



KWEB, Korea University

# Web Application Essentials

Backend Section

Written By: Lee, Chanyoung

October 11, 2022



# Table of Contents

<b>0</b>	<b>Preface</b>	<b>7</b>
0.1	Introduction to WEB . . . . .	8
<b>1</b>	<b>Introduction to Backend</b>	<b>9</b>
1.1	HyperText Transfer Protocol (HTTP) . . . . .	10
1.2	Node.js Platform . . . . .	14
1.3	git and GitHub . . . . .	16
<b>2</b>	<b>Advanced Javascript</b>	<b>21</b>
2.1	Advanced Javascript Syntaxes . . . . .	22
2.2	Modules . . . . .	27
2.3	Asynchronous Programming . . . . .	30
2.4	Advanced Javascript Exercises . . . . .	34
<b>3</b>	<b>Express.js and View Engine</b>	<b>37</b>
3.1	Express.js . . . . .	38
3.2	View Engine . . . . .	43
3.3	Sending POST Request From Browser . . . . .	47
3.4	Express.js and View Engine Exercises . . . . .	48
<b>4</b>	<b>Database Designing</b>	<b>51</b>
4.1	Introduction to Database . . . . .	52

4.2	MariaDB . . . . .	54
4.3	Basics of Designing . . . . .	59
4.4	Relational Designing . . . . .	63
4.5	Database Designing Exercises . . . . .	67
<b>5</b>	<b>Database Querying</b>	<b>69</b>
5.1	CRUD Functions . . . . .	70
5.2	Advanced Querying . . . . .	73
5.3	Join Operations . . . . .	76
5.4	Querying in Node.js . . . . .	82
5.5	Database Querying Exercises . . . . .	86
<b>6</b>	<b>Authentication</b>	<b>89</b>
6.1	Hash Functions and Encryption . . . . .	90
6.2	HTTP Cookie and Session . . . . .	95
<b>7</b>	<b>Web Application Implementation</b>	<b>101</b>
7.1	Application Summary and Structure . . . . .	102
7.2	Database Design and DAO Implementation . . . . .	106
7.3	Routing and Controller Implementation . . . . .	109
7.4	Deploying Web Application . . . . .	116
<b>A</b>	<b>Source Codes</b>	<b>121</b>
A.1	Database Querying Source Codes . . . . .	122
A.2	Web Application Implementation Source Codes . . . . .	126
<b>B</b>	<b>Exercise Answers</b>	<b>135</b>
B.1	Advanced Javascript Exercise Answers . . . . .	136
B.2	Express.js and View Engine Exercise Answers . . . . .	138
B.3	Database Designing Exercise Answers . . . . .	141

B.4 Database Querying Exercise Answers . . . . .	143
--------------------------------------------------	-----



# Chapter 0

# Preface

## Contents

0.1	Introduction to WEB . . . . .	8
-----	-------------------------------	---

## 0.1 Introduction to WEB

### WEB

웹(WEB)이란, 인터넷에 연결된 컴퓨터들을 통해 사람들이 정보를 공유할 수 있는 전 세계적인 정보 공간이다. 웹을 통해 공유되는 정보들은 대개 웹 페이지(web page)의 형태로 공유되며, 웹 페이지는 특수한 양식을 갖춘 텍스트로 구성된다. 각 웹 페이지는 일반적인 텍스트(plain text)뿐만 아니라 이미지, 동영상, 다른 웹 페이지로 연결되는 하이퍼링크(hyperlink) 등의 웹 자원(web resource)으로 다양하게 구성되고, 하나의 주제, 하나의 영역을 공유하는 여러 웹 자원과 웹 페이지는 웹 사이트(website)를 구성한다. 웹 사이트는 웹 서버(web server)라는 디바이스에서 동작하는 웹 애플리케이션(web application)이라고 하는 프로그램의 형태로 구현되고 작동하며, 웹 애플리케이션을 통해 특수한 제한이 없다면 전 세계 어디서든 웹 사이트에 접속할 수 있다.

본 교재에서 다루는 웹의 영역은 흔히 웹 개발자가 다루는 웹의 영역과 밀접하게 관련이 있는 웹 애플리케이션과 관련된 영역이다. 앞으로 여러분은 이 교재를 통해 웹 애플리케이션의 작동 원리와 구조를 이해하고, 웹을 디자인하고 설계하는 학습을 할 것이다.

### Frontend and Backend

우리가 일상에서 사용하는 웹 사이트, 웹 애플리케이션이 어떻게 작동할 지 상상해보자. 예를 들어, 사용자가 블로그 형태의 웹 사이트에 접속했을 때 사용자에게 보여지는 영역에는 어떤 것이 있는가? 블로그의 블로거, 블로거에 대한 상세 정보, 게시물, 게시물에 대한 상세 정보, 댓글, 댓글에 대한 상세 정보 등이 있을 것이다. 뿐만 아니라 블로그를 예쁘게 꾸민 디자인 등도 사용자에게 보여지는 영역이다. 이렇듯 사용자에게 보여지는 영역을 **frontend**라고 부르며, 사용자(client) 쪽에서 동작하는 영역이라고 하여 **client-side**라고도 한다.

반대로, 사용자에게 직접적으로 보이지 않는 영역도 있다. 사용자의 요청에 맞게 적절한 웹 페이지를 구성하여 전달해주는 로직, 게시물이나 댓글, 블로거의 정보를 저장하고 읽어오는 기능, 회원에 따라 글을 작성할 권한을 부여할지, 댓글을 수정할 권한을 부여할지 결정하는 로직 등이 있을 것이다. 이렇게 사용자에게 직접적으로 보이지 않는 영역을 **backend**라고 부르고, 이러한 로직은 서버상에서 동작하기 때문에 **server-side**라고도 한다.

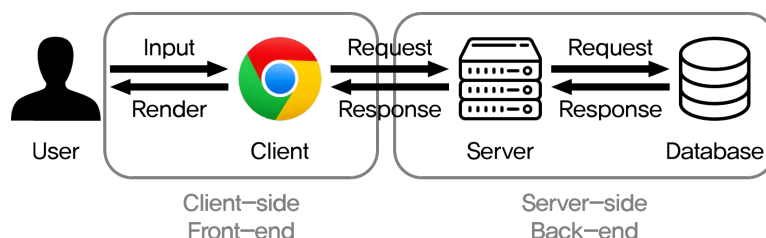


Figure 1 How Typical Web Application Works

본 교재에서는 backend 분야에서 필요한 기초 개념들과 사용되는 각종 기술 스택들을 다루며, Node.js에서 작동하는 Express.js 프레임워크와 데이터베이스를 이용하여 간단한 웹 애플리케이션을 제작하는 실습을 진행한다.



# Chapter 1

## Introduction to Backend

### Contents

1.1	HyperText Transfer Protocol (HTTP) . . . . .	10
1.2	Node.js Platform . . . . .	14
1.3	git and GitHub . . . . .	16

## 1.1 HyperText Transfer Protocol (HTTP)

### What is HTTP?

0.1절에서 다룬 바와 같이 웹 애플리케이션은 클라이언트와 서버가 요청과 응답을 통해 데이터를 교환하는 애플리케이션이다. 이때 요청과 응답에 의해 교환되는 데이터에는 발신 및 수신 IP와 같은 요청과 응답에 관한 여러 정보가 포함되어 있어야 하는데, 이러한 정보를 작성하는 방식이 개발 주체에 따라 다를 경우 클라이언트가 보낸 요청을 서버가 해석하지 못하는 등의 혼란이 발생한다. 이를 방지하기 위해 웹 애플리케이션에서 클라이언트와 서버 간의 통신에서 준수해야 하는 표준 규약(protocol)이 존재하며, HTTP(HyperText Transfer Protocol)는 웹 페이지를 전송하기 위해 사용되는 표준 규약이다.

HTTP는 헤드와 본문으로 이루어져 있고, 헤드(HTTP head)는 요청/응답 라인(request/response line)과 헤더(HTTP header)로 이루어져 있다. 요청/응답 라인에는 요청/응답에 대한 간단한 정보가 한 줄로 기록되어 있고, 헤더에는 요청이나 응답에 대한 상세한 정보나 옵션이 포함된다. 본문(HTTP body)에는 전송하고자 하는 데이터가 포함되어 있으며, 요청에서는 요청에 대한 여러 조건과 값, 응답에서는 웹 페이지의 HTML 문서 등이 포함되어 있다.

### HTTP Request

HTTP 요청(request)은 클라이언트가 서버에 전송하는 메시지이다.

#### Code 1.1 Head of HTTP Request Example

```
GET /api/articles?bid=12&page=3&count=10 HTTP/1.1
Host: kweb.korea.ac.kr           route이다.
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Firefox/79.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: sid=s%3A_2uUdS9bmos5wPmQmMuHvVXHrcdIYHmk.%2F3FX1NwUkH5fFK3cd7gDr8qfMHsrvfMWGS0
```

HTTP 요청의 헤더는 **Code 1.1**과 같은 형태로 구성되어 있다. 요청 라인에는 차례대로 요청 메서드(GET), URL(/api/articles?bid=12&page=3&count=10), HTTP 버전(HTTP/1.1)이 차례대로 나열된다. 헤더에는 Host, User-Agent, Cookie등의 여러 정보와 그에 대한 값으로 구성되어 있다. User-Agent는 사용자가 요청을 보내기 위해 사용한 애플리케이션에 대한 정보를 제시하며, 예시에서의 요청은 Mozilla Firefox 브라우저에서 보낸 요청이다. HTTP 요청에 열거되는 중요한 요소 중 하나인 쿠키(cookie)는 6.2절에서 다룬다.

### HTTP Method

HTTP 메서드(method)는 서버가 수행해야 할 동작의 종류를 나타내는 요소이다. 가장 흔히 쓰이는 메서드는 GET과 POST이고, RESTful하게 구현된 애플리케이션에서는 PUT과 DELETE가 추가로 사용되며, 이 외에도

HEAD, OPTIONS, PATCH 등의 메서드가 사용되기도 한다. 가장 많이 사용되는 네 메서드는 각각 다음과 같은 동작을 명시한다.

- GET: 서버에 존재하는 리소스에 대한 열람(read)을 요청
- POST: 서버에 특정 리소스를 새로 생성(create)할 것을 요청
- PUT: 서버에 존재하는 특정 리소스를 수정(update)할 것을 요청
- DELETE: 서버에 존재하는 특정 리소스를 삭제(delete)할 것을 요청

예를 들어 서버에 저장되어 있는 회원들에 대한 데이터에 관한 요청을 보낼 때, 특정 회원 또는 여러 회원들에 대한 데이터를 열람하고자 할 때는 GET, 새로 가입한 회원의 정보를 추가하고자 할 때는 POST, 이미 존재하는 회원의 정보를 수정하고자 할 때는 PUT, 특정 회원이 사이트를 탈퇴하여 그 정보를 지우고자 할 때는 DELETE를 사용하여 요청한다.

다만, 초기의 HTTP는 GET과 POST 메서드만 있어 많은 웹 애플리케이션들이 두 메서드만으로 구현되어왔고, 이러한 이유로 대부분의 웹 브라우저들이 GET과 POST 메서드만 지원한다. 이러한 경우 GET 메서드는 요청의 주된 목적이 리소스의 열람인 경우, POST 메서드는 요청의 주된 목적이 리소스에 대한 변화인 경우 사용된다.

## URL

URL(Uniform Resource Locator)은 인터넷 상에서 특정 리소스의 위치를 나타내는 문자열로, 흔히 링크 혹은 주소라고 부르기도 한다. URL은 정해진 구조에 따라 리소스에 대한 위치 정보를 세부적으로 담고 있다.

**http://www.kweb.org:3000/session/view?sid=92&t=3s**

protocol                      hostname                      port                      path                      query

Figure 1.1 Brief Structure of URL

Figure 1.1은 간단한 URL의 구조이다. 각 요소는 다음과 같은 정보를 나타낸다.

- protocol: 사용자가 요청을 보내 사용하고자 하는 애플리케이션의 프로토콜; http, https 뿐만 아니라 ftp, mailto 등의 프로토콜도 많이 사용된다.
- hostname: 요청을 보내고자 하는 서버의 주소; 도메인 이름이나 아이피 주소가 사용된다 (예: google.com / korea.ac.kr / 8.8.8.8)
- port: 서버에 접근하기 위한 관문의 번호; 이론적으로 0 이상,  $2^{16} = 65536$  미만의 정수가 가능하고, 프로토콜의 기본 포트에 요청을 보낼 때에는 생략할 수 있다. http와 https의 기본 포트는 각각 80번, 443번.
- path: 서버상에 존재하는 리소스의 위치 경로; 최근에는 서버에 요청하고자 하는 작업을 추상적으로 나타낸다.
- query: 요청에 대한 상세한 조건과 그에 대응하는 값을 key-value pair 형태로 제시하는 요소; ? 문자로 query 임을 나타내고, key=value의 형태로 나타내며, 각 key-value pair는 & 문자를 사용하여 구분한다. (예: ? bid=12&page=3&count=10)

웹 서버는 받은 요청의 메서드와 URL을 기반으로 요청을 수행하고 처리하여 클라이언트에게 응답을 보내는 일을 수행한다. 특히 메서드와 경로(path)를 묶어 라우트(route)라고 하며, “GET /session/view”와 같이 표시한다.

## HTTP Response and Response Status

HTTP 응답(response)은 클라이언트로부터 받은 요청에 대해 서버가 클라이언트에 전송하는 메시지이다.

### Code 1.2 Head of HTTP Response Example

```
HTTP/1.1 200 OK
Server: nginx
Date: Sat, 12 Sep 2020 13:43:29 GMT
Content-Type: text/html
Connection: keep-alive
Content-Length: 224
```

HTTP 응답의 헤더는 **Code 1.2**와 같은 형태로 구성되어 있다. 응답 라인에는 차례대로 HTTP 버전(HTTP/1.1), 상태 코드(200), 상태 메시지(OK)가 나열된다. 응답 상태(status)는 서버가 보낸 응답에 대한 상태를 대략적으로 나타내며, 상태 코드(status code)와 상태 메시지(status message)는 해당 상태를 각각 세 자리의 정수 형태와 문자열로 나타낸 것이다.

표준 HTTP 상태 코드는 63개가 있으며, 다음과 같이 크게 5가지로 분류된다.

- 1XX (Informational Response): 요청을 받았으며, 작업을 수행 중
- 2XX (Successful): 요청한 작업을 성공적으로 수행하였음
- 3XX (Redirection): 요청을 완료하기 위해 클라이언트는 추가 작업을 수행해야 함
- 4XX (Client Error): 클라이언트에서 잘못된 요청을 보내, 요청을 완료할 수 없음
- 5XX (Server Error): 서버상의 문제로 요청을 완료할 수 없음

다음 상태 코드들은 63개의 상태 코드 중 널리 사용되는 코드들이다.

- 200 OK: 요청이 성공적으로 완료됨
- 302 Found: 요청한 리소스가 다른 URL에 존재함
- 304 Not Modified: 응답할 내용이 클라이언트가 이전에 받은 응답과 동일함
- 400 Bad Request: 요청한 데이터의 형태가 잘못되어, 요청을 수행할 수 없음
- 401 Unauthorized: 인증이 실패하였거나 이루어지지 않아, 요청을 수행할 수 없음
- 403 Forbidden: 요청한 클라이언트가 적절한 접근 권한을 갖지 않음
- 404 Not Found: 요청한 리소스가 서버에 존재하지 않음
- 500 Internal Server Error: 서버 내부의 문제로 요청을 수행할 수 없음
- 503 Service Unavailable: 서버의 과부하나 점검으로 인해 요청을 수행할 수 없음

이처럼 상태 코드는 클라이언트(사용자)에게 요청에 대한 응답이 어떤 상태인지 알려준다. 상용 웹 브라우저는 상태 코드에 따라 별도의 작업을 수행하기도 하는데, 예를 들어 클라이언트(브라우저)가 /articles로 요청을 보냈을 때 서버가 302 Found라는 상태 코드와 함께 /articles/page/1이라는 URL을 응답하면, 브라우저는 자동으로 이 URL에 요청을 보내 응답을 받아온다. 이러한 과정을 **리다이렉트(redirect)**라고 한다.

## Insomnia

HTTP 통신은 그 목적에 맞게 인터넷 브라우저를 통해서 이루어질 수 있지만, 브라우저는 일반 사용자를 위한 프로그램인 만큼 HTTP 요청의 속성을 수정하고 응답의 속성을 확인하는 기능을 잘 지원하지 않는다. 따라서 개발자들은 이러한 웹 서버를 개발하면서 HTTP 통신을 용이하게 하기 위해 개발용 클라이언트 프로그램을 사용하곤 한다. 개발용 클라이언트 프로그램에는 Postman이나 Insomnia 등이 있으며, 본 교재에서는 Insomnia를 사용한다.

Insomnia 홈페이지<sup>1</sup>에서 자신의 OS에 맞는 설치파일을 다운로드 받아 설치한다. 이후 Insomnia를 실행하면 나타나는 Dashboard 창에서 우상단의 [Create] > [Request Collection] 버튼을 눌러 Collection을 생성하고, 좌상단의 [+] > [New Request] 버튼을 눌러 새로운 HTTP 요청을 생성한다.

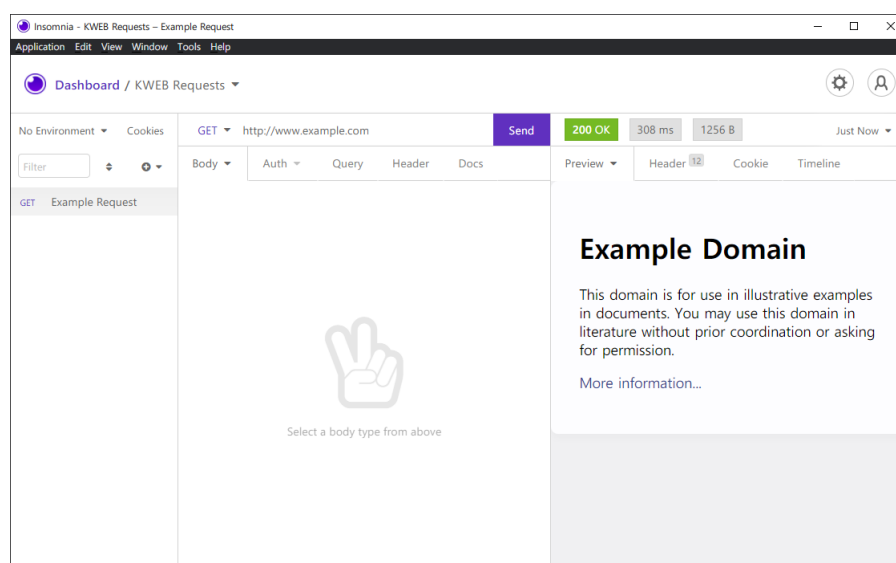


Figure 1.2 Insomnia

Figure 1.2와 같이 URL란에 “http://www.example.com/”을 입력하고, [Send] 버튼을 눌러 응답을 받아 응답을 확인한다. Insomnia를 사용하면 요청을 보낼 때 메서드나 내용, query, 헤더 등을 손쉽게 수정할 수 있고, 받은 응답의 상태 코드, 소요 시간, 크기, 헤더 등도 쉽게 확인할 수 있다. 또한, 응답 내용이 HTML 문서라면 렌더링하여 보여주고, JSON이나 XML 형태라면 formatting하여 보여주며, Raw HTML을 확인할 수 있는 등 여러 편의 기능을 제공한다.

<sup>1</sup><https://insomnia.rest/download>

## 1.2 Node.js Platform

### Node.js

Back-end는 client가 보낸 요청을 분석하여 그에 맞게 응답하는 로직을 수행하는 애플리케이션이므로, front-end와는 달리 Java, Python, C#, JS 등 (이론상) 어떤 언어로든 구현할 수 있다. 본 교재에서는 front-end 과정에서 다룬 바 있고, 이미 널리 사용되고 있는 Javascript를 활용하여 웹 서버를 구현한다.

원래 JS는 웹 브라우저에서만 동작하게끔 설계된 언어이기 때문에 JS 프로그램은 크롬, 파이어폭스 등 웹 브라우저에서만 실행될 수 있었다. 그러나 2009년 Ryan Dahl에 의해 **브라우저 밖에서도 JS를 실행할 수 있는 Node.js 플랫폼이 개발되었고, 이를 이용하여 JS로 웹 서버를 구현할 수 있게 되었다.**

Node.js는 JS가 갖는 대중성과 유연함으로 인해 유지 및 보수가 용이하다는 장점이 있으며, 동시에 가볍고 성능이 비교적 우수하다는 장점을 가지고 있다. 이러한 장점에 힘입어 최근 웹 서버의 구현을 위해 Node.js를 사용하는 사례가 빠르게 늘어나고 있으며, 관련 생태계가 활발하게 돌아가고 있다. 또한, Javascript에 명시적인 자료형을 추가한 Typescript를 이용한 개발도 매우 광범위하게 이루어지고 있다.

### Installing Node.js

Node.js 플랫폼은 홈페이지(<https://nodejs.org/>)에서 제공된다. 홈페이지에서는 LTS 버전과 Current 버전을 비롯해 현재까지 release된 모든 버전의 Node.js 플랫폼을 다운로드받을 수 있다. LTS 버전은 Long Time Support 버전, 즉 개발이 완료되어 안정적이며 신뢰도가 높아 장기간 지원되는 버전으로 실제 서비스를 위한 애플리케이션을 구현하기에 적합한 버전이다. 반면 Current 버전은 가장 최근에 release된 버전으로, 개발이 완료되지 않은 버전이기 때문에 신뢰도가 보장되지 않고 버그가 발생할 수 있어 Node.js의 최신 기능을 사용해볼 수는 있으나 실제 서비스를 목적으로 한 프로젝트에서는 적합하지 않다. 본 교재에서는 LTS 버전을 기준으로 작성되었으며, 2022년 9월 기준 LTS 버전은 16.17.0 버전이다.

Windows와 macOS에서는 홈페이지에서 LTS 버전 설치 파일을 다운로드한 후, 설치 파일을 실행해 추가 옵션 없이 설치한다. Linux 계열 운영체제는 Terminal<sup>2</sup>을 열고 **Shell 1.1**을 따라 설치 파일을 다운로드한 후 설치한다. 이때 <major-version> 부분에 다운로드 하고자 하는 버전의 major 번호<sup>3</sup>를 대입한다.

#### Shell 1.1 Installing Node.js on Linux

```
$ curl -sL https://deb.nodesource.com/setup-<major-version>.x | sudo bash -
$ sudo apt install -y nodejs
```

설치가 완료되면 Windows에서는 cmd, UNIX 계열 운영체제에서는 Terminal 등의 shell을 실행하여 **Shell 1.2**와 같이 Node.js와 npm의 버전을 확인한다.

<sup>2</sup>Shell에서 \$ 문자는 입력란을 뜻하므로 \$ 부분은 입력하지 않는다.

<sup>3</sup>16.17.0 버전의 경우 16

### Shell 1.2 Confirming Node.js Version for v16.17.0

```
$ node -v
v16.17.0
$ npm -v
8.19.2
```

여담으로, node 명령어를 실행하면 웹 브라우저의 개발자 도구의 Console과 같은 REPL shell을 사용할 수 있다.

## Running Simple Node.js Application

Node.js 홈페이지에는 JS로 구현된 아주 간단한 웹 서버 애플리케이션이 제시되어 있다. 프로젝트 디렉토리를 생성하고, index.js 파일을 생성하여 **Code 1.3**과 같이 작성한다.

### Code 1.3 Simple Web Server: index.js

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

이제 shell에서 **Shell 1.3**과 같이 index.js를 실행하고, 브라우저를 실행하여 `http://127.0.0.1:3000/`<sup>4</sup>에 접속하여 응답 결과를 확인한다.

### Shell 1.3 Running index.js

```
$ node index.js
```

Insomnia에서도 GET `http://127.0.0.1:3000/`로 요청을 보내보고, 메서드와 URL을 바꾸어서도 요청을 보내본다.

---

<sup>4</sup>127.0.0.1은 localhost 이므로 `http://localhost:3000/`에 접속하여도 된다.

## 1.3 git and GitHub

### Version Control System(VCS) and git

소프트웨어의 개발 과정에서 소스 코드의 백업, 개발 버전의 관리, 소스 코드의 변경사항 기록 및 추적 등의 작업은 필수적이다. 버전 관리 시스템 (Version Control System, VCS)은 소프트웨어 개발자를 위해 이러한 작업을 수행해주는 시스템이다.

git은 소프트웨어 개발 분야에서 압도적인 점유율을 차지하는 버전 관리 시스템이다. git은 버전 관리뿐만 아니라 branch를 이용하여 여러 사용자가 하나의 프로젝트를 개발하고 merge를 통해 각자 개발한 부분을 조율하여 반영할 수 있게 하고, 별도의 서버에 소스 코드를 저장할 수 있게 하여 소스 코드의 백업과 공유를 용이하게 한다.

### How git works

git은 프로젝트를 저장소(repository)에 저장하여 관리한다. 특정 디렉토리에서 git 저장소를 생성하면, 해당 디렉토리에 속한 모든 하위 파일과 하위 디렉토리는 프로젝트를 구성하는 요소가 되고, 그 디렉토리에 있는 파일에서 발생한 변경사항은 git에 의해 추적(track)된다.

로컬 디렉토리에서 발생한 변경사항은 add 명령을 통해 staging area에 추가되고, commit 명령을 통해 로컬 저장소(local repository)에 변경사항이 반영된다. Staging area는 commit을 실행하기 전 commit할 내용을 정리할 수 있게 도와주는 중간 단계 역할을 수행한다. 이렇게 commit을 이용하여 변경사항을 반영할 때마다 하나의 commit이 생성되는데, git은 버전을 각 commit 단위로 관리하고 다른 commit과의 변경사항을 확인할 수 있게 한다.

이렇게 생성된 commit들은 push 명령을 통해 원격 저장소(remote repository)에 업로드되어 반영된다. 반대로 pull 명령은 원격 저장소에 반영된 변경사항을 다운로드하여 로컬 저장소에 반영하며, clone 명령은 원격 저장소를 복사하여 로컬 디렉토리에 저장하는 명령이다.

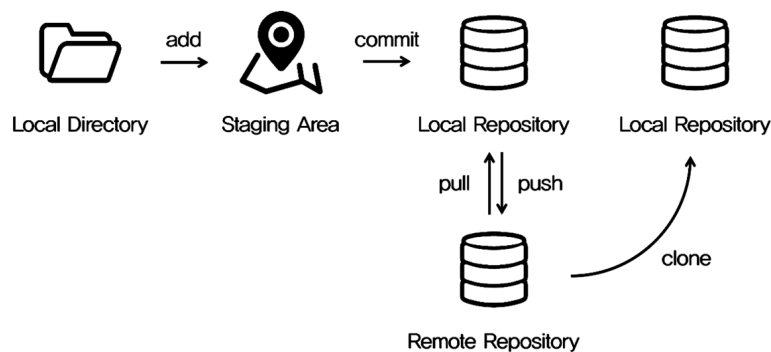


Figure 1.3 How git works



## gitignore

소프트웨어를 개발하는 과정에서 로그 파일, 빌드 파일(.exe, .out 등), 설정(config) 파일 등 소스 코드가 아닌 파일이 프로젝트 디렉토리 내에 생성되곤 한다. 설정 파일은 로컬 개발 환경에 따라 달리 설정해야 하고, 비밀번호나 비밀 키 등과 같이 제 3자가 알 경우 보안 취약점이 발생할 수 있는 내용이 포함되어 있을 수 있어 추적되어서는 안 된다. 빌드 파일은 기계어로 되어있기 때문에 변경사항을 추적하는 것이 용량만 차지하고 무의미하며, 빌드 방법만 정확히 서술되어 있다면 다른 사용자가 충분히 빌드할 수 있기 때문에 추적할 필요가 없다. 이렇게 소프트웨어 그 자체와는 무관한 파일들은 git을 통해 추적하지 않는 것이 권장된다.

이러한 파일이나 디렉토리를 gitignore 파일에 정규 표현식 형태로 명시하여 추적 대상에서 제외할 수 있다. gitignore 파일은 프로젝트 디렉토리의 최상위 디렉토리에 “.gitignore”이라는 이름으로 생성한다.

추적에서 제외되는 파일이나 디렉토리의 목록은 애플리케이션에서 사용되는 언어, 프레임워크, IDE 등 다양한 요소에 영향을 받는다. 이러한 요소에 따라 통상적인 gitignore 파일을 자동으로 생성하는 유용한 기능을 제공해주는 웹 사이트를 소개한다.

- <https://www.toptal.com/developers/gitignore>

## Installing git

git은 소프트웨어 개발에서 거의 빠지지 않고 사용되기 때문에 미리 설치되어 있는 경우가 종종 있다. macOS에서는 Xcode를 설치할 때 같이 설치되며, Linux 계열의 운영체제에는 사전에 설치되어 있는 경우가 많다. **Shell 1.4**와 같이 git이 설치되어 있는지 확인할 수 있다.

### Shell 1.4 Checking git version

```
$ git --version
```

Windows와 macOS에서는 git 홈페이지(<https://git-scm.com/downloads>)에서 자신의 운영체제에 맞는 설치파일을 다운받고, 실행하여 추가 옵션 없이 설치한다. Ubuntu와 같은 Linux 계열의 OS에서는 **Shell 1.5**와 같이 설치한다.

### Shell 1.5 git Installation (Ubuntu)

```
$ sudo apt install -y git
```

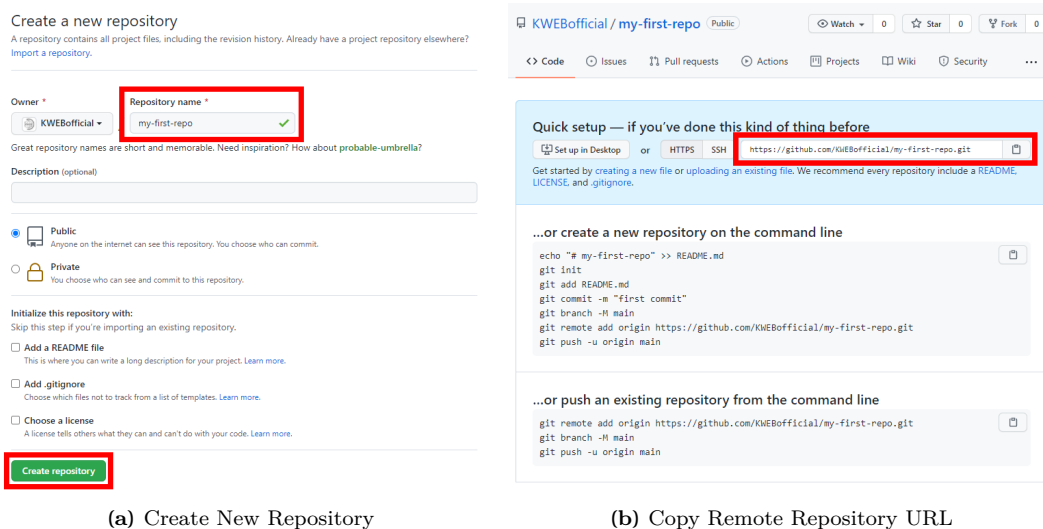
이후 설치가 완료되면 **Shell 1.4**와 같은 방법으로 설치가 잘 되었는지 확인한다.

## Starting GitHub and Creating Repository

GitHub은 대표적인 git 원격 저장소 무료 호스팅 사이트로, git을 이용하여 로컬 컴퓨터에 있는 프로젝트를 저장할 수 있는 원격 저장소를 제공한다. 다른 사람과 소스 코드를 공유하기 쉬울뿐만 아니라 git의 버전 관리 시스템과 branch 등을 이용한 협업 등의 기능, 그리고 그 기능을 보조해주는 여러 기능들을 간단하고 깔끔한 UI로 제공하여 오늘날 가장 대중적인 프로젝트 관리 및 오픈소스 공유 플랫폼이 되었다. 이러한 GitHub에 가입하고, 저장소를 만들어 GitHub 상에 간단한 프로젝트를 업로드하고 변경사항을 반영하는 실습을 진행한다.

먼저 GitHub 홈페이지에 접속하여 우측 상단의 [Sign Up]을 클릭하여 회원가입 페이지에 접속한 뒤, 사용자명, 비밀번호, 이메일 주소를 입력하고 계정 인증 절차를 완료한다. 이후 하단의 [Next: Select a plan]에서 [Free Plan]을 선택한 후, 본인의 정보를 간단하게 입력하고 [Complete setup]을 클릭하여 완료한다. 마지막으로 입력한 이메일을 통해 인증하여 가입을 마무리한다.

먼저 GitHub에 원격 저장소를 생성하는 단계이다.<sup>5</sup> GitHub 홈페이지 좌측의 [Repositories] 메뉴에서 [New] 버튼을 클릭하여 새 저장소를 생성할 수 있는 페이지로 이동하고, **Figure 1.4a**와 같이 [Repository Name]란에 저장소 이름을 “my-first-repo”로 지정하고, [Create Repository]를 클릭하여 새 저장소를 생성한다. 이후 해당 저장소를 열람할 수 있는 페이지에서 **Figure 1.4b**와 같이 원격 저장소의 주소를 복사해둔다.



(a) Create New Repository

(b) Copy Remote Repository URL

**Figure 1.4** Starting GitHub

이제 이 프로젝트를 clone 명령어를 통해 로컬 환경에 다운로드 받을 수 있다. 프로젝트를 다운받으려 하는 디렉토리에서 shell을 실행하고, **Shell 1.6**에서 <repo-url> 부분에 원격 저장소 주소를 입력하여 실행한다. 이후 프로젝트가 원하는 디렉토리에 정상적으로 clone 되었는지 확인한다.

### Shell 1.6 Clone remote repository to local

```
$ git clone <repo-url>
```

<sup>5</sup>로컬 저장소를 먼저 생성하는 방법도 있으나, 여러 설정을 해주어야 하기 때문에 GitHub에서 먼저 원격 저장소를 생성하는 것이 훨씬 간단하다.

## Commit Project Updates

이제 1.2절에서 작성한 코드를 활용하여 로컬 환경에 clone된 프로젝트를 수정하고, 수정한 내역을 commit 해본다. 먼저 프로젝트 내에 .gitignore 파일을 생성<sup>6</sup>하고, **Code 1.4**와 같이 작성하여 확장자가 txt인 모든 파일을 추적하지 않도록 한다.

### Code 1.4 .gitignore

```
*.txt
```

프로젝트 디렉토리에 index.js를 생성하고, **Code 1.3**(15쪽)의 코드를 작성한다. 이후 **Shell 1.7**과 같은 과정을 통해 변경 사항을 commit하고, 이를 원격 저장소에 push한다. 이때 `git add .` 명령어는 프로젝트 내 모든 파일의 변경 사항을 추적하여 staging area로 올린다는 뜻이다. 이후 원격 저장소에서 commit 내역을 확인하여, 변경 사항이 정상적으로 반영되었는지 확인한다.

### Shell 1.7 Commit Changes and Push Commits

```
$ git add .
$ git commit -m <commit-message>
$ git push
```

이제 index.js에서 `port`의 값을 4000으로 바꾸고, config.txt 파일을 생성하여 아무 내용이나 입력해보자. 그러면 index.js에서는 `port`가 선언된 네 번째 줄에서 변경이 발생하였고, config.txt는 .gitignore에 의해 추적되지 않으므로 변경 사항으로 간주되지 않을 것임을 예상할 수 있다. **Shell 1.7** 과정을 통해 변경사항을 다시 commit 및 push하고, 원격 저장소에서 commit 내역을 확인하여 이와 같은 예상이 맞는지 확인해본다.

---

<sup>6</sup>파일 시스템은 .으로 시작하는 파일을 만드는 것을 그다지 좋아하지 않으므로, VS Code 등의 IDE를 이용하여 만드는 것을 추천한다.



## Chapter 2

# Advanced Javascript

### Contents

2.1	Advanced Javascript Syntaxes . . . . .	22
2.2	Modules . . . . .	27
2.3	Asynchronous Programming . . . . .	30
2.4	Advanced Javascript Exercises . . . . .	34

## 2.1 Advanced Javascript Syntaxes

### Template Literal

문자열을 생성할 때 문자열 내에 변수값을 삽입하여 특정 부분의 값이 변수의 값에 따라 바뀌도록 생성하는 경우가 많다. 지금까지는 이러한 변수들을 문자열 연결(concatenation)을 통해 연결하여 문자열을 생성하곤 했다. 예를 들어 **Code 2.1**은 두 인자  $x, y$ 를 받아  $x$ 의  $y$  제곱 값을 출력하는 함수가 구현된 코드이다.

#### Code 2.1 String Formatting without Template Literal

```
const printPower = (x, y) => {  
  const msg = 'Value of ' + x + ' to the power ' + y + ' is ' + x ** y + '.';  
  console.log(msg);  
};  
  
printPower(3, 4);
```

**Code 2.1**은 의도한 기능은 잘 수행하지만, 두 가지 부수적인 문제점을 갖는다. 먼저 `msg` 문자열에서 변하지 않는 부분과 변하는 부분이 분리되어 있어, 실제 문자열의 형태가 직관적으로 표현되지 않는다. 또한, 문자열을 감싸는 문자('), 덧셈 기호(+) 등으로 인해 코드가 불필요하게 길어져 가독성을 저하시킨다. 이러한 문제점을 해결하여 문자열 생성을 더 쉽게 할 수 있도록 ES6 이후의 JS에서는 template literal이라는 문법을 제공한다.

