11 JSON-RPC 서버 만들기

지금까지 웹 서버를 만든 후 채팅 서버까지 만들어 보았습니다. 서버와 클라이언트 사이에서 데이터를 주고받는 과정이 조금은 복잡하게 느껴질 수 있습니다. 특히 서버 쪽에 다양한 기능이 들어가면 소스 코드는 더 복잡해집니다. 이렇게 복잡해지는 코드든 모듈을 만들어 좀 더 간단하게 보이게 하고 각 기능도 분리해서 만들었습니다. 그럼에도 코드가 복잡해 보인다면 '어떻게 간단하게 데이터를 주고받을 수 있을까?' 하는 문제를 다시 생각해 볼 필요가 있습니다. 이 장에서는 JSON-RPC 모듈을 사용해서 서버를 더 간편하게 만드는 방법을 설명합니다. 먼저 JSON-RPC가 무엇인지 알아보고, 이것을 사용해 서버 쪽 모듈을 만들거나 웹 브라우저 쪽에서 JSON-RPC를 호출해서 사용하는 방법을 알아보겠습니다.

질문으로 시작하기

질문1. JSON-RPC는 어떻게 만드는 건가요? ➡ 11-1. JSON-RPC를 웹 서버에 적용하기

질문2. 간단한 더하기 함수를 추가할 때는 어떻게 하나요? ➡ 11-2. 계산기 모듈 추가하여 실행하기

질문3. 데이터베이스의 데이터를 조회하여 보내 줄 수도 있나요? ➡ 11-3. 데이터베이스에서 사용자 리스트 조회하기

질문4. 데이터를 암호화하여 보내 줄 수도 있나요? ➡ 11-4. 데이터 부분을 암호화하기

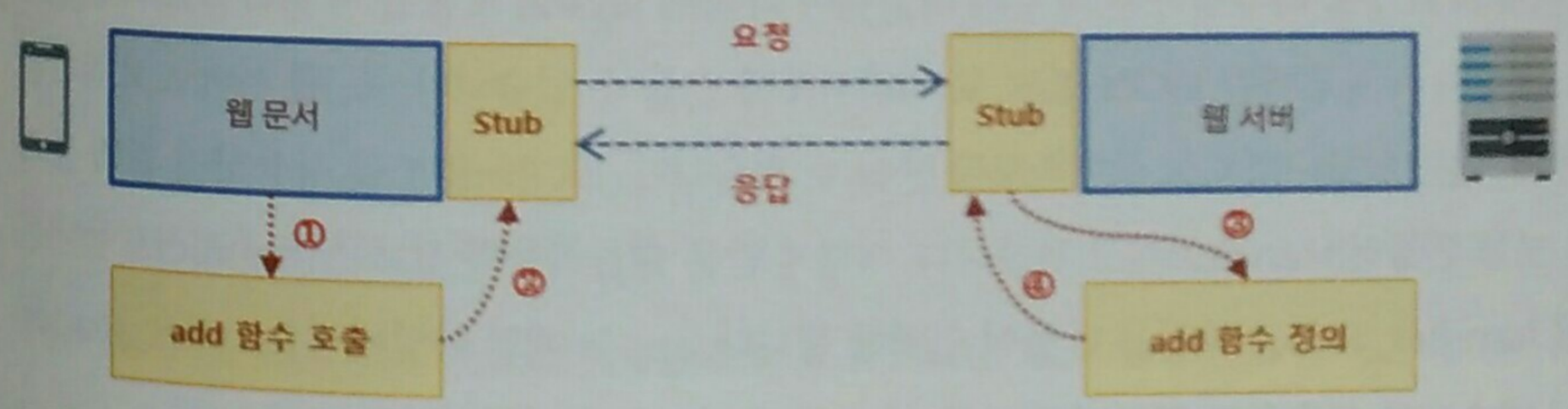
11-1 | JSON-RPC를 웹 서버에 적용하기

RPC(Remote Procedure Call)는 서버에 데이터를 요청하여 응답받는 과정을 라이브러리에서 자동으로 처리합니다. 개발자가 단순히 로컬 단말에서 함수를 호출하는 것처럼 코드를 만들기만 해도 클라이언트와 서버 간에 데이터를 주고받는 기능이 동작합니다. 따라서 이 방법을 사용하면 서버를 훨씬 쉽고 간단하게 만들 수 있습니다. RPC는 여러 가지 방법으로 사용할 수 있는데, 그중에서 특히 JSON-RPC는 JSON 포맷으로 데이터를 주고받을 수 있어 자바스크립트를 사용하는 노드 프로그램에서 훨씬 자연스럽게 사용할 수 있습니다.

네트워킹을 통해 데이터를 주고받는 각각의 기능을 단순히 하나의 함수처럼 사용하려면 먼저 JSON-RPC 모듈을 설치하고 사용 방법을 알아야 합니다. 여기에서는 JSON-RPC를 사용해 서버에 데이터를 보내고 받는 방법을 먼저 알아보겠습니다.

JSON-RPC 모듈 설치하여 사용하기

예전부터 클라이언트와 서버가 데이터를 주고받는 일은 복잡한 과정을 거쳐야 한다고 생각했습니다. 사용자의 단말에 있는 데이터가 아니기 때문에 다른 곳에 있는 데이터를 요청하는 과정과 응답을 받아 처리하는 과정이 추가로 필요하니 복잡해질 수밖에 없다고 생각한 것이죠. 하지만 먼 거리에 있는 서버에 함수를 하나 만들어 두고 클라이언트에서 호출하거나 응답 받는 과정을 모두 라이브러리에서 담당한다면 개발자가 복잡한 네트워킹 과정을 신경 쓰지 않아도 됩니다. 즉, 서버와 데이터를 주고받는 기능은 RPC가 담당하기 때문에 개발자는 클라이언트에서 일반 함수를 사용하듯 서버에 데이터를 요청하기만 하면 됩니다.



▲RPC를 사용한 요청과 응답

예를 들어, 서버에 더하기 함수를 만들어 두고 클라이언트에서 두 값을 더한 결과를 알고 싶을 때 RPC를 사용하면 서버에서 더하기 함수를 실행한 결과 값을 간단하게 가져올 수 있습니다.

이 과정을 더 자세히 살펴볼까요? 먼저 웹 서버에 add 함수를 만들어 둡니다. 그다음 이 함수를 클라이언트인 웹 문서에서 호출합니다. 그러면 라이브러리에서는 **스텁(Stub)**을 통해 서버로 요청을 보내고 서버에서는 add 함수를 실행한 후 그 결과 값을 응답으로 보내 줍니다. 이 과정은 라이브러리를 통해 이루어지기 때문에 개발자는 클라이언트와 서버 사이에 어떤 데이터를 어떻게 주고받는지 신경 쓰지 않아도 됩니다.

이렇게 필요한 데이터를 주고받을 때 사용하는 데이터 포맷은 **XML**이나 **바이너리 포맷** 등이 있습니다. JSON 포맷을 사용하는 경우에 JSON-RPC라 부릅니다. 표준으로 만든 JSON-RPC 프로토콜에 대해 좀 더 자세히 알고 싶다면 다음 사이트를 참조하세요.

```
JSON-RPC 공식 사이트 ▶ http://www.jsonrpc.org/
JSON-RPC Wiki 사이트 ▶ https://en.wikipedia.org/wiki/JSON-RPC
```

JSON-RPC에는 어떤 OS와 언어를 사용하든 서로 데이터를 주고받을 수 있도록 다양한 언어로 작성된 라이브러리가 있습니다. 노드와 익스프레스 환경에서도 JSON-RPC를 사용할 수 있는데 여기에서는 **jayson 모듈**을 사용합니다.

jayson 모듈로 JSON-RPC를 사용하는 서버를 만들기 위해 PassportExample2 프로젝트를 복사하여 JsonRpcExample 프로젝트로 만듭니다. 프로젝트를 만들었다면 먼저 jayson 모듈을 설치합니다. 명령 프롬프트를 열고 다음 명령을 입력하면 jayson 모듈이 설치됩니다.

```
% npm install jayson --save
```

jayson 모듈을 사용하려면 우선 app.js 파일 안에서 require( ) 메소드를 호출해서 모듈을 불러들여야 합니다. 클라이언트의 요청을 라우팅하는 라우터 미들웨어 다음에 jayson 모듈을 사용하면 클라이언트가 특정 패스로 요청하는 경우에만 JSON-RPC로 처리되도록 만들 수 있습니다. 즉, 웹 서버로 요청하는 패스 중 한 가지 패스를 JSON-RPC로 실행하도록 만들 수 있습니다. JSON-RPC를 사용하기 위해 등록한 각 함수들은 보통 **핸들러(Handler)**라고 부릅니다. 이렇게 만든 핸들러들의 정보는 [handlers] 폴더를 만든 후 그 안에 handler_info.js 파일을 만들어 입력해 둡니다. app.js 메인 파일에서 handler_loader.js 파일의 init( ) 메소드를 호출하면 핸들러 정보를 읽어 들여 경로를 설정합니다.

참조 파일 | JsonRpcExample>app.js

```
……

var handler_loader = require('./handlers/handler_loader');

……

// JSON-RPC 사용
var jayson = require('jayson');

……

// JSON-RPC 핸들러 정보를 읽어 들여 핸들러 경로 설정
var jsonrpc_api_path = config.jsonrpc_api_path || '/api';
handler_loader.init(jayson, app, jsonrpc_api_path);

console.log('JSON-RPC를 [' + jsonrpc_api_path + '] 패스에서 사용하도록 설정함.');

……
```

config.js 파일에 등록되어 있는 라우팅 함수 중에서 하나의 패스가 JSON-RPC로 처리되도록 설정되어 있습니다. 이 정보는 jsonrpc_api_path 속성에 들어 있으며, 여기에서 지정한 라우팅 패스로 들어오는 요청은 JSON-RPC로 처리됩니다. 기본 값으로 /api 패스가 들어 있으나 필요에 따라 다른 패스로 바꿀 수도 있습니다.

참조 파일 | JsonRpcExample>/config/config.js

```
module.exports = {

……

  jsonrpc_api_path :  '/api'
}
```

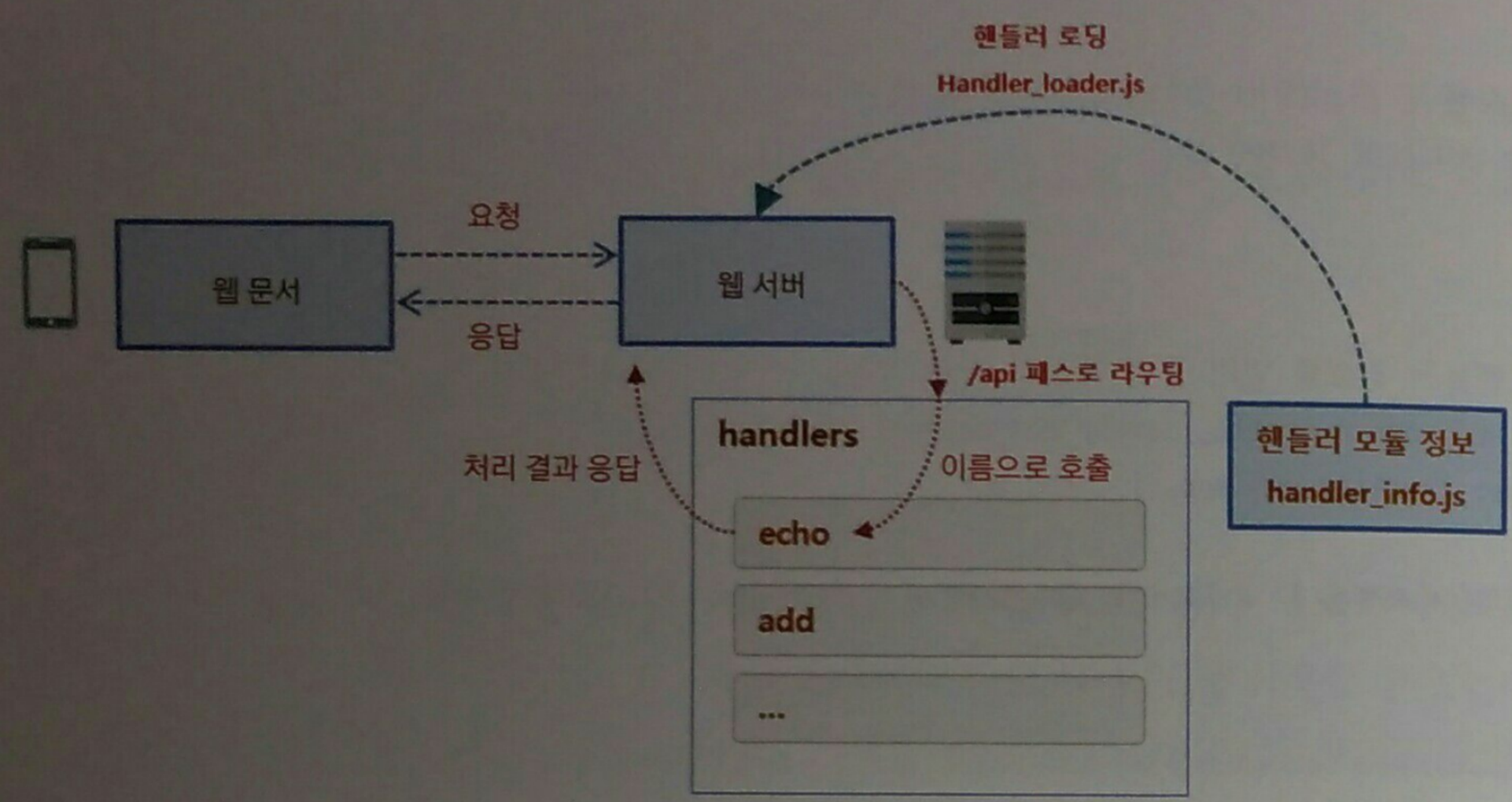
이 문자열은 다음과 같은 패스로 요청할 수 있음을 의미합니다.

▶ http://localhost : 3000/api

handler_loader.js 파일에 있는 init( ) 메소드에는 핸들러 정보들을 로딩하여 JSON-RPC 서버에 설정하는 코드가 들어 있습니다. 코드가 조금 복잡해 보이지만 여기에서는 JSON-RPC 모듈 정

보가 [handlers] 폴더의 handler_info.js 파일에 들어 있다는 것만 기억해도 충분합니다. 다시 말해 handler_loader.js 파일은 핸들러를 좀 더 편리하게 등록할 수 있도록 만들어 둔 것이므로 소스 코드를 이해하지 않아도 상관없습니다. 이 책에서 제공하는 프로젝트 파일의 소스 코드를 참조하여 입력합니다.

그러면 먼저 JSON-RPC 서버에 핸들러를 추가하는 구조를 확인해 보겠습니다.



▲ JSON-RPC 서버의 구성

핸들러는 서버 쪽에 만들어지는 하나의 함수라고 생각하면 쉽습니다. 각각의 함수는 [handlers] 폴더 안에 하나의 독립적인 모듈 파일로 만들 수 있습니다. 핸들러 파일을 만들었다면 [handlers] 폴더 안에 있는 handler_info.js 파일 안에 등록 정보를 넣어 줍니다. handler_info.js 파일의 내용은 다음 코드와 같습니다.

참조 파일 | JsonRpcExample>/handlers/handler_info.js

```
console.log('handler_info 파일 로딩됨.');

var handler_info = [
    {file : './echo', method : 'echo'}
];

module.exports = handler_info;
```

module.exports에 할당되는 객체는 배열로 만들어져 있으며, 그 안에 file과 method 속성을 가진 객체들이 순서대로 들어갑니다. file 속성은 핸들러 모듈 파일의 이름이고 method 속성은 등록한 핸들러의 이름입니다. 이 핸들러의 이름으로 클라이언트가 호출하기 때문에 서버에 핸들러를 등록할 때는 어떤 이름으로 등록했는지 클라이언트가 알고 있어야 합니다.

echo 함수 만들어 실행하기

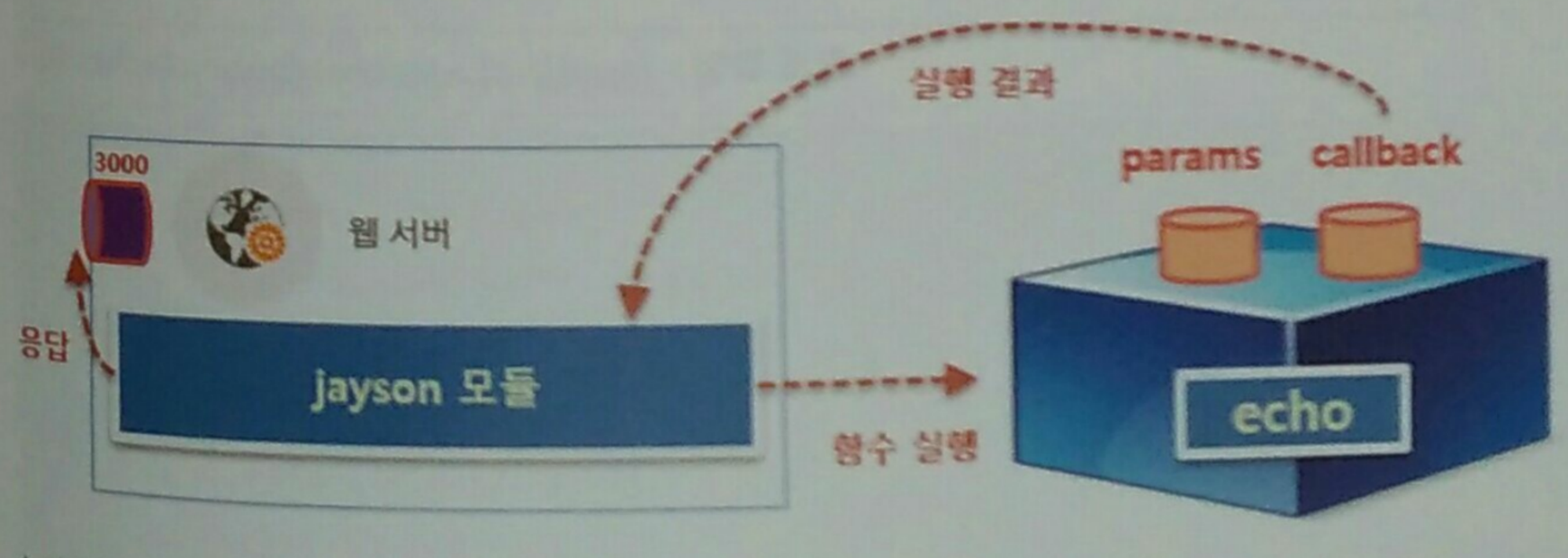
echo 핸들러는 클라이언트에서 보낸 데이터를 그대로 클라이언트로 다시 보내 주는 단순한 기능을 하는 함수입니다. 이 echo 핸들러는 [handlers] 폴더의 echo.js 파일 안에 들어 있습니다. 앞에서 설명한 것처럼, 이렇게 만든 핸들러 파일의 정보는 handler_info.js 설정 파일 안에 들어 있는 handler_info 객체에 추가합니다. file 속성 값은 ./echo로 설정하고 method 속성 값은 echo로 설정합니다. 이제 이 파일에 들어 있는 핸들러 함수는 클라이언트에서 echo라는 이름으로 호출할 수 있습니다.

[handlers] 폴더 안에 echo.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | JsonRpcExample>/handlers/echo.js

```
var echo = function(params, callback) {
  console.log('JSON-RPC echo 호출됨.');
  console.dir(params);
  callback(null, params);
};

module.exports = echo;
```

module.exports에 할당된 함수는 두 개의 파라미터를 전달받습니다. 첫 번째 파라미터는 클라이언트로부터 전달받은 것이며 배열 객체로 되어 있습니다. 두 번째 파라미터는 함수이며 클라이언트로 응답을 보낼 때 사용합니다. 이 callback 함수는 두 개의 파라미터를 전달받을 수 있는데, 첫 번째 파라미터는 오류 전달을 위해 사용하고 두 번째 파라미터는 정상적인 데이터를 전달할 때 사용합니다.



▲서버 쪽에 만드는 JSON-RPC 함수의 구조

클라이언트에서 보내 온 데이터를 그대로 다시 보낼 것이므로 callback( ) 함수를 호출하면서 params 객체를 그대로 넣어 줍니다.

주의! 핸들러 함수는 결과 값을 callback( ) 함수를 사용해서 전달하므로 반환 값이 없다는 점에 주의하세요. 즉, return 키워드를 사용하지 않습니다.

서버 쪽에 echo 함수를 만들고 handler_info.js 파일에 등록했으니 클라이언트에서 echo 함수를 실행하라고 요청할 수 있습니다. [public] 폴더에 echo.html 파일을 만들고 〈body〉 태그 안에 다음 코드를 넣어 줍니다.

참조 파일 | JsonRpcExample>/public/echo.html

```
……

<body>
  <h3>JSON-RPC Echo 테스트</h3>
  <br>
  <textarea name = "messageInput" id = "messageInput"></textarea>
  <br>
  <input type = "button" name = "requestButton" id = "requestButton" value = "요청하기">
  <br>
  <p>결과</p>
  <div id = "results"></div>
</body>
```

〈body〉 태그 안에는 글자를 입력할 수 있는 〈textarea〉 태그를 하나 입력하고 id 속성 값을 messageInput 으로 지정합니다. 그 아래에는 [요청하기] 버튼을 하나 만들고 id 속성 값을 requestButton으로 지정합니다. 결과를 보여 주기 위한 〈div〉 태그를 추가하고 id 속성 값을 results로 지정합니다. 〈script〉 태그 안에 있는 [요청하기] 버튼을 눌렀을 때 서버에 요청하도록 다음과 같이 자바스크립트 코드를 추가합니다.

참조 파일 | JsonRpcExample>/public/echo.html

```
……

  <script src = "jquery-2.1.4.min.js"></script>
  <script src = "jquery.jsonrpc.js"></script>
  <script>
    $(function() {
      $.jsonRPC.setup({
        endPoint : 'http://localhost:3000/api',
        namespace : ''
      });

……
```

웹 문서에서는 제이쿼리를 사용할 것이므로 jquery.com 사이트에서 제이쿼리 라이브러리 파일을 다운로드합니다. 다운로드한 파일은 프로젝트 폴더의 [public] 폴더 안에 넣고 〈script〉 태그의 src 속성으로 지정합니다. JSON-RPC 클라이언트 라이브러리도 필요한데 여기에서는 jquery-jsonrpc 라이브러리를 사용합니다. 다음 사이트에서 자바스크립트 파일을 다운로드한 후 마찬가지로 [public] 폴더 안에 넣습니다.

▶ https://github.com/datagraph/jquery-jsonrpc

〈script〉 태그를 하나 더 추가한 다음 src 속성으로 다운로드한 파일을 지정합니다. 그 아래의 〈script〉 태그에는 문서가 로딩된 후에 자동으로 호출되도록 초기화 함수를 추가합니다. 초기화 함수 안에서 JSON-RPC 클라이언트를 사용할 수 있도록 메소드를 호출합니다. $.jsonRPC.setup( ) 메소드를 호출하면 기본 설정을 할 수 있습니다. endPoint 속성에는 접속할 서버의 URL을 넣어 주고 namespace 속성은 비워 둡니다. 여기에서는 로컬 PC의 3000번 포트로 접속하면서 /api 패스를 사용하므로 URL은 http://localhost:3000/api를 사용합니다.

이제 [요청하기] 버튼을 클릭했을 때 서버에 만들어 둔 echo 함수를 호출하는 코드를 입력합니다.

참조 파일 | JsonRpcExample〉/public/echo.html

```
……

$("#requestButton").click(function() {
  var message = $("#messageInput").val();
  var method = 'echo';

  $.jsonRPC.request(method, {
    id : 1001,
    params : [message],
    success : function(data) {
      println('정상 응답을 받았습니다.');
      console.dir(data);
      println(data.result);
    },
    error: function(data) {
      println('오류 응답을 받았습니다.');
      console.dir(data);
      println(data.error.message);
    }
  });
```

```
        println('[' + method + '] method로 요청을 보냈습니다.');
      });
    });

……
```

method 변수에는 서버 쪽의 함수 이름을 넣어 두었는데 서버 쪽에서 함수를 만들고 이름을 echo로 등록했기 때문에 클라이언트도 같은 이름을 사용해야 합니다. 서버에 요청할 때는 $.jsonRPC.request( ) 메소드를 호출할 수 있습니다. 이 메소드를 호출할 때는 첫 번째 파라미터로 method 변수를 전달하고 두 번째 파라미터에는 전달할 데이터와 콜백 함수가 들어 있는 객체를 전달합니다. 두 번째 파라미터로 전달되는 객체의 속성은 다음과 같습니다.

속성 이름	설명
id	요청 ID를 지정할 수 있습니다. 이 요청 ID는 서버로부터 받는 응답을 구별하는 데 사용합니다.
params	서버로 보낼 데이터를 넣는 배열 객체입니다.
success	응답을 성공적으로 받았을 때 호출되는 콜백 함수입니다.
error	오류 응답을 받았을 때 호출되는 콜백 함수입니다.

[표] 서버에 요청할 때 전달되는 객체의 속성

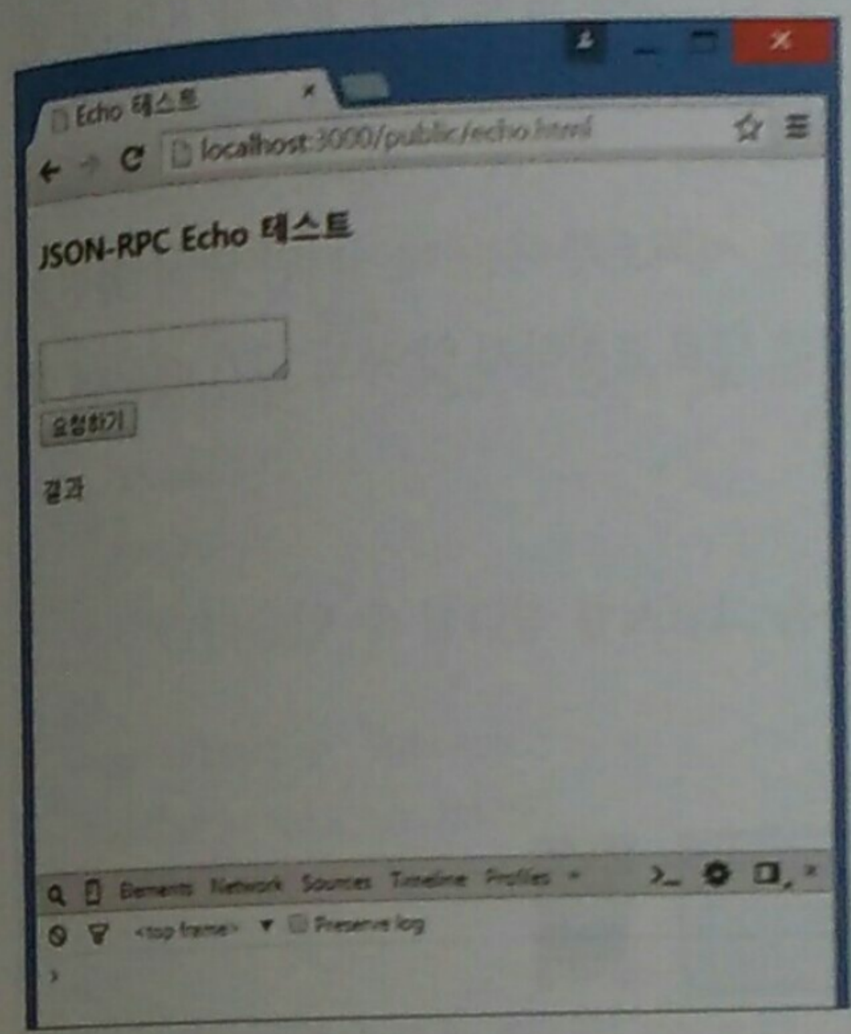
id 속성에 넣는 숫자는 응답 받을 때도 똑같이 들어 있어서 서버에 id 속성을 여러 번 요청하면 각 요청을 구별하는 역할을 합니다. params 속성에는 서버에 있는 함수를 호출하면서 전달할 데이터를 넣어 줍니다. 전달할 데이터는 배열 객체로 만들어 사용자가 웹 문서의 입력 상자에 입력한 값을 넣어 줍니다. 이렇게 서버에 id 속성을 요청하면 응답을 받게 되는데 응답을 받았을 때 호출되는 콜백 함수는 두 가지로 나뉩니다. success 속성에 설정한 함수는 성공 응답을 받았을 때 호출되는 콜백 함수입니다. 이 함수에는 data라는 이름으로 객체가 전달되며 이 객체에는 다음과 같은 속성이 들어 있습니다.

속성 이름	설명
id	요청할 때 전달한 id 값이 들어 있습니다.
jsonrpc	JSON-RPC 스펙의 버전을 표기합니다. 여기에서는 '2.0' 입니다.
result	응답 데이터가 배열 객체로 들어 있습니다.

