생성(Create 또는 Insert)

CRUD의 첫 번째인 Create 또는 Insert기능에 대해서 알아보겠습니다. 일단 무언가를 데이터베이스에 넣어야 조회도 하고, 수정도 하고, 삭제도 하겠죠? 그래서 Create를 먼저 알아봅니다.

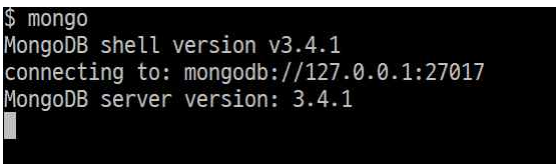
- CRUD

참고로 데이터베이스에는 CRUD라고 불리는 기능이 있습니다. Create, Read, Update, Delete의 약자로 데이터베이스의 자료를 생성, 조회, 수정, 삭제하는 기능입니다. 오늘은 그 중에서 Create에 대한 것을 하는 겁니다.

- Create

먼저 mongod(mongoDB engine)이 실행되고 있어야 합니다. 혹시 자동으로 실행되지 않고 있다면 cmd창에서 mongodb —dbpath C:\mongodb\data를 실행해서 db engine을 실행해 주시기 바랍니다.

cmd창을 하나 더 열고 C:\mongodb\bin에 가서 mongo를 칩니다. 다음과 같이 뜨면 connecting to: test 밑에 쿼리를 입력할 수 있습니다. test는 DB의 이름입니다. 다른 DB를 사용하고 싶으면 use [DB 이름]하세요.



데이터를 입력하기에 앞서 컬렉션부터 만들어봅시다. 컬렉션이란 SQL에서는 테이블과 비슷한 개념입니다. 데이터를 구분짓는 기준이라고 보셔도 됩니다. 예를 들면, 제 홈페이지에는 사용자 컬렉션, 포스트 컬렉션, 댓글 컬렉션, 카테고리 컬렉션이 있습니다. 사용자 데이터 따로, 포스트 데이터, 댓글 데이터 따로 모아둔 겁니다.

몬스터 컬렉션을 한 번 만들어볼까요? 만약 쿼리 입력이 안 된다면, DB를 재시작하세요. mongodb --dpath 했던 cmd를 껐다가 다시 켜고 연결하면 됩니다. 컬렉션을 만드는 법은 간단합니다. 그냥 컬렉션 안에 데이터를 넣으면 컬렉션은 자동으로 생깁니다. 컬렉션의 이름은 항상 복수형으로 합니다.

슬라임(Slime)이라는 몬스터를 만들 건데요. 필드는 이름, hp(체력), xp(경험치), att(공격력)입니다. 필드는 SQL에서 column과 같습니다. 속성명을 뜻합니다. SQL에서는 column이 고정되어 있어 column에서 정하지 않은 데이터를 넣으면 오류가 발생하지만 몽고DB는 그런 게 없습니다. 애초에 필드를 정의하는 부분도 없죠. 그냥 넣고 싶은대로 자유롭게 넣으면 됩니다.

```
db.monsters.save({ name: 'Slime', hp: 25, xp: 10, att: 10 }); // WriteResult({ nInserted: 1 })
```

이렇게 입력하면 WriteResult 객체가 반환됩니다. nInserted: 1이라고 나와있는데요. 1개의 다큐먼트가 입력되었다는 뜻입니다. 다큐먼트(document)가 바로 데이터의 단위입니다. SQL에서는 row라고

불리죠. 하나의 문서를 넣은 겁니다. 진짜 제대로 들어갔는지는 아직 Read 기능을 배우지 않아서 모르겠지만 믿어봅시다.

컬렉션은 다큐먼트를 만들면 자동으로 만들어진다고 했는데, 한 번 확인해보죠. show collections 하면 컬렉션 목록이 나옵니다. 아래를 보면 monsters와 system.indexes가 있네요. monsters는 우리가 만든 거고, systems.indexes는 원래 있던 겁니다. 나중에 index 기능을 배울 때 사용되는 컬렉션입니다. 어쨌든 잘 만들어졌군요.

```
> db
test
> db.monsters.save({ name: 'Slime', hp: 25, xp: 10, att: 10 });
WriteResult({ "nInserted" : 1 })
> show collections
monsters
system.indexes
>
```

save가 바로 몽고DB에서 데이터를 만드는 메소드입니다. save 말고도 insert, insertOne, insertMany가 있습니다. insertOne과 insertMany는 몽고DB 3.2 버전에서 insert를 대체하기 위해서 만들어졌습니다. 각각 하나의 새로운 다큐먼트를 만들거나, 여러 개의 다큐먼트를 만듭니다. 그러나 여전히 save와 insert가 자주 쓰입니다. save와 insert는 살짝 다릅니다. 그 내용은 Update 기능 시간에 알려드리겠습니다.

