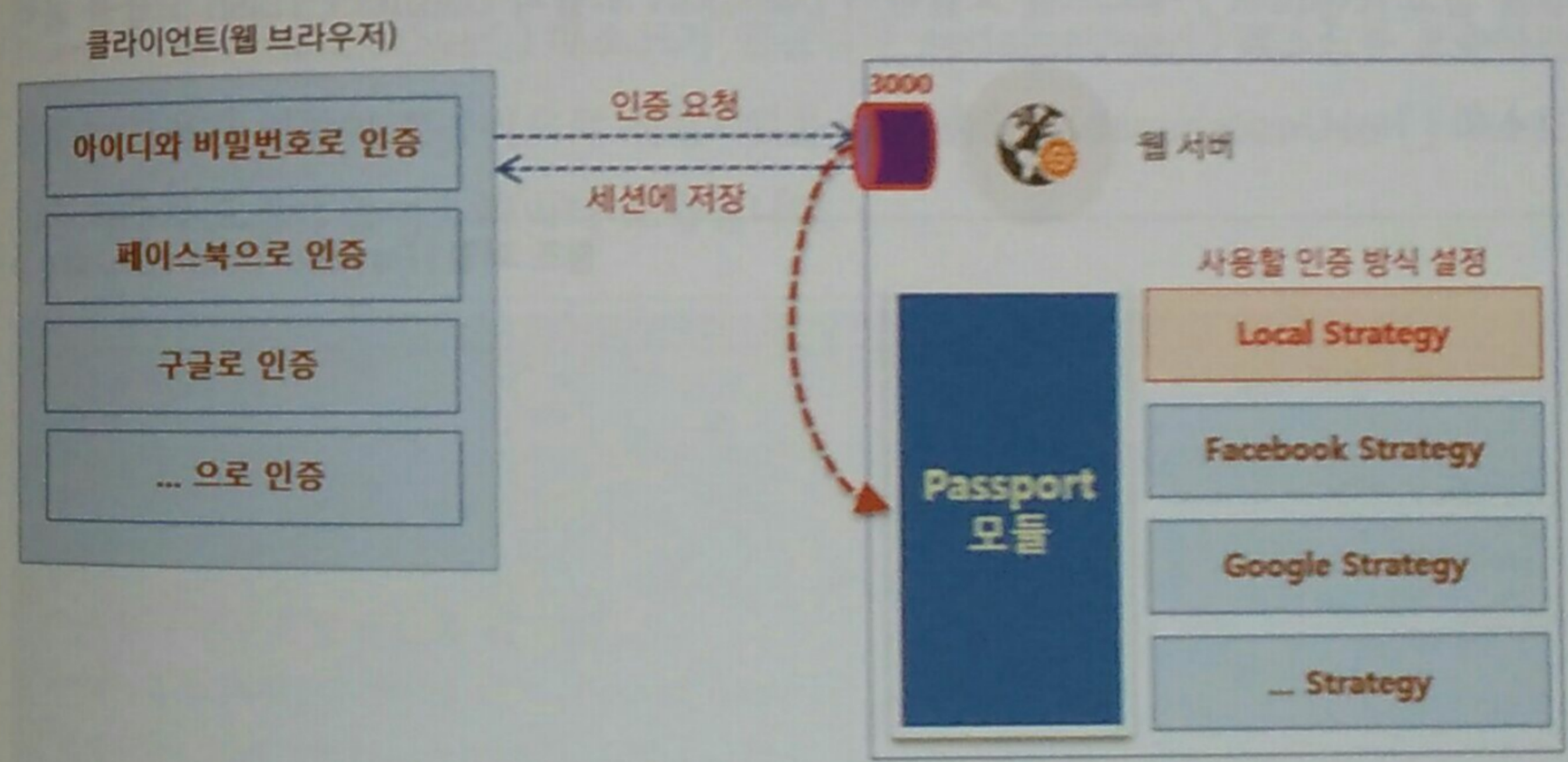
질문2. 패스포트 사용 부분만 모듈로 만들고 싶나요? ➡ 09-2. 패스포트 관련 코드를 모듈화하기

질문3. '페이스북으로 로그인' 기능은 어떻게 만드나요? ➡ 09-3. 페이스북으로 로그인하기

09-1 | 패스포트로 로그인하기

패스포트(Passport)는 노드에서 사용할 수 있는 사용자 인증 모듈입니다. 이 모듈은 사용 방법이 간단할 뿐만 아니라 사용자 인증 기능을 독립된 모듈 안에서 진행할 수 있도록 도와줍니다. 특히 익스프레스를 사용할 경우에는 미들웨어로 끼워 넣을 수 있어 몇 가지 간단한 설정만으로도 로그인 기능을 만들 수 있습니다.

패스포트는 수백 가지의 인증 방식을 제공하는데, 어떤 인증 방식을 사용할 것인지 결정하는 것이 Strategy입니다. 각각의 인증 방식은 Strategy로 만들어져 있기 때문에 어떤 Strategy를 사용하느냐에 따라 인증 방식이 달라집니다. 대표적인 인증 방식으로는 데이터베이스에 저장된 사용자 정보와 비교하는 **로컬 인증**, 페이스북이나 트위터의 계정을 사용하는 **OAuth 인증** 등이 있습니다. 다음은 **패스포트 미들웨어**를 사용해 웹 서버에서 사용자를 인증하는 방식을 보여 줍니다.



▲ 패스포트를 사용해 웹 서버에서 사용자를 인증하는 구조

웹 브라우저에서 사용자 인증을 요청할 때는 단순히 웹 서버의 데이터베이스에 저장된 아이디와 비밀번호를 비교하도록 만들 수 있고, 페이스북이나 구글의 계정을 사용해서 인증하도록 만들 수도 있습니다. 클라이언트가 인증을 요청하면 웹 서버에 있는 패스포트 모듈은 미리 설정해 둔 인증 방식으로 사용자를 인증한 후 성공하면 사용자 정보를 세션에 저장합니다. 이 **세션 정보**는 정상적으로 사용자 인증이 되었을 때만 사용할 수 있으므로 클라이언트에서는 사용자가 로그인되었는지 아닌지를 구별할 수 있습니다.

로컬 인증을 위해 데이터베이스 스키마와 패스포트 설정하기

패스포트 모듈에서 사용할 수 있는 인증 방식 중에서 로컬 인증을 사용하는 방법을 알아봅니다. 먼저 몽고디비 데이터베이스에 저장된 아이디와 비밀번호로 인증하는 과정을 통해 패스포트 모듈 사용 방법을 살펴보겠습니다.

앞 장에서 만들어 둔 ViewExample 프로젝트를 복사하여 PassportExample 프로젝트를 만듭니다. 새로 만든 프로젝트에서는 passport 모듈과 passport-local 모듈 그리고 connect-flash 모듈을 사용합니다. 명령 프롬프트를 열고 세 개의 모듈을 모두 설치합니다.

```
% npm install passport --save
% npm install passport-local -save
% npm install connect-flash --save
```

passport 모듈은 사용자 인증 처리에 필요한 기본 기능을 제공하며, passport-local 모듈은 웹 서버에서 직접 사용자의 아이디와 비밀번호를 전달받아 데이터베이스에 저장된 정보와 비교하는 로컬 인증 기능을 제공합니다. connect-flash 모듈은 요청 객체에 메시지를 넣어 둘 수 있는 기능을 제공하는데, 보통 웹 서버 안의 다른 함수에 메시지를 전달하거나 뷰 템플릿을 처리하는 함수에 메시지를 전달하기 위해 사용합니다.

app.js 메인 파일을 열고 require( ) 메소드를 호출하여 passport 모듈과 connect-flash 모듈을 불러옵니다.

참조 파일 | PassportExample>app.js

```
.....

// = = = = =  Passport 사용 = = = =  = //
var passport = require('passport');
var flash = require('connect-flash');

.....
```

passport 모듈은 사용자 인증 처리에 필요한 기본 기능을 제공하며, passport-local 모듈은 웹 서버에서 직접 사용자의 아이디와 비밀번호를 전달받아 데이터베이스에 저장된 정보와 비교하는 로컬 인증 기능을 제공합니다. connect-flash 모듈은 요청 객체에 메시지를 넣어 둘 수 있는 기능을 제공하는데, 보통 웹 서버 안의 다른 함수에 메시지를 전달하거나 뷰 템플릿을 처리하는 함수에 메시지를 전달하기 위해 사용합니다.

app.js 메인 파일을 열고 require( ) 메소드를 호출하여 passport 모듈과 connect-flash 모듈을 불러옵니다.

참조 파일 | PassportExample>app.js

```
……

// ===== Passport 사용 설정 ===== //
app.use(passport.initialize());
app.use(passport.session());
app.use(flash());

……
```

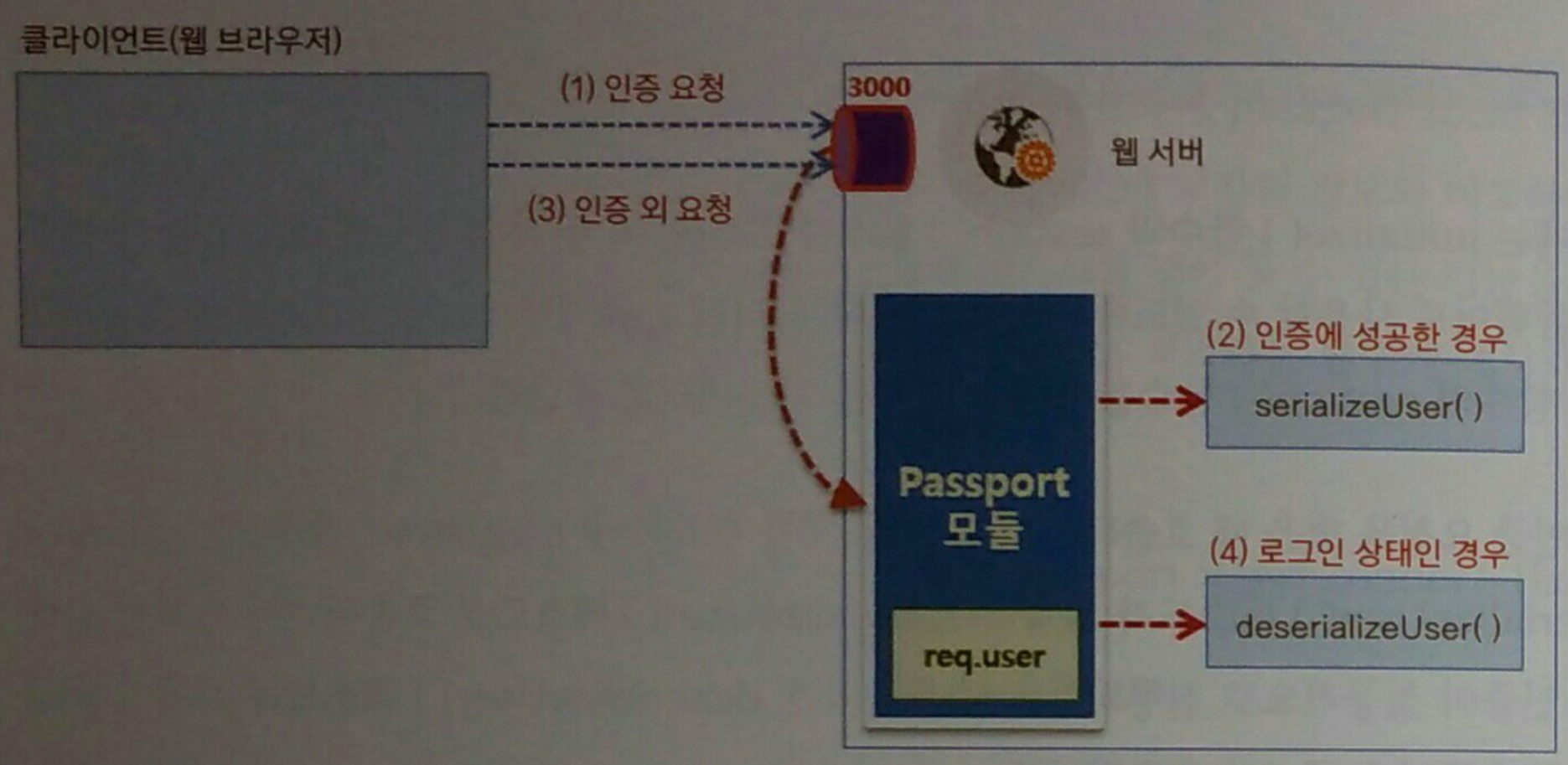
passport 모듈에는 initialize( ) 함수와 session( ) 함수가 있으며, 이 두 개의 함수를 호출했을 때 반환되는 객체를 미들웨어로 사용할 수 있도록 만들면 됩니다. 따라서 app 객체의 use( ) 메소드를 호출하면서 initialize( ) 함수와 session( ) 함수를 호출한 결과를 파라미터로 전달합니다.

이제 사용자가 인증 요청을 했을 때 호출되는 메소드 두 개를 추가합니다. passport 객체에는 serializeUser( )와 deserializeUser( ) 메소드가 있습니다. serializeUser( ) 메소드를 호출하면서 등록한 콜백 함수는 사용자 인증이 성공적으로 진행되었을 때 호출되며, deserializeUser( ) 메소드는 사용자 인증 이후 사용자 요청이 들어올 때마다 호출됩니다.

참조 파일 | PassportExample>app.js

```
……

// 사용자 인증에 성공했을 때 호출
passport.serializeUser(function(user, done) {
  console.log('serializeUser() 호출됨.');
  console.dir(user);

  done(null, user);
});

// 사용자 인증 이후 사용자 요청이 있을 때마다 호출
passport.deserializeUser(function(user, done) {
  console.log('deserializeUser() 호출됨.');
  console.dir(user);

  done(null, user);
});

……
```

사용자 인증이 성공했을 때 호출되는 serializeUser( ) 콜백 함수는 전달받은 user 객체의 정보를 콘솔 창에 출력한 후 **done( ) 메소드**를 호출합니다. done( ) 메소드를 호출하면서 두 번째 파라미터로 객체를 전달하면 이 객체는 그다음에 처리할 함수 쪽으로 전달됩니다. 사용자 인증 이후 사용자 요청이 발생할 때마다 호출되는 deserializeUser( ) 메소드에서는 전달받은 user 객체의 정보를 콘솔 창에 출력해서 확인하세요. done( ) 메소드를 호출했을 때 반환되는 사용자 객체는 요청 객체인 req에 user 속성으로 추가한 후 다른 페이지로 전달됩니다.



▲ serializeUser( )와 deserializeUser( ) 함수가 호출되는 경우

이전까지는 아이디와 비밀번호를 사용해 사용자 정보를 확인했습니다. 이번에는 이메일을 아이디로 사용하고 비밀번호와 함께 인증하도록 변경할 것입니다. 사용자가 이미 다른 용도로 사용하고 있던 고유한 이메일 주소를 아이디로 사용할 수 있도록 설정하면 회원가입이 한결 편리해지겠죠? 이렇게 하려면 먼저 **데이터베이스 스키마**를 변경해야 합니다.

[database] 폴더의 user_schema.js 파일을 열고 다음과 같이 UserSchema 객체의 속성들을 변경합니다.

참조 파일 | PassportExample>/database/user_schema.js

```
.....

// 스키마 정의
var UserSchema = mongoose.Schema({
  email : {type : String, 'default' : ''}
  , hashed_password : {type : String, required : true, 'default' : ''}
  , name : {type : String, index : 'hashed', 'default' : ''}
  , salt : {type : String, required : true}
```

```
    , created_at : {type : Date, index : {unique : false}, 'default' : Date.now}
    , updated_at : {type : Date, index : {unique : false}, 'default' : Date.now}
  });

……,
```

앞에서 만든 스키마 칼럼 중에서 id 칼럼은 email 칼럼으로 대체하였고 age 칼럼은 삭제하였습니다. 또 각각의 속성을 구분하는 콤마(,) 기호도 각 줄의 앞으로 보내어 필요할 때 한 줄씩 삭제하기 쉽도록 수정하였습니다. 앞에서 입력한 id와 name의 유효성 검사 부분을 삭제한 후 그 아래쪽에 path( ) 메소드를 사용해 email과 hashed_password 칼럼 값이 공백은 아닌지 확인합니다.

참조 파일 | PassportExample>/database/user_schema.js

```
……,

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

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

……,
```

path( ) 메소드는 칼럼 값을 확인하는 데 사용합니다. 이 메소드를 호출하면서 칼럼의 이름을 전달하면 됩니다. 여기에서는 emial과 hashed_password 칼럼을 확인하였으며 칼럼 값을 확인한 후 validate( ) 메소드를 호출하면 그 값이 유효한지를 체크합니다.

그 아랫부분을 보면 모델 객체에서 메소드를 사용할 수 있도록 static( ) 함수를 호출하여 메소드를 정의하고 있습니다. 이 메소드 정의 부분 중에서 findById로 되어 있던 메소드를 findByEmail로 바꾸어서 이메일로 사용자 정보를 조회하도록 코드를 다음과 같이 수정합니다.

참조 파일 | PassportExample>/database/user_schema.js

```
……

// 모델 객체에서 사용할 수 있는 메소드 정의
UserSchema.static('findByEmail', function(email, callback) {
  return this.find({email : email}, callback);
});

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

……
```

findByEmail 메소드는 email 데이터로 데이터베이스에 저장된 객체를 조회합니다. 사용자 정보를 저장하기 위한 스키마를 모두 수정했다면 config.js 파일을 열고 스키마 정보를 등록하는 부분을 수정합니다.

참조 파일 | PassportExample>config.js

```
module.exports = {
  server_port : 3000,
  db_url : 'mongodb://localhost:27017/shopping',
  db_schemas : [
    {file : './user_schema', collection : 'users5', schemaName : 'UserSchema',
     modelName : 'UserModel'}
  ],
  route_info : [
  ]
}
```

db_schemas 배열 객체에 들어 있던 객체의 속성 중에서 collection 속성 값을 users3에서 users5로 수정합니다. 이렇게 하면 이전 스키마와 다른 새로운 users5 컬렉션이 만들어집니다. 라우팅 함수들을 등록하기 위해 만들 route_info 배열 객체는 아무 속성도 없는 빈 배열 객체로 만듭니다. 그다음 [routes] 폴더 안에서 user.js 파일을 삭제합니다.

이제 app.js 파일을 다시 열고 로컬 패스포트 인증을 처리할 함수를 등록합니다.

```
.....

var LocalStrategy = require('passport-local').Strategy;

// 패스포트 로그인 설정
passport.use('local-login', new LocalStrategy({
  usernameField : 'email',
  passwordField : 'password',
  passReqToCallback : true
}, function(req, email, password, done) {
  console.log('passport의 local-login 호출됨 : ' + email + ', ' + password);

  var database = app.get('database');
  database.UserModel.findOne({'email' : email}, function(err, user) {
    if(err) {return done(err);}

    // 등록된 사용자가 없는 경우
    if(!user) {
      console.log('계정이 일치하지 않음.');
      return done(null, false, req.flash('loginMessage', '등록된 계정이 없습니다.'));
    }

    // 비밀번호를 비교하여 맞지 않는 경우
    var authenticated = user.authenticate(password, user._doc.salt,
                                          user._doc.hashed_password);
    if (!authenticated) {
      console.log('비밀번호 일치하지 않음.');
      return done(null, false, req.flash('loginMessage', '비밀번호가 일치하지 않습니다.'));
    }

    // 정상인 경우
    console.log('계정과 비밀번호가 일치함.');
    return done(null, user);
  });

}));

.....
```

이 코드는 passport 모듈에서 로컬 사용자 인증에 사용하려고 만든 것이기 때문에 passport-local 모듈에서 제공하는 로컬 인증 방식을 사용하도록 설정되어 있습니다. passport 객체의 use( ) 메소드를 호출하면서 전달하는 파라미터는 두 개인데 첫 번째는 이름이고 두 번째는 인증 방식을 정의한 객체입니다.

첫 번째 파라미터에는 local-login을 입력하여 인증 처리를 위해 등록한 함수를 이 이름으로 구별할 수 있게 합니다. 인증 방식을 정의하는 객체는 LocalStrategy 객체의 인스턴스 객체이며, 사용자 아이디에 해당하는 usernameField는 email 칼럼, passwordField는 password 칼럼으로 지정하였습니다. 여기에서 email, password 칼럼은 데이터베이스에 만들어 둔 컬렉션의 칼럼들 중에서 선택된 것들입니다. 두 번째 파라미터로 전달되는 콜백 함수가 호출되면 데이터베이스의 email과 password 칼럼을 비교하여 정상적으로 인증되었는지를 알려 줍니다.

done( ) 메소드는 오류가 발생했을 때 첫 번째 파라미터만 전달하고, 정상적으로 인증되었을 때는 첫 번째 파라미터를 null로, 두 번째 파라미터에는 사용자 객체를 전달합니다. 만약 이메일 계정이나 비밀번호가 일치하지 않는다면 첫 번째 파라미터를 null, 두 번째 파라미터를 false, 세 번째 파라미터에는 flash( ) 함수를 사용해 메시지를 넣어 줍니다.

사용자 정보가 없는 사용자가 회원가입을 할 수 있도록 회원가입 기능도 만들어 보겠습니다. 사용자 인증 방식인 local-login의 아래쪽에 다음 코드를 입력합니다.

참조 파일 | PassportExample>app.js

```
……

// 패스포트 회원가입 설정
passport.use('local-signup', new LocalStrategy({
    usernameField : 'email',
    passwordField : 'password',
    passReqToCallback : true
  }, function(req, email, password, done) {
    var paramName = req.param('name');
    console.log('passport의 local-signup 호출됨 : ' + email + ', ' + password + ', '
              + paramName);

    // User.findOne이 blocking되므로 async 방식으로 변경할 수도 있음
    process.nextTick(function() {
      var database = app.get('database');
      database.UserModel.findOne({'email' : email}, function(err, user) {
        // 오류가 발생하면
        if(err) {
          return done(err);
        }
```

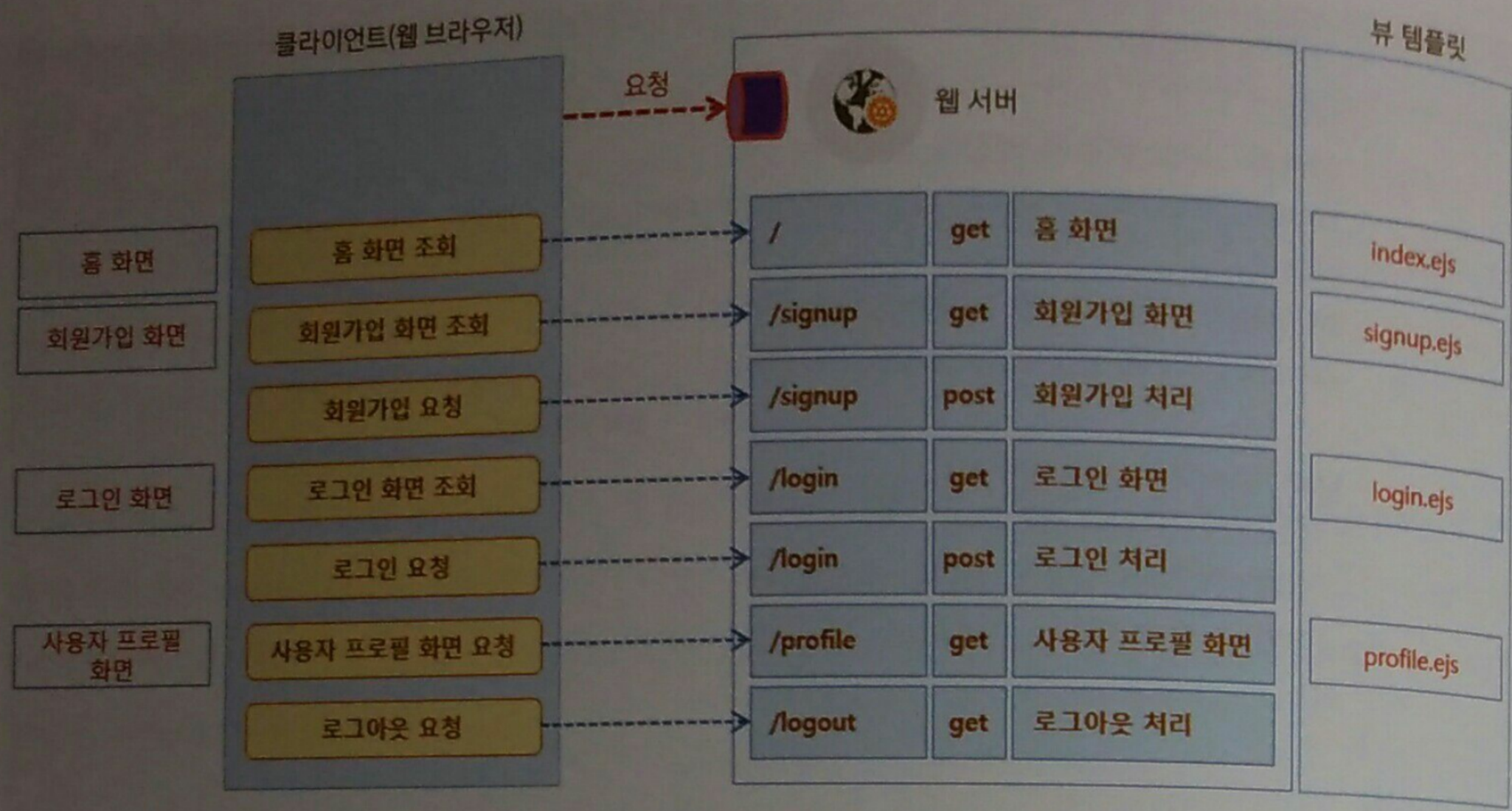
```
    // 기존에 이메일이 있다면
    if(user) {
      console.log('기존에 계정이 있음.');
      return done(null, false, req.flash('signupMessage', '계정이 이미 있습니다.'));
    } else {
      // 모델 인스턴스 객체 만들어 저장
      var user = new database.UserModel({'email' : email, 'password' : password,
                                         'name' : paramName});

      user.save(function(err) {
        if(err) {throw err;}
        console.log("사용자 데이터 추가함.");
        return done(null, user);
      });
    }
  });
});

});

……
```

local-signup이라는 이름으로 등록된 콜백 함수 안의 코드는 앞에서 입력한 local-login과 거의 비슷한 구조를 가지고 있습니다. 이 구조는 나중에 살펴볼 페이스북이나 구글을 사용한 인증 방식에서도 똑같이 사용되므로 코드를 복사한 후 일부만 수정하면 간단하게 입력할 수 있습니다. 회원가입을 위해 만든 이 코드는 사용자가 보내온 이메일, 비밀번호, 사용자 이름을 데이터베이스에 저장합니다. 등록된 이메일이 있는 사용자는 이미 계정이 있다는 오류 메시지를 전달하고, 사용자 정보가 없다면 새로운 모델 인스턴스 객체를 만든 후 저장합니다.

여기까지 진행하면 로컬 인증 방식을 사용하기 위한 패스포트 설정이 모두 끝납니다.

로그인과 회원가입 화면을 만들기 위한 라우팅 함수 등록하기

이제 사용자 로그인과 회원가입 화면을 위한 라우팅 함수를 만들고 웹 브라우저에서 보여 줄 웹 문서를 위해 뷰 템플릿도 만듭니다. 라우팅 함수와 뷰 템플릿이 여러 개이므로 전체 처리 과정을 다음 그림처럼 그려 보면 더 쉽게 이해할 수 있습니다.



▲ 사용자 로그인과 회원가입 처리 과정

클라이언트에서 사용할 수 있는 기능은 일곱 가지입니다. 각 기능별로 라우팅 함수를 만든 후 웹 서버에 등록합니다. 먼저 홈 화면을 조회하는 기능은 웹 브라우저에서 / 패스로 요청합니다. 홈 화면을 위해 만드는 뷰 템플릿의 이름은 index.ejs로 설정합니다.

사용자를 새로 등록할 때에는 회원가입 화면을 띄웁니다. 사용자가 회원가입 화면을 보고 싶다면 웹 브라우저에서 /signup 패스로 요청하면 됩니다. 사용자가 get 방식으로 요청하면 결과 웹 문서로 응답을 전달합니다. 회원가입 화면은 응답 웹 문서로 전달하여 보여 줄 것이므로 회원가입 화면을 위한 뷰 템플릿을 signup.ejs 파일로 만들 것입니다. /signup 패스로 요청하는 경우는 get 방식과 post 방식으로 나눌 수 있으며, post 방식으로 요청하는 경우에는 패스포트를 사용해 회원가입을 진행합니다.

회원가입 후 로그인을 하려고 할 때는 로그인 화면 조회를 요청합니다. 로그인 화면은 /login 패스로 하여 get 방식으로 요청하면 되고 뷰 템플릿은 login.ejs를 사용합니다. 패스포트를 사용해 로그인을 진행하려면 /login 패스로 하여 post 방식으로 요청합니다. 이미 등록된 사용자 프로필 정보를 보고 싶다면 /profile 패스로 정보를 요청하여 profile.ejs 템플릿으로 만든 웹 문서를 받아 볼 수 있습니다.

먼저 패스포트와 관련된 라우팅 함수를 등록해 보겠습니다. 위의 처리 과정에서 정리한 것처럼 라우팅 함수는 크게 두 가지로 나누어 생각해야 합니다.

첫 번째는 get 방식을 사용해 단순히 웹 문서를 조회하는 사용자 요청입니다. 홈 화면을 조회하는 / 패스, 로그인 화면을 조회하는 /login 패스, 회원가입 화면을 조회하는 /signup 패스, 사용자 프로필을 조회하는 /profile 패스가 있습니다.

두 번째는 post 방식을 사용해 사용자 인증이나 회원가입을 진행하도록 요청하는 경우입니다. 패스포트를 사용해 사용자 로그인을 진행하는 /login 패스와 패스포트를 사용해 회원가입을 진행하는 /signup 패스가 있습니다.

새로 추가할 사용자 요청 패스만 다시 정리해 보면 다음과 같습니다.

요청 패스	요청 방식	설명
/	get	홈 화면 조회
/login	get	로그인 화면 조회
/login	post	패스포트로 사용자 인증을 처리하는 함수 호출
/signup	get	회원가입 화면 조회
/signup	post	패스포트로 회원 가입을 처리하는 함수 호출
/profile	get	사용자 프로필 화면 조회
/logout	get	로그아웃을 처리하는 함수 호출

[표] 새로 추가할 사용자 요청 패스

/profile 패스와 /logout 패스는 모두 get 방식으로 요청하며, 각 사용자 프로필 화면을 조회하는 기능과 로그아웃 기능을 수행합니다. 위의 라우팅 함수들은 하나씩 만들어 라우터 미들웨어를 사용하는 코드 아래쪽에 추가합니다. 다음은 홈 화면을 보여 주는 / 패스를 라우팅하는 코드입니다.

참조 파일 | PassportExample>app.js

```
.....

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

// 홈 화면 - 로그인 링크
app.get('/', function(req, res) {
  console.log('/ 패스 요청됨.');
  res.render('index.ejs');
});

.....
```

이 함수는 단순히 [views] 폴더 안에 있는 index.ejs 뷰 템플릿으로 응답 웹 문서를 만든 후 응답을 보내 주는 역할만 합니다. 다음은 get 방식으로 요청하는 /login 패스와 post 방식으로 요청하는 /login 패스를 라우팅하는 코드입니다.

참조 파일 | PassportExample>app.js

```
.....

// 로그인 폼 링크
app.get('/login', function(req, res) {
  console.log('/login 패스 요청됨.');
  res.render('login.ejs', {message : req.flash('loginMessage')});
});

app.post('/login', passport.authenticate('local-login', {
    successRedirect : '/profile',
    failureRedirect : '/login',
    failureFlash : true
}));

.....
```

get 방식으로 요청할 때는 단순히 login.ejs 뷰 템플릿으로 결과 웹 문서를 만들어 응답합니다. login.ejs 뷰 템플릿을 사용해 결과 웹 문서를 만들 때는 connect-flash 모듈을 사용해 전달받은 메시지를 message 속성으로 전달합니다.

post 방식으로 요청할 때는 passport 객체의 **authenticate( ) 메소드**를 호출하여 패스포트 모듈에서 사용자 인증을 처리하도록 넘겨줍니다. 앞에서 사용자 인증 방식을 등록할 때 local-login이라는 이름으로 등록했으므로 첫 번째 파라미터로는 그 이름을 전달합니다. 두 번째 파라미터로 전달되는 객체 안에는 사용자 인증이 성공했을 때와 실패했을 때 어떤 패스로 이동할 것인지를 지정하는 속성들을 넣어 둡니다. 여기에서는 사용자 인증이 성공했을 때 사용자 정보를 표시해 주는 /profile 패스로 이동하고, 사용자 인증이 실패했을 때는 재로그인을 위해 /login 패스로 이동합니다.

이제 회원가입 화면을 위한 라우팅 함수를 만들어 추가합니다.

참조 파일 | PassportExample>app.js

```
……

// 회원가입 폼 링크
app.get('/signup', function(req, res) {
  console.log('/signup 패스 요청됨.');
  res.render('signup.ejs', {message : req.flash('signupMessage')});
});

app.post('/signup', passport.authenticate('local-signup', {
    successRedirect : '/profile',
    failureRedirect : '/signup',
    failureFlash : true
}));

……
```

이 코드는 앞에서 살펴본 /login 패스를 처리할 때와 거의 같습니다. get 방식으로 요청했을 때는 signup.ejs 뷰 템플릿으로 응답을 보내고 post 방식으로 요청했을 때는 패스포트의 authenticate( ) 메소드를 호출하면서 local-signup을 파라미터로 전달합니다. 특히 passport 모듈의 authenticate( ) 함수를 호출하는 부분은 앞에서 로그인 기능을 만드는 코드를 넣을 때와 거의 비슷합니다. 두 경우의 차이점은 파라미터로 local-login을 넣었는지 아니면 local-signup을 넣었는지입니다. 이 파라미터는 앞에서 인증 함수를 만들 때 여러분이 직접 지정한 이름입니다.

그다음은 사용자 프로필을 보여주기 위한 라우팅 함수를 만들어 추가합니다.

참조 파일 | PassportExample>app.js

```
……

// 프로필 링크 - 먼저 로그인 여부를 확인할 수 있도록 isLoggedIn 미들웨어 실행
app.get('/profile', isLoggedIn, function(req, res) {
  console.log('/profile 패스 요청됨.');
  console.dir(req.user);

  if(Array.isArray(req.user)) {
    res.render('profile.ejs', {user : req.user[0]._doc});
  } else {
    res.render('profile.ejs', {user : req.user});
  }
});

……
```

여기에서는 app 객체의 get( ) 메소드를 호출하면서 지정한 콜백 함수로 라우팅하기 전에 isLoggedIn 미들웨어를 먼저 실행하도록 설정되어 있습니다. isLoggedIn 미들웨어는 get( ) 메소드를 호출할 때 두 번째 파라미터로 전달하며 이 미들웨어는 그 아래에 정의합니다. profile.ejs 템플릿으로 결과 웹 문서를 만들 때는 req.user 객체를 사용하는데, 이 객체가 배열 객체인지 아닌지를 if문의 조건 부분에서 확인한 후 res.render( ) 메소드를 호출합니다.

다음은 isLogginIn 미들웨어의 코드입니다. 사용자가 인증된 상태이면 그다음 미들웨어로 넘어가고 그렇지 않으면 홈 화면으로 이동하도록 다음과 같이 코드를 입력합니다.

참조 파일 | PassportExample>app.js

```
.....

// 로그인 여부를 알 수 있는 미들웨어
function isLoggedIn(req, res, next) {
  console.log('isLoggedIn 미들웨어 호출됨.');

  if (req.isAuthenticated()) {
    return next();
  }

  res.redirect('/');
}

.....
```

req 요청 객체에는 isAuthenticated( ) 메소드가 만들어져 있어 사용자 인증 여부를 간단하게 확인할 수 있습니다. 다음은 로그아웃 기능을 만들기 위한 라우팅 코드입니다.

참조 파일 | PassportExample>app.js

```
.....

// 로그아웃
app.get('/logout', function(req, res) {
  console.log('/logout 패스 요청됨.');
  req.logout();
  res.redirect('/');
});

.....
```

로그아웃하려면 요청 객체에 들어 있는 logout( ) 메소드를 호출하면 됩니다. 로그아웃 이후에는 응답 객체의 redirect( ) 메소드를 호출하여 홈 화면으로 이동합니다.

로그인과 회원가입 화면을 만들기 위한 뷰 템플릿 만들기

이제 웹 문서의 기본 틀인 뷰 템플릿 파일을 만들 차례입니다. 여기에서는 뷰 템플릿을 **부트스트랩**(Bootstrap)을 사용하여 구성합니다. 부트스트랩은 모던한 느낌의 화면 스타일을 손쉽게 만들 수 있는 UI 라이브러리입니다. 부트스트랩에 대해 더 자세한 정보를 알고 싶다면 다음 사이트를 참조하기 바랍니다.

▶ http://getbootstrap.com/

여기에서는 부트스트랩 라이브러리를 사용하지만 Semantic UI 라이브러리를 사용해서 뷰 템플릿을 만들어도 됩니다. 클라이언트에 응답을 보내주기 위해 만들어야 할 뷰 템플릿 목록은 다음과 같습니다.

템플릿 이름	설명
index.ejs	홈 화면을 구성하는 뷰 템플릿
login.ejs	로그인 화면을 구성하는 뷰 템플릿
signup.ejs	회원가입 화면을 구성하는 뷰 템플릿
profile.ejs	사용자 프로필 화면을 구성하는 뷰 템플릿

[표] 새로 추가할 뷰 템플릿

사용자는 웹 브라우저를 열었을 때 먼저 홈 화면을 보게 됩니다. 홈 화면에는 링크가 있어 사용자 계정이 있을 때는 로그인 화면으로 이동하고 사용자 계정이 없을 때는 회원가입 화면으로 이동할 수 있습니다. 그리고 로그인 후에는 사용자 프로필 화면을 조회할 수 있습니다.

다음은 부트스트랩으로 만든 홈 화면의 코드입니다. [views] 폴더 안에 index.ejs 파일을 만들고 다음 코드를 입력합니다.

참조 파일 | PassportExample>/views/index.ejs

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

```
<title>홈 화면</title>

<link rel = "stylesheet"
    href = "//netdna.bootstrapcdn.com/bootstrap/3.0.2/css/bootstrap.min.css">
<link rel = "stylesheet"
    href = "//netdna.bootstrapcdn.com/font-awesome/4.0.3/css/font-awesome.min.css">

<style>
  body{padding-top : 80px;}

  @font-face{
    font-family : 'Nanum Gothic'; font-style : normal; font-weight : 400;
    src : url('../public/NanumGothic-Regular.eot');
    src : local('NanumGothic'),
      url('../public/NanumGothic-Regular.eot?#iefix') format('embedded-opentype'),
      url('../public/NanumGothic-Regular.woff2') format('woff2'),
      url('../public/NanumGothic-Regular.woff') format('woff'),
      url('../public/NanumGothic-Regular.ttf') format('truetype');
    }

    h1, p, a{font-family : 'Nanum Gothic'}
  </style>

</head>
<body>
  <div class = "container">
    <div class = "jumbotron text-center">
      <h1><span class = "fa fa-shopping-cart"></span> 쇼핑몰 홈</h1>
      <br>
      <p>로그인하세요.<p>
      <p>계정이 없으시면 회원가입하세요.</p>
      <br>
      <a href = "/login" class = "btn btn-default">
        <span class = "fa fa-user"></span>로그인
      </a>
      <a href = "/signup" class = "btn btn-default">
        <span class = "fa fa-user"></span>회원가입
      </a>
    </div>
  </div>
</body>
</html>
```

〈head〉 태그 안에는 모바일 단말의 가로와 세로 화면 크기에 맞추는 〈meta〉 태그가 추가되어 있습니다. 〈meta〉 태그 안의 name 속성 값을 viewport로 설정하고 단말의 가로·세로 크기를 지정했습니다. 그 아래에는 부트스트랩을 사용하기 위해 css 파일 두 개를 링크합니다. 첫 번째 bootstrap.min.css 파일은 부트스트랩을 사용하기 위한 기본 CSS 파일이며, 두 번째 font-awesome.min.css 파일은 예쁜 모양의 서체를 사용하기 위해 링크한 것입니다.

홈 화면에는 안내글이 표시되며 아래쪽에 [로그인] 버튼과 [회원가입] 버튼이 있어서 필요할 때 로그인을 하거나 회원가입을 할 수 있습니다.

로그인 화면을 위한 뷰 템플릿 화면을 login.ejs 파일로 만들어 보겠습니다.