[표] 성공 응답을 받았을 때 전달되는 객체의 속성

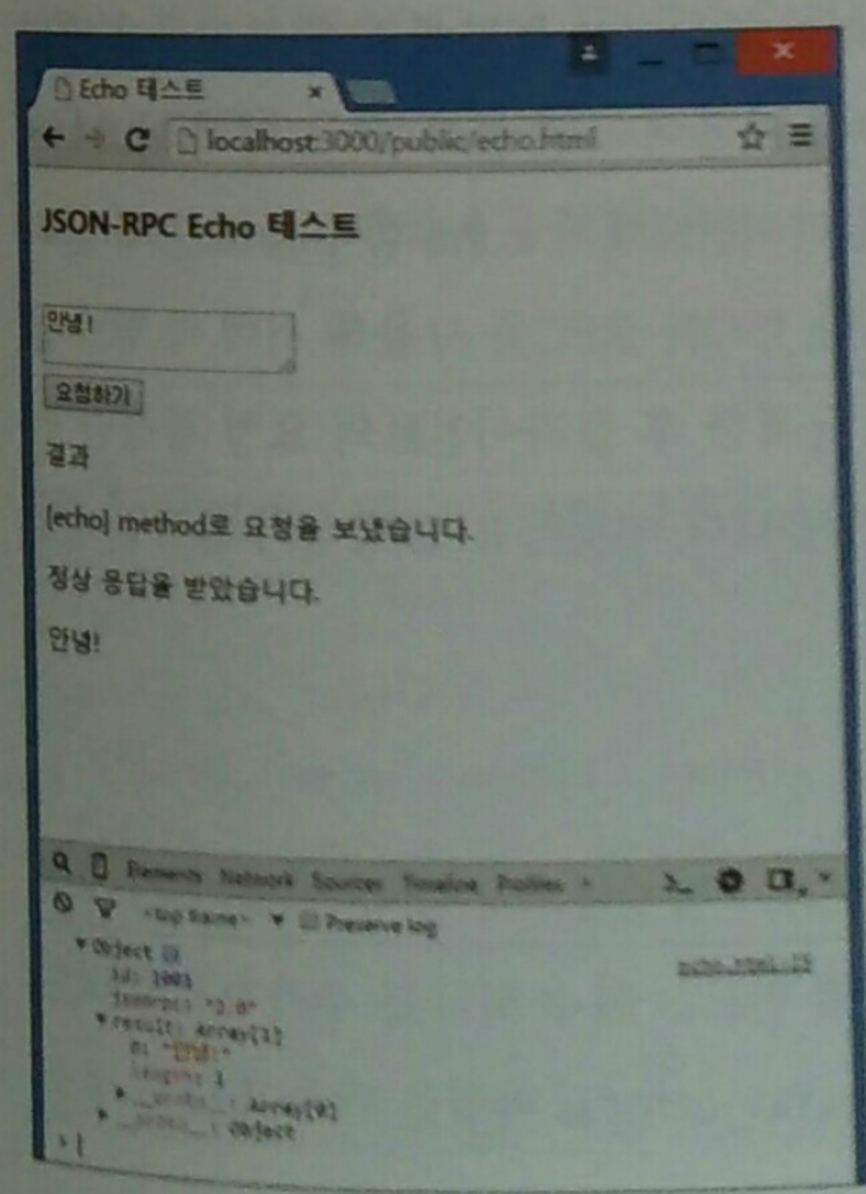
응답 데이터는 result 속성에 들어 있으므로 이 속성 값을 확인하면 서버로부터 받은 값을 알 수 있습니다. 이제 app.js 파일을 실행한 후 웹 브라우저를 열고 echo.html 파일을 띄웁니다.

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



▲ 웹 브라우저에서 echo.html 파일을 연 경우

웹 브라우저의 아래쪽에 있는 개발자 도구 창을 열고 [Console] 탭이 보이도록 설정했습니다. 입력 상자에 '안녕!'이라는 글자를 입력한 후 [요청하기] 버튼을 누릅니다. 그러면 서버 쪽에 넣어 둔 echo 함수가 실행되도록 요청하고 결과 값을 응답받아 표시합니다.

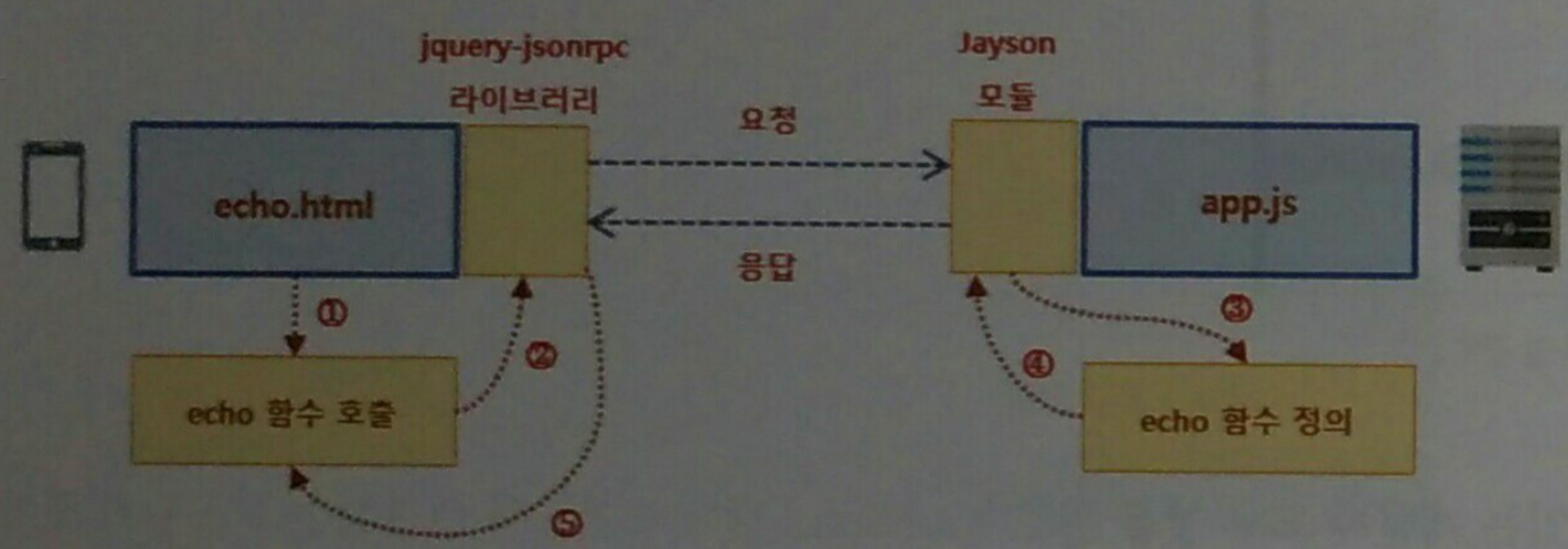


▲ 서버의 echo 함수를 실행하도록 요청하고 응답을 받아 표시한 화면

웹 화면에는 서버로부터 받은 메시지가 나타나며 [Console] 탭에서는 응답으로 받은 상세 데이터를 확인할 수 있습니다. 지금까지 서버에 있는 함수를 실행하고 그 결과를 받아 웹 브라우저 화면에 표시해 보았습니다. RPC를 직접 사용해 보니 웹 서버에 데이터를 요청하고 응답을 받는 기존 방식과 어떻게 다른가요? 좀 더 편한 것 같나요? 많은 사람들이 말하는 RPC와 기존 방식의 가장 큰 차이점은 서버에서 응답 받는 것이 웹 문서가 아니라 데이터라는 것입니다. 물론 데이터가 아니라 웹 문서의 일부를 RPC로 받아 화면에 표시하는 것도 가능하지만 일반적으로는 데이터를 전달받습니다.

웹 문서는 로컬 PC나 모바일 단말에 저장했다가 화면에 보여 주고, 서버로부터는 데이터만 받아 화면의 일부를 업데이트하는 방식을 사용합니다. 따라서 웹 문서를 완전히 새로고침하지 않아도 되기 때문에 응답 속도도 빨라집니다.

JSON-RPC를 사용해 서버의 echo 함수를 실행하는 과정은 다음 그림처럼 정리할 수 있습니다.



▲ 서버의 echo 함수를 실행하는 과정

서버를 실행하는 메인 파일은 app.js이며, 이 파일을 실행하면 미리 만들어서 등록해 둔 핸들러들을 로딩합니다. echo 함수도 핸들러 중의 하나이기 때문에 echo 함수를 만들어 두면 클라이언트에서 jquery-jsonrpc 라이브러리를 사용해 데이터를 요청할 수 있습니다. 클라이언트에서 echo 함수를 호출해 달라고 요청하면 서버에서는 jayson 모듈로 echo 함수를 호출합니다. JSON-RPC를 사용해 서버 쪽 함수를 실행하는 과정을 이해했다면, 서버 쪽에 echo 함수를 만들어 등록한 후 클라이언트의 요청 과정만 반복 작업하면 되므로 여러 가지 기능을 짧은 시간 안에 만들 수 있습니다. 이것이 RPC 방식의 장점입니다.

echo 함수의 오류 테스트하기

그런데 요청을 처리하는 과정에서 오류가 발생했을 때 어떻게 오류가 발생했는지 알 수 있을까요? 오류가 발생했을 때 확인하는 방법을 알아보기 위해 서버 쪽에 오류를 만드는 함수 하나를 추가하고 클라이언트에서 오류 응답을 받아 보겠습니다. [handlers] 폴더 안에 echo_error.js 파일을 만들고 다음 코드를 입력합니다.

참조 파일 | JsonRpcExample>/handlers/echo_error.js

```
// echo 오류 테스트 함수
var echo_error = function(params, callback) {
  console.log('JSON-RPC echo_error 호출됨.');
  console.dir(params);

  // 파라미터 체크
  if (params.length < 2) {  // 파라미터 개수 부족
    callback({
      code :  400,
      message :  'Insufficient parameters'
    }, null);
    return;
  }

  var output = 'Success';
  callback(null, output);
};

module.exports = echo_error;
```

echo_error.js 파일 안에 새로운 함수를 만들고 module.exports에 할당했습니다. 이 함수 안에서는 클라이언트에서 보내 온 데이터를 확인합니다. params 배열 객체에 데이터가 들어 있으니 이 배열 객체에 어떤 요소(Element)들이 들어 있는지 확인한 후 사용합니다. 여기에서도 이 배열에 들어 있는 요소들의 개수를 확인하는데 두 개보다 적을 때는 callback( ) 함수를 호출하여 오류 응답을 보냅니다. 응답으로 보낼 오류 객체에는 code와 message 속성이 들어갑니다. 이렇게 만든 핸들러 모듈 파일을 handler_info.js 파일에 등록합니다.

참조 파일 | JsonRpcExample>/handlers/handler_info.js

```
.....

var handler_info = [
  {file : './echo', method : 'echo'},                    // echo
  {file : './echo_error', method : 'echo_error'},                // echo_error
];

.....
```

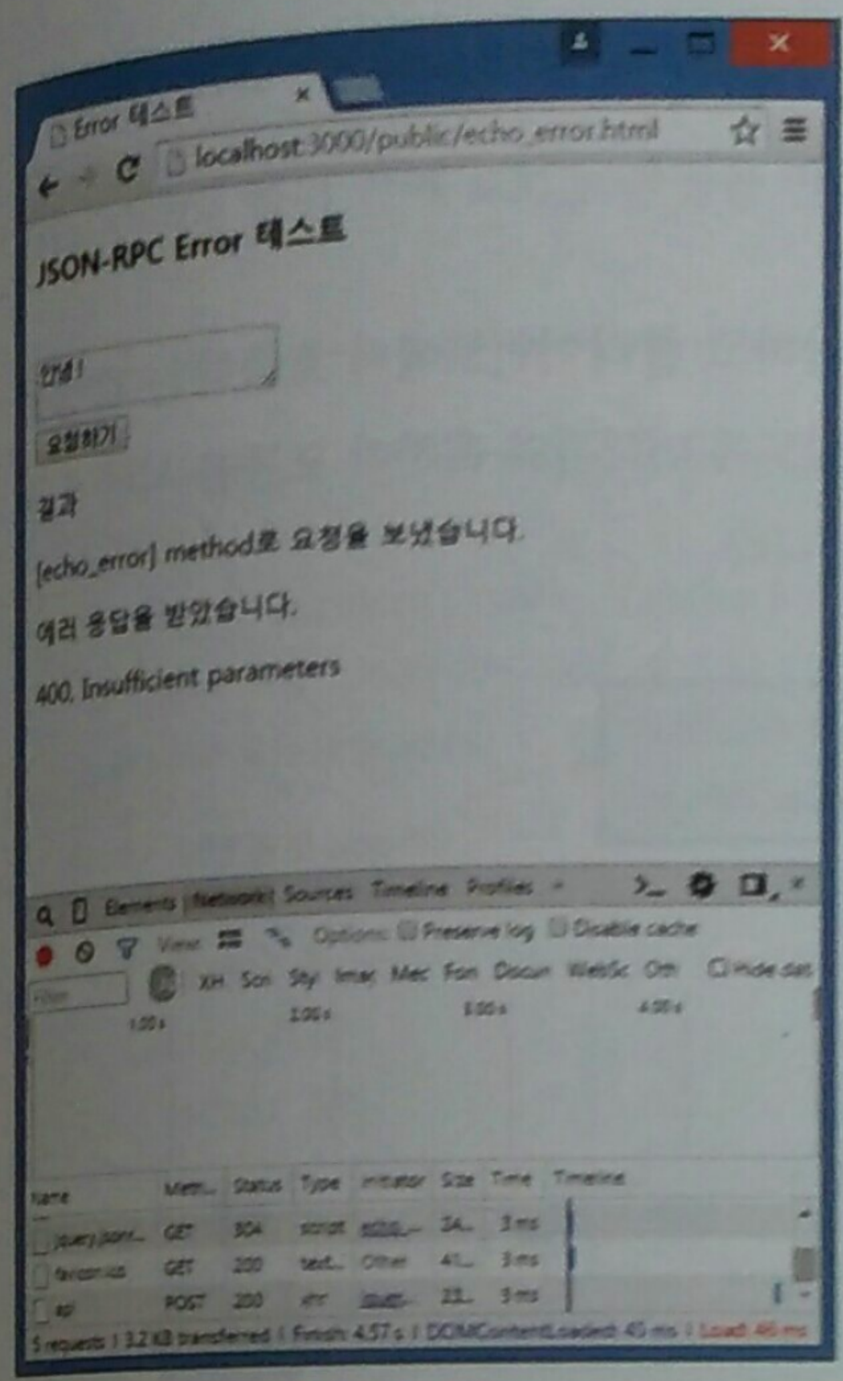
file 속성에는 핸들러 모듈 파일의 이름이 들어가야 하므로 ./echo_error 문자열을 속성 값으로 넣어 줍니다. method 속성은 클라이언트에서 호출할 때 사용할 이름으로 echo_error 문자열을 넣어 줍니다. 이제 echo.html 파일을 복사하여 [public] 폴더에 echo_error.html 파일을 만들고 클라이언트에서 서버에 데이터를 요청하는 다음 코드를 입력합니다.

참조 파일 | JsonRpcExample>/public/echo_error.html

```
.....

  $("#requestButton").click(function() {
    var message = $("#messageInput").val();
    var method = 'echo_error';

    $.jsonRPC.request(method, {
      params : [message],
      success : function(data) {
        println('정상 응답을 받았습니다.');
        console.dir(data);
        println(data.result);
      },
      error : function(data) {
        println('오류 응답을 받았습니다.');
        console.dir(data);
        println(data.error.code + ', ' + data.error.message);
      }
    });

    println('[' + method + '] method로 요청을 보냈습니다.');
  });

.....
```

[요청하기] 버튼을 눌렀을 때 호출되는 코드는 앞에서 만든 echo.html 파일의 코드와 거의 같습니다. 차이점은 요청할 때의 method 즉, 호출할 함수 이름이 echo에서 echo_error로 변경되었다는 것입니다. 이렇게 코드를 만들었으면 app.js 파일을 실행한 후 웹 브라우저에서 echo_error.html 파일을 열고 [요청하기] 버튼을 누릅니다.

▶ http://localhost:3000/public/echo_error.htm

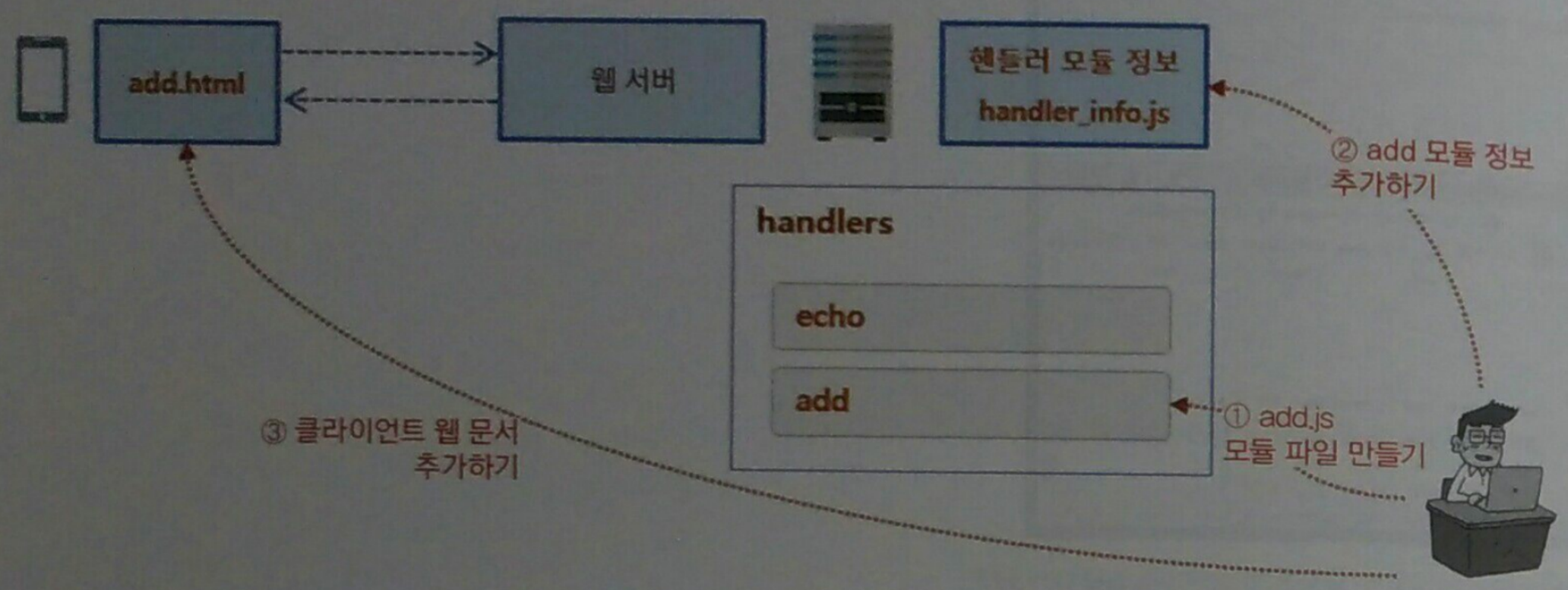


▲echo_error.html 파일에서 [요청하기] 버튼을 눌렀을 때의 화면

서버는 클라이언트로부터 전달받은 파라미터의 개수가 한 개이므로 오류 응답을 보냅니다. 이 오류 응답은 클라이언트에서 error 함수로 전달받습니다. 그리고 파라미터로 전달된 객체의 error 속성에 들어 있는 code와 message 속성 값을 웹 화면에 보여 주면 사용자가 오류 상태를 확인할 수 있습니다.

11-2 | 계산기 모듈 추가하여 실행하기

지금까지 간단한 echo 함수를 만들면서 서버 쪽에 함수를 추가하고 클라이언트에서 호출하는 과정을 살펴보았습니다. 이번에는 이 과정을 좀 더 명확하게 이해할 수 있도록 더하기와 곱하기 모듈을 서버에 추가하고 클라이언트에서 호출하는 과정을 코드로 만들어 보겠습니다.



▲ 서버에 add 모듈을 만들어 추가하는 과정

개발자가 더하기 기능을 만들어 호출하려면 다음 세 단계를 거칩니다.

(1) 첫 번째는 서버 쪽에 함수가 들어 있는 핸들러 모듈 파일을 만드는 단계입니다. [handlers] 폴더 안에 add.js 모듈 파일을 새로 만들고 그 안에 더하기 함수를 정의합니다. 그리고 새로 만든 함수를 module.exports에 할당합니다.

(2) 두 번째는 새로 만든 핸들러 모듈 파일을 등록하는 단계입니다. 핸들러 모듈의 정보는 handler_info.js 파일에 들어 있으므로 이 파일을 열고 핸들러 모듈 파일의 이름과 클라이언트에서 호출할 때 사용할 이름을 등록합니다.

(3) 세 번째는 클라이언트 웹 문서에서 서버에 만들어 둔 함수를 호출하는 단계입니다. 웹 화면에 보이는 버튼을 눌렀을 때 호출하는 코드는 JSON-RPC 라이브러리에서 정한 방식을 사용하며, 서버에 만들어 둔 add 함수의 이름과 파라미터를 입력하면 됩니다.

이 3단계를 잘 생각하면서 더하기 함수를 만든 후 호출해 보겠습니다.

1단계 : 핸들러 모듈 파일 만들기

[handlers] 폴더 안에 add.js 파일을 만들고 다음 코드를 입력합니다.

참조 파일 | JsonRpcExample>/handlers/add.js

```
// 더하기 함수
var add = function(params, callback) {
  console.log('JSON-RPC add 호출됨.');
  console.dir(params);

  // 파라미터 체크
  if (params.length < 2) {  // 파라미터 개수 부족
    callback({
      code : 400,
      message : 'Insufficient parameters'
    }, null);

    return;
  }

  var a = params[0];
  var b = params[1];
  var output = a + b;

  callback(null, output);
};

module.exports = add;
```

모듈 파일 안에 만든 더하기 함수는 전달받은 params 배열 객체에 두 개의 요소가 들어 있는지 확인한 후 두 개의 값을 더하여 응답으로 보냅니다. 결국 일반적인 더하기 함수와 기능은 같지만 결과 값을 반환하지 않고 callback 함수로 보낸다는 차이점이 있습니다.

2단계 : 핸들러 모듈 파일 등록하기

[handlers] 폴더 안에 있는 handler_info.js 파일에 새로 만든 핸들러 모듈 파일을 등록합니다. 등록할 정보는 핸들러 모듈 파일의 이름과 클라이언트에서 호출할 때 사용할 이름입니다.

참조 파일 | JsonRpcExample>/handlers/handler_info.js

```
……

var handler_info = [

……

  {file : './add', method : 'add'},

……

];

……
```

file 속성 값은 ./add로 지정하고 method 속성 값은 add로 지정합니다.

3단계 : 클라이언트 웹 문서에서 호출하기

[public] 폴더 안에 있는 echo.html 파일을 복사하여 add.html 파일을 만든 후 코드를 일부 수정합니다. 먼저 두 개의 입력 값을 받을 수 있도록 <body> 태그 안에 있는 코드를 수정합니다.

참조 파일 | JsonRpcExample>/public/add.html

```
……

  <h3>JSON-RPC Add 테스트</h3>
  <br>
  <input type = "text" value = "10" name = "aInput" id = "aInput">
  <p>+</p>
  <input type = "text" value = "10" name = "bInput" id = "bInput">
  <br>
  <br>
  <input type = "button" name = "requestButton" id = "requestButton" value = "요청하기">
  <br>
  <p>결과<p>
  <div id = "results"></div>

……
```

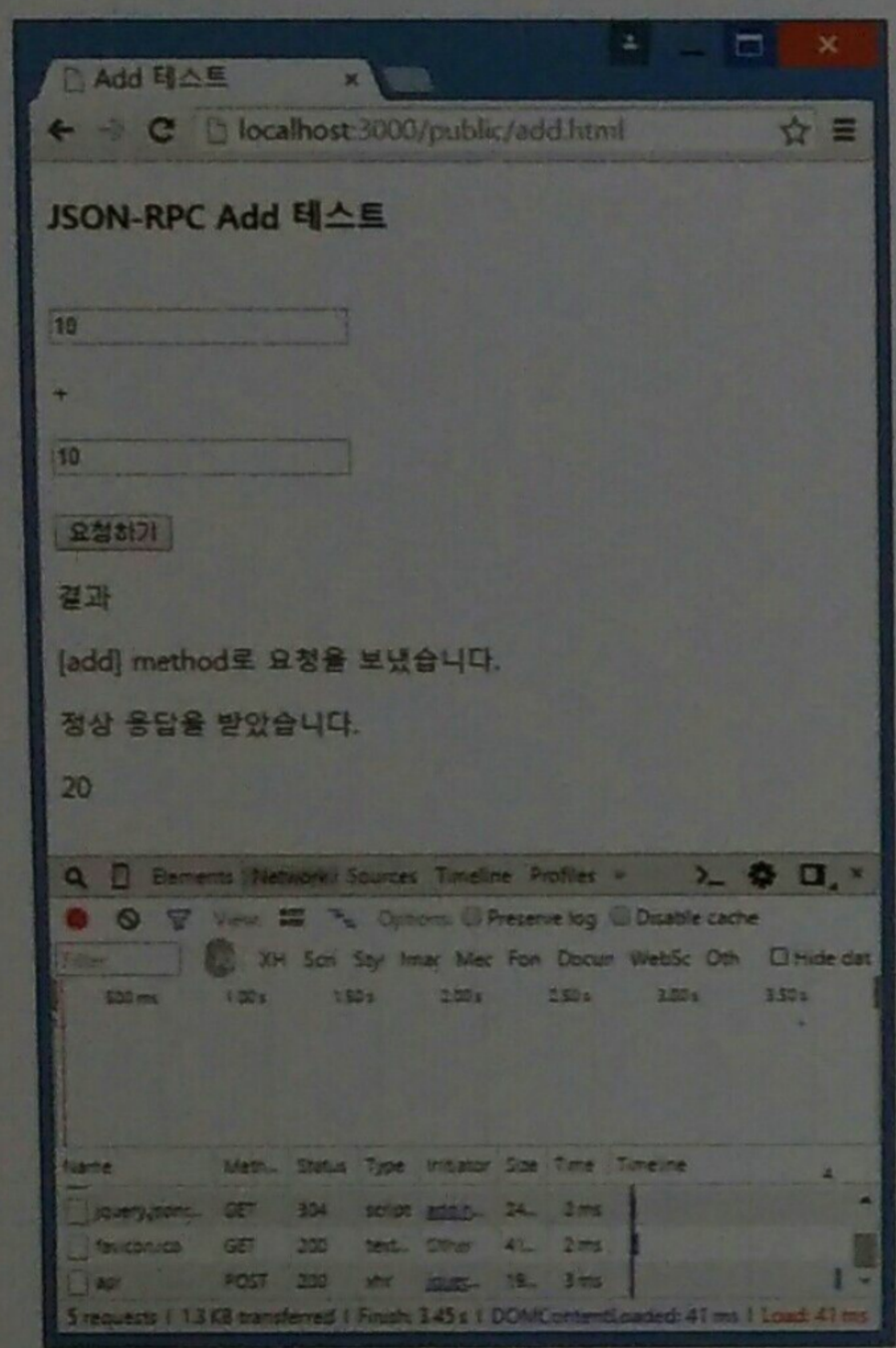
두 개의 입력 상자에 id 속성 aInput과 bInput을 각각 지정합니다. [요청하기] 버튼의 id 속성은 앞에서 만든 것과 똑같이 requestButton으로 지정합니다. 버튼을 클릭했을 때 호출되는 함수는 앞에서 만든 코드와 같지만 사용자가 입력 상자에 입력한 두 개의 값을 전달받은 후 서버에 정의된 add 함수를 호출하도록 코드를 변경합니다.

참조 파일 | JsonRpcExample>/public/add.html

```
……,

$("#requestButton").click(function() {
  var aStr = $("#aInput").val();
  var bStr = $("#bInput").val();
  var a = parseInt(aStr);
  var b = parseInt(bStr);
  var method = 'add';

  $.jsonRPC.request(method, {
    id: 1002,
    params: [a, b],
    success: function(data) {
      println('정상 응답을 받았습니다.');
      console.dir(data);
      println(data.result);
    },
    error: function(data) {
      println('오류 응답을 받았습니다.');
      console.dir(data);
      println(data.error.code + ', ' + data.error.message);
    }
  });

  println('[' + method + '] method로 요청을 보냈습니다.');
});

……,
```

입력 상자에서 두 개의 입력 값을 가져와 변수에 넣으면 문자열이 됩니다. 문자열에 + 연산자를 사용하면 더하기 연산이 아니라 두 개의 문자열을 붙여 주는 연산을 합니다. 따라서 서버에 전달하기 전에 parseInt( ) 메소드를 호출하여 정수로 바꾸어 줍니다. method 변수의 값을 add로 변경한 후 서버에 응답을 요청합니다.

이제 모든 단계가 끝났습니다. app.js 파일을 실행하고 웹 브라우저에서 웹 문서를 띄운 후 [요청하기] 버튼을 누르면 더하기 연산의 결과를 전달받아 화면에 '정상 응답을 받았습니다.'라는 메시지로 보여 줍니다.



▲ 서버의 add 함수를 호출한 후 응답을 받아 결과를 표시한 화면

곱하기 연산도 더하기 연산과 똑같은 과정을 거치는데 서버에 함수를 만들고 클라이언트에서 호출하는 방식으로 간단하게 만들 수 있습니다.

이 세 단계를 반복해서 연습해 볼수록 새로운 기능을 서버에 추가하는 것이 쉬워질 것입니다. 이제 서버에 데이터를 전달하고 그 결과를 받아 웹 화면에 보여 주는 것이 좀 더 쉬워졌나요?