데이터가 하나만 있으면 심심하니까 두 개 정도 더 만들어봅시다. insert 메소드를 이용해 한 번에 여러 개를 추가할 수 있습니다.

```
db.monsters.insert([{ name: 'Skeleton', hp: 50, xp: 20, att: 20 }, { name: 'Demon', hp: 500, xp: 200, att: 200 }]);
```

Demon은 보스급 몬스터입니다. 좀 많이 세죠?

조회(Read 또는 Find)

CRUD에서 R을 담당하는 Read 기능을 알아보겠습니다. 앞에서 몬스터들을 넣었죠? 잘 있나 볼까요?

- find

```
db.monsters.find({ }); // { 결과들 }
db.monsters.find({ name: 'Slime' }); // { 슬라임 }
```

find({ }) 또는 find()를 하면 컬렉션 내의 모든 다큐먼트들을 선택합니다. find({ name: 'Slime' })을 하면 다큐먼트 중에 name 필드가 Slime인 다큐먼트만 선택합니다.

결과는 다음과 같이 나옵니다.

```
> db.monsters.find({ $or: [{ name: 'Slime' }, { hp: 50 }] });
{ "_id" : ObjectId("579b279c87e095640ff6a6e7"), "name" : "Slime", "hp" : 25, "xp" : 10, "att" : 10 }
{ "_id" : ObjectId("579b2ae687e095640ff6a6e8"), "name" : "Skeleton", "hp" : 50, "xp" : 20, "att" : 20 }
> db.monsters.find({ hp: { $lt: 100 } });
{ "_id" : ObjectId("579b279c87e095640ff6a6e7"), "name" : "Slime", "hp" : 25, "xp" : 10, "att" : 10 }
{ "_id" : ObjectId("579b2ae687e095640ff6a6e8"), "name" : "Skeleton", "hp" : 50, "xp" : 20, "att" : 20 }
```

일단 결과는 제대로 나오는 것 같습니다. 그런데 처음 보는 게 있습니다. 바로 **_id 필드의 ObjectId(...)** 입니다. 분명 insert할 때는 넣은 적이 없는 것 같은 데 반환된 결과에는 있네요. 이게 몽고DB에서 자동으로 넣어주는 고유 값입니다. 절대로 겹치지 않습니다. 이 세상에 절대가 어딨어! 하시는 분이 있을 수도 있겠네요. 그런데 첫 4바이트가 insert된 시간을 나타내기 때문에 겹치지 않습니다. 시간이 같을 수는 없겠죠?

find 메소드 안에 { } 부분에 구체적인 쿼리를 작성할 수 있습니다. 몽고DB는 다양한 옵션들을 제공합니다.

모두 다 일치해야 하는 경우(and)는 그냥 쉼표로 구분합니다.

db.monsters.find({ name: 'Slime', hp: 25 }); // name이 Slime이고 hp가 25인 다큐먼트

한 가지만 일치해도 되는 경우(OR)는 **$or**을 사용합니다.

db.monsters.find({ $or: [{ name: 'Slime' }, { hp: 50 }] }); // name이 Slime이거나 hp가 50

숫자에 대한 비교 옵션도 있습니다. 큰 것은 $gt, 작은 것은 $lt, 크거나 같은 것은 $gte, 작거나 같은 것은 $lte 하면 됩니다.

db.monsters.find({ hp: { $lt: 100 } }); // hp가 100보다 다큐먼트

위와 같은 결과가 나왔다면 성공입니다! 쿼리의 옵션은 매우 많기 때문에 차차 알아가도록 하겠습니다.

- findOne

단 하나만 찾고 싶을 때 사용합니다. find(...)[0]과도 같습니다. find 메소드로 찾은 것 중에 첫 번째 것을 선택하는 거죠.

db.monsters.findOne({ name: 'Slime' }); // { **슬라임** }

똑같이 슬라임에 대한 정보가 뜹니다. find 메소드처럼 구체적인 쿼리를 작성할 수 있습니다.

- **객체 조회**
-

지금까지는 내부 객체를 쓰지 않았지만, 그것을 조회하고 싶은 경우를 알아봅시다. 다음과 같은 다큐

먼트가 있다고 칩시다. 다큐먼트 안에 객체나 배열이 필드의 값으로 들어가도 상관 없습니다.

```
db:zero {
  _id: ObjectId('...'),
  profile: {
    name: 'Zero',
    birth: 1994
  },
  interest: ['javascript', 'mongodb']
}
```

이제 프로필 생일이 1994인 위의 문서를 찾아봅시다.

```
db.zero.find({ 'profile.birth': 1994 });
```

객체의 내부에 접근하듯이 .(점)으로 내부 속성을 지목하면 됩니다. 대신 따옴표로 묶어줘야 에러가 발생하지 않습니다.

