

몽고DB 와 API 서버

1. 몽고DB 이해하기
2. 프로그래밍으로 몽고DB 사용하기
3. 익스프레스 프레임워크로 API 서버 만들기

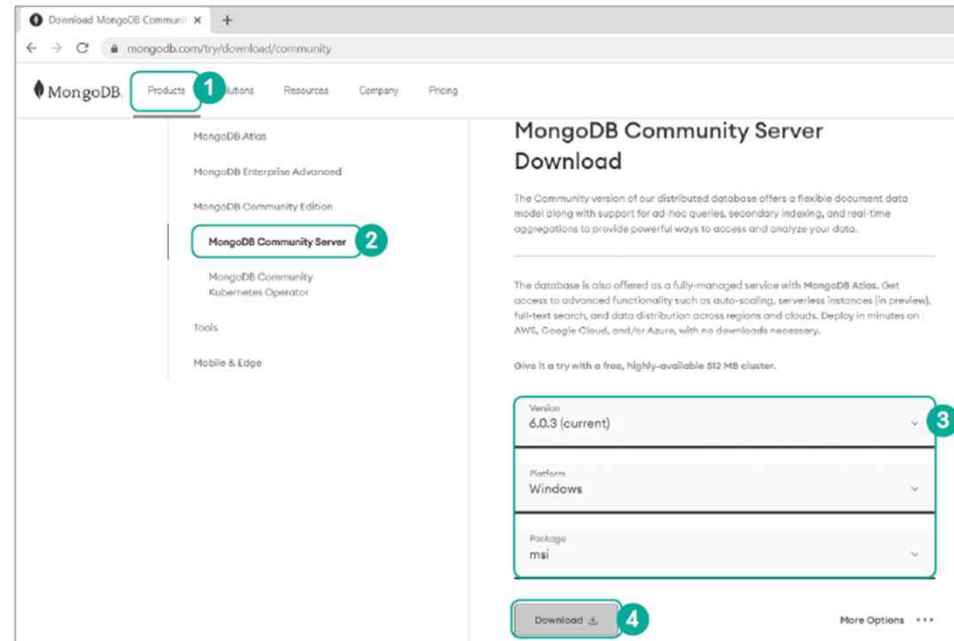
X 본 과정 학습외 다른용도로 사용 및 온라인(개인블로그 등)에 게재하면 저작권에 위배될 수 있습니다.

1 몽고DB란?

- 2009년에 처음 발표되었고, 현재 최신 버전은 6
- 사용자 측면에서 몽고DB의 가장 큰 특징은 자바스크립트를 질의어로 사용한다는 점
- 이 때문에 자바스크립트를 알기만 하면 몽고DB를 사용하려고 특별히 다른 언어를 배울 필요가 없음
- 몽고DB는 JSON 포맷으로 바꿀 수 있는 모든 자바스크립트 객체를 자유롭게 저장할 수 있으므로 매우 편리
- 현재 최신 버전인 6은 그 이전 버전과는 달리 몽고셸(mongosh) 사용. 윈도우 와 맥 등 동작 플랫폼마다 프로그램의 이름과 사용법이 동일 해졌지만, 설치 과정이 이전 버전과 다른 점이 존재

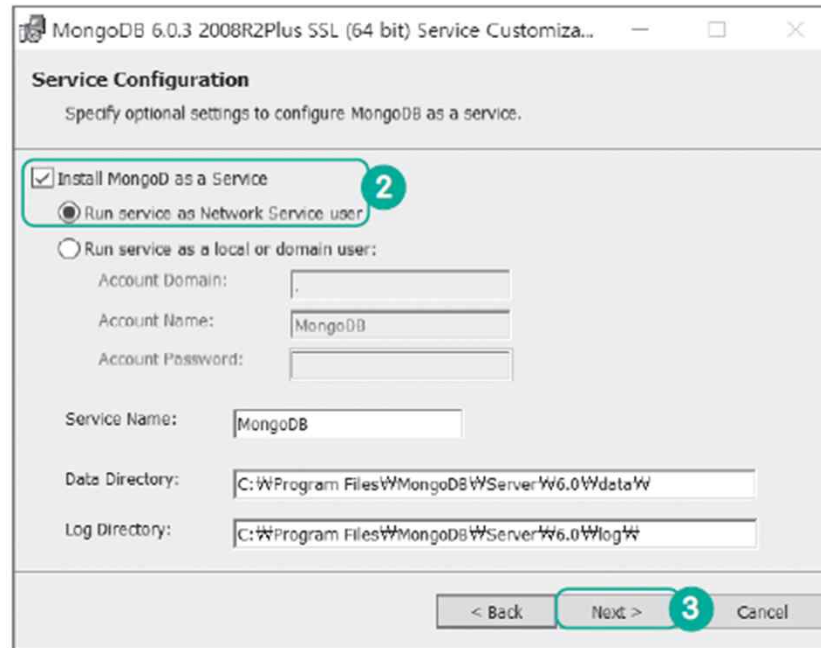
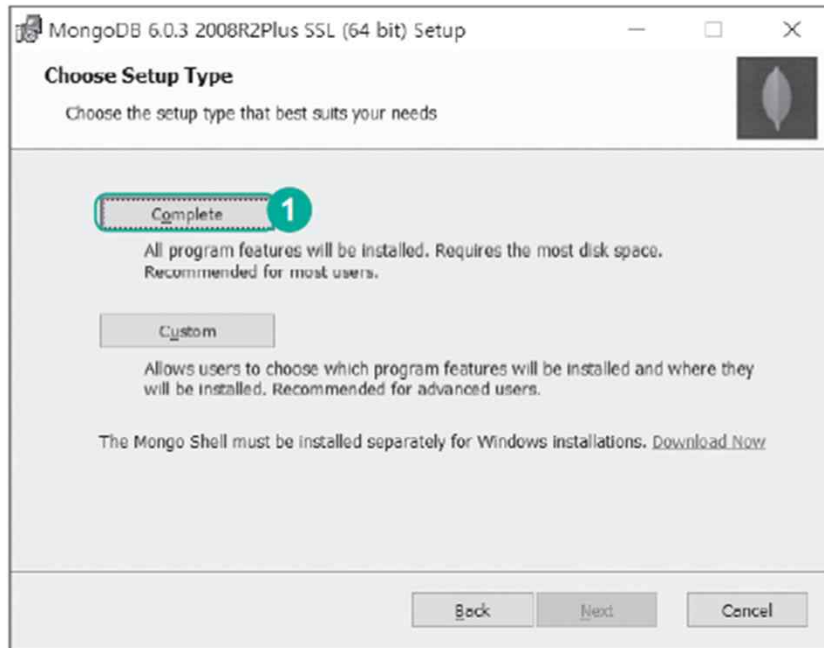
1 몽고DB 설치하기

- 몽고DB 홈페이지(mongodb.com)에서 다음 그림처럼 플랫폼별 설치 파일을 내려 받아 설치



1 계속

- 윈도우에서는 몽고 데몬(mongod)이 서비스 형태로 동작해야 함



1 몽고DB 셸 설치하기

- 몽고DB뿐만 아니라 대다수의 DB 시스템은 데몬 이나 서비스 형태로 동작하는 DB 서버와 DB 서버에 접속하여 다양한 DB 작업을 서버에 전달하는 DB 클라이언트 두 부분으로 동작
- 셸(shell)은 명령 줄 방식으로 입력한 명령을 서버에 전달하여 다양한 DB 작업을 할 수 있게 해 주는 대표적인 DB 클라이언트 프로그램
- 다음은 mongosh을 실행하는 방법(윈도우 와 맥 공통)

A screenshot of a terminal window. The title bar at the top says "T 터미널" (Terminal) and has standard window control buttons (minimize, maximize, close). The main area of the terminal shows a prompt character ">" followed by the text "mongosh".

```
T 터미널
> mongosh
```