11-3 | 데이터베이스에서 사용자 리스트 조회하기

JSON-RPC로 호출하려고 서버에 등록한 핸들러 함수에서도 데이터베이스에 접근할 수 있습니다. 이 핸들러 함수는 데이터를 조회한 후 응답 데이터를 보내 주도록 만들었습니다. 따라서 상품 리스트를 조회하거나 상품 정보를 등록하는 등의 기능을 아주 쉽게 서버에 추가할 수 있습니다. 그러면 사용자 정보를 저장하고 있는 데이터베이스 컬렉션에서 사용자 리스트를 조회한 후 클라이언트로 전달하는 함수를 만들어 보겠습니다. 서버 쪽에 핸들러 함수를 추가하고 클라이언트에서 호출하는 과정은 앞에서 배운 내용과 같습니다. 먼저 [handlers] 폴더에 listuser.js 파일을 만들고 다음 코드를 입력합니다.

참조 파일 | JsonRpcExample>/handlers/listuser.js

```
// 사용자 리스트 조회 함수
var listuser = function(params, callback) {
  console.log('JSON-RPC listuser 호출됨.');
  console.dir(params);

  // 데이터베이스 객체 참조
  var database = global.database;
  if (database) {
    console.log('database 객체 참조됨.');
  } else {
    console.log('database 객체 불가함.');
    callback({
      code : 410,
      message : 'database 객체 불가함.'
    }, null);
    return;
  }

  if (database.db) {
    // 1. 모든 사용자 검색
    database.UserModel.findAll(function(err, results) {

……
```

사용자 리스트를 조회하기 위해 만든 listuser 함수에서는 database 객체를 참조해야 데이터베이스에 접근할 수 있습니다. database 객체는 서버가 실행될 때 global 전역 변수에 속성으로 추가되었으므로 global.database 코드를 사용해 참조할 수 있습니다. 만약 database 객체를 참조할 수 없다면 클라이

언트에 오류 응답을 보냅니다. database 객체에는 사용자 정보를 담고 있는 모델 객체가 UserModel 속성으로 추가되어 있습니다. 따라서 모델 객체의 findAll( ) 메소드를 호출하여 사용자를 조회합니다. 조회 과정에 오류가 없다면 사용자 id와 name을 속성으로 가지는 객체를 만들어 배열에 추가한 후 배열 객체를 응답으로 보냅니다.

참조 파일 | JsonRpcExample>/handlers/listuser.js

```
.....

if (results) {
  console.log('결과물 문서 데이터의 개수 : %d', results.length);

  var output = [ ];
  for (var i = 0; i < results.length; i++) {
    var curId = results[i]._doc.id;
    var curName = results[i]._doc.name;
    output.push({id : curId, name : curName});
  }

  console.dir(output);
  callback(null, output);

.....
```

응답을 보내기 위해 사용하는 callback( ) 함수를 호출할 때는 결과물로 만든 배열 객체를 두 번째 파라미터로 전달합니다. 이렇게 만든 핸들러 모듈 파일을 handler_info.js 파일에 등록합니다.

참조 파일 | JsonRpcExample>/handlers/handler_info.js

```
.....

var handler_info = [

.....

    {file : './listuser', method : 'listuser'}

.....

];

.....
```

이제 클라이언트 웹 문서를 만들고 서버 쪽에 등록되어 있는 listuser 함수를 실행하도록 요청해야 합니다. 먼저 [public] 폴더에 listuser.html 파일을 만들고 JSON-RPC 요청에 필요한 코드를 입력합니다. 사실 클라이언트 웹 문서에 입력하는 자바스크립트 코드 중에서 주의해야 하는 부분은 (1) 요청에 사용하는 이름과 (2) 파라미터 그리고 (3) 정상 응답을 받았을 때 처리하는 코드 부분입니다. 따라서 이 부분을 별도의 함수로 분리하면 소스 코드를 입력하거나 수정할 때 해당 코드만 신경 쓰면 되니 웹 문서를 좀 더 쉽게 만들 수 있습니다.

먼저 버튼을 클릭했을 때 호출하는 코드 부분을 만들어 보겠습니다.

참조 파일 | JsonRpcExample>/public/listuser.html

```
.....

  // 버튼을 클릭했을 때 서버에 요청
  $("#requestButton").click(function() {
    var method = 'listuser';
    var id = 1004;
    var params = [ ];

    sendRequest(method, id, params);

    println('[' + method + '] method로 요청을 보냈습니다.');
  });

.....
```

method, id, params 값만 신경 쓰면 되기 때문에 앞에서 만든 것보다 훨씬 간단한 코드가 되었습니다. sendRequest( ) 함수에는 요청에 필요한 기본 코드를 추가하고, 이 함수 안에서 정상 응답과 오류 응답을 처리할 수 있도록 만듭니다. 그리고 정상 응답을 받았을 때와 오류 응답을 받았을 때 실행되는 코드는 각각 processResponse( )와 processError( ) 함수에서 처리합니다.

참조 파일 | JsonRpcExample>/public/listuser.html

```
.....

  // 요청에 필요한 기본 함수
  function sendRequest(method, id, params) {
    $.jsonRPC.request(method, {
        id : id,
        params : params,
        success : function(data) {
          println('정상 응답을 받았습니다.');
          console.dir(data);
          processResponse(data);
        },
        error :  function(data) {
          println('오류 응답을 받았습니다.');
          console.dir(data);
          processError(data);
        }
    });
  }

.....
```

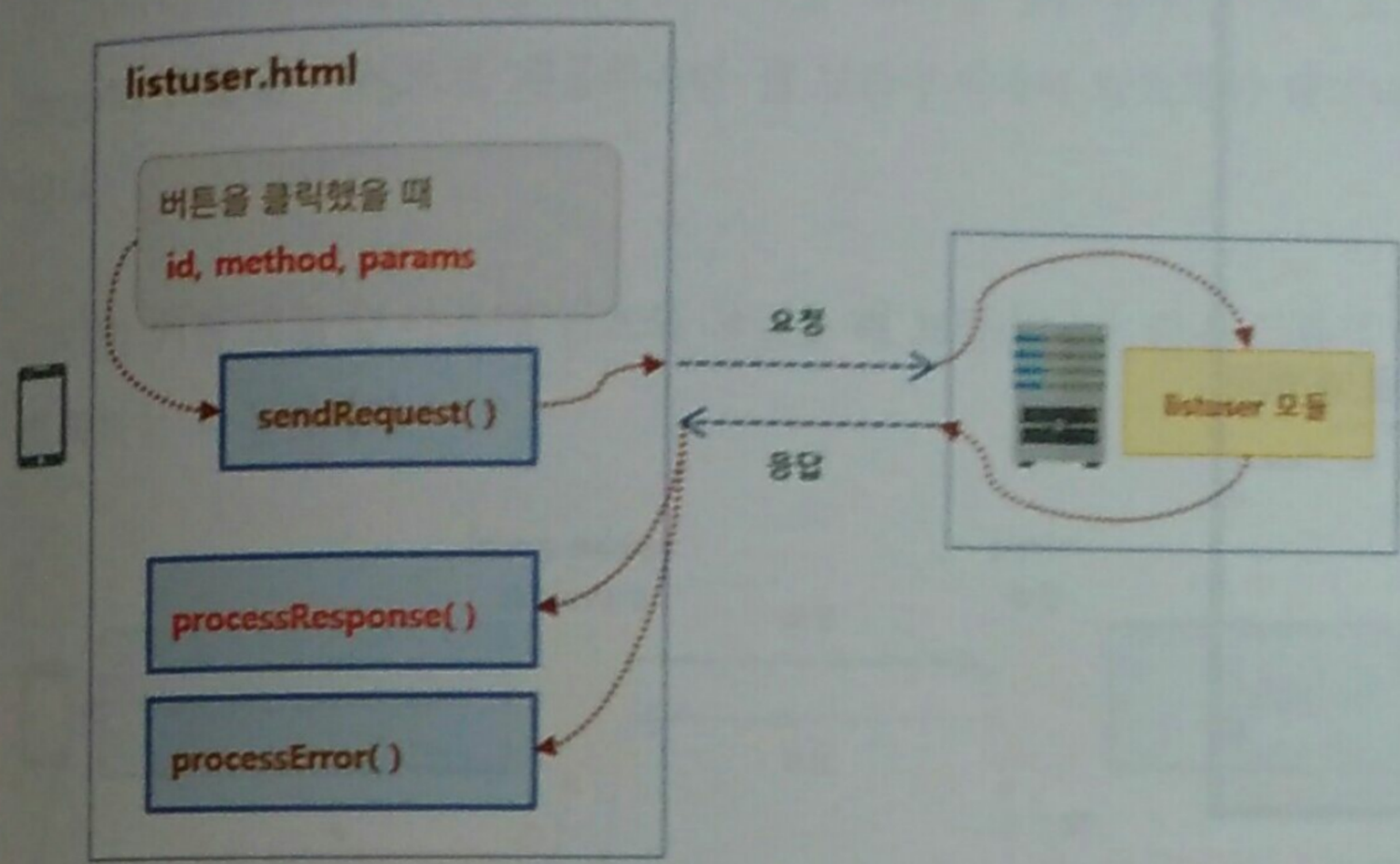
정상 응답을 받았을 때 호출되는 processResponse( ) 메소드의 코드는 다음과 같습니다.

참조 파일 | JsonRpcExample>/public/listuser.html

```
.....

  // 성공 응답을 받은 경우 호출되는 함수
  function processResponse(data) {
    if (Array.isArray(data.result)) {
      println('사용자 수 : ' + data.result.length);
      data.result.forEach(function(item, index) {
        println('#' + index + ' : ' + item.id + ', ' + item.name);
      });
    } else {
      println('결과 데이터가 배열 타입이 아닙니다.');
    }
  }

.....
```

응답으로 받은 결과 객체는 배열이므로 forEach문을 사용하여 각 아이템의 id와 name 값을 웹 화면에 표시합니다.

사용자 정보를 확인하려고 입력한 코드의 구조를 정리하면 다음과 같습니다.

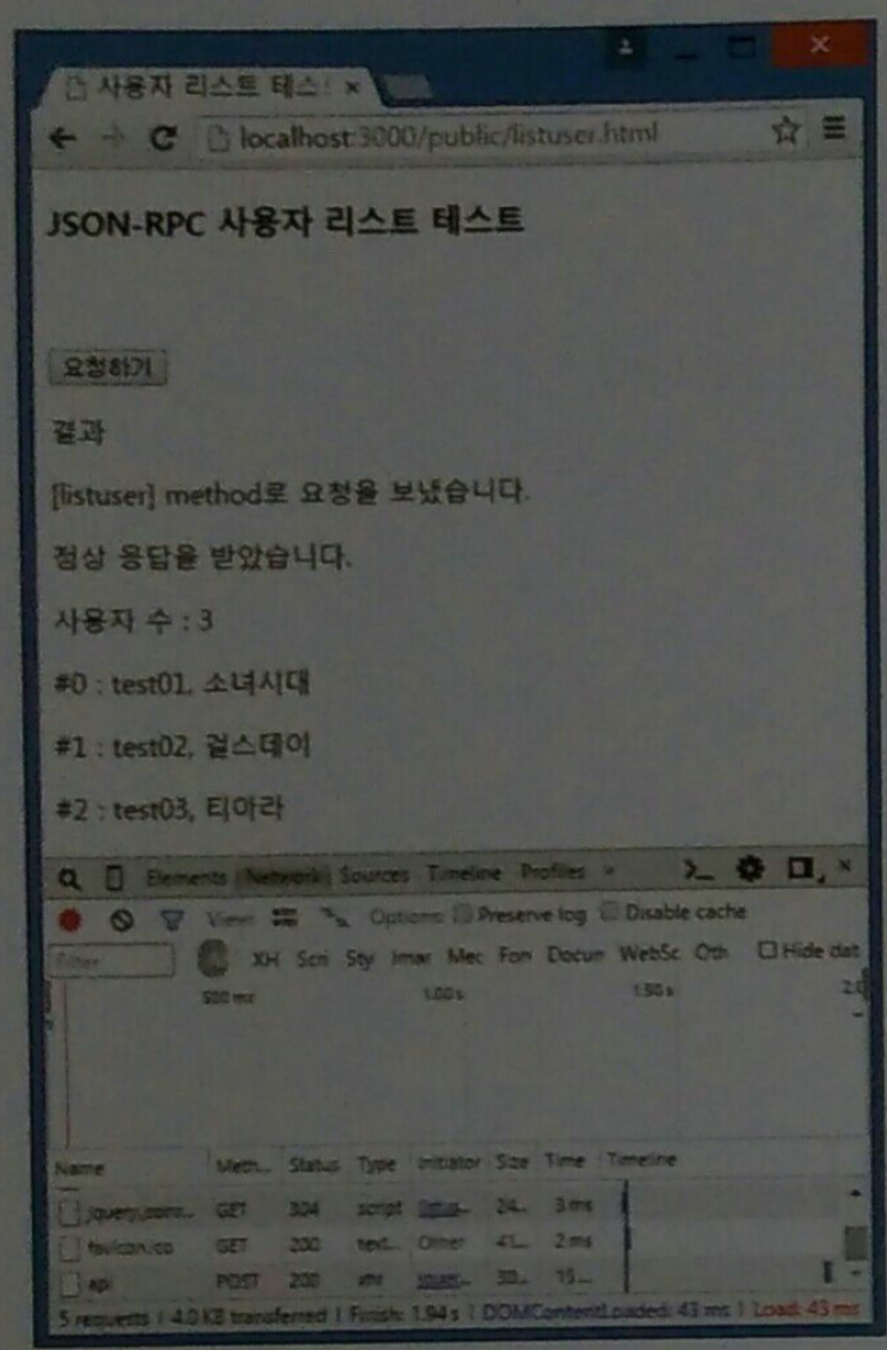


▲서버의 함수를 호출하고 응답을 받아 처리하는 코드 구조

다른 함수를 사용하고 싶다면 클라이언트 요청을 담당하는 웹 문서의 자바스크립트 코드에서 두 가지만 수정하면 됩니다. 첫 번째는 요청을 보낼 때 사용하는 id, method, params입니다. id 변수에는 각 기능별로 고유한 숫자를 부여할 수 있습니다. method는 서버에 만든 함수를 호출할 때 사용하는 이름이며, params는 서버에 전달할 데이터를 배열 형태로 만든 것입니다. 이 부분만 수정하면 서버에 만든 어떤 함수든 호출할 수 있습니다. 두 번째는 성공 응답을 받았을 때 처리하는 processResponse( ) 메소드의 코드입니다. 서버로부터 응답을 받았을 때 화면에 표시할 것인지 또는 응답 데이터를 사용해 다른 작업을 수행할 것인지를 결정한 후 코드로 만듭니다.

이제 app.js 파일을 실행한 후 웹 브라우저에서 listuser.html 파일을 엽니다. [요청하기] 버튼을 누르면 화면에 사용자 리스트가 표시됩니다.

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



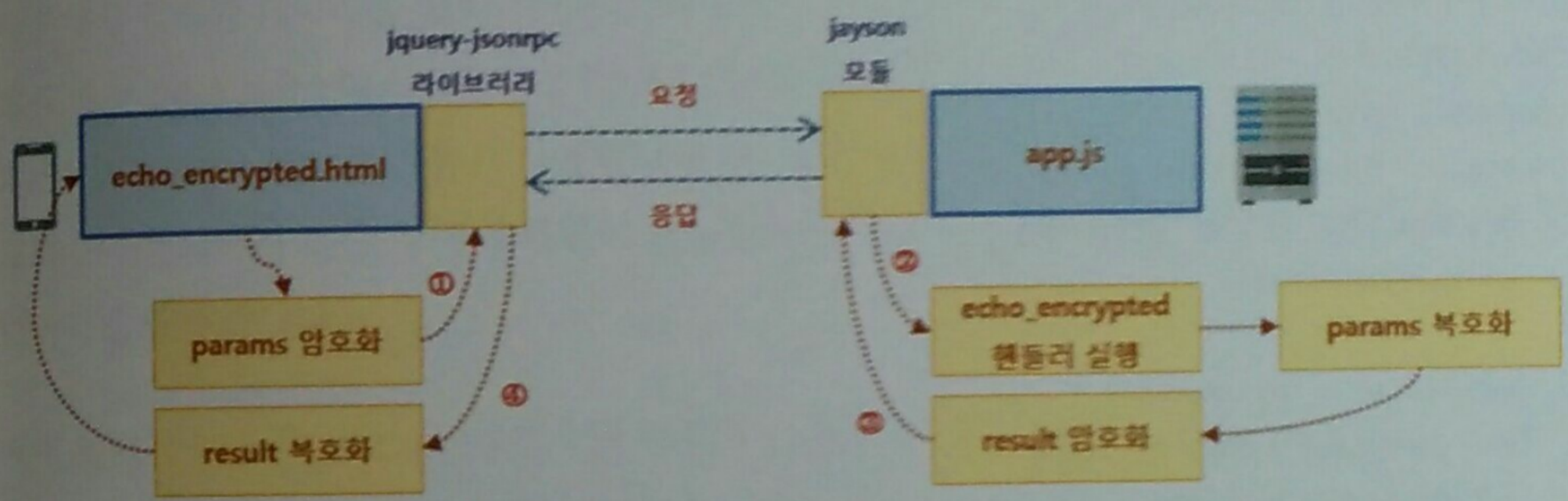
▲ 데이터베이스의 사용자 리스트를 조회한 화면

JSON-RPC를 사용해 서버의 데이터베이스에 저장되어 있는 데이터를 조회하는 방법을 알아보았습니다. 일반적인 웹 방식으로 웹 서버에 요청할 때보다 더 쉽고 단순하게 기능을 만들고 요청할 수 있다는 것이 JSON-RPC의 장점임을 이제 확실히 알 수 있을 것입니다.

11-4 | 데이터 부분을 암호화하기

RPC 방식을 사용할 때는 주로 데이터를 주고받기 때문에 데이터만 암호화하는 경우가 많습니다. 따라서 데이터를 암호화하려면 여러분이 직접 코드를 입력해야 합니다. 노드에서는 데이터를 암호화하는 crypto 모듈을 기본으로 제공하지만 웹 브라우저에서 암호화한 데이터와 호환되지 않는 경우도 있습니다.

그러면 외장 모듈을 사용해 암호화 작업을 해 보겠습니다. 다음 그림은 암호화하여 전송한 데이터를 복호화하는 과정입니다.



▲ JSON-RPC의 데이터를 암호화하여 송수신하는 과정

브라우저에서는 params 배열에 넣어 전달하는 데이터를 암호화합니다. 이렇게 암호화한 데이터는 서버 쪽에서 복호화한 후 처리합니다. 서버 쪽에서 나온 결과도 다시 암호화하여 클라이언트로 보내면 클라이언트에서 이 데이터를 복호화하여 화면에 보여 줄 수 있습니다.

그럼 암호화해서 송수신하는 과정대로 코드를 만들어 보겠습니다. [public] 폴더 안에 있는 echo.html 파일을 복사하여 echo_encrypted.html 파일을 만듭니다. 그런 다음 JSON-RPC로 호출하는 부분을 수정합니다.

참조 파일 | JsonRpcExample>/public/echo_encrypted.html

```
……,

  <script src="jquery-2.1.4.min.js"></script>
  <script src="jquery.jsonrpc.js"></script>
  <script src="cryptojs/aes.js"></script>

……,
```

```
$("#requestButton").click(function() {
  var message = $("#messageInput").val();

  // 암호화 테스트
  var secret = 'my secret';
  var encrypted = '' + CryptoJS.AES.encrypt(message, secret);
  console.log(encrypted);

  // 복호화 테스트
  var decrypted = CryptoJS.AES.decrypt(encrypted, secret).toString(CryptoJS.enc.Utf8);
  console.log(decrypted);
  var method = 'echo_encrypted';

  $.jsonRPC.request(method, {
    id : 1001,
    params : [encrypted],
    success : function(data) {
      println('정상 응답을 받았습니다.');
      console.dir(data);

      var secret = 'my secret';
      var encrypted = data.result[0];
      var decrypted = CryptoJS.AES.decrypt(encrypted, secret).toString(CryptoJS.enc.Utf8);

      console.log(decrypted);
      println(decrypted);
    },
    error :  function(data) {
      println('오류 응답을 받았습니다.');
      console.dir(data);
      println(data.error.message);
    }
  });

  println('[' + method + '] method로 요청을 보냈습니다.');

  ......
```

웹 브라우저의 자바스크립트에서 사용할 수 있는 암호화 라이브러리는 crypto-js입니다. 다음 사이트에서 다운로드할 수 있습니다.

▶ https://code.google.com/p/crypto-js/

다운로드한 파일 중에서 aes.js 파일을 [cryptojs] 폴더 안에 넣어 둔 후 링크하여 사용합니다. aes.js 파일에는 AES 표준 알고리즘으로 암호화하거나 복호화할 수 있는 함수가 들어 있습니다.

정박사의 한마디 암호화에 대한 내용이나 용어를 잘 모르나요?

보통 암호화라고 하면 어렵게 느끼는 분들이 많습니다. 또한 암호화, 복호화 등의 용어 자체를 모르는 분들도 많습니다. 하지만 여기에서는 단순히 클라이언트와 서버 사이에 보내고 받는 데이터를 읽기 힘든 코드로 바꾼다는 정도로 이해하면 됩니다. 데이터를 읽기 힘든 코드로 바꾸는 과정이 암호화, 다시 읽을 수 있는 코드로 바꾸는 과정이 복호화입니다. 그리고 읽기 힘든 코드로 만드는 방법은 여러 가지가 있는데 그중에 가장 많이 사용되는 것이 AES입니다.

사용자가 입력한 데이터를 암호화하려고 [요청하기] 버튼을 누르면 encrypt( ) 메소드를 사용하며, 복호화할 때는 decrypt( ) 메소드를 사용합니다. 암호화할 때는 노출되지 않는 키(Key) 값을 사용하는데 여기에서는 secret 변수로 만들어 사용합니다. 두 가지 메소드를 모두 사용하여 정상적으로 암호화와 복호화가 진행되는지 콘솔 창에 출력하여 확인합니다. 그다음 JSON-RPC로 요청하는 부분에 암호화된 데이터를 넣어 줍니다.

서버 쪽에는 echo_encrypted 함수를 만들어 등록할 것이므로 method 변수에 들어가는 값을 echo에서 echo_encrypted로 수정합니다.

이제 [handlers] 폴더 안에 있는 echo.js 파일을 복사하여 echo_encrypted.js 파일로 만든 후 코드를 수정합니다. 노드에서도 crypto-js 외장 모듈을 사용합니다.

명령 프롬프트를 열고 다음 명령을 입력하여 모듈을 설치합니다.

```
% npm install crypto-js --save
```

서버 쪽에 만드는 echo_encrypted.js 파일은 다음 코드처럼 수정합니다.

참조 파일 | JsonRpcExample>/handlers/echo_encrypted.js

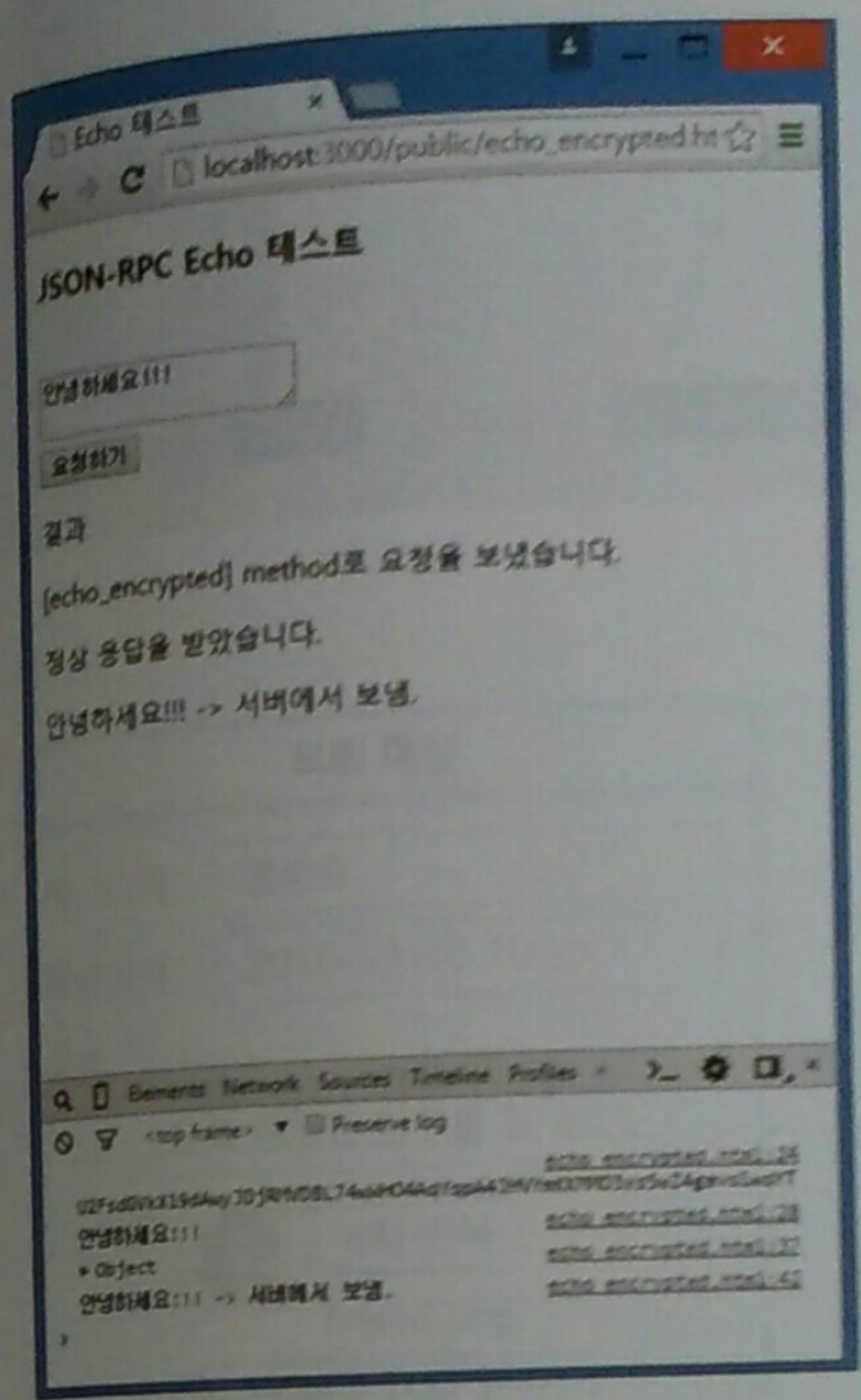
```
var crypto = require('crypto-js');
var echo = function(params, callback) {
  console.log('JSON-RPC echo_encrypted 호출됨.');
  console.dir(params);
```

```
try {
  // 복호화 테스트
  var encrypted = params[0];
  var secret = 'my secret';
  var decrypted = crypto.AES.decrypt(encrypted, secret).toString(crypto.enc.Utf8);

  console.log('복호화된 데이터 : ' + decrypted);

  // 암호화 테스트
  var encrypted = '' + crypto.AES.encrypt(decrypted + ' -> 서버에서 보냄.', secret);

  console.log(encrypted);
  params[0] = encrypted;
 } catch(err) {
  console.dir(err);
  console.log(err.stack);
 }

 callback(null, params);
};

module.exports = echo;
```

암호화하거나 복호화하는 코드는 클라이언트에서 사용한 것과 거의 같습니다. 클라이언트에서 보내 온 파라미터의 데이터를 복호화한 후 콘솔 창에 출력하고 '-> 서버에서 보냄'이라는 글자를 덧붙인 후 암호화하여 클라이언트에 응답으로 보냅니다. crypto 모듈 안에는 AES 객체가 들어 있고, 그 안에 encrypt( ) 와 decrypt( ) 메소드가 들어 있습니다. 이 메소드를 호출하면 암화와 복호화가 진행됩니다.

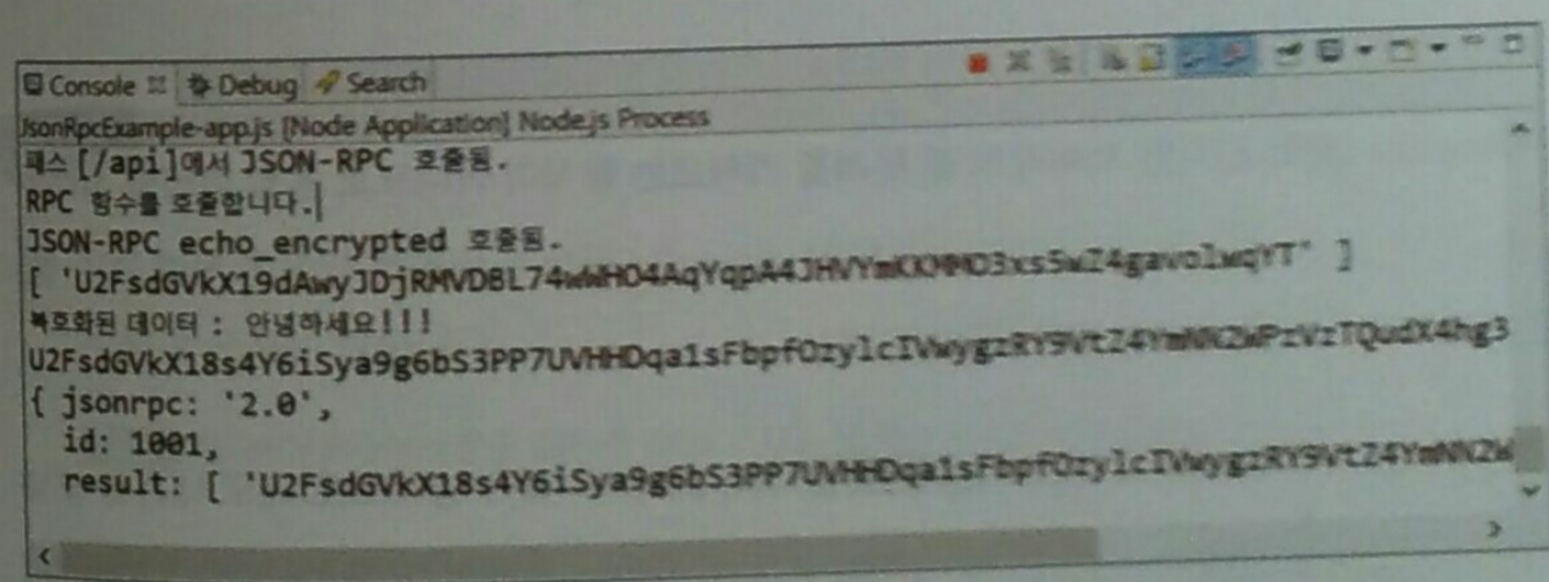
app.js 파일을 실행하고 웹 브라우저에서 echo_encrypted.html 파일을 조회한 후 글자를 입력하고 [요청하기] 버튼을 누릅니다.