- **배열 조회**

이번에는 배열을 조회하는 방법입니다. 위의 다큐먼트를 그대로 사용합니다. mongodb를 요소로 갖고 있는 interest 필드를 찾으려면

```
db.zero.find({ interest: 'mongodb' });
```

하면 됩니다. 배열이라고 해서 특별히 다른 방법을 쓰거나 하진 않습니다.

- Projection

projection 이란 find와 findOne 메소드의 두 번째 인자로 넣어주는 겁니다. RDBMS에서 select를 할 때 보이고자 하는 필드를 나열하는 것과 같습니다

※ select name,hp, id from monster

```
db.monsters.findOne({ name: 'Slime' }, { name: true, hp: true, _id: false});
```

하면 { name: 'Slime', hp: 25 } 라는 결과가 나옵니다. 이전에 쿼리했던 것과 비교하면 많이 짧아졌습니다. projection은 결과로 보여줄 것만 필터링하는 겁니다. name과 hp는 true고, _id는 false로 했더니 name과 hp만 반환되었습니다. 다른 필드는 기본값이 false이지만, _id는 기본값이 true이기 때문에 false를 직접 입력해줘야 나오지 않습니다.

projection이 유용한 경우는 민감한 데이터가 있을 경우입니다. 만약 댓글을 단 회원의 정보를 가져오고 싶은데 회원 정보를 통째로 가져오면 비밀번호나 개인 정보 같은 게 모두 가져와지겠죠? 그럴 때

projection 객체를 사용해서 가져올 데이터만 걸러내는 겁니다. 제 홈페이지도 이렇게 가져올 것만 가져오고 있습니다. 보안은 안심하셔도 됩니다.

또한 한 가지 더 장점은 용량이 줄어듭니다. 게시글 리스트에서 게시글들을 모두 불러오면 용량이 어마어마합니다. 게시글 제목, 내용, 댓글까지 다 불러와지기 때문이죠. 이럴 때 제목만 불러오도록 하면 데이터를 아낄 수 있습니다.

수정(Update,Upsert 또는 Modify)

이번 에는 몽고DB CRUD 기능 중 U를 담당하는 Update 메소드들에 대해서 알아보겠습니다.

일단 수정을 하려면 수정할 대상을 선택해야 합니다. 그러고나서 이제 어떻게 수정할 지를 알려줘야겠죠.

- Update

가장 유명한 메소드로 update가 있습니다. 슬라임 몬스터를 좀 버프해볼까요? hp가 너무 낮은 거 같아서 올려봅니다.

- $set

```
db.monsters.update({ name: 'Slime' }, { $set: { hp: 30 } }); // WriteResult({ nMatched: 1, nUpserted: 0, nModified: 1 });
```

수행 결과를 반환하네요. $set을 해야 해당 필드만 바뀝니다. 만약 $set을 넣지 않고 그냥 { hp: 30 } 하면 Slime 다큐먼트가 다 지워지고 { hp: 30 } 이라는 객체로 통째로 바뀌어버립니다. 처음 하는 분들은 이 실수를 정말 많이 합니다.

hp를 너무 많이 올린 것 같아서 다시 원래대로 복구하겠습니다. 그런데 이번에는 좀 다른 방법으로 하겠습니다.

- $inc

```
db.monsters.update({ name: 'Slime' }, { $inc: { hp: -5 } }); // WriteResult({ nMatched: 1, nUpserted: 0, nModified: 1 });
```

$inc를 사용하면 숫자를 올리거나 내릴 수 있습니다. 음수를 넣으면 내리고 양수를 넣으면 올립니다. 위의 코드는 hp를 5만큼 깎은 겁니다.

- 첫 번째 인자가 수정할
- 두 번째 인자가 수정할 내용입니다.
- 세 번째 인자로 옵션을 넣을 수 있습니다.

옵션에는 크게 multi와 upsert가 있습니다. multi는 여러 개를 동시에 수정할 때 사용합니다. { multi: true } 하면 됩니다. **기본적으로는 한 다큐먼트만 수정**하지만 만약 이름이 Slime인 다큐먼트가 여러 개 있다면 그 다큐먼트들을 모두 수정하는 거죠. upsert는 아래에 따로 설명합니다.

- FindAndModify

또한 많이 쓰이는 메소드가 findAndModify입니다. update 메소드와는 달리 upsert과 remove까지 같이 수행할 수 있습니다. 인자가 옵션 객체 하나인데 여러 개의 속성을 넣어줘야 합니다. query가 대상을 찾는 법, update가 대상을 수정할 내용, new가 수정 이전의 다큐먼트를 반환할지, 수정 이후의 다큐먼트를 반환할 지 결정하는 부분입니다. { new: true }를 넣으면 수정 이후의 다큐먼트를 반환합니다.