**Template literal**을 이용하여 문자열을 생성하기 위해서는 먼저 **일반적인 문자열과는 달리 문자열을 backtick(`) 문자로 감싸야 한다**. Backtick으로 감싸진 문자열 내에서는 **`${}`과 ``${}`로 감싸진 부분이 JS 표현식으로 인식되어** 문자열이 생성될 때 표현식을 나타내는 부분이 감싸진 부분의 표현식의 반환값으로 대체된다.

**Code 2.2**는 **Code 2.1**의 함수를 template literal을 이용하여 표현한 코드로, 기존에 비해 매우 직관적이고 깔끔해진 것을 확인할 수 있다.

#### Code 2.2 String Formatting with Template Literal

```
const printPower = (x, y) => {  
  const msg = `Value of ${x} to the power ${y} is ${x ** y}.`;   
  console.log(msg);  
};  
  
printPower(3, 4);
```

여담으로, `${}` 부분을 표현식이 아닌 일반 문자열로 사용하고 싶은 경우 역슬래시(\)를 이용하여 escape 해주면 된다.

## Destructuring Assignment

**비구조화 할당 (destructuring assignment)**은 배열의 각 요소나 객체의 각 값을 서로 다른 변수에 편리하게 저장할 수 있게 하는 문법이다.

### Code 2.3 Array Destructuring

```
const arr = [1, 2, 3, 4];

const [a1, a2, a3, a4] = arr;
const [b1, , b3] = arr;
const [, , c4, c5, c6 = 10] = arr;
```

**Code 2.3**은 배열의 비구조화를 나타낸 예제이다. 구문에 대한 간단한 설명은 다음과 같다.

- 대입 연산자(=)의 좌변에 배열의 각 요소의 값이 할당될 변수를 배열 형태로 나타내고, 우변에는 비구조화하고자 하는 배열을 나타낸다.
- 좌변의 n번째 변수에는 우변의 n번째 요소의 값이 할당된다.
- 우변의 일부 값은 좌변에서 생략함으로써 무시할 수 있다.
- 좌변의 변수 중 우변에 대응되는 값이 없는 경우 **undefined**가 할당된다.
- 좌변에서 특정 변수의 기본값을 설정해주면, 비구조화의 결과 해당 변수의 값이 **undefined**일 때, 그 변수에는 설정한 기본값이 할당된다.

### Code 2.4 Object Destructuring

```
const obj = { x: 1, y: 2, z: 3 };
const { x, z, u, v = 10 } = obj;
const { y: y1 } = obj;
```

**Code 2.4**는 객체의 비구조화를 나타낸 예제이다. 배열의 비구조화와 전체적으로 유사하며, 차이점은 다음과 같다.

- 배열의 비구조화는 위치(index)를 기준으로 할당하는 반면, 객체의 비구조화는 속성(property) 이름을 기준으로 할당한다.
- 객체의 원래 속성 이름과는 다른 이름의 변수에 값을 할당할 수 있다.

객체의 비구조화는 객체의 원래 속성의 이름을 변수로써 사용할 때 권장되는 문법이다. 예를 들어 **Code 2.4**에서 **obj** 객체의 **x** 속성을 **x**라는 상수에 할당할 때, **const x = obj.x** 같은 표현보다는 **const { x } = obj** 같은 표현이 권장된다.

## Truthy and Falsy

JS의 모든 값은 암묵적으로(implicitly) true나 false로 변환될 수 있다. **암묵적으로 true로 변환되는 값들을 truthy, false로 변환되는 값들을 falsy**하다고 하고, 이러한 암묵적인 변환은 조건문 등에서 조건의 참/거짓을 확인할 때 유용하게 사용된다.

다음 8가지 값은 falsy하며, 나머지 값들은 모두 truthy이다.

- **false / 0 / -0 / 0n / null / undefined / "" / NaN**

Truthy, falsy 표현식은 **Code 2.5**와 같이 사용될 수 있다.

### Code 2.5 Truthy and Falsy Expressions

```
if (arr.length > 0) { ... }  
// can be converted to  
if (arr.length) { ... }  
  
if (foo === undefined) { ... }  
// can be converted to  
if (!foo) { ... }
```

**Code 2.6**은 **truthy/falsy한 값을 true/false로 변환**하는 코드이다.

### Code 2.6 Converting Truthy and Falsy Values

```
const emptyArray = [];  
const val1 = !!emptyArray;           // true  
const val2 = !!emptyArray.length;    // false
```

JS에서 논리적 OR를 계산하는 방식과 truthy/falsy를 이용하여 short-circuit evaluation(단축 평가)을 실시할 수 있다. **논리적 OR를 계산하는 이항 연산자 ||는 연산자 앞에 오는 피연산자가 truthy하면 앞에 오는 피연산자를 반환하고, falsy하면 뒤에 오는 피연산자를 반환한다.**

### Code 2.7 Short Circuit Evaluation

```
const port = config.port || 3000;
```

Short-circuit evaluation은 **Code 2.7**과 같이 변수의 기본값을 설정할 때 유용하게 사용된다. `config.port`의 값이 truthy하면 `port`의 값은 `config.port`의 값이 되고, falsy하면 3000이라는 기본값을 갖는다.



## Error Handling

프로그램에서 에러가 발생했을 때 이를 적절히 처리하는 작업은 매우 중요하다. 코드에서 에러가 발생하면 해당 코드의 실행이 중단되는데, 실제 서비스에서는 프로그램의 일부분에서 발생한 에러로 인해 프로그램 전체가 중단될 경우 심각한 문제가 발생할 수 있다. 이러한 문제를 방지하기 위해 에러 처리(error handling)가 적절히 이루어져야 한다.

### Code 2.8 Function Without Error Handling

```
const getStatusCode = res => res.status.code;

const code1 = getStatusCode({ status: { code: 400 } });
const code2 = getStatusCode({});
```

Code 2.8에서 `getStatusCode` 함수에 의도한 형태의 인자가 들어간다면 에러가 발생하지 않고 값을 정상적으로 반환하지만, 빈 객체와 같이 의도하지 않은 형태의 인자가 들어가면 에러가 발생한다. 이렇게 함수에 의도한 형태의 인자가 들어간다는 보장이 없다면 에러가 발생할 수 있고, 이렇게 발생하는 에러가 적절히 처리되지 않았기 때문에 이 프로그램은 종료되어 버린다.

이러한 에러 처리를 위해서 JS에서는 try-catch문을 지원한다. try-catch문은 `try`, `catch`, `finally`로 이루어져 있고, `try` block에서 에러가 발생하면 `catch` block의 코드가 실행되고, 그 다음 에러 발생 여부와 무관하게 `finally` block의 코드가 실행된다. `try` block은 반드시 필요하고, `catch` block이나 `finally` block 중 적어도 하나는 반드시 있어야 한다. 또한, `catch` 문은 발생한 에러와 관련된 속성이 포함되어 있는 에러 객체를 필요에 따라 인자로 받을 수 있다.

Code 2.8은 try-catch문을 이용하여 에러 처리를 한 코드이다. Shell에 출력되는 내용을 분석하여 코드의 실행 흐름을 파악해 보자.

### Code 2.9 Function With Error Handling

```
const getStatusCode = res => {
  try {
    console.log('try');
    return res.status.code;
  } catch (err) {
    console.log('catch');
    return 0;
  } finally {
    console.log('finally');
  }
};

const code1 = getStatusCode({ status: { code: 400 } });
console.log(code1);
const code2 = getStatusCode({});
console.log(code2);
```

JS에서 기본적으로 제공되는 에러 이외에도, 새로운 에러 클래스(Error)를 이용하여 에러 객체를 생성하고, throw 키워드를 이용하여 개발자의 필요에 따라 에러를 발생시킬 수 있다.

에러 처리를 이용하면 코드의 흐름을 조절할 수 있다. A 함수에서 B 함수를 호출했을 때, B 함수에서 예외 상황이 발생한다고 가정하자. 개발자는 B 함수에서 발생한 예외 상황을 즉시 처리하지 않고, A 함수에서 처리하기를 희망할 수 있다. 에러 처리가 되어있지 않은 함수는 에러 발생 즉시 종료되므로, B 함수에서 예외 상황과 관련된 에러를 발생시키고, A 함수에서 에러를 처리할 수 있다.

**Code 2.10** Controlling Flow with Error Handling

```
const validateData = data => {
  if (!data) throw new Error(500);
  if (!data.length) throw new Error(404);
  return data;
};

const createMessage = data => {
  try {
    const checkedData = validateData(data);
    return `Success: ${data}`;
  } catch (e) {
    return `Failed: ${e.message}`;
  }
};

console.log(createMessage());
console.log(createMessage([]));
console.log(createMessage([1, 2, 3]));
```

**Code 2.10**은 try-catch문을 이용하여 코드 흐름을 조절한 예제이다. validateData 함수에서 데이터를 검증할 때 인자의 형태에 따라 예외적인 경우 그에 맞는 에러를 발생시키면서 예외 처리를 즉시 수행하지 않는다. createMessage에서는 validateData에서 에러가 발생한다면 catch 문으로 코드 흐름이 넘어가고, 그렇지 않다면 try 문의 코드가 실행된다. 이처럼 try-catch문을 이용하면 예외 처리를 편리하고 깔끔하게 할 수 있고, 코드가 복잡해지고 함수 호출의 단계가 깊어질수록 try-catch문을 이용하여 코드 흐름을 편리하게 조절할 수 있다.

## 2.2 Modules

### Necessity of Modules

모듈(module)이란 외부의 영향을 받지 않는 독립적이고, 재사용이 가능한 코드의 묶음으로, 대체로 하나의 모듈에는 유사한 목적을 가진 상수, 클래스, 메서드, 하위 모듈 등이 모여있다. 예를 들어, C에는 `strlen`, `strcat`, `strcpy` 등 문자열을 간편하게 다룰 수 있는 함수들을 모아둔 `string.h`라는 헤더가 있고, Python에는 `listdir`, `mkdir`, `rename` 등 운영체제와 관련된 메서드를 모아둔 `os`라는 라이브러리가 있다. 마찬가지로 Node.js에도 유사한 목적을 가진 코드들의 집합인 모듈이 존재한다.

Node.js는 파일 시스템을 다루기 위한 모듈인 `fs`, HTTP 요청을 보내고 응답을 받는 모듈인 `http` 등을 내장 모듈의 형태로 기본적으로 제공하며, 내장 모듈의 API 문서는 Node.js 홈페이지에서 확인할 수 있다. 특히 `http` 모듈은 **Code 1.3**(15쪽)에서 사용한 바 있다.

내장 모듈 이외에도 인터넷 상에 배포된 외장 모듈을 `npm`이나 `yarn`과 같은 패키지 매니저를 통해 설치하여 사용할 수도 있고, 자신의 코드를 모듈화하고 분류하여 프로젝트를 구조화하거나, 자신이 원하는 기능을 모아둔 모듈을 제작할 수 있다.

소프트웨어 개발 과정에서 모듈을 적절히 도입하여 활용하는 것은 고수준(high-level) 애플리케이션의 개발자에게 필수적으로 요구되는 능력이다. 이론적으로 소프트웨어에서 사용되는 모든 기능을 직접 구현하는 것이 가능하긴 하지만, 구현하고자 하는 기능에 비해 요구되는 비용이 지나치게 많고 유지 및 보수가 매우 어렵기 때문에 지양되는 방법이다. 또한 모듈은 앞서 설명한 프로젝트의 구조화에서도 매우 중요한 역할을 한다. 이번 절에서는 이러한 모듈을 설치하고, 사용하고, 작성하는 방법을 다룬다.

### Installing External Module

외장 모듈은 패키지 매니저를 통해 설치할 수 있으며, 본 교재에서는 1.2절에서 설치한 `npm`을 이용한다. 프로젝트마다 사용하는 모듈의 버전이 달라 호환성 문제가 발생할 수 있기 때문에 모듈은 프로젝트마다 따로 설치하는 것이 원칙이다.

**Shell 2.1**은 JS에서 배열, 객체 등 필수적인 자료형을 간편하게 다루는 여러 유용한 메서드를 제공하는 `lodash` 모듈을 설치하는 방법이다. 설치가 완료되면 프로젝트 디렉토리 내에 “`node_modules`”라는 디렉토리가 생성되고, 그 안에 `lodash` 디렉토리가 생성된 것을 확인할 수 있다.<sup>1</sup>

#### Shell 2.1 Installing External Module

```
$ npm install lodash
```

<sup>1</sup>설치 과정에서 보듯, 외장 모듈들은 인터넷에서 다운로드하여 설치할 수 있으므로 git에 의해 추적되지 않아야 한다.

## Generating Module

모듈은 하나의 파일이 하나의 모듈로 간주된다. `module.exports`에 모듈 밖에서 사용할 상수, 함수, 객체 등의 값을 할당하여 `export` 하면, 다른 파일에서 이 모듈을 `import` 하여 모듈 내의 상수, 함수, 객체 등을 사용할 수 있다. **Code 2.11**은 둥근 도형들의 넓이, 부피 등을 계산하는 모듈을 생성하는 예제이다.

**Code 2.11** Creating Module (circular-shapes.js)

```
const PI = 3.14159265358;

const round = number => Math.round(number * 100) / 100;

const getCircumference = radius => round(2 * PI * radius);

const getCircleArea = radius => round(PI * radius ** 2);

const getCylinderSurfaceArea = (radius, height) => {
  const circleArea = getCircleArea(radius);
  const sideArea = getCircumference(radius) * height;
  return round(2 * circleArea + sideArea);
};

module.exports = {
  getCircumference: getCircumference,
  getCircleArea: getCircleArea,
  getCylinderSurfaceArea: getCylinderSurfaceArea,
  getSphereVolume: radius => round(4 * PI * radius ** 3 / 3),
};
```

다른 파일에서는 **Code 2.11**의 모듈의 변수, 상수, 함수 중 `PI`와 `round`에는 접근할 수 없고, `getCircumference`, `getCircleArea`, `getCylinderSurfaceArea`, `getSphereVolume`에는 접근할 수 있다.

**Code 2.12** Property Shorthand

```
module.exports = {
  getCircumference,
  getCircleArea,
  getCylinderSurfaceArea,
  getSphereVolume: radius => round(4 * PI * radius ** 3 / 3),
};
```

참고로, **Code 2.11**에서 `getCircleArea` 등과 같이 객체의 속성의 이름과 값에 할당되는 변수의 이름이 같을 경우 **Code 2.12**와 같이 줄여쓸 수 있다.

## Importing and Using Module

모듈을 사용하기 위해서는 `require` 함수를 사용하여 import해야 한다. 내장 모듈이나 외장 모듈은 모듈의 이름을 `require` 함수의 인자로 전달하고, 직접 생성한 모듈은 import하는 파일과의 상대 경로를 인자로 전달한다.

먼저, **Code 2.13**은 내장 모듈인 `path`를 `require` 함수를 이용해 import하여 `path`라는 상수에 할당하고, 이를 사용하는 코드이다.

### Code 2.13 Import Internal Module

```
const path = require('path');

const myFile = '/home/ubuntu/kuniv/kweb/example.js';
const dirname = path.dirname(myFile);
const basename = path.basename(myFile);
const extname = path.extname(myFile);

console.log(`path.dirname = ${dirname}`);
console.log(`path.basename = ${basename}`);
console.log(`path.extname = ${extname}`);
```

**Code 2.14**는 앞서 작성한 `circular-shapes.js` 모듈을 파일의 상대 경로를 `require` 함수의 인자로 전달하여 import하고 사용하는 코드이다. 이때 `circular-shapes.js` 파일은 **Code 2.14** 파일과 같은 디렉토리에 있다.

### Code 2.14 Import Custom Module circularShapes

```
const circularShapes = require('./circular-shapes');

const r = 10;
const h = 20;

console.log(`Circumference = ${circularShapes.getCircumference(r)}`);
console.log(`Circle Area = ${circularShapes.getCircleArea(r)}`);
console.log(`Sphere Volume = ${circularShapes.getSphereVolume(r)}`);
console.log(`Cylinder Surface Area = ${circularShapes.getCylinderSurfaceArea(r, h)}`);
```

모듈 전체를 import한 **Code 2.14**와는 달리 **비구조화 할당을 이용**하면 **Code 2.15**와 같이 모듈의 일부분만 import하여 사용할 수 있다.

### Code 2.15 Import Part of Module

```
const { getCylinderSurfaceArea } = require('./circular-shapes');

const r = 10;
const h = 20;

console.log(`Surface Area of Cylinder = ${getCylinderSurfaceArea(r, h)}`);
```

## 2.3 Asynchronous Programming

### Synchronous Programming

동기식(synchronous) 프로그래밍 모델이란 프로그램의 코드가 순차적(sequentially)으로 실행되도록 설계하는 프로그래밍 모델로, 이렇게 설계 및 구현된 프로그램은 코드가 나열된 순서대로 실행된다. 여러분이 지금까지 구현한 대부분의 프로그램은 동기식으로 구현되었을 가능성이 높다.

이러한 동기식 프로그램은 코드가 순차적으로 실행되기 때문에 실행 흐름을 직관적으로 파악할 수 있다는 장점이 있고, 특히 코드가 수행하는 작업이 메모리에 접근하는 횟수가 많은 작업이거나 병렬 처리가 불가능 또는 불필요한 계산 위주의 작업인 경우 동기식 프로그래밍이 적합할 수 있다.

#### Code 2.16 Reading File Synchronously

```
const fs = require('fs');

console.log('String 1');

const data = fs.readFileSync('./file.bin');
console.log(`Data length: ${data.length} bytes`);

console.log('String 2');
```

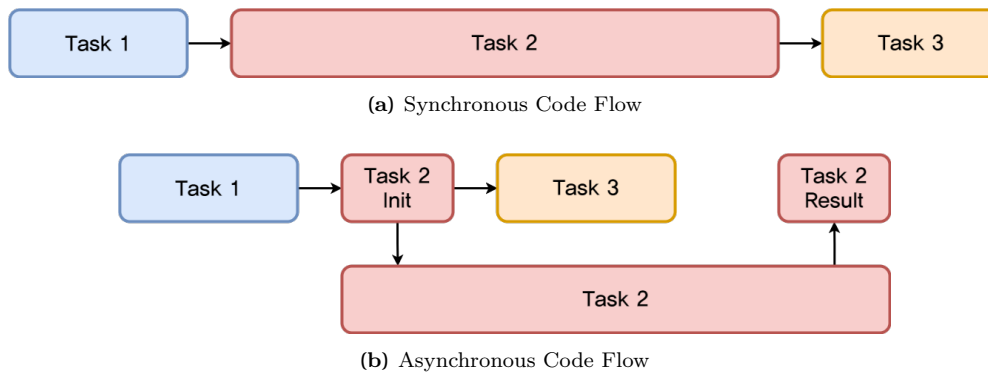
Code 2.16은 file.bin이라는 파일을 처음부터 끝까지 읽어 내용의 길이, 즉 파일의 크기를 표시하는 코드로, fs.readFileSync 메서드는 인자로 받은 파일을 읽어 그 내용을 반환하는 메서드이다. 이 프로그램을 실행하면 “String 1” 문자열이 출력되고, file.bin 파일의 내용을 읽어들이고 후, 파일의 크기가 출력되고 “String 2” 문자열이 출력된다.

그런데 file.bin이 크기가 매우 큰 파일이어서 fs.readFileSync 메서드로 그 내용을 읽는 시간이 오래 소요된다고 가정해보자. 이때 파일을 읽는 과정과는 무관한 “String 2” 문자열은 file.bin 파일을 읽는 동안 출력되지 못한 채 기다려야 한다. 이렇게 동기식으로 설계된 프로그램에서는 프로그램이 특정 작업을 수행하고 있다면 다른 작업들은 그 작업이 끝날 때까지 기다리고 있어야 한다.

동기식으로 설계된 소프트웨어에서는 특정 작업으로 인해 다른 작업의 실행이 막히는 경우는 많이 발생한다. 가장 대표적인 예시는 예제 코드와 같이 입/출력 작업이 수반되는 경우로, 디스크, 네트워크 등 주변 장치와의 통신 과정에서 발생하는 지연으로 인해 코드의 흐름이 막히고, 다른 요청이나 작업을 처리할 수 없는 상황이 발생하곤 한다.

### Asynchronous Programming with Callbacks

비동기(asynchronous) 프로그래밍 모델은 이러한 문제점을 해결할 수 있는 프로그래밍 모델로, 프로그램이 코드의 순서대로 실행되지 않을 수 있는 프로그래밍 모델이다.



**Figure 2.1** Code Flow of Synchronous Code and Asynchronous Code

**Figure 2.1**은 코드를 동기식으로 구현했을 때와 비동기식으로 구현했을 때 코드의 흐름을 개략적으로 나타낸 도식으로, Task 2는 소요 시간이 Task 1과 Task 3에 비해 매우 긴 작업이다. 동기식 코드(**Figure 2.1a**)에서 Task 3은 Task 2가 종료되고 나서야 실행될 수 있다. 그러나 비동기식 코드(**Figure 2.1b**)에서는 Task 2는 Task 1이 종료된 직후 실행되고, Task 3은 Task 2의 종료 여부와 상관없이 Task 2가 실행된 직후에 실행되며, Task 3은 Task 2보다 먼저 끝나고 Task 2의 수행 결과는 종료 이후 따로 처리된다.

#### Code 2.17 Reading File Asynchronously

```

const fs = require('fs');

console.log('String 1');

fs.readFile('./file.bin', (err, data) => {
  if (!err) console.log(`Data length: ${data.length} bytes`);
  else console.error(err);
});

console.log('String 2');
  
```

**Code 2.17**은 **Code 2.16**을 비동기식으로 구현한 코드로, file.bin 파일을 읽는 작업을 먼저 실행한 뒤, 다 읽을 때까지 기다리지 않고 다음 코드를 실행하여 코드의 흐름이 갈라지게 된다. **Figure 2.1**에서 “String 1”을 출력하는 작업은 Task 1, 파일을 읽어 크기를 출력하는 작업은 Task 2, “String 2”를 출력하는 작업은 Task 3에 해당하고, file.bin의 크기가 충분히 클 경우 “String 1”, “String 2”가 출력된 후 파일의 크기가 출력되게 된다.

**Code 2.17**에서 볼드체로 표시된 부분을 콜백 함수(callback function)라고 한다. 콜백 함수는 메서드의 실행 결과를 인자로 받는 함수로, fs.readFile 메서드는 파일 읽기를 실패한 경우 err 인자에 에러 객체를 넘겨주고, 성공한 경우 읽은 데이터를 data 인자에 넘겨준다. 이렇듯 콜백 함수를 사용하여 비동기적으로 수행된 코드의 결과를 사용할 수 있다.

그러나 콜백을 이용한 비동기 프로그래밍은 여러 가지 문제점이 있는데, 가장 대표적인 문제점은 콜백 지옥(callback hell)이다. 콜백 함수 내에서 또 비동기 함수를 사용하여 콜백 함수를 호출하는 코드가 반복되면, 들여쓰기(indent)가 계속되어 **Code 2.18**과 같이 코드의 깊이가 점점 깊어진다. 적당한 들여쓰기는 코드의 흐름을 파악하기 쉽게 하지만, 지나친 들여쓰기는 코드의 가독성을 현저히 저해시킨다.

**Code 2.18** Example of Callback Hell (Abstractly)

```
const fs = require('fs');

fs.readFile('db.info', (err, data) => {
  const tableLocation = identifyTable(data, 'A');
  fs.readFile(tableLocation, (err, data) => {
    const recordLocation = identifyRecord(data, 'B');
    fs.readFile(recordLocation, (err, data) => {
      const record = parseRecord(data);
      // do something based on record's data
      console.log(processedData);
    });
  });
});
```

## async/await

ES6부터는 비동기 프로그래밍을 보다 간편하게 할 수 있는 `async/await` 방식이 도입되었다. 이 방식을 이용하면 비동기식 코드를 기존의 동기식 코드에 가까운 형태로 작성할 수 있다.

**Code 2.19** Reading File Using `async/await`

```
const fs = require('fs');
const util = require('util');

const readFile = util.promisify(fs.readFile);

const printFileSize = async filename => {
  try {
    const data = await readFile(filename);
    console.log(`Data length: ${data.length} bytes`);
  } catch (err) {
    console.error(err);
  }
};

console.log('String 1');
printFileSize('./file.bin');
console.log('String 2');
```

**Code 2.19**는 **Code 2.17**을 `async/await` 방식으로 작성한 코드로, 원래 코드에 비해 동기식 코드와 매우 유사해진 것을 확인할 수 있다. `async/await` 방식은 몇 가지 특징이 있다.

- 비동기 함수는 함수의 앞에 `async` 키워드를 붙여 비동기 함수임을 표시한다.
- 비동기 함수를 호출할 때 `await` 키워드를 사용하면 함수가 끝날 때까지 기다리며, 사용하지 않으면 기다리지 않고 다음 코드를 실행한다.
- `await` 키워드는 비동기 함수 내에서만 사용 가능하다.



여기에서 `util.promisify` 메서드는 일반적인 메서드를 `async/await` 키워드를 사용할 수 있도록 변형(Promise화<sup>2</sup>)시켜주는 메서드이다.

**Code 2.20**은 **Code 2.19**의 `printFileSize` 함수를 이용한 코드이다. `file1.bin`과 `file2.bin` 파일의 크기가 충분히 클 때 출력 순서를 예상할 수 있겠는가?

**Code 2.20** Reading Big File Using `async/await` (Derived from **Code 2.19**)

```
(async () => {
  console.log('String 1');
  await printFileSize('./file1.bin');
  console.log('String 2');
  printFileSize('./file2.bin');
  console.log('String 3');
})();
```

`file1.bin` 파일의 크기를 출력하는 코드는 `await` 키워드가 있으므로 `printFileSize` 함수의 종료를 기다리고, `file2.bin` 파일의 크기를 출력하는 코드는 `await` 키워드가 없으므로 함수의 종료를 기다리지 않는다. 따라서 “String 1”, `file1.bin` 파일의 크기, “String 2”, “String 3”, `file2.bin` 파일의 크기 순으로 출력된다.

**Code 2.21** Callback Hell Resolved with `async/await`

```
const fs = require('fs');
const readFile = util.promisify(fs.readFile);

(async () => {
  try {
    const dbInfo = await readFile('db.info');
    const tableInfo = await readFile(identityTable(dbInfo, 'A'));
    const rawRecord = await readFile(identifyRecord(tableInfo));
    const record = parseRecord(data);
    // do something based on record's data
    console.log(processedData);
  } catch (err) {
    console.error(err);
  }
})();
```

`async/await` 방식을 사용하면 callback hell 역시 해결된다. **Code 2.21**은 **Code 2.18**을 `async/await` 방식으로 작성한 코드로, 콜백 함수의 반복적인 호출로 발생하는 callback hell이 해결되어 가독성이 높아졌다.

이렇듯 코드를 간결하게 바꾸어주는 `async/await` 방식도 단점이 존재한다. `async/await` 방식은 자체적인 에러 처리 방식이 없으므로 **Code 2.19**와 같이 try-catch문을 사용하여야 한다. try-catch문을 사용한 에러 처리는 가독성 측면에서는 깔끔하지만 한 칸의 indent가 강제되고, JS의 try-catch문은 속도가 느려 성능 저하가 발생할 수 있다는 단점이 있다.

<sup>2</sup>`async/await` 방식은 Promise 방식을 통한 비동기 프로그래밍을 간결화한 방식이다. Promise에 관한 자세한 내용은 MDN 문서([https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using\\_promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises))에서 확인할 수 있다.

## 2.4 Advanced Javascript Exercises

### Exercise 2.1: Number of Cases Calculating Module

주어진 식을 참고하여  $n$ 과  $r$ 를 인자로 받아 순열(permutation), 조합(combination), 중복 순열(multiset permutation), 중복 조합(multiset combination) 값을 반환하는 메서드 `permutation`, `combination`, `multiPermutation`, `multiCombination`을 갖는 모듈 `cases`를 구현하여라.

$$\text{Permutation : } {}_n P_r = \frac{n!}{(n-r)!} \quad / \quad \text{Multiset Permutation : } n^r$$

$$\text{Combination : } \binom{n}{r} = \frac{n!}{(n-r)! r!} \quad / \quad \text{Multiset Combination : } \left( \binom{n}{r} \right) = \binom{n+r-1}{r}$$

#### Code 2.22 Exercise 2.1 Example

```
const cases = require('./cases');

const n = 8;
const r = 3;

console.log(`n = ${n}, r = ${r}`)
console.log(`Permutation: ${cases.permutation(n, r)}`);
console.log(`Combination: ${cases.combination(n, r)}`);
console.log(`Multi Permutation: ${cases.multiPermutation(n, r)}`);
console.log(`Multi Combination: ${cases.multiCombination(n, r)}`);
```

**Code 2.22**는 `cases` 모듈을 이용하여 작성한 예시 코드이다. 이 코드를 실행하였을 때 결과는 **Shell 2.2**와 같아야 한다.

#### Shell 2.2 Exercise 2.1 Example Result

```
$ node index.2.1.js
n = 8, r = 3
Permutation: 336
Combination: 56
Multi Permutation: 512
Multi Combination: 120
```

## Exercise 2.2: Directory Search

프로젝트 내의 `test` 디렉토리 내부를 모두 탐색하여 확장자가 `js`인 파일의 목록을 출력하고, 탐색 과정에서 에러가 발생하면 `console.error` 메서드를 이용하여 에러를 출력하는 프로그램을 내장 모듈인 `fs`와 `path` 모듈의 다음 메서드들을 활용하여 구현하여라. 두 모듈의 API 문서<sup>3</sup>는 Node.js 홈페이지에서 확인할 수 있으며, 필요에 따라 **Code 2.19**와 같이 `util.promisify` 메서드를 사용해야 한다.

- `fs` (File System): `readdir`, `stat`, `stats.isDirectory`
- `path`: `extname`, `join`

### Shell 2.3 Exercise 2.2 Example (On Linux)

```
$ tree test
test
├── 01.js
├── 01.py
├── 02.js
├── test1
│   ├── 11.c
│   ├── 11.js
│   ├── 11.py
│   └── 12.js
├── test2
│   ├── 21.js
│   └── 21.ts
├── test3
│   ├── 31.cs
│   └── 31.java
├── test4
│   ├── 41.js
│   └── 41.py
└── test5
    └── test51
        ├── 511.js
        └── 511.py
```

예를 들어 `test` 디렉토리의 내부 구조가 **Shell 2.3**과 같을 때, 프로그램의 실행 결과는 **Shell 2.4**와 같아야 한다.

### Shell 2.4 Exercise 2.2 Example Result

```
$ node index.2.2.js
test\01.js
test\02.js
test\test1\11.js
test\test1\12.js
test\test2\21.js
test\test4\41.js
test\test5\test51\511.js
```

<sup>3</sup>16.17.0 버전의 경우 <https://nodejs.org/dist/latest-v16.x/docs/api>



## Chapter 3

# Express.js and View Engine

### Contents

3.1	Express.js . . . . .	38
3.2	View Engine . . . . .	43
3.3	Sending POST Request From Browser . . . . .	47
3.4	Express.js and View Engine Exercises . . . . .	48

## 3.1 Express.js

### Framework and Express.js

1장에서 웹 서버가 수행해야 하는 기능과 웹 서버 구현에 필요한 기초적인 개념을 학습하였고, 유효한 기능을 수행하는 간단한 서버를 http 모듈을 이용하여 구현해보았다.

그러나 http 모듈은 HTTP 통신을 통해 데이터를 주고받는 메서드를 제공할 뿐, 그 외에 웹 서버가 요구하는 수많은 기능을 제공하지는 않는다. 이러한 기능에는 라우팅(routing), query 파싱, 정적 파일 로딩, 에러 처리 등이 있다. 물론 이러한 기능들을 직접 구현하는 것이 불가능하지는 않지만, 통상적인 웹 서버에서 공통적으로 요구하는 기능이므로 매번 구현하는 것은 비효율적이다.

이러한 종류의 소프트웨어를 개발할 때는 효율적인 개발을 위해 프레임워크를 사용하곤 한다. 프레임워크(framework)는 개발자가 구현하고자 하는 기능의 구현에만 집중할 수 있도록 공통적이며 기본적으로 요구되는 기능들을 미리 갖춰둔 기본 뼈대이다. 웹 서버는 프레임워크를 사용하여 구현하는 것이 일반적이기 때문에 수많은 웹 프레임워크가 존재하며, Python의 Django와 Flask, Java와 Kotlin의 Spring, PHP의 Laravel 등은 대표적인 웹 프레임워크이다.

Node.js에서 동작하는 JS의 웹 프레임워크에는 대표적으로 Express.js, Koa.js, hapi.js, NestJS 등이 있다. Express.js는 장기간 가장 보편적으로 사용되었으며 사용되고 있는 프레임워크이며, Koa.js는 Express.js의 기존 개발팀이 Express.js의 소유권이 이전되면서 장기적인 유지 및 보수가 제대로 이루어지지 않을 것을 우려해 만든 프레임워크로, 최근에는 Express.js로 구현된 프로젝트들이 Koa.js로 migration되는 추세이다. 그러나 Express.js의 점유율은 여전히 높으며 그동안 널리 사용되어왔기 때문에 포럼 등 온라인에서 참고할 수 있는 자료는 Express.js가 훨씬 많으므로, 본 교재에서는 Express.js 프레임워크를 사용한다. 또한, 최근에는 Express.js를 래핑(wrapping)하고 편리한 기능을 다수 탑재한 NestJS가 핫한 웹 프레임워크로 부상하고 있다.

### Simple Express.js Server

Express.js 프레임워크를 이용하여 간단한 웹 서버를 구현해보자.

#### Shell 3.1 Create Express Server

```
$ npm init -y
$ npm install express
```

먼저 **Shell 3.1**과 같이 Node.js 프로젝트를 생성하고, express 모듈을 설치한 뒤, 적절한 gitignore 파일을 생성한다. npm init 명령은 Node.js 프로젝트에 관한 정보를 담은 package.json 파일이 생성되며, -y 옵션을 붙이면 기본 설정을 그대로 사용한다. 프로젝트에서 사용하는 모듈을 npm을 이용하여 설치하면 모듈들의 이름과 버전이 package.json 파일의 dependencies 속성에 기록되어 다른 개발 환경에서 프로젝트를 실행할 때 모듈 파일들을 직접 복사하지 않고도 모듈을 설치할 수 있고, 그렇기 때문에 모듈들이 저장된 node\_modules 디렉토리는 gitignore 파일에 의해 추적되지 않아야 한다.

이제 index.js를 생성하고, **Code 3.1**의 코드를 작성한 후 실행한다.

#### Code 3.1 Simple Express Server

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => res.send('Hello World!'));

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

정보를 꺼낼 때 req  
응답을 할 때 res

**Code 3.1**은 **Code 1.3**(15쪽)을 Express.js로 다시 작성한 코드로, 거의 같은 기능을 수행하면서 코드는 매우 간결해진 것을 확인할 수 있다.

코드를 살펴보면, 먼저 express 모듈을 이용하여 웹 서버의 설계에 대한 모든 정보를 담고있는 app 객체를 생성한다. 그리고 app 객체의 get 메서드를 통해 서버에 GET / 라우트로 요청이 들어오면 “Hello World!” 문자열을 응답으로 보낸다. 다만 **Code 1.3**의 서버는 모든 요청 라우트에 대한 응답이 구현되어 있지만, **Code 3.1**의 서버는 GET / 라우트에 대한 응답만 구현되어 있고 다른 라우트로 들어온 요청에 대해서는 404 Not Found 에러를 응답한다는 차이점이 있다.

## Routing

웹 애플리케이션에서 요청 경로와 메서드, 즉 라우트는 사용자가 서버에게 요청하는 작업, 즉 서버가 수행해야 하는 작업을 나타내는 매우 중요한 요소이다. 따라서 서버 개발자는 요청 라우트에 따라 수행할 작업을 적절히 나누어야 하고, 이러한 작업을 라우팅(routing)이라고 한다.

예를 들어 게시물을 작성하고 열람할 수 있는 웹 애플리케이션의 라우트는 다음과 같이 설계할 수 있다.

- GET /auth/sign\_in: 로그인할 수 있는 페이지 응답
- POST /auth/sign\_in: 아이디와 비밀번호를 받아서 사용자의 인증 요청을 처리하고, 그 결과를 응답
- GET /board/page/1: 게시판의 첫 번째 페이지 응답
- GET /post/3: 서버에 저장된 게시물 중 3번 게시물을 찾아서 응답
- POST /post: 제목과 내용 등을 받아서 서버에 새로운 게시물을 생성하고 저장

Express.js는 이러한 라우팅을 간편하게 할 수 있는 **라우팅 메서드**를 제공한다. **app 객체의 get, post 메서드** 등은 GET, POST 메서드 요청을 처리하고, **put, delete 등의 메서드** 역시 유사한 기능을 수행한다. 라우팅 메서드는 **첫 번째 인자로 매칭하고자 하는 경로를 정규표현식 형태로 받고**, 이후 **라우트에 매칭되었을 때 실행할 함수, 즉 controller 함수를 순서대로 인자로 받는다**. 예를 들어, **Code 3.1**에서 **app.get('/', ...)** 코드는 GET 메서드, / 경로로 들어온 요청을 처리하는 코드이다.

## Request and Response Objects

1.1절에서 다루었듯 HTTP 요청과 응답에는 메서드, 경로, 상태 코드뿐만 아니라 body, header, cookie 등의 정보를 담고 있다. 이러한 정보에 따라 수행되는 작업이나 작업에 사용되는 데이터의 값이 달라질 수 있으므로 controller 함수에서는 이러한 정보에 접근할 수 있어야 한다.

