

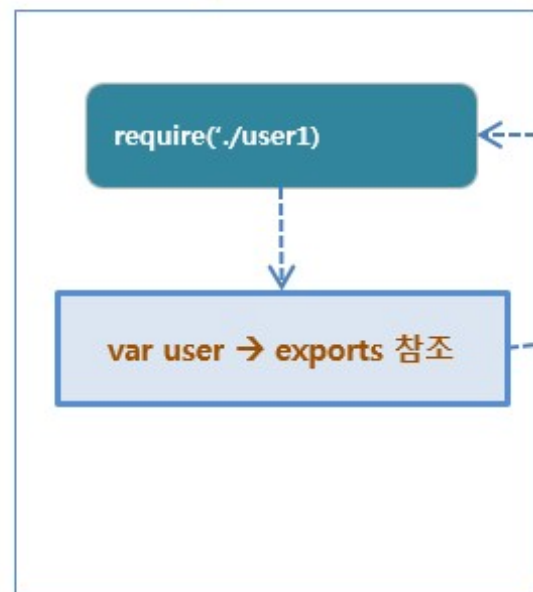
07 익스프레스 프로젝트를 모듈화 하기

07-01 모듈화방법 자세히 살펴보기

exports 전역 변수를 사용하여 모듈 구성하기

- exports 전역 변수에 속성으로 추가하는 방식
 - 다른 파일에서 exports 변수의 속성을 참조하여 사용

module_test1.js



사용자 정보 확인용 기능들
user1.js

모듈 파일 불러들이기



exports 전역변수 사용하기


4

- 모듈 파일
 - exports 전역변수에 속성으로 추가
- 메인 파일
 - 모듈 파일을 불러온 후 함수 실행

user1.js

```
// exports 객체 속성으로 함수 추가
exports.getUser = function() {
  return {id : 'test01', name : '소녀시대'};
}
// exports 객체 속성으로 객체 추가
exports.group = {id : 'group01', name : '친구'};
```

module_test1.js

```
// require( ) 메소드는 exports 객체를 반환함
var user1 = require('./user1'); path 지정 필수
function showUser() {
  return user1.getUser().name + ', ' + user1.group.name;
}
console.log('사용자 정보 : %s', showUser());
```

module.exports를 사용하여 모듈 구성하기

- exports 전역 변수에는 객체 설정 불가
- module.exports에 객체 설정

user3.js

// module.exports에는 객체를 그대로 할당할 수 있음

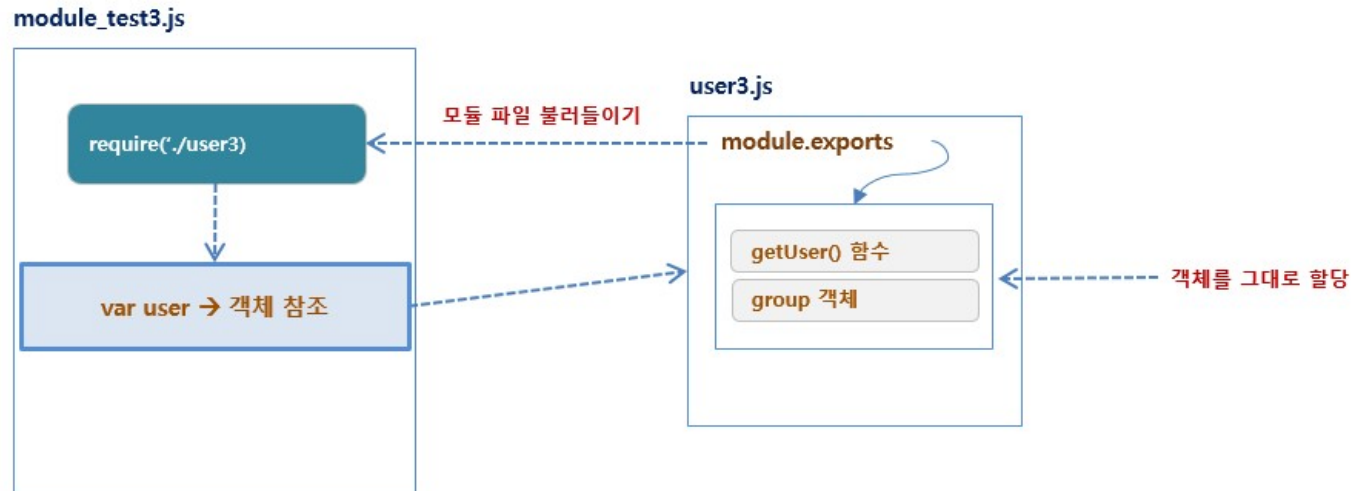
```
var user = {  
  getUser : function() {  
    return {id : 'test01', name : '소녀시대'};  
  },  
  group : {id : 'group01', name : '친구'}  
}
```

```
module.exports = user;
```

- 모듈 파일에서 설정한 객체를 참조하여 사용

module_test3.js

```
// require( ) 메소드는 객체를 반환함
var user = require('./user3');
function showUser() {
  return user.getUser().name + ' ' + user.group.name;
}
console.log('사용자 정보 : %s', showUser());
```



module.exports에 함수 할당하기

- 함수를 할당한 후 메인 파일에서 그 함수를 실행할 수 있음
 - 함수도 객체임

user4.js

```
// 인터페이스(함수 객체)를 그대로 할당할 수 있음
module.exports = function() {
  return {id : 'test01', name : '소녀시대'};
};
```

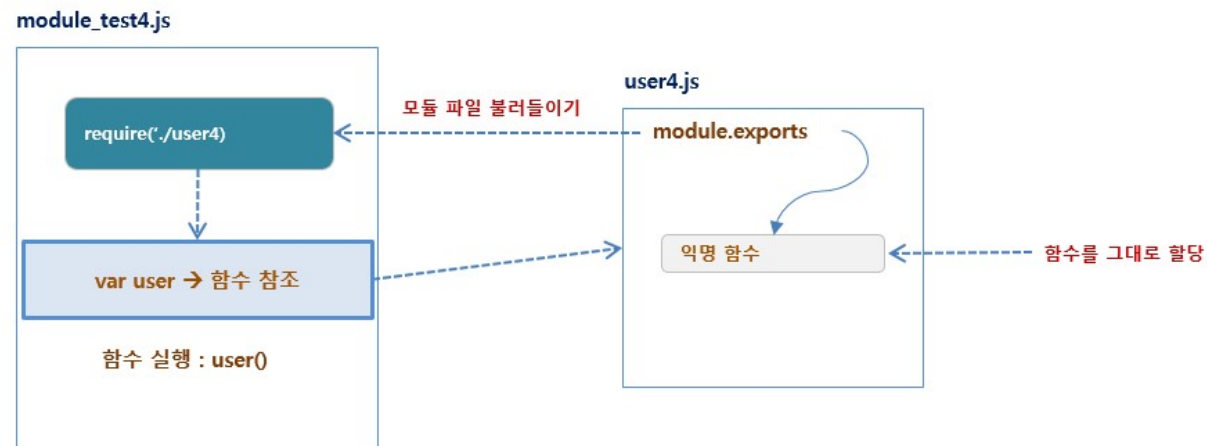
module_test4.js

```
// require( ) 메소드는 함수를 반환함
var user = require('./user4');
function showUser() {
  return user().name + ', ' + 'No Group';
}
console.log('사용자 정보 : %s', showUser());
```

module.exports에 함수를 할당하는 방식

8

```
module_test4.js
module_test4.js
Command: node "C:/Users/user/brackets-nodejs/ModuleExample/module_test4.js"
사용자 정보 : 소녀시대, No Group
Program exited with code 0
```