1 계속

- 윈도우 에서 mongosh 실행 모습

```
PS C:\> mongosh
Current Mongosh Log ID: 63bafb3f9d13023505ff515e
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=20000
Using MongoDB:      6.0.3
Using Mongosh:       1.6.1

For mongosh info see: https://docs.mongodb.com/mongosh/

-----
The server generated these startup warnings when booting
2023-01-08T23:31:56.597+09:00: Access control is not enabled for the database. Read and write access without
authorization is unrestricted.
2023-01-08T23:31:56.597+09:00: You are running on a NUMA machine. We suggest disabling NUMA in the BIOS to
improve performance. See your BIOS documentation for more information.
-----

-----
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
-----

Warning: Found ~/.mongorc.js, but not ~/.mongoshrc.js. ~/.mongorc.js will not be loaded.
You may want to copy or rename ~/.mongorc.js to ~/.mongoshrc.js.
test>
```

1 DB 목록 조회하기

- 몽고셸에서 show dbs 명령은 현재 설치된 데이터베이스의 목록을 알고 싶을 때 사용
- 몽고DB가 설치될 때 기본으로 생성되는 3개의 DB를 보여 줌

```
ch12-2> show dbs
admin      40.00 KiB
config     72.00 KiB
local      72.00 KiB
```

1 DB 선택하기

- DB를 사용하려면 먼저 'use DB_이름' 형태의 use 명령으로 DB를 선택해야 함
- 다음은 local이라는 DB를 선택하는 명령



```
M 몽고셸
> use local
switched to db local
```

The image shows a screenshot of a MongoDB Shell window. The title bar at the top says 'M 몽고셸' (M MongoDB Shell) and has standard window control buttons (minimize, maximize, close). The main area of the window displays a command prompt where the user has entered the command 'use local'. The shell has responded with the message 'switched to db local', indicating that the database has been successfully selected.

1 사용 중인 DB 이름 보기

- 가끔 현재 사용 중인 DB 이름을 알아야 할 때가 있는데 이때는 db 명령을 사용



```
MongoShell
> db
local
```

The image shows a MongoDB Shell window titled 'M 몽고셸'. The command prompt is '> db' and the output is 'local'. The window has standard window controls (minimize, maximize, close) in the top right corner.

1 DB 생성하기

- 몽고DB는 흥미롭게도 새로운 DB를 생성하는 명령이 없음
- 몽고DB는 “있으면 사용하고 없으면 생성하는” 형태로 동작
- 따라서 mydb라는 이름의 DB를 새로 만들고 싶으면 다음처럼 use 명령을 사용

```
test> show dbs
admin    40.00 KiB
config  84.00 KiB
local   72.00 KiB
test> use mydb
switched to db mydb
```

- 몽고DB는 이 시점에서 새로운 DB를 생성하지 않음
- 새로운 DB는 실제 데이터가 새로운 DB에 생성될 때 함께 만들어 짐
- 다음 명령은 실제 데이터를 하나 만드는 예

```
mydb> db.user.insertOne({ name: 'Jack' })
{
  acknowledged: true,
  insertedId: ObjectId("63bb1586686781bc2e486922")
}
```

1 특정 DB 지우기

- 현재 사용 중인 DB를 삭제하려면 일단 use 명령으로 삭제하고 싶은 DB를 선택해야 함
- 다음은 선택한 DB를 삭제하는 명령

```
local> use mydb
switched to db mydb
mydb> db.dropDatabase()
{ ok: 1, dropped: 'mydb' }
```

1 컬렉션과 문서

- 몽고DB에서는 RDBMS의 테이블을 컬렉션(collection)이라고 함
- RDBMS의 레코드에 해당하는 한 건의 데이터를 문서(document)라고 함
- 즉, 몽고DB에서 컬렉션이란 스키마 없이 자유롭게 작성된 여러 개의 문서를 보관하는 저장소

1 새로운 컬렉션 만들기

- DB에 새로운 컬렉션을 생성하려면 다음 명령을 사용
- 다음은 "user"라는 이름의 컬렉션을 기본 옵션으로 생성한 예
- 만약 컬렉션 설정 옵션을 지정하고 싶으면 {} 안에 작성함

```
mydb> db.createCollection("user", {})  
{ ok: 1 }
```

1 DB의 컬렉션 목록 보기

- 현재 사용 중인 DB의 모든 컬렉션을 보려면 다음 명령을 실행

```
mydb> db.getCollectionNames()  
[ 'user' ]
```

1 컬렉션 삭제하기

- 생성한 컬렉션을 삭제하려면 다음 명령을 사용함
- user라는 이름의 컬렉션을 삭제하는 예

```
mydb> db.user.drop()  
true  
mydb> db.getCollectionNames()  
[]
```

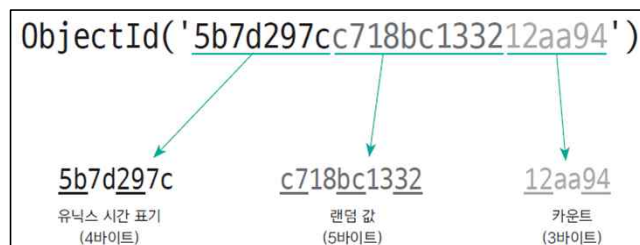
- 컬렉션을 삭제했으므로 db.getCollectionNames()는 더 이상 user를 반환하지 않음

1 _id 필드와 ObjectId 타입

- 모든 몽고DB 문서는 _id라는 특별한 이름의 필드field를 가지는데, 이 필드는 문서가 DB에 저장 될 때 자동으로 만들어 짐
- _id 필드는 UUID와 개념적으로 같지만, - 기호 없이 ObjectId("문자열") 형태로 사용해야 함

```
mydb> db.user.insertOne({name: 'Jack'})
{
  acknowledged: true,
  insertedId: ObjectId("63bb1884686781bc2e486923")
}
mydb> db.user.find({})
[ { _id: ObjectId("63bb1884686781bc2e486923"), name: 'Jack' } ]
```

- 몽고DB는 ObjectId("문자열") 부분을 다음처럼 생성하고 해석



1 컬렉션의 CRUD 메서드

- 몽고DB는 오른쪽 표에서 보는 CRUD 메서드들을 제공
- 이 메서드들의 이름을 살펴보면 문서 1개를 대상으로 하는 메서드는 '이름One' 형태의 'One'이란 접미사를 사용
- 여러 문서를 대상으로 하는 메서드는 '이름Many' 형태의 'Many'란 접미사를 사용(단, find만 제외).

표 7-1 몽고DB의 CRUD 메서드

컬렉션 메서드 이름	의미
insertOne	생성(create operations)
insertMany	
findOne	검색(read operations)
find	
updateOne	수정(update operations)
updateMany	
findOneAndUpdate	
deleteOne	삭제(delete operations)
deleteMany	

1 문서 생성 메서드 사용하기

- 앞서 언급한 대로 몽고DB는 스키마가 없는 DB. 컬렉션을 `db.createCollection` 메서드로 생성하지 않아도, `db.컬렉션_이름.insertOne` 형태의 명령을 사용하면, 해당 이름으로 컬렉션이 자동 생성됨
- 다음은 `insertOne` 과 `insertMany`의 실행 모습

```
mydb> db.user.insertOne({name: 'Jack', age: 32})
{
  acknowledged: true,
  insertedId: ObjectId("63bb56a4686781bc2e486924")
}
```

```
mydb> db.user.insertMany([ {name: 'Jane', age: 22}, {name: 'Tom', age: 11} ] )
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("63bb56ce686781bc2e486925"),
    '1': ObjectId("63bb56ce686781bc2e486926")
  }
}
```