**Code 3.1**에서 컨트롤러 함수는 두 인자 req와 res를 받는다. 두 인자는 요청 객체와 응답 객체로, 각각 HTTP 요청과 HTTP 응답의 각 정보를 담고 있다. **요청 객체에는 HTTP 요청을 분석한 데이터가 query, body, header, cookie 등의 속성에 할당되어 있어 개발자가 controller 함수의 구현 과정에서 필요한 값에 접근할 수 있다. 응답 객체에는 응답으로 보낼 상태 코드를 정할 수 있는 status 메서드, 응답 내용을 정하고 보낼 수 있는 send 메서드, 다른 경로로 리다이렉트 시킬 수 있는 redirect 메서드 등이 있다.**

## Middleware

Express.js로 생성된 애플리케이션은 일련의 미들웨어를 순차적으로 실행하는 애플리케이션이라고 요약할 수 있다. 여기에서 미들웨어(middleware)란, 요청 객체와 응답 객체, 그리고 요청 - 응답 처리 과정 중 그 다음 미들웨어 함수에 대한 접근 권한을 갖는 함수이다.

### Code 3.2 Middleware Example

```
const express = require('express');
const app = express();
const port = 3000;

app.use((req, res, next) => {
  const randomNumber = Math.floor(Math.random() * 10);
  console.log(`Random Number: ${randomNumber}`);
  if (randomNumber % 3) return next();
  else return res.sendStatus(403);
});

app.use((req, res, next) => {
  const { method, path } = req;
  console.log(`${method} ${path}`);
  return next();
});

app.get('/', (req, res) => res.send('GET /'));

app.post('/', (req, res) => res.send('POST /'));

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

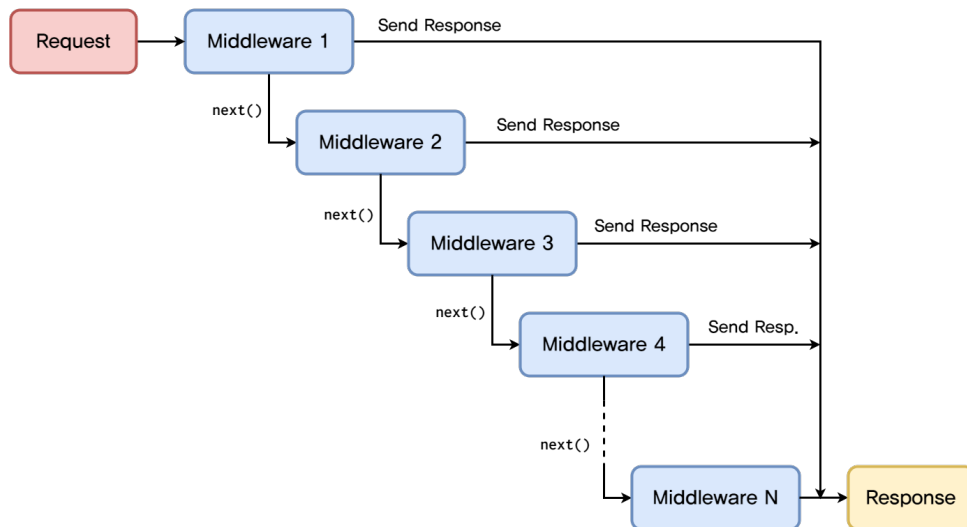
**Code 3.2**는 두 미들웨어를 **use 메서드를 이용하여 웹 서버 애플리케이션에 bind하는 코드**이다. 첫 번째 미들웨어는 0 이상 10 미만 랜덤 정수 randomNumber를 뽑아 출력하고, randomNumber가 3의 배수이면 403 Forbidden 에러를 HTTP 응답으로 보내고, 3의 배수가 아니면 다음 함수(next)를 호출한다. 두 번째 미들웨어는 요청 객체(req)에서 요청의 메서드와 경로를 가져와 라우트를 출력하고, 다음 함수를(next)를 호출한다. 그 이후 라우팅을



통해 각 요청에 맞는 응답을 보낸다.

이 예제를 실행해보면 다음과 같은 몇 가지 사실을 관찰할 수 있다. 먼저 미들웨어는 `use` 메서드를 통해 bind된 순서대로 실행되며, 후술하겠지만 라우팅 메서드 역시 미들웨어로 취급되고 미들웨어 형태로 bind될 수 있다. 두 번째, 미들웨어 역시 요청 객체(`req`)와 응답 객체(`res`)에 접근할 수 있다. 세 번째, 미들웨어에서 `next` 함수를 호출하면 그 다음 미들웨어가 실행되며, 그렇지 않고 응답 객체(`res`)를 이용하여 응답을 보내게 되면 그 이후의 미들웨어는 더 이상 실행되지 않는다.

이러한 미들웨어의 특징을 도식으로 나타내면 **Figure 3.1**과 같다.



**Figure 3.1** Flow of Code using Middlewares

이러한 미들웨어의 특징과 일련의 미들웨어의 순차적 실행이라는 Express.js의 특징으로 인해 Express.js 애플리케이션에서 미들웨어는 매우 중요한 역할을 한다. 요청 객체에 대한 전처리, 로그 기록, 에러 처리<sup>1</sup>, 라우팅 등은 모두 미들웨어의 형태로 작성할 수 있으며, Express.js 애플리케이션을 위해 개발된 모듈은 미들웨어 형태로 되어있는 것이 대부분이다.

라우팅은 `express` 모듈의 `Router` 객체를 이용하여 미들웨어 형태로 작성될 수 있다. `router.js`를 생성하고 **Code 3.3**과 같이 작성한다.

**Code 3.3** Routing with Middleware - `router.js`

```
const { Router } = require('express');

const router = Router();

router.get('/', (req, res) => res.send('GET /'));
router.post('/', (req, res) => res.send('POST /'));

module.exports = router;
```

<sup>1</sup>미들웨어를 이용한 에러 처리는 7장에서 다룰 예정

이후 index.js를 **Code 3.4**와 같이 작성한다.

**Code 3.4** Routing with Middleware - index.js

```
const express = require('express');
const router = require('./router');
const port = 3000;

const app = express();

app.use('/', router);

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

이 프로그램은 라우팅 기능을 미들웨어 형태로 router.js에 구현하고, 이를 index.js에서 모듈 형태로 import하여 app 객체에 추가하여 구현되었다. 이렇게 Router 객체와 모듈을 이용하면 라우팅과 관련된 코드를 분리하여 프로젝트를 구조화할 수 있다.

**Code 3.4**의 use 메서드에는 두 인자가 있는데, 두 번째 인자인 미들웨어는 첫 번째 인자인 HTTP 경로로 시작하는 요청 경로에 대해서만 실행되며, 첫 번째 인자가 생략되면 모든 경로에 대해 실행된다. 즉, **'/' 인자는 /로 시작하는 경로만 router에서 처리하라는 뜻으로**, 이 값을 **'/api'**로 바꾸면 GET /api, POST /api 등의 라우트로 접속해야 적절한 응답을 받을 수 있다. 여기에 router.js에서 get 메서드의 인자를  **'/'**에서  **'/post'**로 바꾸면 GET /api/post로 요청을 보내야 “GET /” 응답을 받을 수 있다.

## 3.2 View Engine

### Necessity of View Engine

3.1절에서는 Express.js를 이용하여 HTTP 요청을 받고 그 요청에 따라 plain text 형태로 적절한 HTTP 응답을 보내는 웹 서버를 구현하였다. 그러나 일반 사용자가 웹 사이트를 이용하기 위해서는 브라우저에서 응답 내용을 렌더링할 수 있도록 HTML 형태의 응답을 보내야 한다. 이러한 서버는 Code 3.5와 같이 구현될 수 있다.

#### Code 3.5 Express Server Responding HTML

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => res.send(`
  <h1>GET /</h1>
  <h3>Hello World!</h3>
`));

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

Code 3.5는 의도한 기능을 문제없이 수행하는 코드이다. 그러나 이렇게 HTML 문서를 직접 코드에 넣어 응답하는 방식은 여러 문제점을 갖는다. 먼저 가장 큰 문제점은 코드의 가독성을 매우 저하시킨다는 점이다. 대부분 웹 페이지의 HTML 문서는 매우 긴데, 이러한 HTML 문서를 그대로 코드 내에 하드 코딩 해버리면 코드에서 잘 나타나야 하는 로직과 흐름이 잘 보이지 않게 된다. 또한, HTTP 요청의 처리 결과에 따라 문서 내의 값이 바뀌거나, 특정 부분이 조건에 따라 바뀌거나, 반복되어야 할 때는 JS 문법을 이용하여 HTML 문서를 생성해야 하는데, HTML 문서와 JS 코드가 난잡하게 섞여 가독성이 저하될 수밖에 없다.

상기한 문제점은 HTML 문서를 응답으로 보낼 때 view engine으로 템플릿 파일을 작성 사용함으로써 해소될 수 있다. 대표적인 view engine으로는 pug, jinja2, ejs 등이 있으며, 이 중 본 교재에서는 간결하여 가독성이 좋고 간편한 pug를 사용한다. 새 Node.js 애플리케이션을 생성하고, Express.js에서 pug로 작성된 파일을 HTML 문서로 렌더링할 수 있도록 Shell 3.2와 같이 pug 모듈을 express 모듈과 함께 설치한다.

#### Shell 3.2 Installing pug

```
$ npm install express pug
```

index.js를 생성하고 Code 3.6과 같이 작성한다. app.set 메서드를 이용하여 프로젝트 디렉토리(\_\_dirname)의 views 디렉토리를 템플릿 파일들을 모아둔 디렉토리로 설정하고, pug를 view engine으로 설정한다. 이제 GET /, GET /page/:page, GET /posts/:until의 controller 함수를 각각 구현하며 pug 문법을 알아보자.

### Code 3.6 pug Example Application - index.js

```
const express = require('express');

const port = 3000;
const app = express();

app.set('views', `${__dirname}/views`);
app.set('view engine', 'pug');

app.get('/', (req, res) => {});

app.get('/page', (req, res) => {});

app.get('/posts', (req, res) => {});

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

## Basic Syntaxes

먼저 views/index.pug를 생성하여 **Code 3.7**과 같이 작성한다.

### Code 3.7 pug Example Application - views/index.pug

```
doctype html
html
  head
    title Index Page
    meta(charset='utf-8')
  body
    .page-header
      h1#page-title Index Title
      ul
        li
          a(href='/board') Go to Board
        li
          a(href='/article/compose') Compose New Article
    .page-body
      | Page Body
      | Hello World!
```

**Code 3.7**을 통해 pug는 다음과 같은 특징을 갖는다는 것을 알 수 있다.

- HTML 태그는 태그의 이름만 작성하고, 태그의 열고 닫음은 들여쓰기(indent)로 표현한다.
- class와 id는 CSS 선택자를 작성하는 방식과 동일하게 작성하며, class나 id가 있는 div 태그는 태그 이름을 생략할 수 있다. 여타 태그는 반드시 태그명을 명시해야 한다.
- class, id를 제외한 요소의 속성(attribute)과 속성값(value)은 괄호 내에 attribute=value의 형태로 작성하며, 각 key-value pair는 쉼표(,)로 구분한다.

- 태그 내부의 내용이 한 줄이면 태그 부분 뒤에 한 칸을 띄고 내용을 작성한다.
- 태그 내부의 내용이 여러 줄이면 다음 줄부터 줄마다 pipe(|)를 삽입한 뒤 한 칸을 띄고 내용을 작성한다.

**Code 3.8** pug Example Application - index.js (GET /)

```
app.get('/', (req, res) => res.render('index.pug'));
```

**Code 3.6**에서 GET /의 controller 함수를 **Code 3.8**과 같이 수정한다. **render** 메서드에 템플릿 파일의 상대 경로를 인자로 전달하면 템플릿 파일이 pug 모듈에 의해 HTML 문서로 변환(render)되어 응답으로 보내지게 된다. 이때 템플릿 파일의 상대 경로는 **Code 3.6**에서 **set** 메서드를 통해 설정한 템플릿 디렉토리를 기준으로 한다.

이제 서버를 실행하고, GET /에 요청을 보내 템플릿 문서가 잘 render되어 응답되는지 확인한다.

## Interpolation Syntaxes

**Code 3.9** pug Example Application - views/board.pug

```
doctype html
html
  head
    title Board Page
  body
    h1 Board Page
    h3 This is page #{page}
```

views/board.pug를 생성하여 **Code 3.9**와 같이 작성한다. 2.1절에서 다룬 template literal과 유사하게 pug에서는 **#{}** 표현을 이용하여 템플릿에 JS 표현식의 반환값을 삽입한다. **page**와 같이 템플릿에서 사용되는 값은 객체의 형태로 **render** 메서드의 두 번째 인자에 전달하면 해당 부분이 JS 표현식의 반환값으로 대체되어 render된다.

**Code 3.10** pug Example Application - index.js (GET /page)

```
app.get('/page', (req, res) => {
  const { page } = req.query;
  res.render('board.pug', { page });
});
```

**Code 3.6**에서 GET /page의 controller 함수를 **Code 3.10**과 같이 수정하여 **render** 메서드의 인자로 **page**의 값을 전달한다. 그리고 GET /page?page=1, GET /page?page=2 등으로 요청을 보내어 템플릿 문서가 잘 render되는지 확인한다.

## Conditional and Iteration Syntaxes

**Code 3.11** pug Example Application - views/posts.pug

```
doctype html
html
  head
    title Posts Page
  body
    if posts.length
      h1 Posts Count: #{posts.length}
      each post, index in posts
        h3 ##{index} - #{post}
    else
      h1 Invalid Page!
```

views/posts.pug를 **Code 3.11**과 같이 작성한다. 이처럼 pug 템플릿에서는 if-else문 등을 이용하여 조건문을, each-in문 등을 이용하여 반복문을 사용할 수 있다.

**Code 3.12** pug Example Application - index.js (GET /posts)

```
app.get('/posts', (req, res) => {
  const { until } = req.query;
  const untilParsed = parseInt(until, 10);

  const posts = [];
  if (!isNaN(untilParsed)) {
    for (let i = 0; i < untilParsed; i++) {
      posts.push(`Post ${i + 1}`);
    }
  }
  res.render('posts.pug', { posts });
});
```

**Code 3.6**에서 GET /posts의 controller 함수를 **Code 3.12**와 같이 수정하여 `posts` 값을 `render` 메서드의 인자로 전달한다. 이후 GET /posts?until=10 요청에 대한 응답과 GET /posts?until=tenth 요청에 대한 응답을 비교해본다.

pug의 자세한 문법은 공식 홈페이지<sup>2</sup>에서 확인할 수 있다.

---

<sup>2</sup><https://pugjs.org/api/getting-started.html>

### 3.3 Sending POST Request From Browser

#### form Tag

지금까지는 Express.js의 작동 방식을 이해하기 위해 Insomnia를 이용하여 웹 서버에 POST 요청을 보냈다. 그러나 대부분의 사용자는 이러한 프로그램을 사용하지 않고, 인터넷 브라우저에 띄워진 웹 페이지에서 요청을 보내게 되며, 따라서 HTML 문서에서 body 데이터와 함께 POST 요청을 보낼 수 있어야 한다.

form 태그는 HTML 문서에서 GET과 POST 요청을 보내는 기능을 수행할 수 있는 태그이다. form 요소 내에 태그가 input이나 button이고, type 속성의 값이 submit인 버튼을 클릭했을 때 form 요소의 method 속성의 값을 메서드, action 속성의 값을 경로로 정하여 HTTP 요청을 보낸다.

이때 form 요소의 body는 요소 내부의 모든 입력 요소(input 또는 textarea 태그)의 데이터를 보내게 되는데, 입력 요소의 name 속성의 값이 key, value 속성의 값이 value가 된다.

#### Code 3.13 form Tag Example

```
<form method="post" action="/login">
  <div>
    <label>Username:</label>
    <input id="username-input" name="username" type="text">
  </div>
  <div>
    <label>Password:</label>
    <input id="password-input" name="password" type="password">
  </div>
  <div>
    <div>Introduce yourself</div>
    <textarea id="introduction-input" name="introduction"></textarea>
  </div>
  <button type="submit">Submit</button>
</form>
```

Code 3.13은 form 요소를 작성한 예제로, 사용자가 “Submit” 버튼을 클릭하면 #username-input, #password-input, #introduce-input 요소의 value 값이 각각 HTTP 요청의 body(GET 요청일 경우 query)에 username, password, introduction 속성의 값이 되고, 이 요청이 POST /login으로 보내지게 된다.

여담으로, form 태그에서 method 속성의 기본값은 get, action 속성의 기본값은 현재 페이지의 경로이다.

## 3.4 Express.js and View Engine Exercises

### Exercise 3.1: Access to Query and Body

GET 요청을 보낼 때는 query를 통해 서버에 데이터를 보내고, POST, PUT, DELETE 등의 요청을 보낼 때는 body를 통해 데이터를 보낸다.

#### Code 3.14 Express urlencoded Config

```
app.use(express.urlencoded({ extended: true }));
```

Express.js에서는 요청 객체를 통해 body 데이터를 정상적으로 받기 위해서는 **Code 3.14**와 같이 미들웨어 설정을 해야 한다.

#### Code 3.15 Stringify Mappable Object

```
Object.keys(obj).map(k => `${k}: ${obj[k]}`).join('\n');
```

GET / 요청을 받으면 query 데이터를, POST /, PUT /, DELETE / 요청을 받으면 body 데이터를 문자열 형태로 변환하여 응답으로 반환하는 웹 서버를 구현하여야. 객체 obj는 **Code 3.15**와 같이 문자열로 변환될 수 있다.

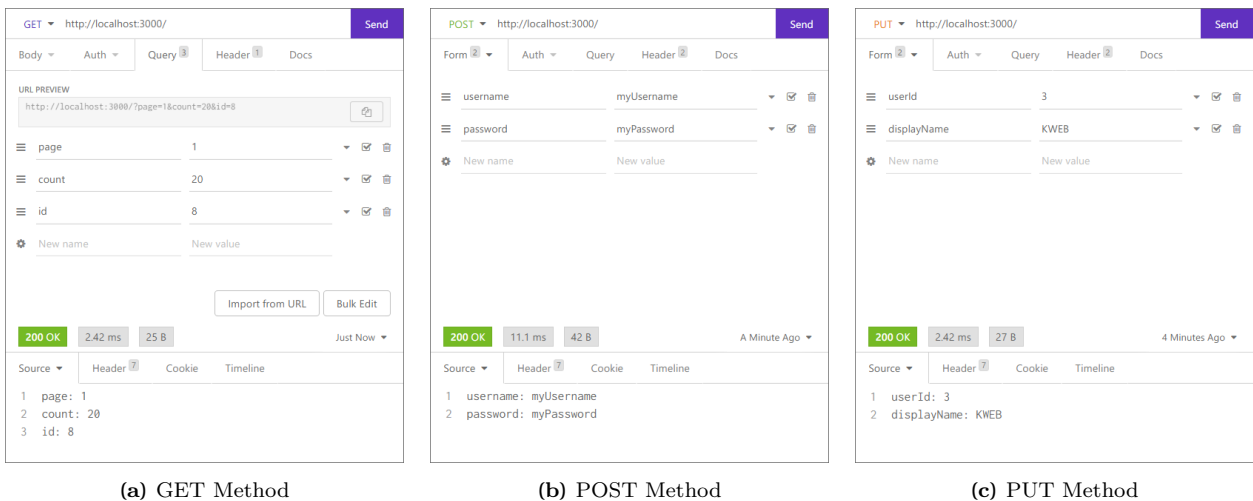


Figure 3.2 Example Results of Exercise 3.1

Insomnia를 이용하여 **Figure 3.2**와 같이 잘 동작하는지 확인해본다. Body 데이터는 [Body] 탭에서 “Form URL Encoded”를 선택한 뒤 데이터를 입력하여 보낼 수 있다.



### Exercise 3.2: Access to URL Parameters

웹 서버는 필요에 따라 데이터를 query와 body가 아닌 경로에 직접 받을 수도 있다. 예를 들어 게시판의 10번째 페이지를 GET /board?page=10 라우트를 통해 요청받을 수도 있지만, GET /board/page/10 라우트로 요청받도록 할 수도 있다. 이러한 형태의 URL을 semantic URL이라고 하고, semantic URL 경로의 가변적인 부분은 변수로 받을 수 있다.

위에서 제시된 예시 경로 /board/page/10는 /board/page/:page의 형태로 경로를 제시하면 match됩니다. 앞에 콜론(:)이 붙은 부분은 그 부분이 가변적인 부분이라는 뜻이며, 정규표현식으로 원하는 형태의 변수만 허용할 수 있다.<sup>3</sup> 이러한 변수의 값은 요청 객체의 `params` 속성에 저장되어 있으며, 예시 경로에서 10이라는 값은 `req.params.page`에 할당되어 있다. 이를 이용하여 GET /board/page/:page 라우트로 요청을 받았을 때 페이지 번호를 표시하는 문자열을 생성하여 응답하는 Express.js 서버 애플리케이션을 구현하여라.

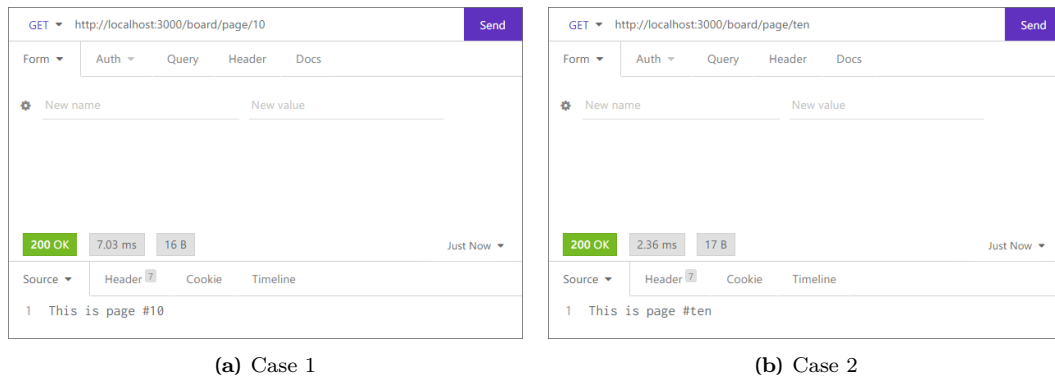


Figure 3.3 Example Results of Exercise 3.2

### Exercise 3.3: Factorial Server with Redirection

number 값을 받아 그 값의 팩토리얼 값을 텍스트로 응답하는 GET /factorial 라우트와 GET /factorial/:number 라우트를 구현하여라. 이때 GET /factorial은 number 값을 query 형태로 받아 GET /factorial/:number 라우트로 리다이렉트한다. 예를 들어, GET /factorial?number=10으로 들어온 요청은 GET /factorial/10으로 리다이렉트되어야 한다.

### Exercise 3.4: Login Server with form Tag

47쪽의 **Code 3.13**을 참고하여 로그인할 수 있는 웹 페이지를 응답하는 GET / 라우트와 로그인 페이지에서 받은 username과 password 값을 텍스트로 응답하는 POST /login 라우트의 controller를 구현하여라. 이때 사용자로부터 body 데이터를 제대로 받기 위해서 **Code 3.14**와 같이 미들웨어를 설정하여라.

<sup>3</sup>7장에서 다룰 예정



## Chapter 4

# Database Designing

### Contents

4.1	Introduction to Database . . . . .	52
4.2	MariaDB . . . . .	54
4.3	Basics of Designing . . . . .	59
4.4	Relational Designing . . . . .	63
4.5	Database Designing Exercises . . . . .	67

## 4.1 Introduction to Database

### Necessity of Database

프로그램은 그 내에서 필요한 데이터의 값을 기본적으로 메모리에 저장하고, 접근하여 사용한다. 메모리에 저장된 데이터는 access time이 매우 빨라 성능 면에서는 유리하지만 휘발성(volatile) 기억장치이므로 기기가 종료되면 데이터가 사라지게 되며, 용량 대비 가격이 매우 비싸기 때문에 대용량의 데이터를 저장하기에는 적합하지 않다.

대부분의 웹 서버에서 다루는 데이터는 프로그램이 종료되더라도 사라지면 안되고, 때로는 그 크기가 매우 크기 때문에 데이터를 메모리에 저장하는 것은 적절하지 않다. 예를 들어 웹 사이트를 이용하는 사용자의 아이디나 비밀번호, 사용자가 작성한 게시물 등이 서버가 종료되었다고 해서 사라진다면 제대로 된 서비스라고 할 수 없다. 특히 금융기관의 서버에서 고객들의 송금 내역, 통장 잔고 등의 데이터나 온라인 쇼핑몰 서버에서 고객들의 주문 내역 등이 사라진다면 금전적인 피해로 이어질 수 있다. 따라서 웹 서버는 프로그램이 종료되더라도 데이터가 유실되지 않는 비휘발성(non-volatile) 기억장치에 데이터를 저장해야 한다.

SSD, HDD와 같은 디스크는 대표적인 비휘발성 기억장치이다. 많은 프로그램은 디스크에 파일을 써서 프로그램이 종료되더라도 데이터가 유실되지 않도록 저장하며, 이러한 파일을 읽어 저장된 데이터를 사용한다. 그러나 plain text 형태로 데이터를 저장하는 방식은 매우 많은 데이터를 다루기 어렵다는 점, 여러 사용자가 동시에 수정할 수 없다는 점 등 많은 문제점을 갖는다.

웹 서버를 비롯한 수많은 애플리케이션은 대용량의 정형화된 데이터를 저장하기 위해 데이터베이스(database; DB)를 사용한다. DB는 저장된 데이터를 프로그램이나 컴퓨터가 종료되더라도 안전하게 보관될 뿐만 아니라, 수많은 데이터를 빠르게 탐색할 수 있는 기능, 여러 사용자가 DB의 데이터를 읽거나 쓸 수 있는 기능, 변경된 데이터를 DB에 거의 실시간으로 반영하는 기능 등 수많은 기능을 제공한다.

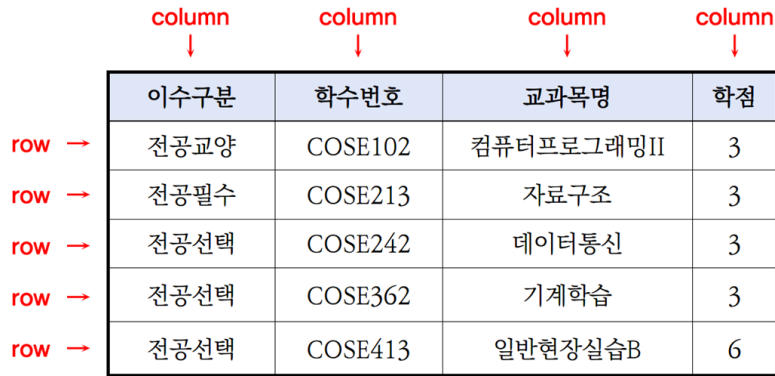
### CRUD Functions

CRUD 기능은 데이터를 다루는 프로그램이 기본적으로 갖추고 있어야 하는 데이터 처리 기능인 Create(생성), Read(조회), Update(수정), Delete(삭제)를 묶어 부르는 용어이다. Create는 데이터를 새로 생성하는 기능, Read는 저장된 데이터를 조회하여 읽는 기능, Update는 이미 저장된 데이터를 수정하는 기능, Delete는 저장된 데이터를 삭제하는 기능을 의미한다. DB 역시 이러한 CRUD 연산에 대응되는 기능을 제공함으로써 데이터를 편리하게 다룰 수 있도록 한다.

여담으로, HTTP에도 CRUD 기능에 대응되는 메서드가 있다. GET은 Read(조회), POST는 Create(생성), PUT은 Update(수정), DELETE는 Delete(삭제) 기능에 해당된다.

## Relational DB

Relational DB, 즉 관계형 데이터베이스(이하 RDB) 모델은 체계화된 DB가 개발된 이래 오늘날까지도 가장 널리 쓰이고 있는 DB 모델로, key와 value들로 이루어진 데이터의 형태를 테이블로 나타내고, row와 column의 형태로 관리하는 모델이다. Row는 각각의 데이터를 나타내며, record라고도 한다. Column은 각 데이터를 구성하는 속성값으로, attribute나 field라고도 하며, 각 데이터의 이름은 column 이름이라고 한다.



The diagram shows a table with four columns and five rows. Above the table, the word 'column' is written in red above each of the four columns, with a red arrow pointing down to the column header. To the left of the table, the word 'row' is written in red to the left of each of the five rows, with a red arrow pointing right to the row header. The table itself has a light blue header row and white data rows with black borders.

이수구분	학수번호	교과목명	학점
전공교양	COSE102	컴퓨터프로그래밍II	3
전공필수	COSE213	자료구조	3
전공선택	COSE242	데이터통신	3
전공선택	COSE362	기계학습	3
전공선택	COSE413	일반현장실습B	6

Figure 4.1 Example of RDB Model Table

Figure 4.1은 컴퓨터학과의 전공과목 목록의 일부를 RDB 모델로 나타낸 것이다. 각 전공과목의 정보는 row 단위로 기록되어 있고, 각 row의 이수구분, 학수번호, 교과목명, 학점 등의 정보가 각 column에 저장되어 있어 모든 row의 데이터 형태가 일정하다. 이러한 row와 column의 집합을 table이라고 하고, 하나의 DB는 다수의 table을 가질 수 있다.

RDB 모델은 이처럼 데이터가 매우 일관적으로 저장되어 있고 직관적이라는 장점이 있으며, 그로 인해 데이터의 분류, 정렬, 탐색 속도가 매우 빠르다는 장점이 있다. 이렇듯 RDB 모델은 가장 기본적이고 대중적인 DB 모델이며, 대부분의 상용 DB는 RDB 모델로 설계되어 있고, 많은 웹 서버에서도 RDB 모델을 기반으로 데이터를 저장한다.

## SQL

SQL(Structured Query Language)은 대부분의 RDB형 DB에서 데이터를 다루기 위해 사용되는 스크립트 언어로, SQL을 이용하여 DB나 table을 생성하거나 CRUD 기능 등의 작업을 수행할 수 있다. 물론 DBeaver, Beekeeper Studio, HeidiSQL와 같이 SQL을 굳이 사용하지 않고도 DB를 관리할 수 있는 프로그램들이 존재하지만, 고수준의 연산을 수행하기 위해서는 여전히 SQL이 필요하고 애플리케이션에서 DB 내의 데이터에 접근할 때에도 여전히 SQL이 필요하다.

## 4.2 MariaDB

### DBMS

DBMS는 데이터베이스 관리 시스템(Database Management System)의 줄임말로, 데이터베이스를 사용하기 편리하도록 만들어진 프로그램이다. DBMS는 데이터를 다루는 작업, 데이터베이스 생성, 사용자 생성, 사용자 권한 설정 등 데이터베이스 전반을 관리하고 호스팅하는 역할을 한다.

RDB형 DBMS에는 대표적으로 MySQL, Oracle, MSSQL, PostgreSQL 등이 있고, 이 중 MySQL은 오픈 소스 소프트웨어라는 점, 표준에 가까운 SQL을 사용한다는 점 등으로 인해 RDB형 DBMS 중 가장 높은 점유율을 자랑해왔다. 그러나 MySQL이 오픈 소스에 적대적인 Oracle에 인수되어 라이선스 상태가 불안정해지자 MySQL의 개발팀이 MySQL을 기반으로 MariaDB라는 DBMS를 개발하였다. MariaDB는 MySQL과 완벽히 호환되며 최근에는 MySQL의 상위호환으로 거듭나 MariaDB의 사용률이 급증하는 추세이다. 따라서 본 교재에서는 가장 최신 LTS 버전<sup>1</sup>을 이용하여 DB와 관련된 실습을 진행한다.

### Installation (Windows)

MariaDB 홈페이지(<https://downloads.mariadb.org/>)에서 자신의 OS 버전과 맞는 msi 패키지를 다운로드하여 실행한다.

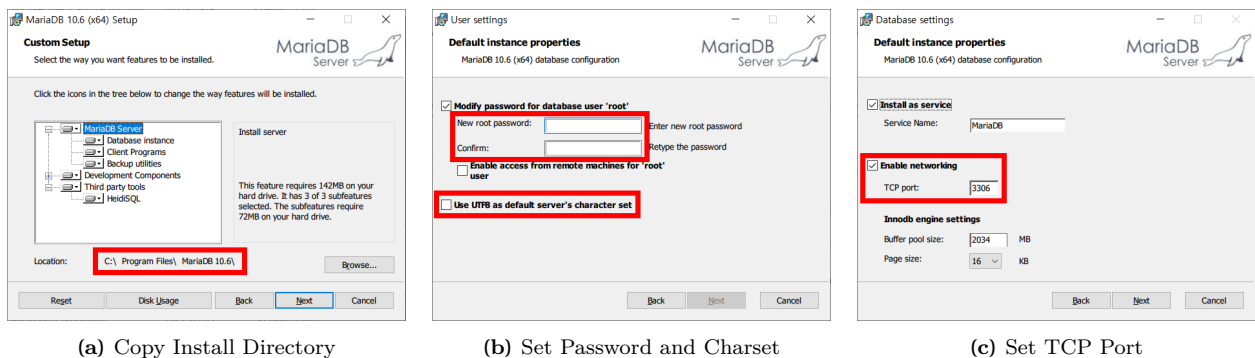


Figure 4.2 Installation of MariaDB server on Windows

먼저 Figure 4.2a와 같이 DBMS의 설치 경로<sup>2</sup>를 미리 파악해둔다. 이후 Figure 4.2b와 같이 root 계정(관리자 계정)의 비밀번호를 설정하고 문자 집합을 UTF8로 체크하여 한글 및 기타 특수문자를 사용할 수 있도록 설정한다. root 계정은 DBMS 내의 모든 DB와 데이터를 다룰 수 있는 권한을 가지고 있고, root 비밀번호를 분실하면 DBMS를 재설치해야 하므로 신중하게 정하고 기억해야 한다.

마지막으로 Figure 4.2c와 같이 MariaDB에 접속하기 위한 port 번호를 TCP port란에 설정한다. MariaDB는 기본적으로 3306번 port를 사용하며, 여타 이유로 선점되어 있다면 3307번 등 다른 port로 변경한다.<sup>3</sup>

<sup>1</sup>2022년 9월 기준 10.6.10

<sup>2</sup>10.6 버전의 경우 C:\Program Files\MariaDB 10.6

<sup>3</sup>보안을 위해 일부러 다른 port로 바꾸기도 한다.

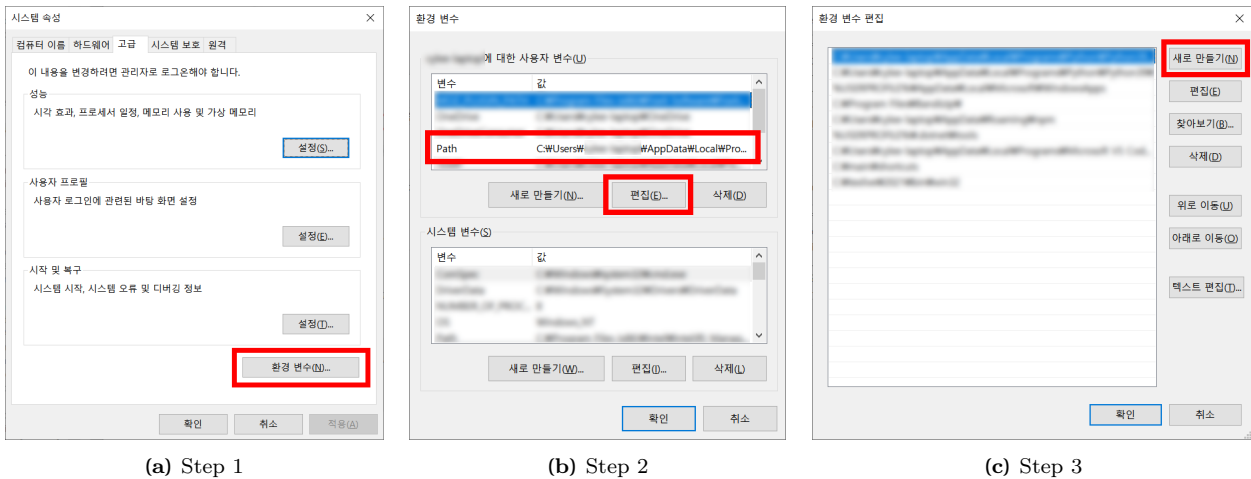


Figure 4.3 Setting Environment Variable

이제 윈도우의 shell인 cmd에서 MariaDB에 접속할 수 있도록 환경 변수를 설정한다. 먼저 윈도우 탐색기를 열고, **Figure 4.2a**에서 파악한 설치 경로를 연다. 설치 폴더 아래의 bin 폴더에 mysql.exe 파일이 있는 것을 확인한 뒤 bin 폴더의 전체 경로<sup>4</sup>를 복사한다. 이후 윈도우 검색을 이용해 [시스템 환경 변수 편집]을 실행하고, **Figure 4.3**과 같이 환경 변수 편집 창에 진입한 뒤, [새로 만들기] 버튼을 클릭하여 bin 폴더의 전체 경로를 붙여넣어 추가한다.

#### Shell 4.1 Checking MariaDB Installation

```
$ mysql --version
```

이제 cmd 창을 열고, **Shell 4.1**과 같이 MariaDB가 제대로 설치되었는지 확인한다.

## Installation (macOS)

#### Shell 4.2 Installing Homebrew

```
$ /bin/bash -c \
  "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
$ brew update
```

macOS에서는 먼저 **Shell 4.2**를 따라 패키지 관리자인 Homebrew를 설치하고, 설치되었는지 확인한다.

#### Shell 4.3 Installing MariaDB (macOS)