- exports와 module.exports의 동시 사용
 - module.exports가 우선시 되어 exports는 무시됨
 - module.exports 위주로 사용을 추천

require() 메소드의 동작 방식 이해하기

- 사용자 자체 제작 require() 메소드

module_test6.js

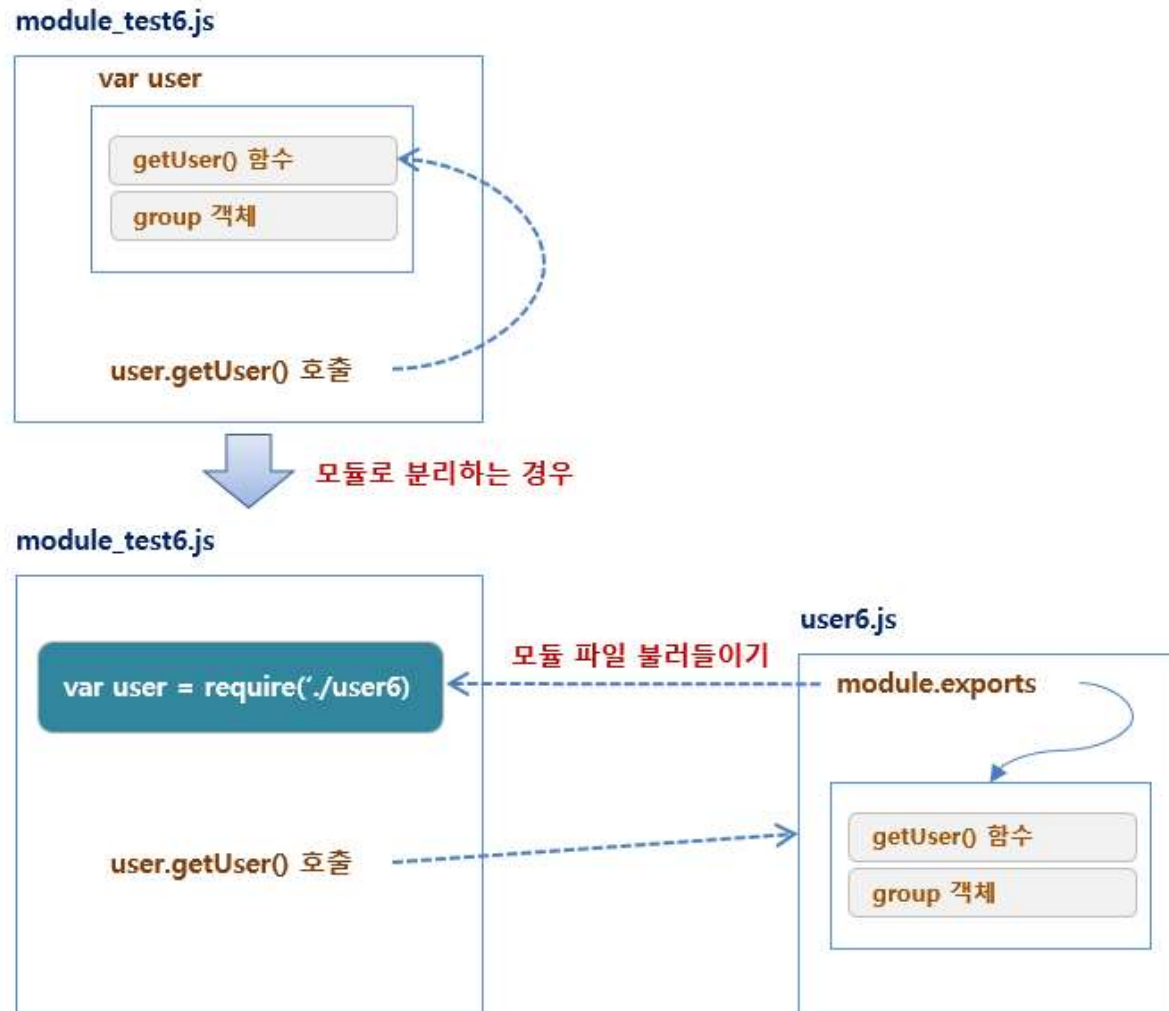
```
var require = function(path) {  
  var exports = {  
    getUser : function() {  
      return {id : 'test01', name : '소녀시대'};  
    },  
    group : {id : 'group01', name : '친구'}  
  }  
  return exports;  
}  
  
var user = require('...');  
function showUser() {  
  return user.getUser().name + ', ' + user.group.name;  
}  
console.log('사용자 정보 : %s', showUser());
```

system 제공 require 함수와
동일한 형태로 사용 가능

require() 메소드의 동작 방식 이해하기

10

- require() 메소드가 반환하는 것을 그대로 사용할 수 있음



- 전형적인 코드 패턴 세 가지가 있음

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

- exports에 함수를 할당하고 메인 파일에서 함수 실행

user7.js

```
exports.printUser = function() {  
  console.log('user 이름은 소녀시대입니다.');
```

```
};
```

module_test7.js

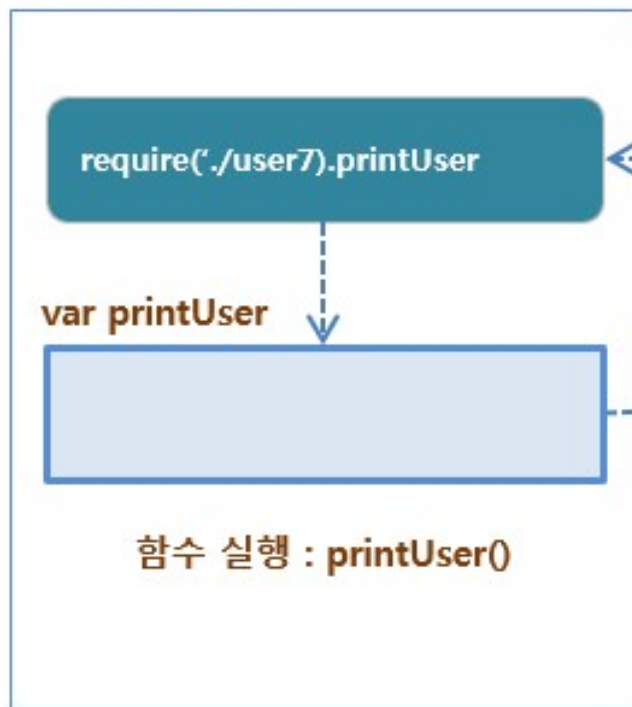
// 사용 패턴: exports에 속성으로 추가된 함수 객체를 그대로 참조한 후 호출함

