

**모델 사용법**

* 이 글은 [4. Model Usage](http://docs.sequelizejs.com/manual/tutorial/models-usage.html)를 번역한 글입니다.
* 아직 입문자이다보니 오역이 있을 수 있습니다. 양해 부탁드립니다.
* 매끄러운 번역을 위하여 의역을 한 경우가 있습니다. 원문의 뜻을 최대한 해치지 않도록 노력했으니 안심하셔도 됩니다.
* 영어 단어가 자연스러운 경우 원문 그대로의 영단어를 적었습니다.
* 저의 보충 설명은 인용문에 적었습니다.

**시리즈 목록**

* [***Sequelize 공식 Document - (1) Model Usage***](https://velog.io/@cadenzah/sequelize-document-1)
* [Sequelize 공식 Document - (2) Querying](https://velog.io/@cadenzah/sequelize-document-2)
* [Sequelize 공식 Document - (3) Instances](https://velog.io/@cadenzah/sequelize-document-3)
* [Sequelize 공식 Document - (4) Instances (상)](https://velog.io/@cadenzah/sequelize-document-4)

**시작하기 전에**

* 이 글은 공식 도큐먼트의 4번째 챕터입니다. 앞부분은 Sequelize의 기본적인 설정과 자료형 등에 대한 내용으로 이해에 큰 어려움이 없으므로 생략합니다.
* 원문에서는 예시 코드를 제공하지만, 코드만 제공할 뿐 CRUD를 위한 데이터셋 세팅, 실습 환경에 대한 정보가 없고, 실행 결과를 확인할 수 있는 방법도 알려주지 않습니다. 그래서 간단하게나마 실습해볼 수 있도록 세팅하는 방법을 알려드리고, 예제 코드 결과를 확인해볼 수 있는 코드를 설명 중간중간에 함께 올려드리니 필요에 따라 활용해주시면 감사드리겠습니다.

**1. NPM 프로젝트 생성**

$ mkdir sequelize-tutorial

$ cd sequelize-tutorial

$ npm init

$ npm install sequelize mysql2

**2. index.js 작성**

const Sequelize = require('sequelize');

const config = {

"username": "root", // user name

"password": "mysql", // password

"database": "sequelize\_tutorial", // schema name

"host": "127.0.0.1", // db address

"dialect": "mysql", // database type

"operatorsAliases": false

}

const sequelize = new Sequelize(

config.database, config.username, config.password, config,

);

/\*

Sequelize code here

\*/

sequelize.sync()

저는 MySQL 기반으로 실습을 진행했습니다. MySQL 이외의 환경으로 실습하고자 하는 분들은 각자 환경에 맞게 설치해주시면 됩니다. dialect 부분을 다른 DB로 변경해준 뒤, npm start를 해보시면 설치해야 하는 DB 모듈을 알려줍니다. 사용할 수 있는 다른 DB로는 'sqlite', 'postgres', 'mssql' 등이 있습니다.

Sequelize는 기본적으로 Promise 기반이기 때문에, 모든 CRUD 작업이 Promise의 형태로 시작하고 끝난다는 점을 감안하여 코드를 작성해주셔야 원하는 결과를 보실 수 있습니다. 위의 코드의 sync()가 실행되면 index.js 내에서 작성된 Sequelize 작업이 실제 DB와 동기화되면서 실행됩니다. 따라서 테이블 정의, 연결 관계 정의 등과 같은 작업은 sync() 이전에 완료해야 합니다. sync()의 결과는 Promise로, 동기화가 완료된 뒤 .then()을 통하여 원하는 CRUD 작업을 수행할 수 있게 됩니다. Connection!!!

index.js의 config 객체에 적은 database의 이름에 맞춰서 실제 DB에 동일한 이름의 데이터베이스를 생성해줘야 합니다. 그렇지 않으면 Sequelize가 동기화를 하지 못합니다.

여기까지 하셨으면 도큐먼트를 보면서 실습을 할 준비가 다 되신 겁니다.

**데이터 조회 / Finder**

Finder 메서드는 DB로부터 데이터를 쿼리하는 데에 사용됩니다. Finder 메서드는 평범한 객체를 반환하는 것이 아니고, 모델 인스턴스를 반환합니다. 이 모델 인스턴스에는 각종 멤버가 들어있어 활용할 수 있습니다.

이 문서에서는 다양한 Finder 메서드에 대하여 알아봅니다.

**find - 단일 요소를 DB에서 검색**

// ID로 검색

Project.findByPk(123).then(project => {

// project는 Project의 인스턴스로, ID가 123인 테이블 항목의 데이터를 담고 있다.

// 이러한 항목이 존재하지 않는다면, null을 반환받는다.

})

// 특정 컬럼으로 검색

Project.findOne({ where: {title: 'aProject'} }).then(project => {

// project는 Project 테이블에서 title이 'aProject'인 첫번째 항목 || 또는 null

})

Project.findOne({

where: {title: 'aProject'},

attributes: ['id', ['name', 'title']]

}).then(project => {

// project는 Project 테이블에서 title이 'aProject'인 첫번째 항목 || 또는 null

// project.title은 프로젝트의 name 값을 가진다 (SQL AS)

})

**findOrCreate - 특정 요소를 검색하거나, 존재하지 않으면 새로 생성**

findOrCreate 메서드는 DB에 특정 요소가 존재하는지 검사합니다. 만약 존재한다면 해당하는 인스턴스를 반환하고, 그렇지 않으면 새로 생성합니다.

빈 DB에 User 모델이 있고, 거기에 username과 job 컬럼이 있다고 가정합시다.

User

.findOrCreate({where: {username: 'sdepold'}, defaults: {job: 'Technical Lead JavaScript'}})

.spread((user, created) => {

console.log(user.get({

plain: true

}))

console.log(created)

/\*

findOrCreate 메서드는 검색되었거나 또는 생성된 객체를 포함한 배열, 그리고 boolean값을 반환합니다. 여기서 boolean값은, 새 객체가 생성되었을 경우 true, 그렇지 않을 경우 false입니다.

[ {

username: 'sdepold',

job: 'Technical Lead JavaScript',

id: 1,

createdAt: Fri Mar 22 2013 21: 28: 34 GMT + 0100(CET),

updatedAt: Fri Mar 22 2013 21: 28: 34 GMT + 0100(CET)

},

true ]

위의 예시에서 "spread" 메서드는 배열을 user와 created 2개 부분으로 나누어 이어지는 콜백에 인자로 전달합니다. 따라서 "user"는 반환된 배열의 0번째 인덱스에 존재하는 객체이고, "created"는 true값을 갖는 boolean 변수입니다.

\*/

})

where에 함께 전달되는 defaults 옵션은 검색 결과가 존재하지 않을 경우 새로 생성되는 요소가 갖는 기본값입니다. 위 예시의 경우, 새로 생성되는 요소의 username은 'sdepold'입니다.

이미 존재하는 요소는 findOrCreate 메서드로 검색하더라도 변경되지 않습니다.

**findAndCountAll - 복수의 요소를 검색하고, 해당하는 데이터와 그 갯수를 반환**

findAll과 count를 합친 편리한 메서드입니다. 특히 Pagination와 관련된 처리를 할 때, limit과 offset을 사용하는 동시에 해당 쿼리에 부합하는 레코드 갯수도 알아야하는 경우 유용합니다.

작업에 성공했을 경우, 콜백에서는 2개의 속성을 가진 객체를 받습니다.

* count: where 절, 연결 관계 등의 조건에 부합하는 레코드의 전체 갯수. (정수)
* rows: where 절, 연결 관계 등의 조건에 부합하는 레코드들을 limit와 offset 범위 내에서 담아낸 객체 배열

Project

.findAndCountAll({

where: {

title: {

[Op.like]: 'foo%'

}

},

offset: 10,

limit: 2

})

.then(result => {

console.log(result.count);

console.log(result.rows);

});

findAndCountAll 메서드는 include를 지원합니다. required 옵션이 true인 경우에 대하여서만 count에 반영됩니다.

프로필이 포함된 사용자 모두를 찾고 싶다고 가정해봅시다:

User.findAndCountAll({

include: [

{ model: Profile, required: true}

],

limit: 3

});

Profile 모델에 대한 include절의 옵션이 required: true를 가지므로, 위 코드는 INNER JOIN의 결과를 반환할 것이고, 오직 프로필을 가지고 있는 사용자만이 카운트될 것입니다. 여기서 required를 제거하면, 프로필 유무와 상관 없이 모든 사용자가 카운트됩니다. where문을 include절에 추가하면 자동으로 해당 모델에 대하여 required 옵션을 추가합니다.

User.findAndCountAll({

include: [

{ model: Profile, where: { active: true }}

],

limit: 3

});

required 옵션이 없는 include 절의 추가는 결국 User 기준으로 LEFT JOIN과 같은 결과를 만들어냅니다.

위에 작성된 쿼리문은 active 상태의 프로필을 가진 사용자만을 카운트합니다. include절이 where 문을 포함할 경우 required는 암묵적으로 true 값이 주어지기 때문이죠.

findAndCountAll에 인자로 전달되는 옵션 객체는 findAll과 동일합니다.

**findAll - 복수의 요소를 검색**

// 복수의 요소 검색

Project.findAll().then(projects => {

// projects는 모든 Project 모델의 항목들을 배열로 받아온다

})

// 아래와 같이 사용하는 것도 가능하다

Project.all().then(projects => {

// projects는 모든 Project 모델의 항목들을 배열로 받아온다

})

// 특성을 지정하여 검색하는 것도 가능하다 - 일부만 검색

Project.findAll({ where: { name: 'A Project' } }).then(projects => {

// projects는 특정된 이름을 가진 Project 모델의 항목을 배열로 받아온다

})

// 특정 범위 내에서 검색

Project.findAll({ where: { id: [1,2,3] } }).then(projects => {

// projects는 id 값이 1, 2 또는 3인 Projects의 항목들을 배열로 받아온다

// 내부적으로는 SQL IN을 사용하여 이루어진다

})

Project.findAll({

where: {

id: {

[Op.and]: {a: 5}, // AND (a = 5)

[Op.or]: [{a: 5}, {a: 6}], // (a = 5 OR a = 6)

[Op.gt]: 6, // id > 6

[Op.gte]: 6, // id >= 6

[Op.lt]: 10, // id < 10

[Op.lte]: 10, // id <= 10

[Op.ne]: 20, // id != 20

[Op.between]: [6, 10], // BETWEEN 6 AND 10

[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15

[Op.in]: [1, 2], // IN [1, 2]

[Op.notIn]: [1, 2], // NOT IN [1, 2]

[Op.like]: '%hat', // LIKE '%hat'

[Op.notLike]: '%hat', // NOT LIKE '%hat'

[Op.iLike]: '%hat', // ILIKE '%hat' (case insensitive) (PG only)

[Op.notILike]: '%hat', // NOT ILIKE '%hat' (PG only)

[Op.overlap]: [1, 2], // && [1, 2] (PG array overlap operator)

[Op.contains]: [1, 2], // @> [1, 2] (PG array contains operator)

[Op.contained]: [1, 2], // <@ [1, 2] (PG array contained by operator)

[Op.any]: [2,3] // ANY ARRAY[2, 3]::INTEGER (PG only)

},

status: {

[Op.not]: false // status NOT FALSE

}

}

})

조건 연산자를 위한 구체적인 범위를 전달할 때, 값이 여러 개일 경우 '배열'을 사용한다는 것에 유의하세요.  
PG 배열은 PostgreSQL만을 위한 것으로, 여기에서는 설명을 생략합니다.