데몬 몬스터가 너무 세니 너프해보겠습니다.

```
db.monsters.findAndModify({
        query: { name: 'Demon' },
        update: { $set: { att: 150 } },
        new: true }); // { 데몬 }
```

수정한 후에 수정된 결과를 다시 가져오고 싶다면 update 대신 findAndModify 메소드를 쓰는 게 낫겠죠?

- UpdateOne, UpdateMany, ReplaceOne

몽고DB 3.2 버전부터 update를 대체하는 세 메소드가 추가되었습니다. update 메소드와 거의 유사하지만,

- updateOne은 매칭되는 다큐먼트 중 첫 번째만 수정
- updateMany는 매칭되는 모든 다큐먼트를 수정

기존의 multi 옵션이 두 메소드로 나누어졌다고 생각하시면 됩니다.

```
db.monsters.updateOne({ name: 'Slime' }, { $set: { hp: 25 });
```

replaceOne 메소드는 다큐먼트를 통째로 다른 것으로 대체합니다. $set을 안 썼을 때 상황과 유사합니다.

- FindOneAndUpdate, FindOneAndReplace

또한 몽고DB 3.2 버전부터 findAndModify 메소드를 대체하는 새로운 두 메소드가 만들어졌습니다. findAndModify는 update, upsert, remove를 모두 담당할 수 있는데요. 이 기능을 쪼개서 하나의 역할만 전담하는 메소드입니다. 역시나 findAndModify 시리즈답게 수정 이전 또는 이후의 다큐먼트를 반

환받습니다. 대신 new 옵션이 아니라 returnNewDocument로 이름이 바뀌었습니다.

```
db.monsters.findOneAndUpdate({
        name: 'Demon' },
        { $set: { att: 150 } },
        { returnNewDocument: true }); // { 데몬 }
```

- Upsert

만약 수정할 대상이 없다면 어떡할까요? 보통의 경우는 아무것도 변하지 않고 종료됩니다. 특정한 옵션을 주어서 수정할 대상이 없는 경우 insert 동작을 수행하도록 할 수 있습니다. 이를 upsert라고 부릅니다. update + insert의 합성어입니다.

Upsert 기능을 하려면 update, updateOne, updateMany, replaceOne 메소드에 옵션으로 { upsert: true } 를 주면 됩니다. 또는 findAndModify, findOneAndUpdate, findOneAndReplace 메소드에 upsert: true를 추가할 수도 있습니다.

Insert 시간에 배웠던 save와 insert 메소드의 차이가 upsert의 개념에 있습니다. save는 upsert 메소드입니다. 만약 해당하는 다큐먼트가 없으면 새로 만듭니다. 하지만 해당하는 다큐먼트가 있으면(_id를 제공해야합니다) 수정합니다.

삭제(Delete 또는 Remove)

이번에는 CRUD 마지막 시간으로 Delete 기능에 대해 알아보겠습니다.

문서를 지우는 작업이니만큼 신중하게 해야합니다. 몽고DB는 롤백(이전으로 되돌리는 기능)을 지원하지 않습니다. 롤백하는 한 가지 방법이 있긴 합니다. 그 방법은 나중에 설명할 oplog를 사용하는 겁니다. oplog란 미리 알려드리자면, 데이터베이스에 어떤 쿼리를 했는지 기록으로 남겨두는 겁니다. 그 기록을 거꾸로 돌리면 롤백과 유사한 효과를 볼 수 있습니다. 하지만 지금은 oplog를 사용하지 않기 때문에 삭제하는 데 주의하도록 합시다.

- Remove

```
db.test.remove({ });
```

를 하면 전체가 지워집니다. 지우는 동작을 하기 전에는 항상 신중하게 판단하고 하세요. 책임은 자신에게 있습니다.

```
db.monsters.insert({ name: 'Zerp' });
```

Zero 몬스터를 만들려다 실수로 오타를 냈습니다. 이것을 삭제해 봅시다. 첫 번째 인자가 지울 다큐먼트를 선택하는 부분입니다.

```
db.monsters.remove({ name: 'Zerp' }); // WriteResult({ 'nRemoved': 1 })
```

잘 지워졌습니다. nRemoved는 지워진 다큐먼트의 수를 의미합니다.

- DeleteOne, DeleteMany

몽고DB 3.2 버전부터는 Remove 메소드를 대체하는 두 메소드가 추가되었습니다.

```
db.monsters.deleteOne({ name: 'Zerp' });
```

사용 방법은 거의 같습니다. 하지만 deleteOne은 매칭되는 첫 번째 다큐먼트만 지우고, deleteMany는 매칭되는 모든 다큐먼트를 지운다는 점에서 차이가 있습니다. remove 메소드를 세분화한 겁니다.