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

- exports에 함수를 할당하고 메인 파일에서 함수 실행

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

module_test7.js



user7.js



모듈 파일 불러들이기

- 모듈 안에서 인스턴스 객체를 만들어 할당

user8.js

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

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

User.prototype.group = {id : 'group1', name : '친구'};
User.prototype.printUser = function() {
  console.log('user 이름 : %s, group 이름 : %s', this.name, this.group.name);
}

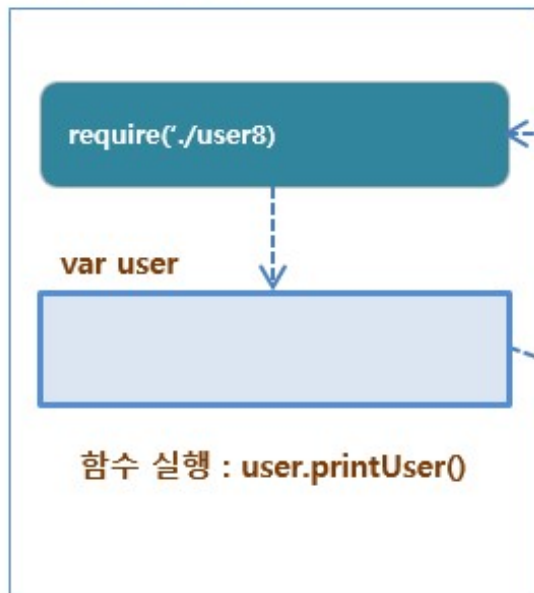
module.exports = new User('test01', '소녀시대');
```

- 메인 파일에서 인스턴스 객체의 메소드를 바로 실행할 수 있음

module_test8.js

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

module_test8.js



user8.js

모듈 파일 불러들이기

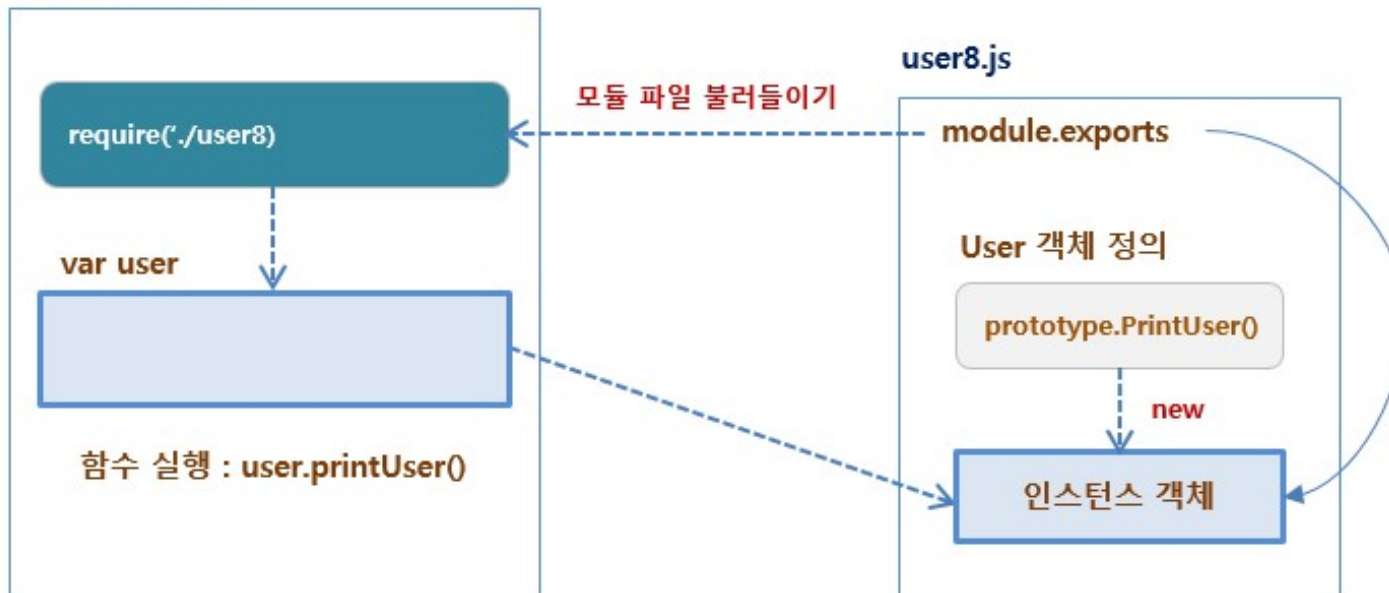
module.exports

User 객체 정의

prototype.PrintUser()

new

인스턴스 객체



- 프로토타입만 정의한 후 module.exports에 할당하면 메인 파일에서 new 연산자 사용 가능

user10.js

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

module.exports = User;
```

module_test10.js

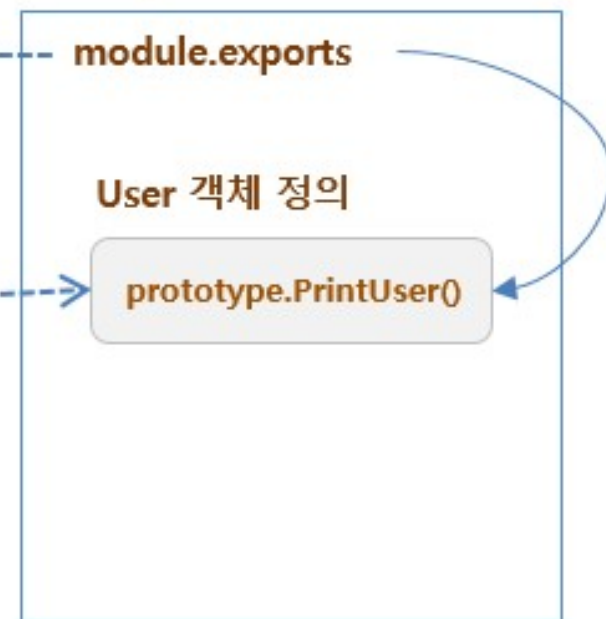
```
var User = require('./user10');
var user = new User('test01', '소녀시대');
user.printUser();
```