참조 파일 | PassportExample〉/views/login.ejs

```
……

<div class = "container">

  <div class = "col-sm-6 col-sm-offset-3">
    <h1><span class = "fa fa-sign-in"></span> 로그인</h1>

    <!— 인증 처리 후 메시지가 있으면 메시지 표시 —>
    <% if(message.length > 0) { %>
      <div class = "alert alert-danger"><% =  message %></div>
    <% } %>

    <form action = "/login" method = "post" >
      <div class = "form-group">
        <label>이메일</label>
        <input type = "text" class = "form-control" name = "email">
      </div>
      <div class = "form-group">
        <label>비밀번호</label>
        <input type = "password" class = "form-control" name = "password">
      </div>

      <button type = "submit" class = "btn btn-warning btn-lg">로그인</button>
    </form>

    <hr>

    <p>계정이 없으세요? <a href = "/signup">회원가입하기</a></p>
    <p><a href = "/">홈으로</a>.</p>
```

```
  </div>

</div>

……
```

로그인 화면에는 이메일과 비밀번호를 입력할 때 필요한 입력 상자와 함께 [로그인] 버튼을 보여 줍니다. 그 아래에는 계정이 없을 때 회원가입 화면으로 이동할 수 있도록 /signup 패스 링크를 추가합니다. 여기에서 주의 깊게 볼 부분은 message 속성 값을 표시하는 부분입니다. 이 message 속성은 사용자가 로그인을 할 때 아이디가 없거나 비밀번호가 일치하지 않는 문제가 발생하면 로그인 처리 함수에서 connect-flash 모듈로 메시지를 넣어 준 것입니다. 이 부분을 보면 앞에서 사용한 connect-flash 모듈의 용도를 잘 이해할 수 있습니다.

회원가입 화면을 위한 뷰 템플릿은 signup.ejs 파일로 만들고 다음 코드를 입력합니다.

참조 파일 | PassportExample>/views/signup.ejs

```
……

  <div class = "container">

    <div class = "col-sm-6 col-sm-offset-3">

      <h1><span class = "fa fa-sign-in"></span> 회원가입</h1>

      <!— 인증 처리 후 메시지가 있으면 메시지 표시 —>
      <% if(message.length > 0) { %>
        <div class = "alert alert-danger">< % = message %></div>
      <% } %>

      <form action = "/signup" method = "post" >
        <div class = "form-group">
          <label>이메일</label>
        <input type = "text" class = "form-control" name = "email">
      </div>
      <div class = "form-group">
        <label>비밀번호</label>
        <input type = "password" class = "form-control" name = "password">
      </div>
      <div class = "form-group">
```

```
    <label>비밀번호</label>
    <input type = "password" class = "form-control" name = "password">
  </div>
  <div class = "form-group">
    <label>별명</label>
    <input type = "text" class = "form-control" name = "name">
  </div>

  <button type = "submit" class = "btn btn-warning btn-lg">회원가입</button>
</form>

  <hr>

  <p>이미 계정이 있으신가요? <a href = "/login">로그인하기</a></p>
  <p><a href = "/">홈으로</a>.</p>

 </div>

</div>

……
```

회원가입 화면을 위한 뷰 템플릿의 구성은 로그인 화면 뷰 템플릿과 거의 같습니다. 회원가입을 위해서는 이메일과 비밀번호, 그리고 별명을 입력받습니다. [회원가입] 버튼을 누르면 /signup 패스로 하여 post 방식으로 요청합니다. 이 파일에서도 회원가입을 할 때 문제가 발생하면 message 속성으로 그 내용을 전달받아 화면에 표시합니다.

다음은 로그인 후에 확인할 수 있는 프로필 화면을 구성하는 뷰 템플릿입니다.

참조 파일 | PassportExample>/views/profile.ejs

```
……

<div class = "container">

  <div class = "page-header text-center">
    <h1><span class = "fa fa-anchor"></span> 사용자 프로필</h1>
    <a href = "/logout" class = "btn btn-default btn-sm">로그아웃</a>
  </div>
  <br>
  <div class = "row">
```

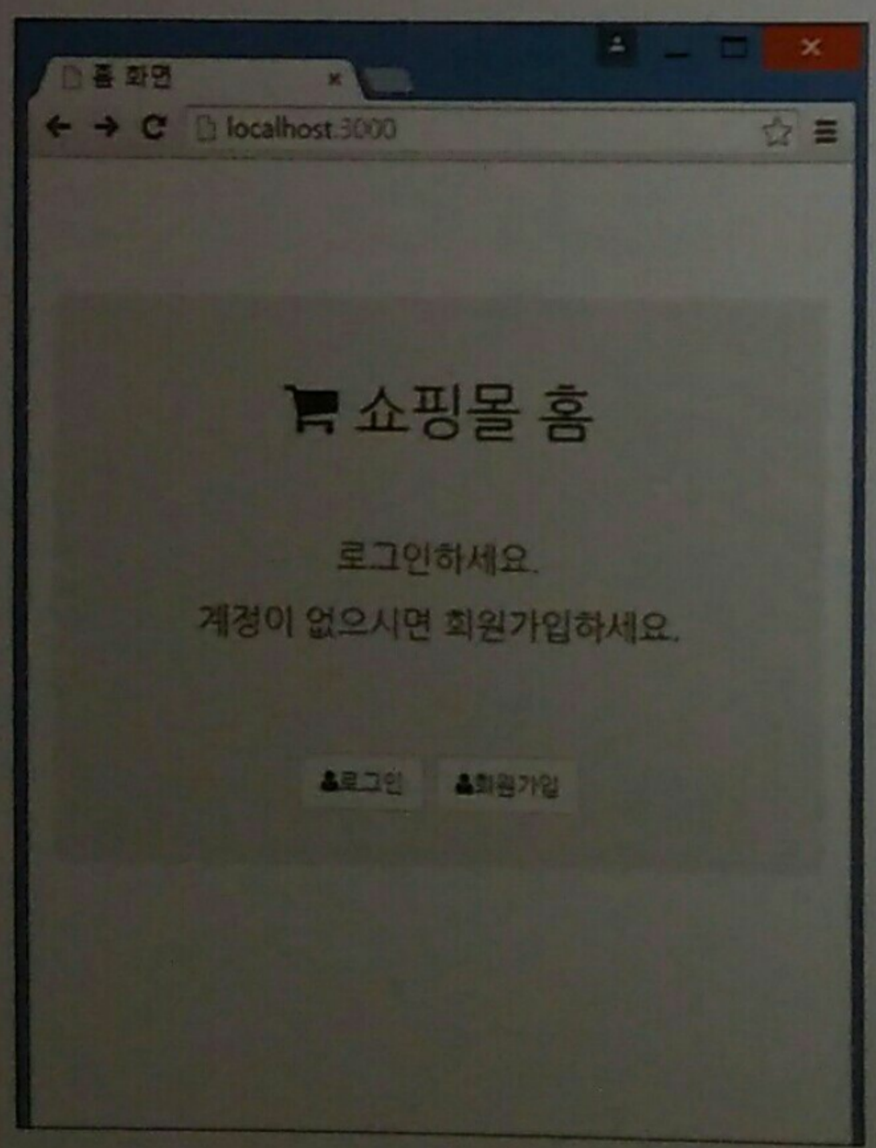
```
<div class = "col-sm-6">
  <div class = "well">
    <h3><span class = "fa fa-user"></span> 로컬 프로필 정보</h3>
    <br>
    <p>
    <strong>이름</strong> : <% =  user.name %><br><br>
    <strong>이메일</strong> : <% =  user.email %>
    </p>
    <br>
  </div>
</div>

</div>

</div>

.....
```

프로필 화면에서는 사용자의 이메일과 별명을 표시하고 위쪽에는 [로그아웃] 버튼을 배치합니다.

로그인과 회원가입 기능 실행하여 확인하기

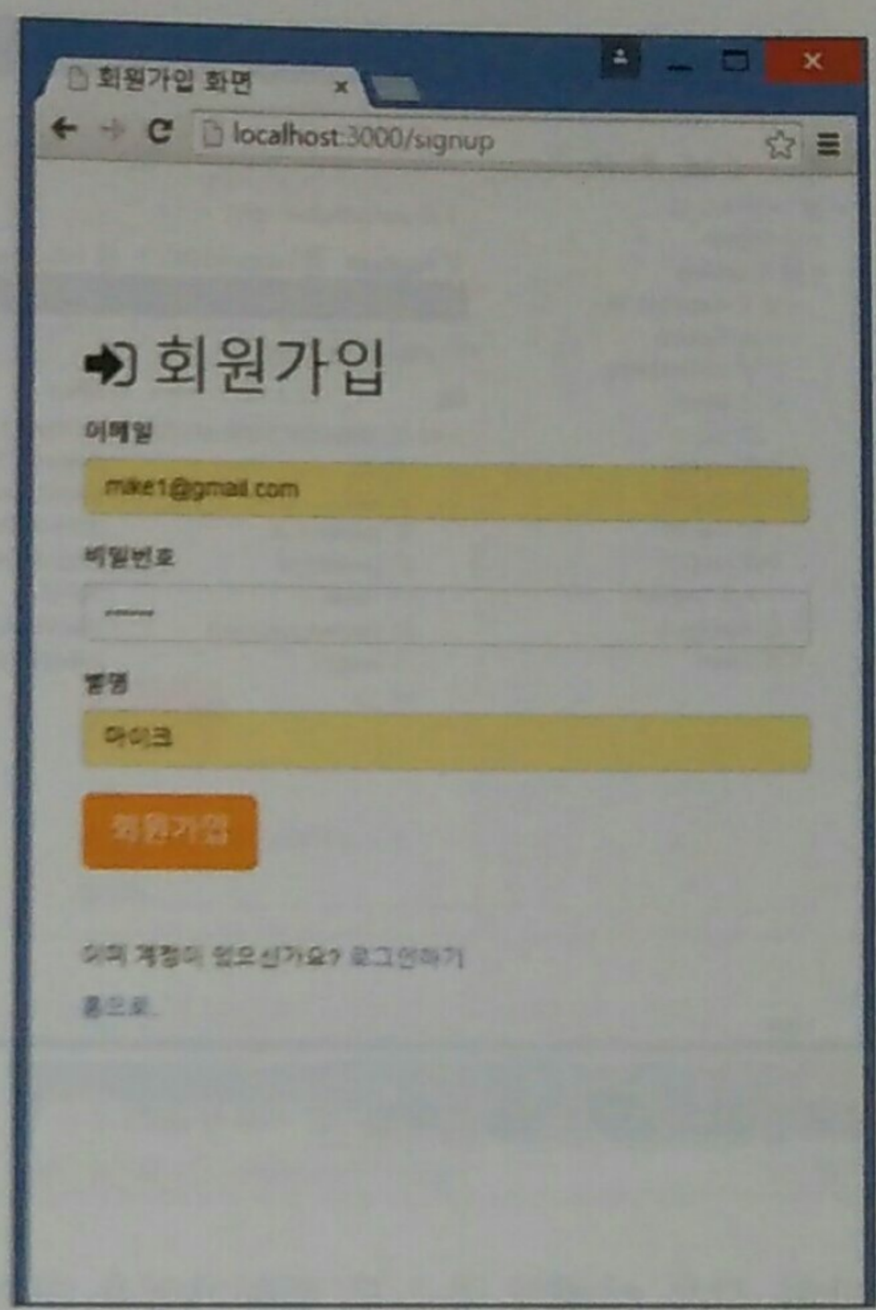
app.js 파일을 실행한 후 웹 브라우저에서 다음 URL 주소를 입력하면 홈 화면이 나타납니다.

▶ http://localhost:3000/



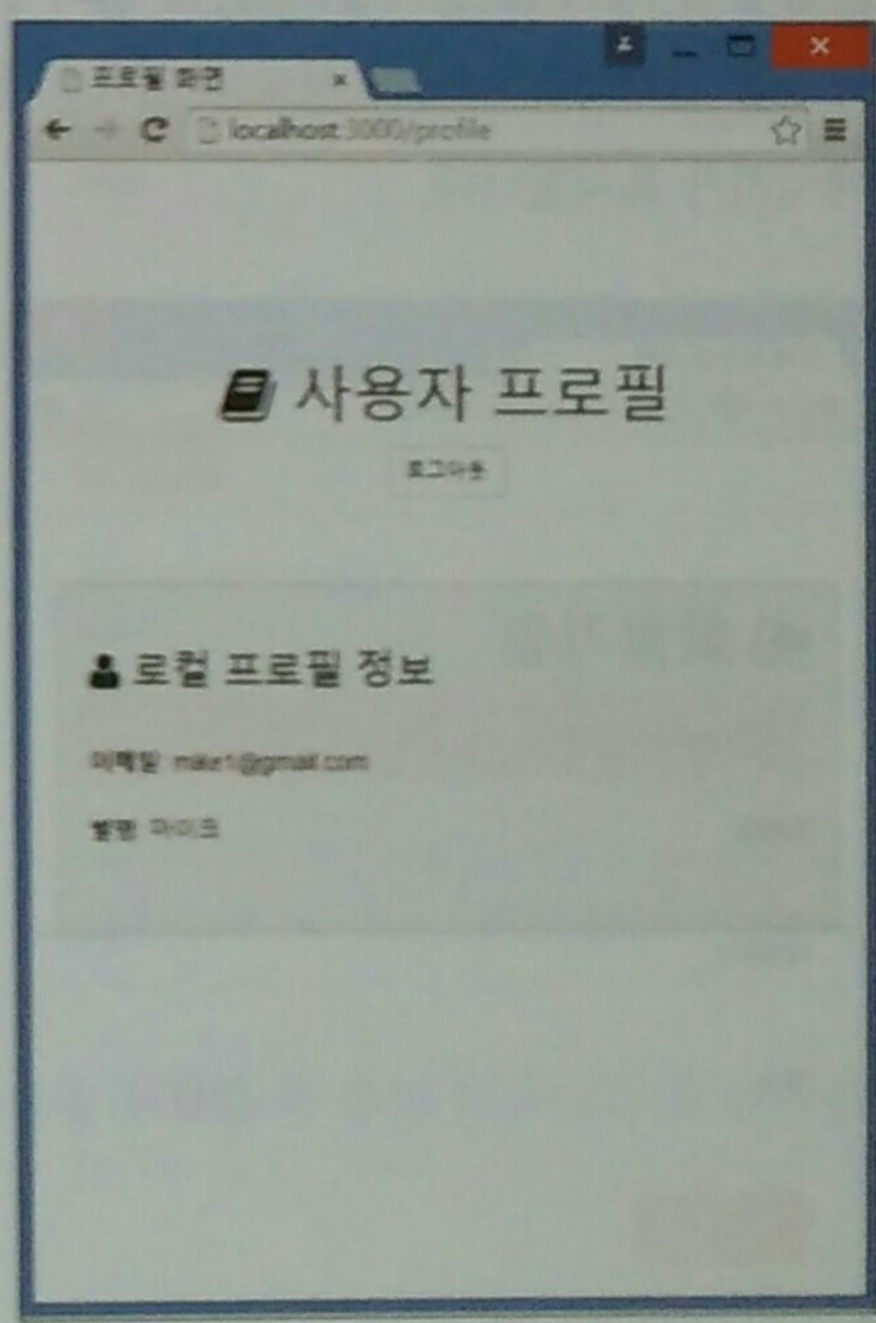
◀ 홈 화면

부트스트랩을 사용하여 뷰 템플릿을 만들었기 때문에 웹 브라우저의 크기에 맞추어 적당한 위치에 글자나 버튼이 배치되었습니다. 처음 실행했을 때는 웹 서버의 데이터베이스에 사용자가 없으므로 [회원가입] 버튼을 눌러 회원가입을 진행합니다. 다음은 [회원가입] 버튼을 눌러 회원가입 화면으로 이동했을 때의 화면입니다.



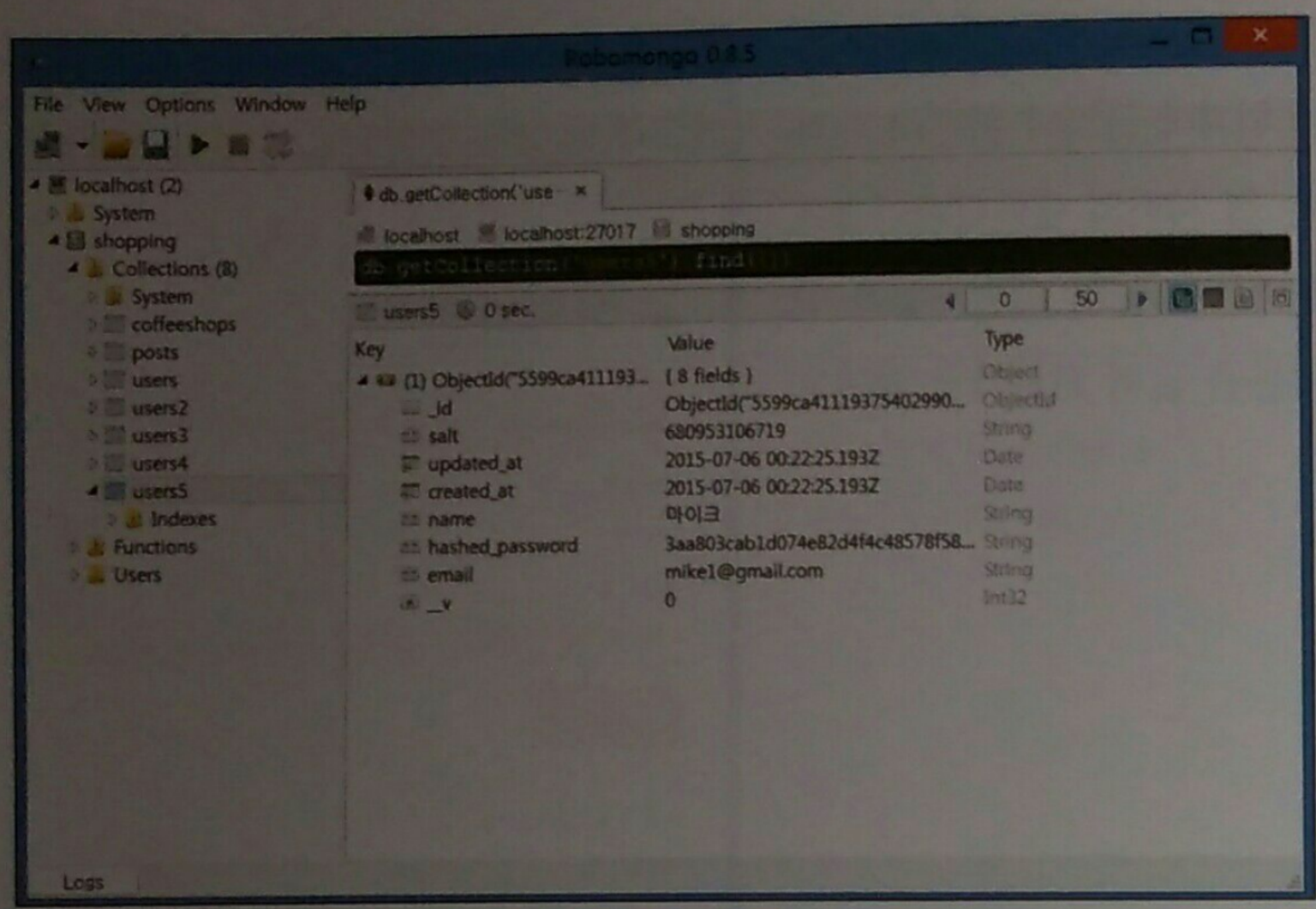
회원가입 화면▶

회원가입 화면으로 이동하면 웹 브라우저의 주소 입력란에 'localhost:3000/signup'이라고 표시된 것을 볼 수 있습니다. 이것은 서버에서 /signup 패스로 변경했기 때문입니다. 회원가입 화면에서는 이메일과 비밀번호, 그리고 별명을 입력할 수 있는 입력 상자를 볼 수 있습니다. 그 아래쪽에는 이미 계정이 있어 회원가입이 필요 없을 때 다시 로그인 화면으로 돌아갈 수 있도록 '로그인하기' 링크와 함께 홈 화면으로 이동할 수 있는 '홈으로' 링크가 함께 표시됩니다. 이메일과 비밀번호, 그리고 별명을 입력한 후 [회원가입] 버튼을 누릅니다. 회원가입에 성공하면 사용자 프로필 화면이 나타납니다.



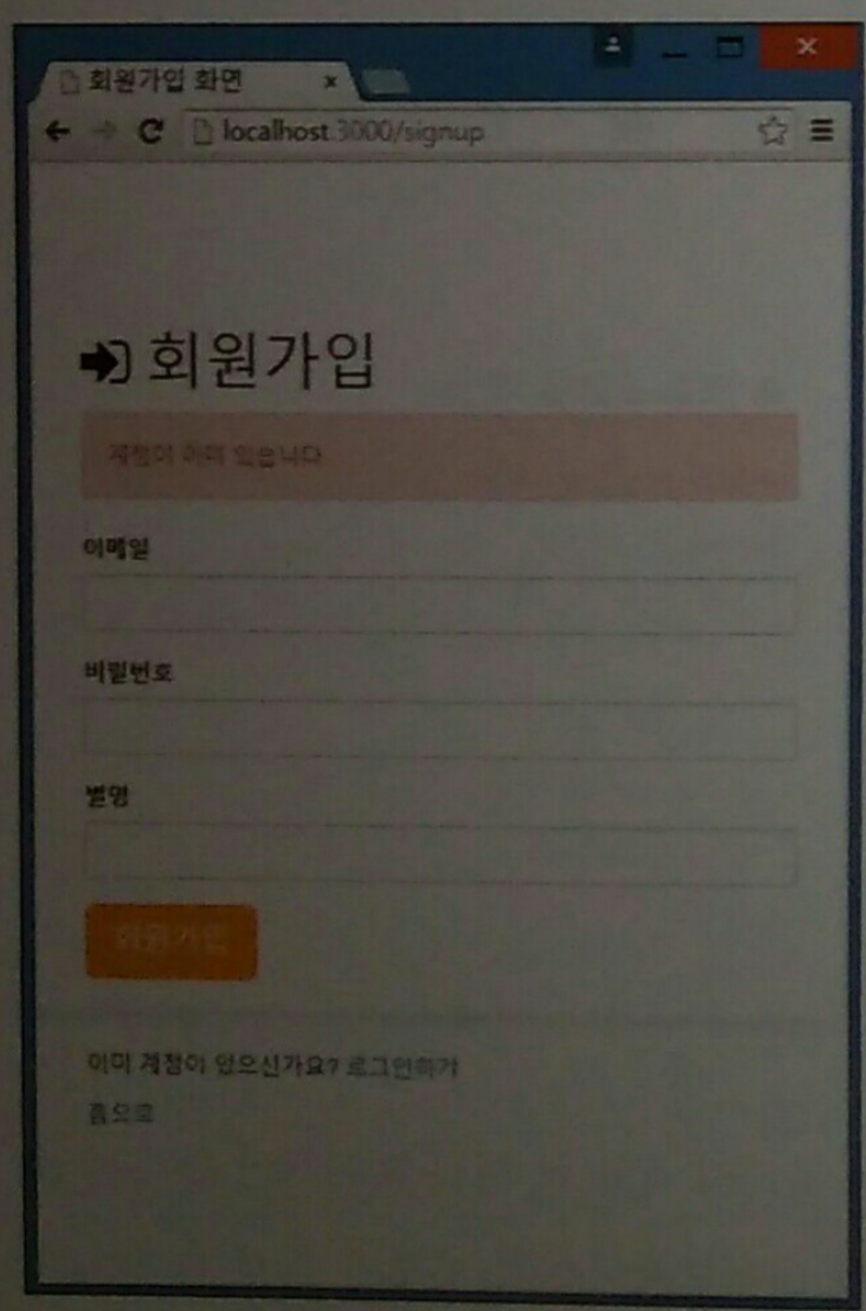
사용자 프로필 화면▶

사용자 프로필 화면에서는 현재 사용자의 이메일과 별명을 보여 줍니다. 이렇게 해서 회원가입 과정이 패스포트 모듈을 통해 이루어지며 회원가입에 성공했다면 서버의 데이터베이스에 사용자 정보가 추가된다는 것을 알 수 있었습니다. 실제로 데이터베이스에 추가된 정보를 확인하고 싶다면 Robomongo 프로그램으로 조회할 수 있습니다. Robomongo 프로그램을 실행한 후 데이터베이스의 users5 컬렉션을 더블클릭하면 새로 추가한 계정 정보가 표시됩니다.



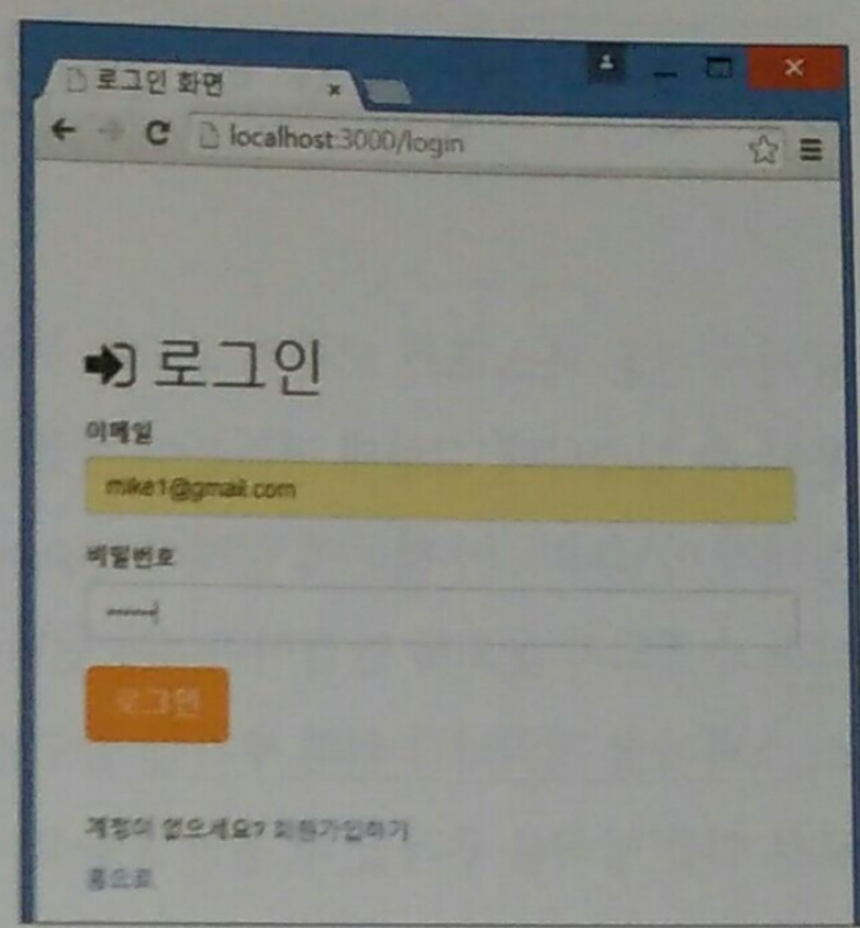
▲ 회원가입으로 데이터베이스에 추가된 사용자 정보

만약 같은 이메일 주소로 회원가입을 하려고 하면 어떻게 될까요? [로그아웃] 버튼을 눌러 홈 화면으로 이동한 후 다시 [회원가입] 버튼을 눌러 회원가입 화면으로 이동합니다. 이메일 입력란에 등록한 이메일 계정을 입력하고 나머지 정보도 입력한 후 [회원가입] 버튼을 누르면 '계정이 이미 있습니다.'라는 오류 메시지가 표시됩니다.



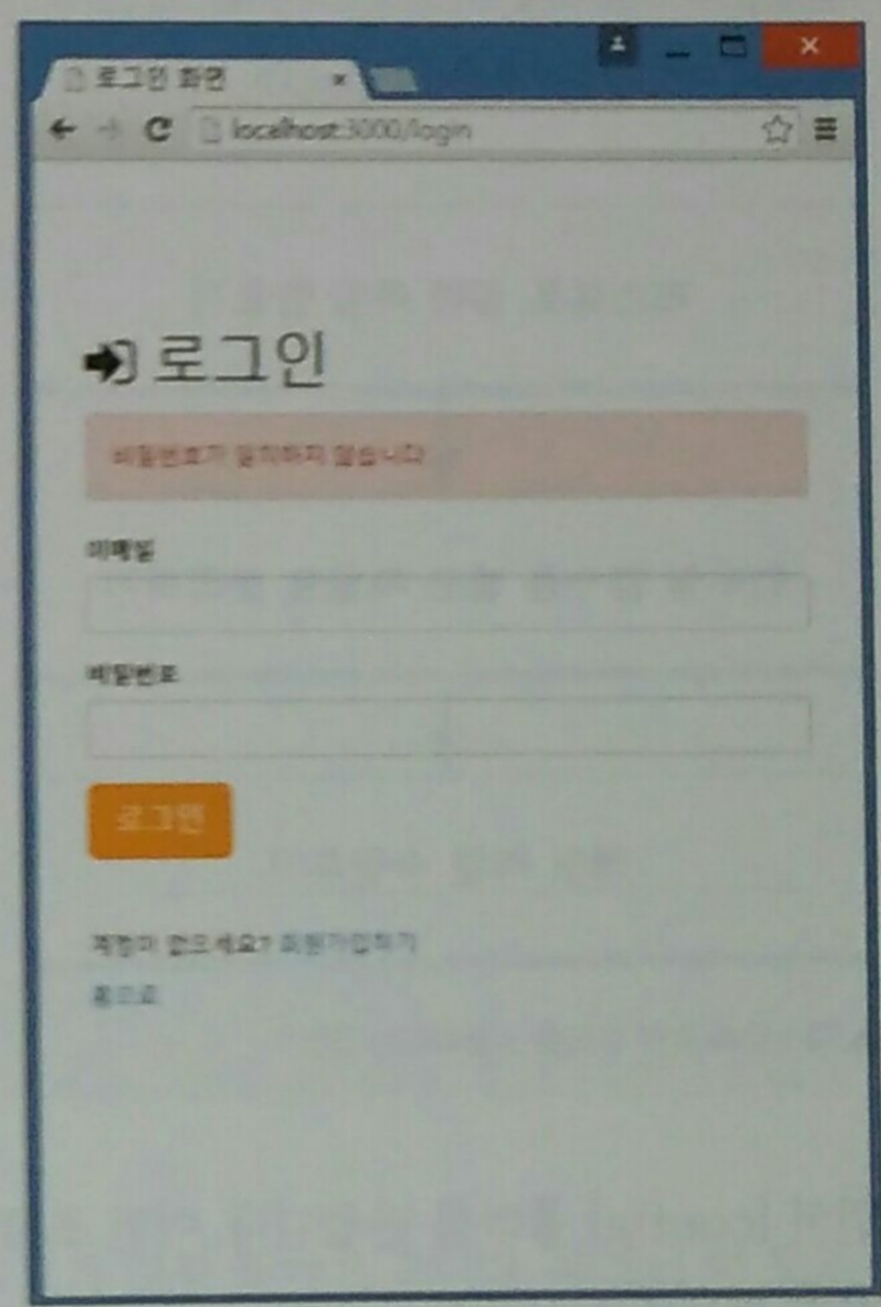
▲ 회원가입 화면에 이미 있는 이메일 계정을 다시 사용하는 경우

이제 화면 하단에 있는 '로그인하기' 링크를 눌러 로그인 화면으로 이동합니다.



로그인 화면▶

회원가입에 사용한 이메일 주소와 비밀번호를 정확히 입력한 후 [로그인] 버튼을 누르면 사용자 인증 후 프로필 화면으로 이동합니다. 프로필 화면의 구성은 앞에서 살펴본 것과 같습니다. 만약 로그인 화면에서 이메일 계정을 잘못 입력했거나 비밀번호를 잘못 입력한 경우에는 오류 메시지가 화면에 표시됩니다.



로그인할 때 비밀번호를 잘못 입력한 경우▶

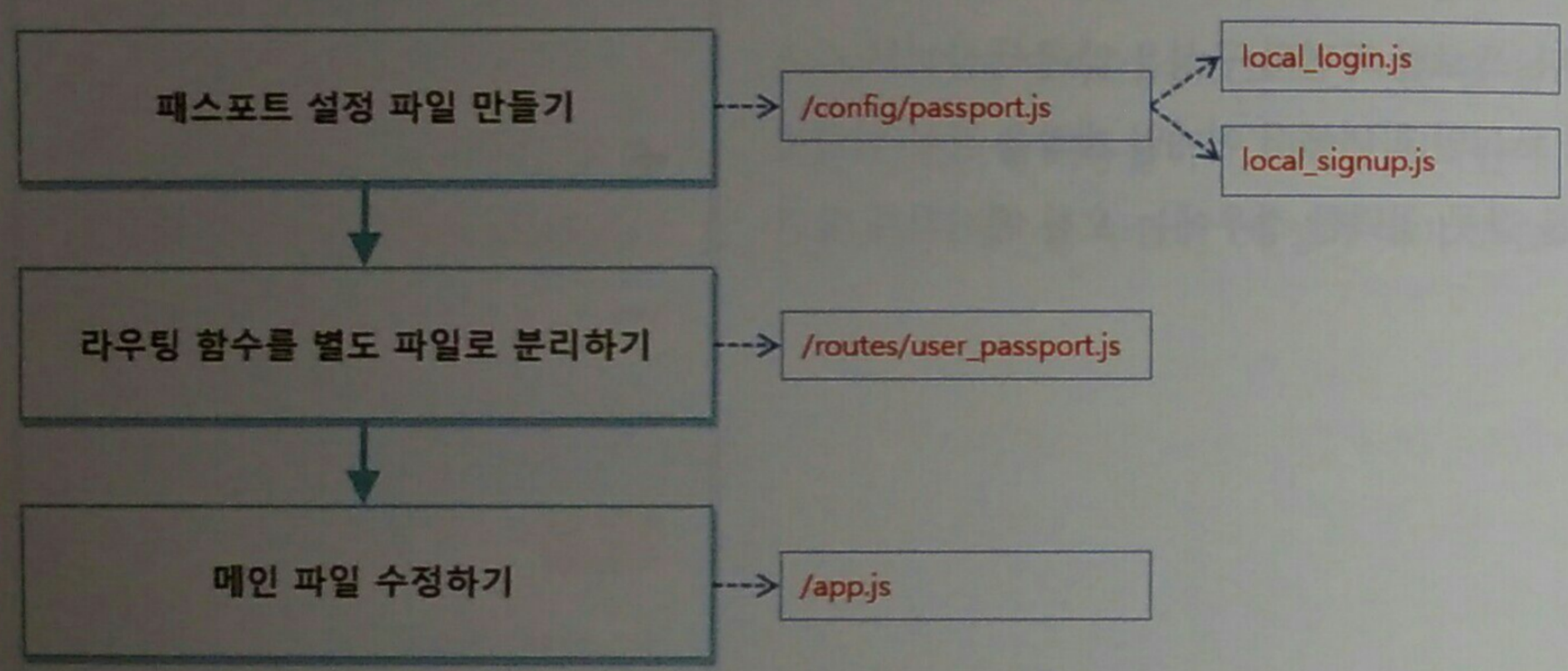
프로필 화면에서 [로그아웃] 버튼을 눌렀을 때는 로그아웃 과정을 진행한 후 항상 홈 화면으로 돌아옵니다.

지금까지 패스포트 모듈로 로그인과 회원가입 기능을 만들어 보았습니다. 패스포트 모듈을 사용하는 데 필요한 코드만큼이나 뷰 템플릿 코드를 많이 입력해야 한다는 것 그리고 패스포트 모듈 설정에도 생각보다 많은 코드를 입력해야 한다는 것을 알 수 있었습니다. 그러나 지금까지 만든 코드는 패스포트 모듈을 사용할 때 입력하는 전형적인 코드이므로 이 코드를 그대로 복사하여 회원가입 기능을 만든 후 필요할 때마다 코드를 부분적으로 수정해서 사용하면 됩니다.

09-2 | 패스포트 관련 코드를 모듈화하기

앞에서 만든 패스포트 모듈의 코드는 부분적으로 수정하면 다양한 사용자 인증 방식으로 변경하여 사용할 수 있습니다. 그런데 여러분이 직접 추가한 함수들은 app.js 파일에 들어 있으므로 아직 모듈화된 상태가 아닙니다. 이 때문에 앞에서 app.js 파일에 입력한 코드를 모듈화하여 app.js 메인 파일이 나중에라도 수정되지 않도록 만들어주어야 합니다. 왜냐하면 패스포트 설정이나 사용자 인증에서 데이터베이스 스키마를 정의하기 위해 추가한 함수들을 모두 모듈화하면, 인증 방식을 부분적으로 수정하거나 새로운 인증 방식을 추가할 때 훨씬 효율적이기 때문입니다.

PassportExample 프로젝트를 복사하여 PassportExample2 프로젝트를 만듭니다. 여기에서는 다음과 같은 과정을 거칠 것입니다.



▲ 패스포트 관련 코드를 모듈화하는 과정

먼저 [config] 폴더를 만듭니다. 어떤 프로젝트든지 기본 설정, 사용자 인증 설정, 데이터베이스 설정 등 설정 정보를 저장해두어야 합니다. 앞으로 설정에 관한 내용은 모두 이 폴더에 저장하겠습니다. 그 안에 passport.js 파일을 만들고 패스포트 설정 관련 코드를 입력합니다. 패스포트 설정은 어떤 인증 방식을 사용하는가에 따라 달라지므로 로컬 인증 방식을 설정하는 코드는 [passport] 폴더 안에 local_login.js 파일을 만든 후 그 파일 안에 넣어 둡니다. passport.js 파일에는 다음과 같이 serializeUser( )와 deserializeUser( ) 메소드를 호출하는 코드와 함께 [passport] 폴더에 만들 모듈 파일을 불러와 사용하는 코드를 입력합니다.

참조 파일 | PassportExample2>/config/passport.js

```
var local_login = require('./passport/local_login');
var local_signup = require('./passport/local_signup');

module.exports = function (app, passport) {
  console.log('config/passport 호출됨.');

  // 사용자 인증에 성공했을 때 호출
  passport.serializeUser(function(user, done) {
    console.log('serializeUser() 호출됨.');
    console.dir(user);

    done(null, user);
  });

  // 사용자 인증 이후 사용자 요청이 있을 때마다 호출
  passport.deserializeUser(function(user, done) {
    console.log('deserializeUser() 호출됨.');
    console.dir(user);

    done(null, user);
  });

  // 인증 방식 설정
  passport.use('local-login', local_login);
  passport.use('local-signup', local_signup);

};
```

module.exports에 할당된 함수 안에는 app.js 파일에 있던 코드가 일부 들어 있습니다. 따라서 이 모듈 파일 안에 들어 있는 코드 중에서 app 객체나 passport 객체를 참조해야 할 때가 생깁니다. 이 때문에 이 함수를 호출할 때 app 객체와 passport 객체를 파라미터로 전달해야 app.js 파일에 있던 코드를 그대로 옮겨 와도 문제없이 실행됩니다. 그 아랫부분에서는 passport 객체의 use( ) 메소드를 호출하여 local-login과 local-signup 인증 방식을 추가합니다. 이 코드도 app.js 파일에 있던 코드를 그대로 가져온 것이지만 콜백 함수는 local_login.js와 local_signup.js 파일에 각각 분리하여 저장합니다. local_login.js 파일과 local_signup.js 파일에는 app.js 파일에 있던 인증 방식 설정 관련 코드를 옮겨와 입력합니다.

참조 파일 | PassportExample2>/config/passport/local_login.js

```
var LocalStrategy = require('passport-local').Strategy;

module.exports = new LocalStrategy({
    usernameField : 'email',
    passwordField : 'password',
    passReqToCallback : true
  }, function(req, email, password, done) {
    console.log('passport의 local-login 호출됨 : ' + email + ', ' + password);

.....
```

passport-local 모듈의 Strategy 객체를 사용해 new 연산자로 만든 인스턴스 객체는 module.exports에 그대로 할당되었습니다. 이렇게 할당되었기 때문에 passport.js 파일에서 이 모듈 파일을 불러와 그대로 use( ) 메소드의 파라미터로 전달할 수 있습니다.

회원가입을 위해 만든 local_signup.js 파일에도 설정 관련 코드를 옮겨와 입력합니다.

참조 파일 | PassportExample2>/config/passport/local_signup.js

```
var LocalStrategy = require('passport-local').Strategy;

module.exports = new LocalStrategy({
    usernameField : 'email',
    passwordField : 'password',
    passReqToCallback : true
  }, function(req, email, password, done) {
    var paramName = req.param('name');
    console.log('passport의 local-signup 호출됨 : ' + email + ', ' + password
                + ', ' + paramName);

.....
```

패스포트 설정 관련 코드를 passport.js 파일로 분리했으니 app.js 파일에서는 해당 코드를 모두 삭제합니다. 그러고 나서 다음 코드처럼 모듈 파일을 불러온 후 함수를 호출합니다.

참조 파일 | PassportExample2>app.js

```
……

// 패스포트 설정
var configPassport = require('./config/passport');
configPassport(app, passport);

……
```

이제는 app.js 파일에 있던 라우팅 관련 함수들을 별도의 모듈 파일로 분리합니다. [routes] 폴더 안에 user_passport.js 파일을 만들고 app.js 파일에 있던 코드를 옮겨와 입력합니다.