비교,논리 쿼리 연산자

이번에는 쿼리 연산자에 대해서 알아보겠습니다.
쿼리 연산자는 다큐먼트를 조회할 때 구체적으로 찾기 위해 조건을 입력하는 것을 도와주는 기호입니다. 몇 개는 지난 시간에 미리 보셨을 겁니다. **쿼리 연산자는 모두 $로 시작합니다.** 오늘은 비교 쿼리 연산자에 대해 알아보겠습니다.

- 비교 연산자

비교연산자를 일단 풀이해 보면 다음과 같이 정리 합니다

$ 연산자	원문	수학적 비교 연산자
$lt	Less Then	<
$le	Less Then and Equal	<=
$gt	Greate Then	>
$ge	Greate Then and Equal	>=
$eq	Equal	=
$ne	Not Equal	/=

- $gt

해당 값보다 더 큰 값을 가진 필드를 찾습니다. 숫자 뿐만 아니라 날짜와 ObjectId도 비교할 수 있습니다.

```
{ 필드: { $gt: 값 } }
```

- $lt

해당 값보다 작은 값을 가진 필드를 찾습니다

{ 필드: { $lt: 값 } }

- $gte

  해당 값보다 크거나 같은 값을 가진 필드를 찾습니다

  { 필드: { $gte: 값 } }

- $lte

  해당 값보다 작거나 같은 값을 가진 필드를 찾습니다

  { 필드: { $lte: 값 } }

- $eq

  해당 값과 일치하는 값을 가진 필드를 찾습니다. 사실 그냥 { 필드: 값 } 하는 것과 같습니다.

  { 필드: { $eq: 값 } }

- $ne

  해당 값과 일치하지 않는 값을 가진 필드를 찾습니다.

  { 필드: { $ne: 값 } }

- $in

  필드의 값이 $in 안에 들어있는 값들 중 하나인 필드를 찾습니다. 아래의 예를 보면 값1, 값2, 값3, ... 이 있는데 필드의 값이 그 중 하나면 반환하는 겁니다.

  { 필드: { $in: [값1, 값2, 값3, ...] }

- $nin

  필드의 값이 $nin 안에 값들이 아닌 필드를 찾습니다. 아래의 예를 보면 값1, 값2, 값3, ...이 있는데 필드의 값이 그 값들이 아니어야 합니다.

  { 필드: { $nin: [값1, 값2, 값3, ...] }

- **논리 연산자**
  논리연산자는 복합적인 조건을 적용할 때 사용합니다

- $or

  $or은 여러 개의 조건 중에 적어도 하나를 만족하는 다큐먼트를 찾습니다.

  ```
  { $or: [{ 조건1 }, { 조건2 }, ...] }
  ```

- $and

  $and는 여러 개의 조건을 모두 만족하는 다큐먼트를 찾습니다. 조건이 간단하면 그냥 { 필드: 값, 필드: 값 } 이렇게 $and가 없어도 되지만, 주로 다음과 같은 경우 때문에 $and가 필요합니다.

  ```
  { $and: [
    { $or: [{ 조건1 }, { 조건2 }] },
    { $or: [{ 조건3 }, { 조건4 }] }
  ] }
  ```

- $nor

  $nor은 여러 개의 조건을 모두 만족하지 않는 다큐먼트를 찾습니다. 조건1, 조건2 등등을 모두 만족하지 않아야합니다.

  ```
  { $nor: [{ 조건1 }, { 조건2 }, ...] }
  ```

- $not

  $not은 뒤의 조건을 만족하지 않는 필드를 찾습니다. $nor의 단일 버전이라고 보시면 됩니다.

  ```
  { $not: { 조건 } }
  ```

요소 쿼리

- $exists

  해당 필드가 존재해야 하는지 존재하지 않아야 하는지를 정합니다.

  ```
  { 필드: { $exists: true/false } }
  ```

- $type

해당 필드의 자료형이 일치하는 다큐먼트를 선택합니다. 선택가능한 자료형으로 double, string, object, array, binData, objectId, bool, date, null, regex, dbPointer, javascript, symbol, javascriptWithScope, int, timestamp, long, minKey, maxKey가 있습니다. 이 중에서 반절은 써본 적이 없는 자료형이네요. 혹시 쓸 일이 있을지도 모르니 알아둡시다.

```
{ 필드: { $type: 자료형 } }
```

평가 쿼리

- $mod

나머지를 구하는 쿼리입니다. 예를 들어 { $mod: [4, 0] }을 하면 0, 4, 8, 12, 등등이 선택되겠죠. 자주 쓰이는 쿼리는 아닙니다.

```
{ 필드: { $mod: [ 나눌값, 나머지 ] } }
```

- $regex, $options

정규표현식 검색을 가능하게 합니다. $regex와 $options 쿼리를 모두 사용해서 검색할 수도 있고, 그냥 일반 정규식처럼 { 필드: /패턴/옵션 } 해도 됩니다.

```
{ 필드: { $regex: 패턴, $options: 옵션 } }
```