**OR와 NOT을 위한 복잡한 필터 쿼리문**

중첩된 AND, OR, NOT 조건 등을 사용하여 여러 단계에 걸친 복잡한 where 쿼리문을 작성하는 것도 가능합니다. 이를 위하여 or, and, not 연산자를 사용할 수 있습니다.

Project.findOne({

where: {

name: 'a project',

[Op.or]: [

{ id: [1,2,3] },

{ id: { [Op.gt]: 10 } }

]

}

})

Project.findOne({

where: {

name: 'a project',

id: {

[Op.or]: [

[1,2,3],

{ [Op.gt]: 10 }

]

}

}

})

위의 두 코드는 동일하게 아래의 코드로 변환됩니다:

SELECT \*

FROM `Projects`

WHERE (

`Projects`.`name` = 'a project'

AND (`Projects`.`id` IN (1,2,3) OR `Projects`.`id` > 10)

)

LIMIT 1;

not의 경우:

Project.findOne({

where: {

name: 'a project',

[Op.not]: [

{ id: [1,2,3] },

{ array: { [Op.contains]: [3,4,5] } }

]

}

});

아래의 코드로 변환됩니다:

SELECT \*

FROM `Projects`

WHERE (

`Projects`.`name` = 'a project'

AND NOT (`Projects`.`id` IN (1,2,3) OR `Projects`.`array` @> ARRAY[3,4,5]::INTEGER[])

)

LIMIT 1;

**limit, offset, order, group으로 데이터 집합을 조작하기**

limit, offset, order, group을 사용하여 보다 연관성이 높은 데이터를 얻을 수 있습니다.

// 반환되는 항목의 개수 제한

Project.findAll({ limit: 10 })

// 첫 10개 항목은 반환받지 않고 넘긴다

Project.findAll({ offset: 10 })

// 첫 10개 항목은 반환받지 않고 넘긴 뒤, 2개 항목만 반환한다

Project.findAll({ offset: 10, limit: 2 })

// ORDER BY title DESC

Project.findAll({order: 'title DESC'})

// GROUP BY name

Project.findAll({group: 'name'})

위에서 마지막 2개 예제의 경우, 조건으로 전달되는 스트링은 이스케이프 되지 않고 그대로 전달됩니다. order 또는 group을 위하여 스트링을 전달할 때는 항상 이와 같이 그대로 전달됩니다. 컬럼 이름을 이스케이프하고 싶다면, 컬럼 하나만 전달할지라도 반드시 인자 목록으로 전달해야 합니다.

위의 코드는 스트링 그대로 전달되지만, 아래의 코드에서는 모두 백틱에 감싸져서 전달되고 있습니다. SQL 함수를 사용하는 경우에는 이것이 필요하겠죠?

something.findOne({

order: [

// `name` 으로 반환된다

['name'],

// `username` DESC 으로 반환된다

['username', 'DESC'],

// max(`age`) 으로 반환된다

sequelize.fn('max', sequelize.col('age')),

// max(`age`) DESC 으로 반환된다

[sequelize.fn('max', sequelize.col('age')), 'DESC'],

// otherfunction(`col1`, 12, 'lalala') DESC 으로 반환된다

[sequelize.fn('otherfunction', sequelize.col('col1'), 12, 'lalala'), 'DESC'],

// otherfunction(awesomefunction(`col`)) DESC 으로 반환된다. 이러한 중첩은 계속 일어날 수 있습니다.

[sequelize.fn('otherfunction', sequelize.fn('awesomefunction', sequelize.col('col'))), 'DESC']

]

})

order/group을 사용할 때에 배열의 요소는 다음과 같이 변환됩니다:

* String: 따옴표로 감싸집니다.
* 배열: 첫번째 요소는 따옴표로 감싸지고, 두번째부터는 첫번째 요소 뒤에 스트링 그대로 붙어서 반환됩니다.
* 객체(모델): Raw 형태의 모델이 따옴표 없이 포함됩니다. Raw 형태가 아닌 경우, 쿼리문은 실행에 실패합니다.
* Sequelize.fn는 함수, Sequelize.col은 따옴표로 감싼 컬럼 이름을 반환합니다.

Raw 형태가 무엇인지는 다음 섹션에서 설명합니다.  
위의 예시에서는 order나 group에서 객체가 사용되는 경우가 없는데, 아래의 **Eager loading된 연결 모델을 정렬하기** 부분에서 사용례가 등장합니다.

**Raw Query**

별다른 조작 없이 거대한 크기의 데이터셋을 그대로 받고 싶을 수도 있습니다. Sequelize는 사용자가 select한 각각의 레코드를 인스턴스로 만들고, 유틸리티 함수 - update, delete, get- 메서드 등 - 를 부여합니다. 레코드가 수천 개라면, 시간이 좀 걸리겠지요. 단지 데이터만 필요할 뿐, 이러한 유틸리티 메서드가 필요하지 않다면 아래와 같은 방법을 사용하면 됩니다.

// DB로부터 대형 사이즈의 데이터셋을 받아야 하는데, 각 요소를 위한 DAO(데이터 접근 객체)를 만드는 데에 시간 쓰기 싫으신가요?

// 데이터 자체만을 받아오기 위한 옵션이 있습니다.

Project.findAll({ where: { ... }, raw: true })

**count - 요소에 대한 빈도 측정**

데이터베이스 내의 항목 개수를 재는 메서드입니다.

Project.count().then(c => {

console.log("There are " + c + " projects!")

})

Project.count({ where: {'id': {[Op.gt]: 25}} }).then(c => {

console.log("There are " + c + " projects with an id greater than 25.")

})

**max - 특정 테이블 내에서 특정 컬럼에 대하여 최대값을 구하기**

특정 컬럼에 대하여 최대값을 찾는 메서드입니다.

/\*

각자의 나이를 가진 3개의 person 객체를 가정합니다:

각각은 10세, 5세, 40세입니다.

\*/

Project.max('age').then(max => {

// 40을 반환합니다

})

Project.max('age', { where: { age: { [Op.lt]: 20 } } }).then(max => {

// 10을 반환합니다

})

**min - 특정 테이블 내에서 특정 컬럼에 대하여 최소값을 구하기**

특정 컬럼에 대하여 최소값을 찾는 메서드입니다.

/\*

각자의 나이를 가진 3개의 person 객체를 가정합니다:

각각은 10세, 5세, 40세입니다.

\*/

Project.min('age').then(max => {

// 5을 반환합니다

})

Project.min('age', { where: { age: { [Op.gt]: 5 } } }).then(min => {

// 10을 반환합니다

})

**sum - 특정 컬럼에 대하여 각 레코드들의 총합을 구하기**

테이블의 특정 컬럼에 대한 총합을 계산하는 메서드입니다.

/\*

각자의 나이를 가진 3개의 person 객체를 가정합니다:

각각은 10세, 5세, 40세입니다.

\*/

Project.sum('age').then(max => {

// 55을 반환합니다

})

Project.sum('age', { where: { age: { [Op.gt]: 5 } } }).then(min => {

// 50을 반환합니다

})

**Eager loading**

쿼리문을 통하여 어떤 데이터를 불러올 때, 해당 데이터와 연결된 다른 모델의 데이터를 불러와야할 때도 있습니다. 이것을 **Eager loading** 이라고 부릅니다. find 또는 findAll 을 호출할 때에 include 특성을 사용하면 됩니다. 이번 예제를 위하여 다음과 같이 설정합니다:

const User = sequelize.define('user', { name: Sequelize.STRING })

const Task = sequelize.define('task', { name: Sequelize.STRING })

const Tool = sequelize.define('tool', { name: Sequelize.STRING })

Task.belongsTo(User)

User.hasMany(Task)

User.hasMany(Tool, { as: 'Instruments' })

sequelize.sync().then(() => {

// 여기서부터 코드를 작성합니다

})

아래에서는 3개 테이블에 각각 데이터가 들어있다고 가정하고 설명하고 있습니다. 직접 데이터를 넣어줄 수도 있겠지만, 저는 Sequelize 문법을 사용해서 넣어봤습니다.

sequelize.sync()

.then(() => { Task.create({ name: 'A Task' })})

.then(() => { Tool.create({ name: 'Toothpick' })})

.then(() => {

return User.create({ name: 'John Doe' })})

.then( user => {

return Task.findOne({ name: 'A Task' })

.then( task => {

user.addTasks(task)

return user })})

.then( user => { Tool.findOne({ name: 'Toothpick' })

.then((tool) => {

user.addInstruments(tool)

return user})})

User와 연결된 모든 Task를 가져옵니다:

Task.findAll({ include: [ User ] }).then(tasks => {

console.log(JSON.stringify(tasks))

/\*

[{

"name": "A Task",

"id": 1,

"createdAt": "2013-03-20T20:31:40.000Z",

"updatedAt": "2013-03-20T20:31:40.000Z",

"userId": 1,

"user": {

"name": "John Doe",

"id": 1,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z"

}

}]

\*/

})

여기서 반환된 인스턴스(tasks)에 포함된 user 속성은 접근자(accessor)라고 하는데, 접근자가 단수(singular)인 점에 유의하세요. 형성된 연걸 관계가 **일대다** 이기 때문에 그렇습니다.

여기서 '일'은 User, '다'은 Task입니다. 아래 예시에서도 그대로 확인합니다.

다음으로는 다대일 관계를 갖는 데이터를 가져옵니다:

User.findAll({ include: [ Task ] }).then(users => {

console.log(JSON.stringify(users))

/\*

[{

"name": "John Doe",

"id": 1,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z",

"tasks": [{

"name": "A Task",

"id": 1,

"createdAt": "2013-03-20T20:31:40.000Z",

"updatedAt": "2013-03-20T20:31:40.000Z",

"userId": 1

}]

}]

\*/

})

여기서 users에 포함된 tasks는 복수(plural)입니다. 형성된 관계가 **다대일** 이기 떄문입니다.

연결 관계에 as 옵션을 사용하여 별칭(alias)이 존재한다면, 모델을 include 할 때에 이를 반드시 명시해야 합니다. 아래의 예시를 보면서, User의 Tool이 Instruments라는 별칭으로 불러지는지 확인하세요. 코드가 제대로 실행되려면, 불러오고자 하는 모델과 별칭을 제대로 지정해야 합니다.