참조 파일 | PassportExample2>/routes/user_passport.js

```
module.exports = function(app, passport) {

  // 홈 화면 - 로그인 링크
  app.get('/', function(req, res) {

……

  });

  // 로그인 폼 링크
  app.get('/login', function(req, res) {

……

  });

  // 회원가입 폼 링크
  app.get('/signup', function(req, res) {

……

  });

  // 프로필
  app.get('/profile', function(req, res) {
    if(!req.isAuthenticated()) {
      res.redirect('/');
    } else {
```

```
    console.log('/profile 패스 요청됨.');
    console.dir(req.user);

    if(Array.isArray(req.user)) {
      res.render('profile.ejs', {user : req.user[0]._doc});
    } else {
      res.render('profile.ejs', {user : req.user});
    }
  }
});

// 로그아웃
app.get('/logout', function(req, res) {

.......

});

// 패스포트 - 로컬 로그인 라우팅
app.post('/login', passport.authenticate('local-login', {

.......

);

// 패스포트 - 로컬 회원가입 라우팅
app.post('/signup', passport.authenticate('local-signup', {

.......

);
```

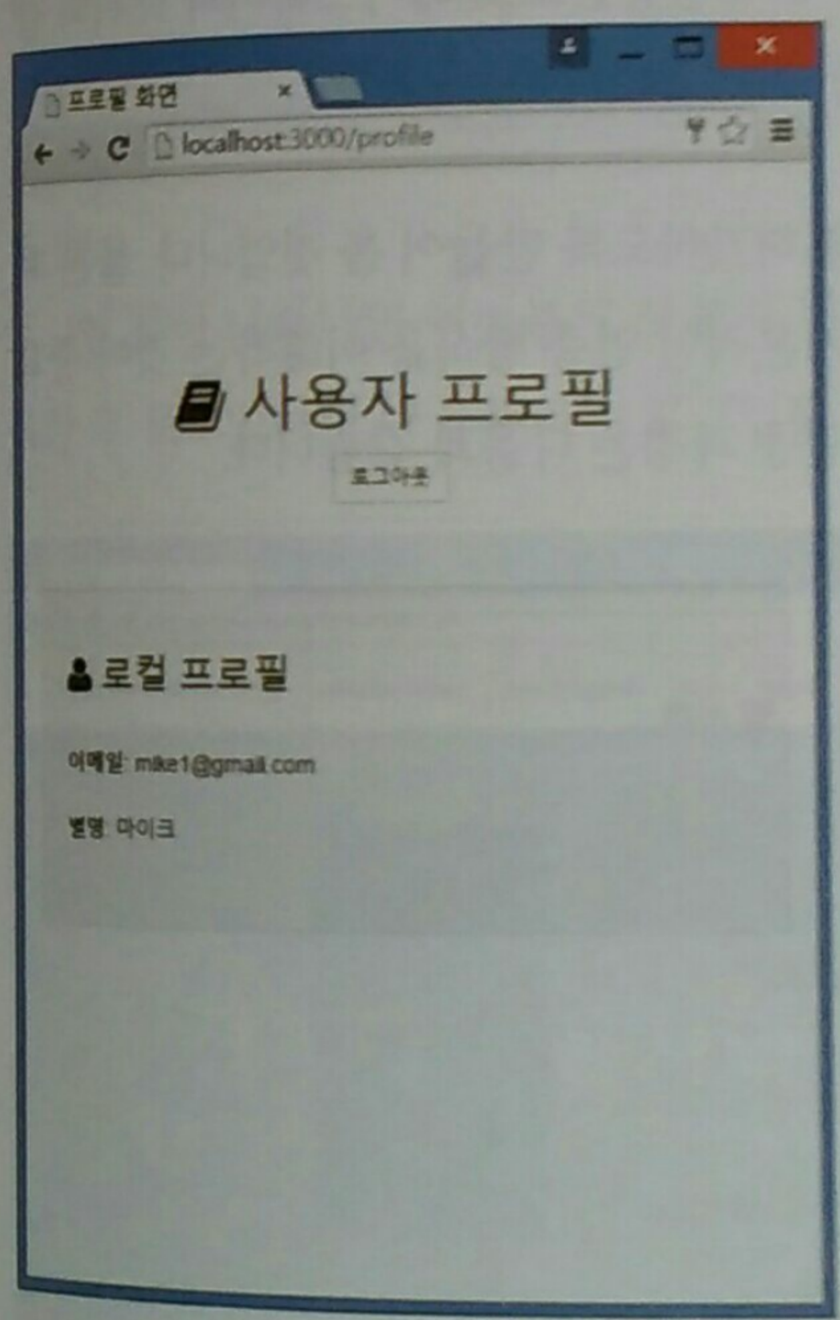
이 모듈 파일 안에서도 module.exports에 함수를 그대로 할당함으로써 이 모듈을 불러와 사용하는 곳에서 함수를 호출하면 이 함수 안에 있는 코드가 실행됩니다. 이 함수에는 app 객체와 passport 객체가 전달되므로 app.js 파일에 있던 코드들을 그대로 실행할 수 있습니다. 라우팅 함수들 중에서 /profile 패스로 요청된 정보를 처리하는 곳을 보면 PassportExample에서와 달리 req.isAuthenticated( ) 함수를 호출하여 로그인 여부를 확인하고 있습니다. 이 함수는 인증이 정상적으로 되었다면 true 값을 반환하므로 PasswordExample2에서는 이 메소드로 수정하였습니다.

라우팅 함수 관련 코드를 user_passport.js 파일로 분리했으니 app.js 파일에서는 해당 코드는 모두 삭제합니다. 그런 다음 모듈 파일을 불러온 후 함수를 호출합니다.

참조 파일 | PassportExample2>app.js

```
……

// 패스포트 관련 함수 라우팅

var userPassport = require('./routes/user_passport');
userPassport(app, passport);

……
```

모듈 파일로 분리하는 작업이 끝났습니다. 이제 app.js 파일을 실행한 후 로그인이나 회원가입 과정을 진행하여 이전과 똑같이 실행되는지를 확인해 봅니다.



▲패스포트 기능을 별도 파일로 모듈화한 후 로그인하여 프로필 화면을 띄운 경우

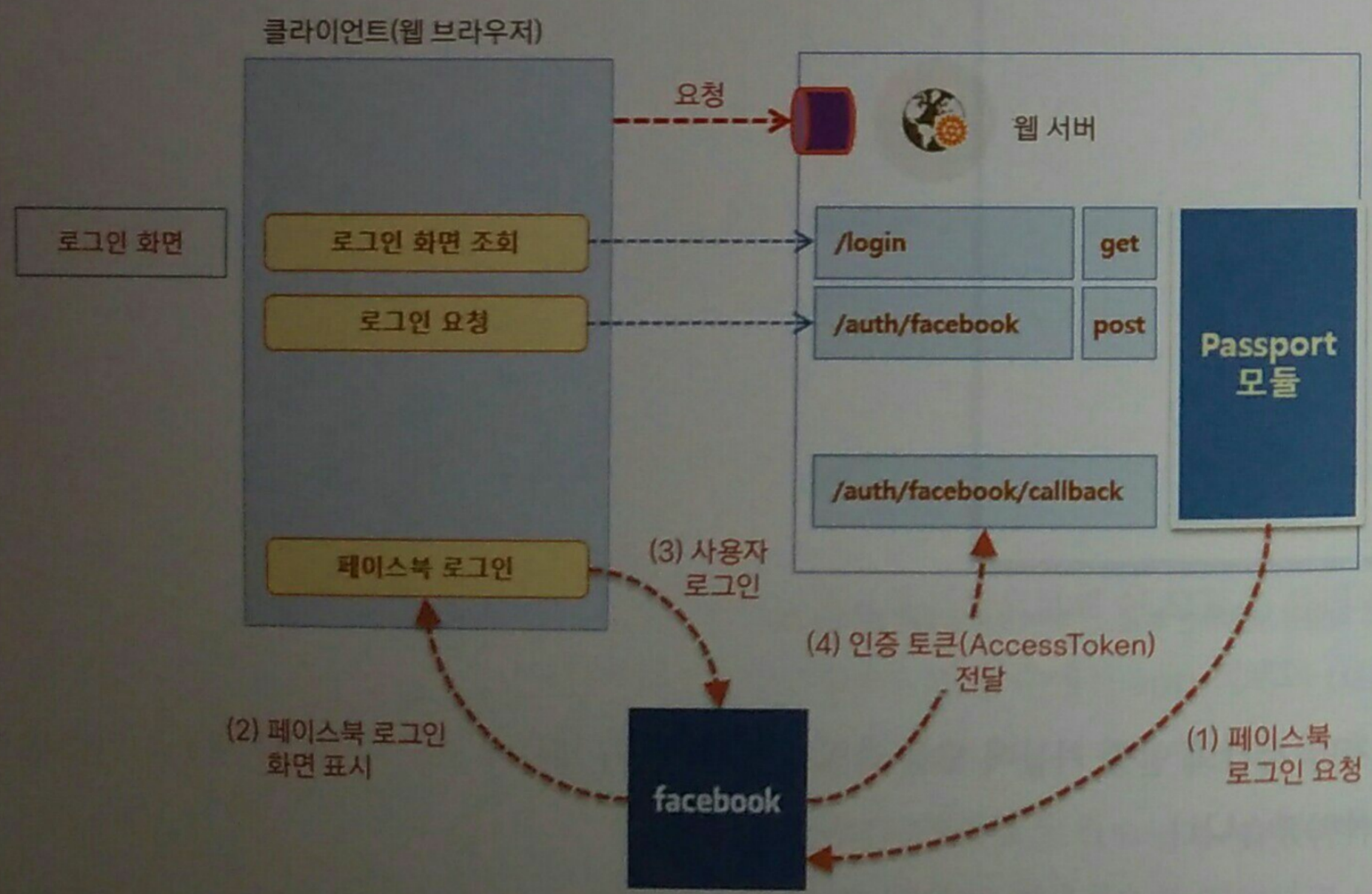
실행 기능은 같지만 각각의 인증 기능이 모듈 파일로 분리되었기 때문에 코드 수정이나 새로운 인증 방식을 추가하기 쉬워졌습니다.

09-3 | 페이스북으로 로그인하기

사용자가 회원가입을 할 때 여러 가지 내용을 입력해야 하는 번거로움을 없애기 위해 자주 사용하는 포털 사이트나 SNS 계정을 사용하는 로그인 방식은 이미 일반화되었습니다. 다른 사이트에 등록된 사용자 정보를 사용해 로그인하는 경우를 보면 '페이스북으로 로그인', '트위터로 로그인' 등 다양합니다. 패스포트 모듈을 사용하는 노드에는 이런 기능이 이미 만들어져 있어 인증 방식만 추가하면 그대로 그 기능을 사용할 수 있습니다.

그런데 '페이스북으로 로그인'이라는 기능은 내부적으로 어떻게 동작하는 것일까요? 페이스북은 **OAuth 인증**으로 외부 앱이나 사이트에서 페이스북 계정에 접근할 수 있도록 허용합니다. OAuth 인증은 **개방형 표준 프로토콜**(Open Standard Protocol)로 만들어졌기 때문에 이 인증만으로 페이스북이나 트위터 같은 인터넷 서비스 기능을 다른 앱이나 웹 사이트에서도 사용할 수 있습니다.

여기에서는 페이스북의 OAuth 인증 기능으로 사용자 로그인을 허가하도록 만들어 볼 것입니다. 물론 패스포트 모듈을 사용하면 쉽게 기능을 추가할 수 있지만 먼저 기본적인 인증 방식을 이해하는 것이 중요하니까요. '페이스북으로 로그인'을 했을 때 진행되는 OAuth 인증 과정은 다음과 같습니다.



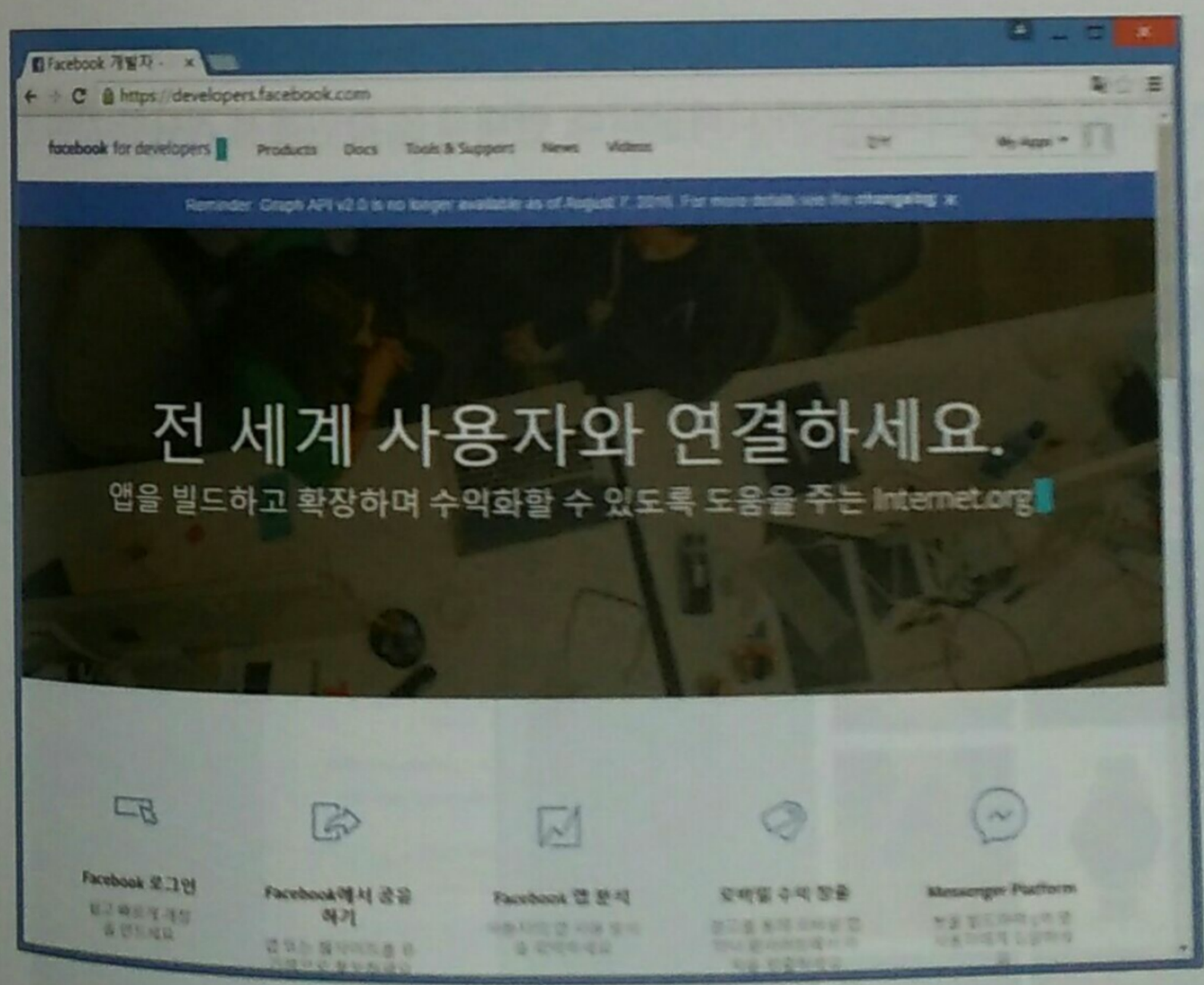
▲ '페이스북으로 로그인' 과정에서 사용하는 OAuth 인증 처리 과정

사용자는 웹 브라우저에서 로그인 화면을 조회한 후 로그인 요청을 하게 되는데 이때 /auth/facebook 같은 패스로 요청합니다. 이 요청을 받으면 패스포트 모듈이 페이스북 서버로 로그인 요청을 하게 되며, 페이스북은 사용자에게 페이스북 로그인 페이지를 보여 줍니다. 결국 사용자 인증은 페이스북 서버에서 진행되는 것입니다. 로그인에 성공하면 여러분이 만든 웹 서버는 그 결과를 알려주는 **인증 토큰**(Access Token)을 전달받을 수 있습니다. 페이스북 서버에서 /auth/facebook/callback 패스로 인증 성공 여부와 함께 사용자 정보를 여러분의 웹 서버로 전달하면, 패스포트 모듈은 이 사용자 정보를 데이터베이스에 저장한 후 클라이언트에 응답을 보낼 수 있습니다.

그런데 페이스북으로 로그인할 때 진행되는 인증 처리 과정을 보면 페이스북 서버가 로그인 기능을 담당합니다. 따라서 페이스북에 계정이 등록되어 있어야 하며 여러분이 만든 애플리케이션도 등록되어 있어야 합니다. 그러면 먼저 페이스북 개발자 콘솔 창에서 애플리케이션을 등록해 보겠습니다. 웹 브라우저를 열고 다음 사이트에 접속합니다.

▶ https://developers.facebook.com/

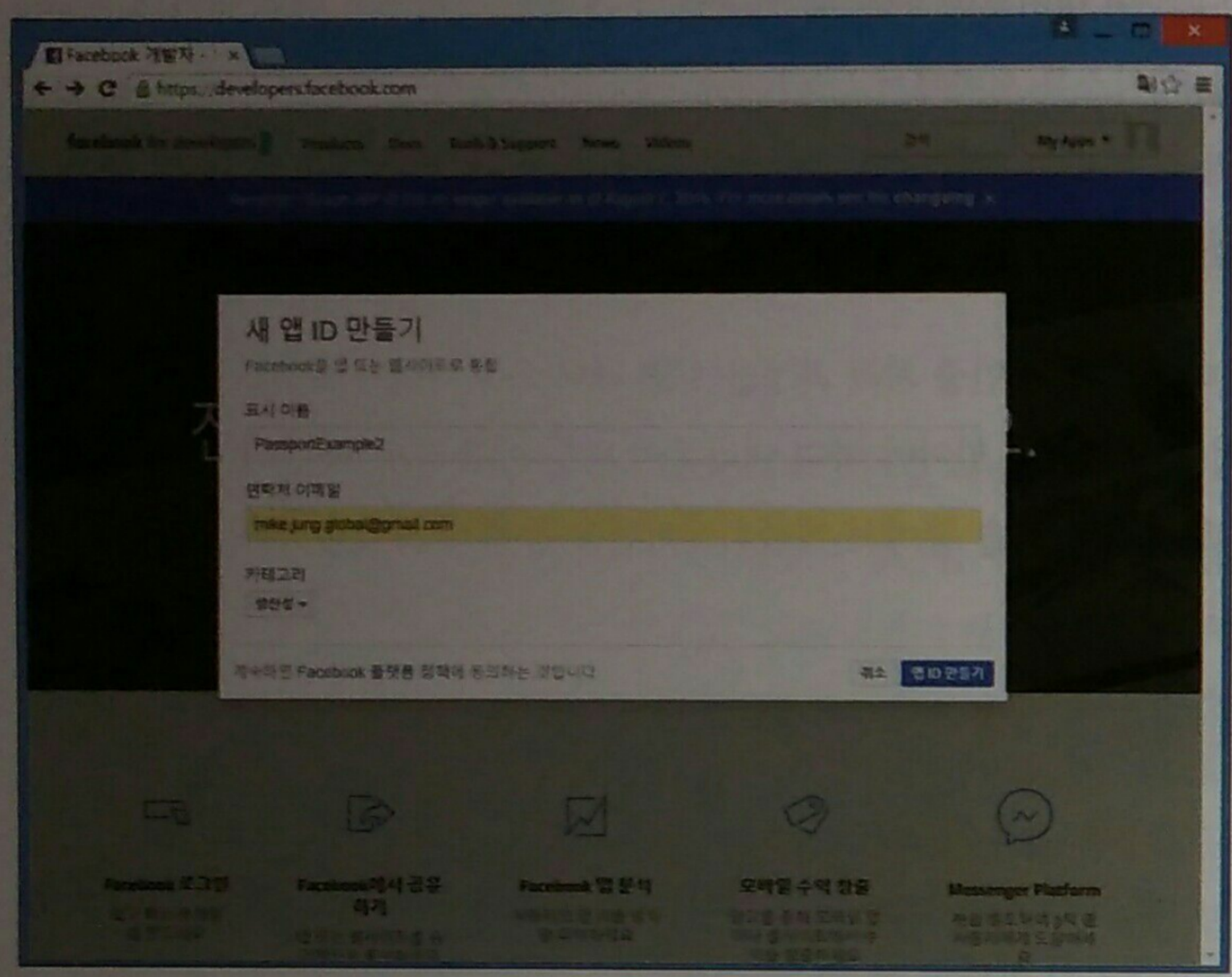
로그인 창이 나타나면 여러분의 사용자 계정으로 로그인합니다. 만약 페이스북 계정이 없으면 계정을 등록한 후 로그인하면 다음부터는 로그인된 상태로 볼 수 있습니다.



▲ 페이스북 개발자 콘솔

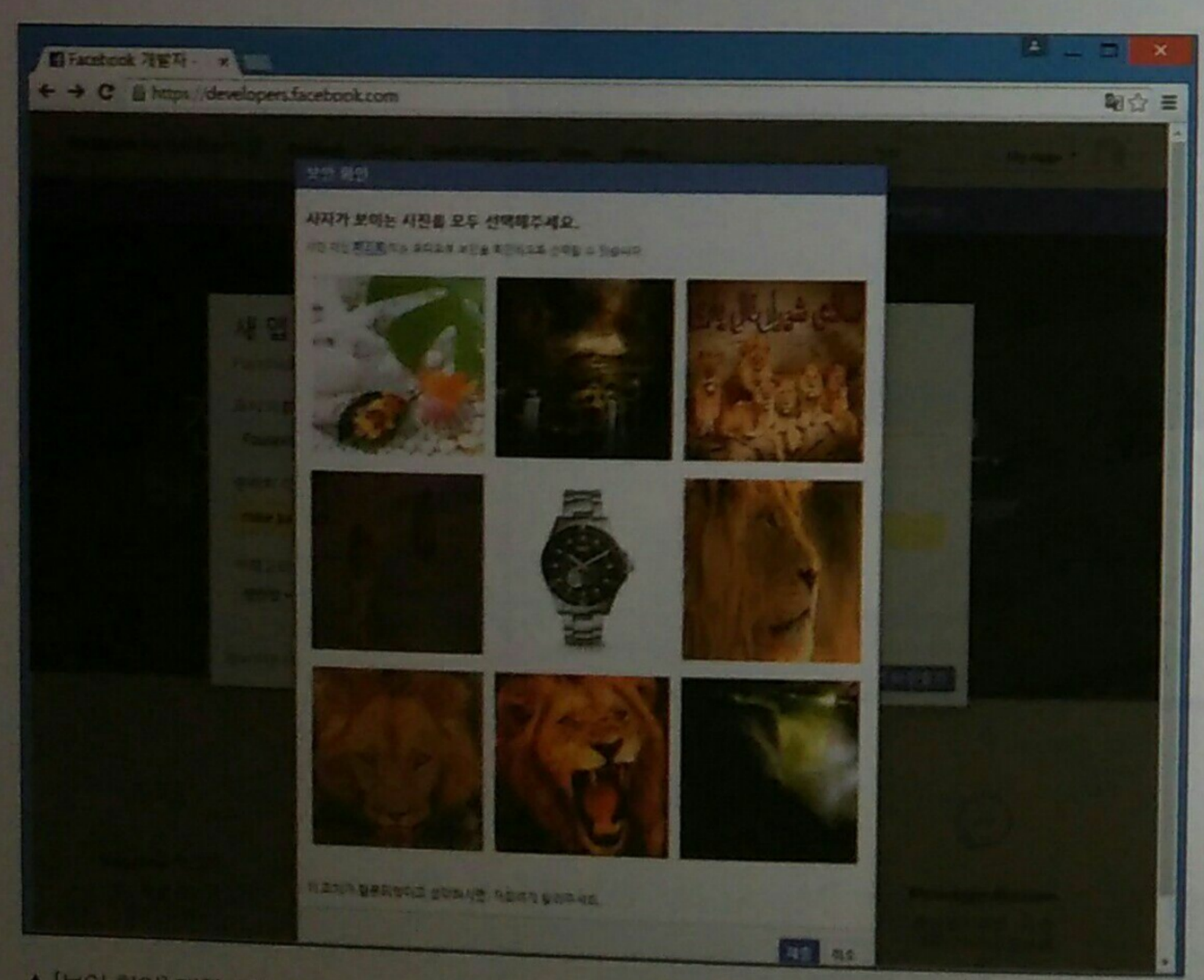
페이스북 개발자 콘솔 페이지의 오른쪽 위에 있는 [My Apps] 콤보박스에 마우스 커서를 올리면 기존에 등록한 앱 목록과 함께 [Add a New App] 메뉴가 나타납니다. [Add a New App] 메뉴를 누르면

[새 앱 ID 만들기] 대화상자가 나타납니다. '표시 이름' 입력 상자에는 PassportExample2를 입력하고 '연락처 이메일' 입력 상자에는 여러분의 이메일 주소를 입력합니다. 카테고리 콤보박스에는 적당한 분야를 선택합니다.



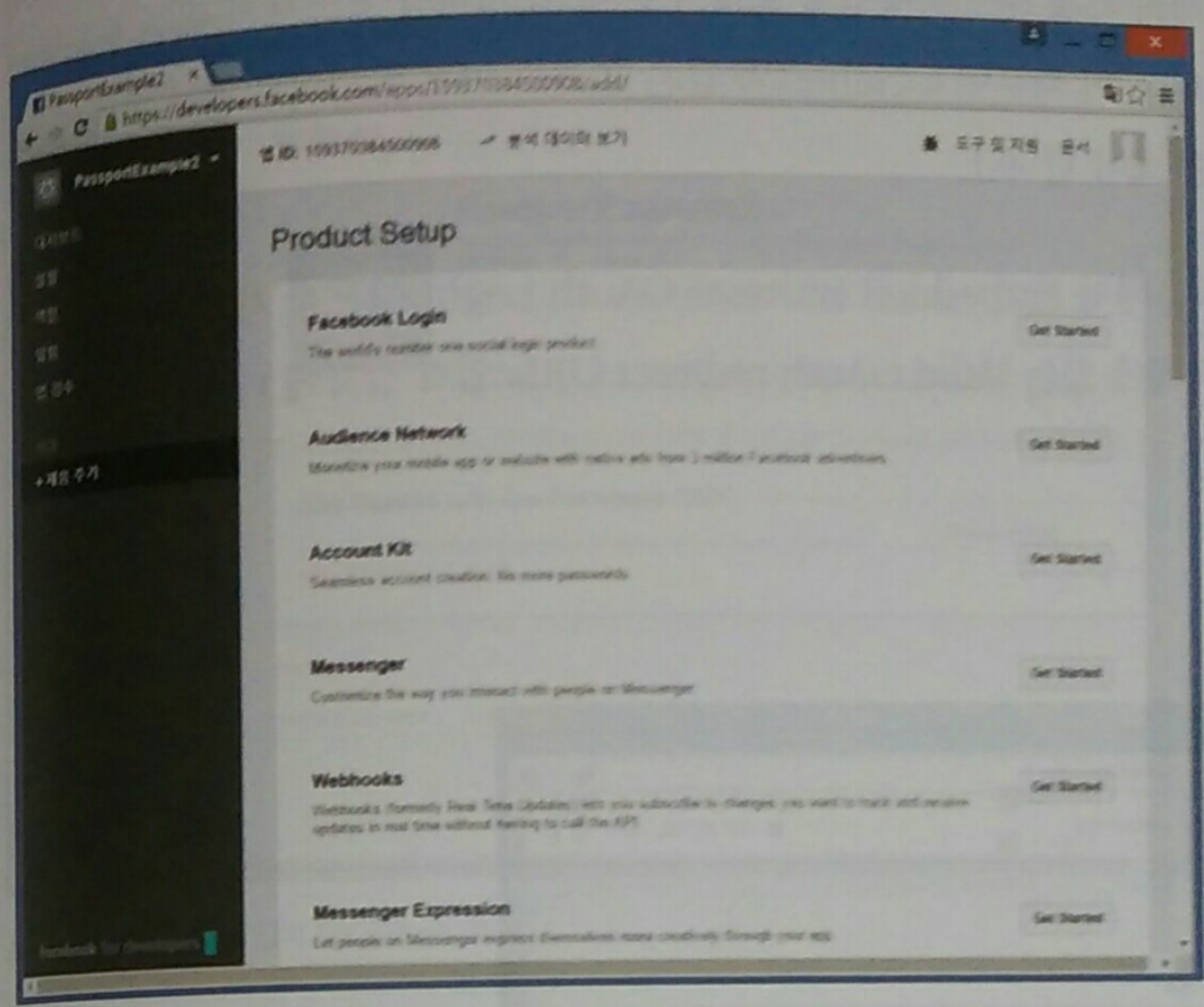
▲ [새 앱 ID 만들기] 대화상자

[앱 ID 만들기] 버튼을 클릭하면 [보안 확인] 대화상자가 나타납니다. 안내글을 확인해서 적당한 것을 선택한 후 [제출] 버튼을 클릭합니다.



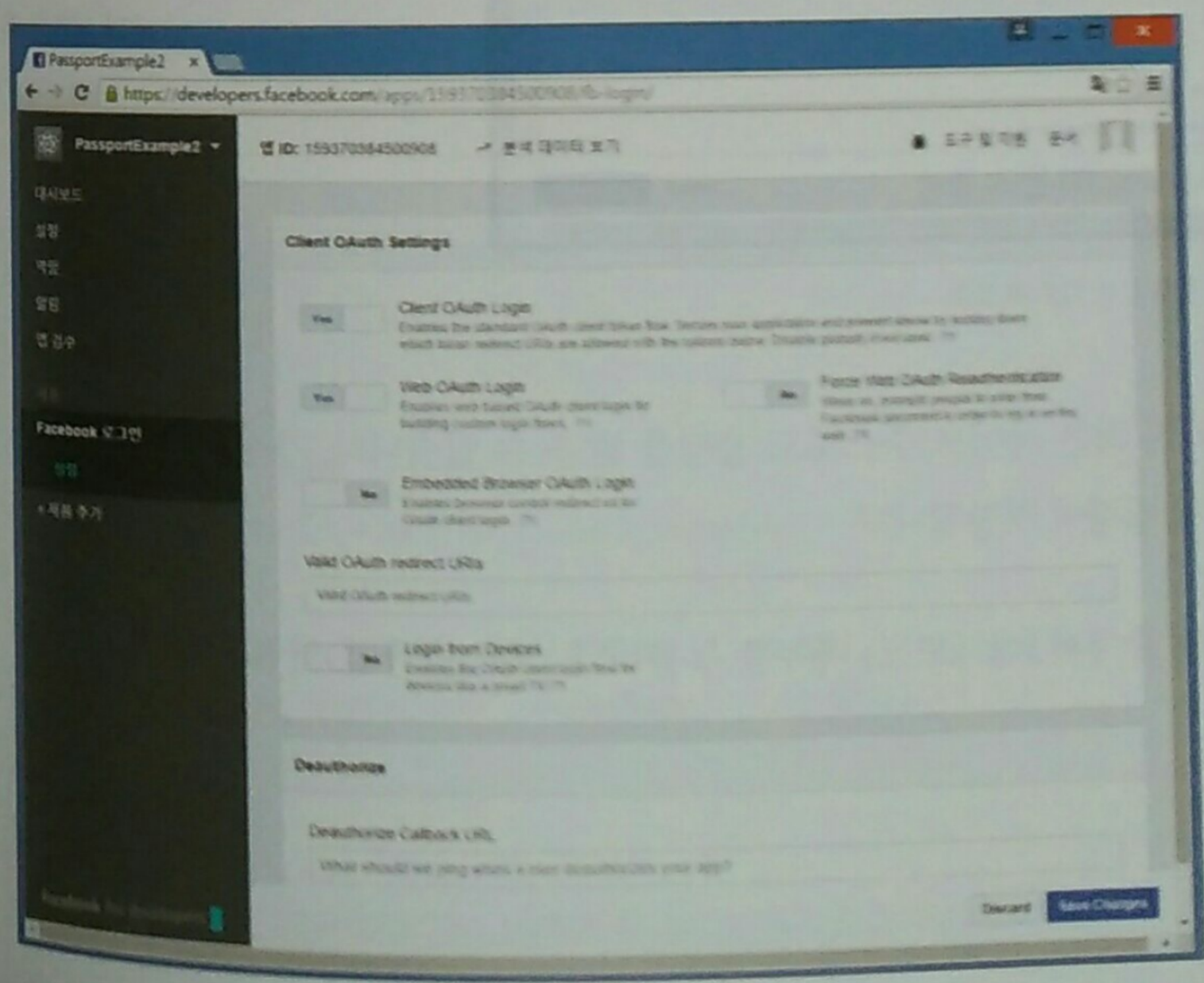
▲ [보안 확인] 대화상자

새로운 앱인 PassportExample2가 만들어진 후 그 앱에 대한 정보가 표시됩니다.



▲PassportExample2 앱의 정보를 표시한 페이지

페이스북 사용자 인증 기능을 사용하려면 Facebook Login 제품을 사용해야 합니다. 사용자 인증 기능을 사용하도록 설정하려면 Facebook Login 항목 오른쪽에 있는 [Get Started] 버튼을 클릭합니다. 그러면 Facebook Login을 사용할 때 필요한 설정 정보가 표시됩니다.

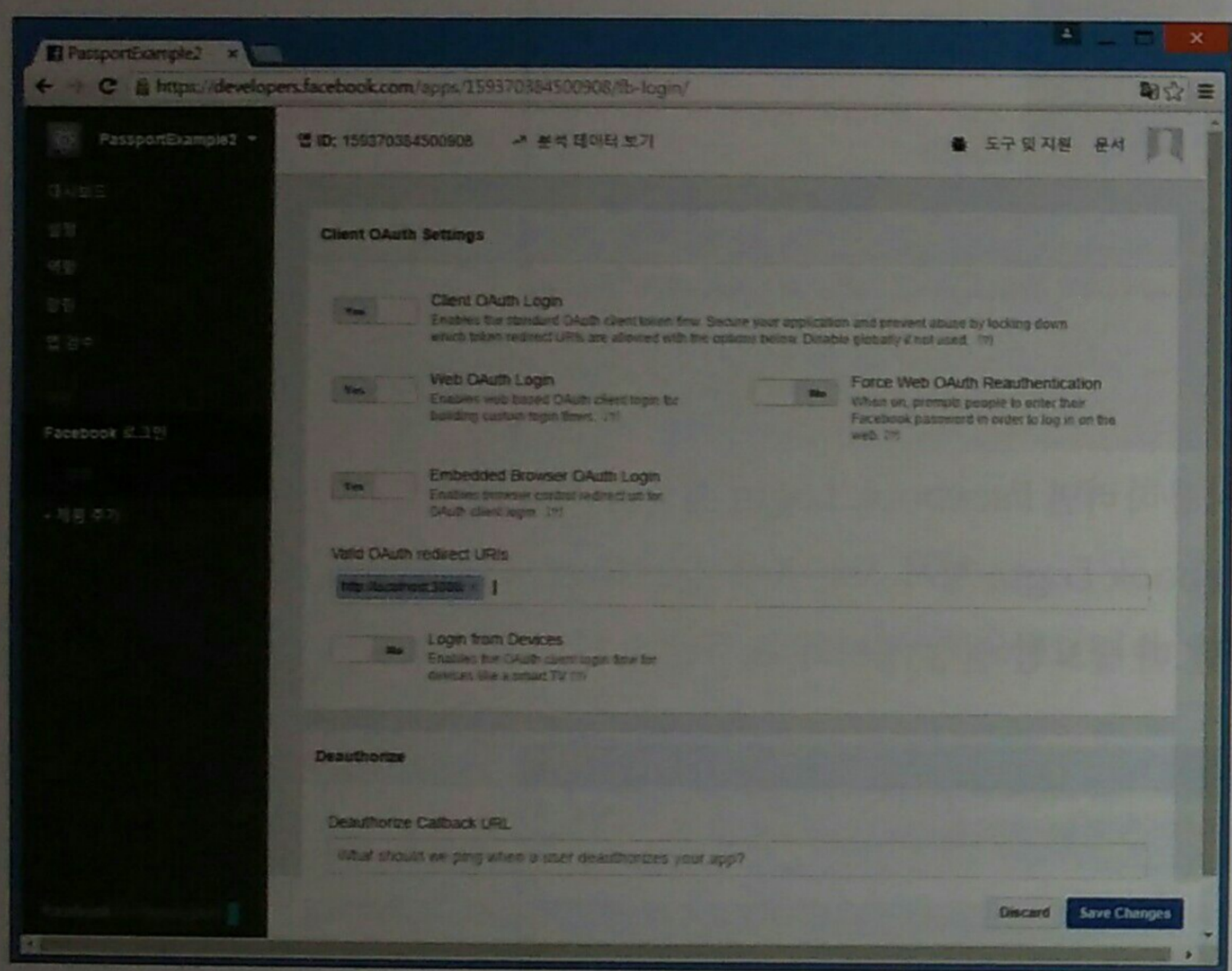


▲Facebook Login 설정 페이지

여러분이 페이스북 서버로 사용자 인증을 요청하면 OAuth 방식으로 응답을 받게 됩니다. 이때 페이스북 서버에서 여러분이 만든 서버의 특정 패스로 응답을 보내 주는데 이 서버의 호스트가 외부에 노출되지 않은 상태라면 문제가 생깁니다. 이 때문에 localhost와 같은 호스트 이름을 넣어도 응답 받을 수 있도록 이 페이지에서 추가 설정을 해야 합니다.

Client OAuth Settings 항목 중에서 Embedded Browser OAuth Login 항목을 사용할 수 있도록 YES로 바꾸어 설정하고, 그 아래에 있는 Valid OAuth redirect URIs 입력 상자에 다음과 같이 입력한 후 Enter 키를 누릅니다.

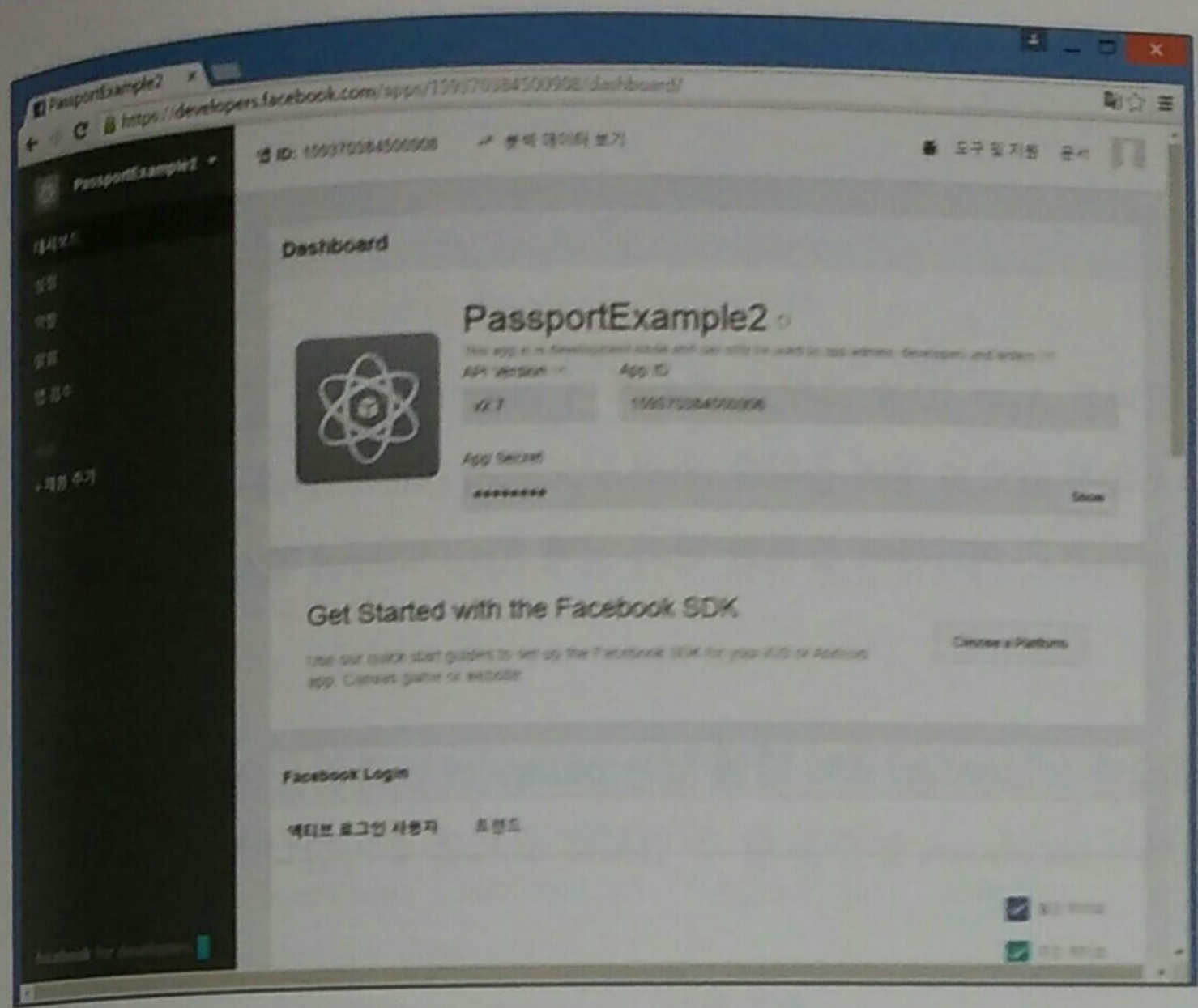
▶ http://localhost:3000/



▲ 외부에 노출되지 않은 사이트 주소로도 응답 받을 수 있도 록 설정

이렇게 설정하면 외부에 노출되지 않은 사이트 주소로도 응답을 받을 수 있습니다. 가장 아래쪽에 있는 [Save Changes] 버튼을 누르면 설정이 저장됩니다.

기본 설정이 끝났으니 왼쪽 메뉴 중에서 [대시보드] 메뉴를 누릅니다. 그러면 앱 ID와 Secret 정보를 볼 수 있습니다.