1 문서 검색 메서드 사용하기

- `findOne`과 `find` 메서드는 다음처럼 검색 조건(query)을 매개변수로 전달해야 하며, 검색 조건은 객체 형태로 표현해야 함
- 만일 컬렉션의 모든 문서를 검색하고 싶다면 검색 조건을 `{}`로 설정

```
mydb> db.user.findOne({})
{ _id: ObjectId("63bb56a4686781bc2e486924"), name: 'Jack', age: 32 }
mydb> db.user.find({})
[
  { _id: ObjectId("63bb56a4686781bc2e486924"), name: 'Jack', age: 32 },
  { _id: ObjectId("63bb56ce686781bc2e486925"), name: 'Jane', age: 22 },
  { _id: ObjectId("63bb56ce686781bc2e486926"), name: 'Tom', age: 11 }
]
```

- 다음은 name이 'Jack'일 때와 age값이 20보다 큰 문서를 찾는 예

```
mydb> db.user.find({name: 'Jack'})
[
  { _id: ObjectId("63bb56a4686781bc2e486924"), name: 'Jack', age: 32 }
]
mydb> db.user.find({age: {$gt: 20}})
[
  { _id: ObjectId("63bb56a4686781bc2e486924"), name: 'Jack', age: 32 },
  { _id: ObjectId("63bb56ce686781bc2e486925"), name: 'Jane', age: 22 }
]
```

1 몽고DB에서 연산자란?

- 몽고DB에서는 \$gt처럼 달러 기호를 접두사로 사용하는 키워드를 연산자라고 함
- 몽고DB는 많은 종류의 연산자를 제공함
- 다음은 검색 연산자 사용법

검색 연산자 사용법

```
{ 필드_이름1: { 검색_연산자1: 값1 }, 필드_이름2: { 검색_연산자2: 값2 }, ... }
```

1 비교 연산자 알아보기

- 비교 연산자로는 일반적인 프로그래밍 언어에서 두 값을 비교할 때 사용하는 '=', '>', '<' 기호와 이들의 조합을 의미하는 연산자들이 있음
- 다음 검색 명령은 name 필드 값이 'Jack'이고, age 필드 값이 10보다 큰 문서를 찾는 예

```
mydb> db.user.find({ name: {$eq: 'Jack'}, age: {$gt: 10} })
[
  { _id: ObjectId("63bb56a4686781bc2e486924"), name: 'Jack', age: 32 }
]
```

표 7-2 비교 연산자

연산자 이름	의미
\$eq	필드_값 == 값
\$ne	필드_값 != 값
\$gt	필드_값 > 값
\$gte	필드_값 >= 값
\$lt	필드_값 < 값
\$lte	필드_값 <= 값

- 가끔 어떤 필드가 여러 값 중에서 일치하는 값을 가지는 문서를 찾거나, 또는 반대로 여러 값 중에서 하나도 일치하지 않는 값을 가지는 문서를 찾아야 할 때 \$in 과 \$nin 연산자를 사용

표 7-3 \$in과 \$nin 연산자

연산자 이름	의미
\$in	{ 필드명: { \$in: [값1, 값2, ...] } } 형태로 사용하며, \$in 연산자에 설정한 배열에서 하나라도 매치되면 해당 문서 반환
\$nin	{ 필드명: { \$nin: [값1, 값2, ...] } } 형태로 사용하며, \$nin 연산자에 설정한 배열에서 하나도 매치되지 않는 문서 반환

```
mydb> db.user.find( {age: {$in: [11, 22]}} )
[
  { _id: ObjectId("63bb56ce686781bc2e486925"), name: 'Jane', age: 22 },
  { _id: ObjectId("63bb56ce686781bc2e486926"), name: 'Tom', age: 11 }
]
mydb> db.user.find( {age: {$nin: [11, 22]}} )
[
  { _id: ObjectId("63bb56a4686781bc2e486924"), name: 'Jack', age: 32 }
]
```

1 논리 연산자 알아보기

- 프로그래밍 언어에서 &&, ||, ! 등의 논리 조건을 표현할 때와 유사한 연산자

표 7-4 논리 연산자

연산자 이름	의미
\$and	모든 검색 조건을 모두 만족하는(AND) 문서를 찾을 때 사용
\$not	모든 검색 조건을 모두 만족하지 않는(NOT) 문서를 찾을 때 사용
\$or	모든 검색 조건 중 하나라도 만족하는(OR) 문서를 찾을 때 사용
\$nor	모든 검색 조건 중 하나도 만족하지 않는(NOR) 문서를 찾을 때 사용

- 다음 형태로 사용

타입스크립트 기반 리액트 프로젝트 생성 명령

```
{ 논리_연산자: [ { 검색_조건1 }, { 검색_조건2 }, ... { 검색_조건N } ] }
```

```
mydb> db.user.find({ $and: [{name: {$in: ['Jack', 'Tom']}}, {age: {$lt: 20}}] })  
[ { _id: ObjectId("63bb56ce686781bc2e486926"), name: 'Tom', age: 11 } ]
```

1 \$regex 정규 식 연산자

- 자바스크립트는 RegEx라는 이름의 클래스로 정규 표현식regular expression이란 기능을 제공함
- 정규식은 “이름이 J로 시작하는 모든 것”처럼 와일드카드 검색을 할 때 유용한 기능
- 다음은 RegEx 클래스를 사용하여 정규식을 만드는 코드

```
const re = new RegEx(정규식)
```

- 정규식을 프로그래밍 언어는 '/' 문자를 정규식 앞뒤에 붙여 좀 더 간결한 방법으로 만들 수 있게 함. RegEx 코드를 '/' 기호로 좀 더 간결하게 구현한 예

```
const re = /정규식/
```

```
mydb> db.user.find({ name: { $regex: /~J.*$/ } })
[
  { _id: ObjectId("63bb56a4686781bc2e486924"), name: 'Jack', age: 32 },
  { _id: ObjectId("63bb56ce686781bc2e486925"), name: 'Jane', age: 22 }
]
```


1 필드 업데이트 연산자 알아보기

- 컬렉션의 update 관련 메서드는 문서의 특정 필드 값을 다른 값으로 바꾸는 데 사용되며, 몽고 DB는 다음 표처럼 필드 업데이트 연산자를 제공함

표 7-5 필드 업데이트 연산자

연산자 이름	용도
\$set	{ \$set: { 필드_이름: 값, ... } } 형태로 문서의 특정 필드값을 변경할 때 사용
\$inc	{ \$inc: { 필드_이름: 값, ... } } 형태로 문서의 숫자 타입 필드값을 증가할 때 사용
\$dec	{ \$dec: { 필드_이름: 값, ... } } 형태로 문서의 숫자 타입 필드값을 감소할 때 사용

1 문서 수정 메서드 사용하기

- 앞서 언급한 대로 `updateOne`, `updateMany`, `findOneAndUpdate` 등은 컬렉션에 저장된 문서의 필드 값을 수정하는 메서드로서, 옵션 부분은 생략 가능

수정 메서드 사용법

```
db.컬렉션_이름.updateOne(검색_조건_객체, 필드_업데이트_연산자_객체, 옵션)
db.컬렉션_이름.updateMany(검색_조건_객체, 필드_업데이트_연산자_객체, 옵션)
db.컬렉션_이름.findOneAndUpdate(검색_조건_객체, 필드_업데이트_연산자_객체, 옵션)
```

-
- 다음 명령은 user 컬렉션에 저장된 문서 가운데 name 필드 값이 'J'로 시작하는 문서를 찾아 'John'으로 바꾸고 age값은 10만큼 증가시키는 예
 - 결과를 보면 검색 조건에 맞는 문서가 한 건이라는 의미로 matchedCount 속성값이 1이며, 수정된 문서가 1건이란 의미로 modifiedCount 속성값도 1임을 확인할 수 있음
 - updateOne은 검색 조건에 맞는 문서 중 1개만 선택하여 수정함. 따라서 'Jack'은 'John'이 되고 age는 +10 증가되었지만, 'Jane'은 영향을 받지 않고 있음

```
mydb> db.user.updateOne( {name: { $regex: /^J.*$/ } }, { $set: {name: 'John'}, $inc: {age: 10} } )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
mydb> db.user.find({})
[
  { _id: ObjectId("63bb56a4686781bc2e486924"), name: 'John', age: 42 },
  { _id: ObjectId("63bb56ce686781bc2e486925"), name: 'Jane', age: 22 },
  { _id: ObjectId("63bb56ce686781bc2e486926"), name: 'Tom', age: 11 }
]
```