User.findAll({ include: [{ model: Tool, as: 'Instruments' }] }).then(users => {

console.log(JSON.stringify(users))

/\*

[{

"name": "John Doe",

"id": 1,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z",

"Instruments": [{ // 연결된 모델 이름이 달라졌다

"name": "Toothpick",

"id": 1,

"createdAt": null,

"updatedAt": null,

"userId": 1

}]

}]

\*/

})

include 옵션에 **연결 관계를 가리키는 별칭** 을 인자로 포함시키는 식으로 사용할 수도 있습니다.

User.findAll({ include: ['Instruments'] }).then(users => {

console.log(JSON.stringify(users))

// User와 Tool 간에 형성된 연결 관계에서 연결된 모델에 대한 별칭이 쓰였겠죠?

/\*

[{

"name": "John Doe",

"id": 1,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z",

"Instruments": [{

"name": "Toothpick",

"id": 1,

"createdAt": null,

"updatedAt": null,

"userId": 1

}]

}]

\*/

})

User.findAll({ include: [{ association: 'Instruments' }] }).then(users => {

console.log(JSON.stringify(users))

// association 옵션으로 연결된 모델에 대한 별칭을 전달하는 것도 가능합니다

/\*

[{

"name": "John Doe",

"id": 1,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z",

"Instruments": [{

"name": "Toothpick",

"id": 1,

"createdAt": null,

"updatedAt": null,

"userId": 1

}]

}]

\*/

})

Eager loading을 할 때에 where 문을 사용하여 연결 모델을 필터링하는 것도 가능합니다. 아래의 예시에서는 User의 항목들을 모두 반환하되, Tool 모델에서 where 문을 만족하는 항목이 연결된 경우만 반환합니다:

User.findAll({

include: [{

model: Tool,

as: 'Instruments',

where: { name: { [Op.like]: '%ooth%' } }

}]

}).then(users => {

console.log(JSON.stringify(users))

/\*

[{

"name": "John Doe",

"id": 1,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z",

"Instruments": [{

"name": "Toothpick",

"id": 1,

"createdAt": null,

"updatedAt": null,

"userId": 1

}]

}],

[{

"name": "John Smith",

"id": 2,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z",

"Instruments": [{

"name": "Toothpick",

"id": 1,

"createdAt": null,

"updatedAt": null,

"userId": 1

}]

}],

\*/

})

Eager loading으로 불러온 모델은 include.where을 통하여 1차적으로 필터링되고, include.required는 암묵적으로 true로 설정됩니다. 그말인즉슨 연결된 모델에서 조건에 맞는 항목들을 가져올 때에 INNER JOIN이 이루어집니다.

**Eager loading 된 모델로 상위 단계의 where 사용하기**

SQL에서 JOIN을 할 때에는 ON을 사용하여 연결된 테이블에 대한 필터링을 진행합니다.

연결 모델을 필터링하는 데에 사용된 조건(ON)을 상위 WHERE로 옮기려면, '$nested.column$'문법을 사용합니다. Fk 또는 PK에 대한 조건 명시

User.findAll({

where: {

'$Instruments.name$': { [Op.iLike]: '%ooth%' }

},

include: [{

model: Tool,

as: 'Instruments'

}]

}).then(users => {

console.log(JSON.stringify(users));

/\*

[{

"name": "John Doe",

"id": 1,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z",

"Instruments": [{

"name": "Toothpick",

"id": 1,

"createdAt": null,

"updatedAt": null,

"userId": 1

}]

}],

[{

"name": "John Smith",

"id": 2,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z",

"Instruments": [{

"name": "Toothpick",

"id": 1,

"createdAt": null,

"updatedAt": null,

"userId": 1

}]

}],

\*/

})

**전부 include하기**

연결된 모델의 모든 컬럼을 포함한 결과를 얻으려면, include 옵션에 all: true 만 전달하면 됩니다.

User.findAll({ include: [{ all: true }]})

.then(users => {

console.log(JSON.stringify(users))

});

/\*

[{

"id":1,

"name":"John Doe",

"createdAt":"2019-01-05T05:55:17.000Z",

"updatedAt":"2019-01-05T05:55:17.000Z",

"tasks":

[{"id":1,

"name":"A Task",

"createdAt":"2019-01-05T05:55:17.000Z",

"updatedAt":"2019-01-05T05:55:17.000Z",

"userId":1}],

"Instruments":

[{"id":1,

"name":"Toothpick",

"createdAt":"2019-01-05T05:55:17.000Z",

"updatedAt":"2019-01-05T05:55:17.000Z",

"userId":1}]

}]

\*/

**Soft 삭제된 레코드 포함하기**

연결된 모델에서 Soft 삭제된 레코드까지 포함하려면, include 옵션에 paranoid: false를 전달하면 됩니다.

User.findAll({

include: [{

model: Tool,

where: { name: { [Op.like]: '%ooth%' } },

paranoid: false

}]

});

**Eager loading된 연결 모델을 정렬하기**

일대다 연결 관계의 경우 활용할 수 있는 내용입니다.

Company.findAll({ include: [ Division ], order: [ [ Division, 'name' ] ] });

Company.findAll({ include: [ Division ], order: [ [ Division, 'name', 'DESC' ] ] });

Company.findAll({

include: [ { model: Division, as: 'Div' } ],

order: [ [ { model: Division, as: 'Div' }, 'name' ] ]

});

Company.findAll({

include: [ { model: Division, as: 'Div' } ],

order: [ [ { model: Division, as: 'Div' }, 'name', 'DESC' ] ]

});

Company.findAll({

include: [ { model: Division, include: [ Department ] } ],

order: [ [ Division, Department, 'name' ] ]

});

include의 인자를 보면 배열 내에 단일 요소만 전달할 때와 객체로 전달할 때가 다른 것을 알 수 있습니다. JOIN할 모델만 명시하고자 할 때는 모델 이름만을 단일 요소로 전달합니다. 반면 JOIN할 모델에 대한 별칭, 기타 JOIN할 모델에 대한 설정을 전달하려면 객체를 사용하여 다같이 작성해주면 됩니다.

include에 전달해주는 여러 속성값들은 마치 sequelize.define을 해줄 때와 유사하게 임시적인 세팅값을 전달해준다고 볼 수 있습니다. Subquery를 직관적으로 떠올리시면 이해가 더 빠를 듯 합니다.

JOIN은 여러번 수행할 수 있습니다. 5번째 findAll()을 보면, include 안에 include를 또 사용하였죠. 여기서 Department 모델은 Division 모델과 연결을 형성하고, Division 모델은 Company와 연결 관계를 형성합니다. 만약 연결 관계가 없는데 이러한 코드를 시도하면 오류가 발생합니다.

이곳에서 사용된 order의 문법은 [다음 글](https://velog.io/@cadenzah/sequelize-document-2#%EC%A0%95%EB%A0%AC)에서 다루고 있습니다. 궁금하신 분들은 먼저 보고 오세요.

다대다 연결 관계의 경우, through 테이블 내의 컬럼으로 정렬할 수도 있습니다.

Company.findAll({

include: [ { model: Division, include: [ Department ] } ],

order: [ [ Division, DepartmentDivision, 'name' ] ]

});

**중첩된 Eager loading**

연결된 모델의 모든 연결된 모델을 불러오기 위하여 중첩된 Eager loading을 할 수 있습니다.

User.findAll({

include: [

{model: Tool, as: 'Instruments', include: [

{model: Teacher, include: [ /\* etc \*/]}

]}

]

}).then(users => {

console.log(JSON.stringify(users))

/\*

[{

"name": "John Doe",

"id": 1,

"createdAt": "2013-03-20T20:31:45.000Z",

"updatedAt": "2013-03-20T20:31:45.000Z",

"Instruments": [{ // 1:M and N:M association

"name": "Toothpick",

"id": 1,

"createdAt": null,

"updatedAt": null,

"userId": 1,

"Teacher": { // 1:1 association

"name": "Jimi Hendrix"

}

}]

}]

\*/

})

위의 코드는 OUTER JOIN의 결과를 만듭니다. 그런데 연결된 모델에서 where문을 사용할 경우 INNER JOIN을 수행하게 되고, 조건에 부합하는 서브 모델의 항목만을 반환할 것입니다. 모든 항목을 반환하려면, required: false 옵션을 추가합니다.

User.findAll({

include: [{

model: Tool,

as: 'Instruments',

include: [{

model: Teacher,

where: {

school: "Woodstock Music School"

},

required: false

}]

}]

}).then(users => {

/\* ... \*/

})

위의 쿼리문은 모든 User, 그리고 각 User가 가진 모든 Instrument를 반환하지만, Teacher의 경우 Woodstock Music School에 소속된 것만 표시될 것입니다.

all 옵션과 함께 nested 속성을 사용하는 것도 가능합니다.

User.findAll({ include: [{ all: true, nested: true }]});

**코멘트**

Sequelize 도큐먼트는 썩 친절하지 않다고 느껴집니다. 제공하는 기능이 많기에 도큐먼트가 방대하겠지만, 모든 기능을 잘 알려줄 정도로 도큐먼트가 충실하다는 인상은 아닙니다. 도큐먼트 뒷부분에서 등장하는 내용을 별다른 설명없이 뜬금포로 사용하여 앞부분 내용을 설명하기도 하는 통에, 저처럼 답답하면 진도를 못 빼는 성격으로는 정말 고통스러운 설명 스타일입니다.

SQL을 알더라도 용어를 묘하게 다르게 쓰다보니 헷갈리는 것도 문제입니다. 이게 이해를 헷갈리게 만드는 경우가 너무 많은 듯 합니다.

원래는 Association 부분만 읽어보려고 했는데, 정리가 제대로 안 되서 그냥 처음부터 쭉 읽어보고 정리해보기로 했습니다.

**그 밖에 번역하다가 생각난 점들**

* ORM은 결국 DAO를 자바스크립트 개발자에게 제공하는 것. 그래서 Sequelize가 제공하는 모든 데이터는 DAO로서 래핑된 것들이다. 한 데이터를 열람하더라도 그와 연관된 다양한 메서드와 데이터를 부가적으로 알아서 제공해준다. include나 as는 그런 때에 사용되는 문법. 게다가 한 테이블은 다른 테이블과 서로 연결된 경우가 흔하므로... 이러한 의도와 설계를 이해하고 공부하면 좀 더 이해가 직관적이고 명확할 것이다.
* 이래저래 Sequelize는 postgreSQL을 밀어주는 ORM처럼 보인다. 전용 문법이 꽤 있다...

다음 편은 [**5장 Querying**](https://velog.io/@cadenzah/sequelize-document-2) 으로 이어집니다.