▲새로 생성된 앱의 App ID와 App Secret 정보

처음에는 App Secret 항목이 비밀번호를 입력했을 때처럼 특수 기호로만 보이지만 [Show] 버튼을 누르면 App Secret 정보를 볼 수 있습니다. 여기에서 확인한 App ID와 App Secret 정보는 프로젝트의 [config] 폴더 안에 있는 config.js 파일에 넣어 줄 것입니다.

이제 설정 파일을 바꿔 보겠습니다. config.js 파일을 열고 route_info 속성 다음에 facebook 속성을 추가합니다.

참조 파일 | PassportExample2>/config/config.js

```
……

module.exports = {
  server_port : 3000,
  db_url : 'mongodb://localhost:27017/shopping',
  db_schemas : [
    {file : './user_schema', collection : 'users6',
    schemaName : 'UserSchema', modelName : 'UserModel'}
  ],
  route_info : [
  ],
  facebook : {
    clientID : '0000000000000',
    clientSecret : '00000000000000000000',
    callbackURL : '/auth/facebook/callback'
```

```
  },

  ......
```

페이스북 개발자 콘솔 창에서 살펴본 App ID를 그대로 복사하여 clientID 속성 값으로 넣고 App Secret도 복사하여 clientSecret 속성 값으로 넣어 줍니다. 위의 코드에서 clientID와 clientSecret 값을 모두 숫자 0으로 입력했지만 여러분은 여러분이 등록한 앱의 실제 App Secret 값으로 넣어 주어야 합니다.

이제 패스포트 설정을 바꿔 보겠습니다. [config] 폴더 안에 있는 passport.js 파일을 열고 다음과 같이 모듈 파일을 불러들여 패스포트에서 facebook 인증을 할 수 있도록 코드를 추가합니다.

참조 파일 | PassportExample2>/config/passport.js

```
  ......

  var facebook = require('./passport/facebook');

  ......

  module.exports = function (app, passport) {

  ......

    passport.use('facebook', facebook(app, passport));

  ......

  }
```

[/config/passport] 폴더 안에 facebook.js 파일을 만들고 패스포트에서 페이스북 인증 방식을 사용할 수 있도록 다음 코드를 입력합니다.

참조 파일 | PassportExample2>/config/passport/facebook.js

```
var FacebookStrategy = require('passport-facebook').Strategy;
var config = require('../config');
```

```
module.exports = function(app, passport) {
  return new FacebookStrategy({
    clientID : config.facebook.clientID,
    clientSecret : config.facebook.clientSecret,
    callbackURL : config.facebook.callbackURL
  }, function(accessToken, refreshToken, profile, done) {
    console.log('passport의 facebook 호출됨.');
    console.dir(profile);

    var options = {
      criteria : {'facebook.id' : profile.id}
    };

    var database = app.get('database');
    database.UserModel.load(options, function (err, user) {
      if(err) return done(err);

      if(!user) {
        var user = new database.UserModel({
          name : profile.displayName,
          email : profile.emails[0].value,
          provider : 'facebook',
          facebook : profile._json
        });

        user.save(function(err) {
          if(err) console.log(err);
          return done(err, user);
        });
      } else {
        return done(err, user);
      }
    });
  });
};
```

'페이스북으로 로그인' 기능을 사용하려면 passport-facebook 모듈을 추가로 설치해야 합니다. 명령 프롬프트 창에서 다음 명령을 입력하여 모듈을 설치합니다.

```
% npm install passport-facebook --save
```

페이스북 인증 방식을 사용하도록 설정하려면 passport-facebook 모듈의 Strategy 객체를 참조한 후 new 연산자를 사용해 인스턴스 객체를 만들어 module.exports에 할당합니다. 인스턴스 객체를 만들 때 전달하는 객체에는 clientID, clientSecret, callbackURL 속성이 들어 있으며, 이 속성 값들은 config.js 파일에 넣어 둡니다. 이 값들은 이미 config.js 파일에 넣어 둔 값들로서 페이스북 개발자 콘솔창에서 복사해 온 것입니다.

인스턴스 객체를 만들 때 전달하는 두 번째 파라미터는 페이스북 인증이 완료되었을 때 호출되는 콜백 함수로서 accessToken, refreshToken, profile 객체를 파라미터로 전달받을 수 있습니다. 콜백 함수에서 전달받은 사용자 정보는 데이터베이스에 저장합니다. 그런데 앞에서 만든 데이터베이스 스키마에는 페이스북을 인증했을 때 전달받은 데이터를 저장할 수 있는 속성이 정의되어 있지 않습니다. 따라서 데이터베이스 스키마에 페이스북으로 인증했을 때의 정보를 저장하기 위해 user_schema.js 파일을 열고 스키마 정의 부분을 수정합니다.

참조 파일 | PassportExample2>/database/user_schema.js

```
.....

var UserSchema = mongoose.Schema({
  email : {type : String, 'default' : ''}
  , hashed_password : {type : String, 'default' : ''}
  , name : {type : String, index : 'hashed', 'default' : ''}
  , salt : {type : String}
  , created_at : {type : Date, index : {unique : false}, 'default' : Date.now}
  , updated_at : {type : Date, index : {unique : false}, 'default' : Date.now}
  , provider : {type : String, 'default' : ''}
  , authToken : {type : String, 'default' : ''}
  , facebook : { }
});

.....
```

스키마에 provider, authToken, facebook 속성을 추가했습니다. provider 속성에는 facebook, twitter, google처럼 사용자 인증 서비스를 제공하는 서비스 제공자 이름을 저장할 것이며, authToken 속성에는 인증 서비스를 제공하는 서버에서 응답받은 access token 값을 넣어 줄 것입니다. facebook 속성에는 응답받은 사용자 정보 객체를 그대로 넣을 것입니다. 필수 칼럼인 salt와 hashed_password도 수정합니다. required 속성을 제거하고 path( ) 메소드로 salt와 hashed_password 칼럼 값이 공백인지를 확인하는 코드도 삭제합니다.

스키마가 변경되었으므로 config.js 파일에서 지정하는 컬렉션의 이름도 users6으로 수정합니다.

참조 파일 | PassportExample2>/config/config.js

```
.....

module.exports = {

.....

    db_schemas : [
        {file : './user_schema', collection : 'users6',
         schemaName : 'UserSchema', modelName : 'UserModel'}
    ],

.....
```

데이터베이스 스키마를 수정했으니 그다음으로 라우팅 함수를 등록합니다. [routes] 폴더 안에 있는 user_passport.js 파일을 열고 다음 코드를 추가합니다.

참조 파일 | PassportExample2>/routes/user_passport.js

```
module.exports = function(app, passport) {

.....

  // 패스포트 - 페이스북 인증 라우팅
  app.get('/auth/facebook',
    passport.authenticate('facebook', {
      scope : 'email'
    })
  );

  // 패스포트 - 페이스북 인증 콜백 라우팅
  app.get('/auth/facebook/callback',
    passport.authenticate('facebook', {
      successRedirect : '/profile',
      failureRedirect : '/'
    })
  );

}
```

웹 브라우저에서 /auth/facebook 패스로 요청하면 페이스북 계정으로 사용자 인증을 진행합니다. /auth/facebook/callback 패스로 요청하면 페이스북 인증이 완료된 상태를 의미하므로 사용자 프로필 화면으로 이동합니다.

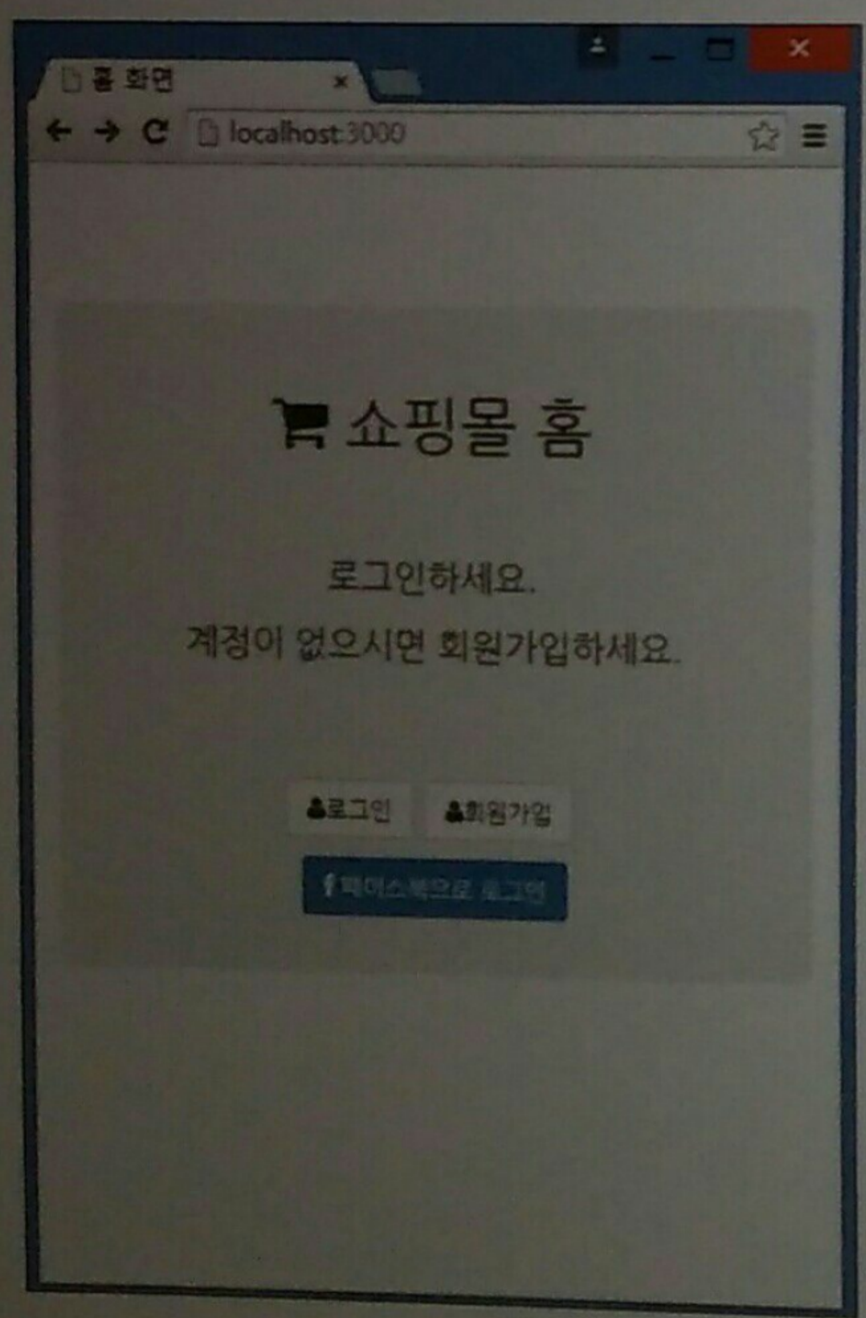
이제 마지막으로 홈 화면을 위한 뷰 템플릿인 index.ejs 파일을 수정합니다. 가장 아래쪽에 [페이스북으로 로그인] 버튼을 추가합니다.

참조 파일 | PassportExample2>/views/index.ejs

```
……

<a href = "/auth/facebook" class = "btn btn-primary">
  <span class = "fa fa-facebook"></span>
  페이스북으로 로그인
</a>

……
```

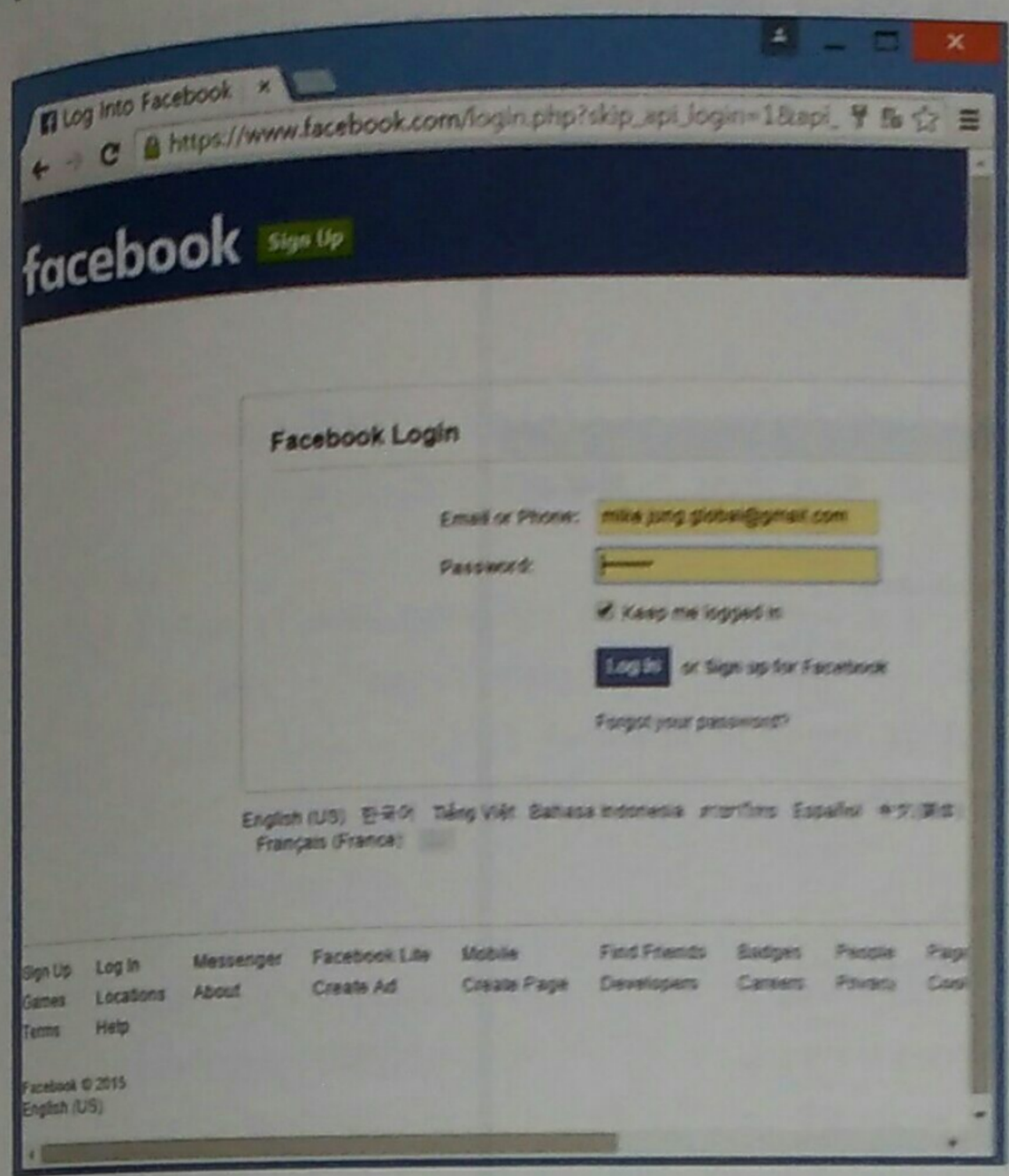
부트스트랩으로 웹 페이지를 만들 때는 그 안에 포함된 페이스북 아이콘을 사용할 수 있습니다. class 속성 값을 fa fa-facebook으로 지정하면 페이스북 아이콘을 보여 줄 수 있습니다.

app.js 파일을 실행한 후 웹 브라우저에서 홈 화면을 조회하면 다음과 같이 [페이스북으로 로그인] 버튼이 표시됩니다.



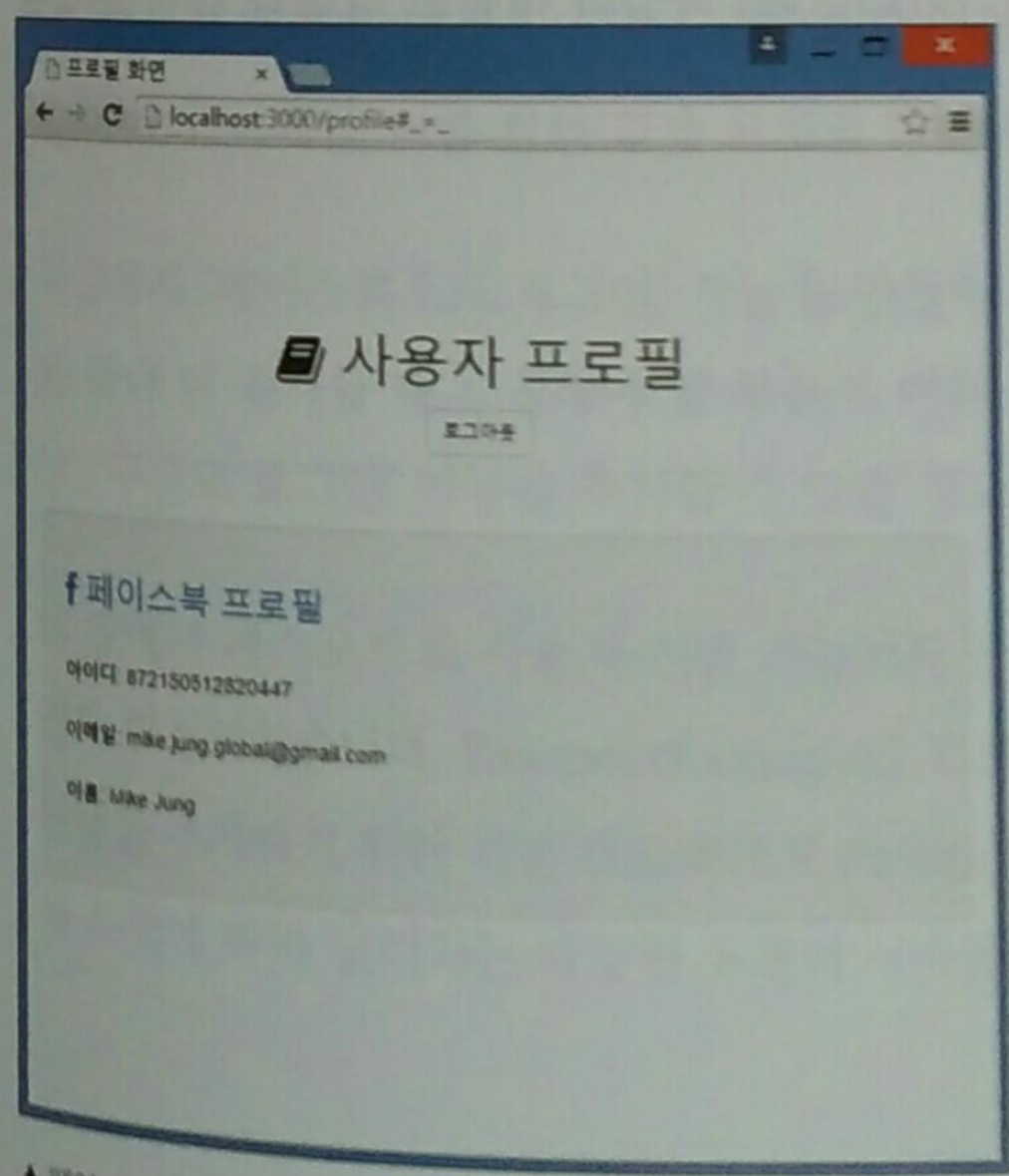
◀ [페이스북으로 로그인] 버튼이 추가된 홈 화면

[페이스북으로 로그인] 버튼을 누르면 페이스북의 사용자 인증 사이트로 이동합니다.



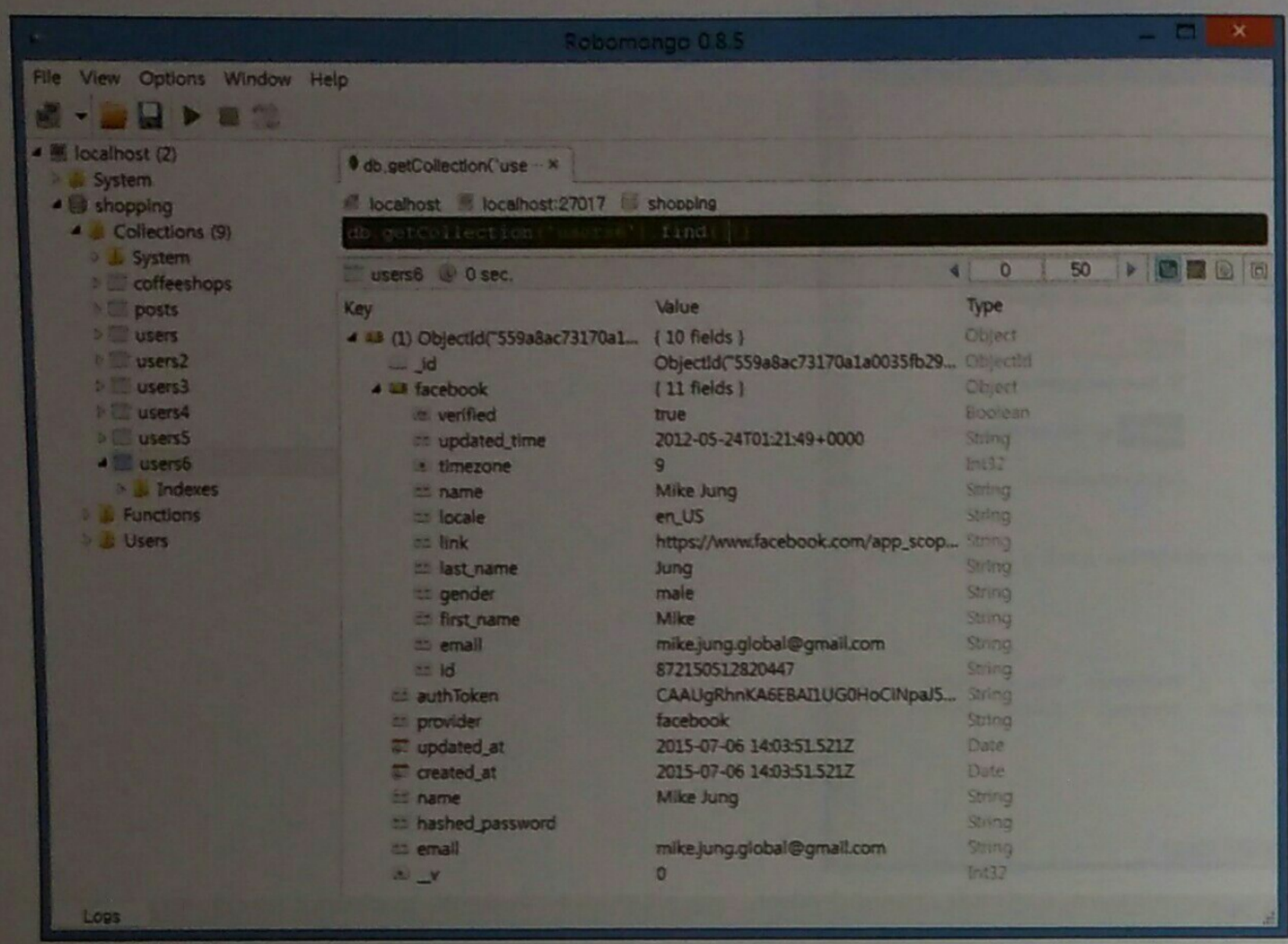
▲ 페이스북의 사용자 인증 화면으로 이동한 경우

이 사이트에서 로그인을 하면 콜백 사이트인 /auth/facebook/callback으로 응답받은 후 사용자 프로필 화면을 보여 줍니다.



▲ 페이스북의 사용자 인증 성공 후 표시된 사용자 프로필 화면

사용자 프로필 화면이 표시되면 성공적으로 사용자 인증을 마친 것입니다. Robomongo 프로그램을 실행한 후 users6 컬렉션에 들어 있는 데이터를 확인해 봅니다. 데이터베이스에 users6 컬렉션이 만들어졌는지, 페이스북으로 로그인했을 때 사용자 인증 정보가 저장되어 있는지를 알 수 있습니다.



▲ 데이터베이스에 저장된 사용자 인증 정보 확인

사용자 인증 정보가 데이터베이스에 잘 저장되었습니다. 그런데 페이스북 프로필 정보는 로컬 사용자 인증을 거친 후 보여 준 프로필 정보와 다릅니다. 이것은 profile.ejs 파일의 내용을 다음과 같이 수정했기 때문입니다.

```
.....

<% if(user.provider == undefined || user.provider == '') { %>
  <h3><span class = "fa fa-user"></span> 로컬 프로필</h3>
  <br>
  <p>
    <strong>이메일</strong> : <% = user.email %><br><br>
    <strong>별명</strong> : <% = user.name %>
  </p>
<% } else if(user.provider == 'facebook') { %>
  <h3 class = "text-primary">
    <span class = "fa fa-facebook"></span> 페이스북 프로필
  </h3>
  <br>
  <p>
    <strong>아이디</strong> : <% = user.facebook.id %><br><br>
    <strong>이메일</strong> : <% = user.facebook.email %><br><br>
    <strong>이름</strong> : <% = user.facebook.name %>
  </p>
<% } %>

.....
```

profile.ejs 템플릿에서는 provider 칼럼 값을 확인하여 facebook이면 페이스북 서버로부터 받아 저장한 아이디, 이메일, 이름 칼럼 값을 프로필 화면에 보여 줍니다.

지금까지 '페이스북으로 로그인' 기능을 만들어 보았습니다. 이제 여러분은 패스포트 모듈의 인증 방식을 하나 더 추가할 때 수정해야 할 부분이 여러 곳이라는 것을 알게 되었습니다. 그리고 '트위터로 로그인', '구글로 로그인' 기능을 추가할 수 있을 정도로 패스포트 모듈에 익숙해졌을 것입니다.

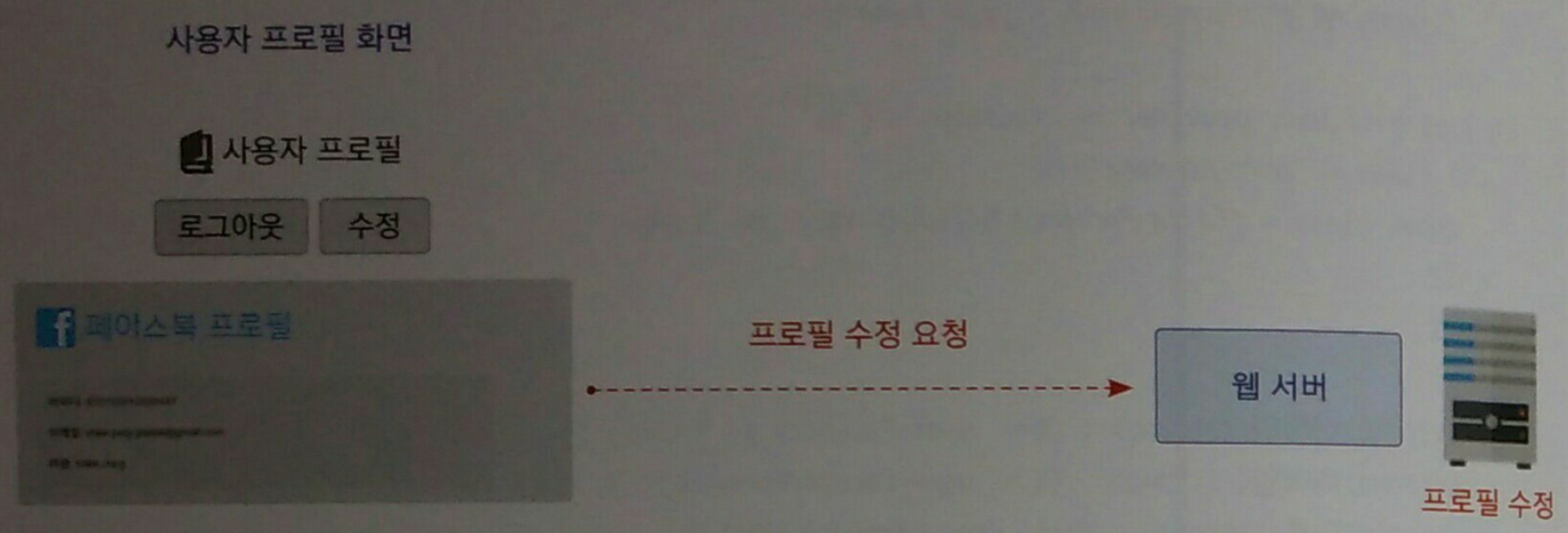
이 장에서 패스포트로 사용자 인증 기능까지 만들어 보았으므로 웹 서버에 들어가는 기본 기능은 어느 정도 완성되었습니다. PassportExample2 프로젝트에는 기본 기능이 구현되어 있으므로, 앞으로 다른 기능을 추가하기 위한 기본 프로젝트로 PassportExample2를 사용할 것입니다. 다음 장부터는 여러 가지 목적에 따라 달라지는 다양한 유형의 서버들을 하나씩 살펴볼 것입니다.

Do it!

도전 문제

01. 프로필 화면에 프로필을 수정할 수 있는 기능을 넣어 보세요.

프로젝트 소스 Mission11 난이도 초급 ★☆☆☆



Hint!

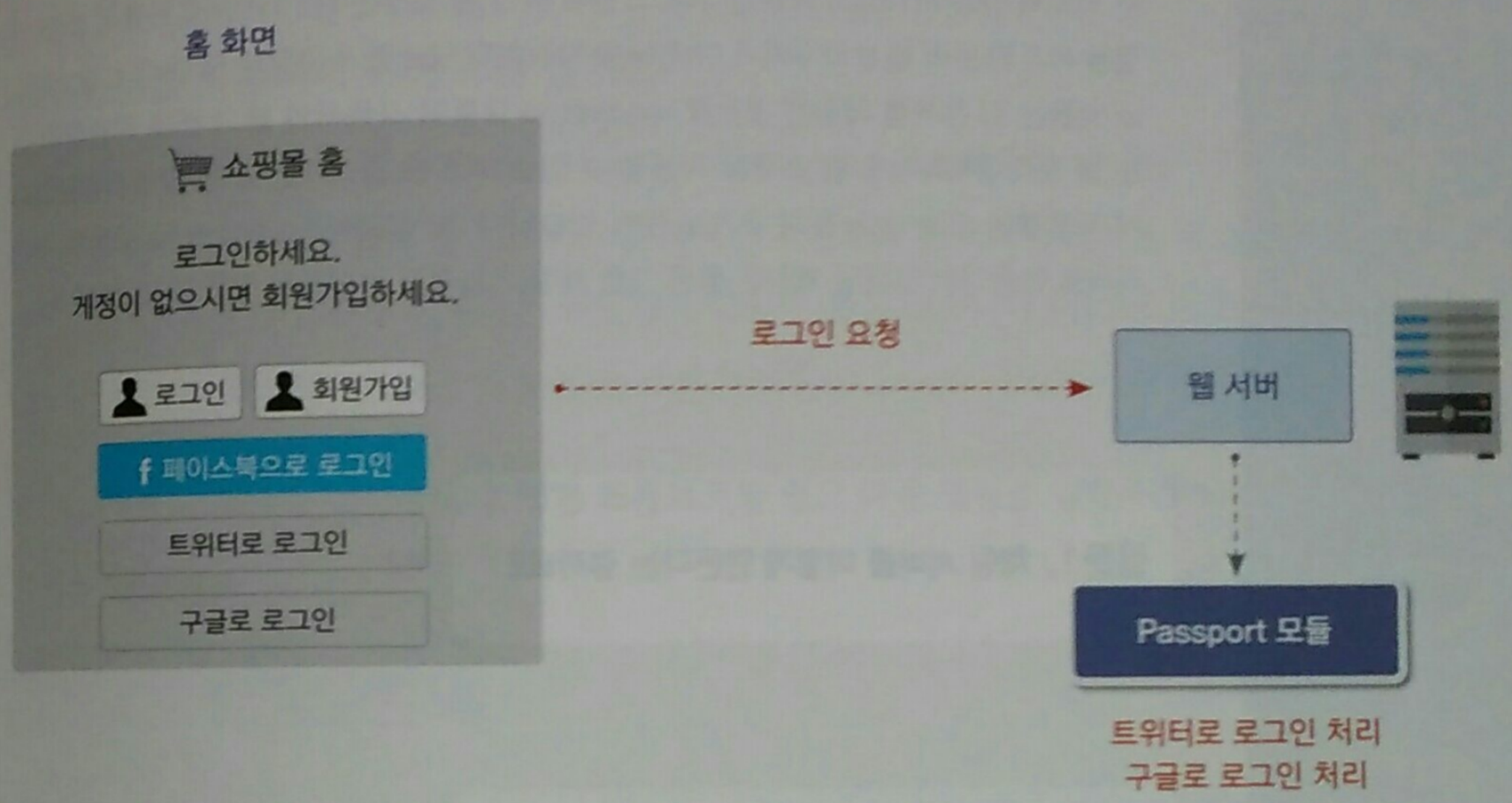
1. 사용자 프로필 화면에 [수정] 버튼을 추가하고, 이 버튼을 눌렀을 때 이메일이나 이름 정보를 수정할 수 있도록 만듭니다.
2. 프로필 수정 화면에서 [저장] 버튼을 누르면 웹 서버로 프로필 수정 요청을 보내고, 웹 서버에서는 사용자 프로필 정보를 수정합니다.
3. 데이터베이스에 저장된 프로필 정보를 정상적으로 수정한 후 응답 메시지를 보냅니다.
4. 클라이언트에서 응답 메시지를 받으면 수정된 내용을 사용자 프로필 화면에서 보여 줍니다.

MEMO

Do it!
도전 문제

02. 09-2장에서 만든 PassportExample2 프로젝트에 '트위터로 로그인' 기능과 '구글로 로그인' 기능을 추가해 보세요.

프로젝트 소스 Mission12 난이도 중급 ★★☆☆



Hint!

1 홈 화면에 [트위터로 로그인]과 [구글로 로그인] 버튼을 추가합니다. 두 개의 버튼은 [페이스북으로 로그인] 버튼의 오른쪽 또는 아래쪽에 추가합니다.

2 [트위터로 로그인] 버튼을 누르면 웹 서버 쪽으로 인증을 요청하며 패스포트 모듈에서 트위터로 로그인 처리를 진행합니다. 이를 위해 서버 쪽에는 미리 인증을 위한 함수들을 만들어 둡니다.

3 [구글로 로그인] 버튼을 누르면 웹 서버 쪽으로 인증을 요청하며 패스포트 모듈에서 구글로 로그인 처리를 진행합니다. 이를 위해 서버 쪽에는 미리 인증을 위한 함수들을 만들어 둡니다.

4 클라이언트에서 응답 메시지를 받았을 때 프로필 화면으로 넘어가는 과정은 이전과 같습니다.

MEMO

10 채팅 서버 만들기

실시간 채팅 기능은 노드의 특징을 가장 잘 반영하고 있습니다. 일대일 채팅이나 그룹 채팅 기능을 만들어 사용하다 보면 노드 프로그램이 다른 언어로 만든 프로그램보다 얼마나 빠르고 간편한지 실감할 것입니다. 노드에서 채팅 기능을 만들려면 웹 소켓(Web Socket)을 사용해야 합니다. 웹 소켓은 웹 서버로 소켓을 연결한 후 데이터를 주고받을 수 있도록 만든 HTML5 표준입니다. 그런데 이런 웹 소켓은 HTTP 프로토콜로 소켓 연결을 하기 때문에 웹 브라우저가 이 기능을 지원하지 않으면 사용할 수 없습니다. socket.io 모듈은 이 문제를 해결한 것으로, socket.io 모듈을 사용하면 웹 소켓을 지원하지 않는 웹 브라우저에서도 웹 소켓을 사용할 수 있습니다. 또 클라이언트와 서버에서 편리하게 사용할 수 있는 기능들이 추가로 들어 있습니다. 이 장에서는 socket.io 모듈로 채팅 서버를 만들어서 일대일 채팅은 물론 그룹 채팅 기능까지 간편하게 만들어 보겠습니다.

질문으로 시작하기

질문1. 채팅 서버를 어떻게 만든다는 걸까요? ➡ 10-1. socket.io 사용하기

질문2. 일대일 채팅 기능을 만들어보고 싶나요? ➡ 10-2. 일대일 채팅하기

질문3. 그룹 채팅 기능도 만들 수 있나요? ➡ 10-3. 그룹 채팅하기

질문4. 웹 문서를 예쁘게 꾸미고 싶나요? ➡ 10-4. 채팅 웹 문서 예쁘게 꾸미기

10-1 | socket.io 사용하기

웹 브라우저에서 채팅을 하려면 먼저 웹 서버에 socket.io 모듈을 설치한 후 어떻게 메시지를 보내고 받는지 알아야 합니다. 여기에서는 socket.io로 서버에 메시지를 보내고 받는 기본적인 방법부터 알아보겠습니다.

socket.io를 사용하기 위해 모듈 설치하기

socket.io로 채팅 기능을 만들기 위해 PassportExample2 프로젝트를 복사하여 ChatExample 프로젝트를 만듭니다. 이미 만들어 둔 프로젝트를 복사한 것이므로 프로젝트 안에는 익스프레스를 사용하는 코드, 그리고 사용자 인증을 진행하는 파일과 폴더들이 들어 있습니다. socket.io 모듈도 익스프레스와 함께 동작할 수 있으므로 앞에서 만들어 둔 프로젝트에 socket.io 처리 코드를 하나씩 추가할 것입니다.

먼저 socket.io 모듈을 설치합니다. 명령 프롬프트를 열고 다음 명령을 입력하면 socket.io 모듈이 추가로 설치됩니다.

```
% npm install socket.io --save
```

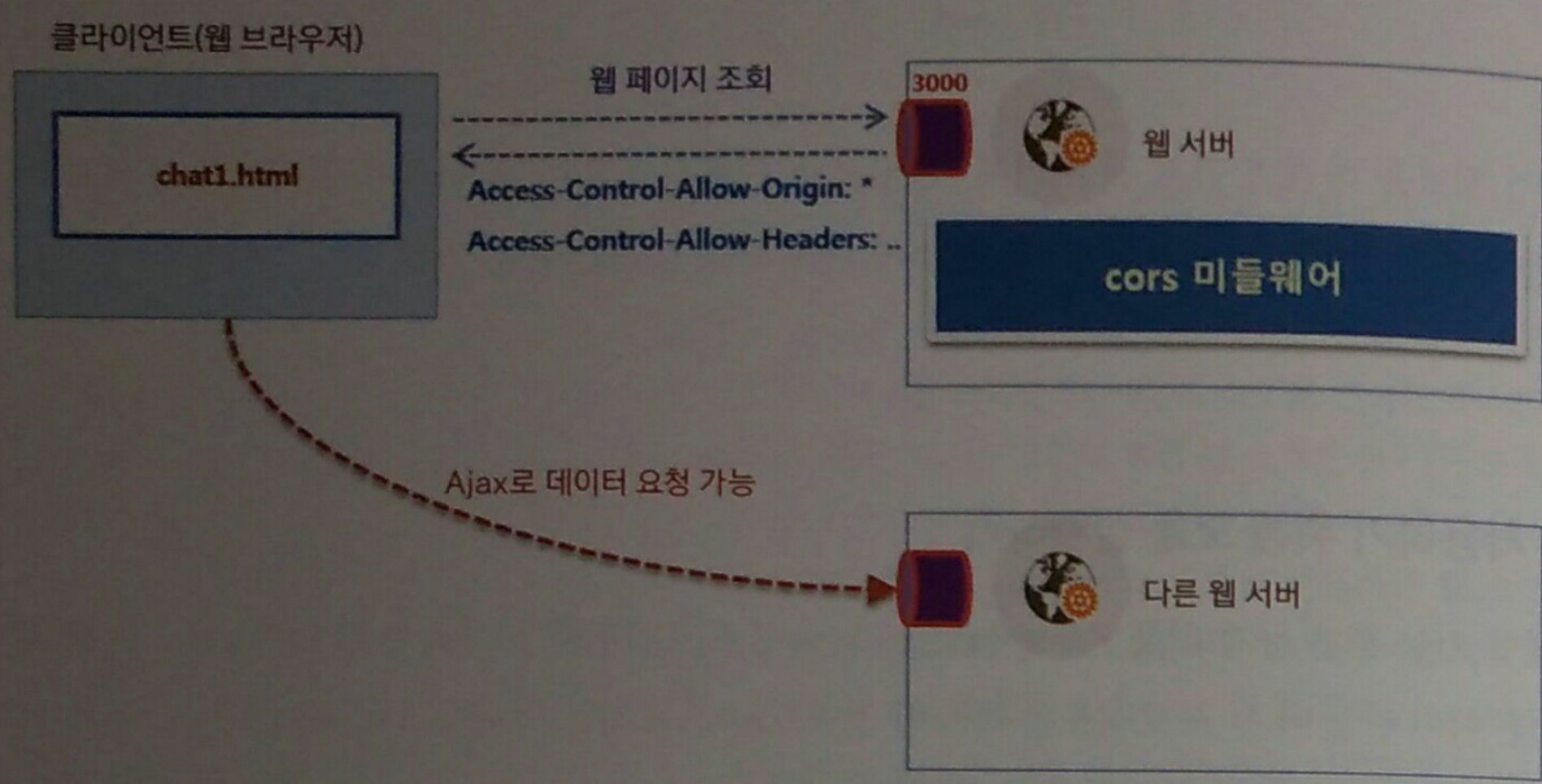
그런데 socket.io를 사용하려면 cors 모듈도 설치되어 있어야 합니다. CORS에 대해 좀 더 자세하게 살펴보면, 인터넷 익스플로러와 같은 브라우저에서는 Ajax를 사용해 데이터를 가져올 수 있도록 XHR(XMLHttpRequest)을 지원했습니다. 그런데 보안 문제를 이유로 웹 사이트를 제공하는 서버 이외의 다른 서버는 접속할 수 없다는 제약을 만들게 되었습니다.