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



▲웹 브라우저에서 echo_encrypted.html 파일을 열고 [요청하기] 버튼을 누른 결과

개발자 도구의 콘솔 창을 열어 보면 서버로 보내고 받는 과정에서 암호화되거나 복호화된 값을 확인할 수 있습니다. 서버 쪽 콘솔 창에도 다음과 같이 출력됩니다.



▲echo_encrypted 핸들러가 실행되었을 때 서버 쪽 콘솔 창에 출력된 결과

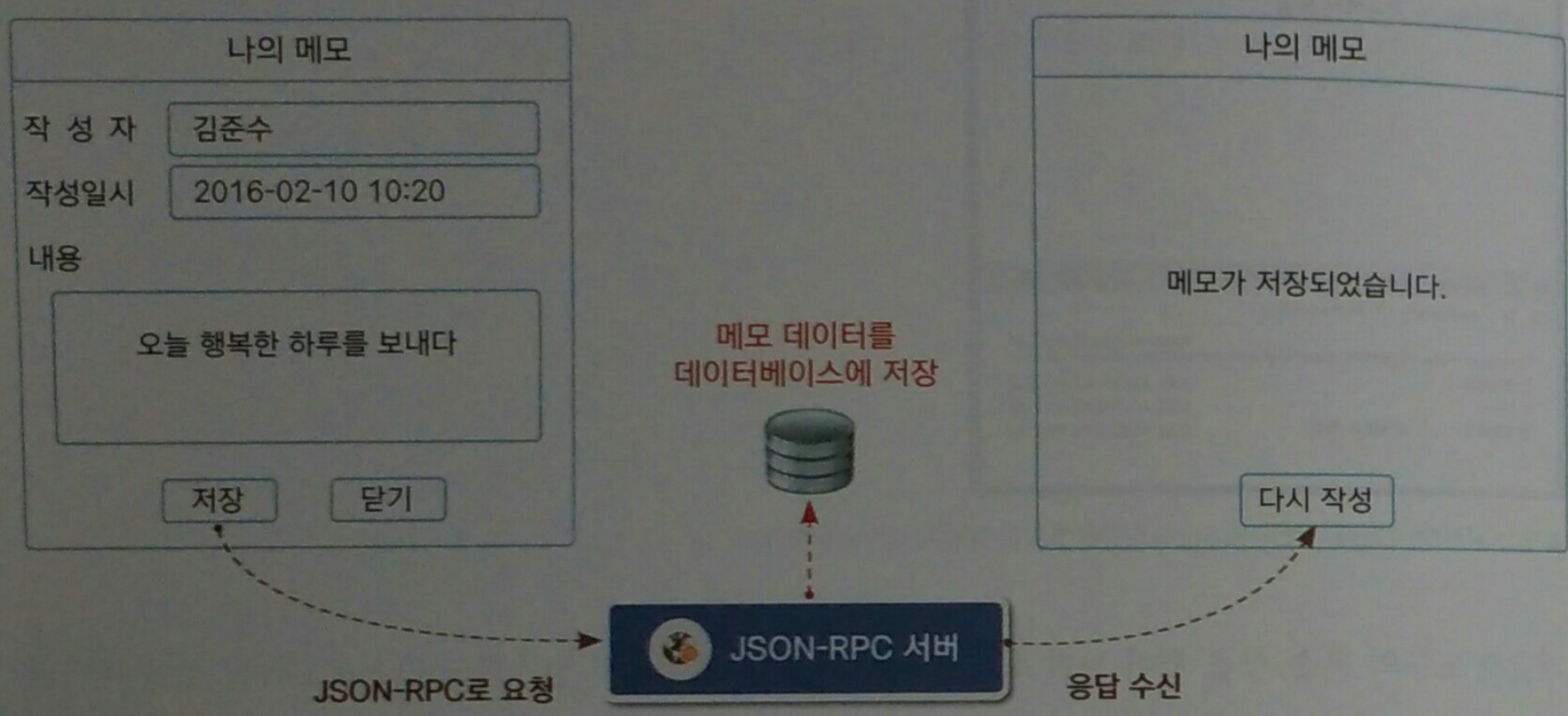
암호화된 데이터는 그 뜻을 이해하기 어려운 알파벳과 숫자의 나열로 만들어집니다. 이렇게 만든 코드는 복호화되면 암호화되기 전의 글자로 복원됩니다.

지금까지 클라이언트에서 암호화하여 보낸 데이터를 핸들러에서 복호화하는 과정을 살펴보았습니다. 만약 모든 클라이언트에서 데이터를 암호화한 후 서버에 요청한다면 각각의 핸들러에서 복호화하는 것이 번거로울 수 있습니다. 이 경우에는 핸들러가 처리하기 전에 미들웨어를 만들어 복호화를 미리 진행하도록 만들 수도 있습니다. 이 작업은 여러분이 직접 해 보기 바랍니다.

Do it!
도전 문제

01. 08장에서 만든 Mission-09의 메모 저장 기능을 JSON-RPC 방식으로 수정해 보세요.

프로젝트 소스 Mission15 난이도 초급 ★☆☆☆



Hint!

1 작성 화면과 응답 화면의 구성은 Mission-09와 같지만 서버에서 웹 문서를 가져오는 방식이 아니므로 응답 화면을 서버에서 받아 오지는 않습니다.

주의! 웹 브라우저에서 응답 화면을 미리 가지고 있지만 보이지 않게 만들어 둡니다.

2 작성 화면에서 [저장] 버튼을 누르면 JSON-RPC 서버로 메모 내용을 보냅니다. JSON-RPC 방식으로 메모 내용을 보낼 때는 데이터만 보냅니다.

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

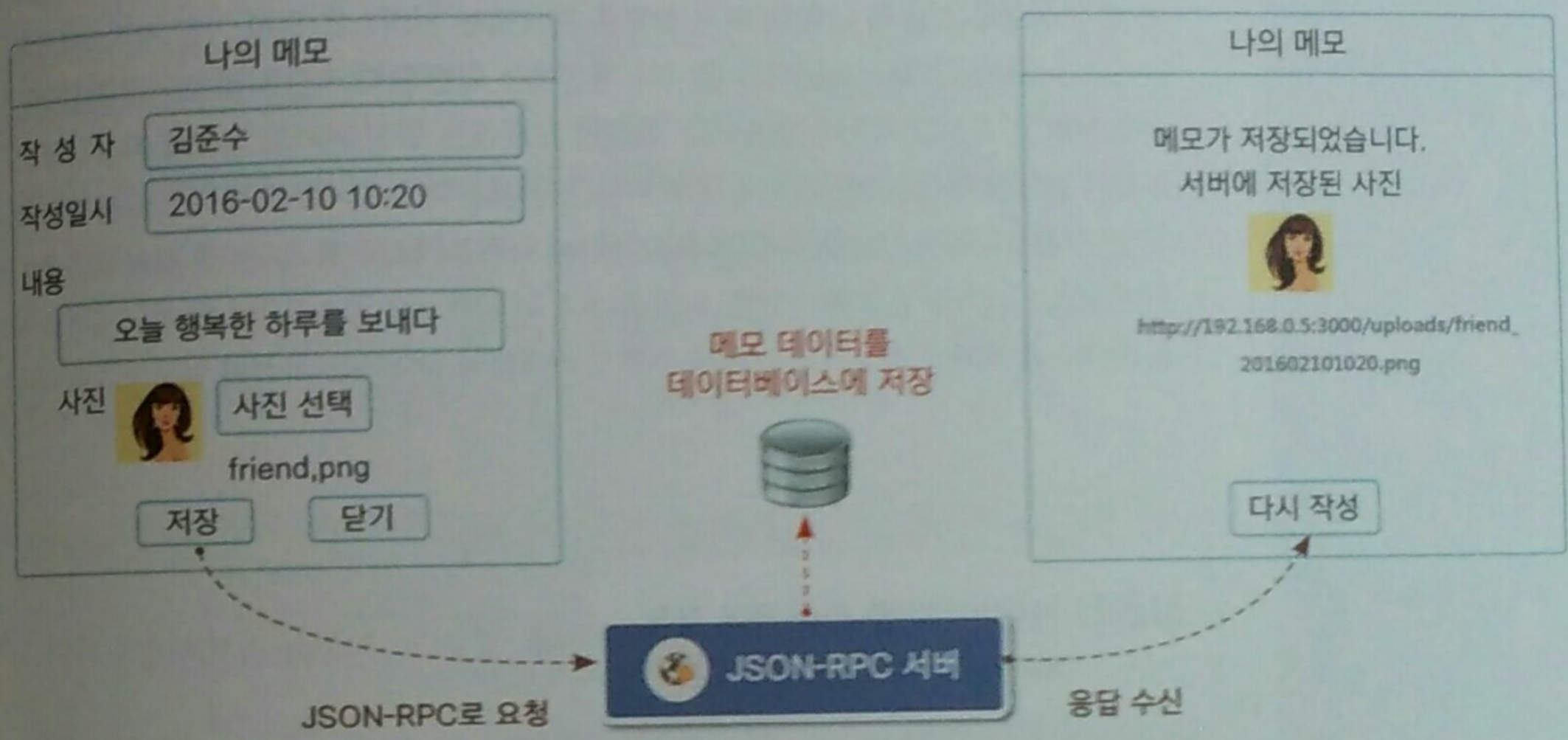
4 클라이언트에서 응답 메시지를 받으면 응답 화면에 메시지를 보여 줍니다.

MEMO

Do it!
도전 문제

02. 08장에서 만든 Mission-10의 메모와 사진 정보 저장 기능을 JSON-RPC 방식으로 수정해 보세요.

프로젝트 소스 Mission16 난이도 중급 ★★☆☆



Hint!

1 작성 화면과 응답 화면의 구성은 Mission-10과 같지만 서버에서 웹 문서를 가져오는 방식이 아니므로 응답 화면을 서버에서 받아 오지는 않습니다.

주의! 웹 브라우저에서 응답 화면을 미리 가지고 있지만 보이지 않게 만들어 둡니다.

2 작성 화면에서 [저장] 버튼을 누르면 JSON-RPC 서버로 메모 내용을 보내고 사진 파일도 업로드합니다. JSON-RPC 방식으로 메모 내용을 보낼 때는 데이터만 보내기 때문에 사진 정보도 포함합니다.

3 데이터베이스에 메모 내용과 사진 정보를 저장한 후 정상적으로 저장되었다는 응답 메시지를 클라이언트로 보냅니다.

4 클라이언트에서 응답 메시지를 받으면 응답 화면에 메시지를 보여 줍니다.

MEMO

12 위치 기반 서비스 서버 만들기

스마트폰 사용이 일상화된 요즘 사용자 자신의 위치를 기준으로 주변의 커피숍을 찾거나 목적지를 설정해서 찾아가는 사람들이 많아졌습니다. 휴대용 단말에서 확인한 사용자의 위치 정보를 사용해 다양한 부가 정보를 제공하는 서비스를 위치 기반 서비스(LBS, Location Based Service)라고 합니다. 예전에는 이런 위치 기반 서비스 서버를 만들기 위해 복잡한 과정을 거쳐야 했습니다. 하지만 요즘에는 특정 위치의 정보를 저장하고 조회하는 방법을 데이터베이스에서 대부분 제공합니다. 이 책의 데이터베이스 부분에서 배운 몽고디비도 이런 기능을 제공합니다. 따라서 익스프레스와 몽고디비를 활용하면 위치 기반 서비스 서버를 손쉽게 구현할 수 있습니다. 이 장에서는 익스프레스와 몽고디비를 사용해 커피숍 위치를 저장하고 사용자 주변의 커피숍을 찾아내는 과정을 살펴보겠습니다.

질문으로 시작하기

질문1. 커피숍 위치를 서버 쪽에 저장하려면 어떻게 해야 하나요?

➡ 12-1. 커피숍 위치 저장하기

질문2. 내가 있는 곳에서 가장 가까운 커피숍을 찾고 싶나요?

➡ 12-2. 가장 가까운 커피숍 찾기

질문3. 영역을 정해서 그 안에 있는 커피숍을 찾고 싶나요?

➡ 12-3. 영역 안의 커피숍 찾기

질문4. 일정 반경 안에 있는 커피숍을 찾을 수도 있나요?

➡ 12-4. 반경 안의 커피숍 찾기

질문5. 커피숍의 위치를 지도 위에 나타내고 싶나요?

➡ 12-5. 지도에 커피숍의 위치 표시하기

12-1 | 커피숍 위치 저장하기

사용자 주변에 있는 커피숍을 찾고 싶다면 먼저 커피숍 위치를 어떻게 서버의 데이터베이스에 저장하고 조회하는지 알아야 합니다. 여기에서는 커피숍의 위치 정보를 서버에 보내면 서버의 데이터베이스에 저장하고 저장된 리스트를 조회하는 방법을 먼저 알아봅니다.

커피숍의 위치 정보 다루기

스마트폰이 대중화되면서 가장 흔하게 사용하는 것이 특정 위치를 찾는 위치 기반 서비스입니다. 이 서비스는 사용자가 있는 곳에서 가장 가까운 커피숍을 찾아서 보여 주거나, 사용자의 현재 위치를 지도에 표시하면서 주변에 분포되어 있는 커피숍을 찾아 보여 주는 기능과 함께 위치 정보도 제공합니다. 이러한 기능을 직접 만들고 싶다면 다음과 같은 과정을 거쳐야 합니다.

기능	설명
커피숍의 위치 정보 확인	특정 지역의 커피숍 이름이나 전화번호, 위치 등의 정보를 확인해야 합니다. 그중에 커피숍의 위치 정보는 경위도 좌표로 알고 있어야 합니다.
커피숍의 위치 정보 저장	위치 정보를 포함한 커피숍 정보를 서버에 저장해야 합니다. 서버의 데이터베이스에 저장합니다.
커피숍을 위치로 조회	가까운 커피숍을 경위도 좌표로 조회해야 합니다.

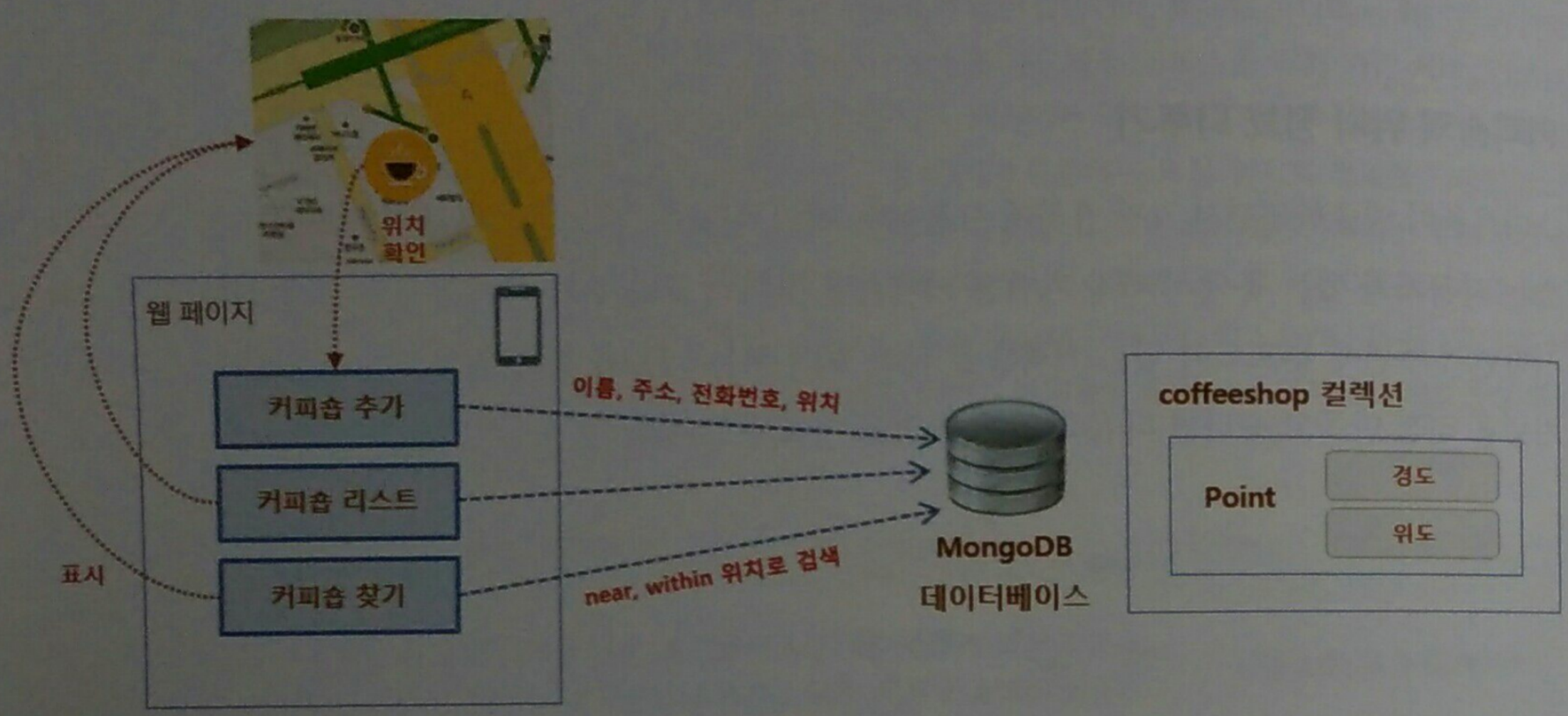
[표] 커피숍 찾기 기능을 만드는 과정

첫 번째 단계는 커피숍의 위치 정보를 확인하는 것입니다. 커피숍 위치가 경위도 좌표로 되어 있어야 하므로 모바일 단말이나 기타 장비를 사용하여 현장의 위치를 알아내거나 이미 조사한 위치 정보를 확인하는 등의 방법을 사용합니다. 이 책에서는 지도를 띄우고 간단하게 몇 개의 커피숍 위치만을 가상으로 확인합니다.

두 번째 단계는 커피숍 위치를 서버의 데이터베이스에 저장하는 것입니다. 그런데 위치 정보는 사람의 이름과 같은 문자열 데이터를 저장하는 것과 다르게 경도와 위도라는 두 가지 정보를 함께 저장합니다. 이 때문에 위치 정보를 저장하거나 조회할 때 일반적인 문자열 데이터와는 다르게 처리해야 합니다. 만약 일반적인 데이터처럼 처리한다면 데이터가 많아질수록 조회 속도가 많이 느려집니다. 그래서 보통은 공간 인덱싱(Spatial Indexing) 방법으로 경도와 위도 좌표의 인덱스(Index)를 별도로 만들어 조회 속도를 높입니다. 이런 방법을 몽고디비에서는 GeoSpatial Indexing이라고 부릅니다.

세 번째 단계는 커피숍 위치를 조회하는 것입니다. 가장 대표적인 조회 방법으로 (1) 사용자가 있는 곳에서 가장 가까운 커피숍 찾기, (2) 사용자가 보고 있는 지도 범위 안의 커피숍을 찾아서 보여 주기, (3) 사용자가 있는 곳에서 반경 몇 미터 안에 있는 커피숍을 찾아서 보여 주기를 들 수 있습니다. 몽고비디에서는 각각의 방법을 near, within, within(circle)으로 부릅니다.

다음은 몽고디비에 커피숍 위치 정보를 어떻게 저장하고 조회할 것인지를 보여 줍니다.



▲몽고디비에 커피숍 위치 정보를 저장하고 조회하는 방법

지도 서비스를 제공하는 몇몇 사이트들은 지도 위의 특정 지점을 클릭하면 경위도 좌표를 알려 줍니다. 이 방식을 사용해 지도에서 커피숍 위치 정보를 몇 개 정도 확인한 후 웹 서버로 보내어 데이터베이스에 저장합니다. 커피숍 정보는 이름, 주소, 전화번호, 위치로 구성되는데, 다음이나 네이버와 같은 포털 사이트에서도 손쉽게 확인할 수 있습니다. 커피숍 위치는 위치 데이터의 유형 중 Point 위치 데이터에 해당합니다. 즉, 점으로 표시할 수 있는 위치인데 커피숍 위치는 몽고디비에서 Point라는 유형으로 저장할 수 있습니다.

정박사의 한마디 위치 데이터의 유형은 다양해요

위치 데이터의 유형은 사용자의 현재 위치나 커피숍 위치처럼 특정한 지점을 나타내는 Point 위치 데이터, 도로와 같이 이어진 LineString 위치 데이터, 청담동이나 방배동처럼 지역을 나타내는 Polygon 위치 데이터 등이 있습니다. 도형의 점, 선, 면 등을 생각하면 쉽습니다.

커피숍 정보를 서버에서 받아 보여주는 기능은 단순히 모든 커피숍의 데이터를 요청하여 리스트로 보여 주는 경우와 찾으려는 위치를 검색해서 보여 주는 경우로 나눌 수 있습니다. 커피숍의 위치 정보를 사용해 찾는 경우는 다시 가장 가까운 커피숍 찾기, 범위로 찾기, 반경으로 찾기로 나뉩니다.

커피숍 스키마 만들기

PassportExample2를 복사하여 GeoExample이라는 새로운 프로젝트를 만든 후 필요한 기능을 하나씩 추가해 보겠습니다. 먼저 커피숍 정보를 저장할 **데이터베이스 스키마**를 만듭니다. 스키마 파일은 [database] 폴더 안에 만들어야 하므로 [database] 폴더 안에 coffeeshop_schema.js 파일을 만듭니다. 스키마 파일에서는 Schema 객체를 만든 후 module.exports에 할당하게 되는데 스키마를 생성하는 함수를 정의한 후 **createSchema 속성**에 할당합니다.

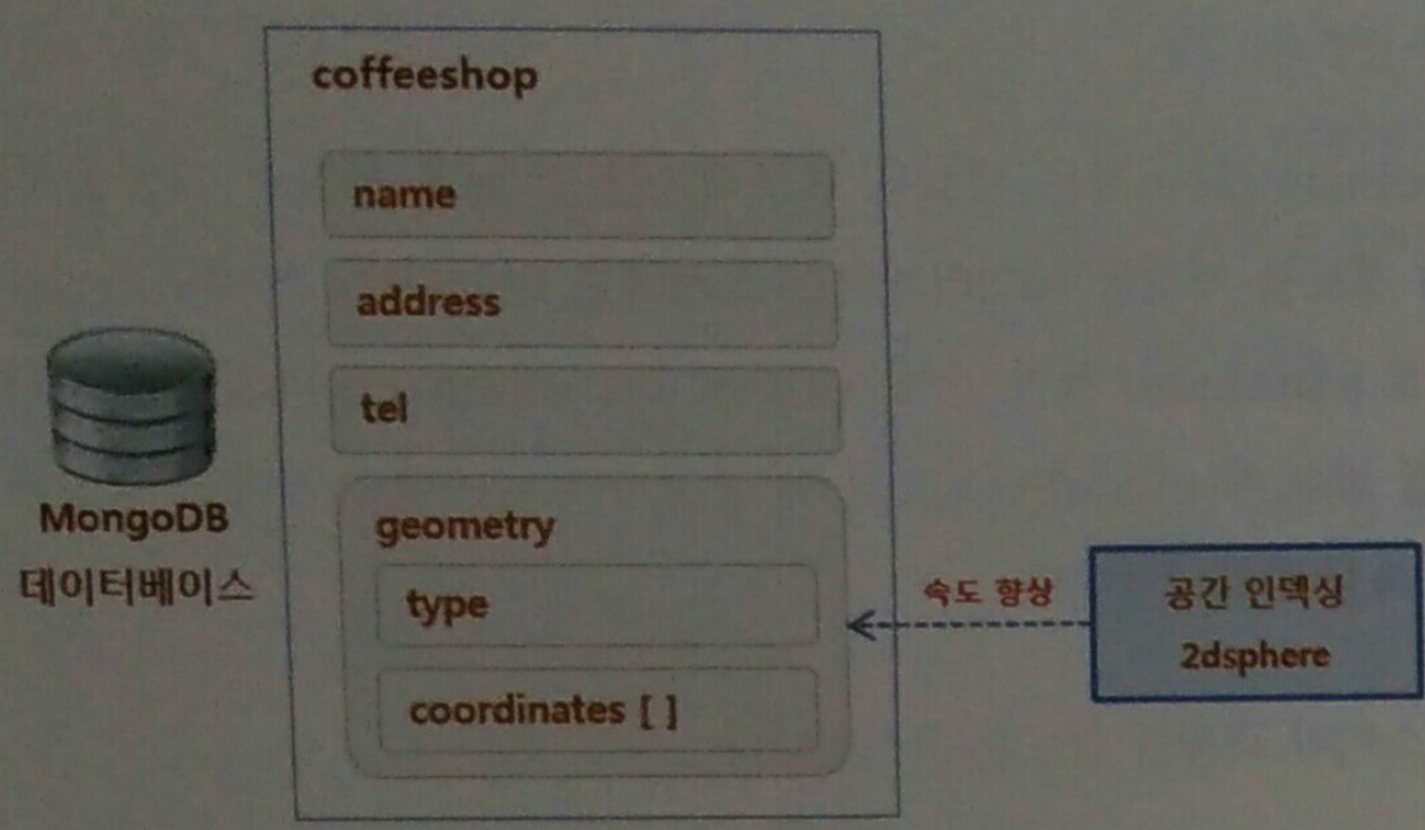
다음은 커피숍 정보를 저장하기 위해 스키마 객체를 만드는 코드입니다.

참조 파일 | GeoExample>/database/coffeeshop_schema.js

```
var Schema = { };

Schema.createSchema = function(mongoose) {

  // 스키마 정의
  var CoffeeShopSchema = mongoose.Schema({
    name : {type : String, index : 'hashed', 'default' : ''},
    address : {type : String, 'default' : ''},
    tel : {type : String, 'default' : ''},
    geometry : {
      'type' : {type : String, 'default' : "Point"},
      coordinates : [{type : "Number"}]
    },
    created_at : {type : Date, index : {unique : false}, 'default' : Date.now},
    updated_at : {type : Date, index : {unique : false}, 'default' : Date.now}
  });

.....
```

커피숍 정보를 저장하는 스키마에는 name, address, tel, geometry, created_at, updated_at 속성이 들어 있습니다. name 속성은 커피숍 이름을 저장하는데 인덱스를 만들도록 설정하였습니다. address와 tel 속성은 각각 주소와 전화번호를 저장합니다. geometry 속성은 경위도 좌표를 저장하는데 그 안에 다시 두 개의 속성을 가진 객체가 들어 있습니다. type 속성은 위치 정보의 유형을 구별하는 것으로 디폴트 값을 Point로 설정합니다. coordinates는 위치 좌표를 넣을 수 있는 배열로 정의되어 있으며, Number 타입의 좌표를 사용할 수 있습니다. 좌표를 사용할 때는 경도, 위도 순으로 들어가므로 하나의 점으로 표시되는 커피숍 위치를 표시하려면 경도 좌표 coordinates[0]과 위도 좌표 coordinates[1]을 사용합니다.

이렇게 만든 커피숍 스키마의 geometry 속성에 인덱스를 만들면 조회 속도가 빨라집니다.



▲ 커피숍 스키마의 속성과 공간 인덱싱

스키마 객체의 index( ) 메소드를 사용하면 인덱스를 만들 수 있는데 위치 좌표는 2dsphere 타입으로 지정합니다.

참조 파일 | GeoExample>/database/coffeeshop_schema.js

```
.....

CoffeeShopSchema.index({geometry : '2dsphere'});

.....
```

커피숍 정보를 추가하고 커피숍 리스트 조회하기

스키마를 만들었으니 데이터베이스에서 이 스키마로 검색할 때 사용할 static( ) 메소드를 추가합니다. 검색 함수는 findAll, findNear, findWithin, findCircle 이렇게 각각 네 가지를 만들 것입니다. 그중에서 일단 데이터베이스에 들어 있는 모든 커피숍 정보를 조회하는 **findAll 메소드**만 먼저 추가합니다.

참조 파일 | GeoExample>/database/coffeeshop_schema.js

```
.....

// 스키마에 static( ) 메소드 추가
// 모든 커피숍 조회
CoffeeShopSchema.static('findAll', function(callback) {
  return this.find({ }, callback);
});

.....
```

커피숍을 조회하는 코드는 스키마 파일에 넣어도 되고 라우팅하여 처리되는 파일에 넣어도 됩니다. 하지만 여기에서는 스키마 파일에 static( ) 메소드로 정의한 후 사용합니다.