[db](https://velog.io/tags/db)[javascript](https://velog.io/tags/javascript)[mysql](https://velog.io/tags/mysql)[ORM](https://velog.io/tags/ORM)[Sequelize](https://velog.io/tags/Sequelize)

# Sequelize 공식 Document - (2) Querying

2019년 1월 5일



# 쿼리

* 이 글은 [5. Querying](http://docs.sequelizejs.com/manual/tutorial/querying.html)를 번역한 글입니다.
* 아직 입문자이다보니 오역이 있을 수 있습니다. 양해 부탁드립니다.
* 매끄러운 번역을 위하여 의역을 한 경우가 있습니다. 원문의 뜻을 최대한 해치지 않도록 노력했으니 안심하셔도 됩니다.
* 영어 단어가 자연스러운 경우 원문 그대로의 영단어를 적었습니다.
* 저의 보충 설명은 인용문에 적었습니다.

## 시리즈 목록

* [Sequelize 공식 Document - (1) Model Usage](https://velog.io/@cadenzah/sequelize-document-1)
* [**Sequelize 공식 Document - (2) Querying**](https://velog.io/@cadenzah/sequelize-document-2)
* [Sequelize 공식 Document - (3) Instances](https://velog.io/@cadenzah/sequelize-document-3)
* [Sequelize 공식 Document - (4) Instances (상)](https://velog.io/@cadenzah/sequelize-document-4)

## 시작하기 전에

* 이 글은 공식 도큐먼트의 5번째 챕터입니다.
* 원문에서는 예시 코드를 제공하지만, 코드만 제공할 뿐 CRUD를 위한 데이터셋 세팅, 실습 환경에 대한 정보가 없고, 실행 결과를 확인할 수 있는 방법도 알려주지 않습니다. 그래서 간단하게나마 실습해볼 수 있도록 세팅하는 방법을 [이전 글](https://velog.io/@cadenzah/sequelize-document-1)에서 소개하였으니 필요에 따라 활용해주시면 감사드리겠습니다.

## 특성 (Attributes)

SQL의 테이블에 대한 인스턴스를 모델이라고 할 때, 원본 테이블에 존재하는 **컬럼** 은 모델 인스턴스의 **특성** 에 대응됩니다.

일부 특성만을 select하려면 attributes 옵션을 사용하면 됩니다. 대부분의 경우 배열을 전달합니다:

Model.findAll({

attributes: ['foo', 'bar']

});

SELECT foo, bar ...

특성 이름은 중첩 배열을 사용하면 새로 지어줄 수 있습니다:

Model.findAll({

// 'bar' 특성을 찾되, 반환된 결과에서는 'baz'로 표시됩니다

attributes: ['foo', ['bar', 'baz']]

});

SELECT foo, bar AS baz ...

집계(Aggregation)를 하려면 sequelize.fn을 사용하면 됩니다:

Model.findAll({

attributes: [[sequelize.fn('COUNT', sequelize.col('hats')), 'no\_hats']]

});

SELECT COUNT(hats) AS no\_hats ...

집계 함수를 사용할 때, 반드시 해당 함수 결과에 대하여 별칭을 부여해야 모델에서 해당 결과에 접근할 수 있습니다. 바로 위의 예시에서는 모자의 개수를 instance.get('no\_hats') 를 통하여 얻을 수 있습니다.

쿼리 결과에 단지 집계값만을 더하기 위하여 모든 특성을 나열하는 것은 귀찮은 일일 수 있습니다:

// This is a tiresome way of getting the number of hats...

Model.findAll({

attributes: ['id', 'foo', 'bar', 'baz', 'quz', [sequelize.fn('COUNT', sequelize.col('hats')), 'no\_hats']]

});

// This is shorter, and less error prone because it still works if you add / remove attributes

Model.findAll({

attributes: { include: [[sequelize.fn('COUNT', sequelize.col('hats')), 'no\_hats']] }

});

SELECT id, foo, bar, baz, quz, COUNT(hats) AS no\_hats ...

비슷한 방식으로 일부 특성을 제외하는 것도 가능합니다:

Model.findAll({

attributes: { exclude: ['baz'] }

});

SELECT id, foo, bar, quz ...

연결된 모델과 JOIN할 때에 사용되는 include와 달리, attributes 속성 안에서 사용된 include는 **어떤 컬럼을 포함할지** 를 제한해주는 옵션입니다. exclude와 세트로 가는 옵션입니다.

## Where

어떤 쿼리문을 작성하든, where 객체를 사용하여 쿼리 내용을 필터링할 수 있습니다. 이는 find/findAll, update, destroy 등에 모두 사용 가능합니다.

where은 **attribute: 값** 의 쌍으로 이루어진 객체이며, **값** 으로는 비교 연산의 경우 기본 자료형의 값, 그 외의 연산자의 경우 또다른 객체가 올 수 있습니다.

or와 and 연산자를 중첩하여 사용하면 복잡한 AND/OR 조건을 만들어내는 것도 가능합니다.

### 기본 원칙

const Op = Sequelize.Op;

Post.findAll({

where: {

authorId: 2

}

});

// SELECT \* FROM post WHERE authorId = 2

Post.findAll({

where: {

authorId: 12,

status: 'active'

}

});

// SELECT \* FROM post WHERE authorId = 12 AND status = 'active';

Post.findAll({

where: {

[Op.or]: [{authorId: 12}, {authorId: 13}]

}

});

// SELECT \* FROM post WHERE authorId = 12 OR authorId = 13;

Post.findAll({

where: {

authorId: {

[Op.or]: [12, 13]

}

}

});

// SELECT \* FROM post WHERE authorId = 12 OR authorId = 13;

Post.destroy({

where: {

status: 'inactive'

}

});

// DELETE FROM post WHERE status = 'inactive';

Post.update({

updatedAt: null,

}, {

where: {

deletedAt: {

[Op.ne]: null

}

}

});

// UPDATE post SET updatedAt = null WHERE deletedAt NOT NULL;

Post.findAll({

where: sequelize.where(sequelize.fn('char\_length', sequelize.col('status')), 6)

});

// SELECT \* FROM post WHERE char\_length(status) = 6;

**{ where: { 컬럼\_이름: { 조건 }}}** 순으로 가면 됩니다. 조건이 기본 자료형의 값이면 그냥 적어주면 되지만, 비교 등의 연산인 경우 그 조건을 객체로 전달해줘야 합니다.  
맨 마지막 쿼리처럼 where에 집계 함수가 들어간 조건을 사용하면, 함수를 그대로 속성값으로 써주는 것이 아니라 sequelize.where로 한번 더 감싸줘야 합니다.

### 연산자

Sequelize는 자바스크립트 Symbol 연산자를 사용하여 복잡한 비교 연산을 지원합니다.

[Op.and]: {a: 5} // AND (a = 5)

[Op.or]: [{a: 5}, {a: 6}] // (a = 5 OR a = 6)

[Op.gt]: 6, // > 6

[Op.gte]: 6, // >= 6

[Op.lt]: 10, // < 10

[Op.lte]: 10, // <= 10

[Op.ne]: 20, // != 20

[Op.eq]: 3, // = 3

[Op.not]: true, // IS NOT TRUE

[Op.between]: [6, 10], // BETWEEN 6 AND 10

[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15

[Op.in]: [1, 2], // IN [1, 2]

[Op.notIn]: [1, 2], // NOT IN [1, 2]

[Op.like]: '%hat', // LIKE '%hat'

[Op.notLike]: '%hat' // NOT LIKE '%hat'

[Op.iLike]: '%hat' // ILIKE '%hat' (case insensitive) (PG only)

[Op.notILike]: '%hat' // NOT ILIKE '%hat' (PG only)

[Op.regexp]: '^[h|a|t]' // REGEXP/~ '^[h|a|t]' (MySQL/PG only)

[Op.notRegexp]: '^[h|a|t]' // NOT REGEXP/!~ '^[h|a|t]' (MySQL/PG only)

[Op.iRegexp]: '^[h|a|t]' // ~\* '^[h|a|t]' (PG only)

[Op.notIRegexp]: '^[h|a|t]' // !~\* '^[h|a|t]' (PG only)

[Op.like]: { [Op.any]: ['cat', 'hat']}

// LIKE ANY ARRAY['cat', 'hat'] - also works for iLike and notLike

[Op.overlap]: [1, 2] // && [1, 2] (PG array overlap operator)

[Op.contains]: [1, 2] // @> [1, 2] (PG array contains operator)

[Op.contained]: [1, 2] // <@ [1, 2] (PG array contained by operator)

[Op.any]: [2,3] // ANY ARRAY[2, 3]::INTEGER (PG only)

[Op.col]: 'user.organization\_id' // = "user"."organization\_id", with dialect specific column identifiers, PG in this example

#### 범위 연산자

지원하는 모든 연산자로 다양한 종류의 범위를 쿼리할 수 있습니다.  
제공되는 범위값으로 경계 포함 여부를 정의할 수도 있습니다.

// All the above equality and inequality operators plus the following:

[Op.contains]: 2 // @> '2'::integer (PG range contains element operator)

[Op.contains]: [1, 2] // @> [1, 2) (PG range contains range operator)

[Op.contained]: [1, 2] // <@ [1, 2) (PG range is contained by operator)

[Op.overlap]: [1, 2] // && [1, 2) (PG range overlap (have points in common) operator)

[Op.adjacent]: [1, 2] // -|- [1, 2) (PG range is adjacent to operator)

[Op.strictLeft]: [1, 2] // << [1, 2) (PG range strictly left of operator)

[Op.strictRight]: [1, 2] // >> [1, 2) (PG range strictly right of operator)

[Op.noExtendRight]: [1, 2] // &< [1, 2) (PG range does not extend to the right of operator)

[Op.noExtendLeft]: [1, 2] // &> [1, 2) (PG range does not extend to the left of operator)

PostgreSQL을 제가 사용하지 않아서 제대로 해석하는지 잘 모르는 점 양해 부탁드립니다.

#### 여러가지 조합

const Op = Sequelize.Op;

{

createdAt: {

[Op.lt]: new Date(),

[Op.gt]: new Date(new Date() - 24 \* 60 \* 60 \* 1000)

}

}

// createdAt < [timestamp] AND createdAt > [timestamp]

{

rank: {

[Op.or]: {

[Op.lt]: 1000,

[Op.eq]: null

}

}

}

// rank < 1000 OR rank IS NULL

{

[Op.or]: [

{

title: {

[Op.like]: 'Boat%'

}

},

{

description: {

[Op.like]: '%boat%'

}

}

]

}

// title LIKE 'Boat%' OR description LIKE '%boat%'

1. 조건 객체에서 속성들을 나열하는 경우 AND로 간주됩니다.
2. 특정 컬럼을 위한 OR인 경우, 단일 조건 객체로 전달합니다.
3. 여러 컬럼을 다루는 OR안 경우, 각 컬럼에 대한 조건을 담은 객체의 배열로 작성합니다. 각 객체에는 특정 컬럼에 대한 조건이 중첩되어 작성됩니다.  
   where에 대한 조건 객체는 속성이 하나인 단일 객체입니다.