```
$ brew info mariadb
$ brew install mariadb
$ brew services start mariadb
```

Homebrew를 이용해 **Shell 4.3**을 참고하여 MariaDB를 설치한 뒤 실행한다. 자세한 설치 방법은 MariaDB에서

<sup>4</sup>10.6 버전의 경우 C:\Program Files\MariaDB 10.6\bin

제공하는 설치 가이드<sup>5</sup>를 참고하여 설치한다. 설치가 완료되면 **Shell 4.1**과 같이 확인해본다.

## Installation (Ubuntu)

### Shell 4.4 Installing MariaDB (Ubuntu)<sup>a</sup>

```
$ sudo apt-get install software-properties-common
$ sudo apt-key adv --recv-keys --keyserver \
    hkps://keyserver.ubuntu.com:80 0xF1656F24C74CD1D8
$ sudo add-apt-repository 'deb [arch=amd64,arm64,ppc64el] \
    http://mariadb.mirror.liquidtelecom.com/repo/10.6/ubuntu bionic main'
$ sudo apt install mariadb-server
```

<sup>a</sup>MariaDB 10.6 기준

대표적인 Linux 계열의 OS인 Ubuntu에서는 먼저 **Shell 4.4**와 같이 MariaDB 웹 사이트에서 repository를 다운로드하고, 이를 apt를 이용하여 설치한다.

### Shell 4.5 Configuration of MariaDB (Ubuntu)

```
$ sudo mysql_secure_installation
Enter current password for root (enter for none): // your root password here
OK, successfully used password, moving on...
Switch to unix_socket authentication [Y/n] n
Change the root password? [Y/n] n
Remove anonymous users? [Y/n] y
Disallow root login remotely? [Y/n] n
Remove test database and access to it? [Y/n] y
Reload privilege tables now? [Y/n] y
Thanks for using MariaDB!
```

이후 **Shell 4.5**와 같이 MariaDB를 설정하고, **Shell 4.1**과 같이 확인해본다.

## Accessing MariaDB Server

### Shell 4.6 Basic Command Logging In MariaDB

```
$ mysql -u<username> -p<password>
```

이제 shell을 통해 MariaDB에 접속할 수 있다. **Shell 4.6**은 MariaDB에 로그인하는 기본적인 명령어로, 현재는 root 계정만 존재하므로 root 계정으로 로그인한다. <username> 부분을 root으로, <password> 부분을 root 계정의 비밀번호로 바꾸어 명령어를 작성한다.

---

<sup>5</sup><https://mariadb.com/resources/blog/installing-mariadb-10-1-16-on-mac-os-x-with-homebrew/>



#### Shell 4.7 Accessing root User using sudo

```
$ sudo mysql
```

Unix 계열의 OS에서는 **Shell 4.6**의 방법으로 root 계정에 접속하지 못하는 경우가 있는데, 이때 **Shell 4.7**과 같이 sudo 권한으로 실행하면 자동으로 root 계정으로 접속된다.

#### Shell 4.8 Querying List of Databases

```
MariaDB [(none)]> SHOW DATABASES;
```

**Shell 4.8**과 같이 SQL을 사용하여 현재 DBMS에 있는 데이터베이스의 목록을 조회할 수 있다. 커서 부분의 대괄호([]) 내부에는 현재 사용자가 작업 중인 데이터베이스의 이름이 표시되며, 현재는 작업 중인 데이터베이스가 없으므로 (none)으로 표시된다.

#### Shell 4.9 Creating and Using Database

```
MariaDB [(none)]> CREATE DATABASE kweb_db;  
MariaDB [(none)]> USE kweb_db;  
MariaDB [kweb_db]>
```

**Shell 4.9**와 같이 CREATE DATABASE 키워드를 이용하여 kweb\_db라는 데이터베이스를 생성할 수 있고, USE 키워드를 이용하여 작업 중인 데이터베이스를 kweb\_db로 바꿀 수 있다.

본 교재에서는 이후 DB shell을 표시할 때 커서 부분에서 DBMS와 DB의 이름은 생략한다.

## Database User

DBMS에는 여러 DB를 생성할 수 있고, 하나의 애플리케이션은 하나의 DB를 사용하는 것이 원칙이므로 여러 애플리케이션이 하나의 DBMS에 접속할 수 있다. root 계정은 소위 관리자 계정에 해당하는 계정으로 DBMS 전체를 조작할 수 있고 모든 DB에 접근하여 데이터를 조작할 수 있다. 여러 애플리케이션에서 각자의 DB를 사용하기 위해 모두 root 계정으로 접속하면 다른 애플리케이션이 사용하는 DB의 데이터를 수정하거나 DB 자체를 없애버리는 등의 조작이 일어나는 등 보안상의 문제가 발생할 수 있다. 따라서 관리자는 애플리케이션마다 적어도 하나의 사용자 계정을 생성하고, 각 사용자의 권한을 제한하여 DB 전체가 공격받는 일이 없도록 해야한다.

#### SQL 4.1 Create User

```
CREATE USER '<username>'@'<host>' IDENTIFIED BY '<password>'
```

먼저 root 계정으로 DB에 로그인한 후, **SQL 4.1**과 같이 사용자 계정을 생성한다. 이때 host는 사용자가 접속하고자 하는 host로, 생성된 계정은 host에 해당하는 컴퓨터에서만 접속할 수 있다. host 값을 %로 설정하면 모든 host에서 접속할 수 있다.

#### Shell 4.10 Create kwebuser User

```
CREATE USER 'kwebuser'@'%' IDENTIFIED BY 'kwebpw';
```

Shell 4.10은 모든 host로부터 접근할 수 있고, 비밀번호는 “kwebpw”인 “kwebuser” 계정을 생성하는 예시이다.

#### Shell 4.11 Create Two DBs

```
> CREATE DATABASE kwebdb1;  
> CREATE DATABASE kwebdb2;
```

Shell 4.11과 같이 두 DB를 생성하고, “kwebuser” 사용자에게 kwebdb1의 권한만 부여해보자.

#### SQL 4.2 Grant All Privileges to User

```
GRANT ALL PRIVILEGES ON <db-name>.* TO '<username>'@'<host>'
```

SQL 4.2는 사용자에게 db-name DB의 모든 권한을 부여하는 SQL문이다. 모든 권한 외에도 옵션에 따라 특정 권한만 부여할 수 있다.

#### Shell 4.12 Grant All Privileges on kwebdb1 Table to “kwebuser” User

```
> GRANT ALL PRIVILEGES ON kwebdb1.* TO 'kwebuser'@'%' ;
```

Shell 4.12와 같이 kwebuser 사용자에게 kwebdb1 DB의 권한만 부여한다. 이제 root 계정에서 로그아웃하고 Shell 4.6을 참고하여 kwebuser 사용자로 로그인해본다.

#### Shell 4.13 Accessing DB with “kwebuser” User

```
$ mysql -ukwebuser -pkwebpw  
> USE kwebdb1;  
> USE kwebdb2;
```

Shell 4.13과 같이 “kwebuser” 사용자로 로그인한 후, kwebdb1 DB와 kwebdb2 DB에 각각 접속을 시도해보면 kwebdb1에는 접속할 수 있으나 kwebdb2에는 접속이 불가능하다는 것을 확인할 수 있다.

## 4.3 Basics of Designing

### Data Types

DB의 모든 column은 자료형(data type)을 가지며, 개발자는 각 column에 적절한 자료형을 부여해야 한다.<sup>6</sup>

Table 4.1 Data Types of MariaDB (MySQL)

Category	Data Types
문자형	CHAR ( $n < 2^8$ ) / <b>VARCHAR</b> ( $n < 2^{16}$ ) / TINYTEXT ( $n < 2^8$ ) <b>TEXT</b> ( $n < 2^{16}$ ) / MEDIUMTEXT ( $n < 2^{24}$ ) / LONGTEXT ( $n < 2^{32}$ )
숫자형	<b>TINYINT</b> ( $n < 2^8$ ) / <b>SMALLINT</b> ( $n < 2^{16}$ ) / <b>MEDIUMINT</b> ( $n < 2^{24}$ ) <b>INT</b> ( $n < 2^{32}$ ) / <b>BIGINT</b> ( $\infty$ ) / FLOAT / DECIMAL / <b>DOUBLE</b>
날짜형	DATE / TIME / <b>DATETIME</b> / <b>TIMESTAMP</b> / YEAR
이진 데이터	BINARY / BYTE / TINYBLOB / BLOB / MEDIUMBLOB / LONGBLOB

Table 4.1은 MariaDB(MySQL)에서 제공되는 자료형의 일부를 나타낸 표이며, 굵은 글씨는 주로 사용되는 자료형이다. TINYINT 형은 주로 boolean 값을 나타내기 위해 사용되며, VARCHAR 형은 80자가 기본이지만 문자열의 길이를 항상 명시해주는 것이 권장된다. 이진 데이터는 이미지나 영상 등 텍스트로 표현되지 않는 데이터를 저장하기 위한 자료형이지만, DB에서 데이터 조회 효율을 떨어뜨리므로 권장되지 않는다. 대신, 이진 데이터를 별도의 파일로 저장하고 해당 파일의 경로를 DB에 저장하는 것이 일반적인 방법이다.

### Designing Simple Table

대학 강의들의 데이터를 저장하는 courses 테이블을 설계하면서 간단한 테이블 설계 방법을 알아보자. 강의와 관련된 정보에는 강의명(name), 강의 코드(code), 이수구분(classification), 학점(credit), 시간(period) 등이 있다. 이러한 정보를 정리하여 Table 4.2와 같이 나타낼 수 있다.

Table 4.2 courses Table

name	code	classification	credit	period
모두를위한파이썬	COSE156	교양	3	4
이산수학	COSE211	전공선택	3	3
운영체제	COSE341	전공필수	3	3
기계학습	COSE362	전공선택	3	3
캡스톤디자인	COSE489	전공선택	3	6

<sup>6</sup>자료형과 무관하게 column의 값이 NULL이 될 수 있으나, NULL의 사용은 지양되어야 한다.

먼저 **Table 4.2**의 각 column에 자료형을 부여해보자. **name** column은 문자열로 표현되고 매우 긴 텍스트가 아니므로 VARCHAR형을 사용할 수 있으며, 과목명의 최대 길이가 20자라고 한다면 VARCHAR(20)형으로 정할 수 있다. 유사하게 **code**와 **classification** column은 VARCHAR(8)로 정하면 적당한 길이가 될 것이다. **credit**과 **period**는 각각 정수이므로 INT형이 적당하다. 이렇게 부여한 자료형을 반영하여 **Table 4.3**과 같이 나타낼 수 있다.

**Table 4.3** courses Table with Data Types

name	code	classification	credit	period
VC(20)	VC(8)	VC(8)	INT	INT
모두를위한파이썬	COSE156	교양	3	4
이산수학	COSE211	전공선택	3	3
운영체제	COSE341	전공필수	3	3
기계학습	COSE362	전공선택	3	3
캡스톤디자인	COSE489	전공선택	3	6

## Avoiding Redundancy

**Table 4.3**은 컴퓨터학과에서 열리는 강의 데이터만 저장한 테이블이다. 이제 이 데이터베이스에 컴퓨터학과 이외의 화학과, 물리학과 등의 강의를 추가하려면 어떻게 하는 것이 좋겠는가?

가장 쉽게 떠올릴 수 있는 방법은 동일한 구조의 화학과 강의 테이블, 물리학과 강의 테이블 등을 만드는 것이다. 그러나 이 방법은 두 가지 문제점을 유발하는데, 첫번째는 구조가 동일한 테이블이 여러 개 생성되어 중복을 초래한다는 점이다. 프로그램을 설계할 때 코드의 중복을 최대한 피하기 위해 함수, 클래스의 상속 등을 활용하는 것처럼 DB를 설계할 때에도 중복을 가급적 피해야 한다.

두 번째 문제점은 새로운 학과의 강의를 추가하려면 그 DB를 사용하는 애플리케이션이 매번 테이블을 직접 생성해야 한다는 것이다. 애플리케이션은 DB의 데이터에 대한 CRUD 연산만 수행하여야 하고, 그 외에 테이블의 추가 및 삭제, column의 추가 및 삭제 등 설계 자체를 바꾸는 작업을 수행해서는 안 된다.

이렇듯 분류에 따라 테이블을 추가하는 방법은 지양되어야 하며, 모든 강의 데이터를 모두 한 테이블에 저장하고 분류를 위한 column을 생성하여 분류하는 것이 가장 이상적인 방법이다. **Table 4.4**는 **Table 4.3**에 **department** column을 추가하여 각 강의의 학과를 명시하여 분류한 테이블이다. 이렇게 테이블을 설계하면 앞에서 제시된 문제점들이 발생하지 않고 분류에 따라 데이터를 조회하기도 간편하다.

**Table 4.4** courses Table with Categories

name	department	code	classification	credit	period
VC(20)	VC(16)	VC(8)	VC(8)	INT	INT
모두를위한파이썬	컴퓨터학과	COSE156	교양	3	4
이산수학	컴퓨터학과	COSE211	전공선택	3	3
운영체제	컴퓨터학과	COSE341	전공필수	3	3
기계학습	컴퓨터학과	COSE362	전공선택	3	3
캡스톤디자인	컴퓨터학과	COSE489	전공선택	3	6
유기화학실험	화학과	CHEM214	전공필수	2	4
고체물리학	물리학과	PHYS482	전공선택	3	3
무기화학II	화학과	CHEM308	전공필수	3	3

**Table 4.4**의 테이블에서 `classification` column을 주목해보자. 이 column의 데이터에는 “교양”, “전공필수”, “전공선택” 등이 있는데, 이들 데이터는 전공 여부와 필수 여부로 나누어질 수 있다. 예를 들어, “교양”은 전공도 아니고 필수도 아니며, “전공선택”은 전공이지만 필수는 아니다. 이러한 데이터는 상황에 따라 유의미한 데이터가 될 수 있으며, 이렇게 column의 값을 최대한 유의미한 단위로 나누어 저장하면 중복을 최소화하고 데이터를 편리하게 조회할 수 있다.

**Table 4.5**는 **Table 4.4**의 `classification` column을 `is_major`와 `is_required`로 나눈 테이블이다. 이때 두 column은 참(1)이나 거짓(0) 값을 가져야 하므로 자료형을 `TINYINT(1)`<sup>7</sup>로 설정하였다.

**Table 4.5** courses Table with classification Subdivisions

name	department	code	is_major	is_required	credit	period
VC(20)	VC(16)	VC(8)	TINYINT(1)	TINYINT(1)	INT	INT
모두를위한파이썬	컴퓨터학과	COSE156	0	0	3	4
이산수학	컴퓨터학과	COSE211	1	0	3	3
운영체제	컴퓨터학과	COSE341	1	1	3	3
기계학습	컴퓨터학과	COSE362	1	0	3	3
캡스톤디자인	컴퓨터학과	COSE489	1	0	3	6
유기화학실험	화학과	CHEM214	1	1	2	4
고체물리학	물리학과	PHYS482	1	0	3	3
무기화학II	화학과	CHEM308	1	1	3	3

<sup>7</sup>(1)은 불이나 안 불이나 차이점은 없으나, boolean 형임을 나타내기 위해 관례적으로 쓴다.

## Create Table With SQL

앞에서 설계한 **courses** 테이블(**Table 4.5**)을 SQL을 이용하여 DB에 생성해보자.

### Code 4.1 Create courses Table

```
CREATE TABLE `courses` (  
  `name` VARCHAR(20) NOT NULL,  
  `department` VARCHAR(16) NOT NULL,  
  `code` VARCHAR(8) NOT NULL,  
  `is_major` TINYINT(1) NOT NULL,  
  `is_required` TINYINT(1) NOT NULL,  
  `credit` INT NOT NULL,  
  `period` INT NOT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

**Code 4.1**은 데이터베이스에 **courses** 테이블을 생성하는 SQL문이다. 각 column을 정의할 때는 column의 이름, 자료형, 옵션<sup>8</sup>을 공백문자(whitespace)로 구분하여 나열하고, 각 column을 쉼표(,)로 구분하여 나열한다. SQL 문법에서는 여러 공백 문자도 하나의 공백 문자로 취급(값은 제외)되므로 개행없이 한 줄에 모두 나열하여도 상관없고, CREATE, INT와 같은 예약어는 모두 대문자로 작성하는 것이 원칙이지만 소문자로 작성해도 차이는 없다. Table이나 column 이름 등은 백틱(`)으로 감싸는 것이 원칙이지만 예약어가 아니라면 감싸지 않아도 된다.<sup>9</sup>

### SQL 4.3 Description of Table

```
DESC <table-name>
```

**SQL 4.3**과 같이 DESC 키워드를 사용하면 테이블의 설계 정보를 확인할 수 있다.

### SQL 4.4 Truncate and Drop Table

```
TRUNCATE <table-name>  
DROP TABLE <table-name>
```

**SQL 4.4**와 같이 테이블을 삭제할 수 있다. TRUNCATE 키워드는 테이블의 구조는 유지한 채 데이터만 삭제하고, DROP TABLE 키워드는 테이블 자체를 삭제한다는 차이가 있다.

<sup>8</sup>주어진 SQL문의 NOT NULL 옵션은 NULL 값을 허용하지 않는다는 뜻이다.

<sup>9</sup>예를 들어 column 이름이 student라면 감싸지 않아도 되나 order라면 반드시 감싸야 한다.

## 4.4 Relational Designing

### Relation Between Tables

4.3절에서는 `courses` 테이블을 설계하면서 테이블을 설계하는 가장 기본적인 방법을 배웠다. 정말 간단한 형태의 데이터를 DB에 저장할 때는 기본적인 방법으로도 충분히 데이터를 저장할 수 있으나, 실제 세상은 여러 테이블 간의 관계(relation)가 존재하여 훨씬 복잡하다.

`courses` 테이블 (Table 4.5)의 `department` column에는 학과 강의가 속해있는 학과 이름이 저장되어 있는데, `courses` 테이블과 같이 학과 이름만 저장할 때는 이러한 방법으로도 충분하지만 각 학과에 대한 정보도 저장하고자 할 때는 `courses` 테이블에 온전히 저장할 수 없고, 별도의 테이블에 저장하여야 한다.

위와 같이 테이블 간 관계(relation)가 존재하는 경우에 대한 DB 설계는 매우 중요하며, 실제 서비스에서도 서로 관계가 있는 테이블이 매우 많이 사용된다. 이번 장에서는 관계형 데이터베이스 모델을 이용하여 앞의 `courses` 테이블과 관계가 있는 `departments` 테이블을 설계해본다.

### Primary Key and Foreign Key

먼저 학과와 관련된 `departments` 테이블을 4.3절과 같은 방법으로 설계하면 Table 4.6과 같이 나타낼 수 있다.

Table 4.6 departments Table

kor_name	eng_name	college
VC(16)	VC(50)	VC(16)
컴퓨터학과	Computer Science and Engineering	정보대학
화학과	Chemistry	이과대학
물리학과	Physics	이과대학
언어학과	Linguistics	문과대학

`courses` 테이블과 `departments` 테이블을 비교해보자. 예를 들어 “운영체제” 강의가 속한 학과의 영문 이름을 조회하는 경우 `courses` 테이블에서 `name` 값이 “운영체제”인 row의 `department` 값을 찾아 “컴퓨터학과”임을 알아낸 뒤, `departments` 테이블에서 `kor_name`의 값이 “컴퓨터학과”인 row의 `eng_name` 값을 확인하면 된다. 이때 `courses.department` column이 `department.kor_name` column을 참조(reference)한다고 한다. 이렇게 설계하면 강의 이름을 통해 그 강의가 속한 단과대 이름 등을 다양하게 조회할 수 있다는 장점이 있다.

다만, 이러한 설계는 몇 가지 중요한 문제점을 가지고 있다. 먼저 `courses` 테이블에서 `department` 테이블의 `kor_name` column을 참조하는데, `kor_name` column 값이 같은 row가 존재할 수 있다. 예를 들어, Table 4.6에 다른 학교에 소속된 화학과 학과를 추가한다고 가정하면 “무기화학II” 과목이 속한 학과의 단과대 이름을 조회할 때 우리 학교의 화학과를 참조하는지, 다른 학교의 화학과를 참조하는지 알 수 없다.

또한 참조되는 값이 바뀔 수 있다는 문제도 있다. 예를 들어 departments 테이블에서 “화학과”의 이름이 “화학공학”으로 바뀐다면 courses 테이블에서 “화학과”를 참조하던 “유기화학실험”과 “무기화학II” row의 department 값이 모두 “화학공학”으로 바뀌어야 한다.

위와 같은 문제점을 해결하기 위해 기본 키(primary key)를 사용하는데, 기본 키는 다음 두 조건을 반드시 만족해야 한다.

- 테이블 내에서 기본 키의 값이 동일한 서로 다른 row가 존재하지 않는다. 즉, 모든 기본 키는 해당 테이블 내에서 유일(unique)하다.
- 기본 키의 각 값은 수정되지 않아야 하며, NULL 값이어서도 안 된다.

기본 키는 한 테이블 내에서 하나만 설정할 수 있고, 한 column의 값으로 설정되거나 여러 column 값의 조합으로 설정될 수 있다. 일반적으로 기본 키는 1부터 시작하여 1씩 증가하는 정수로 설정하며, 첫 번째 row의 기본 키 값은 1, 두 번째 row의 기본 키 값은 2 등으로 증가하도록 설정하는 것이 일반적이다. 관계형 데이터베이스에서는 참조되는 테이블이 아니더라도 기본 키를 설정하는 것이 권장된다.

**Table 4.7** departments Table with Primary Key

id	kor_name	eng_name	college
INT	VC(16)	VC(50)	VC(16)
1	컴퓨터학과	Computer Science and Engineering	정보대학
2	화학과	Chemistry	이과대학
3	물리학과	Physics	이과대학
4	언어학과	Linguistics	문과대학

**Table 4.7**은 **Table 4.6**에 기본 키인 id column을 추가한 테이블이다.

**Code 4.2** Create departments Table with Primary Key

```
CREATE TABLE `departments` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `kor_name` VARCHAR(16) NOT NULL,
  `eng_name` VARCHAR(50) NOT NULL,
  `college` VARCHAR(16) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

**Code 4.2**는 **Table 4.7**의 테이블을 DB에 생성하는 SQL문이다. AUTO\_INCREMENT 키워드를 이용하여 데이터를 추가할 때 id 값을 명시하지 않아도 자동으로 직전 row의 id 값보다 1이 큰 값을 할당하도록 하고, PRIMARY KEY 키워드를 이용하여 id column이 기본 키임을 설정한다.

이제 courses 테이블을 수정하여 각 row가 departments 테이블의 id를 참조할 수 있도록 해주자. 이렇게 다른



테이블의 기본 키를 참조하는 키를 외래 키(foreign key)라고 하고, 외래 키의 자료형은 참조하는 키의 자료형과 동일해야 한다. 이 예제에서는 `courses.department` column이 `departments.id` column을 참조하는 외래 키이다.

**Table 4.8** `courses` Table with Primary Key and Foreign Key

id	name	department	code	is_major	is_required	credit	period
INT	VC(20)	INT	VC(8)	TINYINT(1)	TINYINT(1)	INT	INT
1	모두를위한파이썬	1	COSE156	0	0	3	4
2	이산수학	1	COSE211	1	0	3	3
3	운영체제	1	COSE341	1	1	3	3
4	기계학습	1	COSE362	1	0	3	3
5	캡스톤디자인	1	COSE489	1	0	3	6
6	유기화학실험	2	CHEM214	1	1	2	4
7	고체물리학	3	PHYS482	1	0	3	3
8	무기화학II	2	CHEM308	1	1	3	3

**Table 4.8**은 **Table 4.5**에 `id` column을 추가하고 `department` column이 `departments.id` column을 참조하도록 변경한 테이블이다. 이제 “운영체제” 강의가 속한 학과의 단과대 이름을 조회하려면 “운영체제” 강의의 `department` 값이 1임을 찾고, `departments` 테이블에서 `id` 값이 1인 row의 `college` 값을 찾으면 된다.

다른 테이블로부터 참조를 받는 row가 삭제되는 경우 수행할 작업을 on delete action이라고 한다. 예를 들어 `departments` 테이블에서 2번 “화학과” row가 삭제되면 `courses` 테이블에서 이를 참조하는 8번 “무기화학II” 강의는 참조할 데이터가 사라진다 이렇게 row가 삭제되었을 때 이를 참조하는 row를 단계적으로 모두 삭제하는 cascade action을 수행하도록 설정할 수 있고, 이 외에도 set null, no action, set default, restrict 등이 있다.

**Code 4.3** Create `courses` Table with Foreign Key

```
CREATE TABLE `courses` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(20) NOT NULL,
  `department` INT NOT NULL,
  `code` VARCHAR(8) NOT NULL,
  `is_major` TINYINT(1) NOT NULL,
  `is_required` TINYINT(1) NOT NULL,
  `credit` INT NOT NULL,
  `period` INT NOT NULL,
  PRIMARY KEY (`id`),
  FOREIGN KEY (`department`)
    REFERENCES `departments`(`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

**Code 4.3**은 **Table 4.8**의 테이블을 DB에 생성하는 SQL문이다.

지금까지 관계가 있는 테이블을 기본 키와 외래 키를 이용하여 설계하는 방법에 대해 알아보았다. 이러한 관계에서는 하나의 **departments** row는 여러 **courses** row에게 참조될 수 있고, 하나의 **courses** row는 하나의 **departments** row만 참조할 수 있으므로 one-to-many 관계라고 한다.

앞의 예시를 더 확장하면 하나의 테이블에서 여러 column이 각각 하나의 테이블을 참조할 수 있고, 그 테이블들이 모두 같은 테이블일 수도 있으며, 심지어는 column이 속한 테이블을 다시 참조하여 tree 구조의 테이블을 생성할 수도 있다. 예를 들어 **professors** 테이블을 생성하고 **courses** 테이블에 **professors** 테이블을 참조하는 **professor** column을 생성할 수 있다.

또한 이러한 구조는 nested 될 수 있어 단과대의 정보를 담은 **colleges** 테이블을 생성하고, **departments** 테이블에 **colleges** 테이블을 참조하는 column을 생성할 수도 있다. **Table 4.9**는 이러한 **colleges** 테이블의 구조를 나타낸 것이다.

**Table 4.9** colleges Table

id	kor_name	eng_name	code
INT	VC(16)	VC(50)	INT
1	문과대학	College of Liberal Arts	13
2	이과대학	College of Science	16
3	정보대학	College of Informatics	32

**colleges** 테이블의 데이터를 참조할 수 있도록 **departments** 테이블을 **Table 4.10**과 같이 수정할 수 있다.

**Table 4.10** departments Table with Foreign Key

id	kor_name	eng_name	college
INT	VC(16)	VC(50)	INT
1	컴퓨터학과	Computer Science and Engineering	3
2	화학과	Chemistry	2
3	물리학과	Physics	2
4	언어학과	Linguistics	1

One-to-many 관계를 이용하여 many-to-many 관계를 나타낼 수도 있다. 예를 들어 **students**라는 테이블이 있다고 가정하면, 한 학생은 여러 강의를 수강할 수 있고 한 강의에는 여러 학생이 참여할 수 있으므로 many-to-many 관계가 생성된다. 이러한 관계는 **courses** 테이블을 참조하는 **course** column과 **students** 테이블을 참조하는 **student** column을 갖는 **attendance** 테이블을 생성하여 나타낼 수 있다.

## 4.5 Database Designing Exercises

### Exercise 4.1: students Table Design

학교에 소속된 학생들의 데이터를 저장하는 데이터베이스를 설계하고자 한다.

**Step 1** DB에 학생의 이름, 학번, 입학 연도, 전공, 전화번호, 주소, 누적 이수학점, 평균 평점, 재학 여부 정보를 저장하려고 한다. 이들 정보를 저장하는 테이블을 적절한 column 이름과 자료형을 제시하여 설계하여라.

**Step 2** 학번은 입학 연도, 전공, 개별 번호로 구성되어 있다. 이를 이용하여 Step 1에서 설계한 테이블에서 학번 column을 세분화하여 redundancy를 최대한 없애보자.

**Step 3** Step 2에서 설계한 테이블을 실제 DB에 생성하는 SQL문을 작성하여라.

이때 누적 이수학점, 평균 평점, 재학 여부의 기본값을 각각 0, 0.0, 참으로 설정하여라. Column의 기본값은 DEFAULT 키워드를 이용하여 설정할 수 있다. 예를 들어, **Code 4.4**는 title column의 값을 빈 문자열(''), level column의 값을 1로 설정하는 예제이다.

#### Code 4.4 DEFAULT Keyword Example

```
`title` VARCHAR(20) DEFAULT '',  
`level` INT DEFAULT 1,
```

### Exercise 4.2: Messenger DB Design

메신저 서비스의 DB를 설계하고자 한다. 먼저 각 테이블은 다음과 같은 조건을 만족해야 한다.

- 모든 테이블은 기본 키 id를 반드시 가져야 한다.
- 각 테이블의 column 이름과 자료형을 정하고, 필요에 따라 옵션을 정해야 한다.

각 테이블은 다음과 같은 정보를 포함해야 한다.

- **users**: 메신저 사용자에 관한 정보
  - 사용자 아이디
  - 사용자 비밀번호
  - 사용자 닉네임
  - 프로필 사진 링크
  - 프로필 상태 메시지
  - 탈퇴 여부; 기본값은 0
  - 가입 날짜
- **channels**: 채팅 채널에 관한 정보
  - 이름
  - 채널을 생성한 사용자

- 채널의 링크
  - 최대 수용 인원
  - 탈퇴 여부; 기본값은 0
  - 채널 생성 날짜
- **chats:** 각 채팅에 관한 정보
  - 채팅 내용
  - 채팅 작성자
  - 채팅 채널
  - 채팅 생성 날짜
- **follows:** 팔로우에 관한 정보
  - 팔로우한 사람 (follower)
  - 팔로우되는 사람 (followee)
  - 팔로우 날짜
- **blocks:** 사용자 간 차단에 관한 정보
  - 차단을 한 사람
  - 차단을 당한 사람
  - 차단 날짜

5개 테이블이 서로 적절한 관계(relation)를 갖도록 외래 키를 설정하여 각 테이블을 생성하는 SQL문을 작성하여라. 외래 키가 있는 테이블을 생성할 때 참조되는 column이 있는 테이블이 없으면 테이블이 생성되지 않는다는 점을 주의해야 한다. 예를 들어 4.4절의 **courses** 테이블과 **departments** 테이블에서 **courses.department** column이 **departments.id** column을 참조하는데, **departments** 테이블을 생성하지 않고 **courses** 테이블을 생성하려고 하면 에러가 발생한다.

각 테이블의 SQL문을 모두 **Code 4.5**와 같이 **messenger.sql** 파일에 작성하여라. SQL문을 DB에서 실행시켜 잘 작동시켜 확인해보고, DESC 키워드를 이용해 의도한 바와 같이 설계되었는지 확인해본다.

#### Code 4.5 Exercise 4.2 Example

```
CREATE TABLE `users` (
  -- Your code here
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `channels` (
  -- Your code here
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- chats / blocks table code here

CREATE TABLE `follows` (
  -- Your code here
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

## Chapter 5

# Database Querying

### Contents

5.1	CRUD Functions . . . . .	70
5.2	Advanced Querying . . . . .	73
5.3	Join Operations . . . . .	76
5.4	Querying in Node.js . . . . .	82
5.5	Database Querying Exercises . . . . .	86

## 5.1 CRUD Functions

4.1절에서 DB가 반드시 갖추어야 하는 CRUD 기능에 대해 알아보았다. 이번 장에서는 SQL문을 이용하여 CRUD 기능에 해당하는 기능을 수행해본다.

### INSERT

CRUD 기능 중 Create(생성)에 해당하는 기능은 INSERT 키워드를 통해 수행된다.

#### SQL 5.1 SQL INSERT syntax

```
INSERT INTO <table-name> (<c1>, <c2>, ..., <cn>) VALUES (<v1>, <v2>, ..., <vn>)
```

**SQL 5.1**은 INSERT 키워드를 사용한 SQL문의 기본적인 형태로, `table-name` 테이블에 `c1` column의 값이 `v1`, `c2` column의 값이 `v2`, ..., `cn` column의 값이 `vn`인 row를 생성하는 SQL문이다.

#### Shell 5.1 INSERT Example

```
> INSERT INTO `departments` (`name`, `code`) VALUES ('Liberal Arts', 13);
```

**Shell 5.1**은 INSERT 키워드를 이용하여 `departments` 테이블에 `name`은 “Liberal Arts”, `code`는 13인 row를 삽입하는 예제이다. 이때 `id`는 `AUTO_INCREMENT` 옵션이 있어 값을 명시하지 않아도 그 값이 자동으로 추가된다. 이렇게 기본값이 존재하는 column에는 값을 명시하지 않으면 기본값이 생성된다.

#### Shell 5.2 INSERT Without Column Names Example

```
> INSERT INTO `departments` VALUES (4, 'Liberal Arts', 13);
```

모든 column의 값을 명시하여 row를 생성하는 경우 **Shell 5.2**와 같이 column 이름을 작성하지 않고 column의 순서대로 값을 나열하여 생성할 수 있다.

#### Shell 5.3 INSERT With Default Values Example

```
> INSERT INTO `departments` VALUES (DEFAULT, 'Liberal Arts', 13);
```

기본값이 있는 column에 column 이름과 값을 생략할 때는 **Shell 5.3**과 같이 `DEFAULT` 키워드를 이용하여 생성할 수 있다.

## SELECT

CRUD 기능 중 Read(조회)에 해당하는 기능은 SELECT 키워드를 통해 수행된다.

### SQL 5.2 SQL SELECT Syntax

```
SELECT <c1>, <c2>, ..., <cn> FROM <table-name> WHERE <condition>
```

**SQL 5.2**는 SELECT 키워드를 사용한 SQL문의 기본적인 형태로, `table-name` 테이블에서 `condition`을 만족하는 row들의 `c1`, `c2`, ..., `cn` column의 값을 조회하는 SQL문이다.

### Shell 5.4 SELECT Example

```
> SELECT `stdnt_num`, `name` FROM `students` WHERE `grade` = 'A' AND `credits` > 52;
```

**Shell 5.4**는 SELECT 키워드를 이용하여 `students` 테이블에서 `grade` 값이 “A”이고, `credits`의 값이 52보다 큰 row 중 `stdnt_num`과 `name` column의 값을 조회하는 예제이다. 이렇게 조건(`condition`) 부분은 column 이름과 값의 비교 형태로 작성될 수 있고, 그러한 조건들의 조합으로 작성될 수 있다.

### Shell 5.5 SELECT All Columns Example

```
> SELECT * FROM `students` WHERE `grade` = 'A' AND `credits` > 52;
```

주어진 조건을 만족하는 row의 모든 column 값을 조회할 때는 **Shell 5.5**와 같이 column 이름을 모두 작성하지 않고 \* 문자로 대신할 수 있다.

### Shell 5.6 SELECT Without Condition Example

```
> SELECT * FROM `students`;
```

테이블 내 모든 데이터를 조회할 때는 **Shell 5.6**과 같이 WHERE clause를 생략할 수 있다.

## UPDATE

CRUD 기능 중 Update(수정)에 해당하는 기능은 UPDATE 키워드를 통해 수행된다.

### SQL 5.3 SQL UPDATE syntax

```
UPDATE <table-name> SET <c1>=<v1>, <c2>=<v2>, ..., <cn>=<vn> WHERE <condition>
```

**SQL 5.3**은 UPDATE 키워드를 사용한 SQL문의 기본적인 형태로, `table-name` 테이블에서 `condition`을 만족하는 row들의 column 중 `c1` column의 값을 `v1`, `c2` column의 값을 `v2`, ..., `cn` column의 값을 `vn`으로 수정하는 SQL문이다.

### Shell 5.7 UPDATE Example

```
> UPDATE `students` SET `credits` = 73, `is_attending` = 0 WHERE `dptmt` = 32;
```

**Shell 5.7**은 UPDATE 키워드를 이용하여 `students` 테이블에서 `dptmt`의 값이 32인 모든 row의 `credits` 값을 73, `is_attending` 값을 0으로 수정하는 예제이다.

앞의 SELECT 키워드와 마찬가지로 WHERE clause를 생략하여 테이블 내의 모든 데이터를 일괄적으로 수정할 수 있다.

## DELETE

CRUD 기능 중 Delete(삭제)에 해당하는 기능은 DELETE 키워드를 통해 수행된다.

### SQL 5.4 SQL DELETE Syntax

```
DELETE FROM <table-name> WHERE <condition>
```

**SQL 5.4**는 DELETE 키워드를 사용한 SQL문의 기본적인 형태로, `table-name` 테이블에서 `condition`을 만족하는 모든 row를 삭제하는 SQL문이다.

### Shell 5.8 DELETE Example

```
> DELETE FROM `students` WHERE `dptmt` = 13;
```

**Shell 5.8**은 DELETE 키워드를 이용하여 `students` 테이블에서 `dptmt`의 값이 13인 row를 모두 삭제하는 SQL문이다. DELETE 키워드를 사용할 때 WHERE clause를 생략하면 테이블 내의 모든 데이터가 삭제되므로 주의하여야 한다.

## Summary

이번 장에서는 CRUD 기능을 수행하는 SQL문을 작성하는 방법에 대해 학습하였다.

### SQL 5.5 SQL CRUD Operations Syntaxes