모든 커피숍을 조회하는 메소드는 일반적인 다른 데이터를 조회할 때 사용하던 방식과 차이가 없습니다. 그러나 저장되어 있는 데이터가 1만 개 또는 10만 개 정도로 크게 늘어나면 시스템을 거의 사용할 수 없을 정도로 무리가 옵니다. 또한 조회 응답 속도도 많이 느려집니다. 따라서 필요하다면 limit 속성을 지정하여 조회하는 데이터의 개수에 제한을 두어야 합니다. 여기에서는 스키마 객체의 findAll( ) 메소드를 호출할 때는 아무런 조회 조건이 없는 빈 객체를 파라미터로 전달합니다.

커피숍을 검색할 수 있는 스키마 파일을 만들었으니 웹 서버가 실행될 때 이 파일을 로딩할 수 있도록 config.js 파일에 스키마 파일 정보를 추가합니다.

참조 파일 | GeoExample>/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'},
    {file : './coffeeshop_schema', collection : 'coffeeshop', schemaName : 'CoffeeShopSchema',
      modelName : 'CoffeeShopModel'}
  ],

.....
```

coffeeshop_schema 파일에 정의한 스키마 객체는 몽고디비 데이터베이스의 coffeeshop 컬렉션으로 만들어지도록 설정되었습니다. 스키마 객체를 만든 후 database 객체에 추가될 속성의 이름은 CoffeeShopSchema와 CoffeeShopModel로 정의하였습니다.

이제 커피숍 정보를 추가하는 함수와 데이터베이스의 coffeeshop 컬렉션에 있는 모든 커피숍 정보를 조회하는 함수를 추가합니다. [routes] 폴더에 coffeeshop.js 파일을 만든 후 다음과 같이 add 함수를 만듭니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

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

    var paramName = req.param('name');
    var paramAddress = req.param('address');
    var paramTel = req.param('tel');
    var paramLongitude = req.param('longitude');
    var paramLatitude = req.param('latitude');
    var database = req.app.get('database');

    if (database.db) {
      addCoffeeShop(database, paramName, paramAddress, paramTel,
                    paramLongitude, paramLatitude, function(err, result) {
        if (err) {throw err;}
        if (result) {
          console.dir(result);
          res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
          res.write('<h2>커피숍 추가 성공</h2>');
          res.end();
        } else {
          res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
          res.write('<h2>커피숍 추가 실패</h2>');
          res.end();
        }
      });
    } else {
      res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
      res.write('<h2>데이터베이스 연결 실패</h2>');
      res.end();
    }
};

.....

module.exports.add = add;
```

코드가 길어 보이지만 코드에서 처리하는 주요 내용은 (1) 클라이언트가 보내온 파라미터를 확인한 후 (2) addCoffeeShop( ) 메소드를 호출하여 커피숍 정보를 데이터베이스에 추가하고 (3) 결과를 응답으로 보내는 것입니다.

주의! 함수를 정의한 후 마지막으로 module.exports의 add 속성에 함수를 할당하는 것을 잊지 마세요.

커피숍을 추가하는 addCoffeeShop 함수의 코드는 다음과 같습니다.

```
.....

// 커피숍을 추가하는 함수
var addCoffeeShop = function(database, name, address, tel,
                             longitude, latitude, callback) {
  console.log('addCoffeeShop 호출됨.');

  // CoffeeShopModel 인스턴스 생성
  var coffeeshop = new database.CoffeeShopModel(
                 {name : name, address : address, tel : tel,
                     geometry : {
                         type : 'Point',
                         coordinates : [longitude, latitude]
                     }
                 }
                );

  // save( )로 저장
  coffeeshop.save(function(err) {
    if (err) {
      callback(err, null);
      return;
    }

    console.log("커피숍 데이터 추가함.");
    callback(null, coffeeshop);

  });
}

.....
```

커피숍 스키마 객체는 database.CoffeeShopSchema로 참조하고, 모델 객체는 database.CoffeeShopModel 객체로 참조할 수 있으므로 모델 객체를 사용해 새로운 모델 인스턴스 객체를 만듭니다. new 연산자로 모델 인스턴스 객체를 만들 때 파라미터에는 클라이언트로부터 전달받은 이름, 주소, 전화번호와 함께 경도와 위도 좌표를 넣어 줍니다. 위치 정보의 유형은 Point로 하고 경도와 위도 좌표는 배열 형태로 추가합니다. 마지막으로 모델 인스턴스 객체의 save( ) 메소드를 호출하면 커피숍 정보가 데이터베이스에 저장됩니다.

이번에는 커피숍 리스트를 조회하는 함수를 추가합니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
.....

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

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

  if (database.db) {
    // 1. 모든 커피숍 검색
    database.CoffeeShopModel.findAll(function(err, results) {
      if (err) { throw err; }
      if (results) {
        console.dir(results);
        res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
        res.write('<h2>커피숍 리스트</h2>');
        res.write('<div><ul>');

        for (var i = 0; i < results.length; i++) {
          var curName = results[i]._doc.name;
          var curAddress = results[i]._doc.address;
          var curTel = results[i]._doc.tel;
          var curLongitude = results[i]._doc.geometry.coordinates[0];
          var curLatitude = results[i]._doc.geometry.coordinates[1];

          res.write('    <li>#' + i + ' : ' + curName + ', ' + curAddress + ', '
                  + curTel + ', ' + curLongitude + ', ' + curLatitude + '</li>');
        }
        res.write('</ul></div>');
        res.end();
      } else {
        res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
        res.write('<h2>커피숍 리스트 조회 실패</h2>');
        res.end();
      }
    });
  } else {
    res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
    res.write('<h2>데이터베이스 연결 실패</h2>');
    res.end();
  }
};

.....
```

```
module.exports.list = list;

......
```

모든 커피숍을 리스트로 조회할 수 있도록 스키마 파일에 findAll( ) 메소드를 정의해 두었습니다. 따라서 모델 객체의 findAll( ) 메소드를 호출한 후 그 결과 값을 응답으로 보내 줍니다. 결과 값이 들어 있는 results 객체는 배열로 되어 있는데, 그 배열의 각 요소들에 들어 있는 속성 중에서 위치와 관련된 정보는 다음과 같이 참조할 수 있습니다.

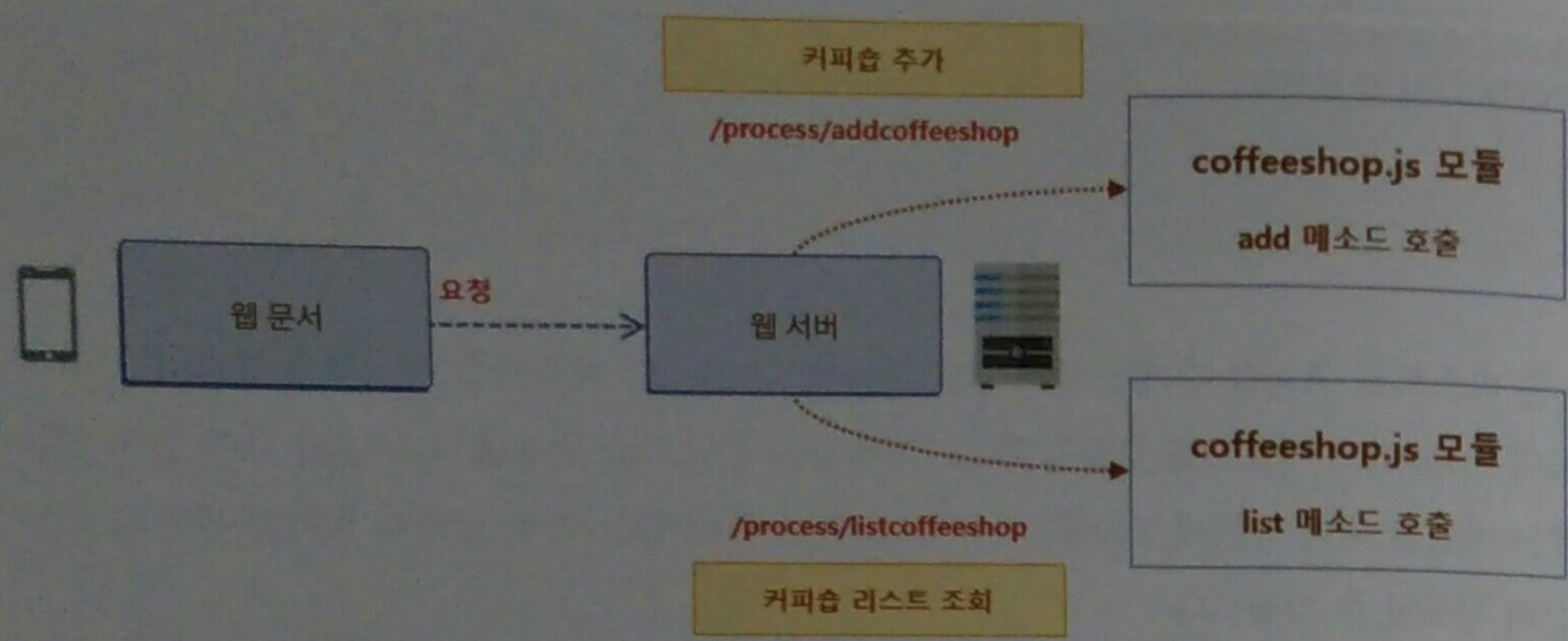
```
경도 좌표 : results[i]._doc.geometry.coordinates[0]
위도 좌표 : results[i]._doc.geometry.coordinates[1]
```

클라이언트로부터 커피숍을 추가해 달라는 요청과 커피숍 리스트를 조회하고 싶다는 요청을 받았을 때 처리하는 함수를 만들었으니 config.js 파일에 라우팅 설정을 추가합니다.

참조 파일 | GeoExample>/config/config.js

```
module.exports = {

......

  route_info : [
    //===== CoffeeShop =====//
   {file : './coffeeshop', path : '/process/addcoffeeshop', method : 'add', type : 'post'}
   ,{file : './coffeeshop', path : '/process/listcoffeeshop', method : 'list', type : 'post'}
  ],

......
```

config.js 파일의 route_info 배열 안에 커피숍 정보를 추가하고 커피숍 리스트를 조회하기 위해 라우팅 정보를 추가했습니다. 사용자 요청 패스가 /process/addcoffeeshop이면 coffeeshop 객체의 add 메소드를 호출하고 process/listcoffeeshop이면 coffeeshop 객체의 list 메소드를 호출합니다.



▲ 커피숍을 추가하고 리스트를 조회하기 위한 라우팅 설정

웹 서버에서 라우팅하여 데이터베이스에 추가하거나 조회하는 기능을 만들었습니다.

이번에는 웹 브라우저에서 조회할 웹 문서를 만들어 보겠습니다. 커피숍을 추가하는 웹 문서는 [public] 폴더 안에 addcoffeeshop.html 파일로 만들고 다음 코드를 입력합니다.

참조 파일 | GeoExample>/public/addcoffeeshop.html

```
.....

  <form method="post" action="/process/addcoffeeshop">
    <table>
      <tr>
        <td><label>이름</label></td>
        <td><input type="text" name="name" value="스타벅스 삼성역점"></td>
      </tr>
      <tr>
        <td><label>주소</label></td>
        <td><input type="text" name="address" value="서울특별시 강남구
                                                 테헤란로103길 9 제일빌딩"></td>
      </tr>
      <tr>
        <td><label>전화번호</label></td>
        <td><input type="text" name="tel" value="02-758-8585"></td>
      </tr>
      <tr>
        <td><label>경도</label></td>
        <td><input type="text" name="longitude" value="127.0638104"></td>
      </tr>
      <tr>
```

```
      <td><label>위도</label></td>
      <td><input type="text" name="latitude" value="37.5101225"></td>
    </tr>
  </table>
  <input type="submit" value="전송" name="">
</form>


.....
```

커피숍 이름과 주소, 전화번호를 작성할 수 있는 입력 상자를 추가하고, 그 아래에는 커피숍 위치를 경도와 위도로 입력할 수 있도록 만들었습니다. 그런데 여기에 입력할 경도와 위도 좌표는 어떻게 확인한 것일까요? 실제로는 커피숍의 정확한 위치를 측정하거나 측정한 데이터를 가져와서 입력해야 합니다. 그러나 정확한 위치를 측정하는 것이 어렵다면 다음과 같이 경위도 좌표를 확인할 수 있는 사이트를 이용합니다.

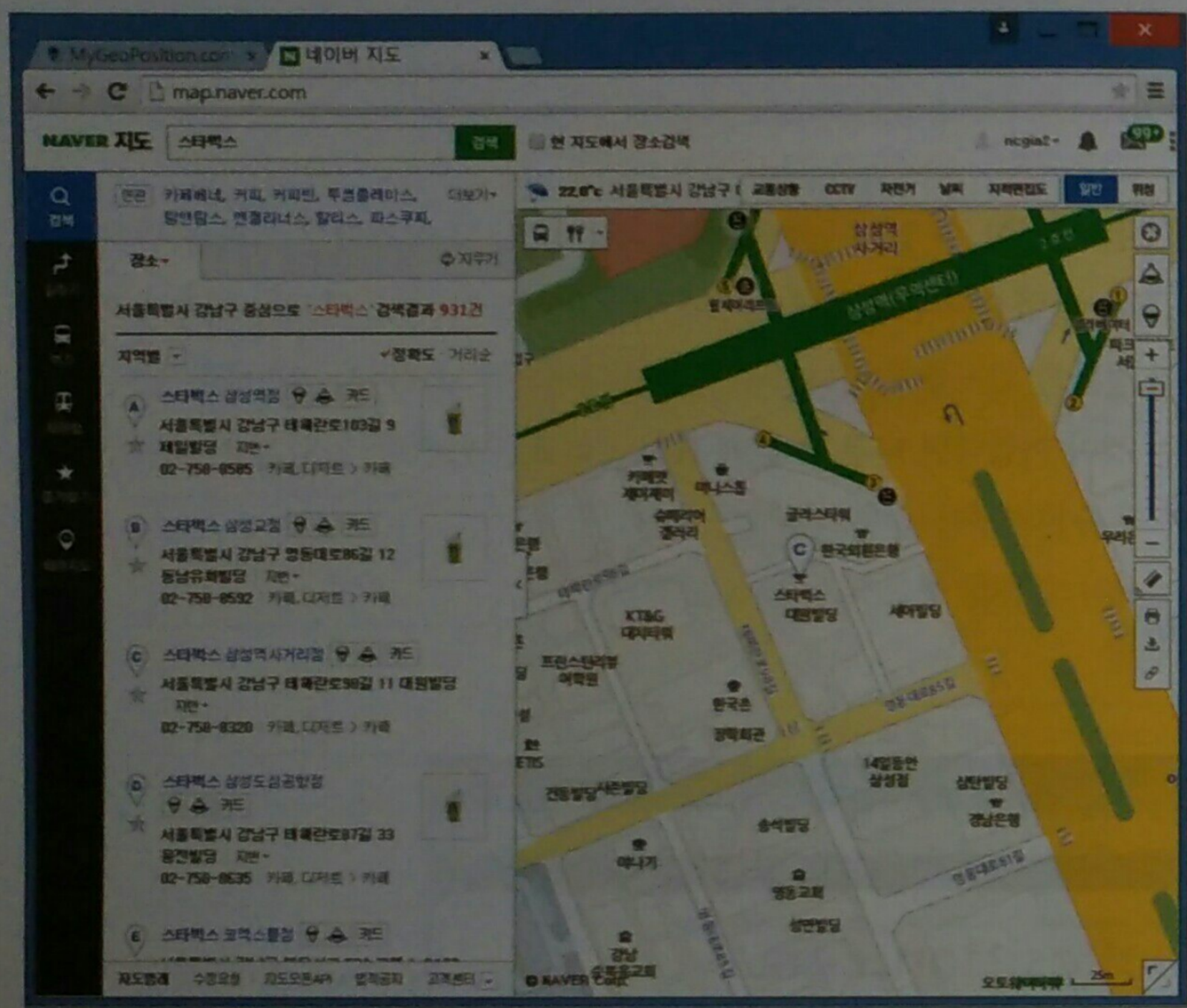
▶ http://mygeoposition.com

▲ 지도에서 경위도 좌표를 확인할 수 있는 사이트

이 사이트에서는 조회한 지도의 특정 위치를 가리키면 경위도 좌표를 알려 줍니다. 이런 기능을 제공하는 사이트는 많으니 적당한 사이트를 찾아 이용하기 바랍니다.

여기에서는 서울의 삼성역 주변 지도를 검색하였습니다. 그런데 이 사이트의 지도에는 커피숍 아이콘도 있고 위치 좌표도 확인할 수 있지만 커피숍 이름이나 전화번호는 확인할 수 없습니다. 따라서 커피숍 정보는 여러분이 자주 사용하는 포털 사이트의 지도 서비스에서 쉽게 확인할 수 있습니다.

다음은 네이버 사이트에서 삼성역 주변에 있는 스타벅스 커피숍을 조회한 것입니다.



▲ 네이버 사이트에서 커피숍 정보를 조회한 경우

삼성역 주변에는 스타벅스 삼성역점, 스타벅스 삼성교점, 스타벅스 삼성역사거리점이 있다는 것을 확인할 수 있습니다. 이 정보를 입력하면 세 개의 커피숍 정보를 데이터베이스에 추가할 수 있습니다.

이제 커피숍을 추가할 수 있는 기능은 모두 만들었으니 커피숍 리스트를 조회할 수 있는 웹 문서를 만든 후 실행합니다. 커피숍 리스트를 조회하는 웹 문서는 [public] 폴더 안에 listcoffeeshop.html 파일로 만든 후 다음 코드를 입력합니다.

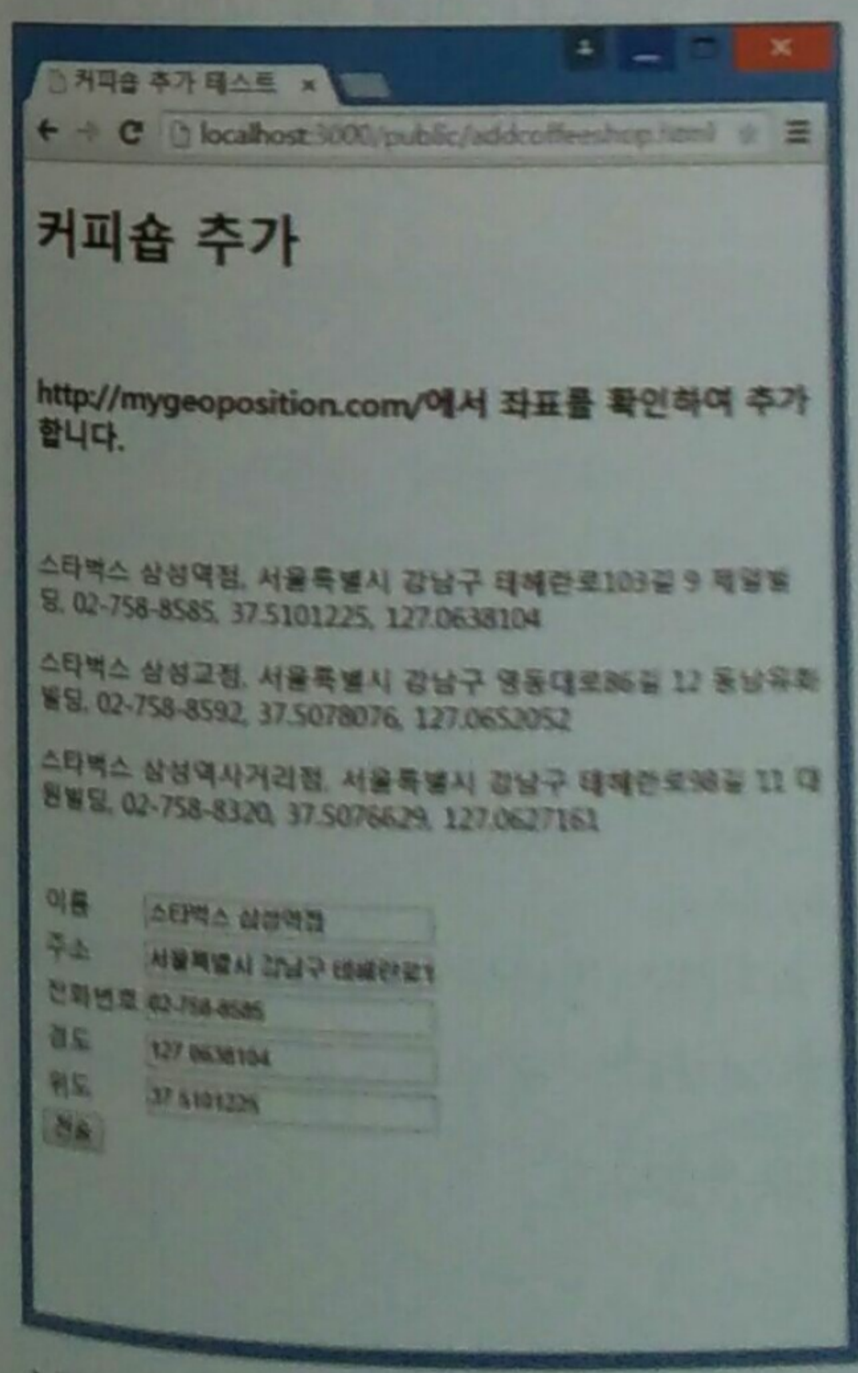
참조 파일 | GeoExample>/public/listcoffeeshop.html

```
……

<h1>커피숍 리스트</h1>
<br>
<form method="post" action="/process/listcoffeeshop">
  <table>
    <tr>
      <td><label>아래 [전송] 버튼을 누르세요.</label></td>
    </tr>
  </table>
  <input type="submit" value="전송" name="">
</form>

……
```

단순히 [전송] 버튼을 누르면 서버에 /process/listcoffeeshop 패스로 요청합니다. 이제 app.js 파일을 실행한 후 웹 브라우저를 열고 addcoffeeshop.html 파일을 엽니다.

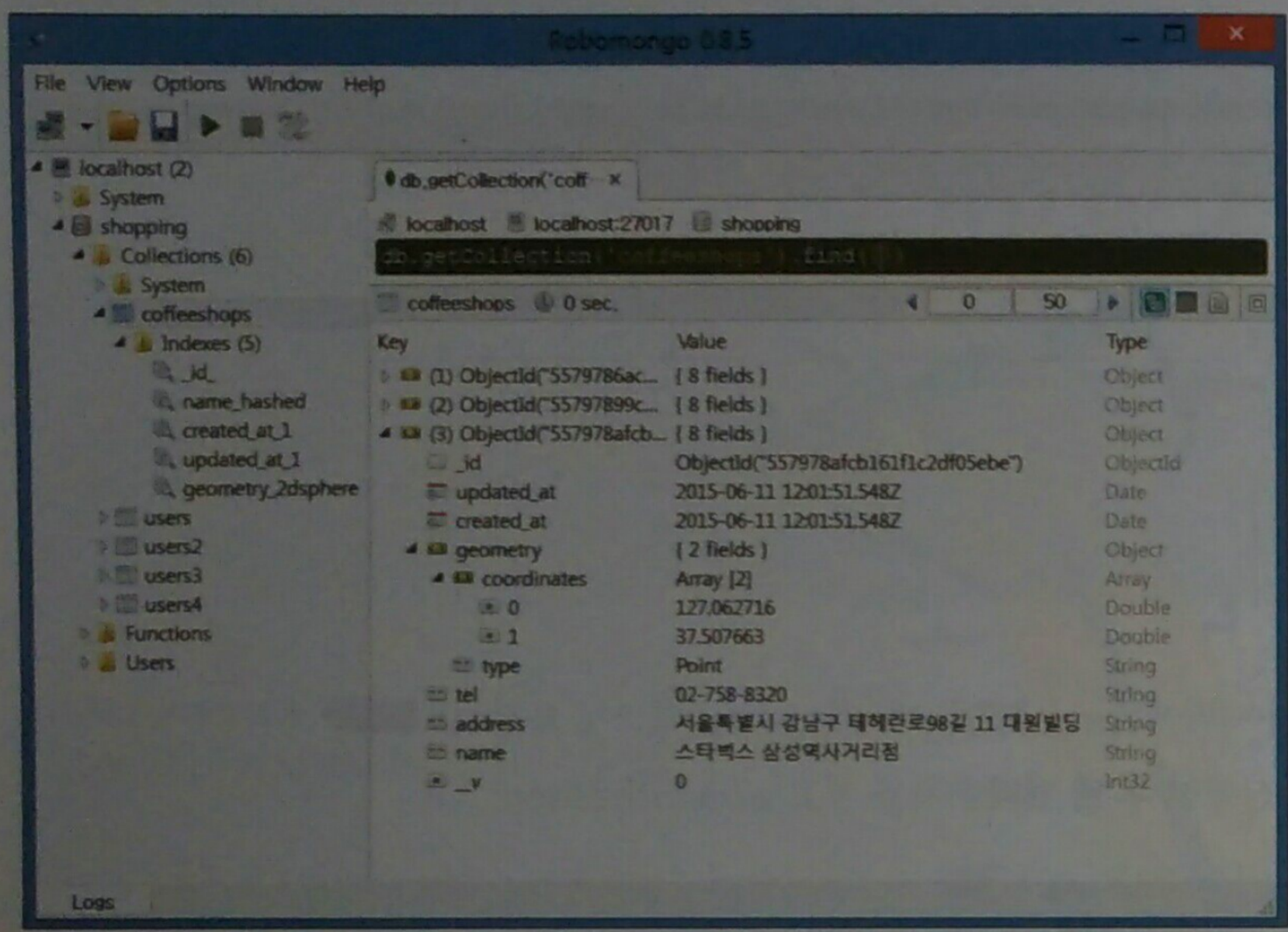
주의! 몽고디비 데이터베이스도 실행되고 있는지 확인하세요.

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



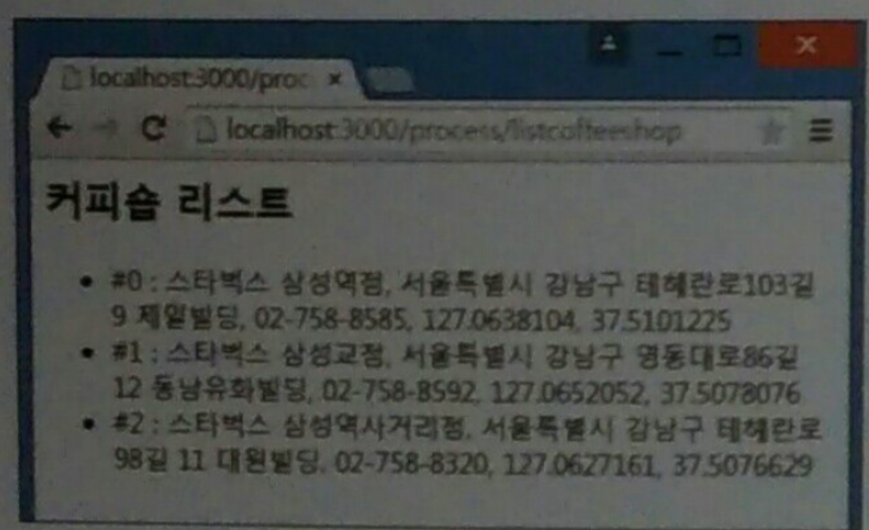
▲ 커피숍을 추가하는 웹 문서

네이버 사이트와 mygeoposition.com 사이트에서 확인한 정보를 웹 문서에 미리 써 두었습니다. 이 정보를 그대로 입력 상자에 입력하고 [전송] 버튼을 누르면 하나씩 서버에 저장됩니다. 이렇게 추가한 커피숍 정보가 데이터베이스에 잘 저장되었는지 그리고 인덱스가 잘 만들어졌는지 Robomongo 화면에서 확인합니다.



▲ Robomongo 화면에서 확인한 커피숍 정보

세 곳의 커피숍 정보가 잘 저장되어 있고 인덱스도 잘 만들어져 있습니다. 이제 웹 브라우저에서 listcoffeeshop.html 파일을 열고 [전송] 버튼을 누르면 커피숍 리스트를 확인할 수 있습니다.



▲ 웹 문서에서 커피숍 리스트를 확인한 경우

지금까지 커피숍 정보를 저장한 후 리스트 형식으로 모든 정보를 조회하는 방법에 대해 알아보았습니다. 몽고디비에 컬렉션을 만들기 위해 스키마를 정의할 때 위치 정보를 저장하는 속성을 어떻게 정의하는지가 가장 중요합니다. 이제 커피숍을 위치로 찾는 방법에 대해 알아보겠습니다.

12-2 | 가장 가까운 커피숍 찾기

앞에서도 말했지만 위치 좌표로 커피숍을 검색할 수 있는 대표적인 방법으로 (1) 가장 가까운 커피숍 찾기, (2) 범위 안의 커피숍 찾기, (3) 반경 안의 커피숍 찾기를 들 수 있습니다. 또한 몽고디비에서는 행정 구역처럼 영역(Polygon) 안에 있는 커피숍을 찾는 것도 가능합니다. 그러나 여기에서는 대표적인 세 가지 경우에 해당하는 기능만 만들어 보겠습니다.

▲ 커피숍을 위치로 찾는 대표적인 세 가지 방법

커피숍을 조회하거나 찾는 함수는 [database] 폴더의 coffeeshop_schema.js 파일 안에서 스키마 객체의 static( ) 메소드를 호출하여 등록했습니다. 가장 가까운 커피숍을 찾는 함수도 이와 마찬가지로 coffeeshop_schema.js 파일 안에 정의합니다.

참조 파일 | GeoExample>/database/coffeeshop_schema.js

```
.....