- $text

텍스트 검색을 하는 쿼리입니다. 제약이 좀 많습니다. 대표적으로 필드에 text index가 설정되어 있어야합니다. 유용한 점은 꼭 정확한 문자가 아니더라도 유사한 문자도 찾아준다는 겁니다.

```
{ $text: { $search: 문자, $language: 언어, $caseSensitive: 대소문자구별 } }
```

- $where

자바스크립트 문법을 사용해 검색할 수 있습니다. 예를 들면 'this.credit == this.debit' 이런 식을 넣으면, credit 필드 값과 debit 필드 값이 같은 다큐먼트를 반환합니다.

```
{ $where: 자바스크립트식 }
```

배열 쿼리

여기서부터는 배열 값을 가진 필드를 조회하는 쿼리입니다

- $all

  $all 쿼리 안에 있는 모든 값을 포함하는 배열을 값으로 가진 태그를 선택합니다. 아래의 배열이 예에서 값1, 값2 등등 모든 값을 가지고 있어야합니다.

  { 필드: { $all: [값1, 값2, ...] } }

- $elemMatch

  $elemMatch는 조건이 배열 안의 요소와 일치하는 필드를 선택합니다.

  { 필드: { $elemMatch: { 조건1, 조건2, ... } } }

- $size

  $size는 말 그대로 배열의 length가 값과 일치하는 필드를 선택합니다.

  { 필드: { $size: 값 } }

공간 쿼리 연산자

몽고DB의 장점은 위치 정보를 계산하기 쉽습니다. 예를 들면 두 지점간의 거리를 구하거나, 한 지점으로부터 100m 떨어진 모든 가게를 가져온다든가 하는 계산이 용이합니다.

주의할 점은 공간 쿼리 연산자를 사용하기 위해서는 위치 정보 필드의 index를 2d나 2dsphere로 지정해야 합니다. 2d는 일반 좌표를 사용할 때 지정합니다. [경도(longitude), 위도(latitude)] 순으로 배열을 만들고, 경도는 ±180, 위도는 ±90의 범위를 가집니다.

2dsphere은 GeoJSON 좌표를 사용할 때 지정합니다.

GeoJSON 객체는 { type: 타입, coordinates: [경도, 위도] } 형식으로 표현됩니다.

타입에는 Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, GeometryCollection 등이 있습니다.Point를 제외하고는 coordinates: [[경도1, 위도1], [경도2, 위도2], ...] 이렇게 여러 점을 한 번에 표현할 수 있습니다. 여러 도형을 표현할 수 있는 게 장점입니다.

- $near, $nearSphere

  기준점으로부터 가까운 점들을 정렬하여 찾아줍니다. $nearSphere은 GeoJSON 객체에 대해서도 찾아줍니다. GeoJSON 객체를 사용하려면 $geometry 쿼리 안에 넣어주면 됩니다.

```
{ $near: [경도, 위도] }
{ $nearSphere: {
  $geometry: {
    type: 'Point',
    coordinates: [경도, 위도]
  }
} }
```

$maxDistance, $minDistance

$near이나 $nearSphere과 조합하여 기준점으로부터 일정 거리 이상(minDistance) 일정 거리 이하(maxDistance)에 해당되는 점들을 조회합니다. minDistance는 2dsphere일 경우에만 사용가능합니다. GeoJSON 객체가 아니면 미터 대신에 라디안이라는 거리 단위를 사용합니다. 라디안을 킬로미터로 바꾸려면 6378.1을 곱해야 하고, 킬로미터를 라디안으로 바꾸려면 6378.1을 나눠야 합니다. 미터로 바꾸려면 6378100을 나눠야겠죠?

```
{ $near: [경도, 위도], $maxDistance: 라디안 }
{ $nearSphere: {
  $geometry: {
    type: 'Point',
    coordinates: [경도, 위도]
  },
  $maxDistance: 미터,
  $minDistance: 미터,
} }
```

$geoWithin

아래의 $center, $centerSphere $box, $polygon 등과 혼합하여 사용합니다. 해당 도형 안의 점들을 찾아줍니다. GeoJSON 객체는 $centerSphere을 제외하고는 혼합하지 않고 사용해도 됩니다.

```
$geoWithin: {
  $geometry: {
    type: "Polygon" 또는 "MultiPolygon",
    coordinates: [ 좌표들 ]
  }
}
```

$center, $centerSphere

중심이 경도, 위도이고 반지름이 반경인 원 안의 점들을 찾아줍니다. 100m 안의 상점, 500m 안의 상점 등등을 찾을 때 자주 쓰입니다. $center는 GeoJSON 객체를 찾지 않기 때문에 (2d index만 가능) GeoJSON 객체를 사용하려면 $centerSphere 쿼리를 사용해야 합니다.

```
{ $geoWithin: { $center: [[경도, 위도], 라디안] } }
```