```
INSERT INTO <table-name> (<c1>, <c2>, ..., <cn>) VALUES (<v1>, <v2>, ..., <vn>)
SELECT <c1>, <c2>, ..., <cn> FROM <table-name> WHERE <condition>
UPDATE <table-name> SET <c1>=<v1>, <c2>=<v2>, ..., <cn>=<vn> WHERE <condition>
DELETE FROM <table-name> WHERE <condition>
```



## 5.2 Advanced Querying

Table 5.1은 세 학생들의 세 과목에 대한 성적을 저장하는 `scores` 테이블을 표로 나타낸 것이다. Code A.1을 이용하면 `scores` 테이블을 생성하고 데이터를 삽입할 수 있다.

Table 5.1 scores Table

id	student	course	midterm	final
1	Barack	Discrete Mathematics	61	87
2	Joe	Discrete Mathematics	97	92
3	Barack	Machine Learning	73	61
4	Donald	Operating Systems	58	98
5	Joe	Machine Learning	63	78
6	Donald	Discrete Mathematics	91	58
7	Donald	Machine Learning	68	82
8	Joe	Operating Systems	72	66

### Arithmetic Calculations Among Columns

`scores` 테이블의 모든 과목은 중간고사 성적(`midterm`)에 45%, 기말고사 성적(`final`)에 55%의 비중을 주어 최종 성적을 계산할 때, 최종 성적을 계산하는 상황을 가정해보자. 앞의 5.1절에서 공부한 `SELECT` 키워드를 이용하여 모든 row를 조회한 다음, 각 row의 `midterm`과 `final` column의 값에 0.45와 0.55를 곱한 뒤 더하여 산출하여야 한다. 그러나 SQL은 이러한 column 간, column과 상수 간 간단한 연산 기능을 제공한다.

#### Shell 5.9 Arithmetic Calculation Example

```
> SELECT *, .45 * `midterm` + .55 * `final` FROM `scores`;
```

Shell 5.9는 `scores` 테이블의 모든 column과 최종 성적 값을 계산하여 조회하는 SQL문이다.

#### Shell 5.10 Arithmetic Calculation Example with AS Keyword

```
> SELECT *, .45 * `midterm` + .55 * `final` AS `total` FROM `scores`;
```

Shell 5.10은 Shell 5.9에서 `AS` 키워드를 이용하여 최종 성적 값을 `total`로 명명하여 조회하는 SQL문이다. 이렇게 `AS` 키워드를 이용하면 column 이름을 다른 이름으로 조회할 수 있으며, SQL문이 너무 길어지는 것을 방지할 수도 있다. 다만 `AS` 키워드는 원래 column의 이름을 바꾸지 않는다.

### Shell 5.11 Rounding Column Value

```
> SELECT *, Round(.45 * `midterm` + .55 * `final`, -1) AS `rough_total` FROM `scores`;
```

Shell 5.11은 Round 함수를 이용하여 최종 성적을 10의 자리로 반올림한 값을 `rough_total`로 명명하여 조회하는 SQL문이다. Round 함수의 두 번째 인자를  $n$ 이라고 하면, 첫 번째 인자의 값을  $10^{-n}$ 의 자리로 반올림한 값을 반환하며, 기본값은 0이다.

## Aggregate Functions

집계 함수(aggregate function)란 여러 row로 이루어진 데이터의 집합을 하나의 값으로 표현할 수 있는 함수로, MariaDB(MySQL)에서는 20가지의 집계 함수<sup>1</sup>를 지원하며, 그 중 7가지 함수만 소개한다.

- Avg: 평균값을 반환하는 함수
- Count: NULL이 아닌 값들의 개수를 반환하는 함수
- Group\_concat: 값들을 연결하여 생성된 문자열을 반환하는 함수
- Max: 최댓값을 반환하는 함수
- Min: 최솟값을 반환하는 함수
- Stddev: 표준편차를 반환하는 함수
- Sum: 총 합계를 반환하는 함수

### Shell 5.12 Example of Avg and Stddev

```
> SELECT Avg(`midterm`), Stddev(`midterm`) FROM `scores`  
WHERE `course` = 'Machine Learning';
```

Shell 5.12는 집계 함수인 Avg와 Stddev 함수를 이용하여 “Machine Learning” 강의의 중간고사 성적의 평균과 표준편차를 구하는 예제이다.

### Shell 5.13 Example of Count and Group\_concat

```
> SELECT Count(*) AS `80s_count`, Group_concat(`student`) AS `students` FROM `scores`  
WHERE `final` >= 80 AND `final` < 90;
```

Row의 개수가 필요한 경우에는 Count 함수의 인자로 \*를 넘겨주면 된다. Shell 5.13은 Count와 Group\_concat 함수를 이용하여 기말고사 성적이 80 이상, 90 미만인 학생의 수(80s\_count)와 명단(students)을 조회하는 예제이다.

---

<sup>1</sup><https://mariadb.com/kb/en/aggregate-functions>

#### Shell 5.14 Example of GROUP BY

```
> SELECT `course`, Count(*) AS `cnt`, Avg(`final`) AS `avg`, Stddev(`final`) AS `stddev`  
FROM `scores` GROUP BY `course`;
```

`scores` 테이블에서 강의별 시험 결과를 조회해야 하는 경우 `GROUP BY` clause를 이용하여 조회 데이터를 그룹화하고 그룹별로 집계 함수의 데이터를 얻을 수 있다. **Shell 5.14**는 `scores` 테이블을 `course` column의 값에 따라 그룹화하여 각 강의의 이름, 응시자 수, 기말고사 평균, 기말고사 표준편차를 조회하는 예제이다.

## Advanced Select Options

`ORDER BY` 키워드를 이용하면 column의 값에 따라 row들을 정렬할 수 있다. `ASC`는 오름차순, `DESC`는 내림차순을 의미하며 기본값은 `ASC`이다. **Shell 5.15**는 학생의 이름과 최종 성적의 평균을 평균 성적의 내림차순으로 정렬하여 조회한 예제이다.

#### Shell 5.15 ORDER BY Example

```
> SELECT `student`, Avg(.45 * `midterm` + .55 * `final`) AS `total_avg` FROM `scores`  
GROUP BY `student` ORDER BY `total_avg` DESC;
```

`LIMIT` 키워드를 이용하면 조회 결과 중 `offset` 번째 row부터 `row_count` 개의 row만 조회할 수 있으며, 이를 `LIMIT <offset>, <row_count>`의 형태로 작성한다. 이때 `offset`은 0부터 count하고, 기본값은 0이다. **Shell 5.16**은 `scores` 테이블에서 최종 성적의 내림차순으로 정렬한 row 중 2번째 row부터 4개의 row를 조회하는 예제이다.

#### Shell 5.16 LIMIT Example

```
> SELECT *, .45 * `midterm` + .55 * `final` AS `total` FROM `scores`  
ORDER BY `total` DESC LIMIT 2, 4;
```

## 5.3 Join Operations

### Join Operation

집합  $A = \{1, 2, 4\}$ ,  $B = \{1, 3, 4\}$ 에 대해 두 집합의 곱집합  $A \times B$ 는 다음과 같이 정의된다.

$$A \times B = \{(x, y) \mid x \in A, y \in B\} = \{(1, 1), (2, 1), (4, 1), (1, 3), (2, 3), (4, 3), (1, 4), (2, 4), (4, 4)\}$$

유사하게, join 연산은 두 개 이상의 테이블을 데카르트 곱(Cartesian Product), 즉 곱집합 하는 연산이다. **Table 5.2**와 같은 테이블 A와 B를 가정해보자.

**Table 5.2** Table A (left) and Table B (right)

id	data	id	data
1	A1	1	B1
2	A2	3	B3
4	A4	4	B4

두 테이블 A와 B를 join 한 테이블은 **Table 5.3**과 같다.

**Table 5.3** Table A and B Joined

A.id	A.data	B.id	B.data
1	A1	1	B1
2	A2	1	B1
4	A4	1	B1
1	A1	3	B3
2	A2	3	B3
4	A4	3	B3
1	A1	4	B4
2	A2	4	B4
4	A4	4	B4

다만, DB에서는 두 테이블을 단순히 join 하여 사용하기보다는 특정 조건을 만족하는 row들만 join 하여 사용하는 경우가 대부분이다.

## Inner Join and Outer Join

DB에서 두 테이블을 join할 때 T1 테이블에 T2 테이블을 join한다는 표현을 사용한다. 이때 편의상 T1 테이블을 왼쪽 테이블, T2 테이블을 오른쪽 테이블이라고 하자.

먼저 inner join은 양 테이블에서 join 조건을 만족하는 row들만 join하는 방식으로, **Table 5.4**는 A 테이블에 B 테이블을  $A.id = B.id$  조건 하에 inner join한 결과이다. A 테이블의 2번 row는 B 테이블에 id가 2인 row가 없어 join에 참여하지 못했고, B 테이블에서도 3번 row가 같은 이유로 join에 참여하지 못했다. A 테이블에 B 테이블을 inner join한 결과는 B 테이블에 A 테이블을 inner join한 결과와 같음을 알 수 있다.

**Table 5.4** Table A Inner Join Table B

A.id	A.data	B.id	B.data
1	A1	1	B1
4	A4	4	B4

Outer join은 특정 테이블의 모든 row를 강제로 join에 참여시키고 대응되는 row가 없어 값이 정해지지 않는 column은 NULL로 채우는 join 방식이다. Outer join에는 left, right, full의 세 종류가 있다.

**Table 5.5** Table A Left Outer Join Table B

A.id	A.data	B.id	B.data
1	A1	1	B1
2	A2	NULL	NULL
4	A4	4	B4

먼저 left outer join은 왼쪽 테이블의 모든 row를 join에 강제로 참여시키는 방식으로, **Table 5.5**는 A 테이블에 B 테이블을  $A.id = B.id$  조건 하에 left outer join한 결과이다. B 테이블에 2번 row가 없음에도 불구하고 A 테이블의 2번 row는 join에 참여하였고, B.id와 B.data의 값이 NULL임을 확인할 수 있다.

**Table 5.6** Table A Right Outer Join Table B

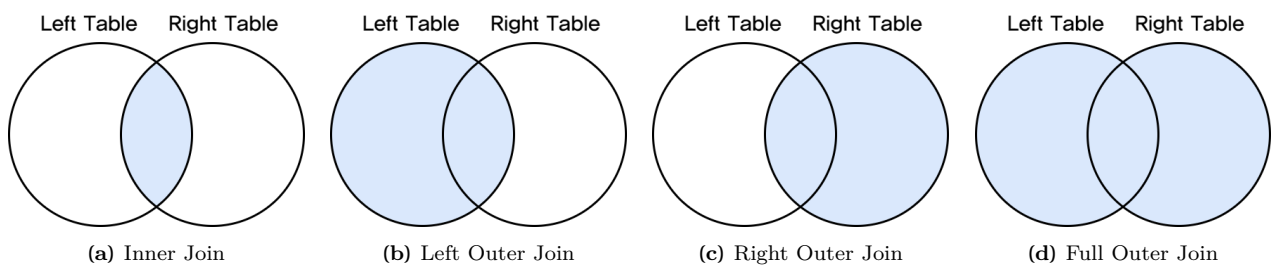
A.id	A.data	B.id	B.data
1	A1	1	B1
NULL	NULL	3	B3
4	A4	4	B4

Right outer join은 left outer join과 정반대로 오른쪽 테이블의 모든 row를 join에 강제로 참여시키는 방식으로, **Table 5.6**은 A 테이블에 B 테이블을 A.id = B.id 조건 하에 right outer join한 결과이다. Outer join의 join 방식에서 알 수 있듯, A 테이블에 B 테이블을 left outer join한 결과는 B 테이블에 A 테이블을 right outer join한 결과와 동일하다. (반대로 성립)

**Table 5.7** Table A Full Outer Join Table B

A.id	A.data	B.id	B.data
1	A1	1	B1
2	A2	NULL	NULL
NULL	NULL	3	B3
4	A4	4	B4

Full outer join은 양쪽 테이블의 모든 row를 join에 강제로 참여시키는 방식으로, **Table 5.7**은 A 테이블에 B 테이블을 A.id = B.id 조건 하에 full outer join한 결과이다. 이 방식은 inner join과 마찬가지로 교환법칙이 성립한다.



**Figure 5.1** Join Operations as Venn Diagrams

**Figure 5.1**은 네 개의 join 연산을 벤 다이어그램 형태로 나타낸 것이다.

## Join Operation in SQL

Join 연산은 관계가 있는 테이블에서 데이터를 가져올 때 매우 유용하게 사용되어 관계형 DB의 꽃과 같은 연산이다. 따라서 SQL문을 이용하여 조건에 따라 join operation을 수행하는 방법을 예시 데이터와 함께 알아본다.

### Shell 5.17 Import SQL File

```
$ mysql -u<username> -p<password> -D<db-name> < <sql-file-path>
```

**Shell 5.17**은 sql-file-path 경로에 위치한 SQL 파일을 db-name DB에 import 하는 명령어로, 이를 이용하여 **Code A.2**를 DB에 import 한다. 이때 username 계정은 db-name DB에 권한을 갖고 있어야 한다. tb1 테이블은 id와 data1 column을, tb2 테이블은 id와 data2 column을 가지며, 각 테이블에는 10개의 데이터가 삽입되어 있다.

### SQL 5.6 SELECT with Join Operation Syntax

```
SELECT <columns> FROM <left-table>
      (INNER/LEFT OUTER/RIGHT OUTER) JOIN <right-table> ON <join-condition>
      WHERE <condition>
```

**SQL 5.6**은 left-table 테이블에 right-table 테이블을 join-condition을 만족하며 join<sup>2</sup>한 테이블을 생성하고, 그 테이블에서 condition 조건을 만족하는 데이터를 SELECT 키워드를 이용해 조회하는 SQL문이다.

### Shell 5.18 Join Operations Using SQL

```
> SELECT `tb1`.`data1`, `tb2`.`data2` FROM `tb1`
      INNER JOIN `tb2` ON `tb1`.`id` = `tb2`.`id`;
> SELECT `tb1`.`data1`, `tb2`.`data2` FROM `tb1`
      LEFT OUTER JOIN `tb2` ON `tb1`.`id` = `tb2`.`id`;
> SELECT `tb1`.`data1`, `tb2`.`data2` FROM `tb1`
      RIGHT OUTER JOIN `tb2` ON `tb1`.`id` = `tb2`.`id`;
```

**Shell 5.18**은 tb1 테이블에 tb2 테이블을 tb1.id = tb2.id 조건 하에 join하는 예제이다. 각 join 방식에 따라 조회 결과가 어떻게 달라지는지 확인해본다.

이제 실제로 one-to-many 관계에 있는 테이블들의 데이터를 join하여 조회해보자. **Code A.3**을 import하여 4.4절에서 설계한 세 테이블(약간의 변화가 있다)을 생성하고 데이터를 추가할 수 있다.

먼저 courses 테이블에서 강의가 속한 학과 이름, 강의 코드, 학점, 시간을 조회하는 SQL문을 작성해보자.

### Shell 5.19 Join with Ambiguity

```
> SELECT `name`, `code`, `credit`, `period` FROM `courses`
      INNER JOIN `departments` on `courses`.`department` = `departments`.`id`;
ERROR 1052 (23000): Column 'name' in field list is ambiguous
```

**Shell 5.19**를 실행하면 name이라는 column이 모호하다(ambiguous)는 에러가 발생한다. 이 에러는 courses와 departments 테이블에 모두 name column이 있어 DBMS는 어떤 name column을 조회해야 하는지 알 수 없기 때문에 발생한다.

### Shell 5.20 Join without Ambiguity

```
> SELECT `departments`.`name`, `code`, `credit`, `period` FROM `courses`
      INNER JOIN `departments` on `courses`.`department` = `departments`.`id`;
```

따라서 join에 참여하는 테이블에 중복된 column 이름이 있다면 모호함을 피하기 위해 **Shell 5.20**과 같이 테이블 이름까지 명시해주어야 한다.

<sup>2</sup>MariaDB(MySQL)에서 full outer join 연산은 각 테이블에서 데이터를 조회한 뒤 합집합을 생성하여야 한다. 그러나 full outer join은 자주 쓰이는 연산이 아니므로 본 교재에서는 생략한다.

**Table 5.8** University Table Joined

departments			courses					
name	college	id	department	id	code	credit	period	others
Comp. Sci. & Engr.	3	1	1	1	COSE156	3	4	...
Comp. Sci. & Engr.	3	1	1	2	COSE211	3	3	...
Comp. Sci. & Engr.	3	1	1	3	COSE341	3	3	...
Comp. Sci. & Engr.	3	1	1	4	COSE362	3	3	...
Comp. Sci. & Engr.	3	1	1	5	COSE489	3	6	...
Chemistry	2	2	2	6	CHEM214	2	4	...
Physics	2	3	3	7	PHYS482	3	3	...
Chemistry	2	2	2	8	CHEM308	3	3	...

**Table 5.8**은 `courses` 테이블에 `departments` 테이블을 `courses.department = departments.id` 조건 하에 inner join 한 결과의 일부이다.

**Table 5.9** University Table Joined with Columns Selected

name	code	credit	period
Comp. Sci. & Engr.	COSE156	3	4
Comp. Sci. & Engr.	COSE211	3	3
Comp. Sci. & Engr.	COSE341	3	3
Comp. Sci. & Engr.	COSE362	3	3
Comp. Sci. & Engr.	COSE489	3	6
Chemistry	CHEM214	2	4
Physics	PHYS482	3	3
Chemistry	CHEM308	3	3

**Table 5.9**는 **Table 5.8**에서 특정 column들만 선택하여 조회한 결과로, **Shell 5.20**의 실행 결과와 동일하다. 이렇게 참조하는 column(`courses.department`)과 참조되는 column(`departments.id`)이 같은 조건 하에 join하여 관계가 있는 테이블들의 데이터를 편리하게 조회할 수 있다.



### Shell 5.21 Multiple Join Operation

```
> SELECT `courses`.`id`, `colleges`.`name` AS `college`, `dept`.`name` AS `department`,  
        `courses`.`name` AS `course`, `courses`.`code`, `credit`, `period` FROM `courses`  
        INNER JOIN `departments` AS `dept` ON `courses`.`department` = `dept`.`id`  
        INNER JOIN `colleges` ON `dept`.`college` = `colleges`.`id`;
```

하나의 테이블에 여러 테이블을 join할 수도 있고, join된 테이블에 또 다른 테이블을 다시 join할 수도 있다. **Shell 5.21**은 모든 강의의 ID(id), 단과대명(college), 학과명(department), 강의명(course), 강의 코드, 학점, 시간을 조회하는 SQL문으로, `courses` 테이블에 `departments` 테이블을 join한 뒤, `departments` 테이블에 `colleges` 테이블을 다시 join한다.

AS 키워드를 사용하면 `departments` 테이블의 이름을 `dept`로 줄인 것과 같이 테이블 이름을 바꾸어 SQL문이 지나치게 길어지는 것을 방지할 수 있다. 또한, 서로 다른 column이 같은 테이블을 참조하여 하나의 테이블이 두 번 이상 join에 참여하게 되어 테이블 이름에 모호함이 발생할 때 AS 키워드는 반드시 필요하다.

### Shell 5.22 Condition on WHERE Clause

```
> SELECT `courses`.`name`, `courses`.`code` FROM `courses`  
        INNER JOIN `departments` ON `courses`.`department` = `departments`.`id`  
        WHERE `departments`.`name` = 'Chemistry';
```

**Shell 5.22**는 WHERE 키워드를 이용하여 화학과(Dept. of Chemistry)에 속한 강의의 이름(name)과 코드(code)를 조회하는 SQL문이다.

### Shell 5.23 Condition on Join Condition

```
> SELECT `courses`.`name`, `courses`.`code` FROM `courses`  
        INNER JOIN `departments` AS `dept`  
        ON `courses`.`department` = `dept`.`id` AND `dept`.`name` = 'Chemistry';
```

**Shell 5.22**에서는 `departments.name`의 값이 “Chemistry”인 조건이 WHERE clause에 명시되었지만, 특정 조건들은 **Shell 5.23**과 같이 join 조건에 명시될 수도 있으며, 결과는 동일하면서 조회 속도가 훨씬 빨라질 수 있다. 다만, 다른 테이블의 column 값과 OR로 연결된 조건 등을 WHERE clause가 아닌 join 조건에 명시하면 전혀 다른 결과가 나올 수 있으므로 주의하여 사용하여야 한다.

## 5.4 Querying in Node.js

이번 장에서는 Node.js에서 DB에 CRUD SQL문을 보내어 데이터를 조작하고 읽는 방법에 대해 학습한다. Production 단계에서는 ORM(Object Relational Mapping)을 사용하여 raw SQL문을 사용하지 않고도 DB에서 데이터를 조작하고 읽는 작업을 수행하는 것이 일반적이며, Node.js에서는 Sequelize, TypeORM 등의 ORM이 있다. 그러나 SQL을 다룰 줄 알아야 ORM도 효과적으로 다룰 수 있으므로, 본 교재에서는 raw SQL문을 이용하여 DB에서 데이터를 조작하고 읽는 방법을 학습한다.

### DB Querying in Node.js App

mysql2 모듈은 Node.js에서 MariaDB(MySQL) DBMS에 접속하고 SQL문을 실행하는 메서드를 제공한다. Node.js 프로젝트를 생성하고 npm을 이용하여 mysql2를 설치한 뒤, database.js를 생성하여 **Code 5.1**과 같이 작성한다.

#### Code 5.1 Implementation of runQuery Function

```
const mysql = require('mysql2/promise');

const pool = mysql.createPool({
  host: <db-host>,
  port: <db-port>,
  user: <db-username>,
  password: <db-password>,
  database: <db-name>,
});

const runQuery = async sql => {
  const conn = await pool.getConnection();
  try {
    const [result] = await conn.query(sql);
    return result;
  } finally {
    conn.release();
  }
};

module.exports = { runQuery };
```

**Code 5.1**은 mysql2 모듈의 `createPool` 메서드에 연결하고자 하는 DB의 host, port, 사용자 이름과 비밀번호, DB 이름 등을 인자로 넘겨주어 connection pool을 생성하고, SQL문을 인자로 받아 실행시켜 그 결과를 반환하는 `runQuery` 함수를 구현한 코드이다.

Connection pool은 미리 DB와의 연결들을 생성하여 pool에 저장해둔 후, 필요할 때마다 연결을 가져가서 사용하고 pool에 반납(release)하도록 설계된 연결 방식이다. 이 방식은 DB에서 SQL문을 실행할 때마다 DB 연결에 필요한 여러 정보를 이용해 연결하고 SQL문을 실행한 뒤 연결을 닫는 과정에 비해 높은 성능을 보인다.

### Code 5.2 Usage of runQuery Function

```
const { runQuery } = require('./database');

const getScoreStats = async () => {
  const sql = 'SELECT course, Count(*) AS cnt, Avg(final) AS avg, ' +
    'Stddev(final) AS stddev FROM scores GROUP BY course';
  const results = await runQuery(sql);
  return results;
};

const getScoreByIdName = async (id, name) => {
  const sql = `SELECT * FROM scores WHERE id = ${id} AND student = '${name}'`;
  const results = await runQuery(sql);
  return results[0];
};

const createScore = async (name, course, midterm, final) => {
  const sql = 'INSERT INTO scores ' +
    `VALUES (DEFAULT, '${name}', '${course}', ${midterm}, ${final})`;
  const result = await runQuery(sql);
  return result;
};

(async () => {
  const stats = await getScoreStats();
  stats.forEach(stat => {
    const { course, cnt, avg, stddev } = stat;
    console.log(`${course} (${cnt} people): Average ${avg}, Std.Dev. ${stddev}`);
  });

  const scoreData = await getScoreByIdName(2, 'Joe');
  const { course, final } = scoreData;
  console.log(`Course: ${course} / Final score: ${final}`);

  console.dir(await getScoreByIdName(9, 'Barack'));
  const newScore = await createScore('Barack', 'Operating Systems', 83, 62);
  console.dir(await getScoreByIdName(9, 'Barack'));
  console.dir(await getScoreByIdName(newScore.insertId, 'Barack'));
})();
```

Code 5.2는 runQuery 함수를 이용하여 Table 5.1의 scores 테이블 데이터를 조회한 결과이다. 조회 결과 데이터는 항상 row들의 배열 형태로 반환되며, 각 row는 column 이름이 key, column 값이 value인 객체의 형태로 반환된다.<sup>3</sup>

getScoreStats 메서드는 75쪽의 Shell 5.14를 실행하는 메서드로, 조회 결과는 course, cnt, avg, stddev를 속성으로 갖는 객체들의 배열 형태이다.

getScoreByIdName 메서드는 id와 name을 인자로 받아 id column의 값이 id, student column의 값이 name인 row를 반환한다. 조회 결과는 언제나 row들의 배열 형태이므로 results[0]의 값은 row 객체거나 undefined이다.

<sup>3</sup>엄밀히는 TextRow, ResultSetHeader 등 클래스의 인스턴스가 반환된다.

createScore 메서드는 name, course, midterm, final 을 인자로 받아 student, course, midterm, final column 의 값이 각각 name, course, midterm, final 인 row를 새로 생성하는 메서드이다. 이때 runQuery 함수는 새로 생성된 row의 id 값(insertId)이나 영향받은 row의 수(affectedRows) 등 SQL문 수행 결과를 반환한다.

이렇게 mysql2 모듈을 이용하여 DB의 데이터를 조작하거나 읽을 수 있다. runQuery 함수에서 result의 값을 console.dir 메서드를 이용하여 출력하면서 결과 객체의 형태를 익혀보자.

## SQL Injection

Code 5.1의 runQuery 함수는 DB의 데이터를 조작하는 기능을 잘 수행하지만, 보안상의 문제점이 존재한다.

### Code 5.3 SQL Injection Example - Force True Condition

```
const scoreData = await getScoreByIdName(3, "abc' OR '1'='1");

// (1) SELECT * FROM scores WHERE id = 3 AND student = 'abc' OR '1'='1'
// (2) SELECT * FROM scores WHERE id = 3 AND student = 'abc' OR '1'='1'
// (3) SELECT * FROM scores
```

scores 테이블에서 성적을 조회하기 위해서는 id와 student 값을 알아야 하고, id 값은 본인 이외의 다른 사람은 모르는 값이라고 가정하자. 즉, id 값이 비밀번호인 셈이다. 그렇다면 자신의 성적을 조회하고자 하는 학생은 Code 5.2에서 getScoreByIdName 메서드에 적절한 id와 name 값을 제시해야만 본인의 성적을 확인할 수 있다.

그런데 만약 어떤 사용자가 Code 5.3과 같이 name 값에 “abc' OR '1'='1” 이라는 값을 넘긴다고 가정해보자. 이때 SQL문에 각 값을 대입해보면 (1)과 같은 SQL문이 생성되고, 이는 (2)와 같이 '1'='1', 즉 항상 참인 조건식을 포함하게 된다. 참값과의 OR 연산은 항상 참이므로 이는 (3)과 같은 SQL문이 된다. 즉 사용자가 id와 name을 전혀 알지 못하는 row의 성적을 모두 조회할 수 있는 것이다.

이렇게 변수를 이용하여 SQL문을 생성하는 과정에 의도적으로 조작된 변수를 넘겨 DB를 공격하는 방식을 SQL Injection이라고 한다. SQL Injection은 예제와 같이 공격 난이도가 높지 않으면서도 공격당했을 때 치명적인 문제를 초래할 수 있는 보안 취약점이다.

### Code 5.4 SQL Injection Example - Drop Table

```
const newScore = await createScore('n', "c'); DROP TABLE scores;", 0, 0);

// (1) INSERT INTO scores VALUES (DEFAULT, 'n', 'c'); DROP TABLE scores;', 0, 0);
// (2) INSERT INTO scores VALUES (DEFAULT, 'n', 'c'); DROP TABLE scores;', 0, 0);
```

database.js의 createPool 메서드의 인자 객체에 multipleStatements: true를 추가<sup>4</sup>하고 Code 5.4를 실행하면 scores 테이블이 DROP TABLE 키워드로 인해 삭제된다. 이렇게 SQL Injection이 발생하면 모든 데이터가 삭제되는 불상사가 발생할 수 있다.

<sup>4</sup>실제 서비스에서는 반드시 false로 설정해야 한다. (명시하지 않으면 기본값이 false)

SQL Injection은 매우 기본적이고 대중적이면서도 치명적인 문제를 초래할 수 있기 때문에 DB에 접속할 수 있는 기능을 제공하는 대부분의 모듈에는 이러한 공격을 방지하는 기능이 탑재되어 있다. 가장 대표적인 방법은 prepared statement를 이용하는 방법으로, SQL문에서 사용자로부터 값을 받는 변수 부분을 미리 약속된 문자로 표시하여 prepared statement를 생성한다. 모듈이 제공하는 메서드에 이 statement와 사용자로부터 받은 인자를 전달하면 모듈은 입력값들을 검증하며 의도치 않은 작업이 수행되지 않도록 SQL문을 생성한다.

Prepared statement의 변수 부분을 작성하는 방식은 애플리케이션의 언어, 모듈, DBMS마다 조금씩 다르지만 대부분은 물음표(?) 문자를 사용하여 변수 부분을 표시한다. mysql2 모듈 역시 prepared statement에 ? 문자를 사용하여 SQL Injection을 방지할 수 있도록 한다.

#### Code 5.5 runQuery Function with Prepared Statement

```
const runQuery = async (pstmt, data) => {
  const conn = await pool.getConnection();
  try {
    const sql = conn.format(pstmt, data);
    const [result] = await conn.query(sql);
    return result;
  } finally {
    conn.release();
  }
};
```

**Code 5.5**는 runQuery 함수를 수정한 코드로, 인자로 prepared statement(pstmt)와 ? 문자 자리에 들어갈 값들이 순서대로 할당된 배열(data)을 전달받아 SQL문을 생성(sql)한 뒤 DB에 전달하여 데이터를 받는다.

#### Code 5.6 Using Prepared Statement

```
const getScoreByIdName = async (id, name) => {
  const sql = 'SELECT * FROM scores WHERE id = ? AND student = ?';
  const results = await runQuery(sql, [id, name]);
  return results[0];
};

const createScore = async (name, course, midterm, final) => {
  const sql = 'INSERT INTO scores VALUES (DEFAULT, ?, ?, ?, ?)';
  const result = await runQuery(sql, [name, course, midterm, final]);
  return result;
};
```

runQuery 함수의 형태가 수정되었으므로 **Code 5.2**에서 SQL문에 변수가 들어가는 함수들은 **Code 5.6**과 같이 수정되어야 한다. **Code 5.3**과 **Code 5.4**를 실행해보면 SQL Injection이 방어됨을 확인할 수 있으며, runQuery 함수에서 sql 값을 확인해보면 조작된 변수가 잘 escape된 것을 확인할 수 있다.

## 5.5 Database Querying Exercises

### Exercise 5.1: Simple CRUD Functions Exercises

Shell 5.17을 참고하여 DB에 Code A.4를 import하여 테이블을 생성하고 데이터를 삽입하여라. crud 테이블에는 100개의 데이터가 있다. 다음 작업들을 수행하는 SQL문을 각각 작성하여라.

1. c1이 11이고 c2가 2인 row의 id, c3, c5 column을 조회
2. c1이 18보다 크거나 c2가 2보다 작은 row의 모든 column을 조회
3. id와 c4는 기본값, c1은 7, c2는 4, c3는 “col101”, c5는 0인 row를 생성
4. id는 103, c1은 3, c2는 3, c3는 “col103”, c4는 기본값, c5는 1인 row를 생성
5. id가 100보다 큰 row의 모든 column을 조회
6. c1이 4보다 크고 9보다 작고, c2가 1인 row의 c3를 “col0”, c5를 0으로 수정
7. c1이 4보다 크고 9보다 작고, c2가 1인 row의 모든 column을 조회
8. c5가 0인 row를 삭제
9. c5가 0인 row의 모든 column을 조회

### Intercity Trains and Tickets System

Exercise 5.2와 Exercise 5.3은 우리나라 간선열차 노선과 예매 현황을 나타내는 데이터베이스를 다루는 문제이다.

Shell 5.17을 참고하여 DB에 Code A.5를 import하여 5개 테이블 stations, types, trains, users, tickets를 생성하여라. 이후 Code A.6을 import하여 각 테이블의 데이터를 모두 import하여라.

stations는 6개의 역, types는 5개의 열차 종류, trains는 20개의 열차 노선, users는 50명의 사용자, tickets는 80개의 예매 현황을 나타내는 테이블이다.

각 열차에는 그 열차 종류(trains.type)의 최대 좌석 수(types.max\_seats)만큼의 사용자만 예매할 수 있으며, 좌석 번호(tickets.seat\_number)는 1부터 최대 좌석 수에 해당하는 번호까지 있다. 즉, 최대 10개의 좌석이 있는 열차에는 1번, 2번, ..., 10번 좌석이 존재한다.

운임률(types.fare\_rate)의 단위는 100km당 원(KRW)이며, 노선 길이(trains.distance)의 단위는 100m이다. 즉, types.fare\_rate의 값이 30,000인 열차의 운임률은 1km당 300원이며, trains.distance의 값이 200인 노선의 길이는 20km이다. 노선의 운임은 운임률과 노선의 거리를 이용하여 도출한 값을 백의 자리로 반올림하여 결정한다.

### Exercise 5.2: Intercity Trains and Tickets SQL Querying

다음 작업들을 수행하는 SQL문을 각각 작성하여라. 3, 6: optional  
6: outer join 이용

1. ID가 11인 노선을 예매한 모든 승객의 ID(id), 이름(name), 좌석 번호(seat\_number)를 좌석 번호의 오름차순으로 조회
2. 각 사용자의 ID(id), 이름(name), 탑승 열차 수(trains\_count), 총 거리(total\_distance)를 총 거리의 내림차순으로 상위 6명만 조회

3. 각 노선의 ID(id), 열차 종류(type), 출발역(src\_stn), 도착역(dst\_stn), 여행 시간(travel\_time)을 여행 시간의 내림차순으로 상위 6개만 조회
4. 각 노선의 열차 종류(type), 출발역(src\_stn), 도착역(dst\_stn), 출발 시각(departure), 도착 시각(arrival), 운임(fare; 원 단위)을 출발 시각의 오름차순으로 모두 조회
5. 각 노선의 ID(id), 열차 종류(type), 출발역(src\_stn), 도착역(dst\_stn), 예매된 좌석 수(occupied), 최대 좌석 수(maximum)를 노선의 ID의 오름차순으로 모두 조회 (예매한 사용자가 없는 노선은 제외)
6. 각 노선의 ID(id), 열차 종류(type), 출발역(src\_stn), 도착역(dst\_stn), 예매된 좌석 수(occupied), 최대 좌석 수(maximum)를 노선의 ID의 오름차순으로 모두 조회 (예매한 사용자가 없는 노선도 포함)

여행 시간은 출발 시각과 도착 시각의 시간 차이로 계산하며, 두 시각의 시간 차는 `Timediff` 함수를 이용하여 구할 수 있다. **Code 5.7**은 `former` 값과 `latter` 값의 시간 차 `diff`를 구하는 SQL문이다.

#### Code 5.7 Timediff Function Example