#### 연산자 별칭

Sequelize는 연산자에 대하여 별칭을 부여할 수 있게 해줍니다.

const Op = Sequelize.Op;

const operatorsAliases = {

$gt: Op.gt

}

const connection = new Sequelize(db, user, pass, { operatorsAliases })

[Op.gt]: 6 // > 6

$gt: 6 // 위와 동일

#### 연산자 보안

별칭을 사용하지 않으면 Sequelize의 보안 향상에 도움이 됩니다. 일부 프레임워크는 사용자 입력을 자바스크립트 객체로 자동 변환하는데, 이를 제대로 검증하지 않으면 문자로 적힌 연산자가 쓰인 객체를 Sequelize에 주입하는 일이 벌어질 수도 있습니다.

SQL Injection을 말하는 것입니다.

문자로 된 별칭을 일절 사용하지 않으면 연산자가 주입되는 것을 극단적으로 막을 수 있겠지만, 그럼ㄹ에도 불구하고 항상 사용자 입력을 적절하게 검증하고 처리해야 합니다.

하위 버전 호환을 위하여 Sequelize는 다음의 별칭들을 기본값으로 설정합니다:

$eq, $ne, $gte, $gt, $lte, $lt, $not, $in, $notIn, $is, $like, $notLike, $iLike, $notILike, $regexp, $notRegexp, $iRegexp, $notIRegexp, $between, $notBetween, $overlap, $contains, $contained, $adjacent, $strictLeft, $strictRight, $noExtendRight, $noExtendLeft, $and, $or, $any, $all, $values, $col

다음의 별칭들도 설정되어있으나, 가까운 미래에는 완전히 제거될 계획입니다:

ne, not, in, notIn, gte, gt, lte, lt, like, ilike, $ilike, nlike, $notlike, notilike, .., between, !.., notbetween, nbetween, overlap, &&, @>, <@

향상된 보안을 위하여 Sequelize.Op를 사용하고, 문자로 된 별칭은 절대로 사용하지 마세요. 여러분의 프로그램에 필요한 별칭을 operatorsAliases 옵션을 사용하여 제한할 수 있습니다. Sequelize의 메서드에 사용자 입력을 전달할 때는, 반드시 적절하게 검증하고 처리해야 한다는 것을 잊어서는 안 됩니다!

const Op = Sequelize.Op;

// 연산자 별칭 없이 Sequelize를 사용

const connection = new Sequelize(db, user, pass, { operatorsAliases: false });

// $and => Op.and 만을 별칭으로서 Sequelize를 사용

const connection2 = new Sequelize(db, user, pass, { operatorsAliases: { $and: Op.and } });

아무런 설정 없이 기본값 별칭을 모두 사용하고자 할 경우 Sequelize는 경고를 띄웁니다. 경고를 보지 않으면서 기본값 별칭을 사용하고 싶다면, 다음의 operatorAliases 옵션을 사용하세요:

const Op = Sequelize.Op;

const operatorsAliases = {

$eq: Op.eq,

$ne: Op.ne,

$gte: Op.gte,

$gt: Op.gt,

$lte: Op.lte,

$lt: Op.lt,

$not: Op.not,

$in: Op.in,

$notIn: Op.notIn,

$is: Op.is,

$like: Op.like,

$notLike: Op.notLike,

$iLike: Op.iLike,

$notILike: Op.notILike,

$regexp: Op.regexp,

$notRegexp: Op.notRegexp,

$iRegexp: Op.iRegexp,

$notIRegexp: Op.notIRegexp,

$between: Op.between,

$notBetween: Op.notBetween,

$overlap: Op.overlap,

$contains: Op.contains,

$contained: Op.contained,

$adjacent: Op.adjacent,

$strictLeft: Op.strictLeft,

$strictRight: Op.strictRight,

$noExtendRight: Op.noExtendRight,

$noExtendLeft: Op.noExtendLeft,

$and: Op.and,

$or: Op.or,

$any: Op.any,

$all: Op.all,

$values: Op.values,

$col: Op.col

};

const connection = new Sequelize(db, user, pass, { operatorsAliases });

### JSON

JSON 타입은 PostgreSQL, SQLite, MySQL 에서만 사용할 수 있습니다.

#### PostgreSQL

PostgreSQL의 JSON 타입은 값을 이진값이 아니라 평문으로 저장합니다. JSON 표현을 단지 저장하거나 반환하고 싶다면, JSON을 사용하는 것이 공간과 시간을 절약할 것입니다. 하지만 JSON 값에 대하여 어떤 연산을 수행하려면, JSONB를 사용하는 편이 좋습니다.

#### MSSQL

MSSQL은 JSON 타입을 가지 않지만, SQL Server 2016 이후로 스트링 형태로 저장된 JSON을 지원하는 함수를 제공합니다. 이 함수를 사용하면 스트링으로 저장된 JSON을 쿼리할 수 있지만, 반환된 값은 파싱되어야 합니다.

// ISJSON - to test if a string contains valid JSON

User.findAll({

where: sequelize.where(sequelize.fn('ISJSON', sequelize.col('userDetails')), 1)

})

// JSON\_VALUE - extract a scalar value from a JSON string

User.findAll({

attributes: [[ sequelize.fn('JSON\_VALUE', sequelize.col('userDetails'), '$.address.Line1'), 'address line 1']]

})

// JSON\_VALUE - query a scalar value from a JSON string

User.findAll({

where: sequelize.where(sequelize.fn('JSON\_VALUE', sequelize.col('userDetails'), '$.address.Line1'), '14, Foo Street')

})

// JSON\_QUERY - extract an object or array

User.findAll({

attributes: [[ sequelize.fn('JSON\_QUERY', sequelize.col('userDetails'), '$.address'), 'full address']]

})

### JSONB

JSONB는 3가지 방법으로 쿼리할 수 있습니다.

#### 중첩 객체

{

meta: {

video: {

url: {

[Op.ne]: null

}

}

}

}

#### 중첩 키

{

"meta.audio.length": {

[Op.gt]: 20

}

}

#### 포함

{

"meta": {

[Op.contains]: {

site: {

url: 'http://google.com'

}

}

}

}

### 연결 관계

// task.state === project.state 를 만족하는 모든 Project를 찾는다

Project.findAll({

include: [{

model: Task,

where: { state: Sequelize.col('project.state') }

}]

})

## Pagination / Limiting

// 10개 인스턴스/행 을 가져온다

Project.findAll({ limit: 10 })

// 8개 인스턴스/행 을 넘어간다

Project.findAll({ offset: 8 })

// 5개 인스턴스/행을 넘어가고, 거기서부터 5개 인스턴스/행을 가져온다

Project.findAll({ offset: 5, limit: 5 })

## 정렬

order는 항목으로 이루어진 배열을 받아서 컬럼 또는 Sequelize 메서드로 정렬합니다. 보통은 특성, 방향, 넘어가는 양 등으로 이루어진 쌍/배열을 사용합니다.

Subtask.findAll({

order: [

// 'title'로 이스케이프하고, DESC를 방향 관련 인자 목록에서 활성화한다

['title', 'DESC'],

// max(age) 로 정렬

sequelize.fn('max', sequelize.col('age')),

// max(age) DESC 로 정렬

[sequelize.fn('max', sequelize.col('age')), 'DESC'],

// otherfunction(`col1`, 12, 'lalala') DESC 로 정렬

[sequelize.fn('otherfunction', sequelize.col('col1'), 12, 'lalala'), 'DESC'],

// (1) 연결된 모델의 createdAt 순으로 정렬하는데, 이때 연결 관계의 이름으로 모델의 이름(Task)을 사용한다.

[Task, 'createdAt', 'DESC'],

// (2) 중첩 연결된 모델(Project)의 createdAt 순으로 정렬하는데, 이떄 각 연결 관계의 이름으로 각 모델의 이름을 사용한다.

[Task, Project, 'createdAt', 'DESC'],

// (3) 연결된 모델의 createdAt 순으로 정렬하는데, 이때 연결 관계의 이름을 사용한다.

['Task', 'createdAt', 'DESC'],

// (4) 중첩 연결된 모델(Project)의 createdAt 순으로 정렬하는데, 이때 각 연결 관계의 이름들을 사용한다.

['Task', 'Project', 'createdAt', 'DESC'],

// (5) 연결된 모델의 createdAt 순으로 정렬하는데, 이때 연결 관계 객체를 사용한다. (추천하는 방법)

[Subtask.associations.Task, 'createdAt', 'DESC'],

// (6) 중첩 연결된 모델(Project)의 createdAt 순으로 정렬하는데, 이때 연결 관계 객체들을 사용한다. (추천하는 방법))

[Subtask.associations.Task, Task.associations.Project, 'createdAt', 'DESC'],

// (7) 연결된 모델의 createdAt 순으로 정렬하는데, 이때 연결 관계가 표현된 객체를 사용한다.

[{model: Task, as: 'Task'}, 'createdAt', 'DESC'],

// (8) 중첩 연결된 모델(Project)의 createdAt 순으로 정렬하는데, 이때 연결 관계가 표현된 객체들을 사용한다.

[{model: Task, as: 'Task'}, {model: Project, as: 'Project'}, 'createdAt', 'DESC']

]

// 인자에 적힌 스트링을 그대로 SQL 문법으로 변환

order: sequelize.literal('max(age) DESC')

// 방향이 생략되면, 오름차순을 기본값으로 정렬한다

order: sequelize.fn('max', sequelize.col('age'))

// age 순으로 오름차순 정렬

order: sequelize.col('age')

// dialect에 따라 무작위 정렬 수행 (fn('RAND') 또는 fn('RANDOM') 대신 사용)

order: sequelize.random()

})

[이전 글](https://velog.io/@cadenzah/sequelize-document-1#eager-loading%EB%90%9C-%EC%97%B0%EA%B2%B0-%EB%AA%A8%EB%8D%B8%EC%9D%84-%EC%A0%95%EB%A0%AC%ED%95%98%EA%B8%B0)에서 답답했던 부분에 대한 해결이 되시나요? 지금도 예시만 던져줄 뿐, 명료한 설명이 부족하여 저도 이해하는 데에 애를 먹었습니다. JOIN된 테이블에 대하여 정렬을 하는 경우에 대한 예시들입니다. 간단하게 요약하자면, order 조건을 보이는 배열에서, 가장 마지막 2개 요소가 각각 **컬럼** 과 **방향** 에 대한 값이고, 그 앞의 것들은 JOIN 연결 관계를 형성한 테이블(모델)을 가리키는 것이라고 생각하면 됩니다.

아직 이 글을 읽는 시점에서는 **Association(연결 관계)** 에 대한 지식을 알지 못하므로, 잘 모른다고 하여 헤매시지 않아도 됩니다. 나중 글에서 연결 관계에 대하여 알고 난 뒤 다시 읽어보면 좀 더 명확하게 이해가 될 겁니다.

원문에 적혀있는 대로 (5)와 (6) 방법이 가장 명료한 표기로 보입니다. 이 표기법을 주로 사용하도록 합시다. 그렇지만 나중 글에서는 (7)와 (8) 표기법도 많이 사용하더군요. ;(

### 테이블 힌트

tableHint는 MSSQL을 사용할 때에 테이블 힌트를 전달하는 데에 사용됩니다. 힌트값은 Sequelize.TableHints이어야 하며, 반드시 필요할 때에만 사용되어야 합니다. 현재는 쿼리마다 단 하나의 테이블 힌트만 사용 가능합니다.