1 문서 삭제 메서드 사용하기

- 컬렉션에 담긴 문서를 삭제할 때는 다음처럼 deleteOne과 deleteMany 메서드를 사용하며 옵션은 생략할 수 있음

문서 삭제 메서드 사용법

```
db.컬렉션_이름.deleteOne(검색_조건_객체, 옵션)
db.컬렉션_이름.deleteMany(검색_조건_객체, 옵션)
```

- 다음은 name 속성이 'J'로 시작하는 문서 한 개를 삭제하는 예로, deleteOne 이므로 'Jane'은 삭제되지 않았음

```
mydb> db.user.deleteOne( { name: { $regex: /^J.*$/ } } )
{ acknowledged: true, deletedCount: 1 }
mydb> db.user.find({})
[
  { _id: ObjectId("63bb56ce686781bc2e486925"), name: 'Jane', age: 22 },
  { _id: ObjectId("63bb56ce686781bc2e486926"), name: 'Tom', age: 11 }
]
```

몽고DB와 API 서버

- 1 몽고DB 이해하기
- 2 프로그래밍으로 몽고DB 사용하기
- 3 익스프레스 프레임워크로 API 서버 만들기

2 몽고 DB 드라이버 설치

- Node.js 프로젝트 생성- Node.js 프로젝트를 만들기 위해서는 먼저다음처럼 프로젝트 디렉터리를 생성해야 함



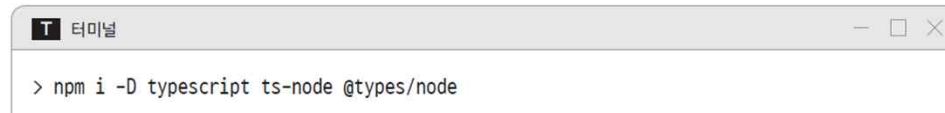
```
T 터미널
> mkdir ch07
> cd ch07
> mkdir ch07_2_server
> code ch07_2_server
```

- package.json 파일 생성 - Node.js프로젝트는 package.json 파일이 필요하며, 이 파일은 다음 명령으로 생성함



```
T 터미널
> npm init --y
```

- 몽고 DB 드라이버 설치 - Node.js와 타입스크립트로 몽고 DB 개발을 하려고 하면, 다음 파일들의 설치



```
T 터미널
> npm i -D typescript ts-node @types/node
```

- tsconfig.json 파일 생성 - 타입스크립트를 개발 언어로 하는 Node.js 프로젝트는 tsconfig.json 파일이 필요하며, 이 파일은 다음 명령으로 생성



```
T 터미널
> tsc --init
```

2 계속

- 오른쪽 파일은 이 과정을 통해 생성한 package.json 파일 내용

```
{ } package.json > ...
1  {
2    "name":    "testServer"    ,
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    ▶ 디버그
7    "scripts": {
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "keywords": [],
11   "author": "",
12   "license": "ISC",
13   "devDependencies": {
14     "@types/mongodb": "^4.0.7",
15     "@types/node": "^17.0.31",
16     "ts-node": "^10.7.0",
17     "typescript": "^4.6.4"
18   },
19   "dependencies": {
20     "mongodb": "^4.5.0"
21   }
22 }
```

2 계속

- prettier 동작을 위해 다음 내용의 .prettierrc.js 파일 생성

```
JS .prettierrc.js > ...
1  module.exports = {
2    bracketSpacing: false,
3    jsxBracketSameLine: true,
4    singleQuote: true,
5    trailingComma: 'none',
6    arrowParens: 'avoid',
7    semi: false,
8    printWidth: 90
9  };
```

- src/index.ts 파일 생성 및 내용 작성

```
PS D:\          본인 작업 경로          > mkdir src

Mode                LastWriteTime         Length Name
----                -
d-----          2023-01-10 오전 1:11             src

PS D:\          본인 작업 경로          > touch src/index.ts

src > TS index.ts
1  console.log('Hello world!')
```

- ts-node 명령 실행

```
PS D:\          본인 작업 경로          > ts-node .\src\index.ts
Hello world!
```

2 몽고DB와 연결하기

- 프로그래밍으로 다음 URL을 사용하면 몽고DB와 연결함
- 여기서 mongodb는 프로토콜 이름이고, localhost는 호스트 이름, 기본 포트는 27017

몽고DB 연결 URL

```
mongodb://localhost:27017
```

2 MongoClient

- **mongodb** 패키지는 몽고DB와 연결을 쉽게 할 수 있도록 다음처럼 **MongoClient** 클래스를 제공

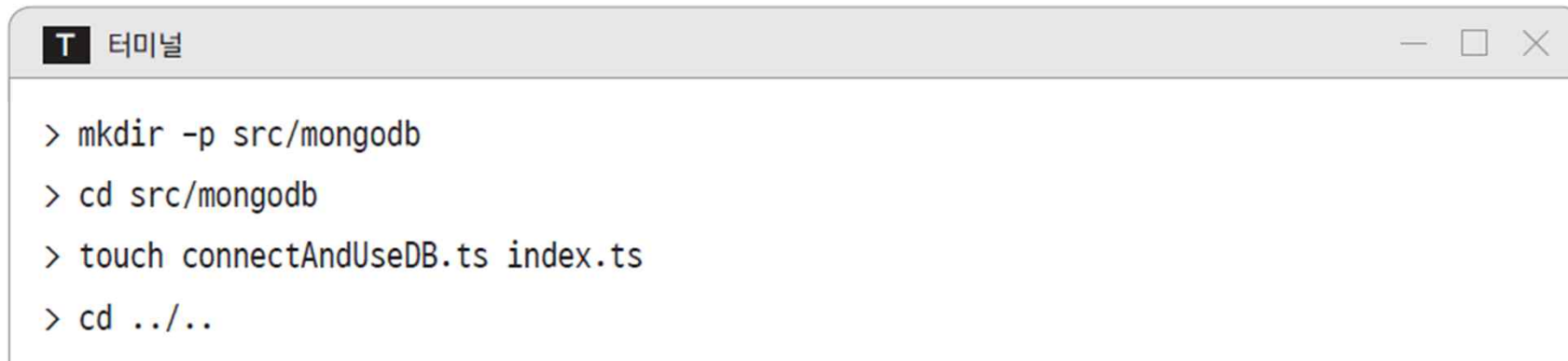
MongoClient 임포트

```
import {MongoClient} from 'mongodb'
```

- **MongoClient** 클래스는 다음처럼 **connect** 정적 메서드를 제공하여 프로미스 형태로 **Mongo Client** 인스턴스를 얻을 수 있게 함

```
static connect(url: string): Promise<MongoClient>
```

-
- 몽고DB 관련 유틸리티 함수를 만들기 위해, src 디렉터리에 mongodb란 이름의 디렉터를 만들고 이 디렉터리에 connectAndUseDB.ts 와 index.ts 파일을 생성함



```
T 터미널
> mkdir -p src/mongodb
> cd src/mongodb
> touch connectAndUseDB.ts index.ts
> cd ../../
```