```
> SELECT Timediff('latter', 'former') AS 'diff' FROM 'table';
```

각 문제의 SQL문 실행 결과는 **Figure 5.2**와 같아야 한다.

id	name	seat_number	id	name	trains_count	total_distance	id	type	src_stn	dst_stn	travel_time
15	Ellery	1	12	Baldric	3	945.6000	14	Mugunghwa	Busan	GwangjuSongjeong	05:42:00
18	Charli	2	11	Deanna	3	945.6000	12	Mugunghwa	Seoul	Busan	05:36:00
11	Deanna	3	22	Beauregard	3	863.6000	10	Mugunghwa	GwangjuSongjeong	Busan	05:33:00
12	Baldric	4	23	Azura	3	863.6000	1	ITX-Saemaeul	Yeosu EXPO	Seoul	04:29:00
38	Hylda	5	25	Esmond	2	859.1000	16	ITX-Saemaeul	GwangjuSongjeong	Seoul	03:40:00
31	Alys	8	32	Corbin	2	847.3000	19	KTX	Yeosu EXPO	Seoul	03:13:00

(a) Exercise 5.2.1

type	src_stn	dst_stn	departure	arrival	fare
Mugunghwa	Busan	GwangjuSongjeong	06:17:00	11:59:00	21100
KTX	Seoul	Yeosu EXPO	07:05:00	10:17:00	61400
KTX	Seoul	Busan	07:30:00	10:12:00	64800
KTX	Busan	Seoul	08:20:00	11:32:00	64800
ITX-Saemaeul	Yeosu EXPO	Seoul	08:55:00	13:24:00	41500
KTX-Eum	Seoul	Gangneung	09:01:00	10:57:00	31400
SRT	Suseo	GwangjuSongjeong	09:40:00	11:26:00	39800
Mugunghwa	Seoul	Busan	09:56:00	15:32:00	28600
Mugunghwa	GwangjuSongjeong	Busan	10:31:00	16:04:00	21100
ITX-Saemaeul	GwangjuSongjeong	Seoul	11:26:00	15:06:00	33500
KTX-Eum	Gangneung	Seoul	12:25:00	14:27:00	31400
SRT	GwangjuSongjeong	Suseo	13:00:00	14:37:00	39800
KTX	Seoul	Busan	14:30:00	17:11:00	64800
SRT	Busan	Suseo	14:30:00	16:58:00	54900
SRT	GwangjuSongjeong	Suseo	16:25:00	18:32:00	39800
KTX	Busan	Seoul	16:38:00	19:12:00	64800
KTX	Seoul	GwangjuSongjeong	17:38:00	19:20:00	47800
KTX	Yeosu EXPO	Seoul	18:05:00	21:18:00	66700
KTX-Eum	Gangneung	Seoul	18:40:00	20:51:00	31400
KTX-Eum	Seoul	Gangneung	19:01:00	21:06:00	31400

(b) Exercise 5.2.2

id	type	src_stn	dst_stn	occupied	maximum
1	ITX-Saemaeul	Yeosu EXPO	Seoul	3	6
2	KTX	Seoul	Busan	8	8
3	KTX-Eum	Gangneung	Seoul	2	7
4	KTX	Busan	Seoul	6	8
5	SRT	GwangjuSongjeong	Suseo	2	8
6	KTX-Eum	Seoul	Gangneung	3	7
7	KTX	Busan	Seoul	6	8
8	KTX-Eum	Gangneung	Seoul	5	7
9	KTX	Seoul	Busan	4	8
10	Mugunghwa	GwangjuSongjeong	Busan	4	5
11	SRT	Suseo	GwangjuSongjeong	6	8
12	Mugunghwa	Seoul	Busan	2	5
13	SRT	GwangjuSongjeong	Suseo	2	8
14	Mugunghwa	Busan	GwangjuSongjeong	4	5
15	KTX	Seoul	Yeosu EXPO	2	8
16	ITX-Saemaeul	GwangjuSongjeong	Seoul	6	6
17	SRT	Busan	Suseo	3	8
18	KTX	Seoul	GwangjuSongjeong	8	8
19	KTX	Yeosu EXPO	Seoul	4	8

(c) Exercise 5.2.3

id	type	src_stn	dst_stn	occupied	maximum
1	ITX-Saemaeul	Yeosu EXPO	Seoul	3	6
2	KTX	Seoul	Busan	8	8
3	KTX-Eum	Gangneung	Seoul	2	7
4	KTX	Busan	Seoul	6	8
5	SRT	GwangjuSongjeong	Suseo	2	8
6	KTX-Eum	Seoul	Gangneung	3	7
7	KTX	Busan	Seoul	6	8
8	KTX-Eum	Gangneung	Seoul	5	7
9	KTX	Seoul	Busan	4	8
10	Mugunghwa	GwangjuSongjeong	Busan	4	5
11	SRT	Suseo	GwangjuSongjeong	6	8
12	Mugunghwa	Seoul	Busan	2	5
13	SRT	GwangjuSongjeong	Suseo	2	8
14	Mugunghwa	Busan	GwangjuSongjeong	4	5
15	KTX	Seoul	Yeosu EXPO	2	8
16	ITX-Saemaeul	GwangjuSongjeong	Seoul	6	6
17	SRT	Busan	Suseo	3	8
18	KTX	Seoul	GwangjuSongjeong	8	8
19	KTX	Yeosu EXPO	Seoul	4	8
20	KTX-Eum	Seoul	Gangneung	0	7

(d) Exercise 5.2.4

(e) Exercise 5.2.5

(f) Exercise 5.2.6

Figure 5.2 Exercise 5.2 Results

## Exercise 5.3: Intercity Trains and Tickets Server

다음 두 라우트를 갖는 웹 서버를 구현하여라. 모든 라우트는 plain text 형태로 응답하고, 인자값에 대한 검증은 하지 않아도 되며, 에러가 발생하면 `console.error` 메시지를 이용하여 에러를 출력하고, 500 Internal Server Error를 응답하여야 한다.

- GET `/fare`: Query로 사용자의 ID인 `uid`를 받아 해당 사용자가 지불해야 하는 총 요금을 응답하는 라우트
- GET `/train/status`: Query로 열차의 ID인 `tid`를 받아 해당 열차가 매진되었는지 판단하여 응답하는 라우트 (Hint: 예매된 좌석 수와 최대 좌석 수를 조회하여 비교한다)

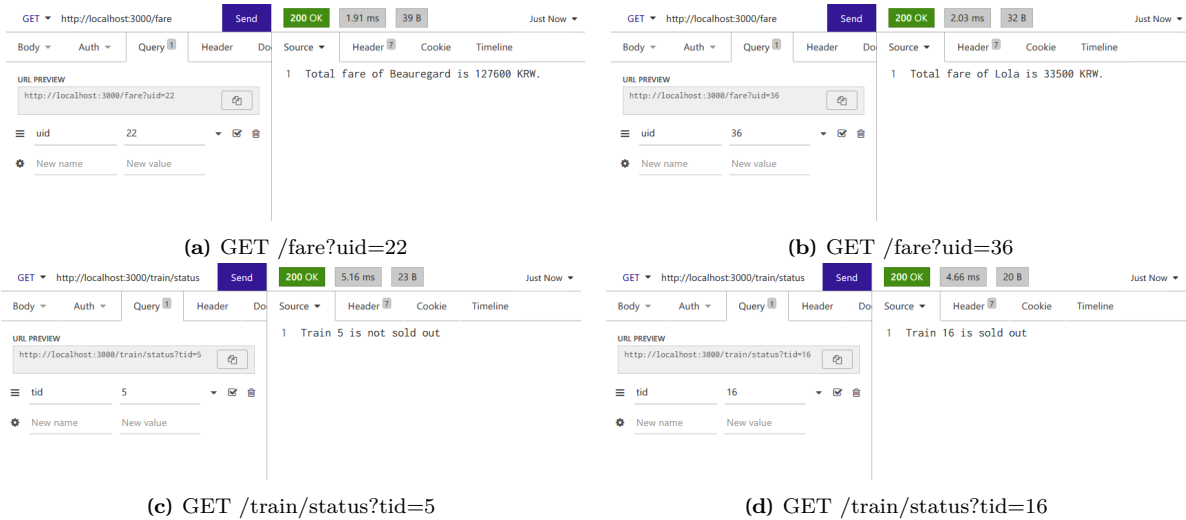


Figure 5.3 Exercise 5.3 Example Results

Figure 5.3은 라우트에 따른 서버의 응답 예시이다.



## Chapter 6

# Authentication

### Contents

6.1	Hash Functions and Encryption . . . . .	90
6.2	HTTP Cookie and Session . . . . .	95

## 6.1 Hash Functions and Encryption

### Encryption

많은 웹 사이트에서는 사용자를 식별하기 위해 사용자 계정을 이용한 인증 시스템을 사용하며, 사용자는 자신이 만든 계정에 로그인이라는 방식을 이용하여 인증한 뒤 웹 사이트에 접근한다. 이때 사용자는 아이디와 비밀번호 등 인증에 필요한 정보를 서버에 전달하여 본인임을 인증하고, 서버에서는 각 사용자의 아이디와 비밀번호 등을 DB에 저장해두고, 사용자로부터 받은 인증 데이터를 DB에 저장된 데이터와 비교하여 요청을 보낸 사용자가 계정에 접근 권한을 갖고 있는지 검증한다.

비밀번호와 같은 민감한 정보를 DB에 저장할 때 plain text로 저장하면 DB가 해킹되었을 때 사용자의 암호가 공격자의 손에 고스란히 넘어갈 수 있다. 따라서 이러한 정보는 회원가입 과정 등에서 반드시 암호화(encryption)하여 저장하여야 한다.

암호화 알고리즘은 크게 단방향 암호화와 양방향 암호화로 나뉘며, 양방향 암호화 알고리즘은 세부적으로 비대칭형 암호화와 대칭형 암호화로 나뉘기도 한다. 암호화된 데이터를 원래 데이터로 변환하는 과정을 복호화(decryption)라고 하는데, 양방향 암호화 알고리즘은 복호화가 가능한 알고리즘인 반면, 단방향 암호화 알고리즘은 복호화가 불가능한 알고리즘이다. 웹 서버에서는 이러한 암호화 알고리즘을 사용하여 사용자의 비밀번호를 암호화한 후, 암호화된 비밀번호를 DB에 저장해야 한다.

웹 서버에서 사용자 인증 과정을 어떻게 수행할지 생각해보자. 먼저 사용자가 보낸 HTTP 요청으로부터 아이디와 비밀번호를 전달받으면, DB에서 아이디 값이 입력받은 아이디와 일치하는 row를 조회하고 조회 결과가 없으면 가입되지 않은 아이디로 판별한다. 아이디가 존재하면 입력받은 비밀번호와 해당 row의 비밀번호 값이 일치하는지 검증하여 일치 여부에 따라 인증의 성공 여부를 판별한다.

비밀번호 등의 암호화 과정에서 사용되는 알고리즘이 동일한 입력에 대해 항상 동일한 결과를 반환하는, 즉 결정론적(deterministic)으로 작동하는 알고리즘이라면, 웹 서버는 인증 과정에서 암호화된 비밀번호로부터 원래 비밀번호를 도출할 필요가 없다. 대신 처음 비밀번호를 암호화한 방법과 같은 방법으로 사용자가 전달한 비밀번호를 암호화하여 DB에 저장된 비밀번호와 같은지 비교하여 판별할 수 있다. 따라서, 비밀번호를 통한 인증 과정에서는 결정론적으로 작동하는 단방향 암호화 알고리즘을 이용한다.

### Hash Function

해시 함수(hash function)는 입력받은 이진 데이터(입력값)를 또 다른 이진 데이터로 암호화(해시값)하여 반환하는 함수로, 비밀번호 암호화뿐만 아니라 데이터의 무결성 검증을 위한 체크섬 생성 등에도 널리 사용된다. 이러한 해시 함수는 다음과 같은 몇 가지 특징을 갖는다.

먼저, 해시 함수는 결정론적 알고리즘으로, 동일한 두 입력값의 해시값은 반드시 같다. 이는 곧 해시값이 다른 두 입력값은 반드시 다르다는 것을 의미한다. 그러나 두 입력값이 다르더라도 해시값이 반드시 다르다는 보장은 없으며, 이를 해시 충돌(hash collision)이라고 한다.

해시 함수의 두 번째 특징은 입력값과 무관하게 해시값의 길이는 일정하다는 것이며, 그 길이는 알고리즘마다 다르다. 입력값의 길이는 제한이 없는데 해시값의 길이는 고정되어 있으므로 이론적으로 해시 충돌은 필연적으로 발생한다. 그러나, 실제로는 입력값에 해당하는 비밀번호 등은 길이 제한이 존재하고, 해시값이 길이는 충분히 기므로 해시 충돌이 발생할 확률은 극히 낮다.<sup>1</sup>

해시 함수의 세 번째 특징은 입력값에서 발생한 아주 작은 변화가 해시값에서 아주 큰 변화를 발생시킨다는 점이다. 크기가 매우 큰 두 입력값이 단 1비트만 달라도 생성되는 해시값은 전혀 다르며, 이러한 특징을 눈사태 효과 (avalanche effect) 라고 한다. 눈사태 효과는 해시값을 이용하여 입력값을 유추할 수 없게 만들고, 해시 함수의 연산 속도가 매우 빠르다는 특징과 맞아떨어져 매우 큰 파일이 전송 과정에서 손상되었는지 확인하는 무결성 검증 (checksum)에서도 널리 사용된다.

통상적으로 사용되는 해시 함수에는 MD5, SHA-1, SHA-256, SHA-512, SHA-3 등이 있다. 이 중 MD5나 SHA-1 알고리즘은 출시된 지 오래되었고 해시 충돌 알고리즘이 발견되어 매우 취약하므로 보안 목적으로 사용하는 것이 권장되지 않는다. 최근에는 주로 SHA-256과 SHA-512 등 SHA-2 계열의 해시 함수가 널리 사용되고 있고, 앞으로는 SHA-3 해시 함수를 사용할 것이 권장되고 있다. 본 교재에서는 SHA-512 함수를 이용하여 비밀번호를 암호화하여 저장하고 검증하는 실습을 진행한다.

## Simple Encryption using PBKDF2

crypto는 입력값을 암호화하여 해시값을 반환하는 해시 함수를 제공하는 Node.js 내장 모듈이다.

### Code 6.1 Simple Example of SHA-512 Hashing

```
const util = require('util');
const crypto = require('crypto');

const pbkdf2 = util.promisify(crypto.pbkdf2);

const encrypt = async text => {
  const ALGO = 'sha512';
  const KEY_LEN = 64;
  const digest = await pbkdf2(text, '', 1, KEY_LEN, ALGO);
  console.log(`${text} | ${digest.toString('base64')}`);
};

(async () => await encrypt('samplepassword'))();
```

Code 6.1의 encrypt 함수는 인자로 받은 문자열을 crypto 모듈의 pbkdf2 메서드를 이용하여 SHA-512 알고리즘으로 암호화한 후, 암호화된 데이터(digest)를 Base64 방식<sup>2</sup>으로 인코딩하여 출력하는 함수이다. KEY\_LEN은 해시값 중 사용할 바이트 수를 나타내는 인자로, 예제에서는 64B(=512bits)를 사용한다.

코드 실행 결과 “samplepassword” 텍스트는 88자(= 문자를 제외하면  $86 = \lceil \frac{512}{6} \rceil$  자)의 문자열로 변환되었음을

<sup>1</sup> 키보드로 입력할 수 있는 문자는 95개이고, 비밀번호의 길이를 8-30자로 제한한다면  $\sum_{i=8}^{30} 95^i \approx 2.17 \times 10^{59}$  가지; 해시값이 512비트인 SHA-512의 경우의 수는  $2^{512} = 1.34 \times 10^{154}$  가지; 해시 충돌 발생 확률은  $1.62 \times 10^{-95}$ . 참고로 우주 전체의 원자 수가 약  $10^{82}$  개이다.

<sup>2</sup> 6bits의 이진 데이터를 하나의 ASCII 문자로 표현하기 위한 인코딩 방식. 0-63의 값이 A-Z, a-z, 0-9, +, /의 64개 문자에 대응되며, 맨 끝에 빈 부분을 = 문자로 채우기도 한다.

확인할 수 있고, 코드를 여러 번 실행하여도 항상 같은 결과가 도출되어 결정론적으로 작동한다는 것을 알 수 있다.

#### Code 6.2 Avalanche Effect (Derived from Code 6.1)

```
(async () => {
  await encrypt('samplepasswordsamplepasswordsamplepasswordsamplepasswordsample');
  await encrypt('samplepasswordsamplepastwordsamplepasswordsamplepasswordsample');
})();
```

Code 6.2는 Code 6.1에서 encrypt 함수의 인자를 바꾼 것이다. 62자의 문자로 이루어진 두 문자열은 단 한 자만 다르지만 코드를 실행해보면 두 문자열의 해시값이 전혀 다르다는 것, 즉 눈사태 효과를 확인할 수 있다. 이렇게 해시 함수는 매우 큰 데이터에서 매우 작은 차이를 탐지하는데 유용하며, 실제로 크기가 큰 파일이 전송 과정에서 손상되었는지 확인하기 위해 사용되는 체크섬(checksum)을 계산하기 위해 널리 사용된다.

이처럼 해시 함수를 이용하면 비밀번호 등의 복호화가 필요없는 민감한 정보를 암호화한 후 저장하여 그 정보를 안전하게 보호할 수 있다. 그러나 입력값을 해시값으로부터 수학적으로 유추할 수는 없지만, 공격자가 brute-force 방식으로 모든 입력값에 대한 해시값을 계산하여 저장해둔다면, 결정론적인 특징 때문에 해시값에 대한 입력값을 알아낼 수 있다. 이처럼 무수히 많은 입력값에 대한 해시값을 저장한 테이블을 레인보우 테이블(rainbow table)이라고 하며, 해시 함수는 연산 속도가 빠르기 때문에 입력값에 제한 조건이 있다면 레인보우 테이블을 계산하는 것은 상대적으로 쉽다. 따라서 이러한 레인보우 테이블을 사용하기 어렵게 하는 조치를 취해야 한다.

## Salting

레인보우 테이블 사용을 어렵게 하는 첫 번째 방법은 salt를 이용하는 것이다. Salt란 랜덤하게 생성된 충분히 긴 길이의 문자열이며, salt를 이용하여 암호화하고자 하는 텍스트를 변조(salting)한 뒤 해시 함수의 입력값으로 사용하는 것이다. 이렇게 얻은 해시값을 salt와 함께 저장하면 검증하고자 하는 비밀번호를 salting하여 도출된 해시값과 저장되어 있는 해시값을 비교하여 검증할 수 있다. 공격자 입장에서는 입력될 수 있는 경우의 수가 매우 많아져 레인보우 테이블을 통해 입력값을 유추하는 것이 거의 불가능하다.

#### Code 6.3 Salting Input Text (Derived from Code 6.1)

```
const randomBytes = util.promisify(crypto.randomBytes);

const encrypt = async text => {
  const ALGO = 'sha512';
  const KEY_LEN = 64;
  const salt = await randomBytes(32);
  const digest = await pbkdf2(text, salt, 1, KEY_LEN, ALGO);
  console.log(`${text} | ${salt.toString('base64')} | ${digest.toString('base64')}`);
};
```

Code 6.3은 Code 6.1을 수정한 코드로, crypto 모듈의 randomBytes 메서드를 이용하여 32B의 salt를 생성한 후, 이를 이용하여 입력값을 salting한 뒤 암호화한 코드이다.

## Key Stretching

레인보우 테이블 사용을 어렵게 하는 두 번째 방법은 암호화를 여러 번 수행하는 것이다. 해시 함수를 통해 salting 한 입력값의 해시값을 얻은 후, 이 해시값을 다시 salting한 값의 해시값을 얻는다. 이를 정해진 횟수만큼 반복하여 최종적으로 도출된 해시값을 저장하며, 반복 횟수는 10만 회 이상, 랜덤하게 결정하는 것이 권장된다. 검증 단계에서는 동일한 횟수만큼 반복하여 얻은 해시값과 저장된 해시값을 비교한다.

**Code 6.4** Key Stretching of Hashing (Derived from **Code 6.3**)

```
const encrypt = async text => {
  const ALGO = 'sha512';
  const KEY_LEN = 64;
  const salt = await randomBytes(32);
  const iter = Math.floor(Math.random() * 200000) + 200000;
  const digest = await pbkdf2(text, salt, iter, KEY_LEN, ALGO);
  console.log(`${text} | ${iter} | ${digest.toString('base64')}`);
};
```

**Code 6.4**는 **Code 6.3**을 수정한 코드로, 암호화를 20만 이상, 22만 미만의 랜덤 횟수만큼 반복적으로 수행하는 코드이다.

## Password Generation and Verification

이제까지 비밀번호 등의 민감한 정보를 적절히 암호화하는 방법에 대해 학습하였다. 이러한 해시값은 암호화 과정에서 사용된 암호화 알고리즘, 해시값의 길이, salt, 반복 횟수 등의 인자들과 함께 저장되어야 한다.

**Code 6.5** Password Generation (Derived from **Code 6.4**)

```
const generatePassword = async password => {
  const ALGO = 'sha512';
  const KEY_LEN = 64;
  const salt = await randomBytes(32);
  const iter = Math.floor(Math.random() * 200000) + 200000;
  const digest = await pbkdf2(password, salt, iter, KEY_LEN, ALGO);
  return `${ALGO}:${salt.toString(
    'base64',
  )}:${iter}:${KEY_LEN}:${digest.toString('base64')}`;
};
```

**Code 6.5**의 generatePassword 함수는 암호화 인자들과 해시값을 콜론(:)으로 구분한 문자열을 생성하여 반환하는 함수이다. 비밀번호는 이러한 형태로 DB에 저장된다.

#### Code 6.6 Password Verification (Derived from Code 6.5)

```
const verifyPassword = async (password, hashedPassword) => {
  const [algo, encodedSalt, iterStr, keyLenStr, encodedDigest] =
    hashedPassword.split(':');
  const salt = Buffer.from(encodedSalt, 'base64');
  const iter = parseInt(iterStr, 10);
  const keyLen = parseInt(keyLenStr, 10);
  const storedDigest = Buffer.from(encodedDigest, 'base64');
  const digest = await pbkdf2(password, salt, iter, keyLen, algo);
  return Buffer.compare(digest, storedDigest) === 0;
};
```

Code 6.6의 `verifyPassword` 함수는 저장된 비밀번호를 이용하여 입력값을 검증하는 함수이다. 저장된 비밀번호로부터 알고리즘, salt, 반복 횟수, 해시값 길이 등 암호화 과정에 사용된 인자들과 해시값을 도출하고, 암호화 과정의 인자들을 이용하여 입력값을 똑같이 암호화한다. 그 결과 도출된 해시값과 저장된 비밀번호의 해시값이 동일하면 옳은 비밀번호로, 다르면 틀린 비밀번호로 판별한다.

#### Code 6.7 Example of Password Generation and Verification (Derived from Code 6.6)

```
(async () => {
  const hashedPassword = await generatePassword('password');
  const result1 = await verifyPassword('password', hashedPassword);
  const result2 = await verifyPassword('passsword', hashedPassword);
  console.log(`hashed: ${hashedPassword}`);
  console.log(`password: ${result1} / passsword: ${result2}`);
})();
```

Code 6.7은 `generatePassword` 함수를 이용하여 “password” 문자열을 암호화한 뒤, `verifyPassword` 함수와 암호화된 문자열을 이용하여 두 입력값 “password”와 “passsword”가 옳은 비밀번호인지 판별하는 예제이다. “password”는 옳은 문자열로, “passsword”는 틀린 문자열로 판별되는 것을 확인할 수 있다.

## 6.2 HTTP Cookie and Session

6.1절에서 알아본 비밀번호와 같은 민감한 정보를 적절히 암호화하여 저장하고 검증하는 방법을 이용하여 사용자가 보낸 로그인 요청을 검증한 뒤 그 결과를 사용자에게 응답하는 로그인 기능을 구현할 수 있다. 서버는 특정 사용자에게만 제공되는 정보를 알맞게 제공하기 위해서는 사용자가 보내는 요청이 실제로 로그인에 성공한 사용자가 보낸 요청인지 구분할 수 있어야 한다. 이번 장에서는 서버에서 클라이언트를 어떻게 식별하는지, 클라이언트가 식별을 위한 정보를 어떻게 관리하는지 알아본다.

### HTTP Cookie

HTTP 프로토콜은 클라이언트가 서버에 요청을 보내면 서버가 그 요청에 대한 응답을 보내도록 설계된 프로토콜이다. 이때 하나의 클라이언트에서 두 개의 요청을 특정 서버에 보내면, 그 서버는 두 요청을 각각 독립적인 요청으로 인지한다. 즉, 서버는 요청을 보낸 클라이언트를 식별하지 못하며(**Figure 6.1a**), HTTP의 이러한 특징을 무상태(stateless)라고 한다.

HTTP에서는 클라이언트를 식별하기 위해 HTTP 쿠키(cookie)를 사용한다. 클라이언트는 서버에 보내는 HTTP 요청 헤더의 Cookie 속성의 값을 설정함으로써 쿠키를 보낼 수 있으며, 이러한 쿠키 값은 서버로부터 받는 HTTP 응답 헤더의 Set-Cookie 속성값을 이용하여 정할 수 있다.

**Figure 6.1b**는 쿠키를 이용하여 서버가 클라이언트를 식별하는 예시이다. 클라이언트에서 아이디와 비밀번호 등 로그인에 필요한 정보와 함께 로그인 요청을 보내면 서버는 이를 검증한 뒤 응답 헤더의 Set-Cookie 속성의 값을 설정하여 검증 결과를 응답으로 보낸다. Chrome과 Firefox와 같은 웹 브라우저는 HTTP 응답을 분석하여 Set-Cookie 속성의 값을 저장해두었다가, 같은 서버에 보내는 요청 헤더의 Cookie 속성을 저장해둔 값으로 설정하여 요청을 보낸다. 서버에서는 이 요청을 받으면 헤더의 Cookie 값을 이용하여 클라이언트를 식별할 수 있다.

이렇게 쿠키를 이용하면 서버가 사용자를 쉽게 식별할 수 있다는 장점이 있으며, 그 외에도 팝업 방지와 같은 임시적인 설정 등을 저장할 수 있다. 그러나 쿠키를 이용한 사용자 식별은 몇 가지 단점을 갖는다. 먼저 쿠키에는 여러 데이터가 저장될 수 있고, 웹 사이트에서 제공하는 서비스가 많아질수록 쿠키가 저장하는 데이터가 커질 수 있는데, 매 HTTP 요청에 쿠키값을 명시하여 보내야하기 때문에 네트워크 부하를 초래한다.

두 번째, 쿠키는 보안에 매우 취약하다. 쿠키에는 민감한 개인정보 등이 포함되어있기 마련인데, 브라우저에서는 HTTP 요청을 보낼 때 사용자의 쿠키를 사용하기 위해 파일 형태로 저장해둔다. 이러한 파일이 탈취되거나 후술할 XSS 공격 등으로 인해 쿠키가 탈취된다면 쿠키의 내용이 유출될 수 있다. 또한 공격자가 쿠키를 조작하여 자신의 정보를 조작하거나 다른 사람인 것처럼 위장한 뒤 서버에 요청을 보내면 서버는 그 정보를 식별할 수 없으므로 공격자에게 잘못된 권한을 부여할 수도 있다.

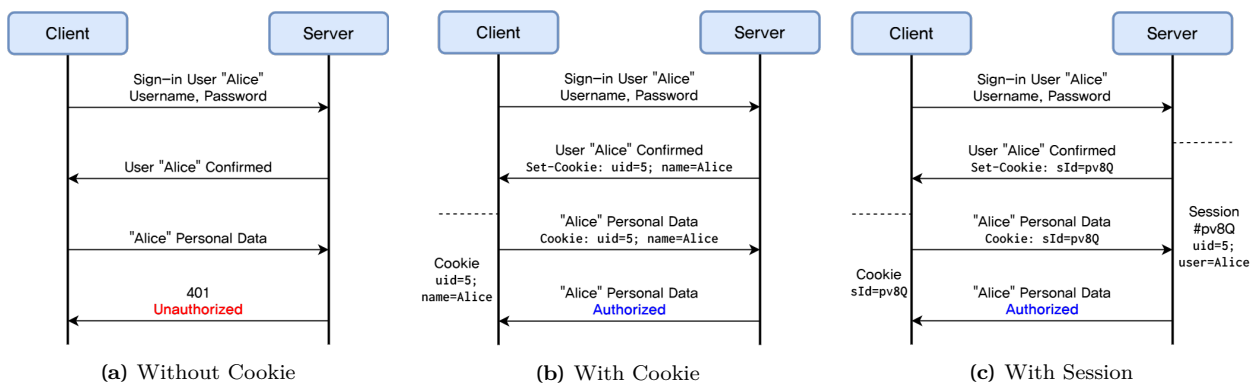
### Session

세션(session)은 사용자에게 대한 정보는 서버에 저장해두고, 해당 데이터에 부여한 고유 키를 쿠키로 주고받아 사용자를 식별하는 인증 방식으로, 쿠키가 갖는 단점을 일부 보완할 수 있다.

세션을 통한 인증은 **Figure 6.1c**와 같이 동작한다. 먼저 로그인 요청이 성공하면 서버는 사용자에게 대한 모든 정보를 서버 내에 메모리나 파일, DB 등의 형태로 저장하고, 각 데이터에 대한 고유 키, 즉 세션 ID를 생성한다. 그리고 HTTP 응답 헤더의 Set-Cookie 값에 세션 ID를 명시하여 클라이언트에 전달한다. 이후 클라이언트가 HTTP 요청 헤더의 쿠키로 세션 ID를 명시하여 요청을 보내면 서버는 그 세션 ID에 해당하는 데이터를 찾아 사용자를 식별하게 된다.

이렇듯 세션은 쿠키가 갖는 단점인 네트워크 부하 문제와, 탈취되었을 때 민감한 정보가 바로 유출되는 문제를 해결하지만, 단점 역시 존재한다. 먼저 세션은 서버 측의 부하를 초래한다. 로그인한 사용자 데이터를 서버 내에 저장하기 때문에 저장 용량 등의 자원이 충분히 확보되어 있어야 하며, 세션 ID를 이용하여 세션 데이터를 찾는 과정이 추가되어 이로 인한 속도 저하가 미세하게나마 발생한다. 따라서 세션을 사용하기 위한 자원이 서버에 충분히 마련되어 있지 않은 경우 서버 프로그램이 kill 되어버리는, 흔히 말하는 “서버 다운”이 발생할 수 있다. 또한, 세션 역시 보안 문제에서 완전히 자유롭지 못하다. 세션 ID는 어쨌든 HTTP 쿠키를 통해 전달되는 데이터이므로 세션 ID 자체가 탈취당할 수 있으며, 공격자가 탈취한 세션 ID를 이용하여 HTTP 요청을 보내면 서버는 이를 탈취당한 사용자로 인식하게 된다.

그러나 이러한 단점에도 불구하고 세션은 쿠키에 비해 더 많은 장점을 가지고 있고, 세션의 완벽한 대체재가 없기 때문에 세션을 통한 인증이 주로 사용되며, 웹 애플리케이션의 특징, 성격, 쿠키를 통해 공유하고자 하는 정보 등에 따라 쿠키와 세션을 적절히 사용하는 것이 좋다. 최근에는 세션이 갖는 부수적인 문제를 해결하기 위해 JWT(JSON Web Token)나 OAuth2 등의 토큰 기반 인증 시스템을 도입하기도 한다.



**Figure 6.1** HTTP Authentication Communication

## Session Management in Express.js

Express.js에서 사용되는 세션 관리 모듈 express-session을 이용하여 세션을 이용한 서버 기반 인증 시스템을 구현한다. express와 express-session 모듈을 설치한 뒤 **Code 6.8**과 같이 코드를 작성한다.

### Code 6.8 Session Configuration in Express.js

```
const express = require('express');
const session = require('express-session');
```



```
const app = express();
const port = 3000;

app.use(session({
  secret: '!@#%$^&*()',
  resave: false,
  saveUninitialized: true,
}));

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

**Code 6.8**에서는 express-session 모듈을 이용하여 세션 관리 미들웨어를 생성하고, 이를 app 객체에 bind하였다. 이때 secret 값은 세션 ID를 암호화하기 위한 key로, 실제 서비스에서는 길고 랜덤하게 생성되어야 하며, 별도로 보관되어야 한다.

#### Code 6.9 Create Session (Derived from Code 6.8)

```
app.get('/set/:id', (req, res) => {
  const { id } = req.params;
  req.session.requester = {
    id: parseInt(id, 10),
    name: `user#${id}`,
    level: Math.floor(Math.random() * 10) + 1,
  };
  return res.send(`Completed /set/${id}`);
});
```

**Code 6.9**는 **Code 6.8**에 params.id 값에 따라 세션을 생성하고, 클라이언트에 응답을 보내는 GET /set/:id 라우트를 구현한 코드이다. 세션은 요청 객체의 session 객체에 값을 할당함으로써 생성할 수 있으며, 이때 세션 ID가 같이 생성되면서 세션 데이터와 함께 메모리에 저장된다. 이후 세션 미들웨어는 HTTP 응답 헤더의 Set-Cookie 값을 생성된 세션 ID로 설정하여 클라이언트에 보낸다.

#### Code 6.10 Access Session (Derived from Code 6.8)

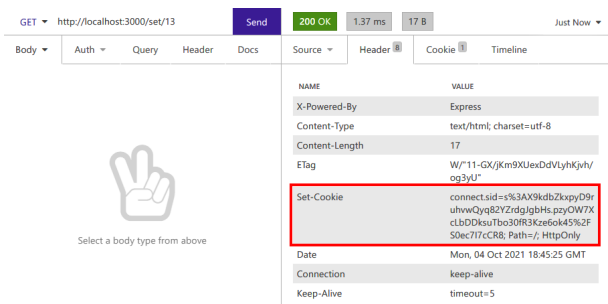
```
app.get('/get', (req, res) => {
  const { requester } = req.session;
  if (!requester) return res.sendStatus(401);
  const { id, name, level } = requester;
  return res.send(`id: ${id} / name: ${name} / level: ${level}`);
});
```

**Code 6.10**은 **Code 6.8**에 세션 ID에 따른 세션 데이터를 텍스트 형태로 응답하는 GET /get 라우트를 구현한 코드이다. 세션 미들웨어는 HTTP 요청의 쿠키에서 세션 ID를 찾고, 세션 ID가 있으면 이를 이용하여 세션 데이터를 요청 객체의 session 객체에 할당한다. 세션 데이터는 **Code 6.9**에서 세션을 생성할 때 사용한 속성명과 같은 이름의 객체, 즉 requester 객체를 통해 접근할 수 있다.

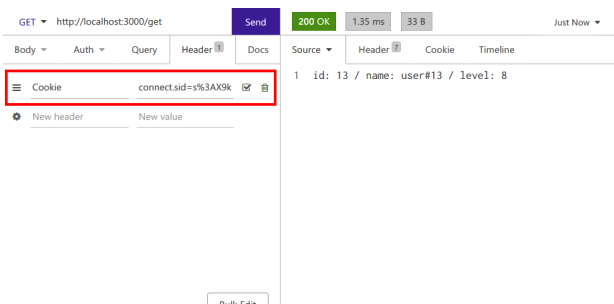
### Code 6.11 Destroy Session (Derived from Code 6.8)

```
app.get('/destroy', (req, res) => {  
  req.session.destroy(err => {  
    if (err) return res.sendStatus(500);  
    else return res.send('Destroy Completed');  
  });  
});
```

Code 6.11은 Code 6.8에 요청한 클라이언트에 해당하는 세션을 삭제하는 GET /destroy 라우트를 구현한 코드이다. 로그아웃 기능은 세션을 삭제하여 수행된다.



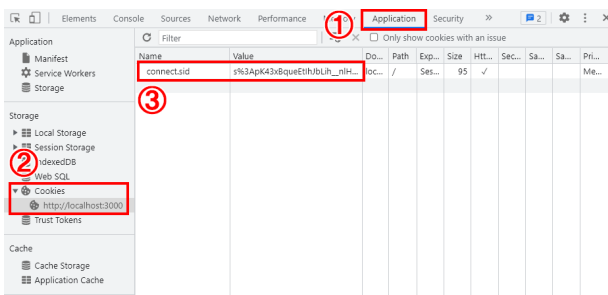
(a) Set-Cookie on Response Header



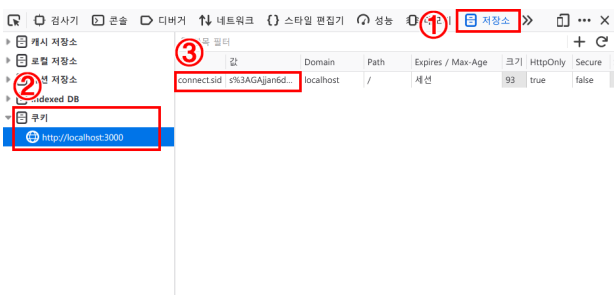
(b) Cookie on Request Header

Figure 6.2 Cookie Management on Insomnia

서버를 실행하고 Insomnia를 이용하여 쿠키를 받고 보내본다. GET /set/:id 라우트로 요청을 보내면 서버는 응답 헤더의 Set-Cookie 속성에 쿠키값을 설정하여 응답하며, GET /get 요청을 보낼 때 Header 탭에서 Cookie의 값을 응답을 통해 받은 쿠키값으로 설정하여 보내면 세션에 저장된 데이터를 응답받을 수 있다.



(a) Chrome



(b) Firefox

Figure 6.3 View Cookie on Browsers

Chrome이나 Firefox 등의 브라우저에서도 웹 서버의 동작을 확인해본다. 각 브라우저의 개발자 도구에서는 쿠키 값을 확인할 수 있는데, Figure 6.3과 같이 개발자 도구의 Application 탭이나 Storage 탭을 이용하여 브라우저 (클라이언트)가 저장하는 쿠키의 값을 확인해본다.

## Cross-Site Scripting (XSS)

앞에서 언급한 바와 같이 쿠키를 이용한 인증 방식은 구조적으로 보안 문제를 가지고 있으며, 세션 방식 역시 이 문제에서 완전히 자유롭지 못하다. 예를 들어 공격자가 A라는 사용자가 서버와의 통신을 위해 저장한 쿠키에 포함된 세션 ID를 탈취하고 이를 이용하여 HTTP 요청을 보내면 서버는 공격자를 사용자 A로 인식한다.

쿠키를 탈취하는 여러 방법 중 가장 기초적인 방법은 Cross-Site Scripting(XSS)이다. 공격자는 다른 사용자의 쿠키값을 공격자의 서버로 전송하는 악성 스크립트를 HTML 문서에 삽입하여 해당 문서를 열람한 사용자의 쿠키를 탈취한다. 이러한 공격은 대체로 게시물 등록 기능, 특히 HTML 작성 기능을 통해 주로 발생하며 악성 스크립트를 포함한 게시글을 열람한 사용자는 공격자에게 쿠키를 탈취당하게 된다.

### Code 6.12 Simple Example of XSS

```
<p>Welcome!</p>

```

**Code 6.12**는 XSS 공격의 간단한 예시로, 공격자는 이러한 내용의 글을 등록하여 서버의 DB에 저장한다. 이후 다른 사용자가 이 게시물을 열람하기 위해 요청을 보내면 이 코드가 포함된 HTML 문서를 응답으로 받게되며, 렌더링 과정에서 `img` 태그의 `src`에 해당하는 링크가 존재하지 않으므로 `onerror` 속성의 JS 코드가 실행된다. `document.cookie` 객체에는 그 HTML 문서를 요청하는데 사용된 쿠키가 저장되어 있으므로 `myserver.io`라는 서버에 쿠키값을 그대로 전송하게 된다.

이렇게 XSS는 공격자 입장에서는 공격하기 쉽고, 개발자 입장에서는 방어하기 힘들며, 일반 사용자 입장에서는 당하기 쉽고 당했는지도 알기 힘든 공격 방식이다. 이로 인해 사용자가 다른 웹 사이트에 접속하려고 할 때 신뢰할 수 있는 사이트에만 접속하라는 메시지를 띄우는 웹 사이트도 있다.

웹 서버 개발자는 XSS 공격을 반드시 방어하여야 한다. 사용자로부터 들어오는 입력값 중 게시글과 같이 DB에 저장된 후 다른 사용자에게 보여지는 입력값을 검증하여, XSS가 발생할 수 있는 부분을 제거해야 하며, 이러한 작업을 `sanitize`(소독)라고 한다. 다행히도 이러한 `sanitize` 작업을 수행해주는 모듈이 많이 개발되어 있어 개발자는 이러한 모듈을 적극 활용하여 XSS 공격을 예방할 수 있다. 또한, 궁극적인 방어 방법은 아니지만 쿠키를 `HttpOnly`로 설정하여 `document.cookie` 객체를 통해 쿠키값에 접근할 수 없게 할 수도 있다. 앞의 **Code 6.8**은 `HttpOnly` 쿠키를 자동으로 추가하며, **Figure 6.2a**와 **Figure 6.3**을 통해 확인할 수 있다.