- $box

  정사각형 안의 점들을 찾아줍니다. $box 배열의 첫 번째 요소가 정사각형 좌하점의 경도 위도이고, 두 번째 요소가 우상점의 경도 위도입니다. 2d index만 가능하므로 GeoJSON은 찾지 않습니다. GeoJSON을 찾으려면 그냥 $geometry 쿼리에서 찾으면 됩니다.

  ```
  { $geoWithin: { $box: [[경도, 위도], [경도, 위도]] } }
  ```

- $polygon

  다각형 안의 점을 찾아줍니다. 2d index만 가능하므로 GeoJSON은 찾지 않습니다. 다각형이니까 각 꼭지점 좌표를 모두 넣어주면 됩니다.

  ```
  { $geoWithin: { $polygon: [[경도1, 위도1], [경도2, 위도2], ...] } }
  ```

- $geoIntersects

  $geoWithin과 유사하지만, 내부에 있지 않아도 겹치는 것까지 찾아줍니다.

Projection 연산자

projection 쿼리 연산자는 조회 후 내용의 일부만 가져오도록 하는 연산자입니다. 배열의 일부만 가져올 때 사용합니다. projection이기 때문에 두 번째 인자로만(투사 부분) 사용합니다.

```
db.monsters.find({ 쿼리 }, { projection });
```

- $

  배열의 몇 번째 요소를 가져올 지 결정합니다. { 'comments.$': 1 } 의 경우 comments 필드의 두 번째 요소만 가져옵니다. (0부터 시작)

  ```
  { 배열필드명.$: 숫자 }
  ```

- $elemMatch

  전 시간에도 $elemMatch가 있었는데요. 이번 시간의 $elemMatch는 좀 다릅니다. 일단 쿼리가 아니라 투사 옵션에 사용되는 겁니다. 배열 안에 객체들이 있을 때 일치하는 첫 번째 객체만 가져옵니다.

  ```
  { 배열필드: { $elemMatch: { 객체속성: 속성값 } }
  ```

- $slice

숫자만큼의 배열 요소만 앞에서부터 잘라서 가져옵니다. 만약 숫자가 음수면 뒤에서부터 자릅니다. 첫 몇 개 또는 뒤에서 몇 개를 가져올 때 편합니다.

{ 배열필드: { $slice: 숫자 } }

필드 수정 연산자

- $inc

  예전에 봤죠? 필드 값을 증가시키거나 감소시키는 연산자입니다. 양수면 증가, 음수면 감소입니다.

  { $inc: { 필드: 숫자 } }

- $mul

  필드 값에 곱하는 연산자입니다. 1보다 큰 수를 곱하면 커지고, 1보다 작은 수를 곱하면 작아집니다. 쿼리의 순서에 조심하세요. $inc는 필드 안의 속성이었다면, 이번에는 필드가 $mul 안의 속성입니다.

  { $mul: { 필드: 숫자 } }

- $rename

  필드 이름을 바꾸는 연산자입니다. 여러 필드를 동시에 교체할 수 있습니다.

  { $rename: { 필드1: 이름, 필드2: 이름, … } }

- $set

  이것도 역시 본 적이 있죠? 해당 필드 값을 다른 것으로 교체합니다.

  { $set: { 필드1: 값, 필드2: 값, … } }

- $setOnInsert

  $set과 비슷한데 upsert의 경우에만 작동합니다. 만약 upsert가 일어나지 않으면 아무 동작도 하지 않습니다.

  { $setOnInsert: { 필드1: 값, 필드2: 값, … } }

- $unset

  해당 필드를 제거합니다. 만약 배열의 요소를 $unset한 경우에는 제거하진 않고 null로 교체합니다.

```
{ $unset: { 필드1: '', 필드2: '', ... } }
```

- $min

  필드의 값이 주어진 값보다 클 경우 새 값으로 교체합니다. 만약 원래 값이 200이었고 $min의 값이 150이었다면 150으로 바뀝니다. 기존 기록을 경신하는 경우 사용됩니다.

  ```
  { $min: { 필드1: 값, 필드2: 값, ... } }
  ```

- $max

  필드의 값이 주어진 값보다 작을 경우 새 값으로 교체합니다. 만약 원래 값이 800이었고 $max의 값이 950이라면, 950으로 바뀝니다. 기존 기록을 경신하는 경우 사용됩니다.

  ```
  { $max: { 필드1: 값, 필드2: 값, ... } }
  ```