- connectAndUseDB.ts 와 index.ts 파일 작성

```
src > mongodb > TS connectAndUseDB.ts > ...
1  import {MongoClient, Db} from 'mongodb'
2
3  export type MongoDB = Db
4  export type ConnectCallback = (db: MongoDB) => void
5
6  export const connectAndUseDB = async (
7    callback: ConnectCallback,
8    dbName: string,
9    mongoUrl: string = 'mongodb://127.0.0.1:27017'
10 ) => {
11   let connection
12   try {
13     connection = await MongoClient.connect(mongoUrl) // 몽고 DB와 연결
14     const db: Db = connection.db(dbName) // 몽고 셸의 'use dbName'에 해당
15     callback(db) // 얻어진 db 객체를 callback 함수의 매개변수로 하여 호출
16   } catch (e) {
17     // 타입스크립트의 타입 가드 구문 필요
18     if (e instanceof Error) {
19       console.log(e.message)
20     }
21   }
22 }
```

```
src > mongodb > TS index.ts
1  export * from './connectAndUseDB'
```

■ connectTest.ts 파일 생성 및 작성

```
T 터미널
> mkdir -p src/test
> touch src/test/connectTest.ts
```

```
src > test > TS connectTest.ts > ...
1  import * as M from '../mongodb'
2
3  const connectCB = (db: M.MongoDB) => {
4    | console.log('db', db)
5  }
6  const connectTest = () => {
7    | M.connectAndUseDB(connectCB, 'ch07')
8  }
9
10 connectTest()
```

```
PS D:          본인 작업 경로          > ts-node src/test/connectTest.ts
db Db {
  s: {
    client: MongoClient {
      _events: [Object: null prototype] {},
      _eventsCount: 0,
```

Ctrl+C 키를 눌러 종료

2 컬렉션의 CRUD 메서드

- 몽고 셸에서는 'db.컬렉션_이름' 형태로 컬렉션에 접근할 수 있었는데, 프로그래밍으로는 다음과 같은 형태로 접근해야 함

```
db.collection(컬렉션_이름)
```

2 문서 생성 메서드 사용하기

```
insertOne(doc: OptionalUnlessRequiredId<TSchema>): Promise<InsertOneResult<TSchema>>
insertMany(docs: OptionalUnlessRequiredId<TSchema>[]): Promise<InsertManyResult<TSchema>>
```

```
src > test > TS insertTest.ts > ...
1  import * as M from '../mongodb'
2
3  const connectCB = async (db: M.MongoDB) => {
4    try {
5      const user = db.collection('user')
6      try {
7        await user.drop()
8      } catch (e) {
9        // ignore error
10      }
11
12     const jack = await user.insertOne({name: 'Jack', age: 32})
13     console.log('jack', jack)
14     const janeAndTom = await user.insertMany([
15       {name: 'Jane', age: 22},
16       {name: 'Tom', age: 11}
17     ])
18     console.log('janeAndTom', janeAndTom)
19   } catch (e) {
20     if (e instanceof Error) console.log(e.message)
21   }
22 }
23
24 const insertTest = () => {
25   M.connectAndUseDB(connectCB, 'ch07')
26 }
27
28 insertTest()
```

```
PS D:\          본인 작업 경로          > ts-node .\src\test\insertTest.ts
jack {
  acknowledged: true,
  insertedId: new ObjectId("63bc46c5ac288035cc2643ca")
}
janeAndTom {
  acknowledged: true,
  insertedCount: 2,
  insertedIds: {
    '0': new ObjectId("63bc46c5ac288035cc2643cb"),
    '1': new ObjectId("63bc46c5ac288035cc2643cc")
  }
}
```


2 문서 검색 메서드 사용하기

- findOne과 find 메서드의 타입 선언문으로 프로미스 객체를 반환하는 findOne과 달리 find는 FindCursor 타입 객체를 반환

```
findOne(filter: Filter<TSchema>): Promise<WithId<TSchema> | null>  
find(filter: Filter<TSchema>, options?: FindOptions): FindCursor<WithId<TSchema>>;
```

- FindCursor 타입 객체는 다음 코드에서 보듯 toArray란 메서드를 제공하여 find의 반환값을 자바스크립트 배열로 바꿔줌

```
const cursor = await find({})  
const arrayResult = cursor.toArray()
```

- fineOne 과 find
- find는 toArray를 호출해야 함을 잊지 말 것

```
PS D:\> ts-node .\src\test\findTest.ts
one {
  _id: new ObjectId("63bc46c5ac288035cc2643ca"),
  name: 'Jack',
  age: 32
}
many [
  {
    _id: new ObjectId("63bc46c5ac288035cc2643ca"),
    name: 'Jack',
    age: 32
  },
  {
    _id: new ObjectId("63bc46c5ac288035cc2643cb"),
    name: 'Jane',
    age: 22
  },
  {
    _id: new ObjectId("63bc46c5ac288035cc2643cc"),
    name: 'Tom',
    age: 11
  }
]
```



```
src > test > TS findTest.ts > ...
1  import * as M from '../mongodb'
2
3  const connectCB = async (db: M.MongoDB) => {
4    try {
5      const user = db.collection('user')
6
7      const one = await user.findOne({})
8      console.log('one', one)
9
10     const many = await user.find({}).toArray()
11     console.log('many', many)
12   } catch (e) {
13     if (e instanceof Error) console.log(e.message)
14   }
15 }
16 const findTest = () => {
17   M.connectAndUseDB(connectCB, 'ch07')
18 }
19
20 findTest()
```

2 문서 수정 메서드 사용하기

- 다음은 컬렉션의 문서 수정 관련 메서드들의 타입 선언문으로 모두 프로미스 객체를 반환함

```
updateOne(filter:Filter<TSchema>,update:UpdateFilter<TSchema>|Partial<TSchema>):  
    Promise<UpdateResult>  
updateMany(filter: Filter<TSchema>, update: UpdateFilter<TSchema>):  
    Promise<UpdateResult | Document>;  
findOneAndUpdate(filter:Filter<TSchema>, update:UpdateFilter<TSchema>,  
    options: FindOneAndUpdateOptions):  
    Promise<ModifyResult<TSchema>>;
```

-
- **findOneAndUpdate** 메서드를 호출할 때 몽고셀에서는 **returnNewDocument** 속성값을 **true**로 했지만, 프로그래밍으로 이와 똑같은 효과를 보려면 **returnDocument** 속성값을 **'after'**로 설정해야 함

```
const findOneResult = await user.findOneAndUpdate(  
  {name: 'John'},  
  {$set: {age: 66}},  
  {returnDocument: 'after'} // 'before'와 'after' 둘 중 하나  
)
```

■ 목적

```
PS D:\          본인 작업 경로          > ts-node .\src\test\updateTest.ts
updateOneResult [
  {
    _id: new ObjectId("63bc46c5ac288035cc2643ca"),
    name: 'John',
    age: 42
  },
  {
    _id: new ObjectId("63bc46c5ac288035cc2643cb"),
    name: 'Jane',
    age: 22
  },
  {
    _id: new ObjectId("63bc46c5ac288035cc2643cc"),
    name: 'Tom',
    age: 11
  }
]
```

```
src > test > TS updateTest.ts > ...
1  import * as M from '../mongodb'
2
3  const connectCB = async (db: M.MongoDB) => {
4    try {
5      const user = db.collection('user')
6
7      await user.updateOne(
8        {name: {$regex: /^J.*$/}},
9        {$set: {name: 'John'}, $inc: {age: 10}}
10     )
11     const updateOneResult = await user.find({}).toArray()
12     console.log('updateOneResult', updateOneResult)
13
14     await user.updateMany({name: {$regex: /^J.*$/}}, {$inc: {age: 10}})
15     const updateManyResult = await user.find({}).toArray()
16     console.log('updateManyResult', updateManyResult)
17
18     const findOneResult = await user.findOneAndUpdate(
19       {name: 'John'},
20       {$set: {age: 66}},
21       {returnDocument: 'after'} // 'before' 와 'after' 두 개 값 중 하나
22     )
23     console.log('findOneResult', findOneResult)
24   } catch (e) {
25     if (e instanceof Error) console.log(e.message)
26   }
27 }
28 const updateTest = () => {
29   M.connectAndUseDB(connectCB, 'ch07')
30 }
31
32 updateTest()
```