## Chapter 7

# Web Application Implementation

### Contents

7.1	Application Summary and Structure . . . . .	102
7.2	Database Design and DAO Implementation . . . . .	106
7.3	Routing and Controller Implementation . . . . .	109
7.4	Deploying Web Application . . . . .	116

## 7.1 Application Summary and Structure

지금까지 back-end, 즉 웹 서버에서 필요한 기초 개념들과 사용되는 각종 기술 스택들을 공부하였다. 이번 장에서는 이러한 개념들과 기술 스택을 이용하여 회원가입, 로그인, 게시물 열람, 작성, 수정, 삭제 등의 기능을 수행하는 간단한 게시판 웹 애플리케이션을 구현해본다.

### Application Summary and Structure

다음 GitHub 저장소에서 스켈레톤 코드를 clone한다. 이 스켈레톤 코드에 주요 기능을 구현하여 애플리케이션을 완성한다.

- <https://github.com/KWEBofficial/node-simple-board-skeleton>

**Code 7.1** package.json

```
...
  "scripts": {
    "start": "nodemon ./src/index.js"
  },
  ...
  "dependencies": {
    "dotenv": "^16.0.3",
    "express": "^4.18.2",
    "express-session": "^1.17.3",
    "morgan": "^1.10.0",
    "mysql2": "^2.3.3",
    "pug": "^3.0.2"
  },
  "devDependencies": {
    "nodemon": "^2.0.20"
  }
}
```

**Code 7.1**은 서버 프로젝트의 package.json 파일이다. dependencies에 명시된 모듈들은 코드 내에서 사용되는 모듈이고, devDependencies 항목에 명시된 모듈은 소스 코드 내에서 직접 사용되지는 않으나 개발 과정에서 필요한 모듈이다. npm install 명령어를 통해 명시된 모듈을 일괄적으로 설치할 수 있으며, --only=prod 옵션을 주면 dependencies 모듈들만 설치한다.

dotenv은 설정값을 가져오는 모듈, morgan은 로그를 출력하는 미들웨어를 제공하는 모듈이다. nodemon 모듈은 Node.js 애플리케이션 개발 과정에서 소스 코드가 수정되면 자동으로 애플리케이션을 재시작하는 모듈로, node 명령어로 실행했을 때에 비해 번거로움이 줄어든다.

### Shell 7.1 npm run script

```
$ npm run start
```

scripts 항목에는 명령어와 그 이름을 미리 작성하여 등록할 수 있으며, `npm run <script-name>` 형태로 해당 명령어를 실행할 수 있다. **Code 7.1**에는 nodemon으로 서버를 실행하는 명령어가 start 스크립트로 등록되어 있고, 이는 **Shell 7.1**과 같이 실행할 수 있다.

### Code 7.2 Writing .env(dotenv) file

```
MODE=<run-mode>

PORT=<web-app-port>

SESSION_SECRET=<session-secret>

DB_HOST=<db-hostname>
DB_PORT=<db-port>
DB_USER=<db-user>
DB_PASS=<db-password>
DB_NAME=<db-name>
```

`./env` 파일을 생성하고 **Code 7.2**와 같이 작성하여 서버의 설정 파일인 dotenv 파일을 작성한다. `<run-mode>`는 실행하는 모드로, 개발 과정에서는 dev로 작성한다. `<web-app-port>`는 서버를 실행할 포트 번호, `<session-secret>`은 세션 ID를 생성(**Code 6.8**)하기 위한 key이다. `<db-host>`, `<db-port>`, `<db-user>`, `<db-password>`, `<db-name>`에는 각각 사용할 DB의 hostname, 포트, 사용자 이름, 사용자 비밀번호, 이름을 작성한다. 이때 hostname과 포트의 기본값은 localhost와 3306이다.

`env` 파일은 `.gitignore` 파일에 의해 추적되지 않으며, 이렇게 실행 환경 등에 따라 달라지는 설정 등은 `env`와 같은 별도의 파일에 작성하여야 한다.

### Shell 7.2 Project Structure (/)

```
$ tree -L 1 ./
./
├── package-lock.json
├── package.json
├── public                : front-end static files (CSS, JS, etc)
├── schema                : database schema
├── src                   : server source codes
└── views                 : pug templates
```

**Shell 7.2**는 프로젝트 최상위 디렉토리를 나타낸 것이다. `public/` 디렉토리에는 프론트엔드에서 필요한 CSS와 JS 파일, `schema/` 디렉토리에는 DB 테이블을 생성하는 SQL 파일, `src/` 디렉토리에는 서버의 소스코드, `views/` 디렉토리에는 pug로 작성된 템플릿 파일이 있다.

### Shell 7.3 Project Structure (views/)

```
$ tree ./views/
./views/
├── articles                : pages associated with articles
│   ├── details.pug        : article content page
│   ├── editor.pug         : article editor page
│   └── index.pug          : article list page
├── auth                   : pages associated with authorization
│   ├── sign-in.pug        : sign in page
│   └── sign-up.pug        : sign up page
├── error.pug              : error page
├── index.pug              : homepage
└── layout.pug             : layout
```

views/ 디렉토리 내에는 pug 템플릿 파일이 있으며, views/layout.pug를 제외한 모든 템플릿 파일에는 첫 두줄에 템플릿 파일이 필요로 하는 인자가 주석으로 명시되어 있다.

### Shell 7.4 Project Structure (src/)

```
$ tree ./src/
./src/
├── app.js                  : server configuration
├── env.js                  : dotenv parser
├── index.js                : server initialization
└── lib                    : library modules
    ├── authentication.js   : password generation and verification
    ├── database.js         : database query processor
    └── error-handler.js     : error handler
```

Shell 7.4는 src/ 디렉토리의 구조이다. 먼저 src/lib/database.js에는 runQuery 함수(Code 5.5)가 구현되어 있고, src/lib/authentication.js에는 비밀번호를 생성하고 인증하는 함수(Code 6.5, Code 6.6)가 구현되어 있다.

src/lib/error-handler.js에는 HTTP 요청 처리 과정에서 발생한 모든 에러를 일괄적으로 처리하는 errorHandler 함수가 구현되어 있다. 이 함수는 400, 401, 404 상태에 해당하는 에러들을 각각 적절히 처리하고, 그 외의 에러들은 500 Internal Server Error 상태에 해당하는 응답을 보내어 처리한다. 다른 미들웨어에서 next 함수에 인자를 전달하여 호출하면 3개의 인자를 갖는 일반적인 미들웨어는 모두 지나치고 4개의 인자를 갖는 미들웨어를 실행하는데, 이때 에러 객체를 인자로 넘겨주면 errorHandler 함수가 처리하게 된다. 이러한 에러 처리 방식을 이용하면 다른 미들웨어에서 발생한 에러를 즉시 처리하지 않아도 되고, 200 OK가 아닌 다른 상태에 해당하는 응답을 보낼 때 고의로 원하는 상태에 해당하는 에러를 발생시킬 수도 있어 코드의 가독성을 향상시킨다.

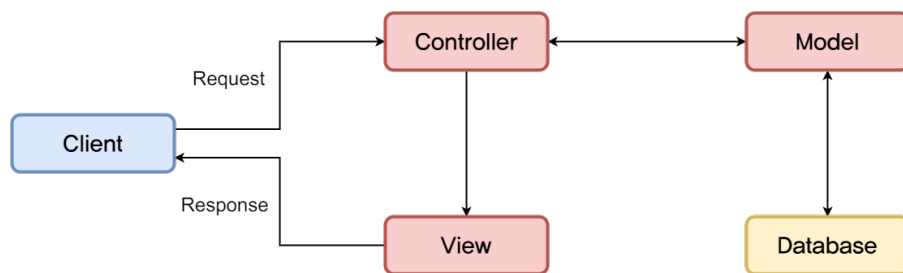
src/env.js에서는 .env 파일을 분석하여 각 값을 process.env 객체에 저장한다. 예를 들어 .env 파일의 PORT 값은 process.env.PORT에 할당된다. src/app.js에서는 서버 객체를 생성하고, view engine을 pug, pug 템플릿 파일의 디렉토리를 views/, 프론트엔드 정적 파일의 디렉토리를 public/으로 설정한 뒤 서버에서 사용할 각종 미들웨어를 등록한다. src/index.js에서는 src/env.js와 src/app.js 모듈을 import하여 설정값을 가져오고 서버 객체를 생성한 뒤 서버를 실행한다.



## Design Pattern and Objectives

7.2절에서는 DB를 설계하고 src/DAO/ 디렉토리에 DAO를 구현한다. DAO는 Data Access Object의 약자로, DB를 통해 데이터를 조회하거나 조작하는 작업을 전담하는 객체이다. 7.3절에서는 라우팅을 설계하고 각 라우트에 대한 controller 함수들을 구현한다.

이렇게 model(DAO), view(pug 템플릿), controller의 세 부분으로 나누어 소프트웨어를 설계하는 방식을 MVC 패턴(**Figure 7.1**)이라고 한다. Model에 해당하는 DAO에서는 오로지 특정 조건을 만족하는 데이터를 DB에서 가져오거나 데이터를 조작하는 역할을 하고, view에 해당하는 pug 템플릿은 오로지 사용자에게 보여지는 영역, 즉 front-end 역할만 한다. 그리고 controller는 model을 적절히 활용하여 HTTP 요청을 처리하고, 그 결과를 view를 통해 클라이언트에 전달한다.



**Figure 7.1** MVC Pattern

이러한 MVC 패턴과 같은 디자인 패턴은 프로젝트에서 각 코드의 역할을 명확히 구분하고 분류하여 생산성의 향상을 가져오고, 추상화 수준을 높여 코드의 재사용성을 높이고 유지 및 보수를 용이하게 한다. 따라서 프로젝트의 성격에 따라 적절한 디자인 패턴을 정하고 그 패턴을 준수하는 것은 중요하다.

## 7.2 Database Design and DAO Implementation

### Database Tables Design

게시판 웹 애플리케이션의 DB를 설계해보자. 사용자와 관련된 데이터에는 어떠한 것이 있겠는가? 사용자 인증을 위한 아이디(username)와 암호화된 비밀번호(password)가 필요하며, 닉네임과 같이 다른 사용자들이 보는 이름인 display\_name이 필요하다. 이때 username과 display\_name은 UNIQUE 키워드를 넣어 중복을 방지한다. 또한 가입 날짜(date\_joined), 탈퇴 여부(is\_active), 관리자 여부(is\_staff) 등이 필요하며, 각각 row 생성 시의 시각, 1, 0이 기본값이 되어야 한다. **Code 7.3**은 이러한 users 테이블을 생성하는 SQL문이다.

**Code 7.3** users Table Schema

```
CREATE TABLE `users` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `username` VARCHAR(16) NOT NULL UNIQUE,  
  `display_name` VARCHAR(32) NOT NULL UNIQUE,  
  `password` VARCHAR(151) NOT NULL,  
  `date_joined` DATETIME NOT NULL DEFAULT current_timestamp(),  
  `is_active` TINYINT(1) NOT NULL DEFAULT 1,  
  `is_staff` TINYINT(1) NOT NULL DEFAULT 0,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

이번에는 게시물 데이터를 저장하는 articles 테이블을 설계해보자. 먼저 게시물의 제목(title), 내용(content), 작성자(author) 등이 필요하며, author column은 users.id를 참조하도록 한다. 또한 작성 시각(created\_at), 마지막 수정 시각(last\_updated), 삭제 여부(is\_deleted) 등의 정보가 필요하다. created\_at과 last\_updated의 값은 row 생성 시각으로 설정하고, ON UPDATE 키워드를 사용하여 column의 값이 row가 수정될 때의 시각으로 바뀌도록 한다. is\_deleted의 기본값은 0으로 설정하고, 사용자가 게시물을 삭제하면 1로 바꾼다. **Code 7.4**는 이러한 articles 테이블을 생성하는 SQL문이다.

**Code 7.4** articles Table Schema

```
CREATE TABLE `articles` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `title` VARCHAR(50) NOT NULL,  
  `content` TEXT NOT NULL,  
  `author` INT NOT NULL,  
  `created_at` DATETIME NOT NULL DEFAULT current_timestamp(),  
  `last_updated` DATETIME NOT NULL DEFAULT current_timestamp()  
    ON UPDATE current_timestamp(),  
  `is_active` TINYINT(1) NOT NULL DEFAULT 1,  
  `is_deleted` TINYINT(1) NOT NULL DEFAULT 0,  
  PRIMARY KEY (`id`),  
  FOREIGN KEY (`author`) REFERENCES `users`(`id`) ON DELETE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Code 7.3과 Code 7.4는 schema/0000.sql에 저장되어 있다.

## User DAO Implementation

src/DAO/user.js를 생성하고, src/lib/database.js의 runQuery 함수를 import 한다. 그리고 다음 두 함수를 구현하고, 구현한 두 함수를 export 하여라. 구현 결과는 Code A.7을 참고하여라.

### getByUsername

- 인자: 사용자의 username
- 반환값: id, password, displayName, isActive, isStaff 속성을 갖는 사용자 객체
- username의 값이 인자로 받은 username인 사용자 객체 반환

### create

- 인자: 사용자의 username, password (암호화된 형태), displayName
- username, password, display\_name의 값이 각각 인자로 받은 username, password, displayName인 사용자 생성

## Article DAO Implementation

src/DAO/article.js를 생성하고, src/lib/database.js의 runQuery 함수를 import 한다. replaceDate 함수는 인자로 받은 게시물(article) 객체가 falsy하면 그대로 반환하고, truthy하면 createdAt 속성과 lastUpdated 속성을 Date 객체에서 문자열 형태로 변환하여 반환하는 함수이다.

다음의 7개 함수를 구현하고, 이를 모두 export 하여라. 구현 결과는 Code A.8을 참고하여라.

### getList

- 인자: start, count
- 반환값: id, title, createdAt, lastUpdated, author의 displayName 속성을 갖는 게시물 객체의 배열
- is\_active가 1, is\_deleted가 0인 게시물 객체를 id의 역순으로 나열한 뒤 start번째 row부터 count개의 row 반환. 각 게시물 객체의 날짜는 문자열화되어 반환되어야 한다.

### getTotalCount

- 반환값: is\_active가 1, is\_deleted가 0인 게시물의 개수

### getById

- 인자: 게시물의 id
- 반환값: id, title, content, createdAt, lastUpdated, author, author의 displayName 속성을 갖는 게시물 객체
- id의 값이 인자로 받은 id, is\_active가 1, is\_deleted가 0인 게시물 객체 반환

### getByIdAndAuthor

- 인자: 게시물의 id, author (사용자 객체)
- 반환값: title, content, author, createdAt, lastUpdated 속성을 갖는 게시물 객체
- id가 인자로 받은 id, author이 인자로 받은 author.id, is\_active가 1, is\_deleted가 0인 게시물 객체 반환

#### create

- 인자: 게시물의 title, content, author (사용자 객체)
- 반환값: 생성된 게시물 데이터의 ID 값
- title, content, author의 값이 각각 인자로 받은 title, content, author.id인 게시물 생성

#### update

- 인자: 게시물의 id, title, content
- id가 인자로 받은 id인 게시물의 title, content 값을 인자로 받은 title, content로 수정

#### remove

- 인자: 게시물의 id
- id가 인자로 받은 id인 게시물의 is\_deleted 값을 1로 수정

## Generating DAO Module

이제 src/DAO/index.js를 생성하고, 앞에서 설계한 두 DAO를 각각 UserDAO, ArticleDAO로 import 한 뒤, 두 객체를 그대로 export 한다. (**Code 7.5**) Node.js에서 import 하는 모듈의 경로가 디렉토리이면 그 아래의 index.js가 모듈 파일로 간주되어, 이제 다른 모듈에서는 src/DAO 모듈을 import 하여 두 DAO에 모두 접근할 수 있다.

#### Code 7.5 DAO Module (src/DAO/index.js)

```
const ArticleDAO = require('./article');
const UserDAO = require('./user');

module.exports = { ArticleDAO, UserDAO };
```

## 7.3 Routing and Controller Implementation

이번 장에서는 웹 서버의 각 기능을 설계하고 라우팅한 뒤, 구현해본다.

### Routing

먼저 웹 사이트의 홈 페이지 라우트는 다음과 같이 정할 수 있다.

- GET /: 홈 페이지

사용자 인증 기능을 담당하는 라우팅을 설계하자. 인증과 관련된 기능은 크게 회원 가입(sign up), 로그인(sign in), 로그아웃(sign out) 등이 있다. 3.4절의 Exercise 3.4에서 이미 GET 메서드를 통해 로그인 페이지를 응답하고, POST 메서드를 통해 로그인 데이터를 받은 바 있다. 따라서 다음과 같이 라우트를 설계할 수 있다.

- GET /auth/sign\_in : 로그인 페이지
- POST /auth/sign\_in : 로그인 데이터를 받아서 인증
- GET /auth/sign\_up : 회원가입 페이지
- POST /auth/sign\_up : 회원가입 데이터를 받아서 인증
- GET /auth/sign\_out : 로그아웃 처리<sup>1</sup>

게시물과 관련된 기능을 담당하는 라우팅을 설계하자. 먼저 각 게시물에 대한 CRUD 기능, 즉 생성, 조회, 수정 삭제 기능을 수행할 라우트가 필요하다.<sup>2</sup> 게시물 생성 기능과 수정 기능은 사용자가 데이터를 입력할 수 있는 페이지가 제공되어야 하므로 다음과 같이 라우트를 설계할 수 있다.

- GET /article/:articleId(\d+) : id가 `articleId`인 게시물을 열람
- GET /article/compose : 새 게시물 작성 페이지
- POST /article/compose : 새 게시물 데이터 반영
- GET /article/edit/:articleId(\d+) : id가 `articleId`인 게시물 수정 페이지
- POST /article/edit/:articleId(\d+) : id가 `articleId`인 게시물 수정사항 반영
- GET /article/delete/:articleId(\d+) : id가 `articleId`인 게시물 삭제<sup>3</sup>

여기에서 `\d+`는 정수와 매칭되는 정규표현식으로, 이렇게 작성하면 `articleId`는 정수 형태의 문자열임이 보장된다.

게시판에서는 사용자의 편의를 위해 게시물의 목록도 조회할 수 있어야 한다. 이 웹 애플리케이션에서는 게시물 목록을 최신 게시물부터 페이지 단위로 조회할 수 있도록 한다. 이를 반영하면 다음과 같이 라우트를 설계할 수 있다.

- GET /articles/page/:page(\d+) : page번째 페이지의 게시물 목록 조회
- GET /articles : 1번째 페이지의 게시물 목록 조회

<sup>1</sup>로그아웃 기능은 POST 메서드로 처리하는 것이 좋으나, 본 교재에서는 일단 GET 메서드로 처리한다.

<sup>2</sup>정책에 따라 특정 기능들은 제외될 수 있다.

<sup>3</sup>게시물을 삭제하는 것은 POST 메서드를 사용하는 것이 옳으나, 본 교재에서는 일단 GET 메서드로 처리한다.

## Home and Auth Routes' Controller

각 라우트에 대한 컨트롤러를 구현해보자. 모든 컨트롤러는 **Code 7.6**과 같은 형태를 가지며, 컨트롤러에서 에러가 발생하면 `next` 함수에 에러 객체 `err`를 전달하여 호출한다.

### Code 7.6 Controller Format

```
const controllerName = async (req, res, next) => {
  try {
    // Controller function
  } catch (err) {
    return next(err);
  }
};
```

`src/controller/ctrl.js`를 생성하고, 다음 controller를 구현한 뒤 `export`하여라. 결과는 **Code A.9**를 참고하여라.

#### indexPage

- GET / → 웹 사이트의 홈페이지를 응답
  - 템플릿: `views/index.pug`
    - `user`: 현재 로그인된 사용자 객체
- Hint
  - 템플릿 파일의 렌더링: 3.2절 참고
  - 사용자 정보는 세션 객체의 `user` 객체에 저장된다. (6.2절)

`src/controller/auth/ctrl.js`를 생성하고, 다음 controller를 구현한 뒤 `export`하여라. 필요에 따라 적절한 모듈을 `import`하여 사용하여라. 결과는 **Code A.10**을 참고하여라.

#### signInForm

- GET /auth/sign\_in
  - 로그인된 사용자 → 홈으로 `redirect`
  - 로그인되지 않은 사용자 → 로그인 페이지 응답
- 템플릿: `views/auth/sign-in.pug`
  - `user`: 현재 로그인된 사용자 객체
- Hint: 로그인되지 않은 사용자의 `user` 값은 `undefined`이다.

#### signIn

- POST /auth/sign\_in → HTTP 요청의 `body`의 아이디와 비밀번호를 받아 로그인 처리
  - 아이디(`username`)나 비밀번호(`password`)가 없으면 `BAD_REQUEST` 에러 발생
  - DB에서 `username`이 일치하는 사용자가 없으면 `UNAUTHORIZED` 에러 발생
  - 요청받은 비밀번호와 사용자의 비밀번호가 불일치하면 `UNAUTHORIZED` 에러 발생
  - 사용자의 `id`, 아이디(`username`), 닉네임(`displayName`), 탈퇴 여부(`isActive`), 관리자 여부(`isStaff`) 정보를 담은 객체를 생성하여 세션의 `user` 객체로 등록
  - 홈으로 `redirect`

- Hint
  - `ERROR_NAME` 에러를 발생시키는 방법: `throw new Error('ERROR_NAME')`
  - 세션을 생성하는 방법: 6.2절 참고

#### signUpForm

- GET `/auth/sign_up` → 회원가입 페이지 응답
- 템플릿: `views/auth/sign-up.pug`
  - `user`: 현재 로그인된 사용자 객체

#### signUp

- POST `/auth/sign_up` → HTTP 요청의 `body`의 아이디와 비밀번호, 닉네임을 받아 회원가입 처리
  - 아이디(`username`), 비밀번호(`password`), 닉네임(`displayName`)이 없거나 저장 가능한 최대 길이를 초과하면 `BAD_REQUEST` 에러 발생
  - 사용자가 입력한 비밀번호를 이용하여 암호화된 비밀번호 생성
  - DB에 새로운 사용자 데이터 생성
  - 로그인 페이지로 `redirect`
- Hint: 저장 가능한 최대 길이는 해당 `column`의 자료형을 확인

#### signOut

- GET `/auth/sign_out` → 로그아웃 처리, 완료 시 홈으로 `redirect`
- Hint: 세션을 삭제하는 방법은 6.2절 참고

#### Code 7.7 Auth Routing

```
const { Router } = require('express');

const ctrl = require('./ctrl');

const router = Router();

router.get('/sign_in', ctrl.signInForm);
router.post('/sign_in', ctrl.signIn);

router.get('/sign_up', ctrl.signUpForm);
router.post('/sign_up', ctrl.signUp);

router.get('/sign_out', ctrl.signOut);

module.exports = router;
```

이제 각 `controller` 함수를 각 라우트에 연결하는 라우팅을 해보자. `src/controller/auth/index.js`를 생성하고, **Code 7.7**과 같이 작성한다. **Code 7.7**은 `express`의 `Router` 객체를 이용하여 경로가 `/auth`로 시작하는 라우트만 라우팅하며, 각 라우트의 경로에서 `/auth`는 생략한다. 그리고 이 `Router` 객체를 `export` 한다.

#### Code 7.8 Index Routing

```
const { Router } = require('express');

const ctrl = require('./ctrl');
const auth = require('./auth');

const router = Router();

router.get('/', ctrl.indexPage);

router.use('/auth', auth);

module.exports = router;
```

src/controller/index.js를 생성하고, **Code 7.8**과 같이 작성한다. **Code 7.8**에서는 express의 Router 객체에 GET / 라우트의 controller 함수를 등록하고, **Code 7.7**에서 작성한 Router 객체를 /auth로 시작하는 경로에 대해서만 실행되는 미들웨어로 bind 한다. 이러한 구조는 인증(auth)과 관련된 라우트를 따로 분리하여 작성할 수 있도록 해준다.

#### Code 7.9 Bind Controller Module on app

```
// omit

const controller = require('./controller');
const errorHandler = require('./lib/error-handler');

// omit

app.use('/', controller);

app.use(errorHandler);

module.exports = app;
```

src/app.js를 **Code 7.9**와 같이 수정하여 /로 시작하는 경로에 대해 src/controller 모듈이 실행될 수 있도록 한다. 이제 **Shell 7.1**과 같이 서버를 실행하고, 지금까지 작성한 기능들을 직접 테스트 해보자.



## Article Routes' Controller

src/controller/article/ctrl.js를 생성하고, 다음 controller를 구현한 뒤 export 하여라. 결과는 **Code A.11**을 참고 하여라.

### readArticle

- GET /article/:articleId(\d+) → **articleId**에 해당하는 게시물 상세 페이지 응답
  - DB에서 id가 **articleId**인 게시물 조회 → 없으면 NOT\_EXIST 에러 발생
  - 상세 페이지 응답
- 템플릿: views/articles/details.pug
  - **user**: 현재 로그인된 사용자 객체
  - **article**: 페이지에 띄우고자 하는 게시물 객체

### writeArticleForm

- GET /article/compose → 게시물 에디터 페이지 응답
- 템플릿: views/articles/editor.pug
  - **user**: 현재 로그인된 사용자 객체

### writeArticle

- POST /article/compose → HTTP 요청의 body의 제목과 내용을 받아 게시물 생성
  - trim 메서드를 이용하여 제목(**title**)과 내용(**content**)에서 whitespace 제거
  - Trim된 제목이나 내용이 빈 문자열이거나 저장 가능한 최대 길이를 초과하면 BAD\_REQUEST 에러 발생
  - DB에 새 게시물을 생성하고, 새 게시물의 상세 페이지로 redirect
- Hint: 게시물의 작성자(**author**) 정보가 어디에 저장되어 있는지 생각해본다.

### editArticleForm

- GET /article/edit/:articleId(\d+) → **articleId**에 해당하는 게시물 에디터 페이지 응답
  - DB에서 id가 **articleId**인 게시물 조회 → 없으면 NOT\_EXIST 에러 발생
  - 수정할 게시물의 제목과 내용이 포함된 게시물 에디터 페이지 응답
- 템플릿: views/articles/editor.pug
  - **user**: 현재 로그인된 사용자 객체
  - **article**: 수정할 게시물 객체

### editArticle

- POST /article/edit/:articleId(\d+) → **articleId**에 해당하는 게시물의 제목과 내용을 HTTP 요청의 body의 제목과 내용으로 수정
  - trim 메서드를 이용하여 제목(**title**)과 내용(**content**)에서 whitespace 제거
  - Trim된 제목이나 내용이 빈 문자열이거나 저장 가능한 최대 길이를 초과하면 BAD\_REQUEST 에러 발생
  - DB에서 id가 **articleId**이고 요청을 보낸 사용자가 작성한 게시물 조회 → 없으면 NOT\_EXIST 에러 발생
  - DB에 해당 게시물의 변경 사항을 반영한 뒤, 해당 게시물의 상세 페이지로 redirect

## deleteArticle

- GET /article/delete/:articleId(\d+) → articleId에 해당하는 게시물을 삭제
  - DB에서 id가 articleId이고 요청을 보낸 사용자가 작성한 게시물 조회 → 없으면 NOT\_EXIST 에러 발생
  - DB에 해당 게시물의 is\_deleted 값을 수정하여 게시물 삭제
  - 게시물 목록의 1 페이지로 redirect

게시물 작성, 수정, 삭제 기능을 수행할 때 모두 요청한 사용자의 정보가 필요하므로 이러한 기능에 관여하는 5개 라우트는 모두 로그인하지 않은 사용자에게 404 Not Found 상태를 응답하여야 한다. 이 로직은 5개 라우트에 각각 구현하여도 되지만, 미들웨어를 이용하여 이 로직을 구현해보자.

### Code 7.10 Auth Middleware

```
const authRequired = async (req, res, next) => {
  try {
    if (req.session.user) return next();
    else return res.redirect('/auth/sign_in');
  } catch (err) {
    return next(err);
  }
};

module.exports = { authRequired };
```

src/controller/auth/middleware.js를 생성하고, **Code 7.10**과 같이 작성한다. authRequired는 세션 객체에 user 객체가 있으면 다음 미들웨어를 실행하고, 없으면 로그인 페이지로 redirect 하는 미들웨어이다.

### Code 7.11 Article Routing

```
const { Router } = require('express');

const ctrl = require('./ctrl');
const { authRequired } = require('../auth/middleware');

const router = Router();

router.get('/:articleId(\d+)', ctrl.readArticle);

router.get('/compose', authRequired, ctrl.writeArticleForm);
router.post('/compose', authRequired, ctrl.writeArticle);

router.get('/edit/:articleId(\d+)', authRequired, ctrl.editArticleForm);
router.post('/edit/:articleId(\d+)', authRequired, ctrl.editArticle);

router.get('/delete/:articleId(\d+)', authRequired, ctrl.deleteArticle);

module.exports = router;
```

이제 게시물 controller 함수를 라우팅하자. src/controller/article/index.js를 생성하고, **Code 7.11**과 같이 작성한다. 여기에서 로그인에 필요한 5개 라우트에 대해 컨트롤러 미들웨어 직전에 **authRequired** 미들웨어가 실행되게 함으로써 로그인하지 않은 사용자로부터 온 요청은 미리 걸러내도록 한다.

src/controller/ctrl.js에 다음 controller를 추가로 구현한 뒤 export하여라. 결과는 **Code A.12**를 참고하여라.

#### **listArticles**

- GET /articles/page/:page(\d+) → page번째 페이지에 나열되어야 하는 게시물 목록 응답
  - 요청에서 page 값 도출 → 0 이하이면 BAD\_REQUEST 에러 발생
  - 한 페이지에는 최대 10개의 게시물 정보를 나열 → 나열될 게시물들이 전체 게시물의 몇 번째 게시물부터 10개인지 계산
  - DB에서 나열할 게시물 데이터의 배열과 전체 게시물 데이터의 개수를 조회
  - 전체 게시물 데이터와 현재 페이지 번호를 이용하여 이전 페이지와 다음 페이지 존재 여부를 계산
  - 게시물 목록 페이지 응답
- 템플릿: views/articles/index.pug
  - user: 현재 로그인된 사용자 객체
  - articles: 나열할 게시물 객체들의 배열
  - page: 현재 페이지 번호 (정수 형태)
  - hasPrev: 이전 페이지의 존재 여부
  - hasNext: 다음 페이지의 존재 여부

#### **latestArticles**

- GET /articles → 게시물 목록 페이지의 1페이지로 redirect

마지막으로, 이제 src/controller/index.js를 **Code 7.12**와 같이 수정하여, 게시물과 관련된 기능을 담당하는 컨트롤러 함수들의 라우팅을 완료한다.

#### **Code 7.12** Index Routing (Added)

```
const { Router } = require('express');

const ctrl = require('./ctrl');
const article = require('./article');
const auth = require('./auth');

const router = Router();

router.get('/', ctrl.indexPage);
router.get('/articles/page/:page(\d+)', ctrl.listArticles);
router.get('/articles', ctrl.latestArticles);

router.use('/article', article);
router.use('/auth', auth);

module.exports = router;
```

이제 **Shell 7.1**과 같이 서버를 실행하고, 지금까지 작성한 기능들을 직접 테스트 해보자.

## 7.4 Deploying Web Application

### Deploying with Web Server Program

지금까지 간단한 게시물 웹 애플리케이션을 구현하였으며, 이를 **Shell 7.1**과 같이 node로 실행하여 사용해 보았다. 이제 이 애플리케이션을 실제 서비스를 위해 deploy해보자. 사실 node로 실행하여도 웹 애플리케이션이 제 기능을 하지 못하는 것은 아니고, 외부에서도 잘 접속된다.

그러나 이러한 실행 방식은 여러 한계가 있는데, 특히 성능과 보안 문제가 가장 큰 문제이다. 먼저 외부에서 웹 서버에 HTTP 요청을 보내려면 서버에 연결되어 있는 포트가 개방(open)되어 있어야 하는데, 서비스가 실제로 사용중인 포트가 외부로 직접 노출되어 있는 것은 보안상 좋지 않다. 또한, Node.js는 기본적으로 싱글 스레드를 기반으로 동작하는 플랫폼이기 때문에 동시다발적인 요청이 들어왔을 때 그 처리 속도가 크게 저하될 수밖에 없다.

이러한 문제점을 해결하기 위해 production에서는 Apache HTTPd, nginx, Cloudflare 등의 웹 서버 프로그램을 클라이언트와 서버 사이에 위치시켜 서비스를 deploy한다. 이 중 nginx는 2002년 러시아의 이고르 시쇼브에 의해 개발된 웹 서버 프로그램으로, 이전까지 압도적인 점유율을 자랑하던 Apache HTTPd의 여러 한계점으로 인해 점유율이 빠르게 늘어나고 있다. 영국의 인터넷 서비스 회사인 Netcraft의 조사에 따르면 2021년 9월 기준 Apache HTTPd와 nginx의 점유율은 각각 24.21%와 20.33%로 nginx의 점유율이 Apache HTTPd의 점유율에 거의 근접하고 있다.<sup>4</sup>

이번 장에서는 Linux 계열의 대표적인 OS인 Ubuntu 20.04에서 nginx를 이용하여 웹 애플리케이션을 deploy해본다.

### Project Configuration and nginx Installation

Shell을 열고 git(**Shell 1.5**, 17쪽), Node.js(**Shell 1.1**, 14쪽), MariaDB(**Shell 4.4**, 56쪽)를 설치한다. 먼저 MariaDB를 실행하여 웹 애플리케이션에 사용할 DB 사용자 계정과 데이터베이스를 생성하고, DB 사용자에게 데이터베이스의 모든 권한을 부여한다. (4.2절 참고)

#### Shell 7.5 Project Configuration

```
$ git clone <git-repo-url>
$ cd <repo-directory-name>
$ npm install --only=prod
$ mysql -u<db-user> -p<db-pass> -D<db-name> < schema/0000.sql
$ vim .env
```

**Shell 7.5**와 같이 git을 이용해 프로젝트를 clone하고, package.json에서 dependencies 항목의 패키지를 설치한 뒤, schema/0000.sql 파일을 DB에 import한다. 이후 Vim을 이용하여 .env 파일을 작성하는데, MODE의 값을 prod로 설정한다.

<sup>4</sup><https://news.netcraft.com/archives/2021/09/29/september-2021-web-server-survey.html>

### Shell 7.6 Install Deploy Packages

```
$ sudo apt update
$ sudo apt upgrade -y
$ sudo apt install -y ufw nginx
```

Shell 7.6과 같이 ufw, nginx 패키지를 설치한다.

### Shell 7.7 Get Public IP of Server Machine

```
$ curl ifconfig.me
```

Shell 7.7과 같이 서버 컴퓨터의 공용(public) IP 주소를 확인한다.

### Shell 7.8 Allow Port 80/TCP

```
$ sudo ufw allow 80/tcp
$ sudo ufw enable
$ sudo ufw verbose
```

Shell 7.8과 같이 80번 포트의 TCP 연결을 허용하고, ufw를 실행한다. 이제 서버 컴퓨터가 아닌 외부 컴퓨터에서 웹 브라우저를 열고, 서버 컴퓨터의 공용 IP 주소에 접속하여 **Figure 7.2**와 같은 응답을 받는지 확인한다.

## Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to [nginx.org](http://nginx.org).  
Commercial support is available at [nginx.com](http://nginx.com).

*Thank you for using nginx.*

**Figure 7.2** nginx Root Page

## nginx Configuration

이제 nginx 서버와 Express 웹 애플리케이션을 연동해보자.

### Shell 7.9 Open nginx Configuration File

```
$ sudo vim /etc/nginx/sites-available/node-simple-board
```

Shell 7.9와 같이 /etc/nginx/sites-available 디렉토리에 node-simple-board 파일을 Vim 편집기로 생성하며 연다.

### Code 7.13 nginx Configuration

```
server {  
    listen 80;  
    server_name <server-public-ip>;  
  
    charset utf-8;  
  
    location / {  
        proxy_pass http://localhost:<web-app-port>;  
    }  
}
```

/etc/nginx/sites-available/node-simple-board 파일을 **Code 7.13**과 같이 작성한다. <server-public-ip>와 <web-app-port>에는 각각 서버 컴퓨터의 공용 IP와 웹 애플리케이션의 실행 포트 번호를 작성한다. 이렇게 설정하면 외부에서 server\_name과 listen의 값, 즉 서버 공용 IP의 80번 포트로 들어오는 요청을 받아들이며, 이들 요청 중 location, 즉 / 경로로 들어오는 모든 요청을 그 내부에 명시된 proxy\_pass, 즉 Express.js 웹 애플리케이션의 주소로 보낸다. 이후 웹 애플리케이션으로부터 응답을 받으면 원래 사용자에게 다시 전송하여 응답한다.

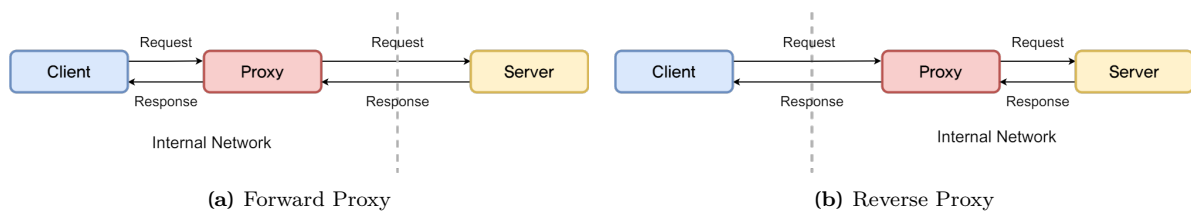


Figure 7.3 Proxy Servers

클라이언트와 서버 사이에 위치하여 한 쪽의 실제 위치를 감추는 서버를 프록시 서버(**Figure 7.3**)라고 하며, nginx와 같이 서버의 실제 위치를 감추는 서버를 리버스 프록시 서버(**Figure 7.3b**)라고 한다. 이러한 리버스 프록시 서버를 이용하면 서버의 실제 위치가 직접적으로 노출되지 않아 외부의 공격으로부터 훨씬 안전하다.