- $currentDate

  해당 필드 값을 현재 날짜로 교체합니다. 두 가지 타입의 현재 날짜가 있는데 하나는 기본적으로 쓰이는 Date이고 다른 하나는 Timestamp입니다. 기본 타입을 사용하려면 그냥 true하면 되고, timestamp 타입을 사용하려면 $type 연산자를 사용해야 합니다.

  ```
  { $currentDate: { 필드: true } }
  { $currentDate: { 필드: { $type: 'timestamp' } } }
  ```

배열 수정 연산자

- $

  Projection 연산자로 쓰이는 $가 아니라 배열을 수정할 때 쓰이는 연산자입니다. list: [1, 2, 3] 이라는 필드가 있다고 칩시다.

  ```
  db.zero.update({ list: 2 }, { 'list.$': 5 }) // list: [1, 5, 3]
  ```

  위의 쿼리를 통해 두 번째 요소를 바꿀 수 있습니다. 즉 찾은 값의 위치를 기억하는 연산자입니다.

- $addToSet

  배열필드에 해당 요소가 없으면 추가하고 있으면 아무것도 하지 않습니다. 몽고DB에서 자체적으로 배열에 해당 요소가 있는지 검사해주기 때문에 편합니다.

{ $addToSet: { 필드1: 값, 필드2: 값, ... } }

- $pop

  배열 메소드처럼 몽고DB 배열에서 맨 앞 또는 맨 뒤 요소를 꺼내는 겁니다. shift와 pop을 합쳐놓은 연산자입니다. -1 값은 shift 기능, 1 값은 pop 기능을 합니다.

  { $pop: { 필드1: ±1, 필드2: ±1, ... } }

- $pull

  배열에서 조건을 만족하는 특정한 요소를 꺼냅니다. 꺼내는 조건은 쿼리 연산자와 같습니다.

  { $pull: { 조건1, 조건2, ... } }

- $pullAll

  $pull 연산자와는 달리 $pullAll은 조건이 아니라 그냥 일치하는 값을 배열에서 꺼냅니다.

  { $pullAll: { 필드: [값1, 값2, ...] } }

- $push

  배열 필드에 값을 push합니다.

  { $push: { 필드1: 값, 필드2: 값, ... } }

  조심해야할 것은 값이 배열일 경우 한 번에 push해버립니다. 만약 원래 [1, 2]라는 배열이 있다면 [3, 4, 5]를 push할 경우 [1, 2, [3, 4, 5]]가 되어버립니다. 3, 4, 5를 따로따로 push하고 싶다면

  { $push: { 필드: { $each: 배열 } } }
  해야 합니다.

- $each

  방금 위에서도 사용되었습니다. 다른 용례로 $addToSet과 같이 사용하는 경우가 있습니다. $addToSet도 $push처럼 한 번에 배열을 집어넣기 때문에 따로따로 넣고 싶다면

  { $addToSet: { 필드: { $each: 배열 } } }
  해야 합니다.

- $slice

  투사 연산자에서 나왔던 $slice인데요. 그 때 $slice는 보여줄 때만 일부 추려서 보여준거라면, 지금의 $slice는 배열을 $push할 때 개수를 제한할 수 있습니다. 또한 반드시 $each과 함께 사용되어야 합니다.

  { $push: { 필드: { $each: 배열, $slice: 숫자 } } }
  숫자가 양수면 처음부터 숫자만큼, 음수면 마지막에서부터 숫자만큼만 저장합니다. 예를 들어 필드에 [1, 2, 3]이 있었고, $each로 [4, 5, 6]을 넣는데 $slice의 숫자가 -5면 마지막에서 5개만 저장하기 때문에 [1, 2, 3, 4, 5, 6]에서 [2, 3, 4, 5, 6]이 저장됩니다. 만약 처음에 [4, 5, 6]이 있었고 $each로 [1, 2, 3]을 넣는데 $slice가 4면 [1, 2, 3, 4]만 저장됩니다.

- $sort

  배열의 요소들을 정렬하는 연산자입니다. $sort 또한 $push를 보조하는 역할로 사용됩니다. 그리고 $each와 함께 사용되어야 합니다.

  { $push: { 필드: { $each: 배열, $sort: { 정렬기준 } } }
  정렬기준은 1이면 오름차순, -1이면 내림차순입니다. 만약 배열 안에 객체가 있다면 정렬 기준으로 { 객체필드: ±1 } 하면 됩니다.

- $position

  역시 $push를 보조하는 역할로 사용되고, $each와 함께 사용되어야 합니다. $push할 위치를 지정하는 역할을 합니다.

  { $push: { 필드: { $each: 배열, $position: 위치 } }
  만약 기존에 [1, 2, 3]이 있고, $each로 추가할 배열이 [4, 5, 6]이며 $position이 0이면 [4, 5, 6, 1, 2, 3]이 됩니다. 만약 $position이 2면, [1, 2, 4, 5, 6, 3]이 됩니다.