테이블 힌트는 특정 옵션을 정의하는 것으로 MSSQL 쿼리 최적화기의 기본 동작을 덮어씁니다. 해당 구절에서 참조되는 테이블 또는 뷰에만 영향을 미칩니다.

const TableHints = Sequelize.TableHints;

Project.findAll({

// adding the table hint NOLOCK

tableHint: TableHints.NOLOCK

// this will generate the SQL 'WITH (NOLOCK)'

})

## 코멘트

역시나 불친절한 도큐먼트입니다. 이전 글을 작성할 때, order에 대한 설명이 부실하여 이해하는 데에 어려움이 컸는데, 이번 챕터라고 해서 설명이 특별히 자세하지는 않네요. 게다가 연결 관계에 대한 설명도 없이 일단 예시부터 던지고 본다니...

다음 편은 [**6장 Instances**](https://velog.io/@cadenzah/sequelize-document-3) 로 이어집니다.

[자바스크립트](https://velog.io/tags/%EC%9E%90%EB%B0%94%EC%8A%A4%ED%81%AC%EB%A6%BD%ED%8A%B8)[시퀄라이즈](https://velog.io/tags/%EC%8B%9C%ED%80%84%EB%9D%BC%EC%9D%B4%EC%A6%88)



**인스턴스**

* 이 글은 [6. Instances](http://docs.sequelizejs.com/manual/tutorial/instances.html)를 번역한 글입니다.
* 아직 입문자이다보니 오역이 있을 수 있습니다. 양해 부탁드립니다.
* 매끄러운 번역을 위하여 의역을 한 경우가 있습니다. 원문의 뜻을 최대한 해치지 않도록 노력했으니 안심하셔도 됩니다.
* 영어 단어가 자연스러운 경우 원문 그대로의 영단어를 적었습니다.
* 저의 보충 설명은 인용문에 적었습니다.

**시리즈 목록**

* [Sequelize 공식 Document - (1) Model Usage](https://velog.io/@cadenzah/sequelize-document-1)
* [Sequelize 공식 Document - (2) Querying](https://velog.io/@cadenzah/sequelize-document-2)
* [***Sequelize 공식 Document - (3) Instances***](https://velog.io/@cadenzah/sequelize-document-3)
* [Sequelize 공식 Document - (4) Instances (상)](https://velog.io/@cadenzah/sequelize-document-4)

**시작하기 전에**

* 이 글은 공식 도큐먼트의 6번째 챕터입니다.
* 원문에서는 예시 코드를 제공하지만, 코드만 제공할 뿐 CRUD를 위한 데이터셋 세팅, 실습 환경에 대한 정보가 없고, 실행 결과를 확인할 수 있는 방법도 알려주지 않습니다. 그래서 간단하게나마 실습해볼 수 있도록 세팅하는 방법을 [이전 글](https://velog.io/@cadenzah/sequelize-document-1)에서 소개하였으니 필요에 따라 활용해주시면 감사드리겠습니다.

**변하는 인스턴스 만들기**

정의된 클래스의 인스턴스를 생성하려면 다음을 따라하면 됩니다. 과거에 Ruby를 사용해봤다면 익숙한 문법일 것입니다. build 메서드를 사용하면 저장이 되지 않는 객체, 그말인즉슨 명시적으로 저장을 해야하는 객체를 반환합니다.

const project = Project.build({

title: 'my awesome project',

description: 'woot woot. this will make me a rich man'

})

const task = Task.build({

title: 'specify the project idea',

description: 'bla',

deadline: new Date()

})

생성된 인스턴스는 클래스가 정의될 때 정해진 기본값을 자동으로 갖습니다:

// 우선, 모델을 정의한다.

const Task = sequelize.define('task', {

title: Sequelize.STRING,

rating: { type: Sequelize.STRING, defaultValue: 3 }

})

// 이제 인스턴스를 생성한다.

const task = Task.build({title: 'very important task'})

task.title // ==> 'very important task'

task.rating // ==> 3 (기본값)

DB에 저장하려면 save 메서드를 사용하고, 필요에 따라 catch로 예외를 처리해줍니다.

project.save().then(() => {

// 제대로 저장된 뒤에 할 일들

})

task.save().catch(error => {

// mhhh, wth!

})

// 인스턴스를 생성하고, 저장한 뒤, 해당 인스턴스에 체이닝을 통하여 접근할 수 있다:

Task

.build({ title: 'foo', description: 'bar', deadline: new Date() })

.save()

.then(anotherTask => {

// anotherTask 변수를 통하여 방금 전에 저정한 task 인스턴스에 접근할 수 있다... 어게이~

})

.catch(error => {

// 저런, 오류 처리를 하자.

})

**변하지 않는 인스턴스를 만들기**

.build()로 생성한 인스턴스를 DB에 저장하려면 명시적으로 .save()를 호출해야 하지만, .create()를 사용하면 이 모든 작업을 생략하고, 인스턴스의 데이터를 자동으로 DB에 저장시켜줄 수 있습니다.

Task.create({ title: 'foo', description: 'bar', deadline: new Date() }).then(task => {

// task 변수를 통하면 새로 생성된 인스턴스에 접근할 수 있다.

})

create 메서드를 통하여 어떤 특성이 설정될지 선택할 수 있습니다. 특히 이것은 사용자 입력을 폼으로 받아서 DB에 저장할 때에 편리합니다. 아래의 예시에서는, User 모델에서 username 특성만 저장하고, isAdmin 플래그 특성은 저장하지 않도록 제한하고 있습니다.

User.create({ username: 'barfooz', isAdmin: true }, { fields: [ 'username' ] }).then(user => {

// isAdmin의 기본값이 false라고 가정하자

console.log(user.get({

plain: true

})) // => { username: 'barfooz', isAdmin: false }

})

**Update / Save / Persisting- DB에 인스턴스를 저장하기**

이번에는 값을 변경하고, 변경점을 DB에 반영해봅시다. 2가지 방법이 있습니다:

// 방법 1

task.title = 'a very different title now'

task.save().then(() => {})

// 방법 2

task.update({

title: 'a very different title now'

}).then(() => {})

save 메서드를 호출했을 때 어떤 특성이 저장되는지를 정의할 수도 있습니다. 컬럼 이름으로 이루어진 배열을 전달하면 됩니다. 이렇게 하면 이미 정의된 객체를 활용하여 특성을 설정할 수 있어서 편리합니다. 웹 사이트의 폼을 통하여 값을 받는 경우 등을 떠올려보면 됩니다. 이 방식은 update에서 내부적으로 사용되는 방식이기도 합니다. 아래와 같이 사용됩니다:

task.title = 'foooo'

task.description = 'baaaaaar'

task.save({fields: ['title']}).then(() => {

// title은 'foooo'라는 값을 가지지만, description은 값이 변하지 않는다.

})

// 동등한 작업을 update로 수행하면 아래와 같다:

task.update({ title: 'foooo', description: 'baaaaaar'}, {fields: ['title']}).then(() => {

// title은 'foooo'라는 값을 가지지만, description은 값이 변하지 않는다.

})

아무런 특성도 변하지 않은 상태에서 save를 실행하면, 이 메서드는 아무 작업도 수행하지 않습니다.

**Destroying / DB의 항목을 삭제하기**

DB에 항목을 생성한 뒤 해당 항목에 대한 참조를 얻었다면, DB에서 제거하는 것도 가능합니다.

Task.create({ title: 'a task' }).then(task => {

// now you see me...

return task.destroy();

}).then(() => {

// now i'm gone :)

})

paranoid 옵션이 true라면, 객체는 제거되지 않고 deleteAt 컬럼이 현재의 timestamp 값을 갖게 됩니다. 강제로 삭제하려면, destroy 호출시 force: true를 인자로 전달하면 됩니다.

task.destroy({ force: true })

**Working in bulk - 다수 항목에 대한 생성, 갱신, 삭제 작업**

단일 인스턴스에 대한 작업뿐만 아니라, 다수 인스턴스를 한번에 생성하거나, 변경하거나, 삭제할 수 있습니다. 여기서 사용할 수 있는 메서드는 다음과 같은 형식을 가집니다:

* Model.bulkCreate
* Model.updated
* Model.destroy

여러 항목에 대한 작업이기 때문에, 이 메서드들은 DAO 인스턴스를 반환하지 않습니다. BulkCreate는 인스턴스/DAO로 이루어진 배열을 반환하지만, create와 달리 autoIncrement 특성의 결과값을 갖지 않습니다. update와 destroy는 영향을 받은 레코드의 개수를 반환합니다.

bulkCreate를 먼저 살펴봅니다:

User.bulkCreate([

{ username: 'barfooz', isAdmin: true },

{ username: 'foo', isAdmin: true },

{ username: 'bar', isAdmin: false }

]).then(() => { // 결과가 전달되지 않으므로, 직접 가져와야 합니다.

return User.findAll();

}).then(users => {

console.log(users) // 그래야 user 인스턴스들의 배열을 얻을 수 있습니다.

})

여러 레코드들을 한번에 update합니다:

Task.bulkCreate([

{subject: 'programming', status: 'executing'},

{subject: 'reading', status: 'executing'},

{subject: 'programming', status: 'finished'}

]).then(() => {

return Task.update(

{ status: 'inactive' }, /\* 변경할 특성의 값을 명시합니다. \*/

{ where: { subject: 'programming' }} /\* 변경될 레코드의 조건을 명시합니다. \*/

);

}).spread((affectedCount, affectedRows) => {

// .update는 배열을 통하여 2개의 값을 전달하므로, spread 메서드를 사용합니다.

// affectedRows는 returning: true를 제원하는 DB에서만 사용할 수 있습니다.

// affectedCount는 2

return Task.findAll();

}).then(tasks => {

console.log(tasks) // 'programming' task들은 둘 다 status 컬럼의 값으로 'inactive'를 갖습니다.

})

마지막으로 delete합니다:

Task.bulkCreate([

{subject: 'programming', status: 'executing'},

{subject: 'reading', status: 'executing'},

{subject: 'programming', status: 'finished'}

]).then(() => {

return Task.destroy({

where: {

subject: 'programming'

},

truncate: true /\* where을 무시하고 대신 truncate를 수행합니다. \*/

});

}).then(affectedRows => {

// affectedRows는 2

return Task.findAll();

}).then(tasks => {

console.log(tasks) // no programming, just reading :(

})

사용자로부터 값을 직접 전달받는다면, DB에 실제로 삽입되는 컬럼을 제한하는 것이 바람직합니다. bulkCreate()는 2번째 인자로 옵션 객체을 전달받습니다. 이 옵션 객체는 fields 속성(배열값)을 가지며, 항목 생성에 사용될 컬럼을 명시적으로 알려줄 수 있습니다.