- 프로토타입만 정의한 후 module.exports에 할당하면 메인 파일에서 new 연산자 사용 가능

module_test10.js

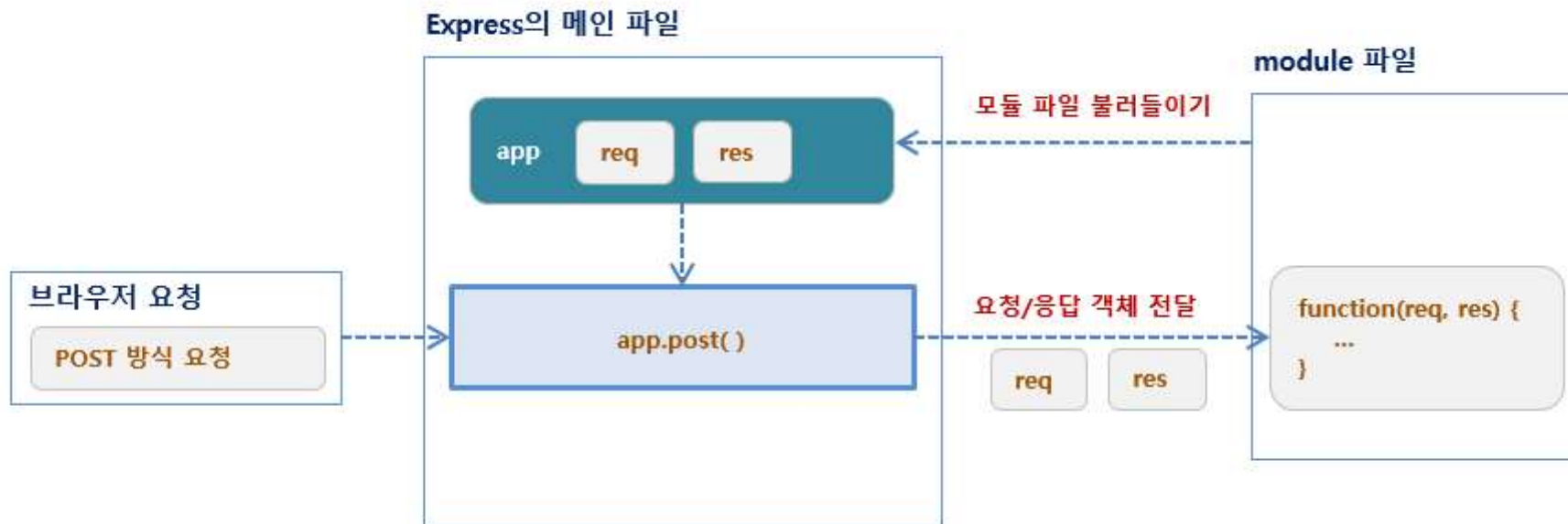


user10.js



07-02 사용자 정보 관련 기능을 모듈 화하기

- 라우터
 - 특정 패스에 따라 클라이언트가 요청한 특정 기능을 수행
 - 요청한 기능을 모듈로 분리하여 실행



- app5.js에서 스키마 정의 부분을 모듈로 분리

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

```
function createUserSchema(){  
  
  userSchema = mongoose.Schema({  
    id: {type: String, required: true, unique: true, 'default': ' '},  
    hashed_password: {type:String, required: true, 'default': ' '},  
    salt:{type:String, required: true},  
    name: {type: String, index:'hashed', 'default': ' '},  
    age:{type: Number, 'default': -1},  
    created_at:{type: Date, index:{unique:false}, 'default': Date.now},  
    updated_at: {type: Date, index:{unique:false}, 'default': Date.now}  
  });  
  .....  
}
```

별도의 모듈
파일(user_schema.js)
로 분리

- database 폴더 안에 user_schema.js 파일을 만들고 스키마 코드 분리

user_schema.js – app5.js에서 복사 후 수정

```
var Schema = { };
Schema.createSchema = function(mongoose) {

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

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

mongoose 를 파라미터로 받음

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

- 분리된 모듈 파일을 불러온 후 사용 가능

app6.js – app5.js에서 복사 후 수정

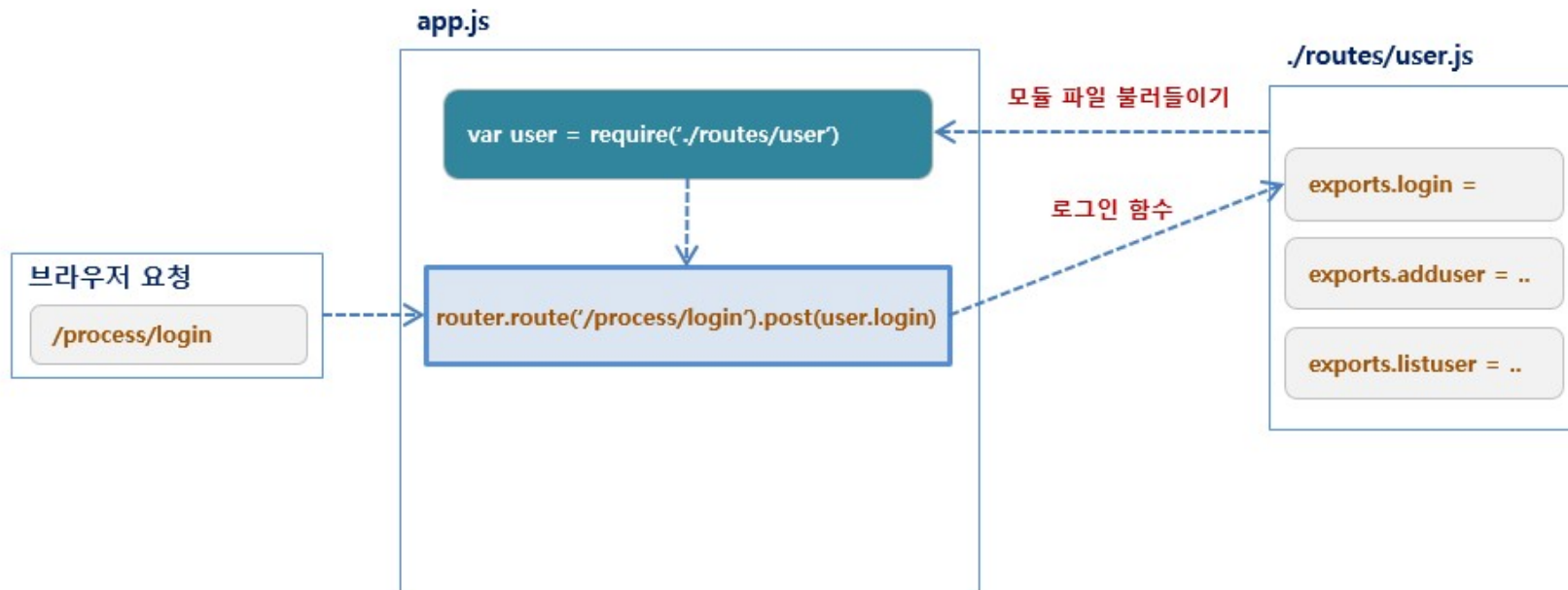
```
// user 스키마 및 모델 객체 생성
function createUserSchema() {
  // user_schema.js 모듈 불러오기
  UserSchema = require('./database/user_schema').createSchema(mongoose);

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

.....

- 라우팅 함수를 정의하는 부분을 별도의 모듈 파일로 분리

```
router.route('/process/login').post(function(req, res) {  
  
router.route('/process/adduser').post(function(req, res) {  
  
router.route('/process/listuser').post(function(req, res) {
```



라우팅 함수들을 별도의 모듈 파일로 분리

- Routes 폴더 안에 user.js 파일을 만들고 라우팅 함수 정의

user.js

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

.....

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

.....