- \$inc 연산자의 덕분으로 모든 문서의 age 속성들이 10씩 증가

```
updateManyResult [
  {
    _id: new ObjectId("63bc46c5ac288035cc2643ca"),
    name: 'John',
    age: 52
  },
  {
    _id: new ObjectId("63bc46c5ac288035cc2643cb"),
    name: 'Jane',
    age: 32
  },
  {
    _id: new ObjectId("63bc46c5ac288035cc2643cc"),
    name: 'Tom',
    age: 11
  }
]
findOneResult {
  lastErrorObject: { n: 1, updatedExisting: true },
  value: {
    _id: new ObjectId("63bc46c5ac288035cc2643ca"),
    name: 'John',
    age: 66
  },
  ok: 1
}
```

2 문서 삭제 메서드 사용하기

- 컬렉션의 `deleteOne`과 `deleteMany` 메서드의 타입 선언문으로 두 메서드 모두 프로미스 객체를 반환하고 있음

```
deleteOne(filter: Filter<TSchema>): Promise<DeleteResult>;  
deleteMany(filter: Filter<TSchema>): Promise<DeleteResult>;
```

PS D: 본인 작업 경로 > ts-node src/test/deleteTest.ts
deleteOneResult { acknowledged: true, deletedCount: 1 }
deleteManyResult { acknowledged: true, deletedCount: 1 }
deleteAllResult { acknowledged: true, deletedCount: 1 }
userDocuments []

```
src > test > TS deleteTest.ts > ...  
1  import * as M from '../mongodb'  
2  
3  const connectCB = async (db: M.MongoDB) => {  
4    try {  
5      const user = db.collection('user')  
6  
7      const deleteOneResult = await user.deleteOne({name: {$regex: /^].*$/}})  
8      console.log('deleteOneResult', deleteOneResult)  
9  
10     const deleteManyResult = await user.deleteMany({name: {$regex: /^].*$/}})  
11     console.log('deleteManyResult', deleteManyResult)  
12  
13     const deleteAllResult = await user.deleteMany({})  
14     console.log('deleteAllResult', deleteAllResult)  
15  
16     const userDocuments = await user.find({}).toArray()  
17     console.log('userDocuments', userDocuments)  
18   } catch (e) {  
19     if (e instanceof Error) console.log(e.message)  
20   }  
21 }  
22 const deleteTest = () => {  
23   M.connectAndUseDB(connectCB, 'ch07')  
24 }  
25  
26 deleteTest()
```


몽고DB와 API 서버

- 1 몽고DB 이해하기
- 2 프로그래밍으로 몽고DB 사용하기
- 3 익스프레스 프레임워크로 API 서버 만들기
- 4 JSON 웹 토큰으로 회원 인증 기능 구현하기

3 TCP/IP 프로토콜 알아보기

- TCP/IP 프로토콜은 IP 프로토콜 기반에서 데이터 전송 방식을 제어하는 TCP 프로토콜을 함께 호칭하는 용어
- TCP/IP 프로토콜을 사용하는 시스템은 항상 데이터를 요청하는 클라이언트 프로그램과 데이터를 제공하는 서버 프로그램으로 구성
- 서버 프로그램은 항상 클라이언트의 데이터 요청이 있는지 알기 위해 특정 포트를 감시하고 있어야 하는데, 이 과정을 리스(listen)이라고 함
- 포트의 목적이 이처럼 클라이언트의 요청을 리스 하는 데 있으므로 어떤 TCP/IP 서버가 특정 포트를 리스 하고 있을 때 다른 TCP/IP 서버는 이 포트에 접근하지 못함

-
- TCP/IP 연결이 되면 클라이언트와 서버 모두 소켓(socket)이라는 토큰을 얻음
 - 연결된 클라이언트와 서버는 이 소켓을 통해 양방향(bidirectional)으로 데이터를 주고받음
 - 이 방식은 여러 클라이언트가 한 대의 서버에 접속하는 상황에서 서버가 각각의 클라이언트를 구분할 수 있게 해줌
 - 클라이언트는 서로 중복되지 않는 소켓 번호를 가지므로 서버 입장에서 소켓 값을 가지고 각각의 클라이언트를 쉽게 구분할 수 있음
 - HTTP 프로토콜은 TCP/IP 프로토콜 위에서 동작하는 앱 수준 프로토콜
 - HTTP 서버, 즉 웹 서버는 TCP/IP 프로토콜을 사용하므로 항상 웹 브라우저와 같은 클라이언트의 요청에 응답할 수 있도록 특정 포트를 리스 하고 있어야 함

3 Node.js 웹 서버 만들기

- Node.js는 웹 브라우저의 자바스크립트 엔진 부분만 떼어 내어, C/C++언어로 HTTP 프로토콜을 구현하여 독립적인 프로그램 형태로 동작하는 웹 서버 기능을 가진 자바스크립트 엔진을 구현한 것
- Node.js는 http란 이름의 패키지를 기본으로 제공하며 이 패키지는 createServer라는 함수를 제공하여 웹 서버 객체를 만들 수 있게 함

```
function createServer(requestListener?: RequestListener): Server;
```

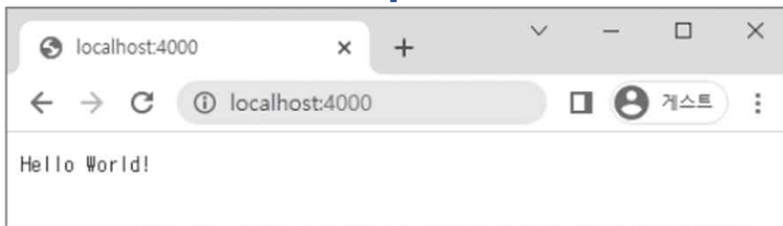
- RequestListener 타입 선언문으로 IncomingMessage와 ServerResponse 타입 매개변수 2개를 입력 받는 함수 타입

```
type RequestListener = (req: IncomingMessage, res: ServerResponse) => void;
```

■ HTTP 웹 서버 만들기

```
터미널
> ts-node src
connect http://localhost:4000
```

↑
접속



```
src > TS index.ts > ...
1 import {createServer} from 'http'
2 import {getPublicDirPath} from './config'
3 import {makeDir} from './utils'
4 import {createExpressApp} from './express'
5 import type {MongoDB} from './mongodb'
6 import {connectAndUseDB} from './mongodb'
7
8 makeDir(getPublicDirPath())
9
10 const connectCallback = (db: MongoDB) => {
11   const hostname = 'localhost',
12     port = 4000
13
14   createServer(createExpressApp(db)).listen(port, () => {
15     console.log(`connect http://${hostname}:${port}`)
16   })
17 }
18 connectAndUseDB(connectCallback, 'ch07')
```