그러나 현실에서는 OpenAPI를 사용해 어떤 클라이언트든 웹에서 데이터를 가져가도록 하는 경우가 많아지고 있습니다. 그리고 하나의 클라이언트에서 여러 서버에 접속하여 다양한 데이터를 가져오는 경우도 많아지면서 이 제약은 큰 문제가 되었습니다. 이 때문에 '데이터 제공 요청을 받은 웹 서버가 허용하면 다른 웹 서버로도 접속할 수 있다'는 것을 CORS 표준으로 만들게 된 것입니다.

정박사의 한마디 CORS가 무엇인지 정확히 알아볼까요?

CORS(Cross-Origin Resource Sharing)를 사용하면 클라이언트에서 Ajax를 사용해 데이터를 가져올 때 현재 보고 있는 브라우저의 웹 문서를 제공한 웹 서버 이외에 다른 웹 서버에서는 접속할 수 없는 제약이 풀립니다. 노드에서는 cors 모듈을 설치하고 미들웨어로 사용하면 이 제약을 간단하게 해결할 수 있습니다.

다음은 노드에서 cors 모듈을 설치해서 사용하는 구성을 보여 줍니다.



▲ cors 미들웨어를 사용해 다른 웹 서버로 요청하도록 만든 구성

이 장에서 사용하는 socket.io 모듈은 기본적으로 웹 소켓을 사용하지만 웹 브라우저가 웹 소켓을 지원하지 않는 경우에는 XHR을 사용할 수도 있으므로 CORS 표준을 사용하지 않으면 다른 웹 서버에 접근할 수 없게 됩니다.

웹 서버에서 어떻게 cors 모듈을 사용할 수 있는지를 알려 주는 위 그림을 보면 여러 대의 웹 서버가 있고, 웹 브라우저(예를 들어, chat1.html 웹 페이지)는 그중 하나의 웹 서버에서 조회하게 됩니다. 이 경우에 Access-Control-Allow-Origin 헤더를 포함한 웹 서버가 응답을 보내 주면 웹 브라우저는 chat1.html 파일을 받아 온 웹 서버 외의 다른 웹 서버로도 데이터를 요청할 수 있다는 의미로 받아들입니다. 따라서 Ajax로 다른 웹 서버에 요청하는 것을 허용해 줍니다.

명령 프롬프트에서 다음 명령을 입력하여 cors 모듈을 설치합니다.

```
% npm install cors --save
```

socket.io 모듈과 cors 모듈을 모두 설치했다면 socket.io를 사용할 준비가 끝난 것입니다. 채팅을 위해 새로운 기능을 추가하기 전에 프로젝트 안에 있는 파일과 폴더 중 몇 가지 기본적인 것들을 다음 표에서 다시 한 번 살펴보겠습니다.

파일 또는 폴더	설명
app.js	기본 코드가 들어 있는 메인 파일입니다.
/config/config.js	설정 정보가 들어 있는 파일입니다. 스키마 모듈이나 라우팅 모듈을 만들어 추가했다면 이 파일에 설정 정보를 추가합니다.

/database	데이터베이스의 스키마 모듈 파일들을 만들어 넣는 폴더입니다.
/routes	라우팅 함수들을 모듈 파일로 만들어 넣는 폴더입니다.
/views	뷰 템플릿 파일들을 만들어 넣는 폴더입니다.

[표] ChatExample 프로젝트 안에 있는 파일과 폴더들

app.js 메인 파일은 설정 정보를 로딩하고 웹 서버를 실행시킵니다. 설정 정보는 [config] 폴더에 있는 config.js 파일 안에 들어 있으며, 직접 만들어 추가한 데이터베이스 스키마 모듈의 정보나 라우팅 모듈을 등록할 수 있습니다. [database] 폴더에는 데이터베이스 스키마 모듈 파일을 넣어 둡니다. [routes] 폴더에는 라우팅 함수들을 모듈 파일로 만들어 넣어두며, [views] 폴더에는 뷰 템플릿 파일들을 만들어 넣어둡니다.

app.js 메인 파일에 기본 코드 하나씩 추가하기

이제 app.js 파일을 열고 코드를 하나씩 추가합니다. 먼저 require( ) 메소드로 socket.io 모듈과 cors 모듈을 불러들인 후 cors 모듈은 미들웨어로 등록합니다.

참조 파일 | ChatExample>app.js

```
……

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

// cors 사용 - 클라이언트에서 ajax로 요청하면 CORS 지원
var cors = require('cors');

……

// cors를 미들웨어로 사용하도록 등록
app.use(cors());

……
```

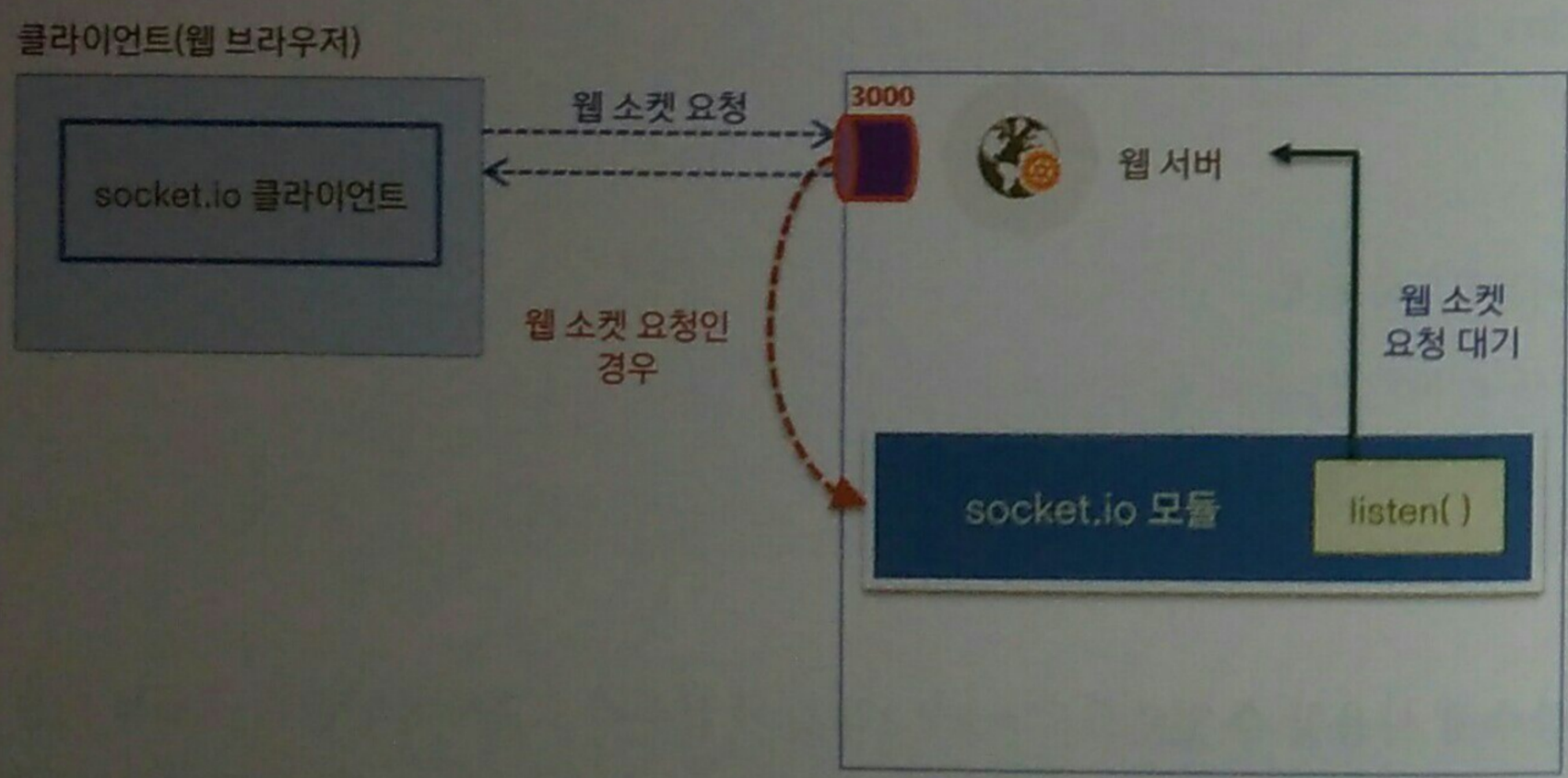
cors 모듈을 불러오면 함수로 사용할 수 있으므로 cors( )와 같이 함수를 호출하여 반환된 객체를 미들웨어로 등록합니다. 이 코드는 라우터 미들웨어를 사용하도록 만드는 코드 바로 위에 추가합니다. 이제 웹 문서를 받아 온 서버가 어떤 것인지에 상관없이 어떤 서버라도 socket.io 모듈로 접근할 수 있게 되었습니다.

이제 socket.io 모듈을 사용해 클라이언트가 보내온 요청을 처리하기 위해 파일의 마지막 부분에서 socket.io 모듈 객체의 listen( ) 메소드를 호출하도록 입력합니다.

참조 파일 | ChatExample>app.js

```
……

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

        database.init(app, config);
});

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

http 객체의 createServer( ) 메소드를 호출하여 익스프레스 서버를 만들고, 서버 객체의 listen( ) 메소드를 호출하여 서버를 실행하는 코드는 이미 익숙합니다. 그 아래에 새로 추가한 코드는 socket.io 모듈입니다. 이 모듈은 익스프레스 서버에서 웹 소켓 요청을 받아들입니다. socket.io 모듈은 실행 중인 웹 서버 위에서 동작할 수 있도록 만들어져 있으므로 socket.io 모듈에서 listen( ) 메소드를 호출하면서 웹 서버 객체를 파라미터로 전달하면 그다음부터 웹 소켓으로 들어오는 요청을 처리할 수 있습니다. http 모듈로 실행한 익스프레스 서버는 server **변수**에 저장되어 있으므로 그 아랫부분에서 socket.io 모듈 객체의 listen( ) 메소드를 호출합니다.



▲웹 서버가 socket.io 모듈로 웹 소켓 요청을 처리하도록 listen( ) 메소드 호출하기

socket.io 모듈로 클라이언트로부터 들어오는 웹 소켓 요청을 처리할 수 있도록 만드는 메소드는 listen( ) 메소드 외에도 attach( ) 메소드가 있습니다.

메소드 이름	설명
attach(httpServer, options)	웹 서버 인스턴스가 socket.io를 처리합니다.
listen(httpServer, options)	위의 attach( ) 메소드와 같은 기능입니다.

[표] socket.io로 요청을 처리하도록 설정하는 메소드

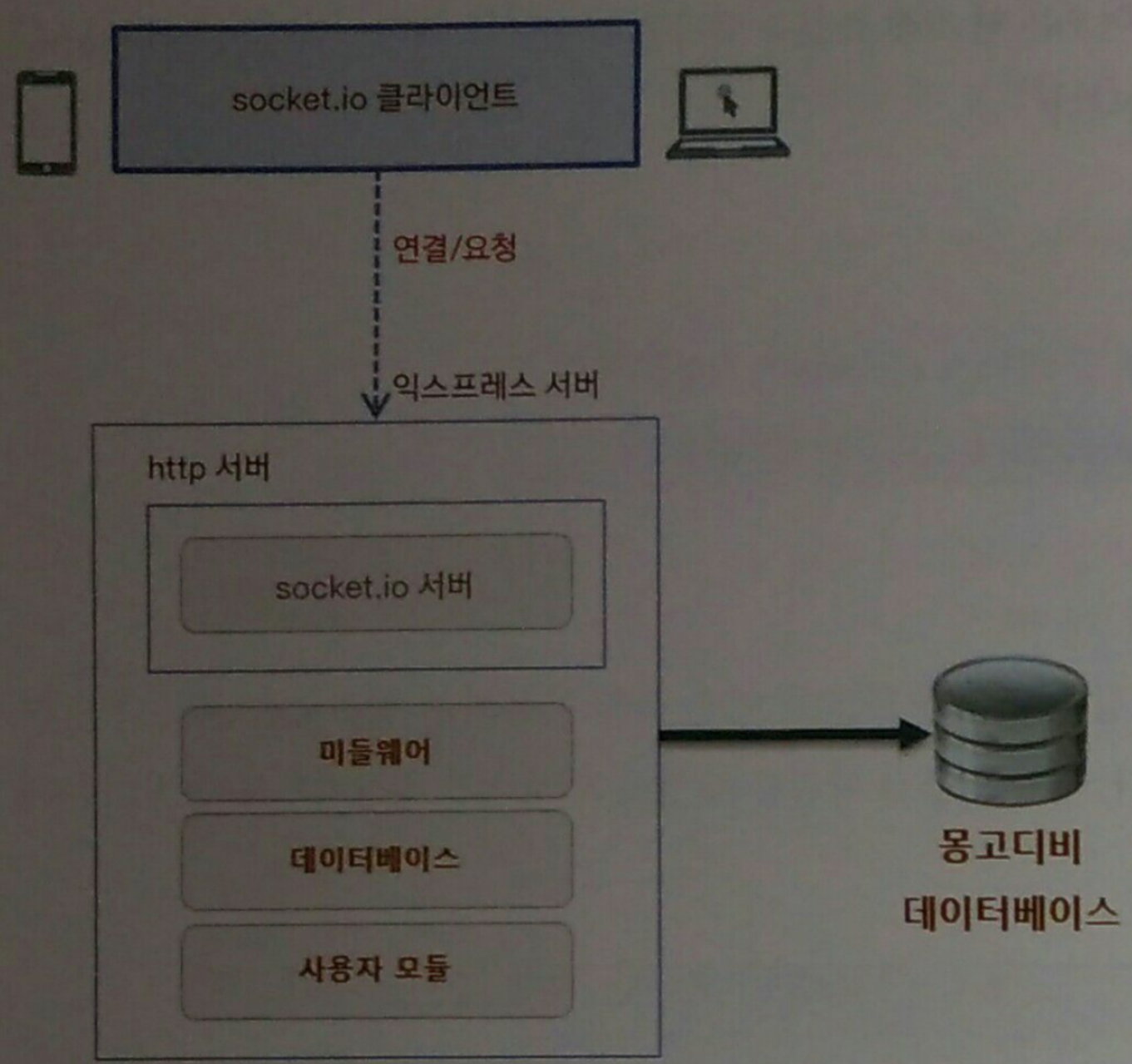
listen( ) 메소드와 attach( ) 메소드는 같은 역할을 하므로 어떤 것을 사용하든 상관없습니다. 다만 여기에서는 listen( ) 메소드를 사용합니다. 이제 app.js 파일을 실행하여 서버가 정상적으로 동작하는지 확인합니다.

```
Console  Debug  Search
ChatExample-app.js [Node Application] Node.js Process
socket.io 요청을 받아들일 준비가 되었습니다.
서버가 시작되었습니다. 포트 : 3000
init() 호출됨.
connect() 호출됨.
데이터베이스에 연결되었습니다. : mongodb://localhost:27017/shopping
설정에 정의된 스키마의 수 : 1
UserSchema 정의함.
./user_schema 모듈을 불러들인 후 스키마 정의함.
users6 컬렉션을 위해 모델 정의함.
스키마 이름 [UserSchema], 모델 이름 [UserModel] 이 database 객체의 속성으로 추가됨.
database 객체가 app 객체의 속성으로 추가됨.
```

▲app.js 파일을 실행하여 서버가 정상적으로 동작하는지 콘솔 창에서 확인한 화면

서버가 실행될 때는 데이터베이스에도 연결되므로 몽고디비 서버가 실행되어 있는지도 확인해야 합니다. 서버를 실행했을 때 콘솔 창을 보면 익스프레스로 만든 웹 서버가 실행되고 그 위에서 socket.io 모듈이 요청을 받아들일 준비가 되었음을 확인할 수 있습니다.

socket.io 모듈과 몽고디비 데이터베이스를 사용하는 구조는 다음과 같습니다.



▲ socket.io 모듈과 몽고디비 데이터베이스를 함께 사용할 때의 구조

socket.io 서버를 웹 서버 위에서 동작하도록 설정하면 웹 소켓과 관련된 요청은 모두 socket.io에서 처리합니다. app.js 파일에 추가한 코드를 보면 socket.io 모듈을 사용하기 위해 listen( ) 메소드를 호출했을 때 socket.io 객체가 반환됩니다. 이 socket.io 객체는 io **변수**에 할당되었는데 이 객체에 들어 있는 sockets 객체는 클라이언트가 접속하거나 데이터를 전송했을 때 이벤트를 발생시킵니다. 따라서 이벤트를 처리할 함수를 on( ) 메소드로 등록하면 이벤트 데이터를 받았을 때 필요한 기능을 실행할 수 있습니다.

클라이언트에서 웹 소켓으로 연결했을 때 발생하는 가장 기본적인 이벤트는 connection 이벤트입니다.

용어! connection 이벤트란? 클라이언트가 웹 소켓으로 서버에 연결했을 때 발생하는 이벤트를 말합니다.

app.js 파일을 열고 socket.io 객체의 listen( ) 메소드를 호출하는 코드 아랫부분에 connection 이벤트 처리 함수를 추가합니다.

참조 파일 | ChatExample>app.js

```
……

// 클라이언트가 연결했을 때의 이벤트 처리
io.sockets.on('connection', function(socket) {
  console.log('connection info : ', socket.request.connection._peername);
```

```
  // 소켓 객체에 클라이언트 Host, Port 정보 속성으로 추가
  socket.remoteAddress = socket.request.connection._peername.address;
  socket.remotePort = socket.request.connection._peername.port;
});
```

on( ) 메소드로 connection 이벤트를 처리하는 콜백 함수를 등록하면 콜백 함수 쪽으로 소켓 객체가 전달됩니다. 이 소켓 객체에는 접속한 클라이언트의 IP 주소와 포트 번호를 확인할 수 있는 속성이 들어 있으므로 그 속성 값을 확인해 접속한 클라이언트의 정보를 로그로 출력합니다.

사용자가 웹 브라우저에서 볼 웹 문서 만들기

이제 사용자가 웹 브라우저에서 보게 될 웹 문서를 만들어 보겠습니다. [public] 폴더 안에 chat01.html 파일을 추가한 후 다음 코드를 입력합니다. 이 웹 문서는 socket.io 클라이언트를 사용해서 웹 서버로 연결합니다.

참조 파일 | ChatExample>/public/chat01.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset = "UTF-8">
    <title>채팅 클라이언트 01</title>
  </head>
  <body>
    <h3>채팅 클라이언트 01</h3>
    <br>
    <div>
      <input type = "text" id = "hostInput" value = "localhost">
      <input type = "text" id = "portInput" value = "3000">
      <input type = "button" id = "connectButton" value = "연결하기">
    </div>

    <hr/>
    <p>결과 : </p>
    <div id = "result"></div>

  </body>
</html>
```

사용자가 보는 화면에는 서버의 IP와 포트 번호를 입력할 수 있는 입력 상자 두 개와 [연결하기] 버튼이 하나 있습니다. 그 아래에는 〈div〉 태그를 하나 추가하고 id 속성 값으로 result를 넣어 둡니다. 이 〈div〉 태그에는 진행 과정을 로그 형태로 출력할 것입니다.

이제 [연결하기] 버튼을 눌렀을 때 socket.io 서버에 연결되도록 자바스크립트 코드를 작성합니다. 여기에서는 jQuery와 socket.io 클라이언트 라이브러리 파일을 사용합니다. jQuery 파일은 jquery.com 사이트에 접속하여 다운로드합니다. 이전에도 jQuery 라이브러리 파일을 사용했으므로 해당 파일을 복사하여 사용하는 것도 좋습니다. socket.io 클라이언트 라이브러리 파일은 socket.io 사이트에 접속하면 다운로드할 수 있습니다. 〈title〉 태그 아래에 다음처럼 외부 자바스크립트 파일을 링크하는 코드를 입력합니다.

참조 파일 | ChatExample>/public/chat01.html

```
.....

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

.....
```

외부 자바스크립트 파일을 링크하기 위해서는 〈script〉 태그에 src 속성만 지정하면 됩니다. jQuery와 socket.io 모두 한 개의 자바스크립트 파일만 링크하여 사용하면 되므로 두 줄을 입력하였습니다. 그 아래에는 〈script〉 태그를 추가한 후 서버에 연결하는 함수를 만듭니다.

참조 파일 | ChatExample>/public/chat01.html

```
.....

<script>
  var host;
  var port;
  var socket;

  // 문서 로딩 후 실행됨
  $(function() {

    $("#connectButton").bind('click', function(event) {
      println('connectButton이 클릭되었습니다.');
```

```
    host = $('#hostInput').val();
    port = $('#portInput').val();

    connectToServer();
  });

});

// 서버에 연결하는 함수 정의
function connectToServer() {

  var options = {'forceNew' : true};
  var url = 'http : //' + host + ' : ' + port;
  socket = io.connect(url, options);

  socket.on('connect', function() {
    println('웹 소켓 서버에 연결되었습니다. : ' + url);
  });

  socket.on('disconnect', function() {
    println('웹 소켓 연결이 종료되었습니다.');
  });

}

function println(data) {
  console.log(data);
  $('#result').append('<p>' + data + '</p>');
}

</script>

.....
```

제이쿼리에서 $(function() { ... } 형태로 함수를 추가하면 문서가 로딩된 후 함수가 자동 실행되므로 초기화에 필요한 코드를 이 함수 안에 넣어 둡니다.

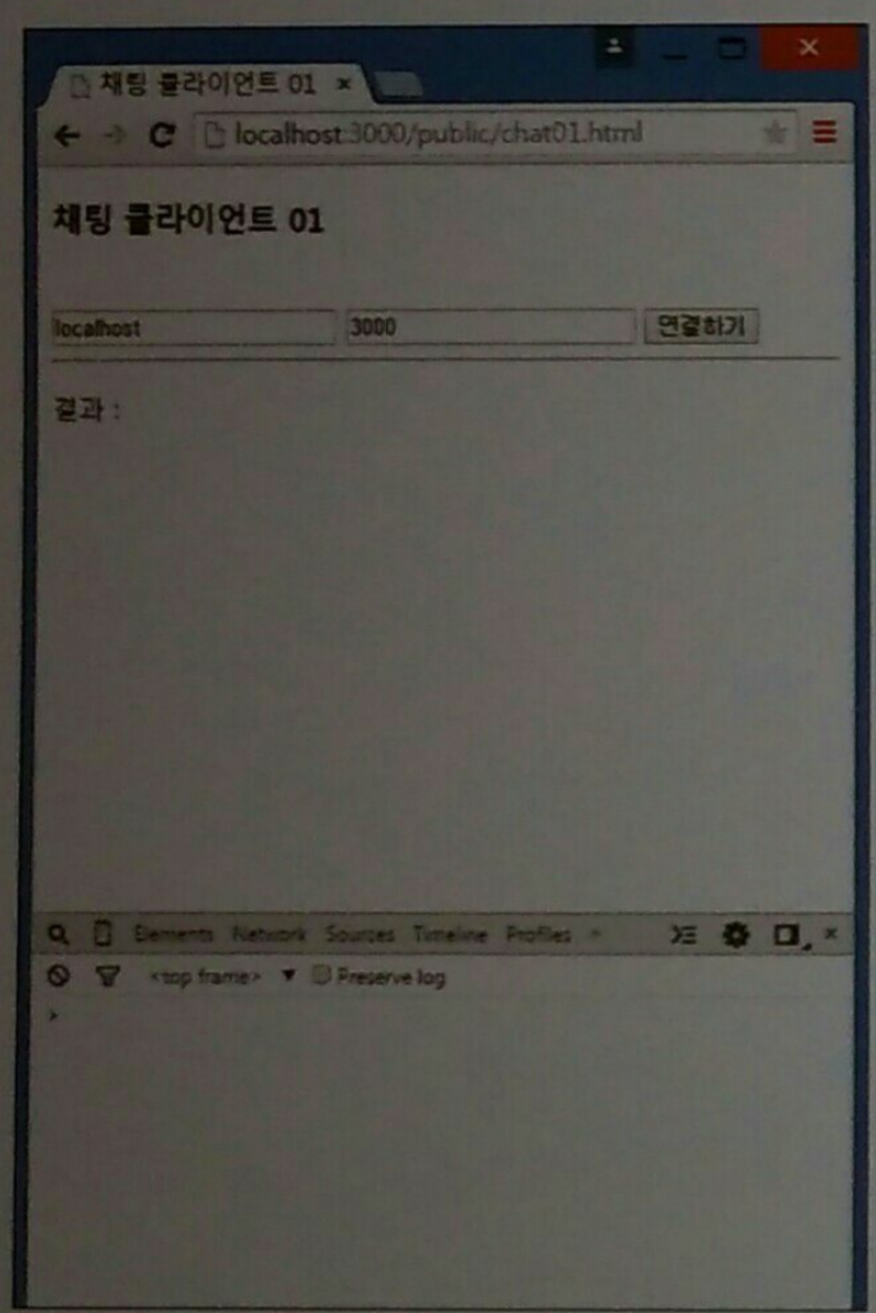
이 함수 안에서는 [연결하기] 버튼의 클릭 이벤트를 처리할 함수를 등록합니다. [연결하기] 버튼의 id 속성 값이 connectButton이므로 이 id 값으로 문서 요소를 찾은 후 bind( ) 메소드로 이벤트를 바인딩합니다.

클릭 이벤트 처리 함수 안에서는 사용자가 입력한 입력 값을 val( ) 함수를 호출하여 가져온 후 connectToServer( ) 함수를 호출합니다. connectToServer( ) 함수에서는 socket.io 클라이언트로 웹 서버에 연결합니다.

socket.io 클라이언트로 웹 서버에 연결하는 방법은 아주 간단합니다. io 객체의 connect( ) 메소드를 호출하면서 서버의 IP와 포트 정보를 넣어주면 됩니다. 이렇게 하면 socket 객체가 반환되는데, 이 객체에서 발생하는 connect와 disconnect 이벤트를 처리하면 서버에 연결되었는지 또는 연결이 끊어졌는지를 알 수 있습니다.

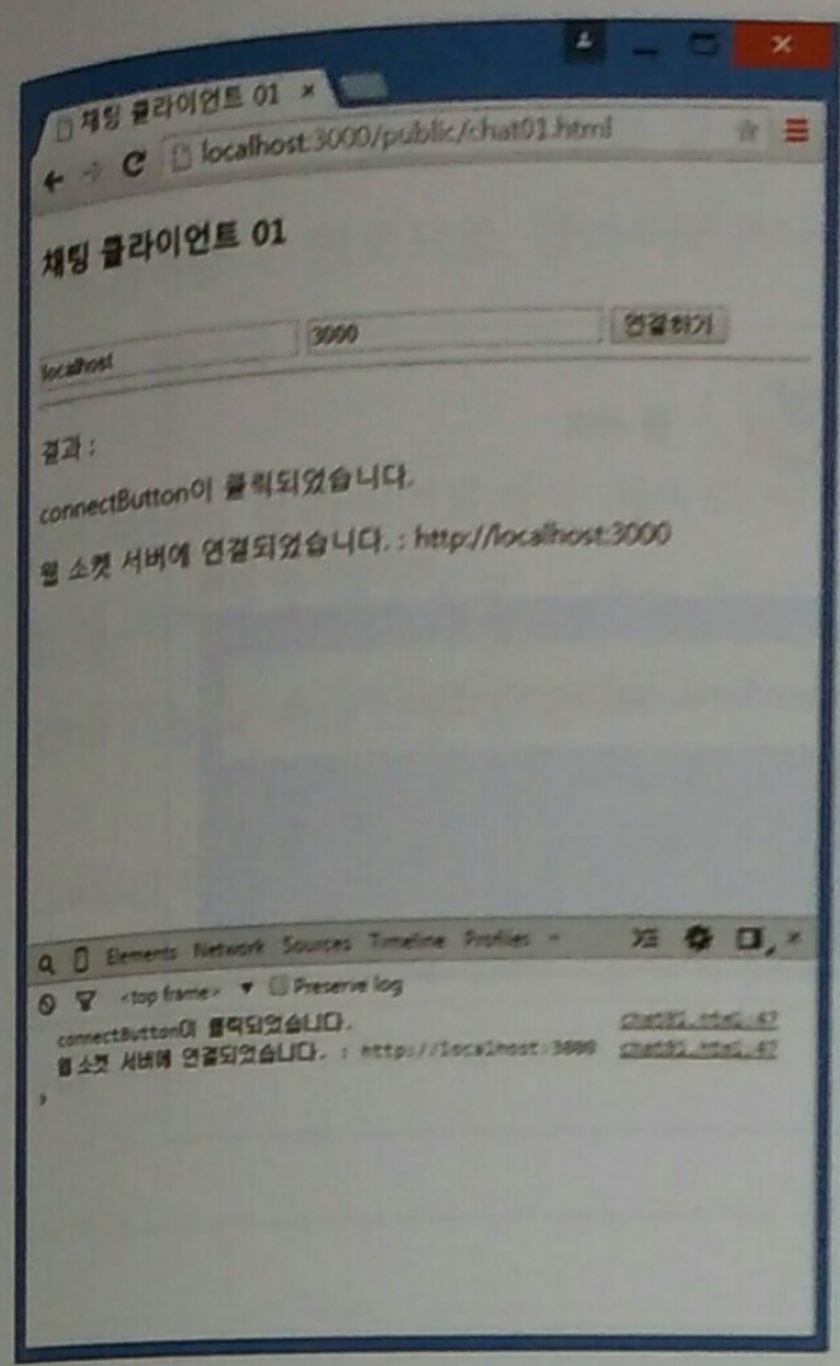
socket.io 모듈로 클라이언트에서 웹 서버로 연결하는 테스트를 진행하기 위해 app.js 파일을 실행합니다. 그다음 웹 브라우저에서 chat01.html 파일을 열면 다음과 같은 웹 페이지를 볼 수 있습니다.

```
▶ http://localhost:3000/public/chat01.html
```



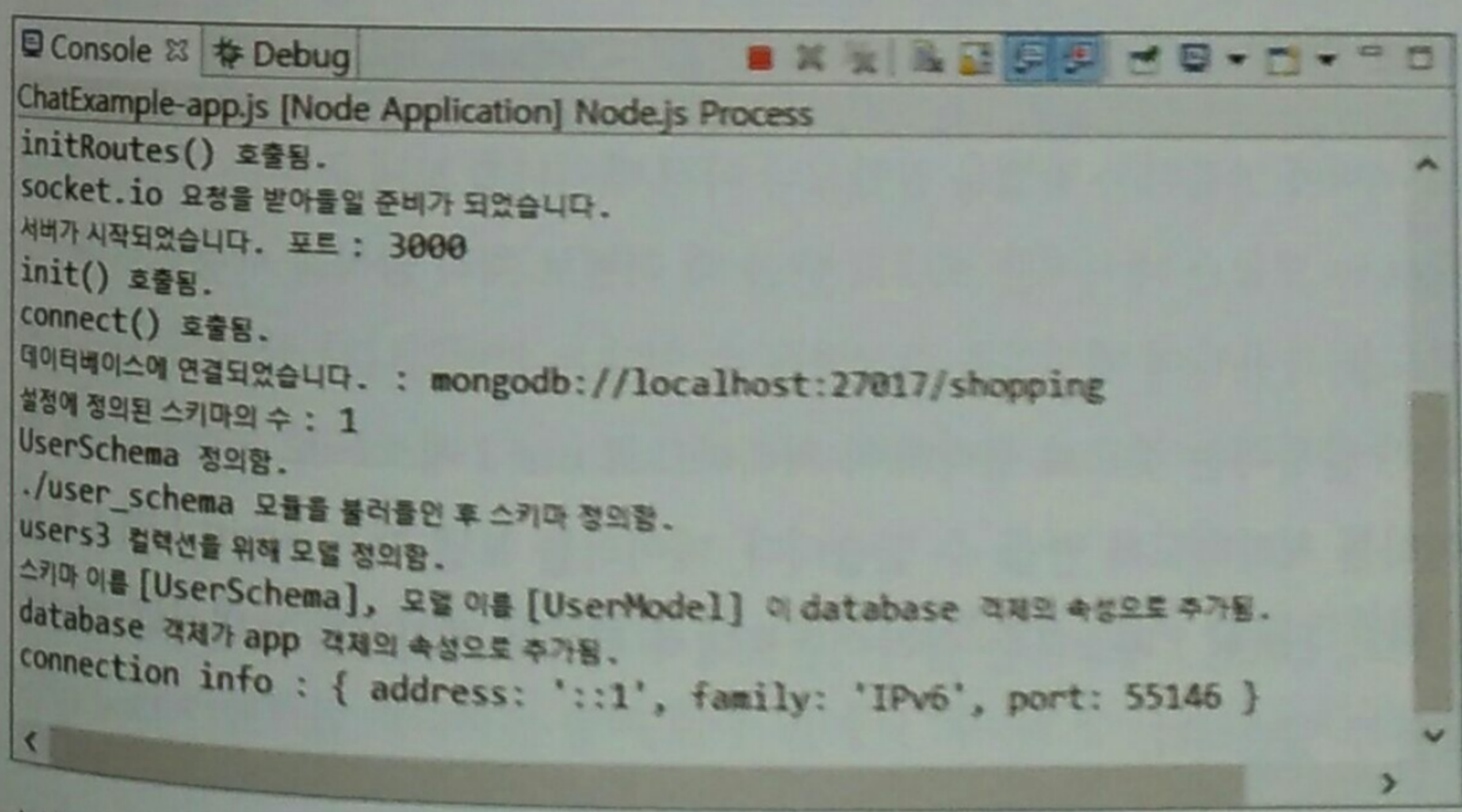
▲ [연결하기] 버튼이 있는 채팅 클라이언트 웹 페이지

크롬 브라우저의 개발자 도구 화면에서 [Console] 탭을 열면 이 탭에서 서버에 연결하는 과정을 로그 메시지로 보여 줍니다. 웹 페이지의 입력 상자에는 localhost와 3000이 입력되어 있습니다. [연결하기] 버튼을 누르면 서버에 연결합니다. 서버 연결에 성공하면 웹 화면과 [Console] 탭에 모두 메시지를 출력합니다. 이 메시지는 서버에 연결되었다는 것을 알려 줍니다.



▲[연결하기] 버튼을 눌러 서버에 연결한 화면

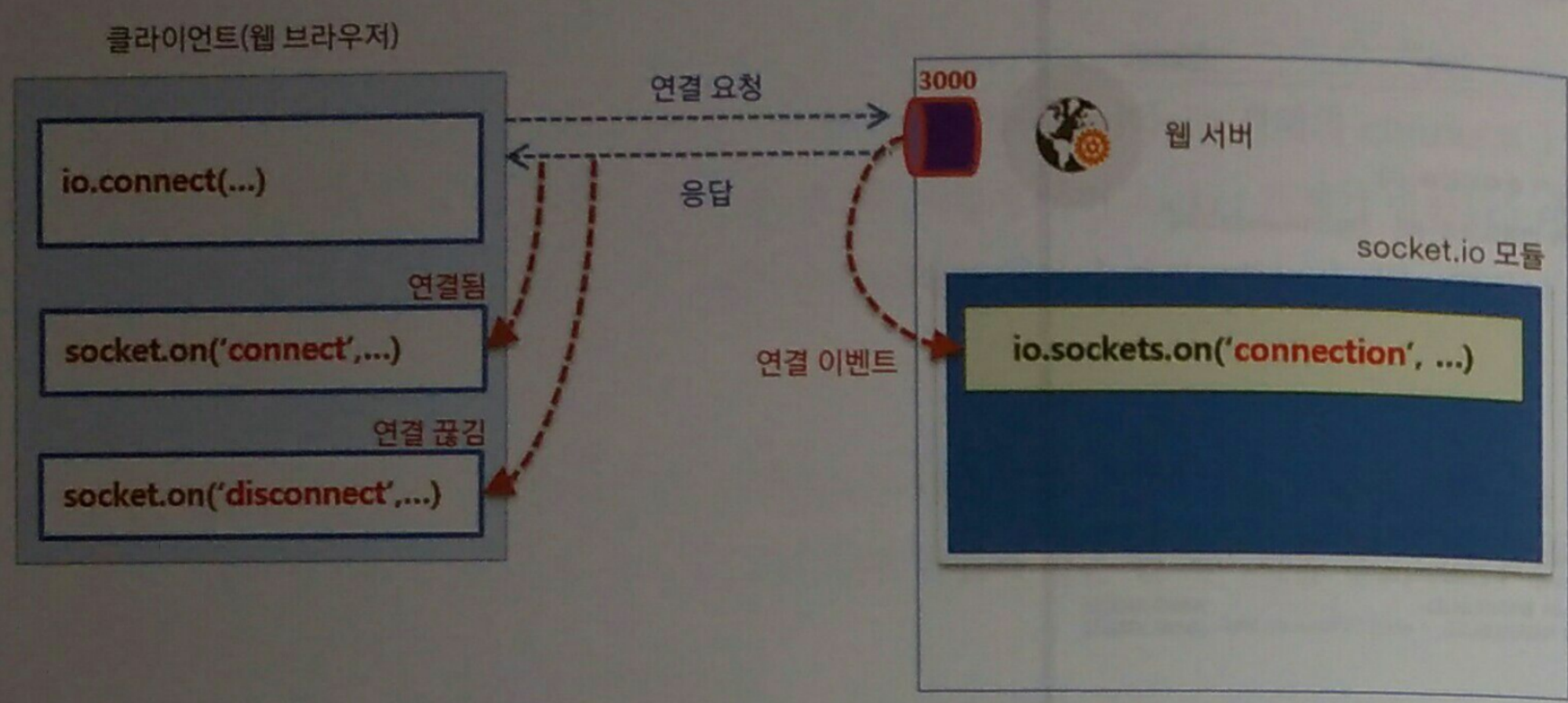
로그는 웹 문서에도 보이고 아래쪽의 개발자 도구 콘솔 창에도 보입니다. 이것은 println( ) 함수가 웹 문서의 〈div〉 태그에 로그 메시지를 추가하면서 동시에 [Console] 탭에도 메시지를 뿌려 주도록 설정했기 때문입니다. 서버 쪽 콘솔 창에도 클라이언트가 접속한 정보가 보이는데, 접속한 클라이언트의 IP와 포트 정보도 확인할 수 있습니다.



▲서버 콘솔 창에 표시된 로그 메시지

같은 PC에서 접속했다면 IP 정보가 제대로 표시되지 않을 수도 있습니다. 하지만 스마트폰 단말이나 다른 PC에서 Wi-Fi에 연결한 후 웹 소켓으로 접속하면 클라이언트의 IP가 정상적으로 표시됩니다.

지금까지 웹 브라우저에서 socket.io 모듈로 서버에 접속하는 코드를 만들어 보았습니다. 접속 과정을 정리하면 다음과 같습니다.



▲ 웹 브라우저에서 socket.io 모듈로 서버에 접속했을 때 처리 과정

웹 브라우저 쪽에서는 connect( ) 메소드를 호출하여 연결을 요청합니다. 이 요청을 받아 웹 소켓이 연결되면 서버 쪽에서는 connection 이벤트가 발생합니다. 이와 마찬가지로 클라이언트 쪽에서도 연결되었을 때 이벤트를 받게 되는데 이벤트의 이름은 connect로 서버 쪽의 이벤트 이름과 조금 다릅니다. 연결이 끊어지면 클라이언트 쪽에 disconnect 이벤트가 발생하며, 서버 쪽에서도 클라이언트가 연결된 소켓 객체에 disconnect 이벤트가 발생합니다.

서버에 보낸 메시지를 그대로 받기

웹 문서에서 socket.io 모듈로 서버에 연결하는 방법을 알았으니 이제 데이터를 보내고 받는 방법을 본격적으로 알아보겠습니다. socket.io 모듈은 메시지를 보내고 받을 때 **이벤트 처리 방식**을 사용합니다. 즉, 이벤트를 처리할 때와 같은 코드를 사용하여 메시지를 주고받을 수 있다는 의미입니다. 클라이언트나 서버는 데이터를 받을 때 이벤트가 발생하는 것으로 생각하여 처리하므로 on( ) 메소드로 이벤트 처리 함수를 등록하면 전달받은 데이터를 처리하도록 만들 수 있습니다. 데이터를 보낼 때는 emit( ) 메소드를 사용해서 이벤트를 일으키면 되고, emit( ) 메소드는 클라이언트에서 서버로 메시지를 보내거나 서버에서 클라이언트로 메시지를 보낼 수 있습니다.

송수신 이벤트 처리 메소드	설명
on(event, callback)	이벤트 수신 형태로 메시지를 수신했을 때 처리할 콜백 함수를 등록합니다. 콜백 함수의 파라미터로 수신한 객체가 전달됩니다.
emit(event, object)	이벤트 송신 형태로 메시지를 송신합니다.

[표] socket.io로 송수신 이벤트를 처리할 때 필요한 메소드

여기에서 이벤트의 이름은 여러분이 마음대로 정할 수 있습니다. 이것은 자바스크립트에서 일반적으로 사용하는 **사용자 정의 이벤트**와 같습니다. 즉, 한쪽에서는 이벤트를 일으키고 다른 쪽에서는 이벤트를 받아 처리할 수 있으므로, 클라이언트나 서버 모두 다른 쪽에서 보내온 이벤트를 받아 처리할 수 있도록 만들면 됩니다.