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

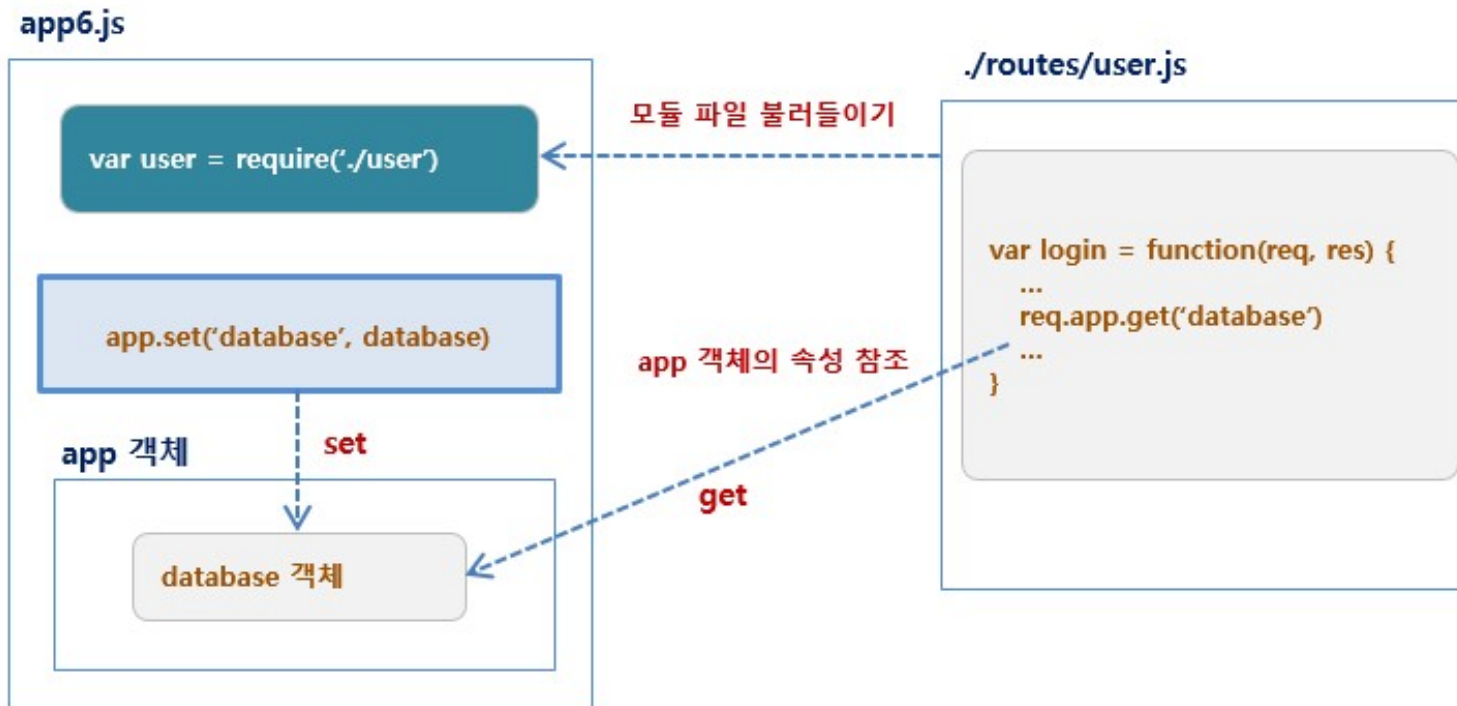
.....

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

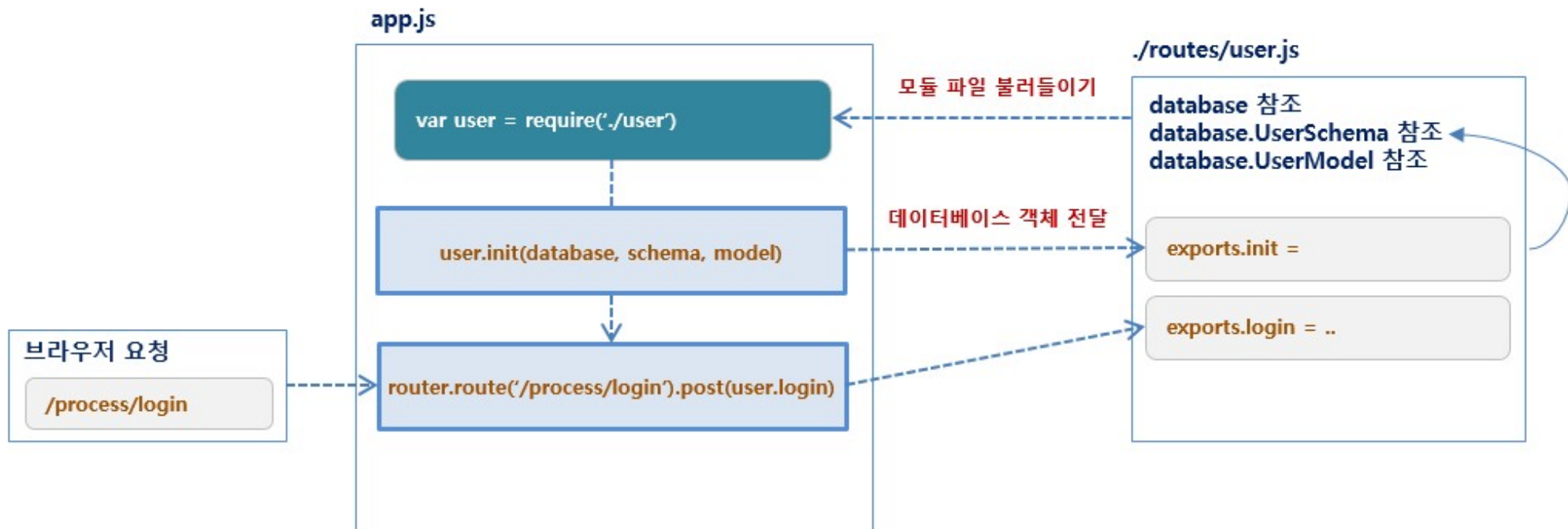
- 그대로 실행 시 에러 발생
- database 관련 객체(데이터베이스 객체, 스키마 객체, 모델 객체) 가 없음
 - ⇒ req 객체가 express의 app 객체를 속성으로 가지고 있으므로 이를 활용
 - ⇒ 또는 별도의 초기화 함수 제작