// 가장 가까운 커피숍 조회
CoffeeShopSchema.static('findNear', function(longitude, latitude, maxDistance, callback) {
  console.log('CoffeeShopSchema의 findNear 호출됨.');

  this.find().where('geometry').near(
              {center : {type : 'Point',
                         coordinates : [parsefloat(longitude), parsefloat(latitude)]
                        },
               maxDistance : maxDistance
              }).limit(1).exec(callback);
  });

.....
```

가장 가까운 커피숍을 조회할 때는 geometry 속성의 near( ) 메소드를 호출합니다. near( ) 메소드를 호출하는 기본 형식은 다음과 같습니다.

```
find().where(속성 이름).near(조회 조건)
```

where( ) 메소드 안에는 조회 대상이 되는 속성 이름을 넣어 주고 near( ) 메소드 안에는 조회 조건을 넣어 줍니다. near( ) 메소드는 어느 위치를 기준으로 할지 알아야 하므로 기준점의 위치와 기준점으로부터의 최대 거리를 조회 조건으로 넣어 줍니다. 기준점의 경도 좌표, 위도 좌표 그리고 기준점으로부터의 최대 거리는 클라이언트로부터 받은 데이터를 파라미터로 전달받아 사용합니다. 그런데 그중에서 longitude, latitude는 숫자 타입으로 들어가야 하므로 parseFloat( ) 메소드를 호출하여 문자열을 숫자로 바꾸어 줍니다.

limit( ) 함수는 조회 결과의 개수를 제한하는 역할을 합니다. 여기에서는 파라미터로 1을 넣어 주어 한 개의 커피숍 정보만 반환합니다.

스키마 객체에 함수를 등록했으므로 이 함수를 사용하면서 클라이언트의 요청을 받아 처리할 라우팅 함수를 만듭니다. [routes] 폴더의 coffeeshop.js 파일을 열고 findNear 함수를 만든 후 module.exports 객체의 findNear 속성에 할당합니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
……

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

  var paramLongitude = req.param('longitude');
  var paramLatitude = req.param('latitude');
  var maxDistance = 1000;
  var database = req.app.get('database');

  if (database.db) {
    // 1. 가까운 커피숍 검색
    database.CoffeeShopModel.findNear(paramLongitude, paramLatitude, maxDistance,
      function(err, results) {
        if (err) { throw err; }
        if (results) {
```

```
console.dir(results);
res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
res.write('<h2>가까운 커피숍</h2>');
res.write('<div><ul>');

for (var i = 0; i < results.length; i++) {
  var curName = results[i]._doc.name;
  var curAddress = results[i]._doc.address;
  var curTel = results[i]._doc.tel;
  var curLongitude = results[i]._doc.geometry.coordinates[0];
  var curLatitude = results[i]._doc.geometry.coordinates[1];

  res.write('    <li>#' + i + ' : ' + curName + ', ' + curAddress + ', '
          + curTel + ', ' + curLongitude + ', ' + curLatitude + '</li>');
 }

 res.write('</ul></div>');
 res.end();
} else {
 res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
 res.write('<h2>가까운 커피숍 조회 실패</h2>');
 res.end();
    }
  });
 } else {
  res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
  res.write('<h2>데이터베이스 연결 실패</h2>');
  res.end();
 }
};


......

module.exports.findNear = findNear;

......
```

클라이언트로부터는 경도 좌표와 위도 좌표를 전달받습니다. 가까운 커피숍을 찾을 때의 최대 거리는 1000으로 지정합니다.

정박사의 한마디 스마트폰에서 확인할 수 있는 경위도 좌표의 단위는 무엇일까요?

웹 사이트나 스마트폰에서 GPS를 사용해서 확인한 GPS 좌표를 보면 127.0638104, 37.5101225처럼 소수점 여섯 자리나 일곱 자리까지 표시합니다. 이 좌표 중에서 경도에 100000.0을 곱하고 다시 1.110을 곱하면 1의 자리가 1m 단위가 됩니다. 위도는 100000.0을 곱하고 다시 0.084를 곱하면 1의 자리가 1m가 됩니다. 여기에서는 원래의 좌표를 그대로 사용하기 위해 최대 거리를 1000이라는 숫자 단위로 넣었습니다.

만약 두 위치 사이의 거리를 알고 싶다면 다음 함수를 만들어 사용할 수 있습니다.

```
var distance = func(startPoint, endPoint) {
  var distance = 0;
  var x = (startPoint.latitude - endPoint.latitude) * 100000.0 * 0.884;
  var y = (startPoint.longitude - endPoint.longitude) * 100000.0 * 1.110;
  distance = Math.sqrt((x * x) + (y * y));

  return distance;
}
```

모델 객체의 findNear( ) 메소드를 호출하면서 경도 좌표와 위도 좌표를 파라미터로 전달하면 가장 가까운 커피숍을 조회할 수 있습니다. 조회한 커피숍 정보는 클라이언트에 응답으로 보내 줍니다.

웹 브라우저에 보여 줄 웹 문서는 [public] 폴더 안에 nearcoffeeshop.html 파일로 만들고 다음 코드를 입력합니다.

참조 파일 | GeoExample>/public/nearcoffeeshop.html

```
.....

<body>
  <h1>가까운 커피숍 찾기</h1>
  <br>
  <h3>http://mygeoposition.com/에서 좌표를 확인하여 추가합니다.</h3>
  <br>
  <p>삼성역 3번 출구 버스 정류장</p>
  <br>
  <form method="post" action="/process/nearcoffeeshop">
    <table>
      <tr>
        <td><label>경도</label></td>
        <td><input type="text" name="longitude" value="127.0632954"></td>
      </tr>
      <tr>
```

```
      <td><label>위도</label></td>
      <td><input type="text" name="latitude" value="37.5079523"></td>
    </tr>
  </table>
  <input type="submit" value="전송" name="">
 </form>
</body>

……
```

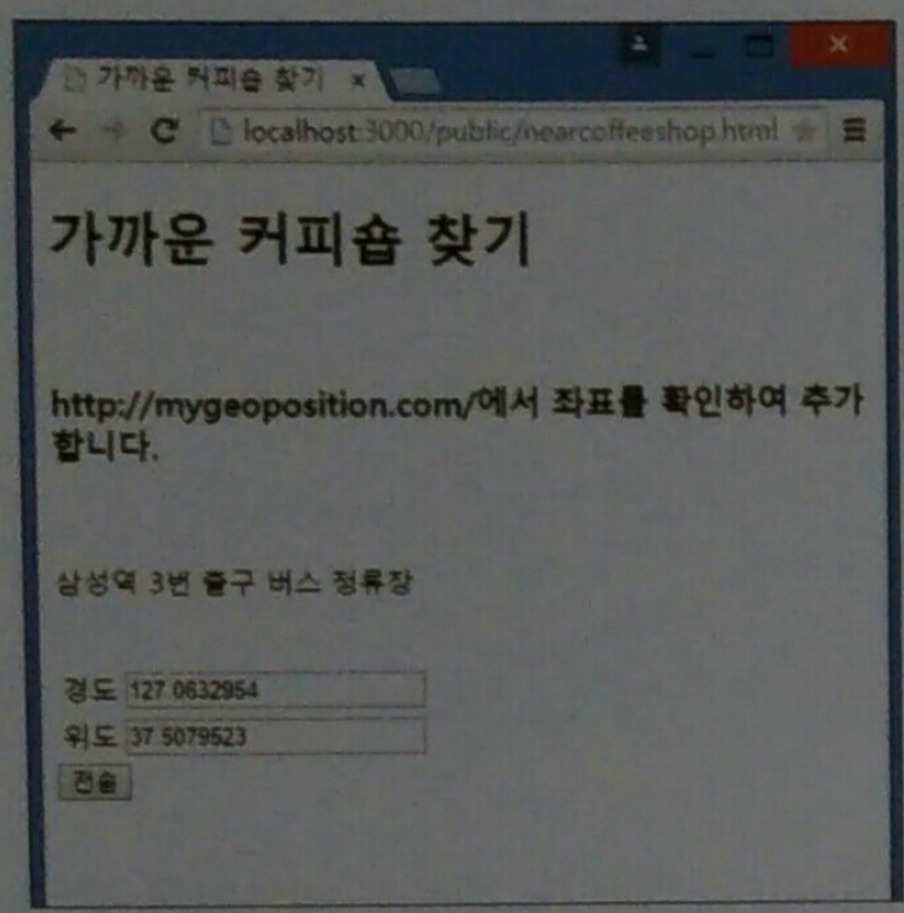
사용자 위치를 삼성역 3번 출구 버스 정류장 부근으로 가정하고 mygeoposition.com 사이트에서 경위도 좌표를 확인합니다. 위 코드에서 입력 상자에 입력해 둔 좌표 값이 미리 확인한 좌표 값이므로 여기에서는 [전송] 버튼만 누르면 됩니다. 마지막으로 config.js 파일에 클라이언트의 요청을 라우팅하는 코드를 추가합니다.

참조 파일 | GeoExample>/config/config.js

```
……

route_info : [
  //===== CoffeeShop =====//
  {file : './coffeeshop', path : '/process/addcoffeeshop', method : 'add', type : 'post'}
  ,{file : './coffeeshop', path : '/process/listcoffeeshop', method : 'list', type : 'post'}
  ,{file : './coffeeshop', path : '/process/nearcoffeeshop', method : 'findNear', type : 'post'}
],

……
```

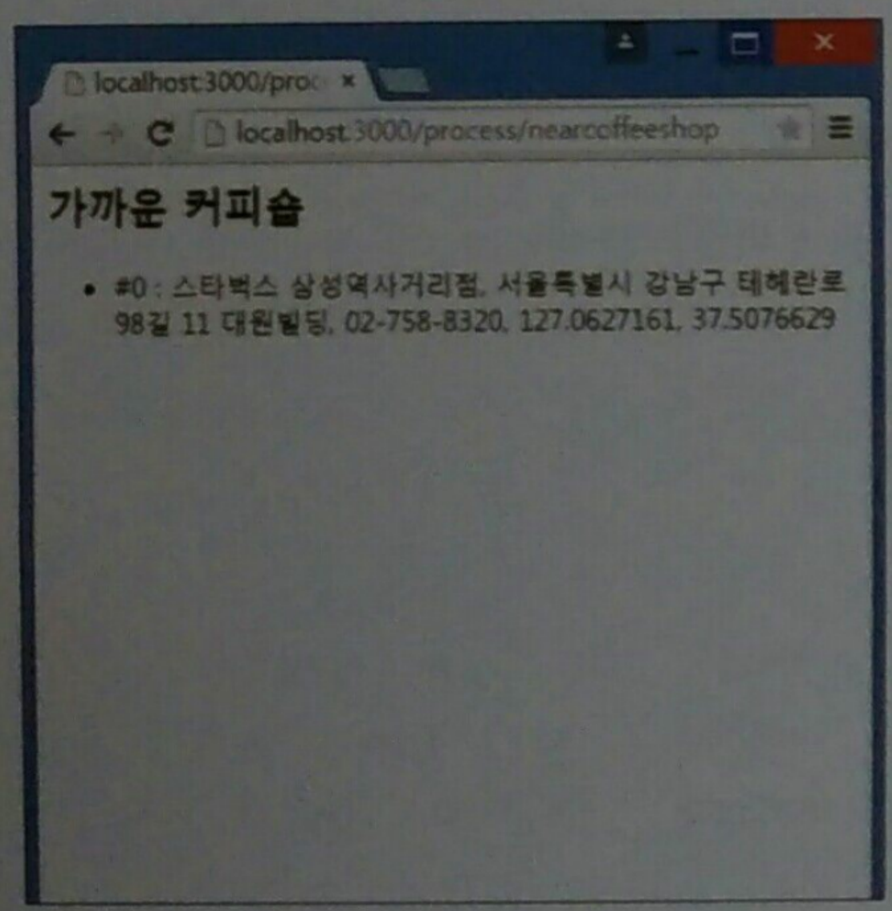
위치 좌표를 사용한 조회 기능이 정상적으로 동작하는지 확인하기 위해 app.js 파일을 다시 실행한 후 nearcoffeeshop.html 파일을 엽니다.

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



◀ nearcoffeeshop.html 파일을 열어 가까운 커피숍 찾기

[전송] 버튼을 누르면 다음처럼 현재 사용자 위치에서 가장 가까운 커피숍을 찾아서 보여 줍니다.



◀ 현재 사용자 위치에서 가장 가까운 커피숍을 찾아 보여 준 경우

지도에서 확인해 보면 실제로 삼성역 3번 출구 버스 정류장에서 가장 가까운 커피숍은 스타벅스 삼성역 사거리점인 것을 알 수 있습니다.

◀ 지도에서 확인한 커피숍 위치

만약 위치로 검색한 커피숍을 지도 위에 보여 주고 싶다면 지도 API를 사용하면 됩니다. 구글맵(Google Map) 지도를 보여주고 그 위에 아이콘으로 커피숍의 위치를 표시하는 방법은 이 장의 마지막에 있는 '12-5 지도에 커피숍의 위치 표시하기'에서 설명합니다.

12-3 | 영역 안의 커피숍 찾기

위치 좌표로 커피숍을 찾는 기능을 한번 만들어 보았으니 특정 범위에서 커피숍을 검색하거나 반경 안에서 커피숍을 찾는 기능도 똑같은 과정을 거치면 만들 수 있습니다. 일정 범위 안에서 커피숍을 검색할 때는 사각형 박스로 범위 값을 주거나 원형의 반경 값을 줄 수 있습니다. 그러면 사각형 박스를 사용해 검색하는 기능을 만들어 보겠습니다.

먼저 사각형 영역을 만들고 커피숍을 찾는 함수를 coffeeshop_schema.js 파일 안에 정의합니다.

참조 파일 | GeoExample)/database/coffeeshop_schema.js

```
.....

// 일정 범위 안의 커피숍 조회
CoffeeShopSchema.static('findWithin', function(topleft_longitude, topleft_latitude,
    bottomright_longitude, bottomright_latitude, callback) {
    console.log('CoffeeShopSchema의 findWithin 호출됨.');

    this.find().where('geometry').within(
            {box : [[parseFloat(topleft_longitude), parseFloat(topleft_latitude)],
                [parseFloat(bottomright_longitude), parseFloat(bottomright_latitude)]]
            }).exec(callback);
});

.....
```

일정 범위 안의 커피숍을 조회할 때는 geometry 속성의 within( ) 메소드를 호출합니다. within( ) 메소드를 호출하는 기본 형식은 다음과 같습니다.

```
find().where(속성 이름).within(조회 조건)
```

where( ) 메소드 안에는 조회 대상이 되는 속성 이름을 넣어 주고 within( ) 메소드 안에는 조회 조건을 넣어 줍니다. within( ) 메소드의 경우에는 검색할 범위를 사각형 모양으로 알려 주어야 하므로 사각형의 왼쪽 윗부분과 오른쪽 아랫부분의 좌표 값을 파라미터로 전달합니다. 여기에 전달되는 좌표 값들은 숫자 타입이어야 하므로 parseFloat( ) 메소드를 호출하여 문자열을 숫자로 바꾸어 줍니다.

스키마 객체에 findWithin 함수를 등록했으므로 이 함수를 사용하면서 클라이언트의 요청을 받아 처리할 함수를 만듭니다. [routes] 폴더의 coffeeshop.js 파일을 열고 findWithin 함수를 만든 후 module.exports 객체의 findWithin 속성에 할당합니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
……

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

  var paramTopLeftLongitude = req.param('topleft_longitude');
  var paramTopLeftLatitude = req.param('topleft_latitude');
  var paramBottomRightLongitude = req.param('bottomright_longitude');
  var paramBottomRightLatitude = req.param('bottomright_latitude');
  var database = req.app.get('database');

  if (database.db) {
    // 1. 가까운 커피숍 검색
    database.CoffeeShopModel.findWithin(paramTopLeftLongitude, paramTopLeftLatitude,
        paramBottomRightLongitude, paramBottomRightLatitude, function(err, results) {
        if (err) { throw err; }

……

module.exports.findWithin = findWithin;
……
```

클라이언트로부터는 검색할 대상들이 포함될 사각형 박스의 왼쪽 윗부분 경위도 좌표와 오른쪽 아랫부분 경위도 좌표를 전달받습니다. 그리고 이 좌표 값들은 모델 객체의 findWithin( ) 메소드를 호출할 때 파라미터로 전달됩니다. 조회한 커피숍 정보는 클라이언트에 응답으로 보내 줍니다.

웹 브라우저에 보여 줄 웹 문서는 [public] 폴더 안에 withincoffeeshop.html 파일로 만들고 다음 코드를 입력합니다.

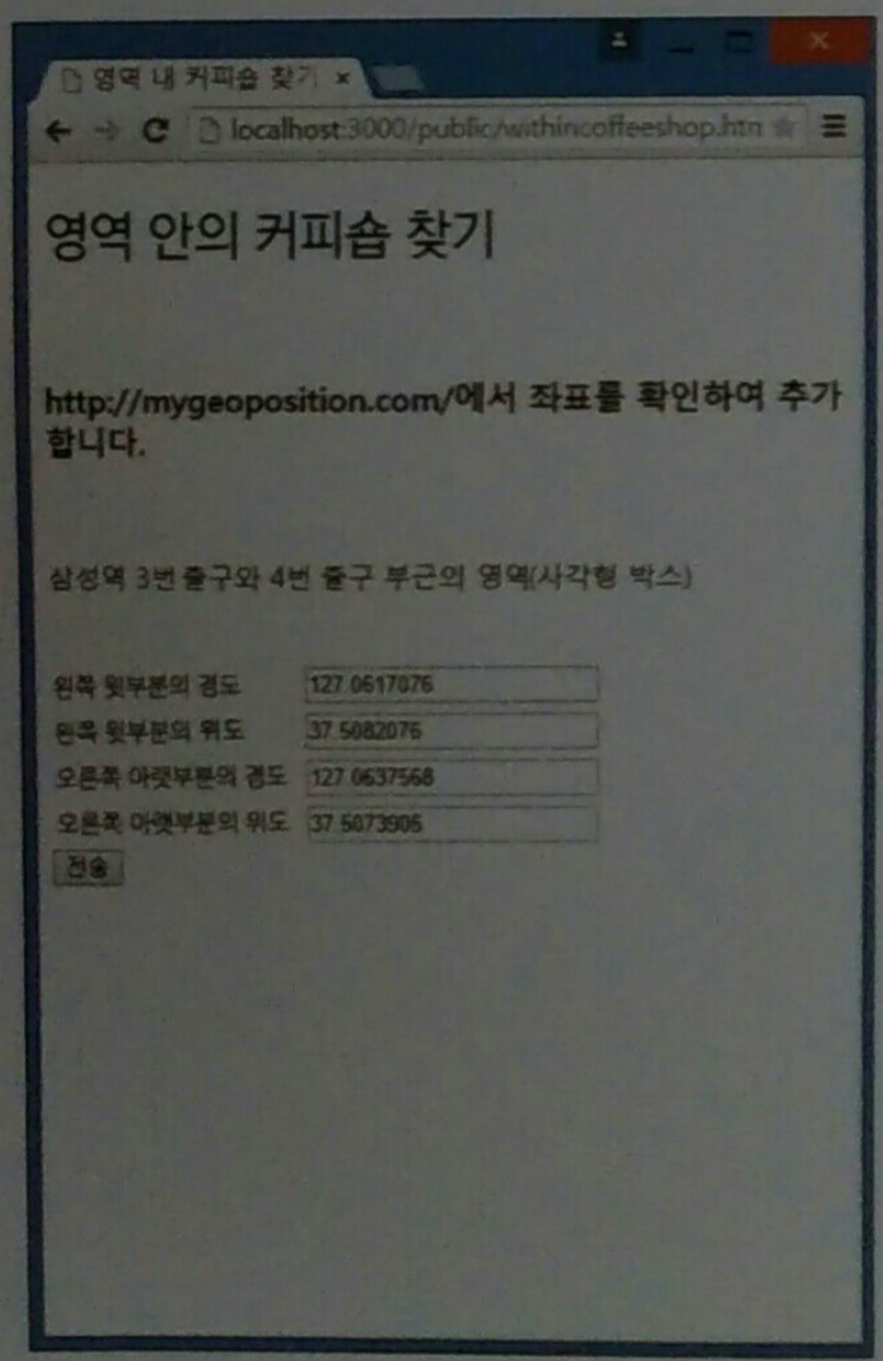
```
.....

<body>
  <h1>영역 안의 커피숍 찾기</h1>
  <br>
  <h3>http://mygeoposition.com/에서 좌표를 확인하여 추가합니다.</h3>
  <br>
  <p>삼성역 3번 출구와 4번 출구 부근의 영역(사각형 박스)</p>
  <br>
  <form method="post" action="/process/withincoffeeshop">
    <table>
      <tr>
        <td><label>왼쪽 윗부분의 경도</label></td>
        <td><input type="text" name="topleft_longitude" value="127.0617076"></td>
      </tr>
      <tr>
        <td><label>왼쪽 윗부분의 위도</label></td>
        <td><input type="text" name="topleft_latitude" value="37.5082076"></td>
      </tr>
      <tr>
        <td><label>오른쪽 아랫부분의 경도</label></td>
        <td><input type="text" name="bottomright_longitude"
                                value="127.0637568"></td>
      </tr>
      <tr>
        <td><label>오른쪽 아랫부분의 위도</label></td>
        <td><input type="text" name="bottomright_latitude"
                                value="37.5073906"></td>
      </tr>
    </table>
    <input type="submit" value="전송" name="">
  </form>
</body>

.....
```

검색할 사각형 박스는 삼성역 3번 출구 버스 정류장 부근에 만듭니다. 이곳의 경위도 좌표는 mygeoposition.com 사이트에서 확인할 수 있습니다. 위의 코드에 입력한 좌표 값을 그대로 둔 채 [전송] 버튼을 누르면 이 좌표 값으로 서버에 요청합니다. 마지막으로 config.js 파일에 클라이언트의 요청을 라우팅하는 코드를 추가합니다.

```
……,

  route_info : [
    //===== CoffeeShop =====//
    {file : './coffeeshop', path : '/process/addcoffeeshop', method : 'add', type : 'post'}
    ,{file : './coffeeshop', path : '/process/listcoffeeshop', method : 'list', type : 'post'}
    ,{file : './coffeeshop', path : '/process/nearcoffeeshop', method : 'findNear', type : 'post'}
     ,{file : './coffeeshop', path : '/process/withincoffeeshop', method : 'findWithin', type :
'post'}
  ],

……,
```

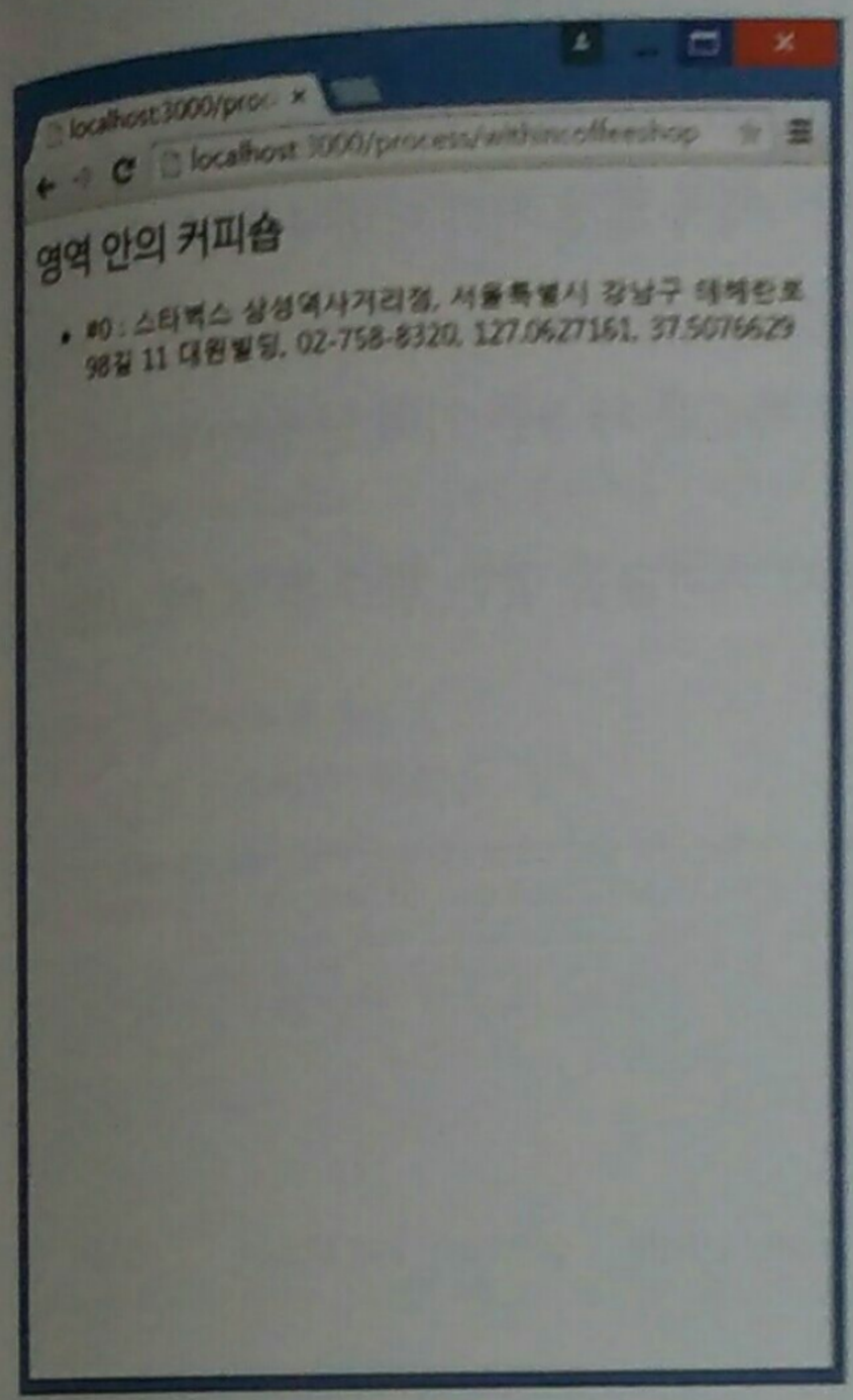
이제 app.js 파일을 다시 실행한 후 withincoffeeshop.html 파일을 엽니다.

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



▲ 영역 안의 커피숍을 찾기 위해 withincoffeeshop.html 파일을 연 경우

[전송] 버튼을 누르면 다음과 같이 사각형 영역으로 지정한 영역 안에 있는 커피숍을 찾아서 모두 보여 줍니다.



▲사각형 영역 안의 커피숍을 찾아 보여 준 경우

커피숍 검색 기능을 지원하는 많은 앱들은 현재 사용자가 보고 있는 지도 영역 안에 들어 있는 커피숍을 찾아 주기 때문에 지금 만들어 본 within( ) 메소드 방법을 많이 사용합니다.

12-4 | 반경 안의 커피숍 찾기

마지막으로 반경 안에 있는 커피숍을 찾는 방법을 알아보겠습니다. 반경은 사용자가 있는 위치를 기준으로 원을 그렸을 때 만들어지는 영역을 말합니다. 즉, 사용자의 위치에서 일정 거리 이내에 있는 커피숍을 찾을 때 사용합니다. 몽고디비에서는 앞에서 사용한 within( ) 메소드에 반경 정보를 넣어 주면 그 안에 들어 있는 것들을 찾을 수 있습니다. 따라서 앞에서 만든 기능과 똑같은 과정을 거쳐 만듭니다.

먼저 기준점과 기준점으로부터의 거리를 파라미터로 전달하면서 커피숍을 찾는 함수를 coffeeshop_schema.js 파일 안에 정의합니다.

참조 파일 | GeoExample>/database/coffeeshop_schema.js

```
.....

  // 일정 반경 안의 커피숍 조회
   CoffeeShopSchema.static('findCircle', function(center_longitude, center_latitude, radius,
callback) {
    console.log('CoffeeShopSchema의 findCircle 호출됨.');

    // change radian : 1/6371 -> 1km
    this.find().where('geometry').within(
          {center : [parseFloat(center_longitude), parseFloat(center_latitude)],
              radius : parseFloat(radius/6371000),
              unique : true, spherical : true
          }).exec(callback);
});

.....
```

일정 범위 안의 커피숍을 조회할 때는 geometry 속성을 찾아 within( ) 메소드를 호출하는데 그 안에 들어가는 조회 조건으로 기준점의 좌표를 전달하는 center 속성과 기준점으로부터의 거리를 알려 주는 radius 속성을 넣어 줍니다. 여기에 전달되는 좌표 값들은 숫자 타입이어야 하므로 parseFloat( ) 메소드를 호출하여 문자열을 숫자로 바꾸어 줍니다.

스키마 객체에 findCircle 함수를 등록했으므로 이 함수를 사용하면서 클라이언트의 요청을 받아 처리할 함수를 만듭니다. [routes] 폴더의 coffeeshop.js 파일을 열고 findCircle 함수를 만든 후 module.exports 객체의 findCircle 속성에 할당합니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
……

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

  var paramCenterLongitude = req.param('center_longitude');
  var paramCenterLatitude = req.param('center_latitude');
  var paramRadius = req.param('radius');
  var database = req.app.get('database');

  if (database.db) {
    // 1. 가까운 커피숍 검색
    database.CoffeeShopModel.findCircle(paramCenterLongitude,
        paramCenterLatitude, paramRadius, function(err, results) {
        if (err) { throw err; }

……

module.exports.findCircle = findCircle;

……
```

클라이언트로부터는 기준점의 경위도 좌표와 기준점으로부터의 거리를 전달받습니다. 그리고 이 파라미터 값들은 모델 객체의 findCircle( ) 메소드를 호출하면서 전달합니다. 조회한 커피숍 정보는 클라이언트에 응답으로 보내 줍니다.