먼저 하나의 클라이언트에서 서버로 메시지를 보냈을 때 서버에서 메시지를 그대로 다시 돌려보내는 Echo 기능을 만들어 보겠습니다.

용어! Echo 기능은 클라이언트에서 서버로 데이터를 보내고 받는 가장 기초적인 과정으로 네트워킹의 첫 번째 단계로 많이 사용합니다. Echo는 메아리가 울리는 것처럼 서버에 보낸 데이터를 그대로 다시 돌려받는다는 의미입니다.

[public] 폴더의 chat01.html 파일을 복사하여 chat02.html 파일을 만든 후 다음과 같이 connect 이벤트를 처리하는 콜백 함수 안에서 on( ) 메소드를 호출하여 message 사용자 정의 이벤트를 처리합니다.

참조 파일 | ChatExample>/public/chat02.html

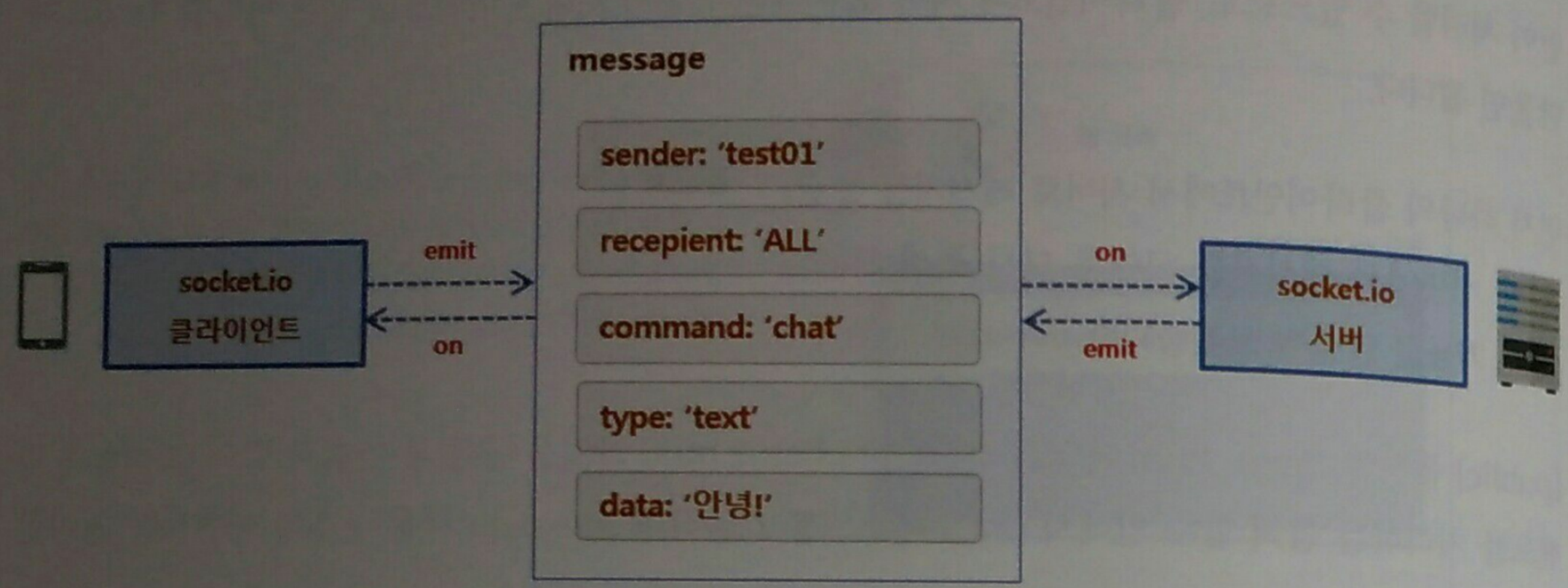
```
.....

  socket.on('connect', function() {
    println('웹 소켓 서버에 연결되었습니다. : ' + url);

    socket.on('message', function(message) {
      console.log(JSON.stringify(message));

      println('<p>수신 메시지 : ' + message.sender + ', ' + message.recepient + ', '
            + message.command + ', ' + message.type + ', ' + message.data + '</p>');
    });

  });

.....
```

클라이언트와 서버 간에 데이터를 주고받기 위해서는 먼저 클라이언트가 서버에 연결되어야 하므로 connect 이벤트를 처리하는 콜백 함수 안에서 데이터 송수신과 관련된 이벤트를 처리하도록 만듭니다.

정박사의 한마디 서버에 연결되기 전에 데이터 송수신 관련 이벤트를 처리하도록 등록해도 상관없어요

서버에 연결된 후 데이터 송수신과 관련된 사용자 정의 이벤트를 등록하는 것이 일반적입니다. 하지만 connect 이벤트를 등록하는 콜백 함수 바깥에서 사용자 정의 이벤트를 등록해도 별다른 문제는 생기지 않습니다.

데이터를 주고받기 위해 message 이벤트를 하나 정의하였는데, 이 이벤트를 사용해서 보낼 데이터에는 다음과 같은 속성이 들어가도록 정의합니다.



▲ 새로 정의한 message 객체의 속성들

sender 속성에는 보내는 사람의 아이디를 넣고 recepient 속성에는 받는 사람의 아이디를 넣습니다. 글자만 받아서 보여 주는 것이 아니라 다른 기능도 수행할 수 있도록 만들기 위해 보내는 데이터의 종류를 구별할 수 있는 command 속성을 추가합니다. 이 command 속성에는 단순히 채팅에 필요한 글자를 전송한다는 의미로 chat 값을 넣어 줍니다. type 속성은 전송될 데이터의 형태를 지정합니다. 이 type 속성에는 여러 가지 값을 넣을 수 있는데 text 값은 단순 텍스트 형태를 의미합니다. data 속성에는 데이터가 들어갑니다. '안녕!'처럼 글자를 넣어서 보내 주면 클라이언트가 이 글자를 화면에 보여 줄 수 있습니다. 이렇게 만든 message 객체의 포맷은 여러분이 원하는 대로 바꿀 수 있습니다. 'message'라는 이름도 마음대로 바꿀 수 있고 message 객체의 포맷도 마음대로 바꿀 수 있으니 데이터를 주고받는 데 제약은 거의 없는 셈입니다.

이제 웹 문서에 데이터를 전송하는 입력 상자와 버튼을 만들어 보겠습니다. chat02.html 파일을 열고 〈body〉 태그 안의 [연결하기] 버튼 아래쪽에 다음 코드를 입력합니다.

참조 파일 | ChatExample>/public/chat02.html

```
……

  <div>
    <div>
        <span>보내는 사람 아이디 : </span>
        <input type="text" id="senderInput" value="test01">
    </div>
    <div>
```

```
        <span>받는 사람 아이디 : </span>
        <input type="text" id="recepientInput" value="ALL">
    </div>
    <div>
        <span>메시지 데이터 : </span>
        <input type="text" id="dataInput" value="안녕!">
    </div>
    <br>
    <input type="button" id="sendButton" value="전송">
  </div>

  ……,
```

메시지를 보내는 사람과 받는 사람의 아이디를 입력하는 입력 상자와 전송할 메시지를 입력하는 입력 상자가 추가되었습니다. [전송] 버튼도 추가했으며 버튼의 id 속성 값은 sendButton을 입력하였습니다.

파일 위쪽의 〈script〉 태그 안을 보면 [연결하기] 버튼을 클릭했을 때 발생하는 이벤트를 처리하는 코드가 있는데 그 아랫부분에 [전송] 버튼 클릭 이벤트를 처리하는 코드를 추가합니다.

참조 파일 | ChatExample3/public/chat02.html

```
……,

// 전송 버튼을 클릭하면 처리
$("#sendButton").bind('click', function(event) {
  var sender = $('#senderInput').val();
  var recepient = $('#recepientInput').val();
  var data = $('#dataInput').val();

  var output = {sender : sender, recepient : recepient, command : 'chat', type : 'text', data :
data};
  console.log('서버로 보낼 데이터 : ' + JSON.stringify(output));

  if(socket == undefined) {
    alert('서버에 연결되어 있지 않습니다. 먼저 서버에 연결하세요.');
    return;
  }

  socket.emit('message', output);
});

……,
```

사용자가 세 개의 입력 상자에 데이터를 입력하면 그 값을 가져와 사용합니다. 첫 번째 값은 보낼 사람의 아이디인데 test01과 같이 입력할 수 있습니다. 두 번째 값은 받을 사람의 아이디이며 ALL을 입력합니다. 그러면 연결되어 있는 모든 사람에게 메시지를 전송합니다. 세 번째 값으로는 서버에 전송할 데이터를 입력받게 됩니다.

JSON 객체로 만든 데이터는 message라는 이벤트 이름과 함께 emit( ) 메소드를 사용해 서버로 전송됩니다. 이렇게 전송된 데이터는 서버에서 on( ) 메소드로 등록한 함수가 이벤트를 받아 처리합니다. app.js 파일을 복사하여 app2.js 파일을 만들고 다음처럼 message 이벤트를 처리하는 코드를 추가합니다.

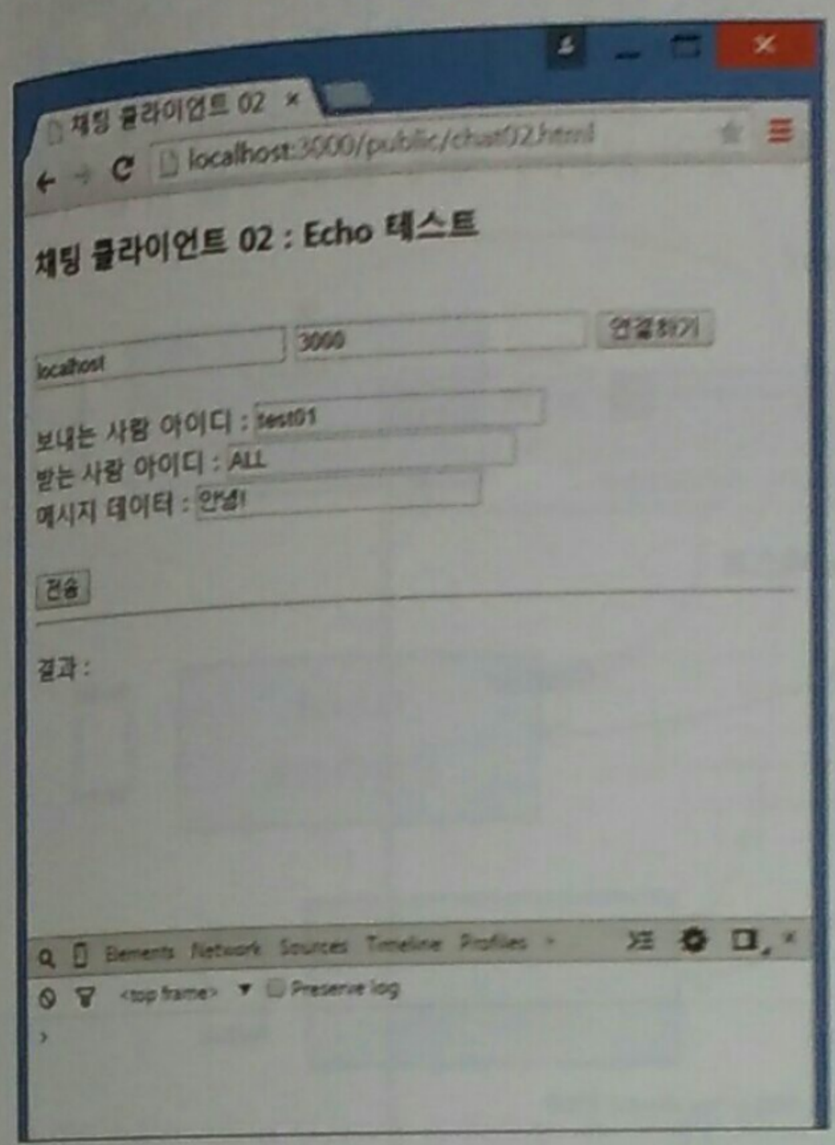
참조 파일 | ChatExample>app2.js

```
……

// 클라이언트가 연결했을 때의 이벤트 처리
io.sockets.on('connection', function(socket) {
……

  // 'message' 이벤트를 받았을 때의 처리
  socket.on('message', function(message) {
    console.log('message 이벤트를 받았습니다.');
    console.dir(message);

    if(message.recepient ='ALL') {
      // 나를 포함한 모든 클라이언트에게 메시지 전달
      console.dir('나를 포함한 모든 클라이언트에게 message 이벤트를 전송합니다.')
      io.sockets.emit('message', message);
    }
  });
});

……
```

클라이언트가 연결되었을 때 처리하는 콜백 함수 안에서 message 이벤트를 처리하도록 on( ) 메소드로 콜백 함수를 등록합니다. message 이벤트를 받으면 message 객체가 전달되므로 이 객체에 들어 있는 속성을 확인합니다. recepient 속성 값이 'ALL'이면 나를 포함한 모든 클라이언트에 message 이벤트를 보냅니다. 서버에 연결된 모든 클라이언트 소켓은 io.sockets 안에 들어 있으니 이 소켓들 모두에 데이터를 전송하면 될 것입니다. 그런데 소켓들을 일일이 찾아 데이터를 전송하려면 어떻게 해야 할까요? 이 io.sockets 객체의 emit( ) 메소드를 호출하면 연결된 모든 클라이언트로 메시지를 보낼 수 있습니다. 여기에서는 클라이언트로부터 받은 메시지를 수정하지 않고 그대로 모든 클라이언트에 보내 줍니다.

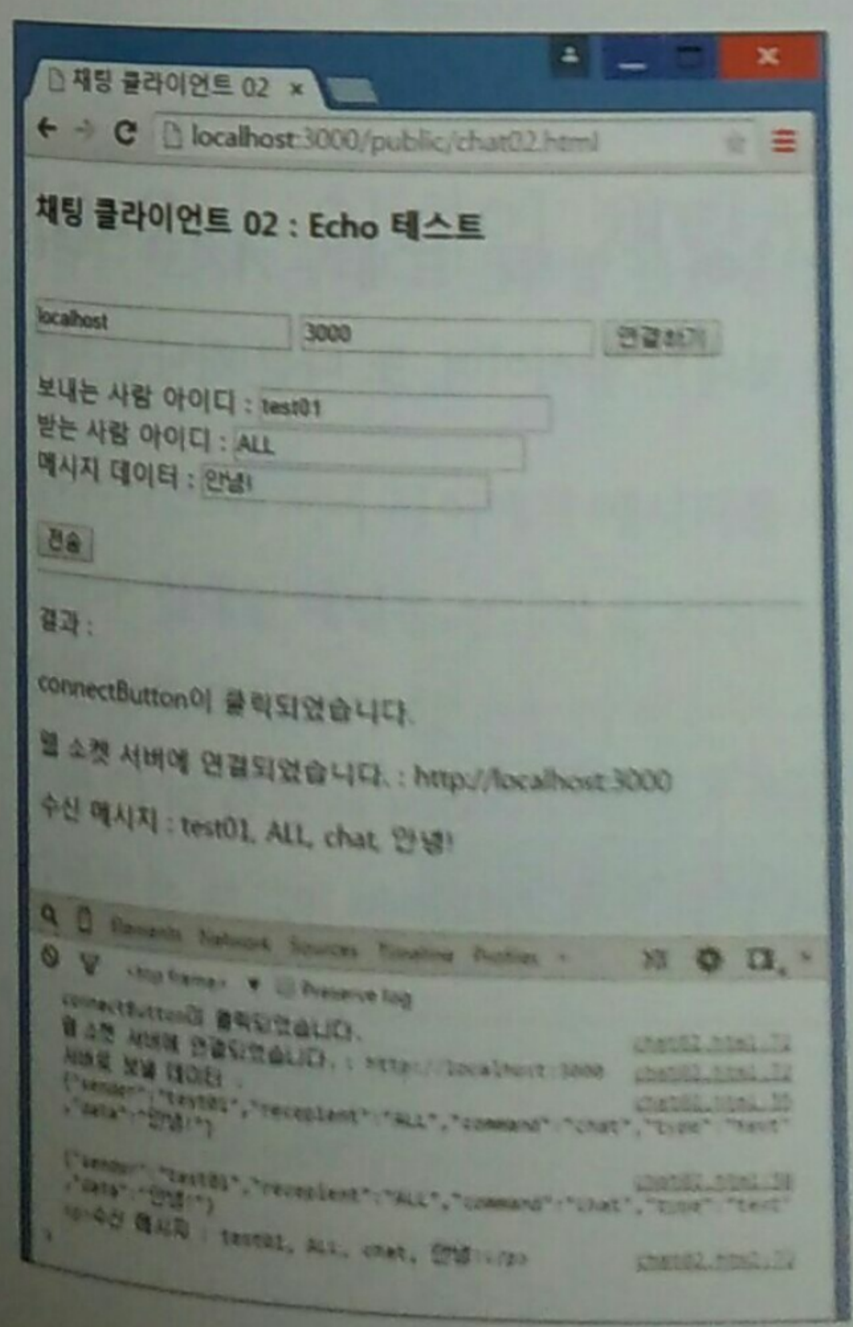
app2.js 파일을 실행한 후 웹 브라우저에서 chat2.html 파일을 열면 다음과 같은 결과 화면이 나타납니다.

▶ http://localhost:3000/public/chat2.html



◀ chat2.html 파일을 웹 브라우저에서 열었을 때의 화면

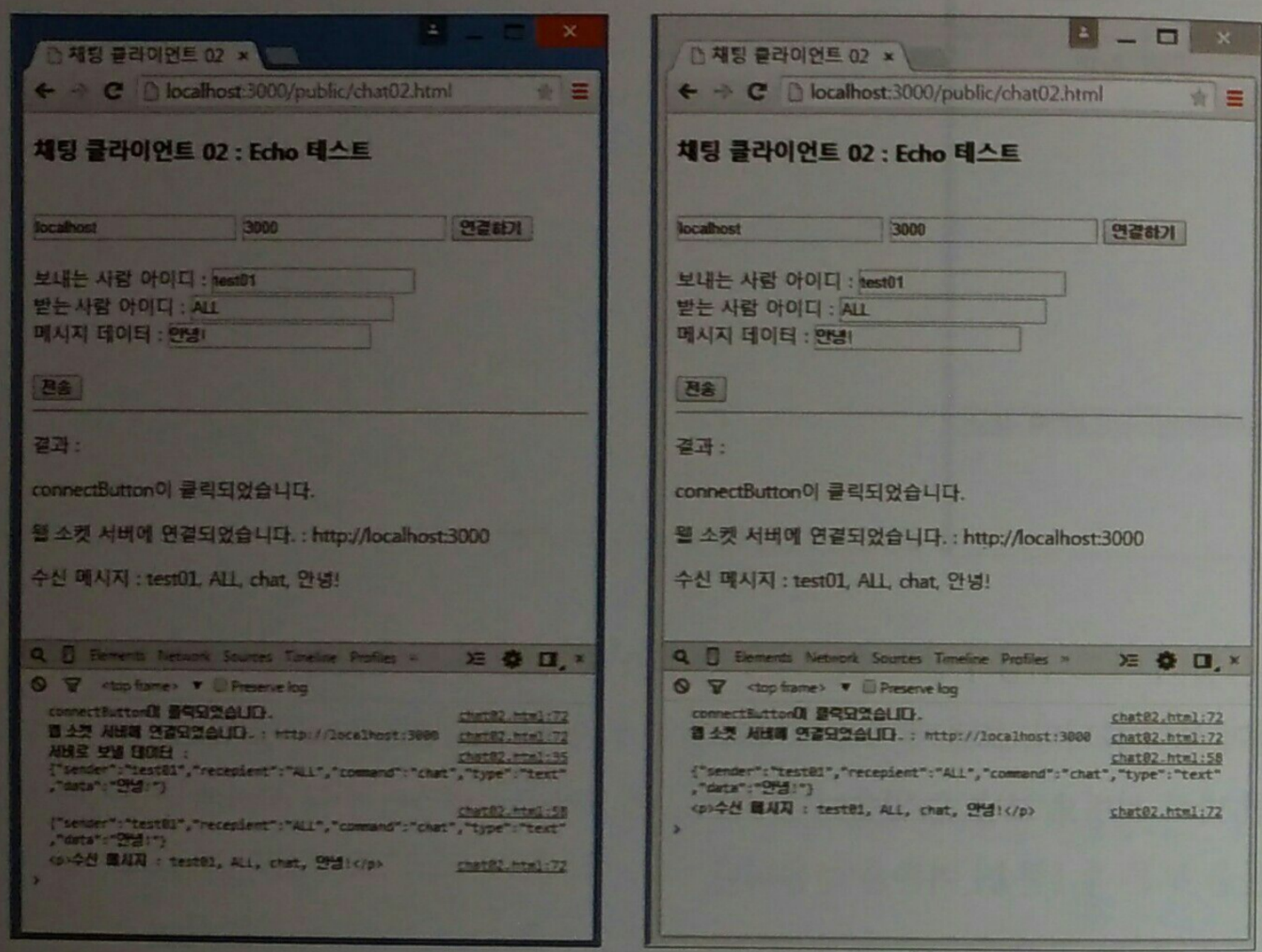
보내는 사람과 받는 사람의 아이디 입력 상자에는 각각 'test01'과 'ALL'이 들어 있습니다. 아래쪽에는 개발자 도구 화면을 열어 보여 줍니다. 보낼 메시지 데이터 입력 상자에는 '안녕!'이라고 입력되어 있으므로 [전송] 버튼을 누르면 바로 서버로 보낼 수 있습니다. 서버로 메시지를 보내려면 먼저 서버에 연결해야 하므로 [연결하기] 버튼을 누른 후 [전송] 버튼을 누릅니다.



◀ [연결하기] 버튼을 누른 후 [전송] 버튼을 눌렀을 때의 결과 화면

[전송] 버튼을 누르면 서버로 메시지를 보낸 후 다시 받아서 화면에 표시합니다. 아래쪽에 있는 개발자 도구의 콘솔 창을 보면 서버로 보낼 때와 서버에서 받을 때의 상태 및 message 객체에 들어 있는 속성 값을 확인할 수 있습니다.

클라이언트를 하나의 브라우저가 아닌 여러 개의 브라우저에서 띄운 상태라면 각각의 클라이언트에서 서버에 연결한 후 한쪽에서 보내면 모두 받는 것을 알 수 있습니다. 이것은 하나의 클라이언트로부터 받은 메시지를 서버에서 모든 클라이언트로 전송하기 때문입니다. 두 개의 브라우저에 웹 문서를 똑같이 띄운 후 확인해 보면 다음과 같은 결과 화면을 볼 수 있습니다.



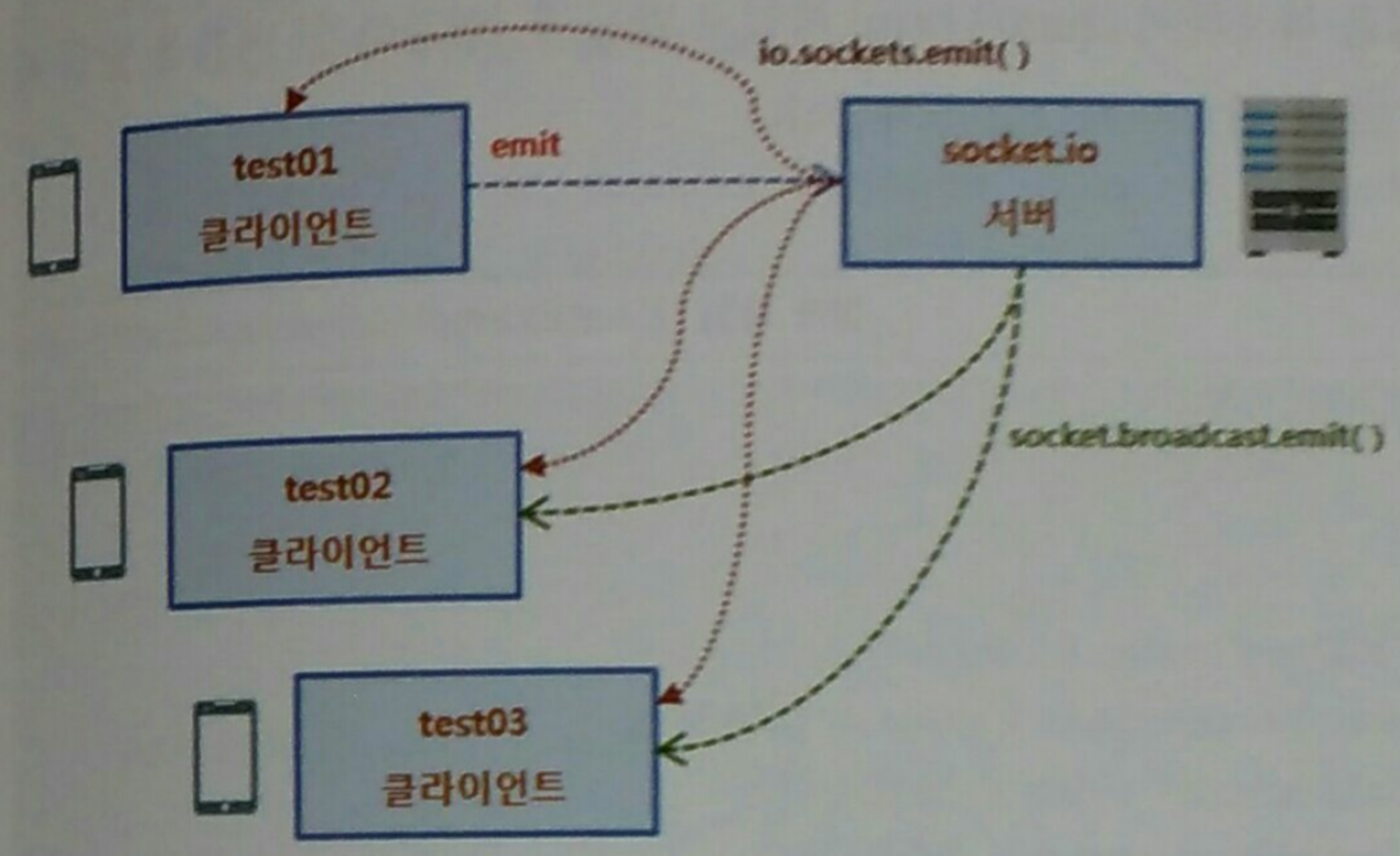
▲ 두 개의 클라이언트가 연결되어 한쪽에서 보냈을 때 모두 메시지를 수신

socket.io 서버에 접속되어 있는 모든 클라이언트에 메시지를 전송하는 방식은 크게 두 가지로 나뉩니다. 하나는 지금 본 것처럼 나를 포함한 모든 사람에게 메시지를 보내는 방식이며, 또 다른 하나는 나를 제외한 나머지 모두에게 보내는 방식입니다.

메소드 이름	설명
io.sockets.emit(event, object)	나를 포함한 모든 클라이언트에 전송합니다.
socket.broadcast.emit(event, object)	나를 제외한 모든 클라이언트에 전송합니다.

[표] 여러 사람에게 메세지를 전송하는 두 가지 방식

보낸 사람을 포함한 모든 클라이언트에 메시지를 전송하고 싶다면 io.sockets 객체의 emit( ) 메소드를 호출합니다. io.sockets 객체는 현재 연결되어 있는 모든 소켓 객체를 의미하며, 각 클라이언트의 소켓 정보를 가지고 있습니다. 보내는 사람을 제외한 모든 클라이언트에 전송하고 싶은 경우에는 하나의 소켓 객체에 들어 있는 **broadcast 객체**를 사용합니다. 이 socket.broadcast 객체는 주체가 되는 socket 객체를 제외한 나머지 소켓 객체를 가리키며, broadcast 객체의 emit( ) 메소드를 호출하면 이 socket 객체를 제외한 나머지 소켓 객체를 사용해 메시지를 전송합니다.



▲모든 클라이언트에 전송하는 경우와 보내는 사람을 제외하고 전송하는 경우

지금까지 하나의 클라이언트가 보내온 메시지를 연결되어 있는 모든 클라이언트에 보내는 기능을 간단하게 만들어 보았습니다. 간단한 기능이지만 노드와 socket.io 모듈을 사용하면 그리 많지 않은 코드로도 훌륭한 실시간 채팅 기능을 만들 수 있음을 엿볼 수 있을 것입니다.

10-2 | 일대일 채팅하기

클라이언트와 서버 사이에서 메시지를 주고받을 수 있게 되었으니 이제 일대일 채팅 기능을 만들어 보겠습니다. 일대일 채팅은 상대방을 지정하여 메시지를 보내야 하므로 서버에 연결된 각 클라이언트마다 고유한 정보가 있어야 합니다. 이 때문에 socket.io 모듈은 접속한 클라이언트마다 고유한 id를 만들어 주는데 이 id 값은 사용자가 구별하기 힘들 정도로 복잡한 형태입니다. 따라서 클라이언트가 로그인할 때 사용하는 로그인 아이디를 사용해서 클라이언트를 구별할 수 있도록 만드는 것이 좋습니다.

정박사의 한마디 모바일의 경우 전화번호를 아이디로 사용할 수 있어요

카카오톡이나 라인과 같은 모바일용 메신저는 단말의 전화번호부로 상대방의 정보를 관리합니다. 이 경우에 국가번호를 포함한 단말의 전화번호를 고유한 아이디로 사용할 수 있습니다.

로그인한 아이디로 접속한 클라이언트를 구별하려면 클라이언트에서 서버에 연결한 후 로그인하도록 수정해야 합니다. chat02.html 파일을 복사하여 chat03.html 파일로 만든 후 [연결하기] 버튼을 만들 때 추가한 태그 아래에 다음처럼 로그인에 필요한 태그를 추가합니다.

참조 파일 | ChatExample>/public/chat03.html

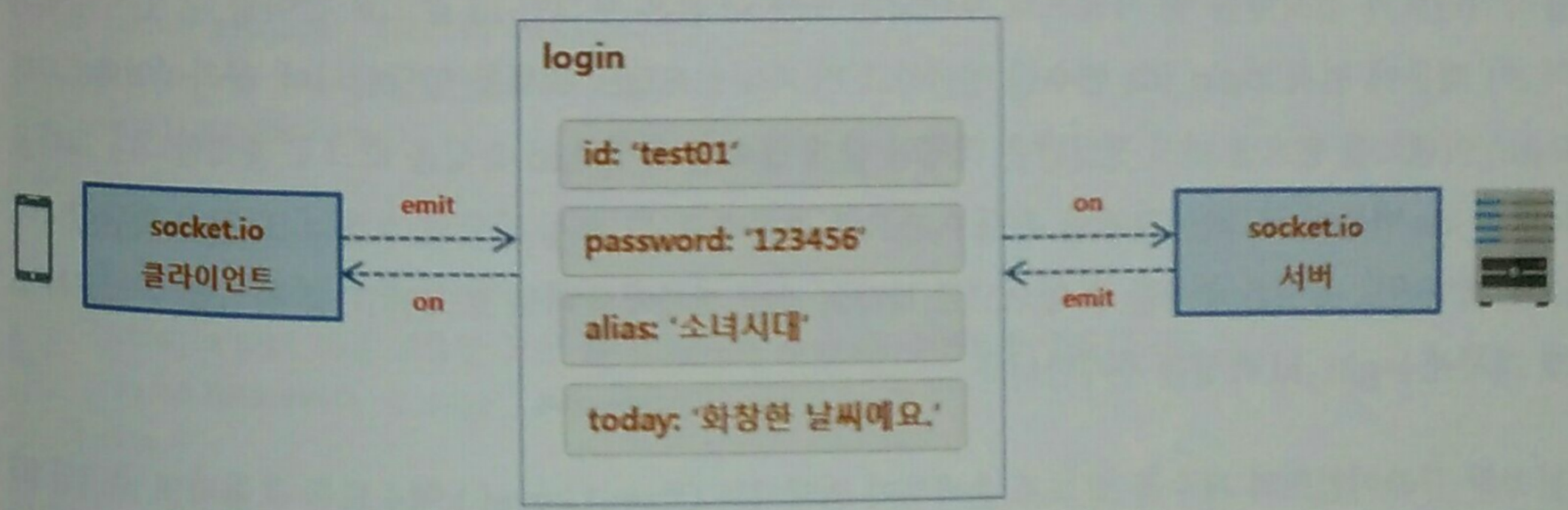
```
.....

  <div>
    <input type = "text" id = "idInput" value = "test01">
    <input type = "password" id = "passwordInput" value = "123456">
    <input type = "text" id = "aliasInput" value = "소녀시대">
    <input type = "text" id = "todayInput" value = "좋은 날!">

    <input type = "button" id = "loginButton" value = "로그인">
    <input type = "button" id = "logoutButton" value = "로그아웃">
  </div>

.....
```

로그인하는 사람의 아이디, 비밀번호, 별명을 입력할 수 있는 입력 상자를 추가합니다. 그리고 그 아래에는 자신의 상태 메시지를 입력할 수 있는 입력 상자도 추가합니다. 비밀번호가 화면에 그대로 드러나지 않도록 비밀번호를 입력하는 입력 상자의 type 속성 값에 password를 넣어 줍니다. [로그인] 버튼과 [로그아웃] 버튼의 id 속성은 각각 loginButton과 logoutButton으로 지정합니다.

파일 위쪽의 <script> 태그 안에는 bind( ) 메소드로 [로그인] 버튼을 클릭했을 때 처리할 함수를 등록합니다.

```
……,

  // 로그인 버튼을 클릭하면 처리
  $("#loginButton").bind('click', function(event) {
    var id = $('#idInput').val();
    var password = $('#passwordInput').val();
    var alias = $('#aliasInput').val();
    var today = $('#todayInput').val();

    var output = {id : id, password : password, alias : alias, today : today};
    console.log('서버로 보낼 데이터 : ' + JSON.stringify(output));

    if(socket == undefined) {
      alert('서버에 연결되어 있지 않습니다. 먼저 서버에 연결하세요.');
      return;
    }

    socket.emit('login', output);
  });

……
```

사용자가 입력한 데이터를 모두 변수에 넣은 후 하나의 객체로 만듭니다. 이 객체는 login 사용자 정의 이벤트를 사용해 서버로 전송됩니다. 즉, 로그인을 위해 전송하는 객체의 포맷은 다음과 같이 정의합니다.



▲로그인을 위해 서버에 전송하는 객체의 속성들

이렇게 전송한 로그인 이벤트를 처리할 수 있는 코드를 서버 쪽에 추가합니다. app2.js 파일을 복사하여 app3.js 파일로 만든 후 다음처럼 login 이벤트를 처리하는 함수를 추가합니다.

```
……

// 로그인 아이디 매핑(로그인 ID -> 소켓 ID)
var login_ids = {};

……

// 'login' 이벤트를 받았을 때의 처리
socket.on('login', function(login) {
  console.log('login 이벤트를 받았습니다.');
  console.dir(login);

  // 기존 클라이언트 ID가 없으면 클라이언트 ID를 맵에 추가
  console.log('접속한 소켓의 ID : ' + socket.id);
  login_ids[login.id] = socket.id;
  socket.login_id = login.id;

  console.log('접속한 클라이언트 ID 개수 : %d', Object.keys(login_ids).length);

  // 응답 메시지 전송
  sendResponse(socket, 'login', '200', '로그인되었습니다.');
});

……
```

클라이언트가 접속했을 때 자동으로 부여되는 소켓 id 값은 로그인 id 값으로 찾아낼 수 있어야 합니다. 이 때문에 먼저 **login_ids 변수**를 선언하고 빈 자바스크립트 객체를 할당합니다. 클라이언트로부터 login 이벤트를 받으면 먼저 전달받은 객체와 함께 접속한 소켓의 id 속성을 로그로 출력합니다. 그다음 login_ids 변수에 로그인 ID 값을 속성 이름으로 추가하고, 그 속성 값으로 소켓의 ID 값을 할당합니다. 소켓 객체에도 속성을 추가할 수 있으므로 각각의 소켓 객체에서 바로 로그인 ID를 확인할 수 있도록 소켓 객체에 login_id 속성을 추가합니다.

접속한 클라이언트의 개수를 로그로 출력하기 위해서는 **Object.keys( ) 메소드**를 호출하여 로그인 ID들을 배열로 만든 후 **length 속성** 값을 확인합니다. 마지막에는 **sendResponse( ) 메소드**를 호출했는데 이 메소드는 클라이언트에 처리 상태를 알려주기 위해 정의한 함수입니다.

참조 파일 | ChatExample>app3.js

```
……,

// 응답 메시지 전송 메소드
function sendResponse(socket, command, code, message) {
  var statusObj = {command : command, code : code, message : message};
  socket.emit('response', statusObj);
}

……,
```

클라이언트가 로그인 이벤트를 서버로 보냈을 때 정상적으로 처리되면 클라이언트에 성공했다는 응답 메시지를 보냅니다. 응답을 보낼 때 전달되는 객체의 속성에는 명령 문자열, 코드 값, 메시지 문자열이 들어갑니다. 명령 문자열은 어떤 이벤트의 응답인지를 알려 주는 파라미터이고, 코드와 메시지 값은 상태를 알려 주는 파라미터입니다.

로그인했을 때 로그인 ID를 사용해서 소켓 ID를 찾을 수 있도록 만들었습니다. 따라서 message 이벤트를 받았을 때 처리하는 코드를 다음처럼 수정합니다.

참조 파일 | ChatExample>app3.js

```
……,

// message 이벤트를 받았을 때의 처리
socket.on('message', function(message) {
  console.log('message 이벤트를 받았습니다.');
  console.dir(message);

  if(message.recepient ='ALL') {
    // 나를 제외한 모든 클라이언트에 메시지 전달
    console.dir('나를 포함한 모든 클라이언트에 message 이벤트를 전송합니다.')
    io.sockets.emit('message', message);
  } else {
    // 일대일 채팅 대상에게 메시지 전달
    if(login_ids[message.recepient]) {
      io.sockets.connected[login_ids[message.recepient]].emit('message', message);

      // 응답 메시지 전송
      sendResponse(socket, 'message', '200', '메시지를 전송했습니다.');
```

```
    } else {
        // 응답 메시지 전송
        sendResponse(socket, 'login', '404', '상대방의 로그인 ID를 찾을 수 없습니다.');
    }
  }
});

......
```

message 이벤트와 함께 전달되는 객체의 속성 중에서 recepient 속성 값이 ALL이 아닌 경우에는 특정 대상에 메시지를 전송해야 하므로 해당 소켓을 먼저 찾아야 합니다. io.sockets.connected 객체에 들어 있는 소켓들은 제각각 소켓 ID를 가지고 있습니다. 따라서 대상 소켓을 찾으려면 다음 코드를 사용해야 합니다.

```
io.sockets.connected[login_ids[message.recepient]]
```

클라이언트가 로그인하지 않으면 login_ids 객체에 로그인 ID가 없을 수도 있습니다. 따라서 if문을 사용해 로그인 ID가 속성으로 들어 있는지 확인한 후 대상 소켓에 있는 emit( ) 메소드를 호출하여 메시지를 전송합니다. 메시지를 전송한 후에는 메시지를 요청한 클라이언트에 전송이 완료되었다는 응답 메시지를 보냅니다.

▲클라이언트가 연결되었을 때 connected 객체에 보관되는 소켓 객체들

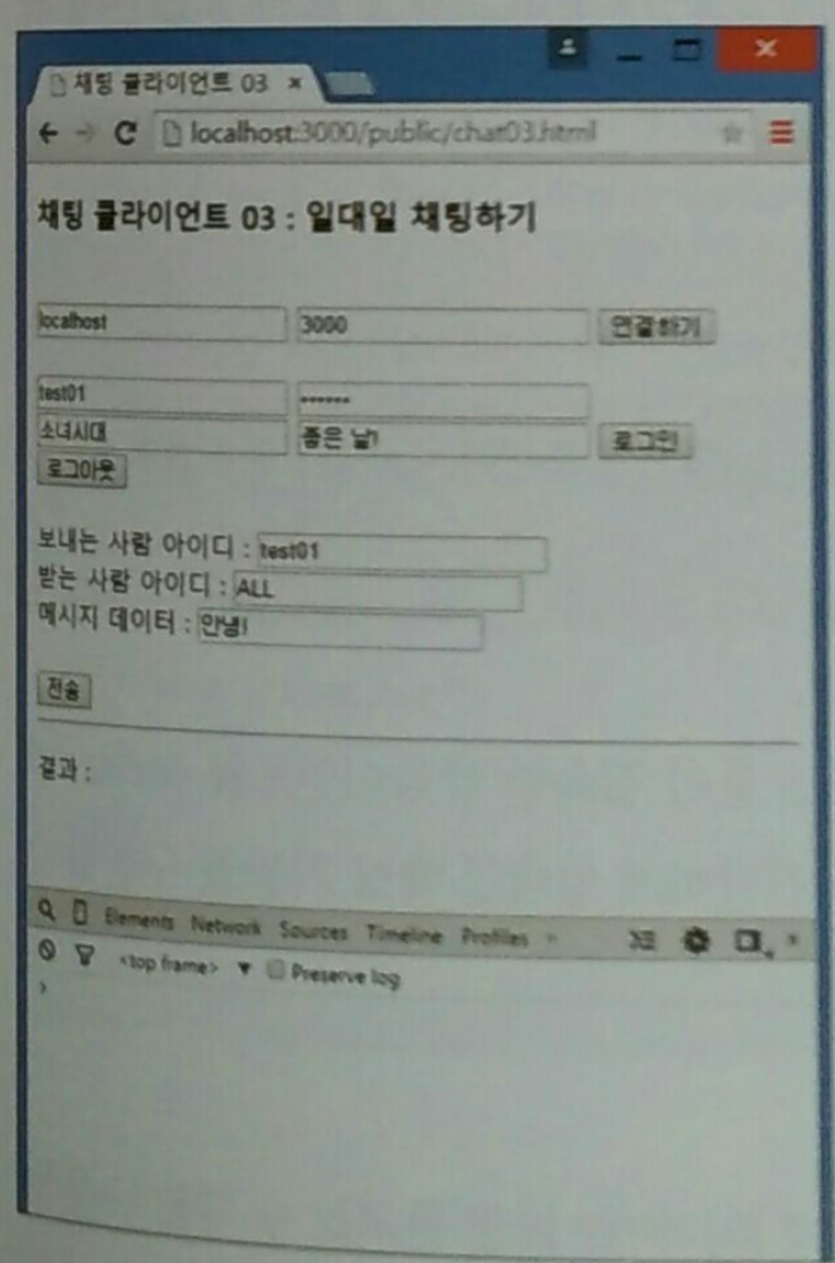
마지막으로 chat03.html 파일의 자바스크립트 코드 입력 부분에 response 이벤트를 처리하는 함수를 추가합니다.