① app 객체의 set()과 get() 메소드 사용

- 필요한 데이터베이스 객체를 모듈 파일로 전달 가능



② init() 메소드를 만들어 설정



init() 메소드를 호출하여 객체 전달

- init() 메소드를 만들고 호출했을 때 객체 전달하도록 구성

user.js

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



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



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

메인 파일에서 init() 메소드 호출

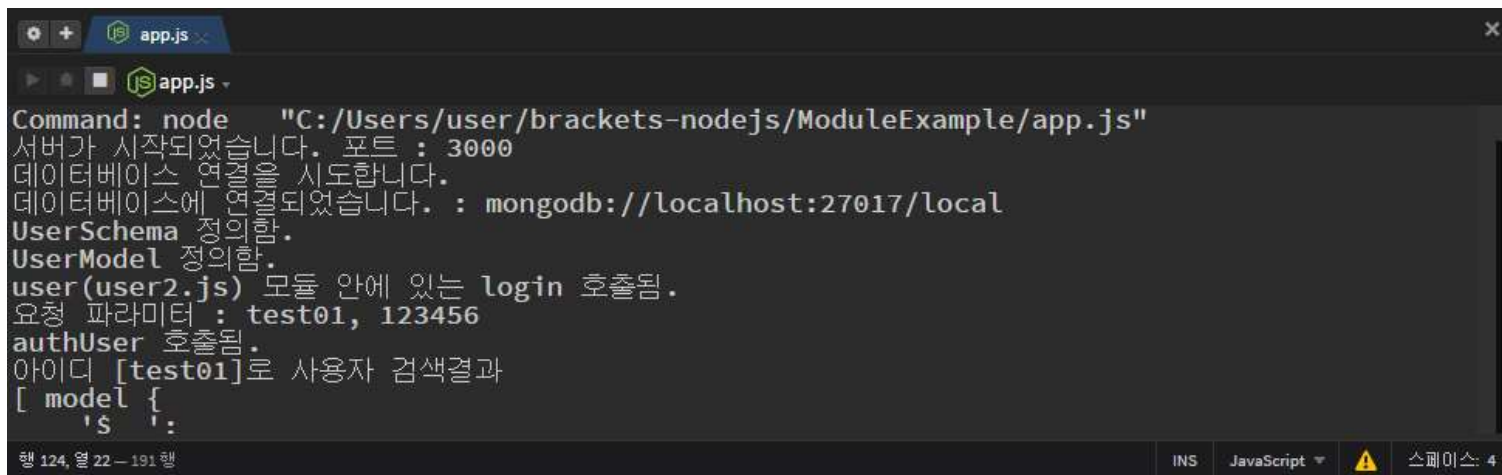
- 데이터베이스 스키마를 만든 후 user.js 파일의 init() 메소드 호출

app6.js

```
function createUserSchema() {  
  .....  
  
  // init 호출  
  user.init(database, UserSchema, UserModel);  
  
}  
  
.....
```

- mongod 실행되어 있는지 확인하고 app.js 파일 실행
- 웹브라우저 열고 테스트

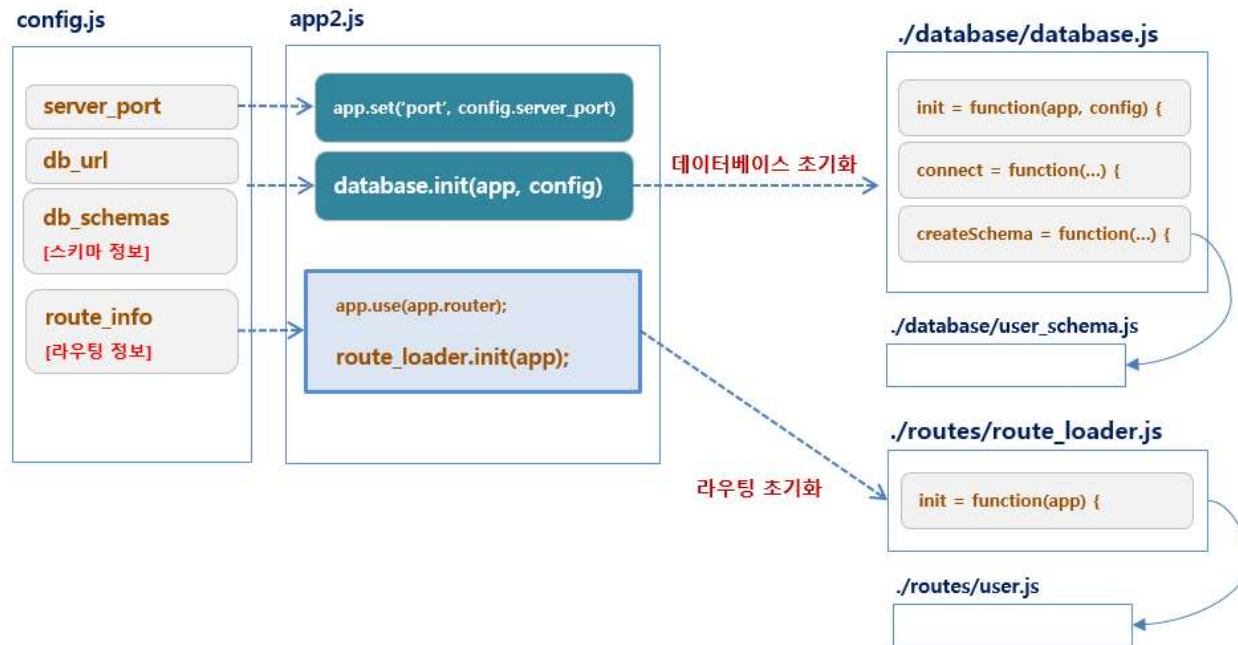
- ▶ <http://localhost:3000/public/login.html>
- ▶ <http://localhost:3000/public/adduser.html>
- ▶ <http://localhost:3000/public/listuser.html>



```
Command: node "C:/Users/user/brackets-nodejs/ModuleExample/app.js"
서버가 시작되었습니다. 포트 : 3000
데이터베이스 연결을 시도합니다.
데이터베이스에 연결되었습니다. : mongodb://localhost:27017/local
UserSchema 정의함.
UserModel 정의함.
user(user2.js) 모듈 안에 있는 login 호출됨.
요청 파라미터 : test01, 123456
authUser 호출됨.
아이디 [test01]로 사용자 검색결과
[ model {
  '$_id':
```

07-03 설정 파일 만들기

- 설정 정보
 - 서버에서 수시로 변경되는 정보
 - 예) 새로 추가된 모듈, 포트번호 등
 - 별도의 파일로 작성 후 읽어서 적용 ⇒ 변경이 발생할 때 마다 소스코드를 추가 수정하는 불편함이 사라짐
- 설정 정보를 config.js 파일에 분리한 후에 메인 파일에서 읽어서 적용함



- 포트 정보와 db url 정보

config.js

```
module.exports = {  
  server_port : 3000,  
  db_url : 'mongodb://localhost:27017/local',  
  .....
```

app2.js – app.js 복사 후 추가 수정

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


설정 파일에 데이터베이스 스키마 정보 넣기

- 데이터베이스 스키마 정보를 db_schemas 배열로 저장

config.js

```
module.exports = {
  server_port : 3000,
  db_url : 'mongodb://localhost:27017/local',
  .....
  db_schemas: [
    {file: './user_schema', collection: 'users3', schemaName: 'UserSchema', modelName: 'UserModel'}
  ],
  .....
}
```