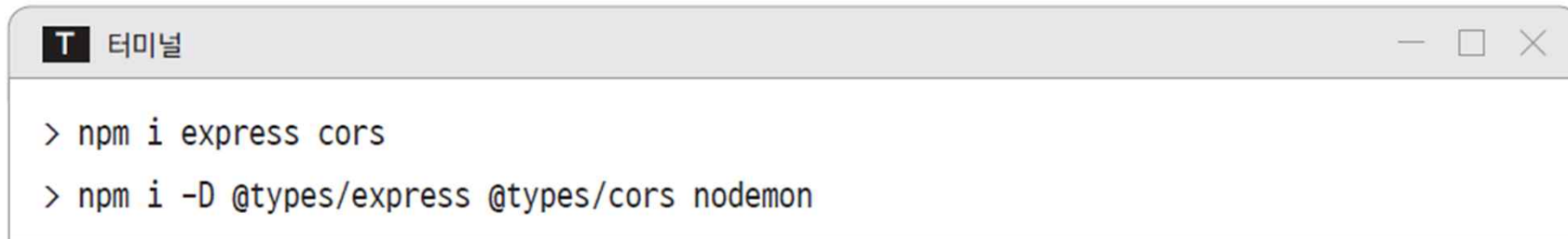
3 REST 방식 API 서버

- 웹 서버는 원래 웹 브라우저와 같은 HTTP 클라이언트에게 HTML 형식의 데이터를 전송해 주는 목적으로 설계되었지만, 점차 HTML이 아닌 JSON 형식의 데이터를 전송해 주는 방식으로 진화되었음.
- 이 두 방식을 구분하기 위해 HTML 형식 데이터를 전송하는 서버를 웹 서버, JSON 형식 데이터를 전송하는 서버를 API 서버라고 함
- REST(representational state transfer)라는 용어는 HTTP 프로토콜의 주요 저자 중 한 사람인 로이 필딩Roy Fielding의 2000년 박사학위 논문에서 처음 소개됨
- REST는 req.method의 설정 값을 다르게 하여 API 서버 쪽에서 DB의 CRUD 작업을 쉽게 구분할 수 있게 하는 용도로 사용

-
- REST API의 기본 원리는 'http://localhost:4000/user', 'http://localhost:4000/list'처럼 경로에 자신이 원하는 '/user'나 '/list' 같은 자원을 명시하고, 이 자원에 새로운 데이터를 생성하고 싶으면 POST 메서드를, 단순히 자원을 검색하고 싶으면 GET 메서드를 사용하는 것
 - 이러한 방식은 특정 자원을 항상 일정하게 사용하므로 일관된 방식으로 API를 설계할 수 있도록 함

3 익스프레스 설치하기

- 익스프레스 프레임워크(express framework)는 Node.js 환경에서 사실상 표준 웹 프레임워크
- 익스프레스를 사용하면 웹 서버는 물론 REST 방식 API 서버를 쉽게 만들 수 있음
- 익스프레스를 사용하여 REST 방식 API 서버를 만들려면 express와 cors라는 패키지를 설치해야 합니다. 함
- 좀 더 편리하게 개발하고자 할 때는 nodemon 패키지도 설치(nodemon은 개발할 때만 필요하므로 -D 옵션으로 설치)



```
T 터미널
> npm i express cors
> npm i -D @types/express @types/cors nodemon
```


- REST 방식 express의 package.json 파일 내용
- nodemon은 윈도우 와 맥에서 그 사용법이 다름

```
"scripts": {  
  "start": "nodemon -e ts --exec ts-node src --watch src",  
  "start-mac": "nodemon -e ts --exec 'ts-node src' --watch src"  
},
```

```
{  
  "name": "testDBServer",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "nodemon -e ts --exec ts-node src --watch src",  
    "start-mac": "nodemon -e ts --exec 'ts-node src' --watch src"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "cors": "^2.8.5",  
    "express": "^4.18.1",  
    "mongodb": "^4.5.0"  
  },  
  "devDependencies": {  
    "@types/cors": "^2.8.12",  
    "@types/express": "^4.17.13",  
    "@types/mongodb": "^4.0.7",  
    "@types/node": "^17.0.31",  
    "nodemon": "^2.0.16",  
    "ts-node": "^10.7.0",  
    "typescript": "^4.6.4"  
  }  
}
```

3 익스프레스 REST API 서버 만들기

- 익스프레스로 웹 서버를 만들 때는 항상 다음 코드 패턴으로 app 객체를 먼저 만들어야 함

```
import express from 'express'
const app = express()
```

- app 객체는 다음 표에서 보듯 앞서 언급한 4개의 HTTP 메서드에 대응하는 4개의 메서드를 제공하며, 이 메서드들은 항상 app 객체를 다시 반환함
- app의 get 등의 메서드는 항상 app를 반환하므로,
다음처럼 메서드 체인 형태의 구현 가능

```
app
  .get(경로, (req, res) => {})
  .post(경로, (req, res) => {})
  .put(경로, (req, res) => {})
  .delete(경로, (req, res) => {})
```

표 7-7 HTTP 메서드별 app 메서드

HTTP 메서드 이름	app 메서드
POST	post
GET	get
PUT	put
DELETE	delete

3 익스프레스 미들웨어와 use 메서드

- 익스프레스 객체 `app`은 `use` 메서드를 제공하며, `use` 메서드의 매개변수로 사용되는 콜백 함수를 미들웨어(middleware)라고 함
- 익스프레스 미들웨어는 다양한 종류가 있으며, 익스프레스는 여러 가지 다양한 기능을 미들웨어를 통해 쉽게 사용할 수 있도록 함

```
app.use(미들웨어)
```

- 익스프레스 미들웨어는 다음처럼 매개변수 3개로 구성된 함수 형태로 구현. 3번째 매개변수 `next`는 함수로서 이 함수를 호출하면 모든 것이 정상으로 동작. 반면에 `next`를 호출하지 않으면 이 미들웨어 아래쪽에 메서드 체인으로 연결된 메서드들이 호출되지 않음

```
const middleware = (req, res, next) => {}
```

3 express.static 미들웨어

- expresss.static 미들웨어 - 익스프레스 객체가 public 디렉터리에 있는 .html, .css, .js, .png와 같은 파일을 웹 브라우저에 응답할 수 있게 하는 정적 파일 서버로 동작할 수 있게 하는 미들웨어

```
app.use(express.static('public'))
```

3 express.json 미들웨어

- express.json 미들웨어 - 웹 브라우저에서 동작하는 리액트 코드는 다음 코드 형태로 HTTP POST 메서드를 통해 body 부분의 데이터를 서버로 전송 가능. 익스프레스는 이렇게 전달받은 데이터를 req.body 형태로 얻을 수 있도록 express.json 미들웨어 제공

```
fetch(url, {  
  method: 'POST',  
  headers: {'Content-Type': 'application/json'},  
  body: JSON.stringify(data)  
})
```

cors 미들웨어

- cors는 자바스크립트 코드에서 HTTP POST 메서드로 데이터를 보낼 때 프리플라이트(preflight) 요청과 응답 통신 기능을 추가하여, 악의적인 목적의 데이터를 POST나 PUT 메서드로 서버 쪽에 보내지 못하게 하는 기술
- 웹 브라우저 쪽에서 동작하는 리엑트 코드가 다음 코드 형태로 HTTP POST 메서드를 사용하여 데이터를 서버로 전송할 때, 서버는 먼저 프리플라이트 응답을 해줘야 함

```
fetch(url, {  
  method: 'POST',  
  mode: 'cors',  
  cache: 'no-cache',  
  credentials: 'same-origin',  
})
```

■ 미들웨어 적용 코드

```
src > express > TS index.ts > ...
1  import express from 'express'
2  import cors from 'cors'
3  import {setupRouters} from './setupRouters'
4
5  export const createExpressApp = (...args: any[]) => {
6    const app = express()
7
8    app
9      .use((req, res, next) => {
10        console.log(`url=${req.url}, method=${req.method}`)
11        next()
12      })
13      .use(express.static('public'))
14      .use(express.json())
15      .use(cors())
16      .get('/', (req, res) => {
17        res.json({message: 'Hello express World!'})
18      })
19
20    return setupRouters(app, ...args)
21  }
```

3 익스프레스 라우터

- **express** 패키지는 다음처럼 Router라는 함수를 제공

```
Router 함수 импорт  
  
import {Router} from 'express'
```