이러한 리버스 프록시 서버를 여러 번 중첩시키므로써 해킹 등을 더 안전하게 방어할 수 있고, DDoS 공격 등을 보다 효율적으로 방어할 수 있다. 또한, 이러한 server block을 여러 개 만들어서 HTTP 요청을 여러 서버에서 나누어 처리하게끔 할 수도 있고, 서버 애플리케이션과 가장 바깥쪽 프록시 서버 사이에 캐시 서버를 넣어 성능을 향상시킬 수도 있다.

### Shell 7.10 Create Symbolic Link and Restart nginx

```
$ sudo ln -s /etc/nginx/sites-available/node-simple-board \  
    /etc/nginx/sites-enabled/node-simple-board  
$ sudo service nginx restart
```

이제 **Shell 7.10**과 같이 /etc/nginx/sites-enabled 디렉토리 아래 **Code 7.13** 파일의 symbolic link<sup>5</sup>를 생성하고, nginx를 재실행한다.

<sup>5</sup>Windows의 바로가기와 비슷한 기능

#### Shell 7.11 Run Express Web Application

```
$ cd <web-app-directory>  
$ node src/index.js
```

마지막으로, Express.js 애플리케이션 프로젝트 디렉토리로 돌아와서 애플리케이션을 실행한다. 그리고 서버 컴퓨터의 공용 IP로 접속하여 이번 장에서 구현한 웹 애플리케이션이 잘 작동하는지 확인한다.





# Appendix A

## Source Codes

### Contents

A.1 Database Querying Source Codes . . . . .	122
A.2 Web Application Implementation Source Codes . . . . .	126

## A.1 Database Querying Source Codes

### Code A.1 scores Table SQL

```
CREATE TABLE `scores` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `student` VARCHAR(32) NOT NULL,  
  `course` VARCHAR(32) NOT NULL,  
  `midterm` INT NOT NULL,  
  `final` INT NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
  
INSERT INTO `scores` VALUES (1,'Barack','Discrete Mathematics',61,87),  
(2,'Joe','Discrete Mathematics',97,92),(3,'Barack','Machine Learning',73,61),  
(4,'Donald','Operating Systems',58,98),(5,'Joe','Machine Learning',63,78),  
(6,'Donald','Discrete Mathematics',91,58),(7,'Donald','Machine Learning',68,82),  
(8,'Joe','Operating Systems',72,66);
```

### Code A.2 SQL Join Example Data

```
CREATE TABLE `tb1` (`id` INT, `data1` VARCHAR(4), PRIMARY KEY (`id`)) ENGINE=InnoDB;  
CREATE TABLE `tb2` (`id` INT, `data2` VARCHAR(4), PRIMARY KEY (`id`)) ENGINE=InnoDB;  
  
INSERT INTO `tb1` VALUES (1,'1-1'),(6,'1-6'),(7,'1-7'),(8,'1-8'),(10,'1-10'),  
(11,'1-11'),(12,'1-12'),(13,'1-13'),(14,'1-14'),(15,'1-15');  
INSERT INTO `tb2` VALUES (2,'2-2'),(3,'2-3'),(4,'2-4'),(5,'2-5'),(7,'2-7'),(8,'2-8'),  
(9,'2-9'),(11,'2-11'),(13,'2-13'),(14,'2-14');
```

### Code A.3 University System SQL

```
CREATE TABLE `colleges` (  
  `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  `name` VARCHAR(50) NOT NULL,  
  `code` INT NOT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
  
CREATE TABLE `departments` (  
  `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  `name` VARCHAR(50) NOT NULL,  
  `college` INT NOT NULL,  
  FOREIGN KEY (`college`) REFERENCES `colleges`(`id`) ON DELETE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
  
CREATE TABLE `courses` (  
  `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  `name` VARCHAR(50) NOT NULL,  
  `department` INT NOT NULL,
```

```

`code` VARCHAR(8) NOT NULL,
`is_major` TINYINT(1) NOT NULL,
`is_required` TINYINT(1) NOT NULL,
`credit` INT NOT NULL,
`period` INT NOT NULL,
FOREIGN KEY (`department`) REFERENCES `departments`(`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `colleges` VALUES (1,'Liberal Arts',13),(2,'Science',16),
(3,'Informatics',32);

INSERT INTO `departments` VALUES (1,'Computer Science and Engineering',3),
(2,'Chemistry',2),(3,'Physics',2),(4,'Linguistics',1);

INSERT INTO `courses` VALUES (1,'Python for Everybody',1,'COSE156',0,0,3,4),
(2,'Discrete Mathematics',1,'COSE211',1,0,3,3),
(3,'Operating Systems',1,'COSE341',1,1,3,3),
(4,'Machine Learning',1,'COSE362',1,0,3,3),(5,'Capstone Design',1,'COSE489',1,0,3,6),
(6,'Organic Chemistry Laboratory',2,'CHEM214',1,1,2,4),
(7,'Solid State Physics',3,'PHYS482',1,0,3,3),
(8,'Inorganic Chemistry II',2,'CHEM308',1,1,3,3);

```

#### Code A.4 crud Table SQL

```

CREATE TABLE `crud` (
  `id` INT NOT NULL AUTO_INCREMENT, `c1` INT NOT NULL, `c2` INT NOT NULL,
  `c3` VARCHAR(8) NOT NULL, `c4` TIMESTAMP NOT NULL DEFAULT current_timestamp(),
  `c5` TINYINT(1) NOT NULL DEFAULT 1, PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `crud` (c1,c2,c3,c5) VALUES (5,5,'col1',1),(17,3,'col2',1),
(7,1,'col3',1),(12,1,'col4',1),(7,2,'col5',1),(20,2,'col6',1),(20,2,'col7',1),
(11,2,'col8',1),(13,2,'col9',1),(19,5,'col10',1),(5,4,'col11',1),(7,1,'col12',1),
(7,3,'col13',1),(16,5,'col14',0),(17,3,'col15',1),(15,2,'col16',1),(15,5,'col17',1),
(10,4,'col18',0),(14,2,'col19',1),(5,5,'col20',1),(8,5,'col21',1),(20,3,'col22',0),
(9,3,'col23',1),(10,1,'col24',1),(10,4,'col25',1),(4,4,'col26',1),(12,1,'col27',1),
(3,5,'col28',1),(14,5,'col29',1),(16,3,'col30',1),(3,3,'col31',1),(9,2,'col32',1),
(1,3,'col33',1),(15,5,'col34',0),(6,1,'col35',1),(2,2,'col36',1),(3,2,'col37',0),
(14,3,'col38',1),(6,3,'col39',1),(13,4,'col40',1),(19,4,'col41',1),(16,2,'col42',1),
(12,5,'col43',1),(14,1,'col44',0),(17,2,'col45',0),(18,3,'col46',1),(18,5,'col47',1),
(9,1,'col48',1),(16,3,'col49',1),(18,4,'col50',1),(18,4,'col51',0),(11,3,'col52',0),
(8,4,'col53',1),(4,4,'col54',0),(11,2,'col55',0),(14,3,'col56',1),(9,2,'col57',1),
(12,1,'col58',1),(5,4,'col59',1),(19,5,'col60',1),(12,5,'col61',1),(6,5,'col62',1),
(1,3,'col63',1),(1,1,'col64',0),(3,1,'col65',1),(7,2,'col66',1),(18,2,'col67',1),
(17,1,'col68',1),(11,5,'col69',1),(1,5,'col70',1),(20,5,'col71',1),(8,1,'col72',1),
(20,5,'col73',1),(2,3,'col74',1),(12,3,'col75',1),(5,3,'col76',1),(2,1,'col77',1),
(5,4,'col78',0),(5,4,'col79',1),(11,5,'col80',1),(19,5,'col81',1),(16,3,'col82',1),
(4,1,'col83',0),(9,1,'col84',0),(8,4,'col85',1),(20,3,'col86',1),(15,1,'col87',1),
(5,5,'col88',0),(2,1,'col89',1),(6,4,'col90',0),(14,2,'col91',1),(3,2,'col92',1),
(18,2,'col93',1),(14,3,'col94',1),(3,1,'col95',1),(1,3,'col96',1),(12,4,'col97',0),
(10,3,'col98',1),(11,3,'col99',1),(9,3,'col100',1);

```

### Code A.5 Train System Tables SQL

```
CREATE TABLE `stations` (  
  `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  `name` VARCHAR(20) NOT NULL  
) ENGINE=InnoDB;  
  
CREATE TABLE `types` (  
  `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  `name` VARCHAR(20) NOT NULL,  
  `max_seats` VARCHAR(20) NOT NULL,  
  `fare_rate` INT NOT NULL  
) ENGINE=InnoDB;  
  
CREATE TABLE `trains` (  
  `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  `type` INT NOT NULL,  
  `source` INT NOT NULL,  
  `destination` INT NOT NULL,  
  `departure` TIME NOT NULL,  
  `arrival` TIME NOT NULL,  
  `distance` INT NOT NULL,  
  FOREIGN KEY (`type`) REFERENCES `types`(`id`) ON DELETE CASCADE,  
  FOREIGN KEY (`source`) REFERENCES `stations`(`id`) ON DELETE CASCADE,  
  FOREIGN KEY (`destination`) REFERENCES `stations`(`id`) ON DELETE CASCADE  
) ENGINE=InnoDB;  
  
CREATE TABLE `users` (  
  `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  `name` VARCHAR(20) NOT NULL  
) ENGINE=InnoDB;  
  
CREATE TABLE `tickets` (  
  `id` INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
  `train` INT NOT NULL,  
  `user` INT NOT NULL,  
  `seat_number` INT NOT NULL,  
  FOREIGN KEY (`train`) REFERENCES `trains`(`id`) ON DELETE CASCADE,  
  FOREIGN KEY (`user`) REFERENCES `users`(`id`) ON DELETE CASCADE  
) ENGINE=InnoDB;
```

### Code A.6 Train System Data SQL

```
INSERT INTO `stations` VALUES (1,'Seoul'),(2,'Busan'),(3,'GwangjuSongjeong'),
(4,'Gangneung'),(5,'Suseo'),(6,'Yeosu EXPO');

INSERT INTO `types` VALUES (1,'KTX',8,15524),(2,'SRT',8,13712),(3,'KTX-Eum',7,14095),
(4,'ITX-Saemaeul',6,9645),(5,'Mugunghwa',5,6475);

INSERT INTO `trains` VALUES (1,4,6,1,'08:55','13:24',4299),
(2,1,1,2,'14:30','17:11',4174),(3,3,4,1,'18:40','20:51',2231),
(4,1,2,1,'08:20','11:32',4174),(5,2,3,5,'13:00','14:37',2905),
(6,3,1,4,'09:01','10:57',2231),(7,1,2,1,'16:38','19:12',4174),
(8,3,4,1,'12:25','14:27',2231),(9,1,1,2,'07:30','10:12',4174),
(10,5,3,2,'10:31','16:04',3253),(11,2,5,3,'09:40','11:26',2905),
(12,5,1,2,'09:56','15:32',4417),(13,2,3,5,'16:25','18:32',2905),
(14,5,2,3,'06:17','11:59',3253),(15,1,1,6,'07:05','10:17',3956),
(16,4,3,1,'11:26','15:06',3473),(17,2,2,5,'14:30','16:58',4001),
(18,1,1,3,'17:38','19:20',3078),(19,1,6,1,'18:05','21:18',4299),
(20,3,1,4,'19:01','21:06',2231);

INSERT INTO `users` VALUES (1,'Kelda'),(2,'Jaqueline'),(3,'Jaymes'),(4,'Elicia'),
(5,'Lenny'),(6,'Sissy'),(7,'Joni'),(8,'Axl'),(9,'Kenyon'),(10,'Marmaduke'),
(11,'Deanna'),(12,'Baldric'),(13,'Lorn'),(14,'Jayla'),(15,'Ellery'),(16,'Karen'),
(17,'Candis'),(18,'Charli'),(19,'Clint'),(20,'Phillip'),(21,'Glanville'),
(22,'Beauregard'),(23,'Azura'),(24,'Teale'),(25,'Esmond'),(26,'Mayme'),(27,'Lincoln'),
(28,'Arline'),(29,'Earleen'),(30,'Cher'),(31,'Alys'),(32,'Corbin'),(33,'Krista'),
(34,'Bill'),(35,'Jena'),(36,'Lola'),(37,'Journey'),(38,'Hylida'),(39,'Norah'),
(40,'Davy'),(41,'Trixie'),(42,'Alysia'),(43,'Karrie'),(44,'Dolph'),(45,'Wil'),
(46,'Roland'),(47,'Gayle'),(48,'Cindi'),(49,'Keila'),(50,'Asher');

INSERT INTO `tickets` VALUES (1,17,1,8),(2,14,2,4),(3,4,3,3),(4,2,3,5),(5,4,4,2),
(6,2,4,1),(7,19,5,3),(8,10,6,4),(9,3,7,5),(10,19,8,4),(11,8,9,1),(12,18,9,8),
(13,17,10,6),(14,11,11,3),(15,16,11,6),(16,18,11,5),(17,11,12,4),(18,16,12,4),
(19,18,12,6),(20,15,13,4),(21,4,14,8),(22,2,14,2),(23,11,15,1),(24,13,15,5),
(25,16,16,1),(26,18,16,3),(27,18,17,7),(28,11,18,2),(29,16,18,3),(30,1,19,5),
(31,18,19,1),(32,4,20,1),(33,9,21,5),(34,6,22,5),(35,8,22,4),(36,2,22,4),(37,6,23,3),
(38,8,23,7),(39,2,23,3),(40,4,24,6),(41,18,24,4),(42,12,25,1),(43,7,25,5),(44,10,26,2),
(45,6,27,6),(46,8,27,2),(47,18,27,2),(48,3,28,7),(49,19,29,6),(50,12,30,2),(51,11,31,8),
(52,16,31,2),(53,1,32,4),(54,2,32,6),(55,19,33,7),(56,5,34,4),(57,14,35,5),(58,16,36,5),
(59,9,37,1),(60,7,37,1),(61,11,38,5),(62,5,38,7),(63,7,39,8),(64,2,40,7),(65,9,41,6),
(66,7,41,2),(67,15,42,6),(68,10,43,3),(69,7,43,6),(70,14,44,2),(71,1,45,2),(72,9,46,7),
(73,17,46,4),(74,4,47,5),(75,2,47,8),(76,8,48,6),(77,14,49,3),(78,13,49,1),(79,10,50,1),
(80,7,50,7);
```

## A.2 Web Application Implementation Source Codes

**Code A.7** User DAO (src/DAO/user.js)

```
const { runQuery } = require('../lib/database');

const getByUsername = async username => {
  const sql = 'SELECT id, display_name AS displayName, password, ' +
    'is_active AS isActive, is_staff as isStaff FROM users ' +
    'WHERE username = ?';
  const result = await runQuery(sql, [username]);
  return result[0];
};

const create = async (username, password, displayName) => {
  const sql = 'INSERT INTO users (username, password, display_name) VALUES (?, ?, ?)';
  await runQuery(sql, [username, password, displayName]);
};

module.exports = {
  getByUsername,
  create,
};
```

**Code A.8** Article DAO (src/DAO/article.js)

```
const { runQuery } = require('../lib/database');

// formatDate, replaceDate

const getList = async (start, count) => {
  const sql = 'SELECT a.id, a.title, a.created_at AS createdAt, ' +
    'a.last_updated AS lastUpdated, u.display_name AS displayName ' +
    'FROM articles AS a INNER JOIN users AS u ' +
    'ON a.author = u.id AND a.is_active = 1 AND a.is_deleted = 0 ' +
    'ORDER BY a.id DESC LIMIT ?, ?';
  const articles = await runQuery(sql, [start, count]);
  return articles.map(replaceDate);
};

const getTotalCount = async () => {
  const sql = 'SELECT Count(*) AS articleCount FROM articles ' +
    'WHERE is_active = 1 AND is_deleted = 0';
  const { articleCount } = (await runQuery(sql))[0];
  return articleCount;
};

const getById = async id => {
  const sql = 'SELECT a.id, a.title, a.content, a.created_at AS createdAt, ' +
    'a.last_updated AS lastUpdated, a.author, ' +
```

```

        'u.display_name AS displayName FROM articles AS a ' +
        'INNER JOIN users AS u ON a.author = u.id AND ' +
        'a.id = ? AND a.is_active = 1 AND a.is_deleted = 0';
const articles = await runQuery(sql, [id]);
return replaceDate(articles[0]);
};

const getByIdAndAuthor = async (id, author) => {
    const sql = 'SELECT title, content, author, created_at AS createdAt, ' +
        'last_updated AS lastUpdated FROM articles ' +
        'WHERE id = ? AND author = ? AND is_active = 1 AND is_deleted = 0';
    const articles = await runQuery(sql, [id, author.id]);
    return replaceDate(articles[0]);
};

const create = async (title, content, author) => {
    const sql = 'INSERT INTO articles (title, content, author) VALUES (?, ?, ?)';
    const { insertId } = await runQuery(sql, [title, content, author.id]);
    return insertId;
};

const update = async (id, title, content) => {
    const sql = 'UPDATE articles SET title = ?, content = ? WHERE id = ?';
    await runQuery(sql, [title, content, id]);
};

const remove = async id => {
    const sql = 'UPDATE articles SET is_deleted = 1 WHERE id = ?';
    await runQuery(sql, [id]);
};

module.exports = {
    getList,
    getTotalCount,
    getById,
    getByIdAndAuthor,
    create,
    update,
    remove,
};

```

#### Code A.9 Index Controller (src/controller/ctrl.js)

```

// GET /
const indexPage = async (req, res, next) => {
    try {
        const { user } = req.session;
        return res.render('index.pug', { user });
    } catch (err) {
        return next(err);
    }
};

```



```
module.exports = { indexPage };
```

**Code A.10** Auth Controller (src/controller/auth/ctrl.js)

```
const { UserDAO } = require('../..../DAO');
const { generatePassword, verifyPassword } = require('../..../lib/authentication');

// GET /auth/sign_in
const signInForm = async (req, res, next) => {
  try {
    const { user } = req.session;
    if (user) return res.redirect('/');
    else return res.render('auth/sign-in.pug', { user });
  } catch (err) {
    return next(err);
  }
};

// POST /auth/sign_in
const signIn = async (req, res, next) => {
  try {
    const { username, password } = req.body;
    if (!username || !password) throw new Error('BAD_REQUEST');

    const user = await UserDAO.getByUsername(username);
    if (!user) throw new Error('UNAUTHORIZED');
    const isValid = await verifyPassword(password, user.password);
    if (!isValid) throw new Error('UNAUTHORIZED');

    const { id, displayName, isActive, isStaff } = user;
    req.session.user = { id, username, displayName, isActive, isStaff };
    return res.redirect('/');
  } catch (err) {
    return next(err);
  }
};

// GET /auth/sign_up
const signUpForm = async (req, res, next) => {
  try {
    const { user } = req.session;
    return res.render('auth/sign-up.pug', { user });
  } catch {
    return next(err);
  }
};

// POST /auth/sign_up
const signUp = async (req, res, next) => {
  try {
    const { username, password, displayName } = req.body;
    if (
```

```

        !username ||
        username.length > 16 ||
        !password ||
        !displayName ||
        displayName.length > 32
    )
    throw new Error('BAD_REQUEST');

    const hashedPassword = await generatePassword(password);
    await UserDAO.create(username, hashedPassword, displayName);

    return res.redirect('/auth/sign_in');
} catch (err) {
    return next(err);
}
};

// GET /auth/sign_out
const signOut = async (req, res, next) => {
    try {
        req.session.destroy(err => {
            if (err) throw err;
            else return res.redirect('/');
        });
    } catch (err) {
        return next(err);
    }
};

module.exports = {
    signInForm,
    signIn,
    signUpForm,
    signUp,
    signOut,
};

```

**Code A.11** Article Controller (src/controller/article/ctrl.js)

```

const { ArticleDAO } = require('../../DAO');

// GET /article/:articleId(\\d+)
const readArticle = async (req, res, next) => {
    try {
        const { user } = req.session;
        const { articleId } = req.params;

        const article = await ArticleDAO.getById(articleId);
        if (!article) throw new Error('NOT_FOUND');

        return res.render('articles/details.pug', { user, article });
    } catch (err) {

```

```

        return next(err);
    }
};

// GET /article/compose
const writeArticleForm = async (req, res, next) => {
    try {
        const { user } = req.session;
        return res.render('articles/editor.pug', { user });
    } catch (err) {
        return next(err);
    }
};

// POST /article/compose
const writeArticle = async (req, res, next) => {
    try {
        const { user } = req.session;
        const title = req.body.title.trim();
        const content = req.body.content.trim();
        if (!title || title.length > 50 || !content || content.length > 65535)
            throw new Error('BAD_REQUEST');

        const newArticleId = await ArticleDAO.create(title, content, user);
        return res.redirect(`/article/${newArticleId}`);
    } catch (err) {
        return next(err);
    }
};

// GET /article/edit/:articleId(\\d+)
const editArticleForm = async (req, res, next) => {
    try {
        const { user } = req.session;
        const { articleId } = req.params;

        const article = await ArticleDAO.getByIdAndAuthor(articleId, user);
        if (!article) throw new Error('NOT_FOUND');

        return res.render('articles/editor.pug', { user, article });
    } catch (err) {
        return next(err);
    }
};

// POST /article/edit/:articleId(\\d+)
const editArticle = async (req, res, next) => {
    try {
        const { user } = req.session;
        const { articleId } = req.params;

        const article = await ArticleDAO.getByIdAndAuthor(articleId, user);
        if (!article) throw new Error('NOT_FOUND');
    }
};

```

```

        const title = req.body.title.trim();
        const content = req.body.content.trim();
        if (!title || title.length > 50 || !content || content.length > 65535)
            throw new Error('BAD_REQUEST');

        await ArticleDAO.update(articleId, title, content);
        return res.redirect(`/article/${articleId}`);
    } catch (err) {
        return next(err);
    }
};

// GET /article/delete/:articleId(\\d+)
const deleteArticle = async (req, res, next) => {
    try {
        const { user } = req.session;
        const { articleId } = req.params;

        const article = await ArticleDAO.getByIdAndAuthor(articleId, user);
        if (!article) throw new Error('NOT_FOUND');

        await ArticleDAO.remove(articleId, user);
        return res.redirect('/articles/page/1');
    } catch (err) {
        return next(err);
    }
};

module.exports = {
    readArticle,
    writeArticleForm,
    writeArticle,
    editArticleForm,
    editArticle,
    deleteArticle,
};

```

#### Code A.12 Articles Controller (Added to Code A.9)

```

const { ArticleDAO } = require('../DAO');

// indexPage

// GET /articles/page/:page(\\d+)
const listArticles = async (req, res, next) => {
    try {
        const { page } = req.params;
        const { user } = req.session;
        const pageNum = parseInt(page, 10);
        if (pageNum <= 0) throw new Error('BAD_REQUEST');
    }
};

```

```

const ARTICLES_PER_PAGE = 10;
const startIndex = (pageNum - 1) * ARTICLES_PER_PAGE;

const articles = await ArticleDAO.getList(startIndex, ARTICLES_PER_PAGE);
const articleCount = await ArticleDAO.getTotalCount();
const pageCount = Math.ceil(articleCount / ARTICLES_PER_PAGE);

return res.render('articles/index.pug', {
  user,
  articles,
  page: pageNum,
  hasPrev: pageNum > 1,
  hasNext: pageNum < pageCount,
});
} catch (err) {
  return next(err);
}
};

// GET /articles
const latestArticles = async (req, res, next) => {
  try {
    return res.redirect('/articles/page/1');
  } catch (err) {
    return next(err);
  }
};

module.exports = {
  indexPage,
  listArticles,
  latestArticles,
};

```



# Appendix B

## Exercise Answers

### Contents

B.1	Advanced Javascript Exercise Answers . . . . .	136
B.2	Express.js and View Engine Exercise Answers . . . . .	138
B.3	Database Designing Exercise Answers . . . . .	141
B.4	Database Querying Exercise Answers . . . . .	143

## B.1 Advanced Javascript Exercise Answers

### Exercise 2.1

#### Code B.1 Exercise 2.1 Answer

```
const factorial = number => {
  let result = 1;
  for (let i = 0; i < number; i++) result *= (i + 1);
  return result;
};

const permutation = (n, r) => factorial(n) / factorial(n - r);

const combination = (n, r) => permutation(n, r) / factorial(r);

const multiPermutation = (n, r) => n ** r;

const multiCombination = (n, r) => combination(n + r - 1, r);

module.exports = {
  combination,
  permutation,
  multiCombination,
  multiPermutation,
};
```

### Exercise 2.2

#### Code B.2 Exercise 2.2 Answer

```
const fs = require('fs');
const path = require('path');
const util = require('util');

const getDirList = util.promisify(fs.readdir);
const getFileStat = util.promisify(fs.stat);

const PATH = './test';

const searchDirectory = async directory => {
  const files = await getDirList(directory);
  files.forEach(async file => {
    const filePath = path.join(directory, file);
    const stat = await getFileStat(filePath);
    if (stat.isDirectory()) await searchDirectory(filePath);
    else if (path.extname(filePath) === '.js') console.log(filePath);
  });
};
```



```
(async () => {  
  try {  
    await searchDirectory(PATH);  
  } catch (err) {  
    console.error(err);  
  }  
})();
```

## B.2 Express.js and View Engine Exercise Answers

### Exercise 3.1

#### Code B.3 Exercise 3.1 Answer

```
const express = require('express');

const app = express();
const port = 3000;

app.use(express.urlencoded({ extended: true }));

const objectToString = obj =>
  Object.keys(obj).map(k => `${k}: ${obj[k]}`).join('\n');

app.get('/', (req, res) => res.send(objectToString(req.query)));
app.post('/', (req, res) => res.send(objectToString(req.body)));
app.put('/', (req, res) => res.send(objectToString(req.body)));
app.delete('/', (req, res) => res.send(objectToString(req.body)));

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

### Exercise 3.2

#### Code B.4 Exercise 3.2 Answer

```
const express = require('express');

const app = express();
const port = 3000;

app.get('/board/page/:page', (req, res) => {
  const { page } = req.params;
  res.send(`This is page #${page}`);
})

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

### Exercise 3.3

#### Code B.5 Exercise 3.3 Answer

```
const express = require('express');

const app = express();
const port = 3000;

app.get('/factorial', (req, res) => {
  const { number } = req.query;
  return res.redirect(`/factorial/${number}`);
});

app.get('/factorial/:number', (req, res) => {
  const { number } = req.params;
  const parsedNumber = parseInt(number, 10);
  let result = 1;
  for (let i = 0; i < parsedNumber; i++) result *= (i + 1);
  return res.send(`${parsedNumber}! = ${result}`);
});

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

### Exercise 3.4

#### Code B.6 Exercise 3.4 Answer (index.js)

```
const express = require('express');

const app = express();
const port = 3000;

app.set('views', `${__dirname}/views`);
app.set('view engine', 'pug');

app.use(express.urlencoded({ extended: true }));

app.get('/', (req, res) => res.render('login.pug'));

app.post('/login', (req, res) => {
  const { username, password } = req.body;
  return res.send(`Username: ${username} / Password: ${password}`);
});

app.listen(port, () => console.log(`Server listening on port ${port}!`));
```

**Code B.7** Exercise 3.4 Answer (views/login.pug)

```
doctype html
html
  head
    title Login Page
  body
    form(method='post', action='/login')
      div
        label Username:
        input#username-input(name='username', type='text')
      div
        label Password:
        input#password-input(name='password', type='password')
      div
        div Introduce yourself
        textarea#introduction-input(name='introduction')
        button(type='submit') Submit
```

## B.3 Database Designing Exercise Answers

### Exercise 4.1

#### Code B.8 Exercise 4.1 Answer

```
CREATE TABLE `students` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `name` VARCHAR(20) NOT NULL,  
  `admission_year` INT NOT NULL,  
  `major` VARCHAR(20) NOT NULL,  
  `individual_number` INT NOT NULL,  
  `phone_number` VARCHAR(11) NOT NULL,  
  `address` VARCHAR(100) NOT NULL,  
  `total_credit` INT NOT NULL DEFAULT 0,  
  `avg_credit` DOUBLE NOT NULL DEFAULT 0.0,  
  `is_attending` TINYINT(1) NOT NULL DEFAULT 1,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

### Exercise 4.2

#### Code B.9 Exercise 4.2 Answer

```
CREATE TABLE `users` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `username` VARCHAR(16) NOT NULL,  
  `password` VARCHAR(64) NOT NULL,  
  `display_name` VARCHAR(16) NOT NULL,  
  `profile_image` VARCHAR(128) NOT NULL,  
  `profile_message` VARCHAR(128) NOT NULL,  
  `is_deleted` TINYINT(1) NOT NULL DEFAULT 0,  
  `date_joined` DATETIME NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
  
CREATE TABLE `channels` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `name` VARCHAR(32) NOT NULL,  
  `created_by` INT NOT NULL,  
  `channel_link` VARCHAR(128) NOT NULL,  
  `capacity` INT NOT NULL,  
  `is_deleted` TINYINT(1) NOT NULL DEFAULT 0,  
  `created_at` DATETIME NOT NULL,  
  PRIMARY KEY (`id`),  
  FOREIGN KEY (`created_by`) REFERENCES `users`(`id`) ON DELETE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
  
CREATE TABLE `chats` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `channel_id` INT NOT NULL,  
  `created_by` INT NOT NULL,  
  `message` VARCHAR(255) NOT NULL,  
  `is_deleted` TINYINT(1) NOT NULL DEFAULT 0,  
  `created_at` DATETIME NOT NULL,  
  PRIMARY KEY (`id`),  
  FOREIGN KEY (`channel_id`) REFERENCES `channels`(`id`) ON DELETE CASCADE,  
  FOREIGN KEY (`created_by`) REFERENCES `users`(`id`) ON DELETE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```

    `id` INT NOT NULL AUTO_INCREMENT,
    `message` TEXT NOT NULL,
    `created_by` INT NOT NULL,
    `channel` INT NOT NULL,
    `created_at` DATETIME NOT NULL,
    PRIMARY KEY (`id`),
    FOREIGN KEY (`created_by`) REFERENCES `users`(`id`) ON DELETE CASCADE,
    FOREIGN KEY (`channel`) REFERENCES `channels`(`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `blocks` (
    `id` INT NOT NULL AUTO_INCREMENT,
    `blocked_by` INT NOT NULL,
    `blocked_user` INT NOT NULL,
    `created_at` DATETIME NOT NULL,
    PRIMARY KEY (`id`),
    FOREIGN KEY (`blocked_by`) REFERENCES `users`(`id`) ON DELETE CASCADE,
    FOREIGN KEY (`blocked_user`) REFERENCES `users`(`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `follows` (
    `id` INT NOT NULL AUTO_INCREMENT,
    `follower` INT NOT NULL,
    `followee` INT NOT NULL,
    `created_at` DATETIME NOT NULL,
    PRIMARY KEY (`id`),
    FOREIGN KEY (`follower`) REFERENCES `users`(`id`) ON DELETE CASCADE,
    FOREIGN KEY (`followee`) REFERENCES `users`(`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

## B.4 Database Querying Exercise Answers

### Exercise 5.1

#### Code B.10 Exercise 5.1 Answer

```
SELECT `id`, `c3`, `c5` FROM `crud` WHERE `c1` = 11 AND `c2` = 2;
SELECT * FROM `crud` WHERE `c1` > 18 OR `c2` < 2;
INSERT INTO `crud` (`c1`, `c2`, `c3`, `c5`) VALUES (7, 4, 'col101', 0);
INSERT INTO `crud` VALUES (103, 3, 3, 'col103', DEFAULT, 1);
SELECT * FROM `crud` WHERE `id` > 100;
UPDATE `crud` SET `c3` = 'col0', `c5` = 0 WHERE `c1` > 4 AND `c1` < 9 AND `c2` = 1;
SELECT * FROM `crud` WHERE `c1` > 4 AND `c1` < 9 AND `c2` = 1;
DELETE FROM `crud` WHERE `c5` = 0;
SELECT * FROM `crud` WHERE `c5` = 0;
```

### Exercise 5.2

#### Code B.11 Exercise 5.2 Answer

```
-- (1)
SELECT `users`.`id`, `users`.`name`, `tickets`.`seat_number` FROM `tickets`
INNER JOIN `users` ON `users`.`id` = `tickets`.`user` AND `tickets`.`train` = 11
ORDER BY `tickets`.`seat_number`;

-- (2)
SELECT `users`.`id`, `users`.`name`, Count(*) AS `trains_count`,
       Sum(`trains`.`distance`) / 10 AS `total_distance`
FROM   `tickets`
       INNER JOIN `trains` ON `trains`.`id` = `tickets`.`train`
       INNER JOIN `users` ON `users`.`id` = `tickets`.`user`
GROUP BY `users`.`id`
ORDER BY `total_distance` DESC
LIMIT 6;

-- (3)
SELECT `trains`.`id`, `types`.`name` AS `type`, `src`.`name` AS `src_stn`,
       `dst`.`name` AS `dst_stn`, Timediff(`arrival`, `departure`) AS `travel_time`
FROM   `trains`
       INNER JOIN `types` ON `types`.`id` = `trains`.`type`
       INNER JOIN `stations` AS `src` ON `src`.`id` = `trains`.`source`
       INNER JOIN `stations` AS `dst` ON `dst`.`id` = `trains`.`destination`
ORDER BY `travel_time` DESC
LIMIT 6;

-- (4)
SELECT `types`.`name` AS `type`, `src`.`name` AS `src_stn`,
       `dst`.`name` AS `dst_stn`, `trains`.`departure`, `trains`.`arrival`,
       Round(`types`.`fare_rate` * `trains`.`distance` / 1000, -2) as `fare`
```

```

FROM   `trains`
      INNER JOIN `types` ON `types`.`id` = `trains`.`type`
      INNER JOIN `stations` AS `src` ON `src`.`id` = `trains`.`source`
      INNER JOIN `stations` AS `dst` ON `dst`.`id` = `trains`.`destination`
ORDER  BY `departure`;

-- (5)
SELECT `trains`.`id`, `types`.`name` AS `type`,
      `src`.`name` AS `src_stn`, `dst`.`name` AS `dst_stn`,
      Count(*) AS `occupied`, `types`.`max_seats` AS `maximum`
FROM   `tickets`
      INNER JOIN `trains` ON `trains`.`id` = `tickets`.`train`
      INNER JOIN `types` ON `types`.`id` = `trains`.`type`
      INNER JOIN `stations` AS `src` ON `src`.`id` = `trains`.`source`
      INNER JOIN `stations` AS `dst` ON `dst`.`id` = `trains`.`destination`
GROUP  BY `tickets`.`train`
ORDER  BY `trains`.`id`;

-- (6)
SELECT `trains`.`id`, `types`.`name` AS `type`,
      `src`.`name` AS `src_stn`, `dst`.`name` AS `dst_stn`,
      Count(`tickets`.`id`) AS `occupied`, `types`.`max_seats` AS `maximum`
FROM   `tickets`
      RIGHT JOIN `trains` ON `trains`.`id` = `tickets`.`train`
      INNER JOIN `types` ON `types`.`id` = `trains`.`type`
      INNER JOIN `stations` AS `src` ON `src`.`id` = `trains`.`source`
      INNER JOIN `stations` AS `dst` ON `dst`.`id` = `trains`.`destination`
GROUP  BY `tickets`.`train`
ORDER  BY `trains`.`id`;

```

## Exercise 5.3

### Code B.12 Exercise 5.3 Answer

```

const express = require('express');
const { runQuery } = require('./database');

const app = express();
const port = 3000;

app.get('/fare', async (req, res, next) => {
  try {
    const { uid } = req.query;
    const sql = 'SELECT users.name, ' +
      'Sum(Round(types.fare_rate * trains.distance / 1000, -2)) AS totalFare ' +
      'FROM tickets ' +
      'INNER JOIN users ON tickets.user = users.id AND users.id = ? ' +
      'INNER JOIN trains ON tickets.train = trains.id ' +
      'INNER JOIN types ON trains.type = types.id';
    const { name, totalFare } = (await runQuery(sql, [uid]))[0];
    return res.send(`Total fare of ${name} is ${totalFare} KRW.`);
  }
});

```



```

    } catch (err) {
      console.error(err);
      return res.sendStatus(500);
    }
  });

app.get('/train/status', async (req, res, next) => {
  try {
    const { tid } = req.query;
    const sql = 'SELECT Count(*) AS occupied, max_seats AS maximum FROM tickets ' +
      'INNER JOIN trains ON trains.id = tickets.train AND trains.id = ? ' +
      'INNER JOIN types ON trains.type = types.id';
    const { occupied, maximum } = (await runQuery(sql, [tid]))[0];
    const isSeatsLeft = occupied < maximum;
    return res.send(`Train ${tid} is ${isSeatsLeft ? 'not ' : ''}sold out`);
  } catch (err) {
    console.error(err);
    return res.sendStatus(500);
  }
});

app.listen(port, () => console.log(`Server listening on port ${port}!`));

```

database.js 모듈은 **Code 5.1**과 **Code 5.5**를 참고하여라.