웹 브라우저에 보여 줄 웹 문서는 [public] 폴더 안에 circlecoffeeshop.html 파일로 만들고 다음 코드를 입력합니다.

참조 파일 | GeoExample>/public/circlecoffeeshop.html

```
……

<body>
  <h1>반경 안의 커피숍 찾기</h1>
  <br>
  <h3>http://mygeoposition.com/에서 좌표를 확인하여 추가합니다.</h3>
  <br>
  <p>삼성역 3번 출구 버스 정류장 반경 100m (원)</p>
  <br>
  <form method="post" action="/process/circlecoffeeshop">
```

```
    <table>
      <tr>
        <td><label>중심점 경도</label></td>
        <td><input type="text" name="center_longitude" value="127.0632954"></td>
      </tr>
      <tr>
        <td><label>중심점 위도</label></td>
        <td><input type="text" name="center_latitude" value="37.5079523"></td>
      </tr>
      <tr>
        <td><label>반경</label></td>
        <td><input type="text" name="radius" value="100"></td>
      </tr>
    </table>
    <input type="submit" value="전송" name="">
  </form>
</body>

……,
```

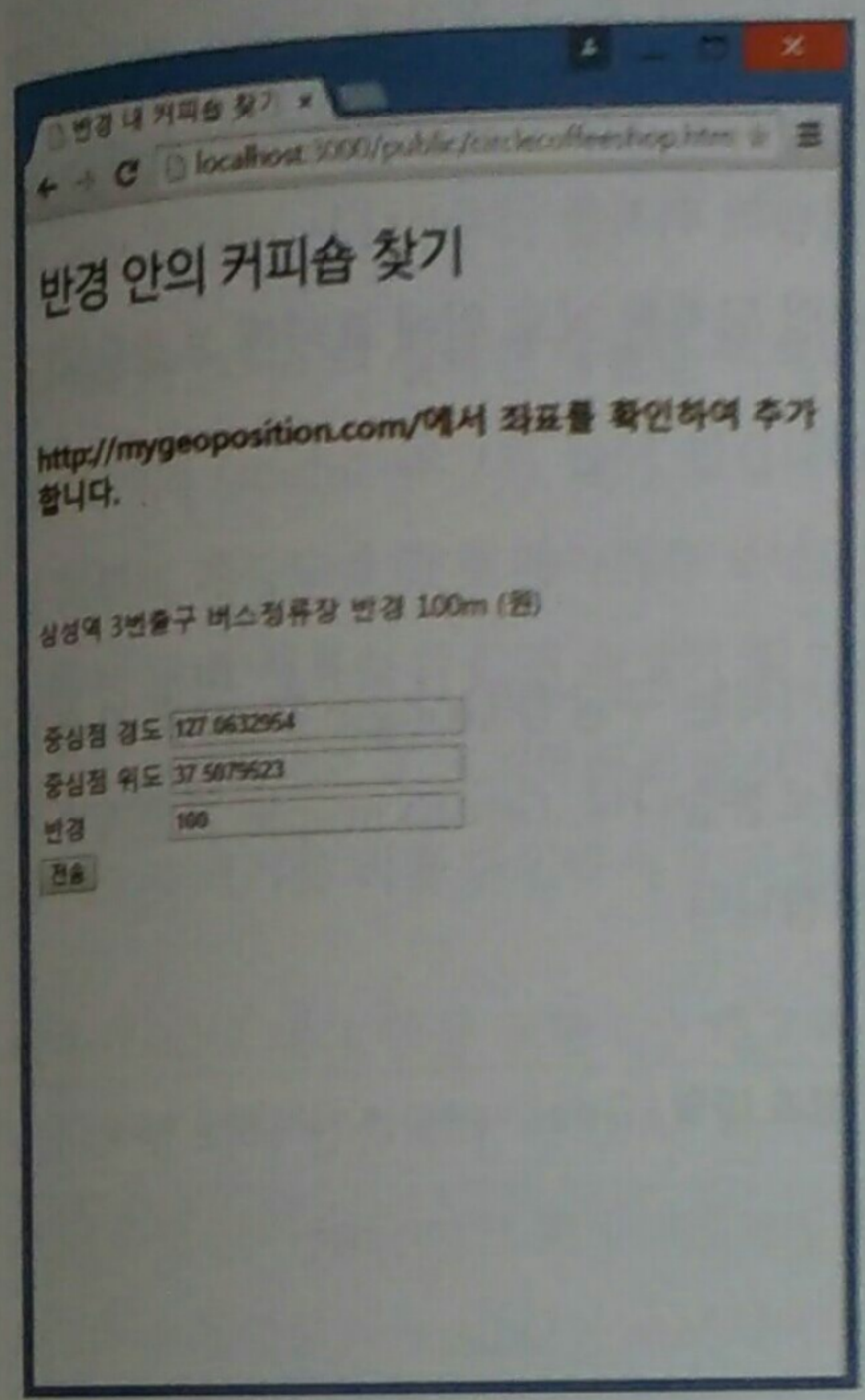
검색할 반경의 기준점은 삼성역 3번 출구 버스 정류장 부근으로 합니다. 이곳의 경위도 좌표는 이전과 마찬가지로 mygeoposition.com 사이트에서 확인할 수 있습니다. 위의 문서에 입력한 좌표 값과 기준점으로부터의 거리 값을 그대로 둔 채 [전송] 버튼을 누르면 이 좌표 값으로 서버에 요청합니다. 마지막으로 config.js 파일에 클라이언트의 요청을 라우팅하는 코드를 추가합니다.

참조 파일 | GeoExample>/config/config.js

```
……,

route_info : [
  //===== CoffeeShop =====//
  {file : './coffeeshop', path : '/process/addcoffeeshop', method : 'add', type : 'post'}
  ,{file : './coffeeshop', path : '/process/listcoffeeshop', method : 'list', type : 'post'}
  ,{file : './coffeeshop', path : '/process/nearcoffeeshop', method : 'findNear', type : 'post'}
  ,{file : './coffeeshop', path : '/process/withincoffeeshop', method : 'findWithin', type :
'post'}
  ,{file : './coffeeshop', path : '/process/circlecoffeeshop', method : 'findCircle', type :
'post'}
],

……,
```

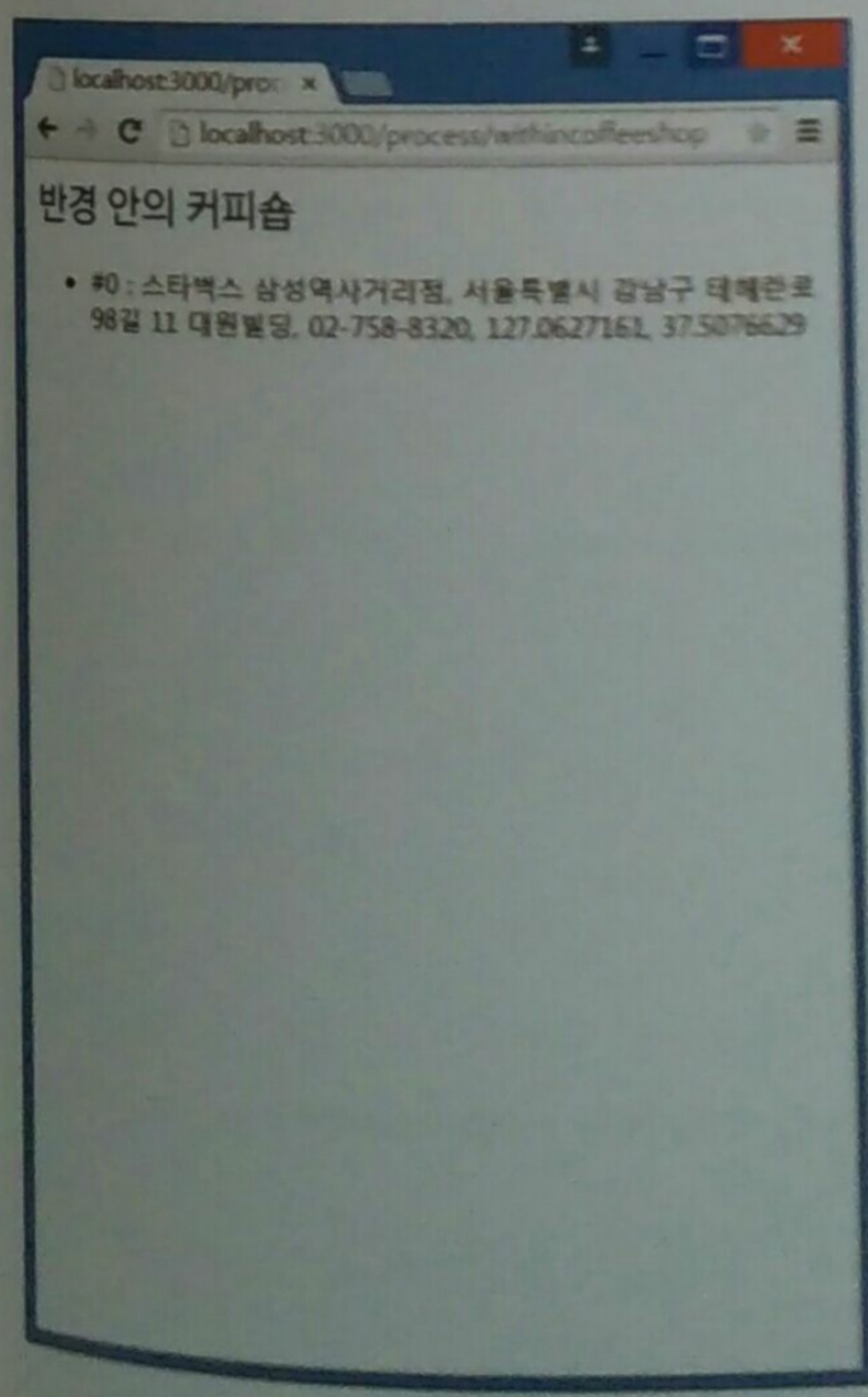
이제 app.js 파일을 다시 실행한 후 circlecoffeeshop.html 파일을 엽니다.

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



◀ 반경 안의 커피숍을 찾기 위해 circlecoffeeshop.html 파일을 연 경우

[전송] 버튼을 누르면 다음과 같이 일정한 반경으로 지정한 곳에 있는 커피숍을 찾아서 모두 보여 줍니다.



◀ 일정 반경 안의 커피숍을 찾아 보여 준 경우

12-5 | 지도에 커피숍의 위치 표시하기

지금까지는 사용자의 현재 위치를 이용해 주변 커피숍의 위치를 조회한 후 그 결과를 글자로 보여 주었습니다. 그런데 커피숍의 위치가 좌표 값으로 표시되어 정확히 어디에 있는지 알아보기 힘든 문제가 있습니다. 지도는 위치를 알려주는 가장 직관적인 방법이므로 커피숍의 위치를 글자보다는 지도로 보여 주는 것이 더 좋습니다. 이번에는 데이터베이스에서 조회한 커피숍의 위치를 지도 위에 표시해 보겠습니다.

지도에 내 위치 보여 주기

웹 페이지에 지도를 넣을 수 있는 방법은 여러 가지가 있지만 여기서는 구글맵(Google Map)을 사용합니다. 먼저 웹 페이지 안에 지도를 어떻게 넣을 수 있는지 알아보겠습니다. GeoExample 프로젝트의 [public] 폴더 안에 map.html 파일을 만들고 다음 코드를 입력합니다.

참조 파일 | GeoExample>/public/map.html

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

  <style>

    * {
        margin: 0;
        padding: 0;
    }

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

    #map {
        width: 100%;
        height: 95%;
    }
  </style>
```

```
</head>
<body onload="onLoad()">
  <p>구글맵으로 내 위치 표시하기</p>
  <div id="map"></div>
</body>
</html>
```

간단한 웹 문서이니 입력한 내용을 이해하는 것은 어렵지 않습니다. 입력한 코드를 보면 〈head〉 태그 안에 두 개의 〈meta〉 태그가 들어 있습니다. 첫 번째 태그는 인코딩을 설정해 주고, 두 번째 태그는 모바일 화면의 해상도에 맞게 페이지가 보이도록 설정합니다. 그 아래에 있는 〈style〉 태그에는 보통 웹 페이지를 만들 때 사용하는 CSS 속성이 몇 가지 들어 있습니다. 웹 페이지 안의 모든 태그에 대해 margin과 padding 값을 0으로 초기화하고 문서의 가로와 세로 크기를 100%로 맞추었습니다. 이것은 그 안에 넣을 지도의 크기를 비율 단위인 %로 조정하려고 추가한 것입니다.

이제 〈body〉 태그 안을 보면 〈p〉 태그와 함께 〈div〉 태그가 추가되어 있고 〈div〉 태그의 id 속성 값은 map으로 되어 있습니다. 이 〈div〉 태그는 지도를 나타낼 화면 영역을 할당받는 역할을 합니다. 이 태그의 id 속성 값이 map이므로 자바스크립트 코드에서 이 〈div〉 태그를 찾아 사용할 수 있습니다. 〈body〉 태그의 속성으로는 onload가 추가되어 있습니다. 이 속성을 추가하면 문서가 모두 로딩되었을 때 그 값으로 설정된 자바스크립트 코드가 실행됩니다.

onload 속성 값으로 넣은 자바스크립트 코드에서는 **onLoad 함수**를 호출하므로 〈head〉 태그 안에 〈script〉 태그를 입력한 후 다음 코드를 추가합니다.

참조 파일 | GeoExample>/public/map.html

```
.....

  <script src="https://maps.googleapis.com/maps/api/js"></script>

  <script>
    function onLoad() {
      initMap();
    }

    var map;
    var centerLocation = {lat: 37.5079523, lng: 127.0632954};

    function initMap() {
```

```
    map = new google.maps.Map(document.getElementById('map'), {
        center: centerLocation,
        zoom: 17
  });

  var marker = new google.maps.Marker({
    position:centerLocation,
    icon:'mylocation.png',
    animation:google.maps.Animation.BOUNCE
  });

    marker.setMap(map);
  }

 </script>

 ......
```

첫 번째 〈script〉 태그는 구글맵 라이브러리를 지정합니다. 즉, 다음 주소는 구글맵 라이브러리가 있는 주소입니다.

https://maps.googleapis.com/maps/api/js

정박사의 한마디 구글맵을 사용하려면 API 키가 있어야 해요

사용자에게 지도 서비스를 하려면 〈script〉 태그를 사용해 구글맵 라이브러리가 있는 주소를 지정할 때 다음과 같이 key 속성을 같이 넣어주어야 합니다. 여기에서 OOOOO으로 표시된 부분에는 API 키를 발급 받아 넣어주어야 합니다.

https://maps.googleapis.com/maps/api/js?key=OOOOOOOOOOOOOOOOOOOO

API 키를 어떻게 발급 받을 수 있는지 알고 싶다면 다음 사이트를 참조하기 바랍니다.

https://developers.google.com/maps/documentation/javascript/get-api-key?hl=ko

구글맵은 구글맵 서버로부터 지도 이미지를 받아오기 때문에 항상 인터넷에 연결된 상태여야 합니다. 그런데 구글맵 라이브러리가 완전히 로딩되기 전에는 구글맵 라이브러리의 기능을 사용할 수 없습니다. 따라서 문서가 모두 로딩된 후에 구글맵 지도를 초기화해야 합니다. 이 때문에 〈body〉 태그 안에 onload 속성을 추가하고 그 속성 값으로 onLoad 함수가 실행되도록 합니다. 그러면 이 onLoad 함수는 문서가 모두 로딩된 후에 호출됩니다.

onLoad 함수 안에서는 initMap 함수를 호출하도록 하고, 그 아래에 initMap 함수를 정의합니다. 구글맵은 google.maps.Map 프로토타입으로부터 객체를 만들어낼 수 있습니다. 따라서 new 연산자와 함께 google.maps.Map 프로토타입을 사용하면 새로운 구글맵 객체가 만들어집니다. 구글맵 객체를 만들 때는 두 개의 파라미터를 전달합니다. 첫 번째는 웹 페이지에 들어있는 문서 요소로, 여기에서는 id가 map인 〈div〉 태그를 찾아 전달합니다. 이렇게 하면 그 문서 요소에 지도가 표시됩니다. 두 번째는 지도를 초기화할 때 필요한 파라미터입니다. 지도 초기화를 위한 파라미터는 여러 가지가 있지만 여기에서는 center와 zoom 속성만 사용합니다. 파라미터들은 자바스크립트 객체로 만들어 전달되어야 하므로 중괄호를 입력한 후 그 안에 두 개의 속성을 추가합니다. 두 속성의 의미는 다음과 같습니다.

속성	설명
center	지도가 어느 지역을 보여줄 것인지를 좌표로 지정합니다. 자바스크립트 객체 안에 lat와 lng 속성으로 좌표 값을 넣어 줍니다. 예) {lat: 37.5079523, lng: 127.0632954}
zoom	지도 확대 수준을 지정합니다. 구글맵은 이미지를 받아오는 방식이기 때문에 확대 수준(zoom level)에 따라 서버에 이미지가 만들어져 있습니다. 값이 클수록 더 자세한 지도 이미지를 받아오게 됩니다.

[표] 지도 초기화를 위한 파라미터 중 center와 zoom

지도가 보일 지역의 좌표 값은 centerLocation 변수에 할당한 후 center 속성의 값으로 넣어 주었습니다. 그리고 확대 수준은 17로 설정했습니다. centerLocation의 값으로 설정한 위치 좌표는 지금까지 사용했던 삼성역 부근의 위치와 동일합니다.

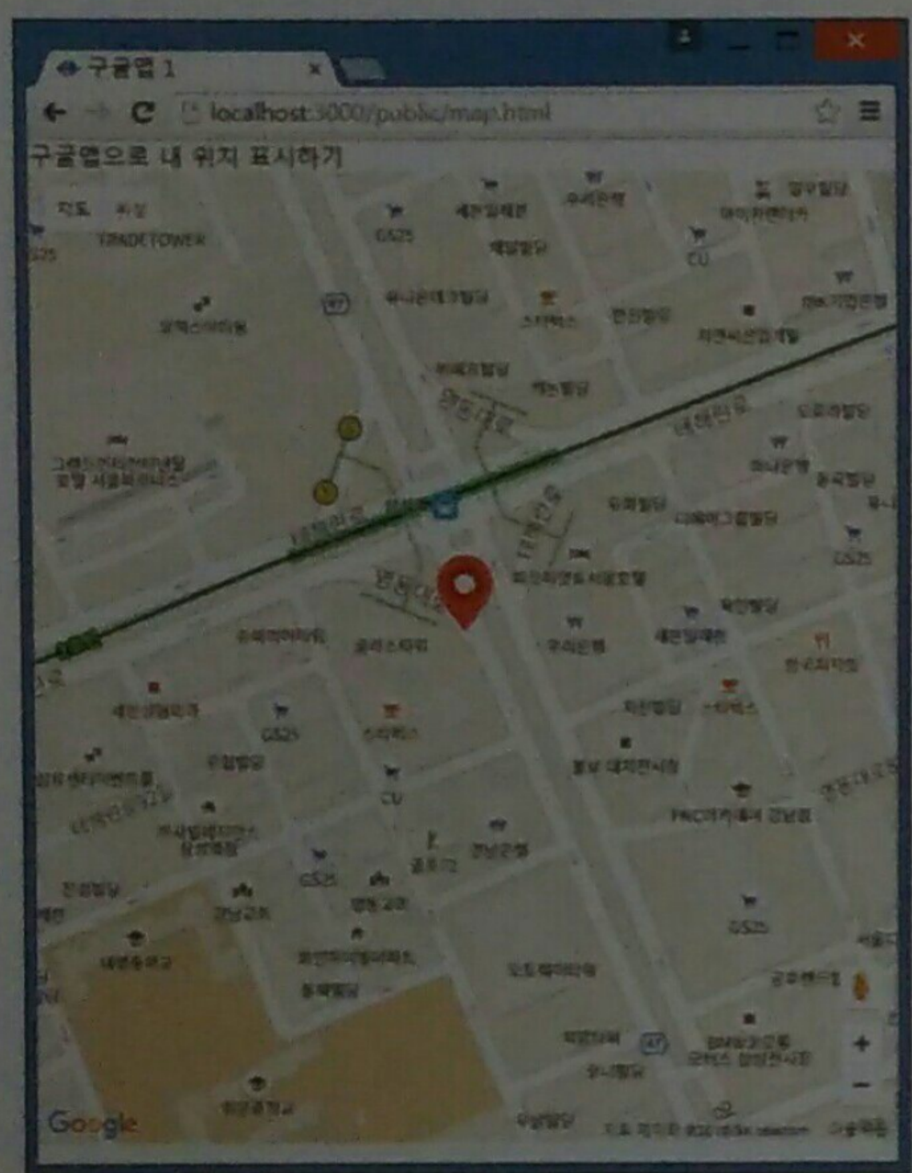
그런데 지도를 보여 줄 때 내 위치도 아이콘으로 함께 보여 주면 좀 더 명확하게 알려 줄 수 있습니다. 특정 위치를 아이콘으로 표시하기 위해서는 마커(Marker)를 사용할 수 있으며, google.maps.Marker 프로토타입을 이용해 객체를 만들면 됩니다. 마커 객체를 만들면 이 마커를 지도에 추가해야 하는데 마커 객체의 setMap 메소드를 호출하면 지도에 추가됩니다. setMap 메소드를 호출할 때는 지도 객체를 파라미터로 전달합니다.

마커 객체를 만들 때 자바스크립트 객체를 전달하면 그 안에 있는 속성으로 마커 객체를 초기화합니다. 여기에서는 position, icon, animation 속성이 사용되었습니다. position은 마커가 추가될 위치를 지정할 때 사용하고 icon은 아이콘 이미지를 지정하고 싶을 때, 그리고 animation은 애니메이션을 넣고 싶을 때 사용합니다. position 속성의 값으로 centerLocation 변수를 넣어 지도의 가운데 부분에 마커가 표시되도록 합니다. 그리고 마커 위치에 보여 줄 이미지는 mylocation.png라는 이름으로 지정합니다. 이 이미지 파일은 [public] 폴더에 넣어두면 됩니다. animation 속성을 넣어주면 눈에 더 잘 띄게 됩

니다. animation 속성의 값으로 google.maps.animation.BOUNCE를 지정하여 이미지가 통통 튀도록 만들어 줍니다.

서버를 실행한 후 웹 브라우저를 열고 아래 주소를 입력하면 다음과 같은 결과를 볼 수 있습니다.

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



▲ 웹 페이지에 지도를 추가하고 내 위치를 마커로 보여준 결과

지도 가운데 부분에 아이콘이 표시되고 그 아이콘이 통통 튀는 것을 볼 수 있습니다. 이제 내 위치를 좌표 값으로 보던 것보다 훨씬 쉽게 확인할 수 있게 되었습니다.

가장 가까운 커피숍을 찾아 지도 위에 보여 주기

12-2에서는 내가 있는 곳에서 가장 가까운 커피숍을 찾아 그 좌표 값을 결과 웹 페이지에 나타냈습니다. 이제 그 결과를 지도 위에 표시하도록 수정해 보겠습니다.

먼저 [public] 폴더 안에 있는 nearcoffeeshop.html 파일을 복사하여 nearcoffeeshop2.html 파일을 만들고 다음처럼 코드를 수정합니다.

참조 파일 | GeoExample>/public/nearcoffeeshop2.html

```
.....

  <form method="post" action="/process/nearcoffeeshop2">
    <table>
      <tr>
        <td><label>경도</label></td>
        <td><input type="text" name="longitude" value="127.0632954" /></td>
      </tr>
      <tr>
        <td><label>위도</label></td>
        <td><input type="text" name="latitude" value="37.5079523" /></td>
      </tr>
    </table>
    <input type="submit" value="전송" name="" />
  </form>
.....
```

<body> 태그 안에 있는 <h1> 태그의 글자는 '가까운 커피숍 찾아 지도로 보여 주기'처럼 적당한 글자로 변경합니다. 그리고 <form> 태그의 속성으로 들어 있는 action 속성의 값을 /process/nearcoffeeshop에서 /process/nearcoffeeshop2로 수정합니다. 이렇게 하면 [전송] 버튼을 클릭했을 때 서버에 nearcoffeeshop2라는 이름으로 등록된 함수가 실행됩니다.

이 웹 페이지에서 버튼을 클릭했을 때 실행되는 nearcoffeeshop2 함수를 등록하기 위해 [config] 폴더 안에 있는 config.js 파일을 엽니다. 그리고 다음 코드를 추가합니다.

참조 파일 | GeoExample>/config/config.js

```
.....

module.exports = {

.....

  route_info: [
.....,
      ,{file:'./coffeeshop', path:'/process/nearcoffeeshop2', method:'findNear2', type:'post'}
  ],

.....,
```

route_info 속성 값으로 들어 있는 배열 안에 새로운 객체를 하나 추가했습니다. 이것은 coffeeshop.js 파일 안에 들어 있는 findNear2 메소드를 사용해 /process/nearcoffeeshop2 패스로 들어온 요청을 처리하라는 설정입니다. 이제 [routes] 폴더 안에 들어 있는 coffeeshop.js 파일을 열고 findNear2 메소드를 추가합니다. findNear2 메소드는 기존에 만들었던 findNear 메소드의 기능 중에서 결과 웹 문서를 만들어 클라이언트로 응답하는 부분이 달라지므로 findNear를 복사하여 findNear2 메소드를 만듭니다. findNear2 메소드의 전체 코드는 다음과 같습니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
.....

var findNear2 = function(req, res) {
    console.log('coffeeshop 모듈 안에 있는 findNear2 호출됨.');

    var paramLongitude = req.param('longitude');
    var paramLatitude = req.param('latitude');
    var maxDistance = 1000;

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

    if (database.db) {
        // 1. 가까운 커피숍 검색
        database.CoffeeShopModel.findNear(paramLongitude, paramLatitude, maxDistance,
                    function(err, results) {
            if (err) { throw err; }

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

                if (results.length > 0) {
                    res.render('findnear.ejs', {result: results[0]._doc,
                                                paramLatitude: paramLatitude,
                                                paramLongitude: paramLongitude});
                } else {
                    res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
                    res.write('<h2>가까운 커피숍 데이터가 없습니다.</h2>');
                    res.end();
                }

            } else {
                res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
                res.write('<h2>가까운 커피숍 조회  실패</h2>');
                res.end();
```

```
            }
        });
    } else {
        res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
        res.write('<h2>데이터베이스 연결 실패</h2>');
        res.end();
    }

};

......
```

클라이언트로부터 latitude와 longitude 파라미터를 받아 데이터베이스에서 조회하는 코드 부분은 기존 findNear의 것과 같습니다. 하지만 CoffeeShopModel 객체의 findNear 메소드를 실행한 결과를 받아 처리하는 코드는 다음처럼 변경되었습니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
......
    if (results.length > 0) {
        res.render('findnear.ejs', {result: results[0]._doc,
                                    paramLatitude: paramLatitude,
                                    paramLongitude: paramLongitude});

......
```

이 코드에서는 응답 객체인 res의 render 메소드를 실행합니다. render 메소드는 뷰 템플릿을 이용해 응답 웹 문서를 만든 후 클라이언트로 응답을 보내 줍니다. 따라서 데이터베이스에서 조회한 결과가 하나라도 있다면 findnear.ejs 뷰 템플릿을 사용해 결과 웹 문서를 만든 후 클라이언트로 응답을 보내게 됩니다.

이렇게 만든 findNear2 메소드는 module.exports.findNear2 속성에 할당되도록 다음과 같이 coffeeshop.js 파일의 가장 아래쪽에 입력합니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
......

module.exports.findNear2 = findNear2;
```

findnear.ejs 파일을 사용해 응답 웹 문서를 만들 때 전달되는 자바스크립트 객체에는 응답 결과 객체와 함께 조회에 사용된 paramLatitude, paramLongitude 속성이 들어 있습니다. 이 속성들이 ejs 파일로 전달될 것이므로 확인한 후 [views] 폴더 안에 findnear.ejs 파일을 만듭니다. ejs 파일에는 웹 문서가 그대로 들어갈 수 있고 ejs 파일로 전달된 변수를 사용해 값을 바꿔 줄 부분만 〈%= %〉 기호를 사용해서 수정하면 되므로 map.html 파일의 내용을 그대로 findnear.ejs 파일에 입력합니다. 그리고 〈script〉 태그 안에 들어가는 코드를 다음처럼 수정합니다.

참조 파일 | GeoExample>/views/findnear.ejs

```
.....

  <script>
    function onLoad() {
      initMap();
    }

    var map;
    var centerLocation = {lat: <%= paramLatitude %>, lng: <%= paramLongitude %>};
    var coffeeLocation =
      {lat: <%= result.geometry.coordinates[1] %>, lng: <%= result.geometry.coordinates[0] %>};

    function initMap() {

      map = new google.maps.Map(document.getElementById('map'), {
        center: centerLocation,
        zoom: 17
      });

      var myMarker = new google.maps.Marker({
        position:centerLocation,
        icon:'/public/mylocation.png',
        animation:google.maps.Animation.BOUNCE
      });

      myMarker.setMap(map);

      var coffeeMarker = new google.maps.Marker({
        position:coffeeLocation,
        icon:'/public/coffee.png'
      });

      coffeeMarker.setMap(map);
```

```
}

</script>

......
```

centerLocation은 내 위치의 좌표 값을 담고 있어야 하므로 paramLatitude와 paramLongitude 변수 값을 할당합니다. 그리고 coffeeLocation 변수에는 데이터베이스에서 조회한 커피숍의 위치 좌표를 할당합니다. 데이터베이스에서 조회한 결과는 result 변수에 할당된 후 findnear.ejs 파일로 전달되므로 lat 속성 값은 result.geometry.coordinates[1], lng 속성 값은 result.geometry.coordinates[0]으로 설정합니다. 코드의 아래쪽을 보면 coffeeLocation 변수에 할당된 커피숍의 위치 좌표를 사용해 coffeeMarker라는 이름의 마커를 하나 생성한 후 지도에 추가한 것을 알 수 있습니다. 이때 [public] 폴더 안에 있는 coffee.png 이미지를 사용하도록 했으므로 [public] 폴더 안에 coffee.png라는 이름의 커피숍 이미지를 하나 넣어두어야 합니다.