User.bulkCreate([

{ username: 'foo' },

{ username: 'bar', admin: true}

], { fields: ['username'] }).then(() => {

// 'bar'의 admin 컬럼은 무시된다

})

본래 bulkCreate는 레코드를 삽입하는 빠르고 주로 쓰이는 방법으로서 고안되었습니다. 하지만, *세련된 모델 검증* 을 하면서도 동시에 여러 레코드를 삽입해야 할 때도 있을 것입니다. 그럴 때는 옵션 객체로 validate: true 속성을 넘겨주면 됩니다.

const Tasks = sequelize.define('task', {

name: {

type: Sequelize.STRING,

validate: {

notNull: { args: true, msg: 'name cannot be null' }

}

},

code: {

type: Sequelize.STRING,

validate: {

len: [3, 10]

}

}

})

Tasks.bulkCreate([

{name: 'foo', code: '123'},

{code: '1234'},

{name: 'bar', code: '1'}

], { validate: true }).catch(errors => {

/\* console.log(errors)는 아래와 같은 결과를 보입니다:

[

{ record:

...

name: 'SequelizeBulkRecordError',

message: 'Validation error',

errors:

{ name: 'SequelizeValidationError',

message: 'Validation error',

errors: [Object] } },

{ record:

...

name: 'SequelizeBulkRecordError',

message: 'Validation error',

errors:

{ name: 'SequelizeValidationError',

message: 'Validation error',

errors: [Object] } }

]

\*/

})

설명에는 적혀있지 않지만, validate를 정의하면 미리 정해놓은 오류 메세지가 출력해줄 수 있습니다. create를 사용할 때에는 알아서 validate가 설정되있는지 체크하여 해당 메세지를 띄우지만, bulkCreate는 Sequelize가 기본으로 설정한 오류 처리를 하게 됩니다. 이때 validate 옵션을 활용하고 싶다면 예시 코드와 같이 실행하면 됩니다.

**인스턴스의 값**

console.log 등을 사용하여 인스턴스의 내부를 들여다보면, 여러가지가 많이 들어있다는 것을 알 수 있습니다. 그런 것들을 다 숨기고 관심있는 정보만 추리려면, get- 속성을 사용하면 됩니다. 이 메서드를 plain: true 옵션과 함께 실행하면, 해당 인스턴스의 데이터만 반환됩니다.

Person.create({

name: 'Rambow',

firstname: 'John'

}).then(john => {

console.log(john.get({

plain: true

}))

})

// 결과:

// { name: 'Rambow',

// firstname: 'John',

// id: 1,

// createdAt: Tue, 01 May 2012 19:12:16 GMT,

// updatedAt: Tue, 01 May 2012 19:12:16 GMT

// }

**참고** : 인스턴스를 JSON으로 변환할 수 있습니다. JSON.stringify(instance)를 사용해보세요. 위의 예시와 거의 유사한 결과를 얻을 수 있습니다.

아주 중요한 팁입니다. 쿼리 결과를 편리하게 활용하려면 중요한 데이터만 추려야겠죠?

**인스턴스 다시 불러오기**

어떤 인스턴스를 DB와 동기화하려면, reload 메서드를 사용하세요. 메서드를 호출한 인스턴스에 대하여 DB에서 값을 가져와 해당하는 컬럼에 덮어써줍니다.

Person.findOne({ where: { name: 'john' } }).then(person => {

person.name = 'jane' // DB와 무관한 현재 인스턴스에 대한 변경

console.log(person.name) // 'jane'

person.reload().then(() => {

console.log(person.name) // 'john'

})

})

**증가**

동시 실행의 문제 없이 인스턴스의 값을 증가시키려면 increment를 사용하세요. 여러 가지의 방법을 제공합니다.

첫번째, 더하고 싶은 컬럼과 더하고 싶은 수치를 정합니다.

User.findById(1).then(user => {

return user.increment('my-integer-field', {by: 2})

}).then(user => {

// PostgresSQL의 경우, 기본값으로 갱신된 user를 반환한다.

// 다른 DB의 경우, 갱신된 값을 확인하려면 user.reload()를 해야 한다.

})

둘째, 여러 개의 칼럼에 대하여 수행할 수 있습니다.

User.findById(1).then(user => {

return user.increment([ 'my-integer-field', 'my-very-other-field' ], {by: 2})

}).then(/\* ... \*/)

셋째, 더하고자 하는 칼럼과 더하고자 하는 수치에 대한 키-값으로 이루어진 객체를 정의할 수 있습니다.

User.findById(1).then(user => {

return user.increment({

'my-integer-field': 2,

'my-very-other-field': 3

})

}).then(/\* ... \*/)

**감소**

동시 실행의 문제 없이 인스턴스의 값을 감소시키려면 decrement를 사용하세요. 여러 가지의 방법을 제공합니다.

첫번째, 빼고 싶은 컬럼과 빼고 싶은 수치를 정합니다.

User.findById(1).then(user => {

return user.decrement('my-integer-field', {by: 2})

}).then(user => {

// PostgresSQL의 경우, 기본값으로 갱신된 user를 반환한다.

// 다른 DB의 경우, 갱신된 값을 확인하려면 user.reload()를 해야 한다.

})

둘째, 여러 개의 칼럼에 대하여 수행할 수 있습니다.

User.findById(1).then(user => {

return user.decrement([ 'my-integer-field', 'my-very-other-field' ], {by: 2})

}).then(/\* ... \*/)

셋째, 빼고자 하는 칼럼과 빼고자 하는 수치에 대한 키-값으로 이루어진 객체를 정의할 수 있습니다.

User.findById(1).then(user => {

return user.decrement({

'my-integer-field': 2,

'my-very-other-field': 3

})

}).then(/\* ... \*/)

다음 편은 **7. Associations** 로 이어집니다.

[db](https://velog.io/tags/db)[javascript](https://velog.io/tags/javascript)[sql](https://velog.io/tags/sql)[mysql](https://velog.io/tags/mysql)[ORM](https://velog.io/tags/ORM)[Sequelize](https://velog.io/tags/Sequelize)[시퀄라이즈](https://velog.io/tags/%EC%8B%9C%ED%80%84%EB%9D%BC%EC%9D%B4%EC%A6%88)[자바스크립트](https://velog.io/tags/%EC%9E%90%EB%B0%94%EC%8A%A4%ED%81%AC%EB%A6%BD%ED%8A%B8)

**0개의 댓글**

[**로그인**](https://velog.io/?next=/@cadenzah/sequelize-document-3) 후 댓글을 작성 할 수 있습니다.

# Sequelize 공식 Document - (4) Associations (상)

2019년 1월 8일



# 관계

* 이 글은 [7. Associations](http://docs.sequelizejs.com/manual/tutorial/associations.html)를 번역한 글입니다.
* 아직 입문자이다보니 오역이 있을 수 있습니다. 양해 부탁드립니다.
* 매끄러운 번역을 위하여 의역을 한 경우가 있습니다. 원문의 뜻을 최대한 해치지 않도록 노력했으니 안심하셔도 됩니다.
* 영어 단어가 자연스러운 경우 원문 그대로의 영단어를 적었습니다.
* 저의 보충 설명은 인용문에 적었습니다.

## 시리즈 목록

* [Sequelize 공식 Document - (1) Model Usage](https://velog.io/@cadenzah/sequelize-document-1)
* [Sequelize 공식 Document - (2) Querying](https://velog.io/@cadenzah/sequelize-document-2)
* [Sequelize 공식 Document - (3) Instances](https://velog.io/@cadenzah/sequelize-document-3)
* [**Sequelize 공식 Document - (4) Associations (상)**](https://velog.io/@cadenzah/sequelize-document-4)

## 시작하기 전에

* 이 글은 공식 도큐먼트의 7번째 챕터입니다. 너무 길어서 상/하로 나눴습니다.
* 원문에서는 예시 코드를 제공하지만, 코드만 제공할 뿐 CRUD를 위한 데이터셋 세팅, 실습 환경에 대한 정보가 없고, 실행 결과를 확인할 수 있는 방법도 알려주지 않습니다. 그래서 간단하게나마 실습해볼 수 있도록 세팅하는 방법을 [이전 글](https://velog.io/@cadenzah/sequelize-document-1)에서 소개하였으니 필요에 따라 활용해주시면 감사드리겠습니다.

이번 장에서는 Sequelize에서 제공하는 다양한 연결 관계에 대하여 알아봅니다. 예를 들어 User.hasOne(Project)와 같은 메서드를 호출하면, 여기서 User 모델(함수를 호출한 모델)이 **source** 이고 Project 모델(함수에 인자로 들어가는 모델)이 **target** 입니다.

연결 관계를 정의하는 메서드 기준으로 왼쪽, 즉 함수를 호출하는 모델 객체가 **source**, 메서드 안에 인자로 전달되는 모델 객체가 **target** 입니다. 어떤 메서드를 사용하더라도 기본적으로 적용됩니다.

## 1:1 관계

일대일 관계는 하나의 외래 키로 연결된 두 모델 간의 연결 관계입니다.

### BelongsTo

BelongsTo 관계는 일대일 관계의 외래키가 **source 모델** 에 존재하는 연결 관계입니다.  
이 관계를 보여주는 간단한 예시로는, **Team** 의 일부분으로 **Player** 가 존재하면서 외래 키를 가지는 경우가 있습니다.

const Player = this.sequelize.define('player', {/\* attributes \*/});

const Team = this.sequelize.define('team', {/\* attributes \*/});

Player.belongsTo(Team); // 이제 Player 모델에는 teamId 특성이 추가되며, 여기에는 Team의 주요 키가 들어있습니다.

#### Foreign Keys

belongsTo 관계를 위한 외래 키는 [target 모델의 이름 + target 모델의 주요 키 이름] 로 자동 생성되는 것이 기본값입니다.  
기본 명명 규칙은 camelCase이지만, source 모델이 underscored: true로 설정되어있다면, 외래 키는 snake\_case로 명명됩니다.

const User = this.sequelize.define('user', {/\* attributes \*/})

const Company = this.sequelize.define('company', {/\* attributes \*/});

User.belongsTo(Company); // User 모델에 companyId 를 추가

const User = this.sequelize.define('user', {/\* attributes \*/}, { underscored: true })

const Company = this.sequelize.define('company', {

uuid: {

type: Sequelize.UUID,

primaryKey: true

}

});

User.belongsTo(Company); // User 모델에 company\_uuid 를 추가

Sequelize에서 외래 키는 모델 정의할 때에 만들어지는 것이 아니라, **두 모델의 연결 관계를 정의** 할 때에, 연결 관계에 따라 Sequelize가 적절한 모델에 컬럼을 자동으로 만들어줍니다. 이 작업과 관련된 설정을 Sequelize에게 맡길 수도 있고, 아래에 후술하겠지만 몇몇 옵션을 주어서 사용자가 제어할 수도 있습니다.