참조 파일 | ChatExample>/public/chat03.html

```
……

socket.on('response', function(response) {
  console.log(JSON.stringify(response));
  println('응답 메시지를 받았습니다. : ' + response.command + ', ' + response.code
          + ', ' + response.message);

});

……
```

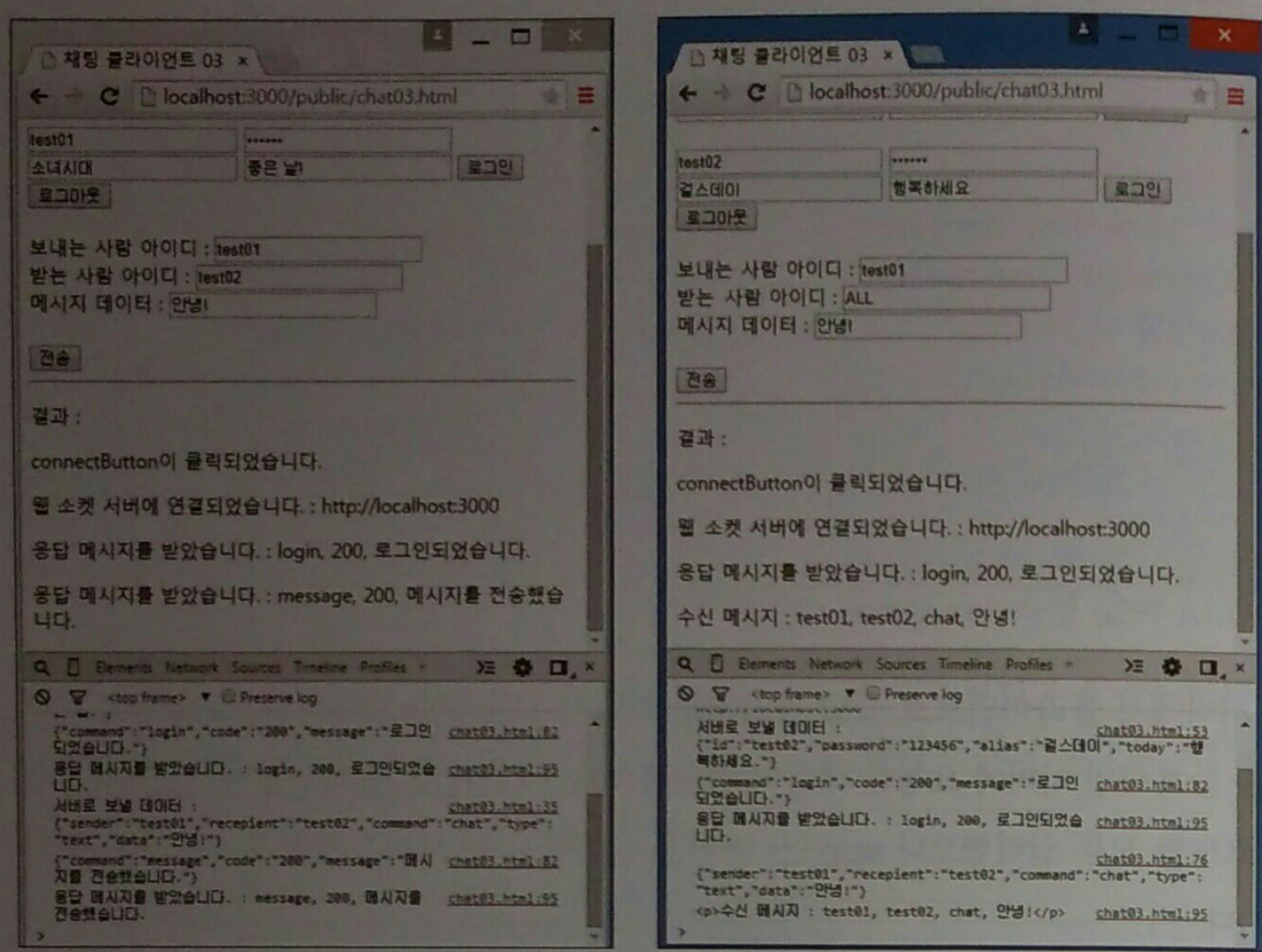
response 이벤트는 서버에서 클라이언트로 응답 메시지를 보내기 위해 만든 것입니다. 그래서 이 이벤트를 처리하는 함수를 등록하면 서버에 요청한 내용이 정상적으로 처리되었는지 확인할 수 있습니다. 일대일 채팅에 필요한 코드를 모두 입력했으니 app3.js 파일을 실행하고 웹 브라우저를 두 개 띄운 후 다음 사이트에 접속합니다.

▶ http://localhost:3000/public/chat03.html



◀ 일대일 채팅 웹 페이지를 열었을 때의 화면

한쪽 웹 브라우저에서는 [연결하기] 버튼을 누른 후 test01 아이디로 로그인하고, 다른 쪽 웹 브라우저에서는 [연결하기] 버튼을 누른 후 test02 아이디로 로그인합니다. 양쪽 모두 로그인되었으면 첫 번째 웹 브라우저의 '받는 사람 아이디' 입력란에 test02를 입력한 후 [전송] 버튼을 누릅니다. 그러면 다른 쪽 웹 브라우저에서는 전달받은 메시지가 표시됩니다.



▲ 두 개의 웹 브라우저 사이에서 일대일 채팅을 하는 화면

[로그아웃] 버튼을 눌렀을 때 logout 이벤트를 서버로 보내고, 서버에서 login_ids에 들어 있는 해당 아이디의 정보를 삭제하는 과정은 여러분이 직접 추가해보기 바랍니다.

정박사의 한마디 배열의 요소는 여러 가지 방법으로 삭제할 수 있어요

배열에서 특정 요소를 삭제할 때는 delete 키워드나 배열의 splice( ) 메소드 등 여러 가지 방법을 사용할 수 있습니다.

이렇게 해서 일대일 채팅 기능을 만들어 보았습니다. socket.io 모듈이 접속한 클라이언트를 socket 객체의 id 속성으로 구별할 수 있는 방법을 제공하기 때문에 아주 간단하게 일대일 채팅 기능을 구현할 수 있다는 것을 알 수 있습니다.

10-3 | 그룹 채팅하기

일대일 채팅은 한 사람이 다른 사람을 지정하여 메시지를 보내는 방식입니다. 그러나 그룹 채팅(Group Chatting)은 방을 만들고 그 방에 초대된 사람끼리 동시에 메시지를 주고받습니다. 보통 카카오톡이나

라인과 같은 채팅 앱에서는 일대일 채팅, 그룹 채팅, 파일 전송 등이 가장 기본적인 기능입니다. 여기에서는 어떻게 그룹 채팅 기능을 만들 수 있는지 살펴봅니다.

방 만들기

방을 만들고 방에 들어온 몇몇 사용자가 그룹 채팅을 하는 기능은 메신저에서 아주 일반적입니다. socket.io 모듈은 이렇게 몇 명의 사용자를 하나의 방에 모아 두고 방에 들어온 특정 클라이언트에만 메시지를 전송할 수 있는 기능도 제공합니다. 이 때문에 그룹 채팅 기능도 코드 몇 줄로 간단하게 만들 수 있습니다.

일대일 채팅과 그룹 채팅의 가장 큰 차이는 방을 만든다는 데 있습니다. 그러므로 먼저 방을 만들거나 없애는 기능을 추가해 보겠습니다. [public] 폴더의 chat03.html 파일을 복사하여 chat04.html 파일을 만들고 로그인을 만드는 태그 아랫부분에 다음과 같이 방을 만드는 태그를 추가합니다.

참조 파일 | ChatExample>/public/chat04.html

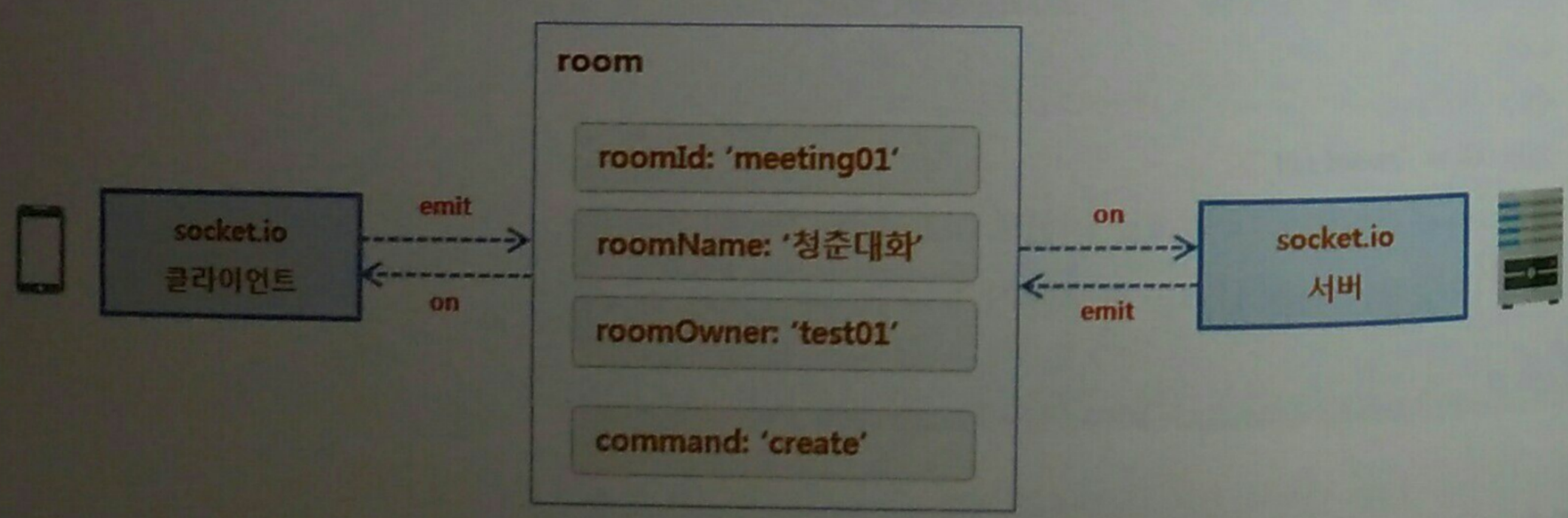
```
.....

<div>
  <input type = "text" id = "roomIdInput" value = "meeting01">
  <input type = "text" id = "roomNameInput" value = "청춘들의 대화">

  <input type = "button" id = "createRoomButton" value = "방 만들기">
  <input type = "button" id = "updateRoomButton" value = "방 이름 바꾸기">
  <input type = "button" id = "deleteRoomButton" value = "방 없애기">
</div>
<br>
<div id = "roomList">

</div>

.....
```

방을 구별할 수 있도록 방의 아이디와 이름을 입력할 입력 상자를 두 개 만들고 버튼은 세 개를 추가합니다. 버튼은 각각 [방 만들기], [방 이름 바꾸기], [방 없애기]라는 이름으로 만들었고, 그 아래에는 서버에서 방 리스트를 받아 오면 화면에 보여 줄 태그가 추가되어 있습니다.

입력 상자와 버튼을 만드는 데 필요한 태그를 추가했으니 파일의 위쪽에 있는 <script> 태그 부분을 찾아 [방 만들기] 버튼을 클릭했을 때 처리할 코드를 추가합니다.

```
……

// 방 만들기 버튼을 클릭하면 처리
$("#createRoomButton").bind('click', function(event) {
  var roomId = $('#roomIdInput').val();
  var roomName = $('#roomNameInput').val();
  var id = $('#idInput').val();

  var output = {command : 'create', roomId : roomId, roomName : roomName, roomOwner : id};
  console.log('서버로 보낼 데이터 : ' + JSON.stringify(output));

  if(socket == undefined) {
    alert('서버에 연결되어 있지 않습니다. 먼저 서버에 연결하세요.');
    return;
  }

  socket.emit('room', output);
});

……
```

[방 만들기] 버튼을 누르면 방의 아이디와 이름을 입력 상자에서 가져오고 로그인한 사람의 아이디도 가져옵니다. 이렇게 가져온 정보는 자바스크립트 객체로 만든 후 room 이벤트의 형태로 서버에 보내집니다. room 이벤트로 보내질 객체에 들어 있는 속성들은 다음과 같습니다.



▲ 방(Room)을 위해 서버에 전송하는 객체의 속성들

방의 정보는 roomId, roomName, roomOwner 속성으로 만들어 객체에 넣은 후 서버로 보냅니다. roomName은 어떤 방인지를 표시할 때 사용하며 roomOwner는 방을 만든 방장을 의미합니다. command 속성은 방을 어떻게 할 것인지를 결정하는 데 사용합니다. command 속성 값이 create일

때는 방을 만들고, update일 때는 방의 정보를 수정하며, delete일 때는 방을 삭제하도록 정의하였습니다. 방을 만들 때 필요한 코드를 입력했으니 방에 대한 정보를 수정하거나 방을 삭제할 수 있도록 다음 코드도 추가합니다.

참조 파일 | ChatExample>/public/chat04.html

```
……

  // 방 이름 바꾸기 버튼을 클릭하면 처리
  $("#updateRoomButton").bind('click', function(event) {
    var roomId = $('#roomIdInput').val();
    var roomName = $('#roomNameInput').val();
    var id = $('#idInput').val();

    var output = {command : 'update', roomId : roomId,
                  roomName : roomName, roomOwner : id};
    console.log('서버로 보낼 데이터 : ' + JSON.stringify(output));

    if(socket == undefined) {
      alert('서버에 연결되어 있지 않습니다. 먼저 서버에 연결하세요.');
      return;
    }

    socket.emit('room', output);
  });

  // 방 없애기 버튼을 클릭하면 처리
  $("#deleteRoomButton").bind('click', function(event) {
    var roomId = $('#roomIdInput').val();

    var output = {command : 'delete', roomId : roomId};
    console.log('서버로 보낼 데이터 : ' + JSON.stringify(output));

    if(socket == undefined) {
      alert('서버에 연결되어 있지 않습니다. 먼저 서버에 연결하세요.');
      return;
    }

    socket.emit('room', output);
  });

……
```

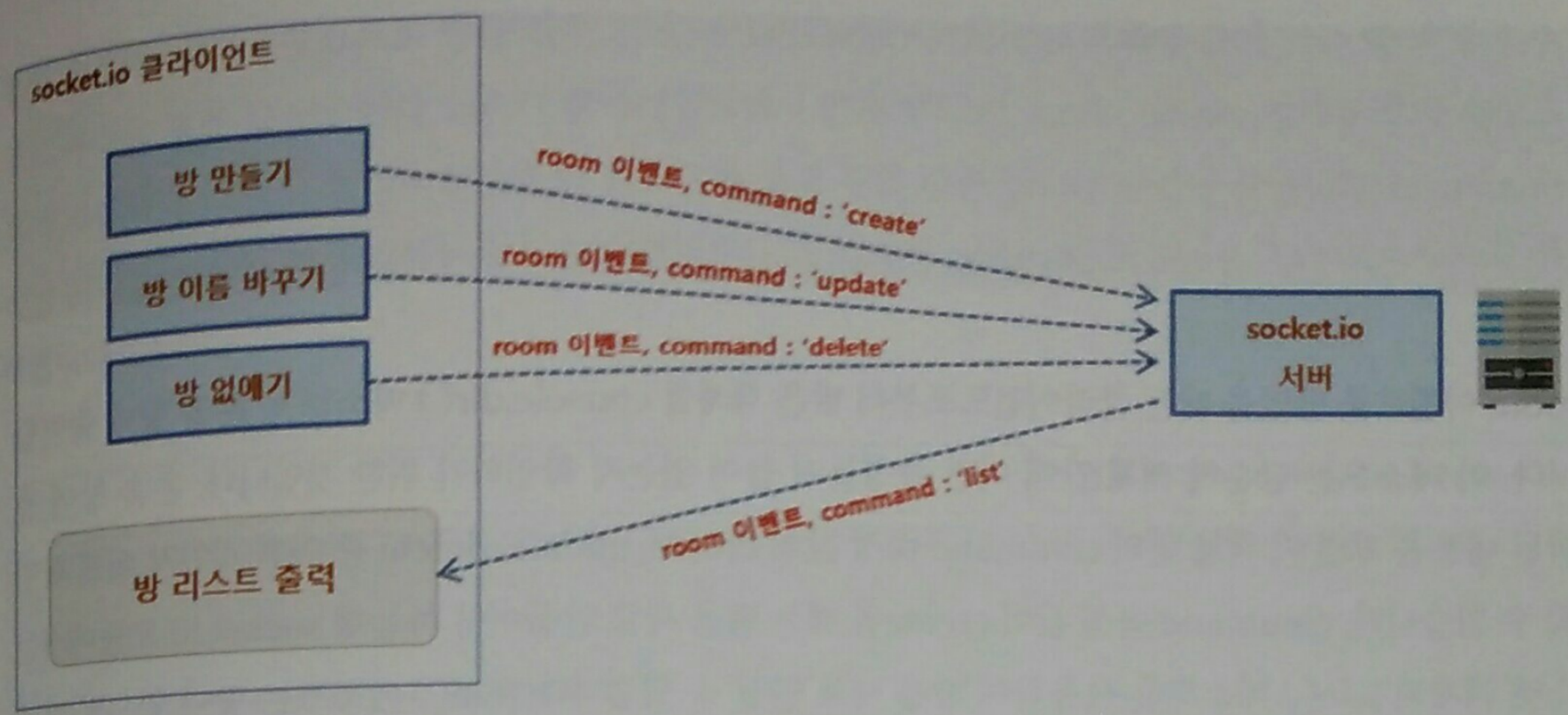
[방 이름 바꾸기] 버튼의 id는 updateRoomButton이므로 이 버튼을 클릭했을 때 발생하는 click 이벤트를 처리할 함수를 추가합니다. 방 이름을 업데이트하기 위해 전송하는 socket 객체의 이벤트 이름은 room으로 설정하고, 서버로 전송할 객체에 들어갈 command 속성 값은 update로 합니다. [방 없애기] 버튼의 경우에는 서버로 전송할 객체에 들어갈 command 속성 값을 delete로 설정합니다.

서버 쪽에서는 방을 만들거나 삭제하는 room 이벤트를 받으면 그에 따른 적절한 처리를 한 다음 응답 메시지와 함께 방 리스트를 클라이언트로 보내 줄 것입니다. 방 리스트의 command 속성 값은 list로 합니다. 따라서 room 이벤트를 받아서 처리할 함수를 on( ) 메소드를 호출하여 등록하고, 그 함수 안에서 전달받은 객체의 command 속성이 list인지 확인하여 처리합니다.

참조 파일 | ChatExample>/public/chat04.html

```
……

  // 그룹 채팅에서 방과 관련된 이벤트 처리
  socket.on('room', function(data) {
    console.log(JSON.stringify(data));

    println('<p>방 이벤트 : ' + data.command + '</p>');
    println('<p>방 리스트를 받았습니다.</p>');
    if(data.command == 'list') {// 방 리스트
      var roomCount = data.rooms.length;
      $("#roomList").html('<p>방 리스트 ' + roomCount + '개</p>');
      for (var i = 0; i < roomCount; i++) {
        $("#roomList").append('<p>방 #' + i + ' : ' + data.rooms[i].id + ', '
                    + data.rooms[i].name + ', ' + data.rooms[i].owner + '</p>');
      }
    }
  });

……
```

서버에서 보내오는 이벤트는 클라이언트에서 보낼 때처럼 room 이벤트이지만 전달되는 객체에 들어 있는 command 속성 값은 list인지만 확인하면 됩니다. 즉, 클라이언트에서는 만든 방의 리스트만 전달받습니다. command 속성 값이 list라면 전달받은 객체 안에 rooms 속성이 들어 있고 그 속성 값은 방의 배열이 됩니다. 방 리스트를 객체에서 확인할 수 있으므로 방 리스트를 화면에 출력합니다. 방 리스트는 웹 문서에 id가 roomList인 태그에 보여주어야 하므로 그 태그 안에 <p> 태그를 차례대로 추가해서 보여 줍니다.



▲방(Room)과 관련된 이벤트 처리

이제 방에 대한 요청을 처리하는 서버 코드를 만들어 보겠습니다.

app3.js 파일을 복사하여 app4.js 파일을 만든 후 클라이언트로부터 room 이벤트를 받았을 때 처리하는 코드를 추가합니다. 먼저 전달받은 객체 안에 있는 command 속성 값이 create일 때 처리하는 코드를 추가합니다.

참조 파일 | ChatExample>app4.js

```
……

// room 이벤트를 받았을 때의 처리
socket.on('room', function(room) {
  console.log('room 이벤트를 받았습니다.');
  console.dir(room);

  if(room.command == 'create') {
    if(io.sockets.adapter.rooms[room.roomId]) { // 방이 이미 만들어져 있는 경우
      console.log('방이 이미 만들어져 있습니다.');

    } else {
      console.log('방을 새로 만듭니다.');

      socket.join(room.roomId);

      var curRoom = io.sockets.adapter.rooms[room.roomId];
      curRoom.id = room.roomId;
      curRoom.name = room.roomName;
      curRoom.owner = room.roomOwner;
```

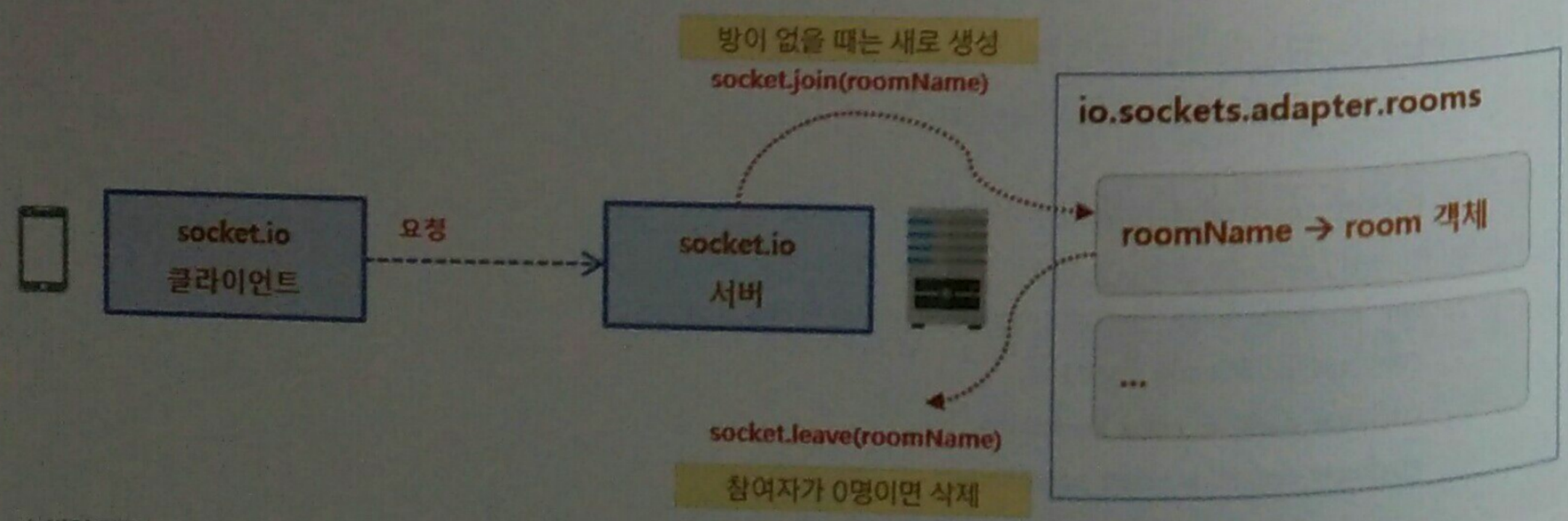
```
    }
}

......
```

room 이벤트를 받았을 때는 클라이언트로부터 받은 객체를 console.dir( ) 메소드로 콘솔 창에 출력합니다. 이 메소드는 단순히 객체 안에 어떤 속성들이 들어 있는지 확인하기 위한 것입니다. 콘솔 창에 출력된 내용을 보면 이 객체 안의 command 속성 값이 create, update, delete 중 어떤 것인지 쉽게 확인할 수 있습니다. command 속성 값이 create일 때는 방을 새로 만들어야 하는데 socket.io 모듈에서는 소켓 객체의 join( ) 메소드를 사용하여 방을 새로 만들 수 있습니다. join( ) 메소드는 방이 있으면 방에 입장하고 방이 없으면 방을 만든 후 방에 참여합니다. 이렇게 방에 입장하거나 방에서 나갈 때 사용하는 소켓 객체의 메소드들은 다음과 같습니다.

메소드 이름	설명
join(roomName)	방에 입장합니다. 방이 없으면 방을 새로 만듭니다.
leave(roomName)	방에서 나옵니다.

[표] 방의 출입을 처리하는 소켓 객체의 메소드

그런데 단순히 방 이름만 가지고 방에 입장했는지 아니면 방에서 나왔는지를 처리한다면 클라이언트로부터 전달받은 방장 정보는 어디에 넣어 둘 수 있을까요? 그리고 만약 방을 만든 일시나 다른 속성을 더 추가하여 관리하고 싶다면 이런 속성들을 넣어 둘 수 있는 무언가가 필요합니다. 방 정보를 관리하고 싶다면 단순히 변수를 새로 만든 후 배열 형태로 방 객체를 넣어 둘 수 있습니다. 그러나 이미 socket.io 모듈에서 방 정보를 관리하고 있으므로 socket.io 모듈에서 제공하는 방 객체에 속성을 추가하는 것도 가능합니다. socket.io 모듈에서 방 정보는 io.sockets.adapter.rooms에 들어 있습니다. 이 객체에는 방 ID를 속성 이름으로 지정하여 방을 추가하거나 삭제할 수 있습니다.



▲ 방이 만들어지거나 삭제되는 과정

방이 없으면 join( ) 메소드로 방을 만들고 io.sockets.adapter.rooms 객체에 들어 있는 방 객체에 추가로 속성을 넣을 수 있습니다. 그러면 클라이언트에서 받은 방의 id, name, owner 정보를 방의 속성으로 추가합니다. 그다음 클라이언트에 방 리스트를 보내 주면 방을 만드는 과정이 끝납니다. command 속성 값이 create일 때 처리하는 코드를 입력했으니 그 아래에 update와 delete를 처리하는 코드도 입력합니다.

참조 파일 | ChatExample>app4.js

```
……

} else if(room.command == 'update') {
  var curRoom = io.sockets.adapter.rooms[room.roomId];
  curRoom.id = room.roomId;
  curRoom.name = room.roomName;
  curRoom.owner = room.roomOwner;

} else if(room.command == 'delete') {
  socket.leave(room.roomId);

  if(io.sockets.adapter.rooms[room.roomId]) { // 방이 만들어져 있는 경우
    delete io.sockets.adapter.rooms[room.roomId];
  } else { // 방이 만들어져 있지 않은 경우
    console.log('방이 만들어져 있지 않습니다.');
  }
}

……
```

command 속성 값이 update인 경우에는 방 id를 사용해 io.sockets.adapter.rooms에 들어 있는 방 객체를 하나 찾아내고 속성을 변경한 후 클라이언트 쪽으로 변경된 방 리스트를 보냅니다. delete인 경우에는 소켓 객체의 leave( ) 메소드를 호출하여 방에서 나온 후 delete 키워드를 사용하여 io.sockets.adapter.rooms에서 방을 삭제합니다. 마지막 부분에서는 역시 마찬가지로 클라이언트에 변경된 방 리스트를 보냅니다. 어떤 경우든 마지막에 클라이언트로 방 리스트를 보내기 때문에 if~else문이 끝나는 곳에 다음 코드를 추가합니다.

```
……
  var roomList = getRoomList();

  var output = {command : 'list', rooms : roomList};
  console.log('클라이언트로 보낼 데이터 : ' + JSON.stringify(output));

  io.sockets.emit('room', output);
……
```

getRoomList( ) 메소드는 방 리스트를 배열로 만들어 줍니다. 그리고 클라이언트로 보낼 객체에는 command 속성과 rooms 속성을 넣어 줍니다. command 속성 값은 list로 설정하고 rooms 속성 값은 배열로 만든 방 리스트로 설정합니다. 클라이언트로 보낼 때는 emit( ) 메소드를 사용하며 이벤트의 이름은 room으로 지정합니다.

getRoomList( )는 기본 방을 제외하고 사용자가 만든 방 리스트를 배열로 만들어 줍니다. 따라서 기본 방은 socket.join( ) 메소드가 만든 것이 아니라 처음부터 만들어져 있던 방입니다. 예를 들어, getRoomList( ) 함수의 가장 첫 부분에 console.dir( ) 메소드로 io.sockets.adapter.rooms 객체의 속성을 출력해 보면 다음과 같습니다.

[콘솔 출력 결과]

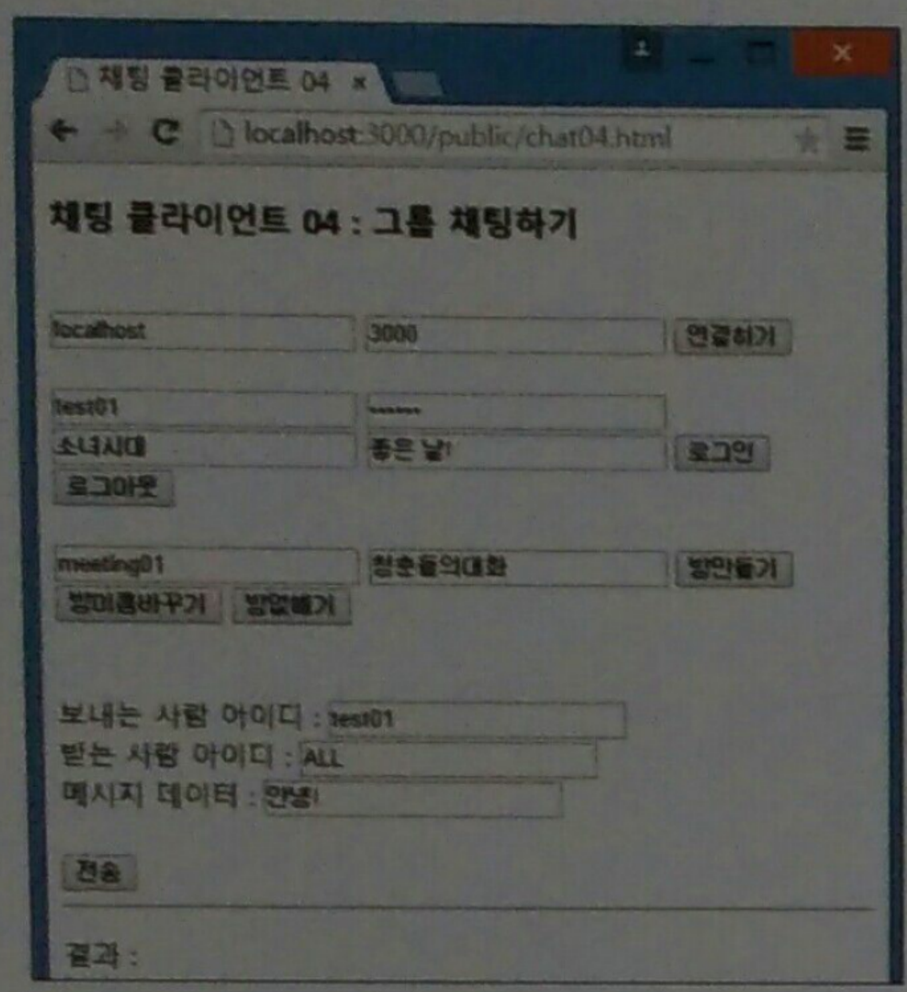
```
{nH38GKDoCGbUcXnAAAAA : {nH38GKDoCGbUcXnAAAAA : true},
 'ihq-Oi2Krn4lWdr9AAAB' : {'ihq-Oi2Krn4lWdr9AAAB' : true},
 meeting01 :                                  ► 새로 만든 meeting01 방의 속성 이름
  {nH38GKDoCGbUcXnAAAAA : true,
   id : 'meeting01',
   name : '청춘들의 대화',                      ► meeting01 방의 소켓 객체
   owner : 'test01'}}
```

새로 만든 meeting01 방의 속성 이름으로 meeting01이 추가되어 있습니다. 그런데 이 방 외에도 위쪽에 두 개의 방이 더 만들어져 있습니다. nH38GKDoCGbUcXnAAAAA처럼 쉽게 구별하기 힘든 기호로 된 속성 이름과 중괄호 { } 안에 있는 속성 역시 구별이 쉽지 않습니다. 이런 구조의 기본 방에는 모두 각각 소켓 객체가 들어 있습니다. 소켓 객체의 broadcast( ) 메소드를 호출하면 socket.io 모듈 내부에서 각 방에 들어 있는 다른 소켓 객체들을 참조하여 다른 클라이언트로 메시지를 전송합니다. 따라서 기본 방의 특성은 방의 이름과 그 안에 있는 속성의 이름이 같다는 것을 의미합니다. 결국 getRoomList( ) 함수에서는 방의 이름과 그 안에 들어 있는 속성의 이름이 같은 경우를 제외하고 사용자가 추가한 방들만 골라 리스트를 만들어 반환합니다.

```
……

function getRoomList() {
  console.dir(io.sockets.adapter.rooms);

  var roomList = [ ];

  Object.keys(io.sockets.adapter.rooms).forEach(function(roomId) { // 각각의 방에 대해 처리
    console.log('current room id : ' + roomId);
    var outRoom = io.sockets.adapter.rooms[roomId];

    // find default room using all attributes
    var foundDefault = false;
    var index = 0;
    Object.keys(outRoom).forEach(function(key) {
      console.log('#' + index + ' : ' + key + ', ' + outRoom[key]);

      if(roomId == key) { // default room
        foundDefault = true;
        console.log('this is default room.');
      }
      index++;
    });

    if(!foundDefault) {
      roomList.push(outRoom);
    }
  });

  console.log('[ROOM LIST]');
  console.dir(roomList);

  return roomList;
}

……
```

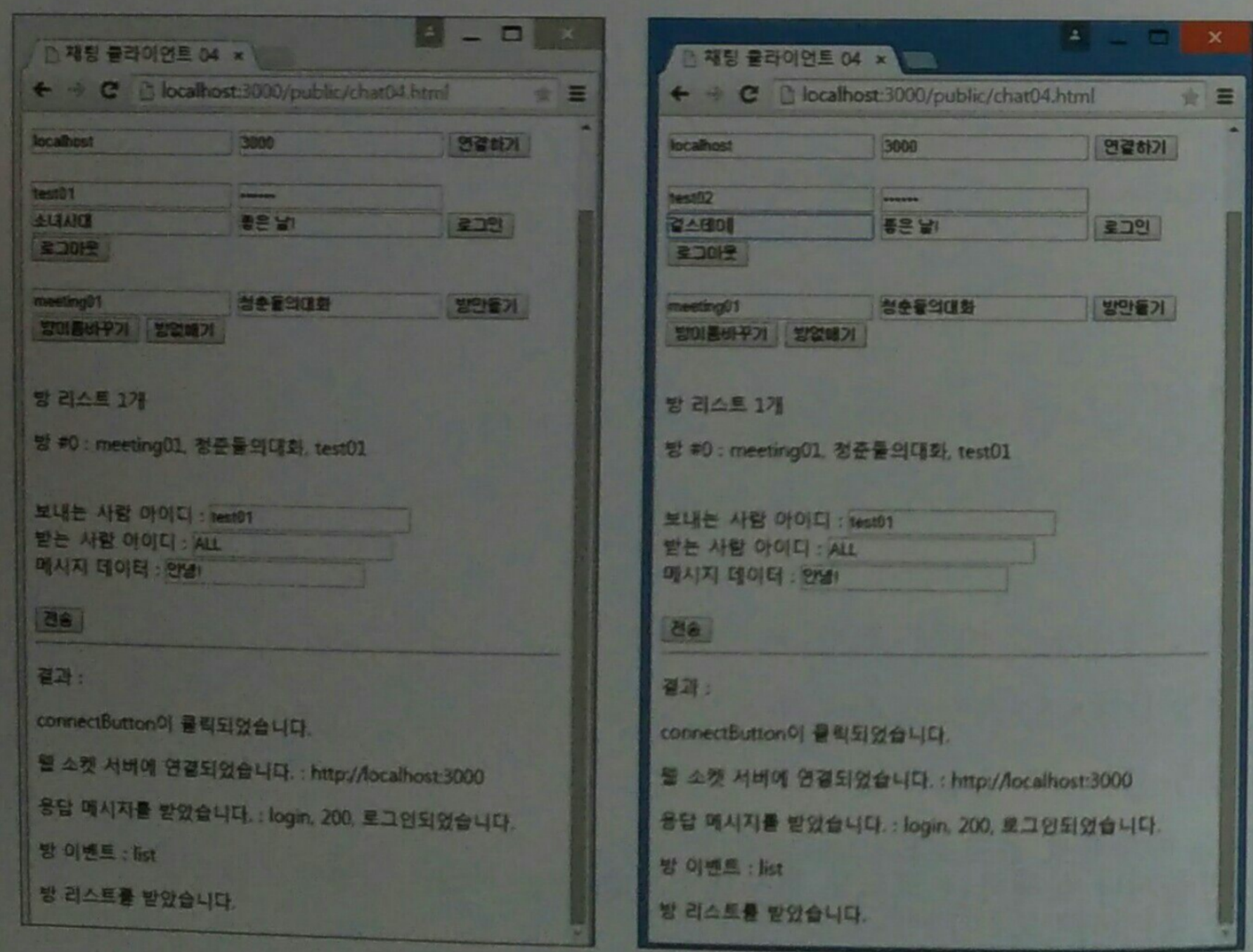
이제 서버에서 방을 만들거나 삭제하는 코드도 추가했으니 app4.js 파일을 실행하고 웹 브라우저에서 chat04.html 사이트에 접속합니다.

▶ http://localhost:3000/public/chat04.html



◀ 웹 브라우저에서 chat04.html 파일을 열었을 때의 화면

웹 브라우저 두 개를 띄운 후 양쪽 모두에 chat04.html 파일이 나타나게 합니다. 한쪽 브라우저에서 [연결하기] 버튼을 누른 후 test01로 로그인하고, 다른 쪽 브라우저에서는 [연결하기] 버튼을 누른 후 test02로 로그인합니다. 그다음 한쪽에서 [방 만들기] 버튼을 누르면 두 개의 웹 브라우저 모두에 새로 만든 방의 리스트가 전달됩니다. 방의 이름을 '내 친구'로 바꾼 후 [방 이름 바꾸기] 버튼을 클릭하면 수정된 방 리스트를 받아 화면에 표시합니다. 마지막으로 [방 없애기] 버튼을 누르면 지금까지 만든 방 중에서 입력한 ID의 방이 삭제됩니다.



▲ 방을 새로 만들었을 때 화면에 표시되는 내용

그룹 채팅을 위해 방을 만들거나 삭제하는 기능을 만들어 보았습니다. 실제로는 방장의 권한을 가지고

있는 사람이 바뀌었을 때 사용하는 코드가 더 필요하지만, 이러한 코드는 기존 코드를 조금만 수정하면 어렵지 않게 만들 수 있습니다.

그룹 채팅에서 메시지 보내기

방을 만들 수 있게 되었으니 이제 그룹 채팅 메시지를 보내는 기능을 추가해 보겠습니다. chat04.html 파일을 복사하여 chat05.html 파일을 만든 후 [방 입장하기]와 [방 나가기] 버튼을 태그로 추가합니다. 그 아래에 있는 메시지 전송 부분도 일대일 채팅과 그룹 채팅을 선택하여 보낼 수 있도록 〈select〉 태그를 추가하는 등 약간 수정합니다.