지금까지 추가한 기능을 순서대로 정리하면 다음과 같습니다.

❶ 사용자가 조회할 웹 문서인 nearcoffeeshop2.html 파일을 만듭니다.

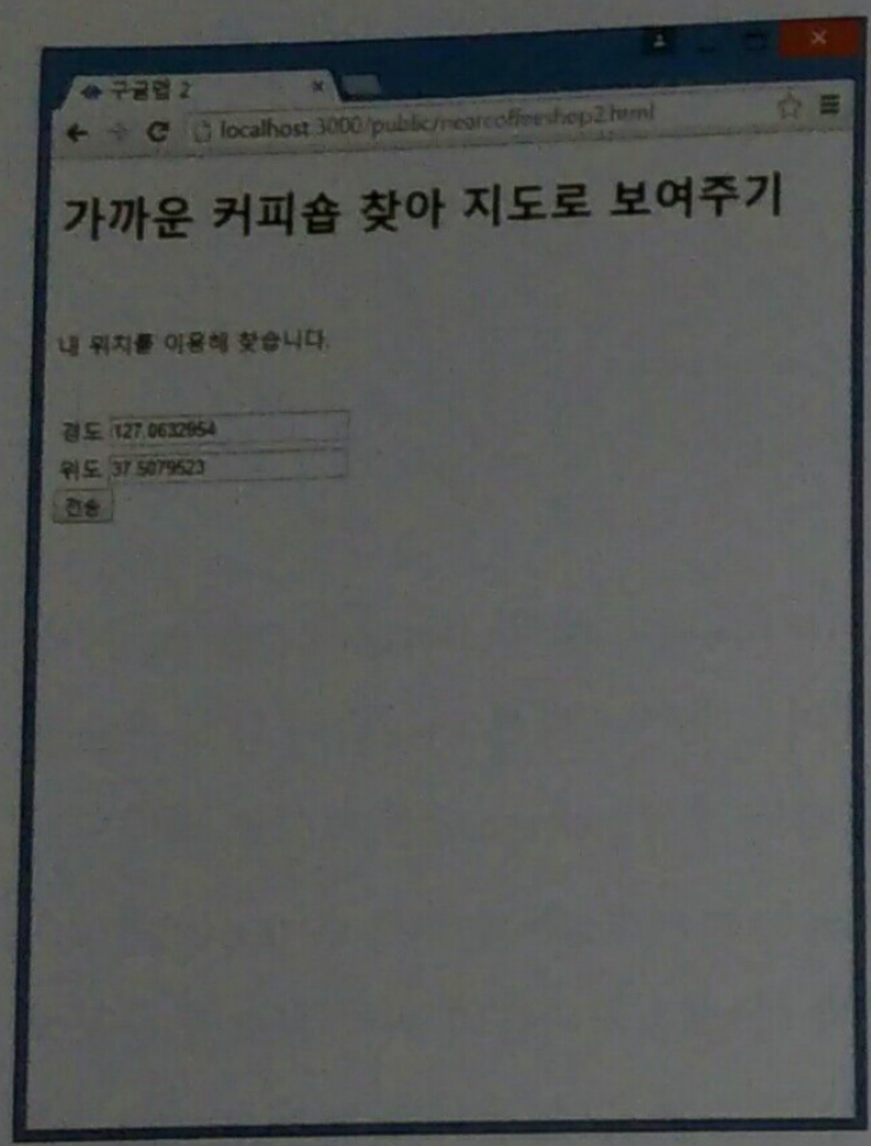
❷ 웹 문서의 버튼을 눌렀을 때 요청할 findNear2 메소드를 config.js 파일에 등록합니다.

❸ coffeeshop.js 파일 안에 findNear2 메소드를 추가하고, 결과 웹 문서를 만들 때 findnear.ejs 뷰 템플릿을 사용하도록 코드를 수정합니다.

❹ findnear.ejs 파일을 만들고 findNear2 메소드로부터 전달된 데이터를 사용해 지도 위에 마커를 표시합니다.

결과 웹 문서에 지도를 추가하고 지도 위에 내 위치와 커피숍 위치를 표시하도록 했으므로 이제 정상적으로 기능이 동작하는지 확인할 차례입니다. app.js 파일을 실행하여 서버를 시작한 후 웹 브라우저를 열고 아래 주소로 접속합니다.

▶ http://localhost:3000/public/nearcoffeeshop2.html



▲ 가까운 커피숍 조회를 요청하기 위한 웹 페이지를 연 경우

[전송] 버튼을 누르면 서버에서 데이터베이스에 들어있는 커피숍 중에서 가장 가까운 것을 찾아 보여 줍니다. [전송] 버튼을 누르면 다음과 같이 데이터베이스에서 조회한 커피숍의 위치가 지도에 표시됩니다.

▲ 지도와 함께 조회한 커피숍의 위치가 표시된 결과 웹 문서

내 위치 옆에 커피숍 아이콘이 표시되었습니다. 조회 결과를 지도로 볼 수 있으니 이제 훨씬 쉽게 커피숍의 위치를 파악할 수 있게 되었습니다.

일정 범위 안의 커피숍을 찾아 지도 위에 보여주기

가장 가까운 커피숍을 찾아 지도에 표시하는 기능을 만들어 보았으니 12-3에서 만들었던 기능을 수정하는 것도 어렵지 않을 것입니다. 12-3에서는 일정 범위 안의 커피숍을 찾아 좌표 값을 결과물로 보여 주었는데 이것을 지도에 표시되도록 바꿔 보겠습니다. 이번에 진행할 과정은 가까운 커피숍을 찾아 지도에 보여 주기 위해 변경했던 과정과 거의 같습니다.

웹 페이지부터 하나씩 수정해 보겠습니다. 먼저 [public] 폴더 안에 있는 withincoffeeshop.html 파일을 복사하여 withincoffeeshop2.html 파일을 만들고 다음과 같이 코드를 수정합니다.

참조 파일 | GeoExample>/public/withincoffeeshop2.html

```
……

<form method="post" action="/process/withincoffeeshop2">
  <table>
    <tr>
      <td><label>경도</label></td>
      <td><input type="text" name="longitude" value="127.0632954" /></td>
    </tr>
    <tr>
      <td><label>위도</label></td>
      <td><input type="text" name="latitude" value="37.5079523" /></td>
    </tr>

……
```

〈body〉 태그 안에 있는 〈h1〉 태그의 글자는 '영역 안의 커피숍 찾아 지도로 보여 주기'처럼 적당한 글자로 변경합니다. 그리고 〈form〉 태그의 속성으로 들어 있는 action 속성 값을 /process/within coffeeshop에서 /process/withincoffeeshop2로 수정합니다. 이렇게 하면 [전송] 버튼을 클릭했을 때 서버에 withincoffeeshop2라는 이름으로 등록된 함수가 실행됩니다. 사용자가 버튼을 클릭했을 때 서버로 전송되는 파라미터는 사각형 영역을 만들기 위한 좌표 값과 함께 내 현재 위치 좌표 값을 전달합니다.

이 웹 페이지에서 버튼을 클릭했을 때 실행되는 withincoffeeshop2 함수를 등록하기 위해 [config] 폴더 안에 있는 config.js 파일을 엽니다. 그리고 다음 코드를 추가합니다.

참조 파일 | GeoExample>/config/config.js

```
……

module.exports = {

……

    route_info: [
……
      ,{file:'./coffeeshop', path:'/process/withincoffeeshop2', method:'findWithin2', type:'post'}
    ],

……
```

route_info 속성 값으로 들어 있는 배열 안에 새로운 객체를 하나 추가했습니다. 이것은 coffeeshop.js 파일 안에 들어 있는 findWithin2 메소드를 사용해 /process/withincoffeeshop2 패스로 들어온 요청을 처리하라는 설정입니다. 이제 [routes] 폴더 안에 들어있는 coffeeshop.js 파일을 열고 findWithin2 메소드를 추가합니다. findWithin2 메소드는 기존에 만들었던 findWithin 메소드의 기능 중에서 결과 웹 문서를 만들어 클라이언트로 응답하는 부분이 달라지므로 findWithin을 복사하여 findWithin2 메소드를 만듭니다. findWithin2 메소드의 코드는 다음과 같습니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
……

var findWithin2 = function(req, res) {
  console.log('coffeeshop 모듈 안에 있는 findWithin2 호출됨.');

  var paramLongitude = req.param('longitude');
  var paramLatitude = req.param('latitude');

  var paramTopLeftLongitude = req.param('topleft_longitude');
  var paramTopLeftLatitude = req.param('topleft_latitude');
  var paramBottomRightLongitude = req.param('bottomright_longitude');
  var paramBottomRightLatitude = req.param('bottomright_latitude');

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

  if (database.db) {
      // 1. 가까운 커피숍 검색
```

```
database.CoffeeShopModel.findWithin(paramTopLeftLongitude, paramTopLeftLatitude,
                paramBottomRightLongitude, paramBottomRightLatitude, function(err, results)
{
    if (err) { throw err; }

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

      if (results.length > 0) {
        res.render('findwithin.ejs', {result: results[0]._doc,
                        paramLongitude: paramLongitude,
                        paramLatitude: paramLatitude,
                        paramTopLeftLongitude: paramTopLeftLongitude,
                        paramTopLeftLatitude: paramTopLeftLatitude,
                        paramBottomRightLongitude: paramBottomRightLongitude,
                        paramBottomRightLatitude: paramBottomRightLatitude});
      } else {
        res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
        res.write('<h2>영역 안에 커피숍 데이터가 없습니다.</h2>');
        res.end();
      }

    } else {
        res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
        res.write('<h2>영역 내 커피숍 조회 실패</h2>');
        res.end();
    }
  });
} else {
  res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
  res.write('<h2>데이터베이스 연결 실패</h2>');
  res.end();
}

};



......
```

클라이언트로부터 파라미터를 받아 데이터베이스에서 조회하는 코드 부분은 기존 findWithin의 것과 같습니다. 하지만 CoffeeShopModel 객체의 findWithin 메소드를 실행한 결과를 받아 처리하는 코드는 다음과 같이 변경되었습니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
……
  if (results.length > 0) {
    res.render('findwithin.ejs', {result: results[0]._doc,
      paramLongitude: paramLongitude,
      paramLatitude: paramLatitude,
      paramTopLeftLongitude: paramTopLeftLongitude,
      paramTopLeftLatitude: paramTopLeftLatitude,
      paramBottomRightLongitude: paramBottomRightLongitude,
      paramBottomRightLatitude: paramBottomRightLatitude});

……
```

이 코드에서는 응답 객체인 res의 render 메소드를 실행합니다. render 메소드는 뷰 템플릿으로 응답 웹 문서를 만든 후 클라이언트로 응답을 보내 줍니다. 따라서 데이터베이스에서 조회한 결과가 하나라도 있다면 findwithin.ejs 뷰 템플릿으로 결과 웹 문서를 만든 후 클라이언트로 응답을 보내게 됩니다.

이렇게 만든 findWithin2 메소드는 module.exports.findWithin2 속성에 할당되도록 coffeeshop.js 파일의 가장 아래쪽에 다음과 같이 입력합니다.

참조 파일 | GeoExample>/routes/coffeeshop.js

```
……

module.exports.findWithin2 = findWithin2;
```

findwithin.ejs 파일을 사용해 응답 웹 문서를 만들 때 전달되는 자바스크립트 객체에는 응답 결과 객체와 함께 조회에 사용된 파라미터들이 속성으로 들어 있습니다. 이제 [views] 폴더 안에 findwithin.ejs 파일을 만듭니다. 이 파일에는 지도를 보여 줄 수 있는 코드를 그대로 입력합니다. 그리고 〈script〉 태그 안에 들어가는 코드를 다음과 같이 수정합니다.

참조 파일 | GeoExample>/views/findwithin.ejs

```
……

  <script>
    function onLoad() {
     initMap();
```

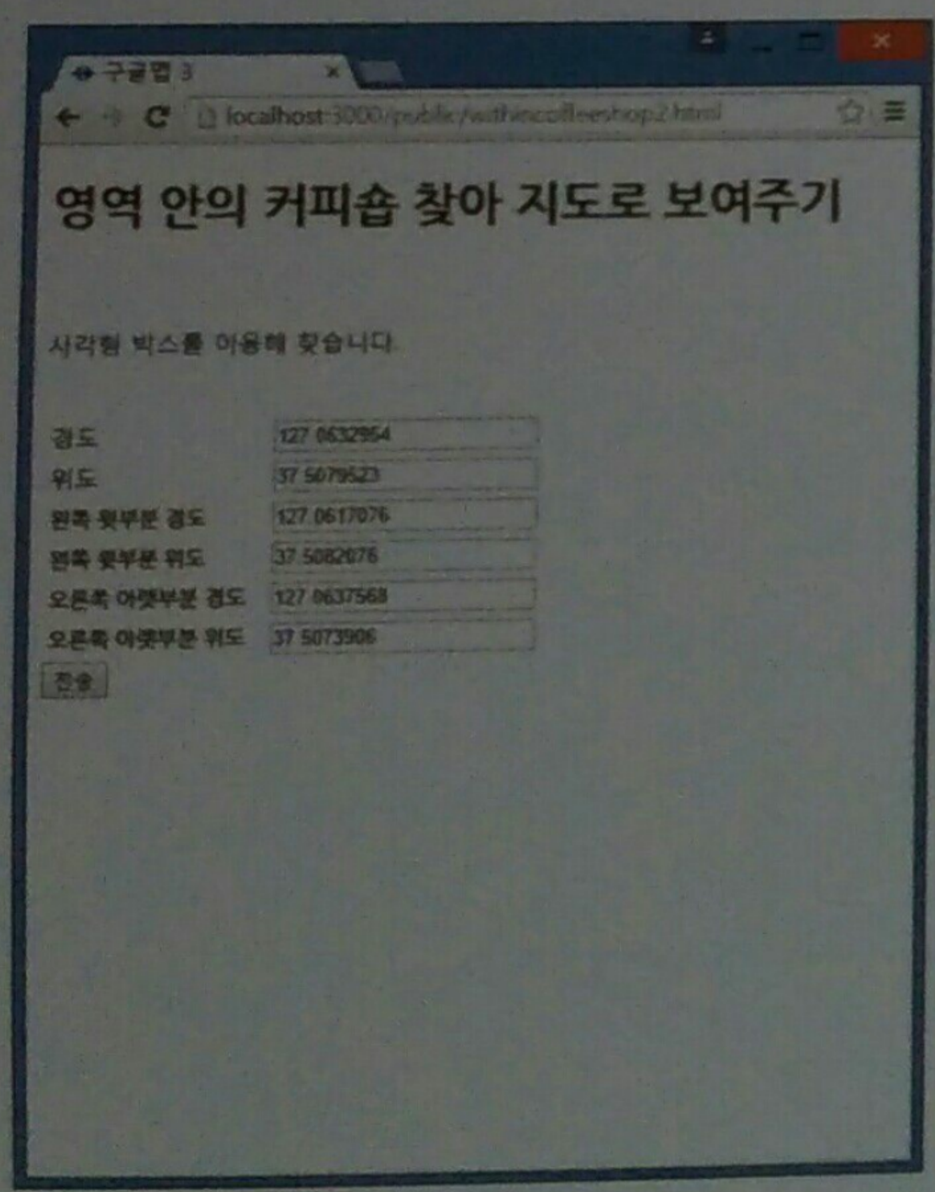
```
}

var map;

// 내 위치를 위한 좌표
var centerLocation = {lat: <%= paramLatitude %>, lng: <%= paramLongitude %>};

// 커피숍 위치를 위한 좌표
var coffeeLocation =
  {lat: <%= result.geometry.coordinates[1] %>, lng: <%= result.geometry.coordinates[0] %>};

// 사각형 검색 영역을 위한 좌표
var coords =new google.maps.LatLngBounds(
  {lat:<%= paramTopLeftLatitude %>, lng:<%= paramTopLeftLongitude %>},
  {lat:<%= paramBottomRightLatitude %>, lng:<%= paramBottomRightLongitude %>}
);

function initMap() {

    map = new google.maps.Map(document.getElementById('map'), {
     center: centerLocation,
     zoom: 17
    });

    // 내 위치 표시
    var myMarker = new google.maps.Marker({
      position:centerLocation,
      icon:'/public/mylocation.png',
      animation:google.maps.Animation.BOUNCE
     });
     myMarker.setMap(map);

    // 커피숍 위치 표시
    var coffeeMarker = new google.maps.Marker({
      position:coffeeLocation,
      icon:'/public/coffee.png'
     });
     coffeeMarker.setMap(map);

    // 검색하려는 영역 표시
    var rectangle = new google.maps.Rectangle({
      bounds:coords,
      strokeColor:"#0000FF",
      strokeOpacity:0.8,
```

```
        strokeWeight:2,
        fillColor:"#0000FF",
        fillOpacity:0.4
      });

      rectangle.setMap(map);
    }

</script>

……
```

findnear.ejs 파일의 내용과 비교해 달라진 점은 검색을 위해 사용했던 사각형 영역도 지도 위에 표시해 준다는 것입니다. coords 변수에는 사각형 영역을 표시하기 위한 좌표가 들어 있으며, 이를 사용해 Rectangle 객체를 만들었습니다. google.maps.Rectangle 프로토타입으로 사각형 객체를 만들면 지도에 추가할 수 있습니다. 사각형을 만들 때는 strokeColor, fillColor 등의 속성을 파라미터로 전달되는 객체에 넣어 파란색 사각형으로 보이도록 했습니다.

결과 웹 문서에 지도를 추가하고 지도 위에 내 위치와 커피숍 위치, 그리고 사각형 영역을 모두 표시하도록 설정했으므로 정상적으로 기능이 동작하는지 확인해 보겠습니다. app.js 파일을 실행하여 서버를 시작한 후 웹 브라우저를 열고 아래 주소로 접속합니다.

▶ http://localhost:3000/public/withincoffeeshop2.html



▲ 영역 안의 커피숍 조회를 요청하는 웹 페이지

[전송] 버튼을 누르면 서버에서 데이터베이스에 들어 있는 커피숍 중에서 영역 안에 포함된 것을 찾아 보여줍니다. [전송] 버튼을 누르면 다음과 같이 조회한 커피숍의 위치가 지도에 표시됩니다.

▲ 지도와 함께 조회한 커피숍의 위치, 그리고 사각형 영역이 표시된 결과 웹 문서

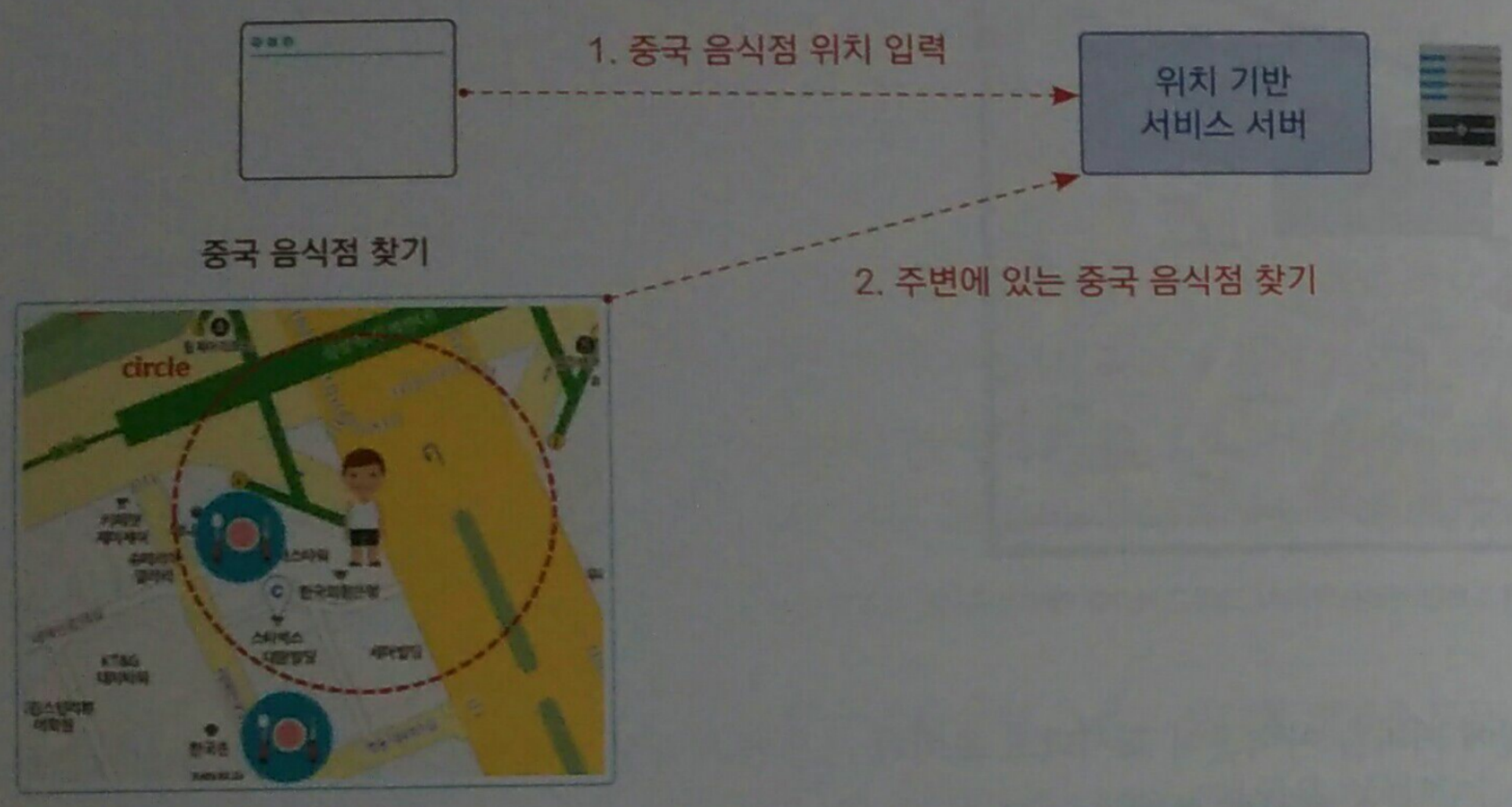
내 위치 옆에 커피숍 아이콘이 표시되고 조회에 사용된 사각형 영역까지 보입니다. 조회에 사용된 사각형 영역까지 눈으로 확인할 수 있으니 더 쉽게 이해됩니다.

지금까지 커피숍의 위치를 웹 서버에 보내 데이터베이스에 저장한 후 사용자의 현재 위치를 사용해 조회하는 방법까지 살펴보았습니다. 그리고 조회한 결과물을 지도 위에 표시하는 방법도 알아보았습니다. 스마트폰이나 사물인터넷(IoT) 단말이 많아지면서 위치 정보로 대상을 검색하거나 화면에 보여 주는 경우가 점점 많아지고 있습니다. 따라서 위치 좌표를 서버에 저장했다가 필요할 때 조회하여 화면에 보여 주는 기능은 잘 알아두는 것이 좋습니다. 이 장에서 여러분과 함께 만들어 본 위치 기반 서비스 서버를 사용하면 다양한 곳에 흩어져 있는 대상을 위치 정보로 빠르게 찾을 수도 있고 우버(Uber)와 같은 앱을 만들 수도 있다는 것을 기억하기 바랍니다.

Do it!
도전 문제

01. 이 장에서 사용한 지도 위의 한 지점을 중심으로 중국 음식점 몇 곳의 위치를 입력하여 서버에 저장한 후 주변에 있는 중국 음식점들을 찾아 주는 기능을 만들어 보세요.

프로젝트 소스 Mission17 난이도 초급 ★☆☆☆



Hint!

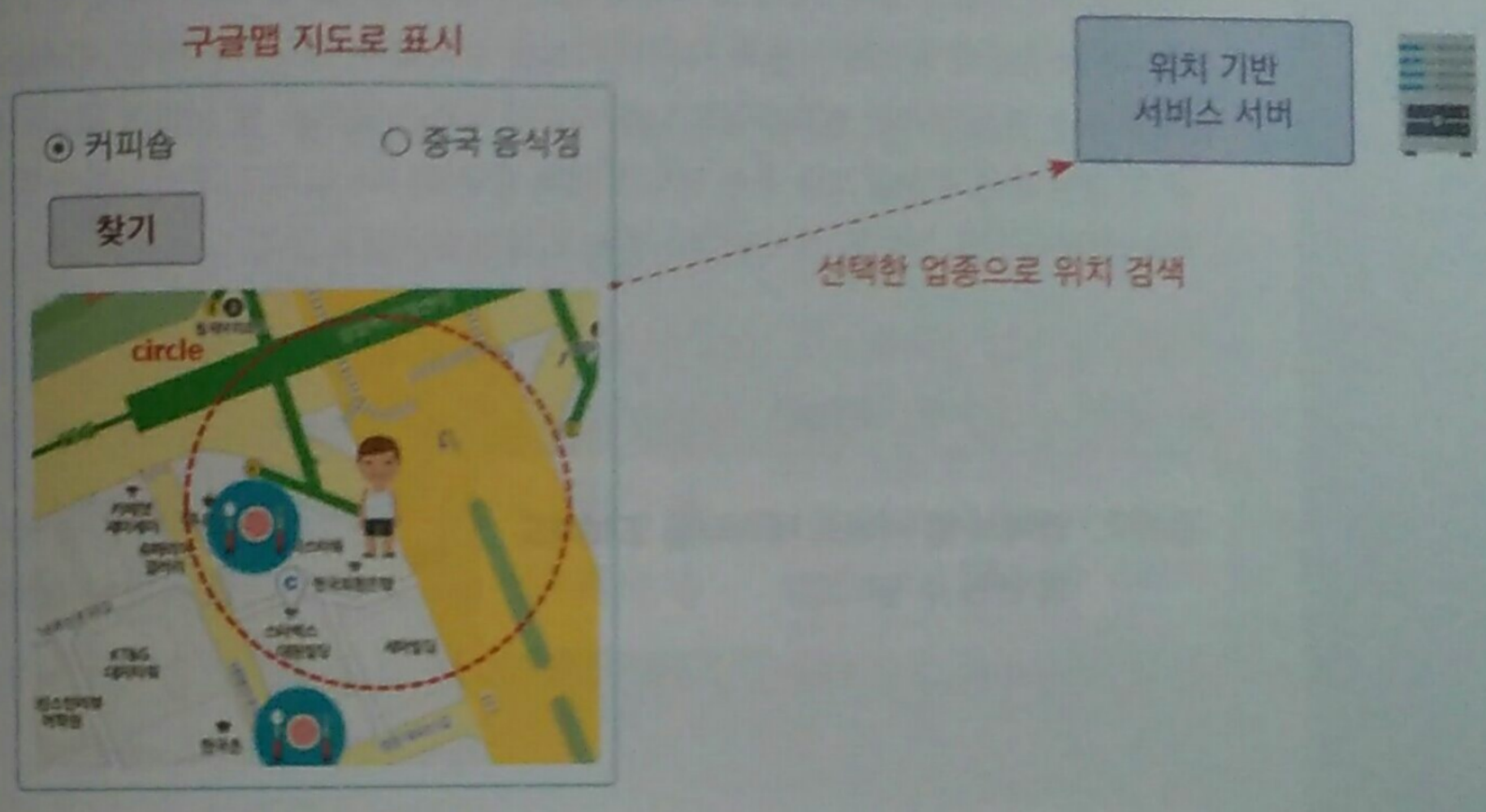
1. 커피숍 위치를 입력한 웹 문서를 사용해서 중국 음식점의 위치를 입력하도록 만듭니다.
2. 데이터베이스 스키마에는 커피숍의 위치 정보를 입력하므로 중국 음식점의 위치도 저장할 수 있도록 만듭니다. 즉, 커피숍인지 중국 음식점인지를 구별할 수 있도록 설정합니다.
3. 데이터베이스에 저장한 후 정상적으로 저장되었다는 응답 메시지를 클라이언트로 보냅니다.
4. 사용자의 위치를 보내면 주변에 있는 중국 음식점을 검색하여 응답으로 보내 줍니다.

MEMO

DO it!
도전 문제

02. 구글맵 지도를 사용해서 웹 문서에 사용자 위치를 보여 준 후 커피숍과 중국 음식점 중 어느 것을 찾을지 선택한 다음 주변 지점 정보를 찾아서 보여 주는 기능을 만들어 보세요.

프로젝트 소스 Mission18 난이도 중급 ★★☆☆



Hint!

1 웹 문서에 구글맵 지도를 띄운 후 사용자 위치를 보여 줍니다.

2 사용자가 커피숍과 중국 음식점 중 하나를 선택하도록 만들고, [찾기] 버튼을 누르면 선택한 업종을 주변에서 찾아 보여 줍니다.

3 데이터베이스에는 주변에 있는 몇 개의 커피숍 위치와 몇 개의 중국 음식점 위치가 들어 있어야 합니다.

MEMO