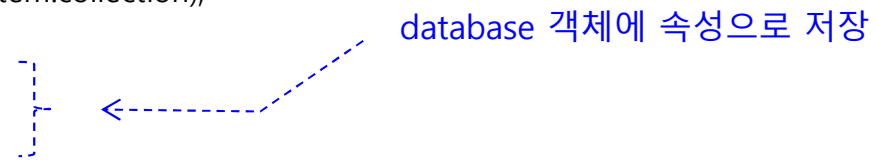
- 데이터베이스 스키마 정보를 위해 정의한 속성

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

- 데이터베이스 처리를 위한 database.js 모듈 분리
 - 설정 정보에 등록된 정보를 이용해 모듈 파일 로딩하고 app.set() 으로 데이터베이스 객체를 속성으로 설정

database.js

```
function createSchema(app, config) {  
  var schemaLen = config.db_schemas.length;  
  console.log('설정에 정의된 스키마의 수 : %d', schemaLen);  
  
  for (var i = 0; i < schemaLen; i++) {  
    var curItem = config.db_schemas[i];  
  
    var curSchema = require(curItem.file).createSchema(mongoose);  
    console.log('%s 모듈을 불러들인 후 스키마 정의함.', curItem.file);  
  
    var curModel = mongoose.model(curItem.collection, curSchema);  
    console.log('%s 컬렉션을 위해 모델 정의함.', curItem.collection);  
  
    database[curItem.schemaName] = curSchema;  
    database[curItem.modelName] = curModel;  
    console.log('스키마 이름 [%s], 모델 이름 [%s]이 database 객체의 속성으로 추가됨.', curItem.schemaName, curItem.modelName);  
  }  
  app.set('database', database);  
  console.log('database 객체가 app 객체의 속성으로 추가됨.');
```



database 객체에 속성으로 저장

```
}
```

데이터베이스 스키마를 새로 정의하는 과정

- 스키마 파일을 만든 후 config.js에 등록만 하면 되도록 구성

config.js

db_schemas

[스키마 정보]

file: './board_schema'

collection: 'board'

schemaName: 'BoardSchema'

modelName: 'BoardModel'

./database/board_schema.js

스키마 정의

1단계 : 스키마 정의 파일 추가

2단계 : 설정 정보 추가

- 설정 파일에 라우팅 함수의 정보를 배열로 넣어줌

config.js

```
var user = require('./routes/user2');
module.exports = {
  .....
  route_info : [
    {file : './user', path : '/process/login', method : 'login', type : 'post'}
    , {file : './user', path : '/process/adduser', method : 'adduser', type : 'post'}
    , {file : './user', path : '/process/listuser', method : 'listuser', type : 'post'}
  ]
}
```

- 라우팅 정보를 위해 정의한 속성

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

- 라우팅 함수를 정의하는 모듈 파일을 만든 후 config.js 파일에 설정 추가

config.js

route_info
[라우팅 정보]

file: './board'

path: '/process/listboard'

method: 'listboard'

type: 'post'

./routes/board.js

라우팅 코드

1단계 : 라우팅 코드 파일 추가

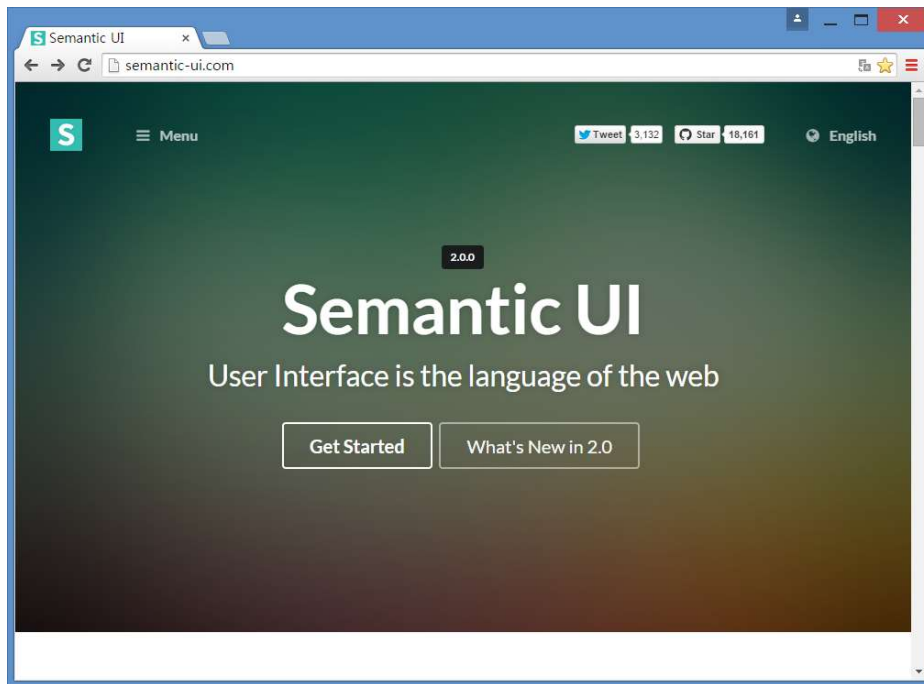
2단계 : 설정 정보 추가

07-04 UI 라이브러리로 웹 문서 예쁘게 꾸미기

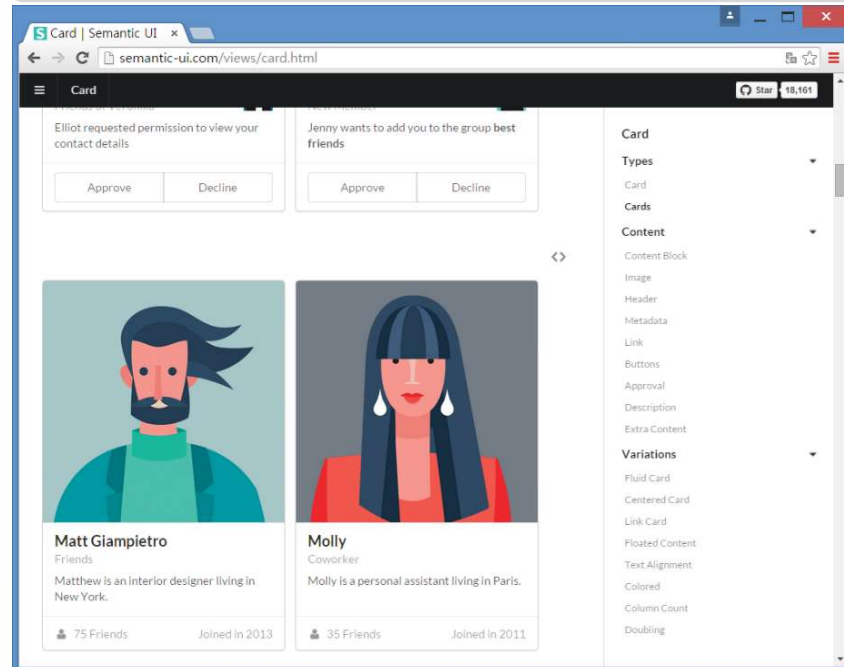
웹 페이지를 라이브러리를 이용해 꾸미기

- 부트스트랩이나 Semantic UI 등을 이용해 웹 페이지를 예쁘게 꾸밀 수 있음

▶ <http://semantic-ui.com>



% npm install semantic-ui --save



• CSS 파일과 자바스크립트 파일 불러오기

login.html

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

  <title>로그인</title>

  <link href = "../semantic.min.css" rel = "stylesheet">
  <script src = "../jquery-2.1.4.min.js"> </script>
  <script src = "../semantic.min.js"> </script>
  .....
```

- name속성이 viewport 인 meta 태그는 모바일 용으로 많이 사용
- 단말기의 가로 세로 크기에 맞추어 웹 페이지 크기 조정

- semantic-ui 패키지 설치 후
node_modules/semantic-ui 폴더에 있음
- 현재 폴더로 복사

- 태그를 이용해 카드 모양으로 만듦

login.html

로그인에 필요한 컴포넌트를 모으기 위해
container 사용

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

<table> 태그로 로그인 입력상자와 버튼 추가

- <table> 태그로 아이디,비번, 로그인 버튼 화면 모양 구성

login.html