참조 파일 | ChatExample>/public/chat05.html

```
.....

  <div>
    <input type = "button" id = "joinRoomButton" value = "방 입장하기">
    <input type = "button" id = "leaveRoomButton" value = "방 나가기">
  </div>
  <br>
  <div>
    <div>
      <span>보내는 사람 아이디 : </span>
      <input type = "text" id = "senderInput" value = "test01">
    </div>
    <div>
      <span>받는 사람 아이디 : </span>
      <input type = "text" id = "recepientInput" value = "ALL">
    </div>

    <!— command 선택 <select> 채팅, 그룹 채팅 —>
    <select name = "chattype" id = "chattype">
      <option value = "chat">채팅</option>
      <option value = "groupchat" selected>그룹 채팅</option>
    </select>

    <div>
      <span>메시지 데이터 : </span>
      <input type = "text" id = "dataInput" value = "안녕!">
    </div>
    <br>
    <input type = "button" id = "sendButton" value = "전송">
  </div>

.....
```

[방 입장하기]와 [방 나가기] 버튼을 추가했고 메시지를 보내는 태그 사이에 〈select〉 태그를 추가했습니다. 〈select〉 태그는 일대일 채팅과 그룹 채팅 중 하나를 선택하기 위해 만든 것입니다. 여기에서 선택한 정보는 message 이벤트를 서버에 보낼 때 command 속성 값을 chat 또는 groupchat으로 지정하는 데 사용됩니다. 〈body〉 태그 안에 필요한 태그를 추가했으니 이제 자바스크립트 코드를 추가합니다. [방 입장하기]와 [방 나가기] 버튼을 클릭했을 때 실행될 함수를 다음과 같이 추가합니다.

참조 파일 | ChatExample>/public/chat05.html

```
……

  // 방 입장하기 버튼을 클릭하면 처리
  $("#joinRoomButton").bind('click', function(event) {
    var roomId = $('#roomIdInput').val();

    var output = {command : 'join', roomId : roomId};
    console.log('서버로 보낼 데이터 : ' + JSON.stringify(output));

    if(socket == undefined) {
      alert('서버에 연결되어 있지 않습니다. 먼저 서버에 연결하세요.');
      return;
    }

    socket.emit('room', output);
  });

  // 방 나가기 버튼을 클릭하면 처리
  $("#leaveRoomButton").bind('click', function(event) {
    var roomId = $('#roomIdInput').val();

    var output = {command : 'leave', roomId : roomId};
    console.log('서버로 보낼 데이터 : ' + JSON.stringify(output));

    if(socket == undefined) {
      alert('서버에 연결되어 있지 않습니다. 먼저 서버에 연결하세요.');
      return;
    }

    socket.emit('room', output);
  });

……
```

[방 입장하기] 버튼을 누르면 서버에 room 이벤트를 보냅니다. 방을 만들거나 삭제하기 위해 추가한 입력 상자에서 방 ID를 가져옵니다. [방 입장하기] 버튼을 눌렀을 때 보내는 객체의 command 속성 값은 join으로 설정하고, [방 나가기] 버튼을 눌렀을 때 보내는 객체의 command 속성 값은 leave로 설정합니다. [전송] 버튼을 클릭했을 때 처리하는 함수도 일대일 채팅인지 그룹 채팅인지 구별하는 〈select〉 태그에서 값을 가져오도록 수정합니다.

참조 파일 | ChatExample>/public/chat05.html

```
……

// 전송 버튼을 클릭하면 처리
$("#sendButto").bind('click', function(event) {

  // chattype 구별
  var chattype = $('#chattype option : selected').val();

  var sender = $('#senderInput').val();
  var recepient = $('#recepientInput').val();
  var data = $('#dataInput').val();

  var output = {sender : sender, recepient : recepient, command : chattype,
                type : 'text', data : data};
  console.log('서버로 보낼 데이터 : ' + JSON.stringify(output));

  if(socket == undefined) {
    alert('서버에 연결되어 있지 않습니다. 먼저 서버에 연결하세요.');
    return;
  }

  socket.emit('message', output);
});

……
```

〈select〉 태그의 id 속성이 chattype 값을 가지고 있으므로 이 〈select〉 태그의 〈option〉 태그 값 중에서 selected 속성이 들어 있는 것의 value 속성 값을 확인합니다. 이렇게 확인한 값은 message 이벤트와 함께 서버로 보내는 객체의 command 속성에 할당됩니다.

다음은 일대일 채팅과 그룹 채팅의 메시지 전송 방식을 보여 줍니다.

▲ 일대일 채팅과 그룹 채팅의 메시지 전송 방식

일대일 채팅에서는 io.sockets.connected 객체에 들어 있는 소켓 객체들 중에서 대상이 되는 소켓 객체를 소켓 ID로 찾은 후 emit( ) 메소드를 호출하여 메시지를 전송했습니다. 이와 다르게 그룹 채팅에서는 io.sockets.in( ) 메소드를 사용하여 대상이 되는 방에 들어 있는 소켓 객체들을 찾은 후 메시지를 전송합니다.

클라이언트 웹 페이지의 코드를 모두 변경했으니 app4.js 파일을 복사하여 app5.js 파일을 만듭니다. 그런 후 방에 대한 요청을 처리하는 코드와 방에 들어 있는 소켓 객체들에 메시지를 전송하는 코드를 추가합니다. 먼저 '방 입장하기'와 '방 나가기' 요청을 처리하는 코드를 추가합니다.

참조 파일 | ChatExample>app5.js

```
……

} else if(room.command == 'join') { // 방에 입장하기 요청

    socket.join(room.roomId);

    // 응답 메시지 전송
    sendResponse(socket, 'room', '200', '방에 입장했습니다.');
} else if(room.command == 'leave') { // 방 나가기 요청

    socket.leave(room.roomId);

    // 응답 메시지 전송
    sendResponse(socket, 'room', '200', '방에서 나갔습니다.');
}

……
```

방에 입장하거나 방에서 나가는 코드는 아주 간단합니다. socket.io 모듈이 이미 이 기능을 지원하므로

방에 입장할 때는 socket.join( ) 메소드를 호출하고, 방에서 나갈 때는 socket.leave( ) 메소드를 호출합니다.

메시지 전송 요청은 message 이벤트로 전달되는데 클라이언트에서 이 이벤트와 함께 전달되는 객체의 command 속성 값을 chat와 groupchat으로 구별하고 있습니다. 따라서 서버로 전달된 객체에 들어 있는 command 속성을 구별하여 처리할 수 있도록 코드를 수정합니다.

참조 파일 | ChatExample>app5.js

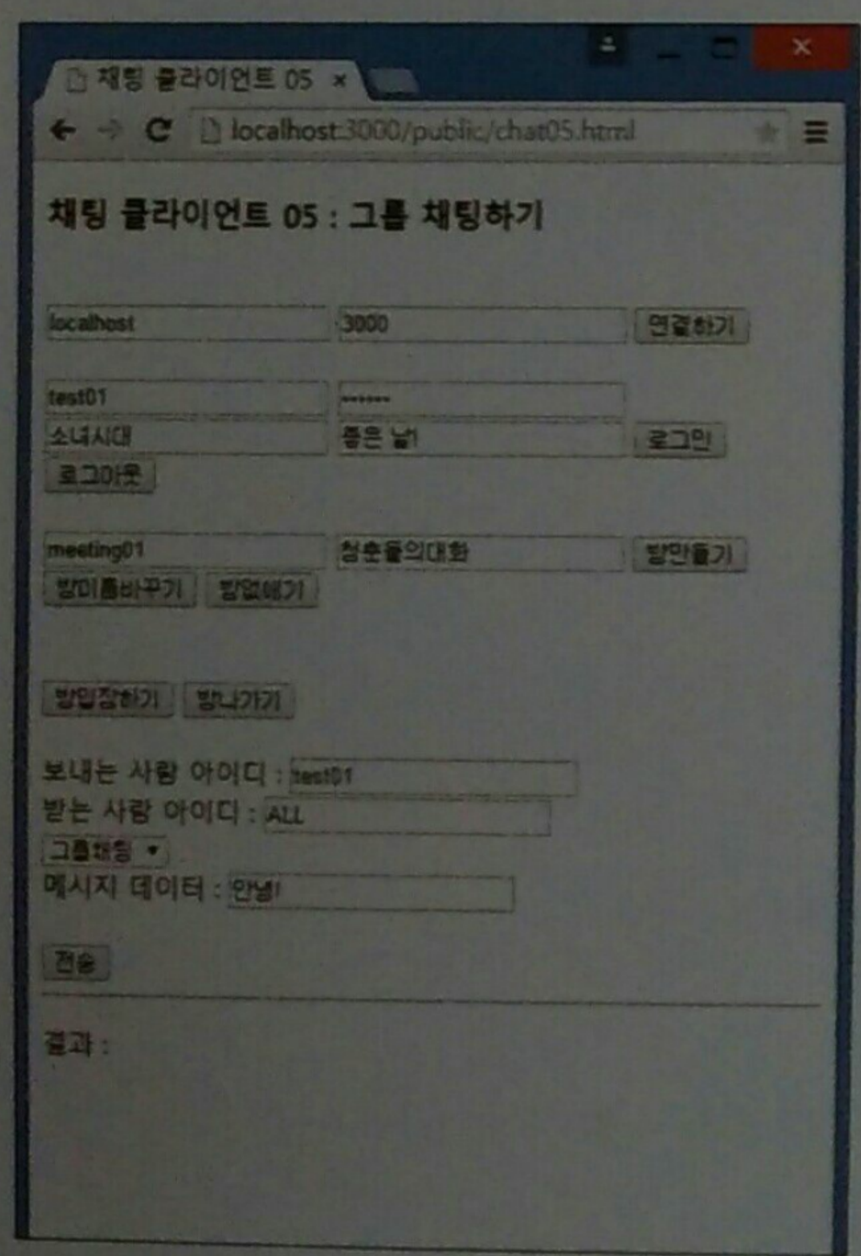
```
……

// 'message' 이벤트를 받았을 때의 처리
socket.on('message', function(message) {
  console.log('message 이벤트를 받았습니다.');
  console.dir(message);

  if(message.recepient ='ALL') {
    // 나를 포함한 모든 클라이언트에 메시지 전달
    console.dir('나를 포함한 모든 클라이언트에 message 이벤트를 전송합니다.')
    io.sockets.emit('message', message);
  } else {
    // command 속성으로 일대일 채팅과 그룹 채팅 구별
    if(message.command == 'chat') {
      // 일대일 채팅 대상에게 메시지 전달
      if(login_ids[message.recepient]) {
        io.sockets.connected[login_ids[message.recepient]].emit('message', message);

        // 응답 메시지 전송
        sendResponse(socket, 'message', '200', '메시지를 전송했습니다.');
      } else {
        // 응답 메시지 전송
        sendResponse(socket, 'login', '404', '상대방의 로그인 ID를 찾을 수 없습니다.');
      }
    } else if(message.command == 'groupchat') {
      // 방에 들어 있는 모든 사용자에게 메시지 전달
      io.sockets.in(message.recepient).emit('message', message);

      // 응답 메시지 전송
      sendResponse(socket, 'message', '200', '방 [' + message.recepient + ']의
                    모든 사용자들에게 메시지를 전송했습니다.');
    }
  }
});

……
```

recepient 속성 값이 ALL일 때는 모든 클라이언트들에 메시지를 전송하도록 그대로 두고 그렇지 않을 경우에는 command 속성 값이 chat인 경우와 groupchat인 경우를 구별하여 처리하도록 수정합니다. chat인 경우에 처리하는 코드는 이전과 같으며, groupchat인 경우에는 io.sockets.in( ) 메소드를 사용해 방에 들어 있는 모든 소켓 객체를 찾아낸 후 메시지를 전송합니다.

정박사의 한마디 io.sockets.in(...) 코드를 입력하니 오류가 뜨나요?

io.sockets.in(...) 코드를 입력하면 이클립스에서 오류로 표시되는 것을 볼 수 있습니다. 이것은 in을 sockets 객체의 메소드 이름이 아니라 키워드로 인식하기 때문입니다. 오류 표시가 보이더라도 파일을 실행하는 데는 문제가 없습니다.

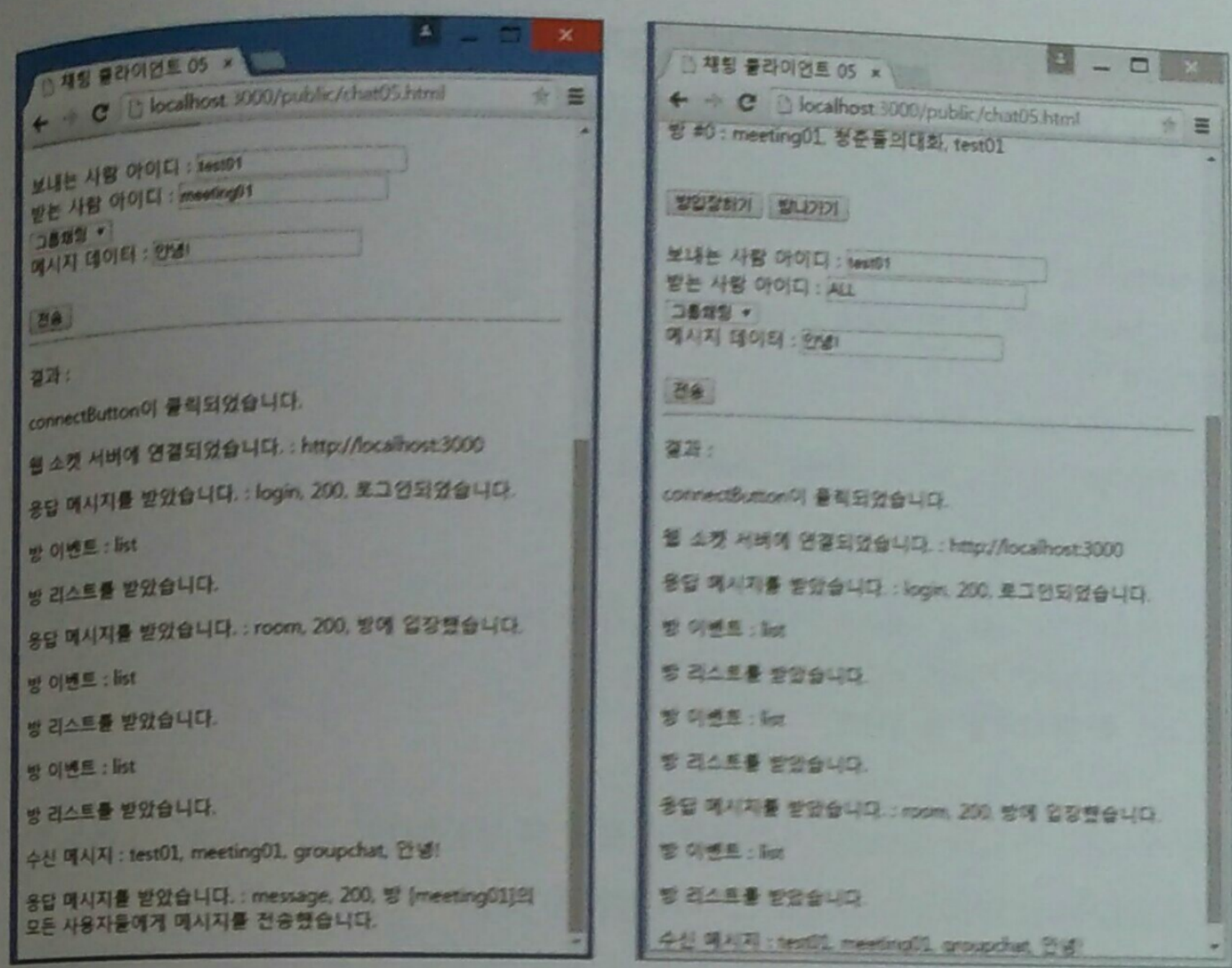
이제 방 안에 있는 사용자들에게 메시지를 전송할 수 있는 그룹 채팅 기능을 완성했습니다. app5.js 파일을 실행한 후 웹 브라우저를 두 개 실행하고 chat5.html 파일을 엽니다.



◀ 웹 브라우저에서 chat05.html 파일을 열었을 때 표시되는 내용

한쪽에서는 [연결하기] 버튼을 누른 후 [로그인] 버튼을 눌러 test01 아이디로 로그인합니다. 다른 쪽에서는 [연결하기] 버튼을 누른 후 test02 아이디로 로그인합니다. 한쪽에서 [방 만들기] 버튼을 눌러 방을 만들면 양쪽 모두 [방 입장하기] 버튼을 눌러 방으로 들어갑니다.

그다음 한쪽에서 '받는 사람 아이디' 입력 상자에 meeting01을 입력한 후 [전송] 버튼을 누르면 양쪽 모두에 메시지가 전송되는 것을 볼 수 있습니다.



▲ 방에 입장한 모든 클라이언트에 메시지를 전송했을 때 결과 화면

방에 입장하지 않은 클라이언트에는 메시지가 전송되지 않는다는 것을 확인하려면 웹 브라우저를 세 개 실행한 후 세 번째 화면에서는 방에 입장하지 않은 채 결과를 확인하면 됩니다.

10-4 | 채팅 웹 문서 예쁘게 꾸미기

지금까지는 일대일 채팅과 그룹 채팅 기능을 빠르게 만들기 위해 웹 문서에 들어가는 입력 상자나 버튼은 기본 태그만을 사용했습니다. 이 장에서는 Semantic UI를 사용해 웹 문서를 좀 더 예쁘게 꾸미는 작업을 간단하게 해 보겠습니다.

일대일 채팅을 하는 웹 문서가 chat03.html이므로 이 파일을 복사하여 client03.html 파일을 만든 후 다음처럼 태그를 수정합니다.

참조 파일 | ChatExample>/public/client03.html

```
.....

<div class = "container">
  <div id = "cardbox" class = "ui blue fluid card">
    <div class = "content">
      <div class = "left floated author">
        <img id = "iconImage" class = "ui avatar image" src = "./images/author.png">
      </div>
      <div id = "titleText" class = "header">일대일 채팅</div>
      <div id = "contentsText" class = "description">
        연결 및 로그인 후 메시지를 보내세요.
      </div>
    </div>
  </div>

.....
```

Semantic UI 라이브러리로 웹 문서를 꾸미는 방법은 앞에서 다루었기 때문에 자세한 설명은 하지 않겠습니다. 여기에서는 카드 모양의 UI를 만들고 제목으로 '일대일 채팅'을 써 주었습니다. 그 아래에는 [연결하기], [로그인], [로그아웃] 그리고 [전송하기] 버튼이 들어 있는 태그들을 차례로 입력합니다. 다음은 [연결하기] 버튼이 들어 있는 태그 부분입니다.

참조 파일 | ChatExample>/public/client03.html

```
.....

<div>
  <div class = "ui input">
    <input type = "text" id = "hostInput" value = "localhost">
  </div>
  <div class = "ui input">
    <input type = "text" id = "portInput" value = "3000">
  </div>
  <br><br>
  <input class = "ui primary button" type = "button" id = "connectButton"
         value = "연결하기">
</div>
<br>

.....
```

입력 상자와 버튼에 class 속성을 추가하고 ui input이나 ui primary button을 속성 값으로 넣어 준 것을 볼 수 있습니다. 가장 아래쪽에는 결과를 보여 주는 태그를 추가하고 id 값을 result로 넣어 줍니다.

참조 파일 | ChatExample>/public/client03.html

```
……,

  <h4 class="ui horizontal divider header">메시지</h4>
  <div class="ui segment" id="result">
  </div>

……,
```

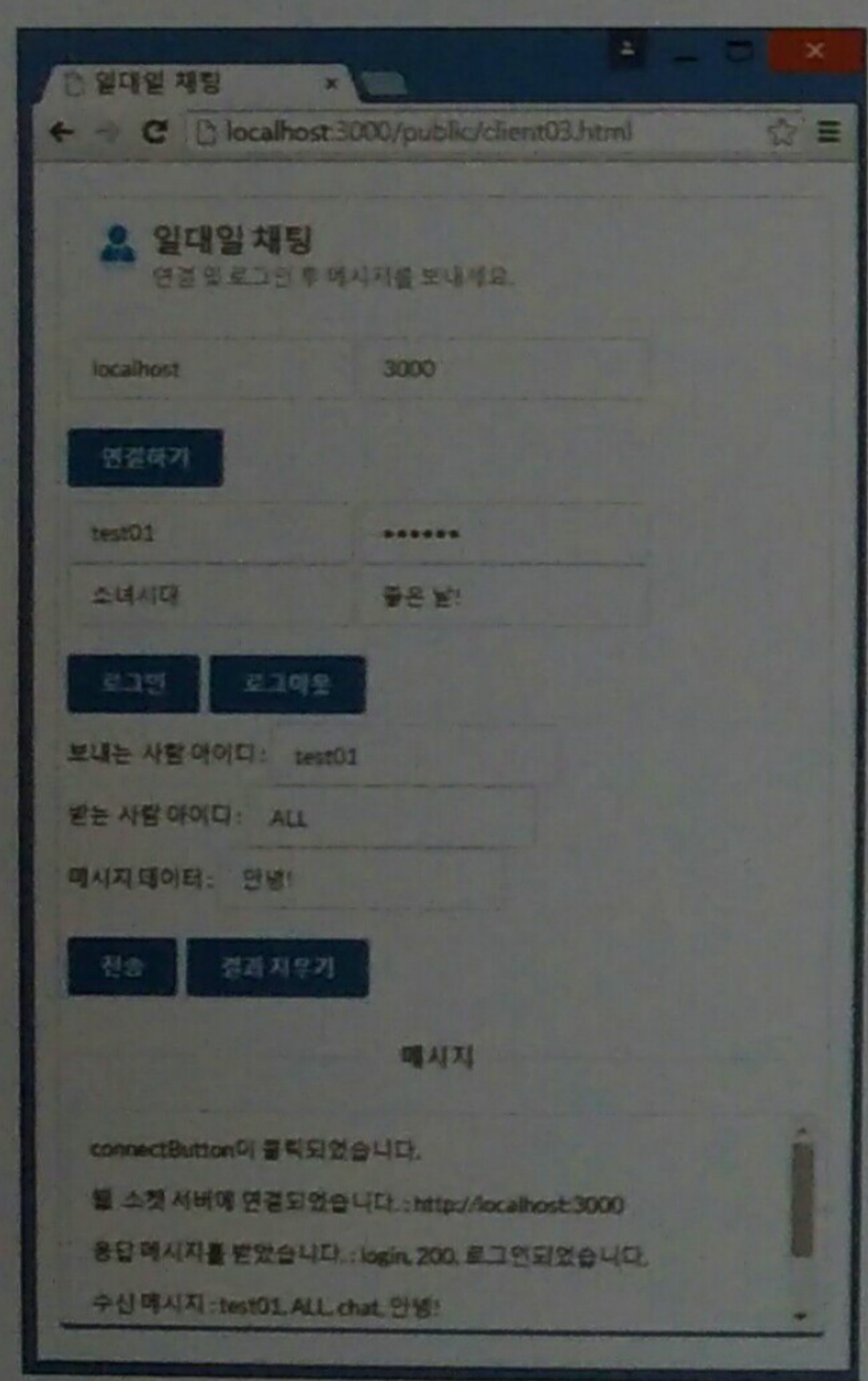
웹 문서의 〈head〉 태그 안에서는 **뷰 포트**를 설정하는 〈meta〉 태그와 CSS 속성을 지정하는 〈style〉 태그, 그리고 Semantic UI를 사용하기 위한 자바스크립트 파일과 CSS 파일을 링크하는 코드 등을 추가합니다.

참조 파일 | ChatExample>/public/client03.html

```
……,

  <meta name = "viewport"
        content = "width = device-width, height = device-height, initial-scale = 1">

……,

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

  <style>

……,

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

    #contentsText {
      color : #999;
    }
```

```
  #result {
    height : 10em;
    overflow : auto;
  }

</style>

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

……
```

app5.js 파일을 실행한 후 client4.html 파일을 조회하면 다음과 같이 UI가 바뀐 것을 볼 수 있습니다.



▲ 일대일 채팅 화면에 Semantic UI를 적용한 결과

태그와 CSS를 조금만 변경했는데도 좀 더 예쁜 모양의 화면이 만들어졌습니다. [연결하기], [로그인], [전송] 버튼을 차례대로 누르면서 그 아래에 추가되는 메시지를 확인해 보기 바랍니다.

그런데 채팅 화면은 서로가 주고받는 내용이 가장 중요하므로 가장 아래쪽에 있는 화면을 수정하여 채팅 메시지를 보여 주는 화면을 더 예쁘게 만들 수도 있습니다. client03.html 파일을 복사하여 client04.html 파일로 만든 후 결과를 표시하기 위해 추가한 태그 안에 다음 태그를 추가합니다.

참조 파일 | ChatExample>/public/client04.html

```
……,

  <ol class = "discussion">
    <li class = "other">
      <div class = "avatar">
        <img src = "/public/images/user2.png"/>
      </div>
      <div class = "messages">
        <p>어디쯤이야? 다들 기다리고 있어.</p>
        <time datetime = "2016-02-10 18 : 10">18시 10분</time>
      </div>
    </li>
    <li class = "self">
      <div class = "avatar">
        <img src = "/public/images/user1.png"/>
      </div>
      <div class = "messages">
        <p>차가 막히네. 조금 늦을 듯.</p>
        <time datetime = "2016-02-10 18 : 00">18시 00분</time>
      </div>
    </li>

……,

  </ol>

……,
```

<ol> 또는 <ul> 태그를 사용해서 리스트를 만드는 태그를 추가하고 그 안에 각각의 <li> 태그를 순서대로 추가합니다. 내가 직접 말하는 부분과 상대방이 말하는 부분이 화면의 오른쪽과 왼쪽에 보이도록 만들기 위해 CSS 코드를 입력합니다.

```
.....

  <style>

.....

    .discussion {
      list-style : none;
      background : #ededed;
      margin : 0;
      padding : 0 0 50px 0;
    }

    .discussion li {
      padding : 0.5em;
      overflow : hidden;
      display : flex;
    }

    .discussion .avatar {
      width : 40px;
      position : relative;
    }

    .discussion .avatar img {
      display : block;
      width : 100%;
    }



.....

    .self {
      justify-content : flex-end;
      align-items : flex-end;
    }

.....
```

나와 상대방이 말하는 내용을 오른쪽과 왼쪽에 표시하기 위해 flexbox를 사용하였습니다. 이 책에서는 CSS에 대한 자세한 설명은 생략하니 필요하다면 CSS나 하이브리드 앱에 대해 설명하는 다른 책들을 참조하기 바랍니다. 자바스크립트 코드를 넣어 두는 〈script〉 태그 안에는 addToDiscussion 함수를 추가합니다.

참조 파일 | ChatExample>/public/client04.html

```
……

function addToDiscussion(writer, msg) {
  println("addToDiscussion 호출됨 : " + writer + ", " + msg);

  var img = '/public/images/user1.png';
  if(writer == 'other') {
    img = '/public/images/user2.png';
  }

  var contents = "<li class='" + writer + "'>"
              + "  <div class='avatar'>"
              + "    <img src='" + img + "'/>"
              + "  </div>"
              + "  <div class='messages'>"
              + "    <p>" + msg + "</p>"
              + "    <time datetime='2016-02-10 18 : 30'>18시 30분</time>"
              + "  </div>"
              + "</li>";

  println("추가할 HTML : " + contents);
  $(".discussion").prepend(contents);
}

……
```

이 함수는 내가 다른 사람들에게 메시지를 보내거나 다른 사람에게서 메시지를 받을 때 class 속성 값이 discussion인 태그에 항목을 추가합니다. 여기에서는 메시지를 받은 시간을 임의로 고정 시간으로 해 두었습니다. 실제 웹 문서를 만들 때는 실제 시간을 표시할 수 있습니다. 이 함수를 호출하는 코드는 다음과 같이 사용합니다.

```
……

$("#sendButton").bind('click', function(event) {

……

  addToDiscussion('self', data);
});

……

socket.on('message', function(message) {

……

  addToDiscussion('other', message.data);
});

……
```

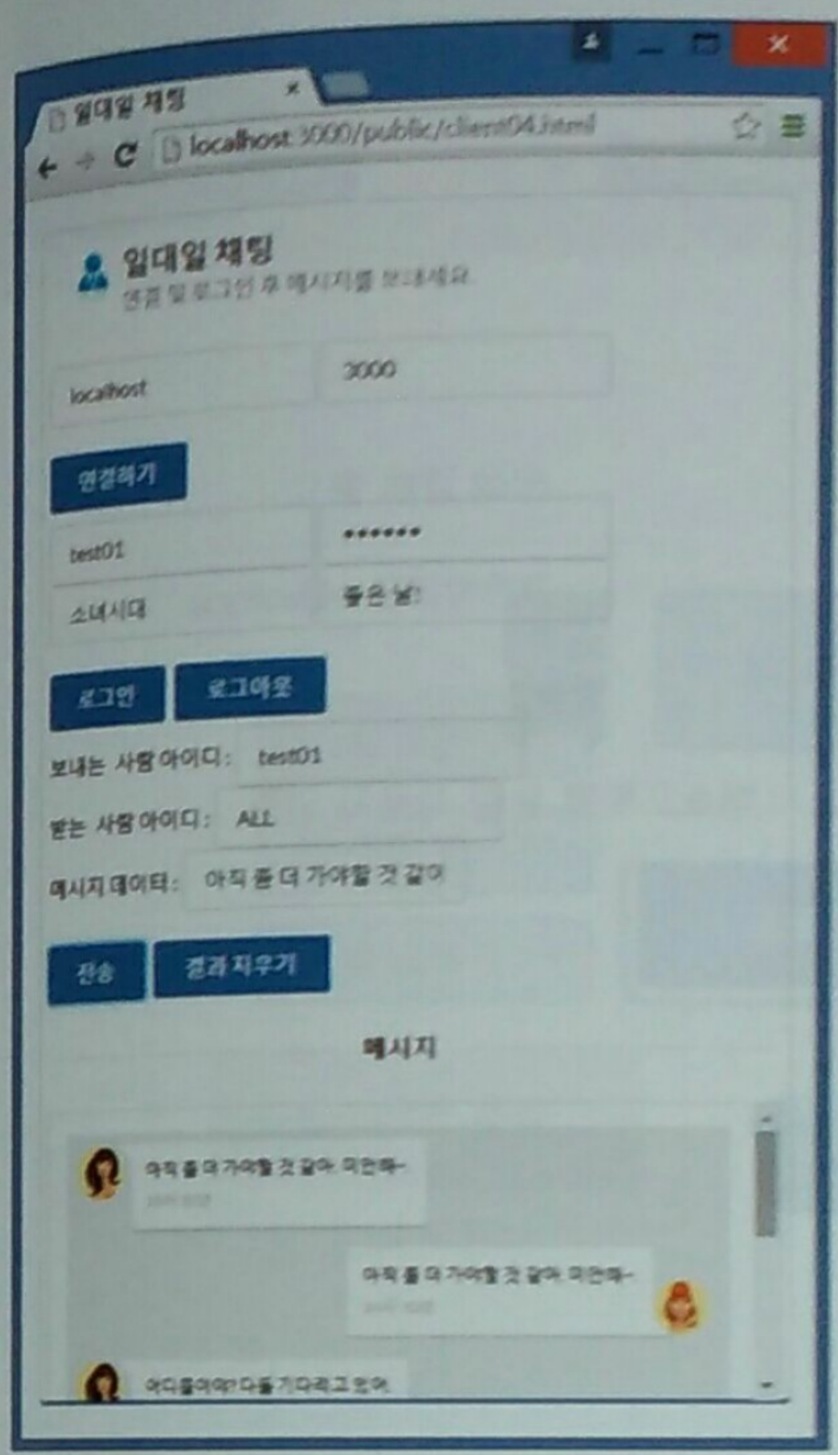
내가 직접 쓴 글이면 self, 상대방이 쓴 글이면 other를 파라미터로 넘겨줍니다.

정박사의 한마디 채팅 메시지 표시를 위해 만든 CSS에 대해 좀 더 자세한 설명이 필요한가요?

이 코드는 다음 사이트에서 설명하는 코드를 일부 수정한 것으로 좀 더 자세한 설명은 사이트를 직접 참조하기 바랍니다.

▶ https://css-tricks.com/replicating-google-hangouts-chat/

이제 client4.html 파일을 조회한 후 연결 및 로그인을 하고 메시지를 보내거나 받아 보면 다음과 같이 표시됩니다.



▲채팅 메시지를 보여 주는 화면을 더 예쁘게 꾸민 경우

이제 채팅 메시지가 더 예쁘게 만들어졌습니다. 필요하다면 아래쪽에 있는 채팅 메시지 표시 부분을 화면의 위쪽으로 올려 화면의 대부분을 차지하도록 만드는 등 일반적인 일대일 채팅 화면처럼 바꿀 수도 있습니다.

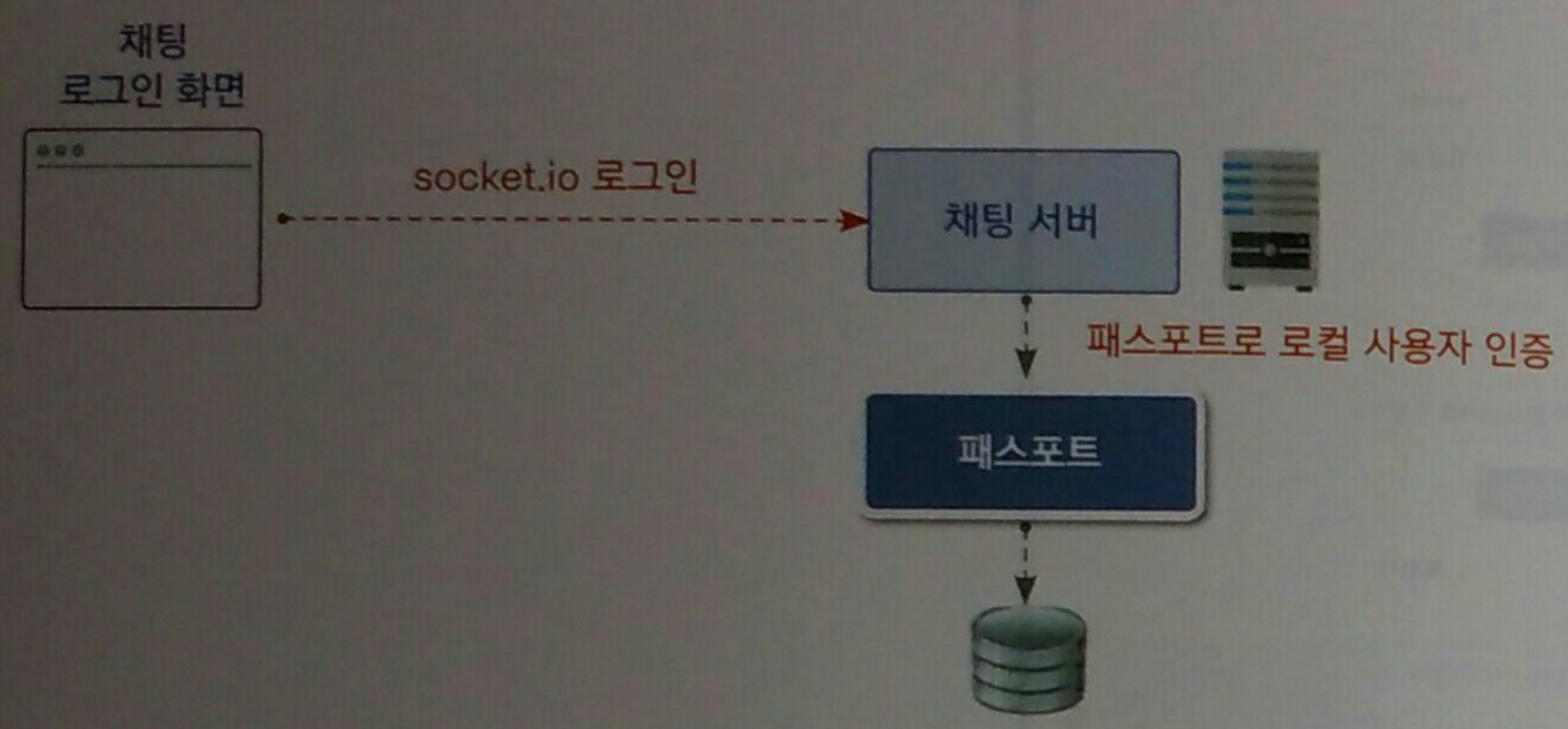
일대일 채팅 화면으로 전체 화면을 채우는 과정과 그룹 채팅 화면을 예쁘게 만드는 과정은 여러분이 직접 진행해 보기 바랍니다.

지금까지 socket.io를 사용해서 채팅 서버를 만들고 일대일 채팅과 그룹 채팅 기능을 추가하는 방법에 대해 알아보았습니다.

도전 문제

01. socket.io를 사용해 클라이언트가 로그인했을 때, 패스포트 모듈로 로컬 인증을 진행하는 기능을 만들어 보세요.

프로젝트 소스 Mission13 난이도 초급 ★☆☆☆



Hint!

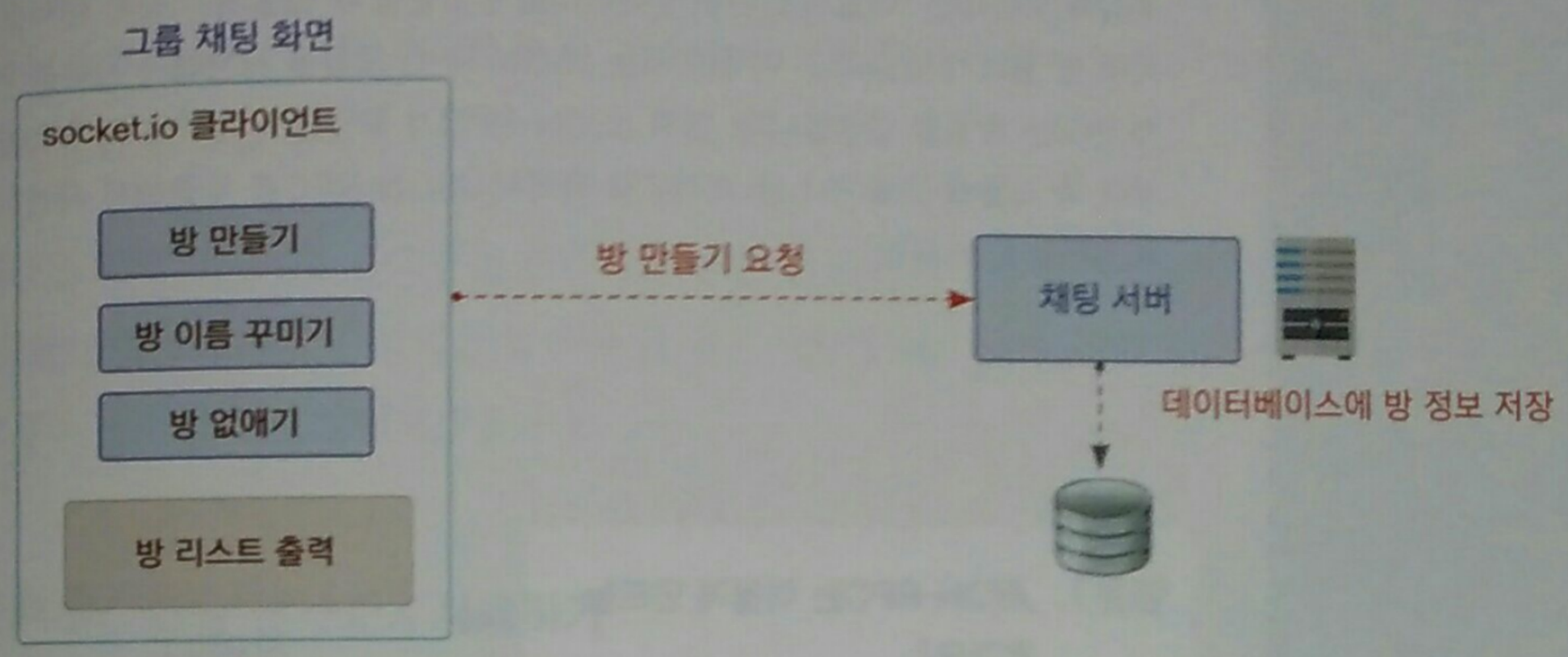
1. 로그인 화면의 구성은 이 장에서 만든 로그인 화면을 그대로 사용합니다.
2. 사용자가 로그인하면 socket.io로 로그인 이벤트를 보내는데 채팅 서버에서는 패스포트 모듈로 로컬 사용자 인증을 진행합니다.
3. 데이터베이스에 저장한 후 정상적으로 저장되었다는 응답 메시지를 클라이언트로 보냅니다.
4. 클라이언트에서 응답 메시지를 받으면 응답 화면에 메시지를 보여 줍니다.

MEMO

DO it!
도전 문제

02. 그룹 채팅 방을 만든 후 채팅 서버의 데이터베이스에 방 정보를 저장하고, 클라이언트의 그룹 채팅 화면에는 Semantic UI를 적용해 보세요.

프로젝트 소스 Mission14 난이도 중급 ★★☆☆



Hint!

1 그룹 채팅 화면은 이 장에서 만든 화면을 그대로 사용하되 Semantic UI를 사용해 꾸며 줍니다.

2 방 만들기를 요청하면 채팅 서버에서는 방을 만들고 데이터베이스에도 방 정보를 저장합니다. 이것은 채팅 서버가 비정상 종료되었다가 다시 시작되었을 경우에도 방 정보를 유지하기 위한 것입니다.

3 데이터베이스에 저장한 후 정상적으로 처리되었다는 응답 메시지를 클라이언트로 보냅니다.

4 클라이언트에서 응답 메시지를 받으면 응답 메시지를 보여 줍니다.

MEMO