- 익스프레스 Router 함수를 호출하여 얻은 객체를 라우터라고 함

```
const router = Router()
```

- 라우터 메서드는 다음처럼 어떤 경로에 웹 브라우저가 접속하면, 설정된 라우터 콜백 함수를 호출

```
app.use(경로, 라우터)
```


-
- 웹 브라우저가 '/test' 경로에 접속하면 R.testRouter 콜백 함수가 호출되도록 설정하고 있음

```
src > express > TS setupRouters.ts > ...  
1  import {Express} from 'express'  
2  import * as R from '../routers'  
3  
4  export const setupRouters = (app: Express, ...args: any[]): Express => {  
5    |   return app.use('/test', R.testRouter(...args))  
6  }
```

3 몽고DB 연결하기

- connectAndUseDB라는 함수를 만들어 몽고DB와 연결한 적이 있는데, 다음 src/index.ts 파일은 이 함수를 사용하여 몽고DB의 db 객체를 createExpressApp 함수 호출 때 매개변수로 넘겨 줌

```
src > TS index.ts > ...
1 import {createServer} from 'http'
2 import {getPublicDirPath} from './config'
3 import {makeDir} from './utils'
4 import {createExpressApp} from './express'
5 import type {MongoDB} from './mongodb'
6 import {connectAndUseDB} from './mongodb'
7
8 makeDir(getPublicDirPath())
9
10 const connectCallback = (db: MongoDB) => {
11   const hostname = 'localhost',
12     port = 4000
13
14   createServer(createExpressApp(db)).listen(port, () =>
15     console.log(`connect http://${hostname}:${port}`)
16   )
17 }
18 connectAndUseDB(connectCallback, 'ch07')
```

3 몽고DB 기능 추가하기

- testRouter의 구현 일부 내용
- args에서 몽고DB 객체를 얻어, DB 작업을 수행하고 있음

```
export const testRouter = (...args: any[]) => {  
  const db: MongoDB = args[0]  
  const test = db.collection('test')  
  
  return router  
    .get('/', async (req, res) => {  
      try {  
        const findResult = await test.find({}).toArray()  
        res.json({ok: true, body: findResult})  
      } catch (e) {  
        if (e instanceof Error) res.json({ok: false, errorMessage: e.message})  
      }  
    })  
  })
```

3 클라이언트 만들기

- 여기서 부터 클라이언트

3 fetch 함수로 JSON 형식 데이터 가져오기

- 웹 브라우저에서 동작하는 자바스크립트 코드가 REST API 서버에 접속하려면 자바스크립트 엔진이 제공하는 fetch 함수 사용이 필요
- fetch는 HTTP 프로토콜의 GET, POST, PUT, DELETE와 같은 메서드를 프로그래밍으로 사용할 수 있게 해줌
- 다음은 fetch API의 타입 선언문으로, fetch는 blob, json, text와 같은 메서드가 있는 Response 타입 객체를 Promise 방식으로 얻을 수 있게함

```
function fetch(input: RequestInfo, init?: RequestInit): Promise<Response>
interface Response {
  // 이미지 등 Blob 타입 데이터를 텍스트나 바이너리 형태로 수신할 때 사용
  blob(): Promise<Blob>;
  json(): Promise<any>;      // JSON 형식 데이터를 수신할 때 사용
  text(): Promise<string>;   // HTML 형식 데이터를 수신할 때 사용
}
```

-
- fetch의 첫 번째 매개변수 input의 타입 RequestInfo는 다음과 같은 타입으로, 보통은 `http://localhost:4000/test` 등의 문자열 사용

```
type RequestInfo = Request | string
```

- fetch는 Promise 타입 객체를 반환하므로 fetch로 실제 데이터를 얻으려면 Promise 객체의 then 메서드를 반드시 호출해야 함
- 또한 서버가 동작하지 않는 등의 이유로 통신 장애가 날 수 있으므로 catch 메서드로 장애의 구체적인 내용을 텍스트 형태로 얻어야 함
- 다음은 HTTP GET 메서드를 사용하여 API 서버에서 JSON 형식 데이터를 가져오는 코드로서, JSON 형식 데이터를 가져와야 하므로 Response 타입 객체의 json 메서드를 호출하고 있음

```
fetch(API_서버_URL)
  .then((res) => res.json())
  .catch((error: Error) => console.log(error.message))
```

3 서버 URL 가져오는 함수 구현하기

- 다음은 `fetch('http://localhost:4000/test')` 부분

```
fetch(getServerUrl('/test'))
```

- 위 코드는 `fetch`의 두 번째 매개변수 `init` 부분을 다음처럼 구현할 수도 있음

```
fetch(getServerUrl('/test'), {method: 'GET'})
```

- 다음은 유틸리티 함수 `get` 과 `del`의 구현 예

Do it! GET과 DELETE 메서드 호출용 함수 구현하기

• `src/server/getAndDel.ts`

```
import {getServerUrl} from './getServerUrl'
```

```
export const get = (path: string) => fetch(getServerUrl(path))
```

```
export const del = (path: string) => fetch(getServerUrl(path), {method: 'DELETE'})
```

3 HTTP GET과 DELETE 메서드 호출용 함수 구현하기

- HTTP POST와 PUT 메서드는 method 설정 값만 다를 뿐 사용법은 동일. 다음은 POST나 PUT 메서드를 사용하여 data 변수에 담긴 데이터를 서버에 보내는 기본 코드

```
fetch(getServerUrl(path), {  
  method: 'POST' 혹은 'PUT',  
  headers: {'Content-Type': 'application/json'},  
  body: JSON.stringify(data)  
})
```

- 그런데 POST와 PUT 메서드는 cors 문제가 있으므로 다음처럼 mode와 cache, credentials 속성 값을 추가해 줘야 함

```
fetch(getServerUrl(path), {  
  method: 'POST' 혹은 'PUT',  
  headers: {'Content-Type': 'application/json'},  
  body: JSON.stringify(data),  
  mode: 'cors',  
  cache: 'no-cache',  
  credentials: 'same-origin'  
})
```


3 몽고DB에서 데이터 가져오기

- 다음은 get 함수 사용 예

```
src > routes > RestTest > TS GetTest.tsx > ...
1  import {useState, useCallbakck} from 'react'
2  import {get} from '../server'
3
4  export default function GetTest() {
5    const [data, setData] = useState<object>({})
6    const [errorMessage, setErrorMessage] = useState<string | null>(null)
7
8    const getAllTest = useCallbakck(() => {
9      get('/test')
10     .then(res => res.json())
11     .then(data => setData(data))
12     .catch(error => setErrorMessage(error.message))
13   }, [])
14   const getTest = useCallbakck(() => {
15     get('/test/1234')
16     .then(res => res.json())
17     .then(data => setData(data))
18     .catch(error => setErrorMessage(error.message))
19   }, [])
```

3 컬렉션에 데이터 저장하기

- post 함수로 데이터를 서버에 저장하는 예

```
src > routes > RestTest > TS PostTest.tsx > ...
1  import {useState, useCallback} from 'react'
2  import {post} from '../server'
3  import * as D from '../data'
4
5  export default function PostTest() {
6    const [data, setData] = useState<object>({})
7    const [errorMessage, setErrorMessage] = useState<string | null>(null)
8    const postTest = useCallback(() => {
9      post('/test', D.makeRandomCard())
10     .then(res => res.json())
11     .then(data => setData(data))
12     .catch(error => setErrorMessage(error.message))
13   }, [])
```

3 데이터 지우기

- 오른쪽 코드는 del 함수를 사용하여 몽고DB test 컬렉션에 저장된 문서들 중 id값이 '1234'인 문서를 지우는 기능을 구현

```
src > routes > RestTest > TS DeleteTest.tsx > ...
1  import {useState, useCallback} from 'react'
2  import {del} from '../server'
3
4  export default function DelTest() {
5    const [data, setData] = useState<object>({})
6    const [errorMessage, setErrorMessage] = useState<string | null>(null)
7    const deleteTest = useCallback(() => {
8      del('/test/1234')
9      .then(res => res.json())
10     .then(data => setData(data))
11     .catch(error => setErrorMessage(error.message))
12   }, [])
```