```
<form id = "form1" method = "post" action = "/process/login">
  <table>
    <tr class = "row">
      <td class = "col1"><label id = "contentsText">아이디</label></td>
      <td class = "col2" colspan = "2">
        <div class = "ui input">
          <input class = "inputbox" type = "text" name = "id" />
        </div>
      </td>
    </tr>
    <tr class = "row">
      <td class = "col1"><label id = "contentsText">비밀번호</label></td>
      <td class = "col2" colspan = "2">
        <div class = "ui input">
          <input class = "inputbox" type = "password" name = "password" />
        </div>
      </td>
    </tr>
  </table>
</form>
```

- <style> 태그로 전체 화면 꾸밈

login.html

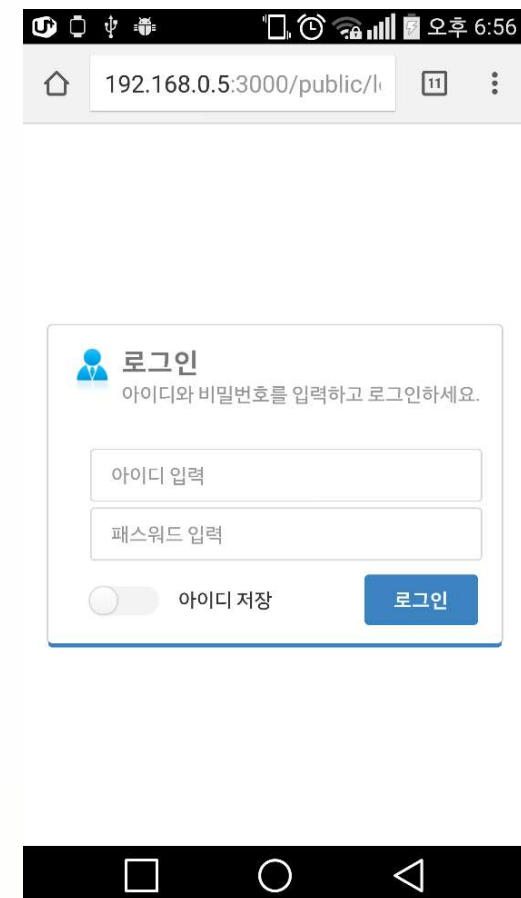
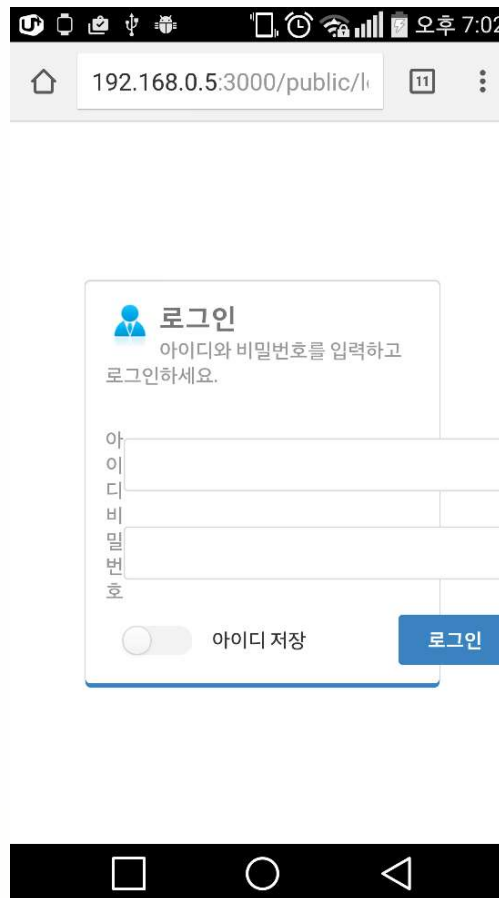
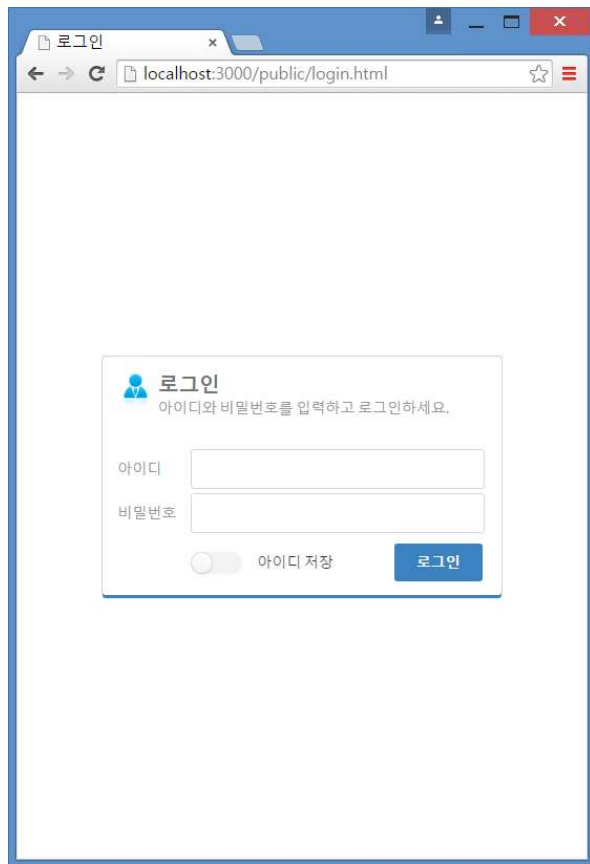
```
<style>
* {
  padding : 0;
  margin : 0;
  box-sizing : border-box;
}

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

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

페이지 열고 확인

- login.html 페이지 열고 확인
- 모바일 화면에 맞도록 추가적인 CSS 조정 필요

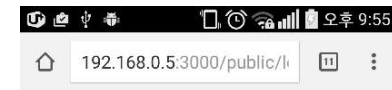


- 단말의 종류에 따라 다르게 보이도록 만들 수 있음
 - 일종의 반응형 웹

login_responsive.html – login.html 복사 후 수정

```
@media screen and (min-width : 320px) and (max-width : 768px) {  
    #cardbox {  
        width : 90%;  
    }  
  
    label[id = contentsText] {  
        display : none;  
    }  
}
```

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

A login form titled '로그인' (Login). Below the title is a subtitle: '아이디와 비밀번호를 입력하고 로그인하세요.' (Enter your ID and password and log in). There are two input fields: one for the ID and one for the password. Below the password field is a checkbox labeled '아이디 저장' (Remember ID) which is currently unchecked. To the right of the checkbox is a blue button labeled '로그인' (Login).