외래 키 자동 생성의 대원칙은 [target 모델의 이름 + target 모델의 주요 키 이름]입니다. 물론, 이제 나올 as 때문에 조금 헷갈립니다.

as가 사용되면 여기서 정의된 이름이 target 모델의 이름으로 사용됩니다.

const User = this.sequelize.define('user', {/\* attributes \*/})

const UserRole = this.sequelize.define('userRole', {/\* attributes \*/});

User.belongsTo(UserRole, { as: 'role' }); // userRoleId 가 아니라 roleId 가 추가된다

외래 키 설정의 관한 부분은 foreignKey 옵션으로 항상 대체할 수 있습니다. foreignKey 옵션이 사용되면 Sequelize는 무조건 이를 기반으로 외래 키를 설정합니다.

foreignKey 옵션을 주면, as 옵션을 모두 무시하고 foreignKey 옵션의 값에 따라 외래 키 컬럼을 이름짓습니다.

const User = this.sequelize.define('user', {/\* attributes \*/})

const Company = this.sequelize.define('company', {/\* attributes \*/});

User.belongsTo(Company, {foreignKey: 'fk\_company'}); // User 모델에 fk\_company 특성을 추가한다

#### Target keys

Target key는 target 모델에 있는 컬럼으로, source 모델에 있는 외래 키 컬럼은 targetKey 컬럼을 가리키게 됩니다. belongsTo 관계에서 targetKey는 기본값으로 target 모델의 주요 키를 가리킵니다. 이에 대하여 별도의 컬럼을 지정하려면 targetKey 옵션을 사용합니다.

const User = sequelize.define('user', {/\* attributes \*/})

const Company = sequelize.define('company', {

name: Sequelize.STRING

});

User.belongsTo(Company, {foreignKey: 'fk\_companyname', targetKey: 'name'});

// User 모델에 fk\_companyname을 추가합니다.

// 여기서 fk\_companyname이 가리키는 target 모델의 컬럼은 name 컬럼입니다.

MySQL를 사용하여 위 예제를 실행할 경우 오류가 발생합니다. **외래 키가 참조하는 다른 테이블의 컬럼은 반드시 주요 키** 이어야 하기 때문입니다. 다른 DB는 제가 확인을 해보지 않아서 잘 모르겠습니다만, MySQL을 사용하시는 분들은 당황하지 마시기 바랍니다. 앞으로도 이런 경우가 종종 발생합니다.  
**참고:**[**MySQL Foreign key 사용 시 주의 사항**](http://jason-heo.github.io/mysql/2014/03/05/mysql-foreign-key.html)

Unhandled rejection SequelizeDatabaseError: Failed to add the foreign key constraint. Missing index for constraint 'users\_ibfk\_1' in the referenced table 'companies'

### HasOne

HasOne 관계는 일대일 관계를 위한 외래 키가 **target 모델** 에 존재하는 관계입니다.

const User = sequelize.define('user', {/\* ... \*/})

const Project = sequelize.define('project', {/\* ... \*/})

// 일방향적 관계

Project.hasOne(User)

/\*

이 예제에서 hasOne()은 User 모델에 projectId을 자동으로 추가합니다.

또한, define할 때 사용된 첫번째 인자(테이블 이름)를 바탕으로 Project.prototype에 getUser와 setUser 메서드가 추가됩니다.

underscore 스타일 옵션을 설정했다면, 추가된 컬럼은 projectId가 아니라, project\_id로 명명됩니다.

이제 외래 키는 users 테이블에 존재합니다.

이미 존재하던 테이블에 대하여 외래 키를 추가하고 싶다면, 그것도 가능합니다. (하단 참조)

\*/

Project.hasOne(User, { foreignKey: 'initiator\_id' })

MySQL의 경우, 이미 정의가 완료되어 DB에 실제로 생성된 테이블에는 외래 키 설정을 통한 새로운 컬럼의 추가 등을 할 수 없습니다. 해당 테이블을 제거한 뒤 다시 Sequelize 코드를 실행해야 제대로 적용됩니다.

/\*

Sequelize는 접근자 메서드를 만들 때 모델의 이름(define할 때 사용된 첫번째 인자)을 활용하므로, hasOne을 위한 특별한 옵션값을 전달하는 것도 가능합니다. (하단 참조)

\*/

Project.hasOne(User, { as: 'Initiator' })

// 이제 Project.getInitiater와 Project.setInitiator를 사용할 수 있습니다.

**as 옵션은 연결 관계에서 사용할 별칭을 설정합니다.** 이 별칭은 Sequelize가 제공하는 JOIN 관련 유틸리티 메서드의 이름으로 활용됩니다. 이 메서드를 사용하면 연결 관계를 활용하여 데이터에 접근할 수 있습니다. 이 메서드들은 **source 모델 객체의 prototype** 에 생성됩니다. console.log()로 확인해보세요.

위의 예시의 경우, Project 모델의 prototype에 3개 메서드(get-, set-, create-)가 추가됩니다.  
위의 예시에서 project.setInitiator(user)를 실행하게 되면, 이것은 **project와 user 간에 형성된 연결 관계에서 user 인스턴스를 Initiator로 설정한다** 로 이해하면 됩니다. 그러면 project 인스턴스의 id가 user 인스턴스의 InitiatorId(또는 initiator\_id)에 대입됩니다.

직관적으로 바로 납득이 되지 않는 상황입니다. project에 대한 initiator는 user인데, user 인스턴스에 들어있는 InitiatorId는 project의 id를 가집니다. (??) 이것이 정 헷갈리면, foreignKey 설정을 사용해서 다른 이름으로 설정해주면 좀 낫겠죠? projectId 라든가...

### 더 알아보기: as 옵션과 has-, belongsTo 관계

더 나은 이해를 위해서 상황을 반대로 뒤집어보죠.

User.belongsTo(Project, { as: 'Initiator' })

user.setInitiator(project)

위의 코드에서는 User 모델에 InitiatorId 특성이 생성되고, 거기에 Project 인스턴스의 id가 들어갑니다. 그리고 **user 인스턴스의 Initiator는 project 인스턴스입니다.** 아직도 문맥이 이상하죠? 유틸리티 메서드도 관계가 뒤집혀져 있습니다.

그럼 이건 어떨까요?

Project.belongsTo(User, { as: 'Initiator' })

project.setInitiator(user)

여러 시행착오 끝에, 가장 자연스러운 상황이 되었네요. user는 project에 대한 Initiator로서 존재하게 되었습니다. project 인스턴스가 가진 InitiatorId에는 user 인스턴스의 id가 들어가게 됩니다. 마지막으로 project의 유틸리티 메서드를 사용하여 어색하지 않은 방법으로 관계를 설정해줄 수 있게 되었습니다.

### 결론

* as 별칭은 **target** 모델을 가리킨다.
* has- 를 사용할 경우, 접근 유틸리티 메서드의 결과가 **target** 인스턴스에 반영되고, target모델의 외래 키 이름이 별칭을 따른다.
* belongsTo-를 사용할 경우, 접근 유틸리티 메서드의 결과가 **source** 인스턴스에 반영되고, source 모델의 외래 키 이름이 별칭을 따른다.
* as로 인하여 컬럼 이름이 문맥과 어울리지 않고 헷갈리게 지어질 수 있으므로, **반드시 *foreignKey* 옵션과 함께 사용하자.**
* as 옵션은 has- 와는 가급적이면 사용하지 말자.

다행스러운 점은, 아래에 후술하겠지만 as가 자동으로 컬럼 이름을 짓는 일은 일대일 관계에서만 벌어집니다. 다대다 관계에서는 참조되는 테이블의 이름을 사용하여 짓습니다. 왜 이렇게 일관적이지 않은 동작을... ㅜㅜ

// 또는, 자기 자신과 연결 관계를 맺는 것도 가능합니다.

const Person = sequelize.define('person', { /\* ... \*/})

Person.hasOne(Person, {as: 'Father'})

// 여기서 FatherId라는 컬럼이 Person에 추가됩니다.

// 이것도 가능합니다. (하단 참조)

Person.hasOne(Person, {as: 'Father', foreignKey: 'DadId'})

// 여기서 DadId라는 컬럼이 Person에 추가됩니다.

// 두 경우 모두 아래 메서드를 사용할 수 있게 됩니다.

Person.setFather

Person.getFather

as 옵션과 foreignKey 옵션이 동시에 사용될 경우, as 옵션의 값은 유틸리티 메서드의 이름을 정하는 데에만 사용됩니다. 실제 DB 상에서 외래 키 컬럼의 이름은 foreignKey 옵션의 값이 사용됩니다.  
재귀 관계의 경우, set- 메서드를 썼을 때에 **어떤 인스턴스에 있는 외래 키 값을 설정하는 것인지** 헷갈릴 수 있습니다. 위에서 자세하게 다뤘듯, 연결 관계를 정의할 때에 belongsTo와 hasOne 중 어떤 것을 사용했는지에 따라 메커니즘이 달라지니, 둘 다 해보시고 그 차이를 확인해보세요.

// 테이블을 2번 Join해야 한다면, 동일 테이블에 대하여 2번 Join할 수도 있습니다.

Team.hasOne(Game, {as: 'HomeTeam', foreignKey : 'homeTeamId'});

Team.hasOne(Game, {as: 'AwayTeam', foreignKey : 'awayTeamId'});

Game.belongsTo(Team);

위의 예시에서는 Game 모델에 2개의 외래 키가 생성됩니다.  
외래 키의 이름 지정 규칙에 대하여 정리해볼게요.

(1) as만 사용

* [as에 사용된 테이블 별칭 + 참조하는 테이블의 주요 키 이름]

(2) as 유무와 상관 없이 foreignKey를 사용

* [foreignKey의 값]

위의 예시에서는 hasOne 연결 관계를 사용했지만, 대부분의 1:1 관계에서는 BelongsTo 관계를 사용하게 될 겁니다. 왜냐하면 BelongsTo는 source 모델(함수를 호출한 모델)에 외래 키를 추가하기 때문입니다. HasOne은 반대로 target 모델에 외래 키를 추가합니다.

제가 위에서 결론내린 것과 동일한 이야기를 도큐먼트에서도 하는군요! 그런데 이걸 이렇게 아무런 설명도 없이 적어주면... ㅜㅜ

### HasOne과 BelongsTo의 차이

Sequelize에서 일대일 관계는 HasOne과 BelongsTo를 사용하여 설정할 수 있습니다. 둘 다 각각에 적절한 상황이 있습니다. 예시를 통해 알아봅시다.

**Player** 와 **Team** 이라는 두 테이블을 연결하는 상황을 봅시다. 우선 모델을 정의합시다.

const Player = this.sequelize.define('player', {/\* attributes \*/})

const Team = this.sequelize.define('team', {/\* attributes \*/});

Sequelize에서 두 모델을 연결할 때에는 **source** 모델과 **target** 모델의 쌍으로 지칭할 수 있습니다. 아래와 같이 말이죠.

**Player** 가 **source** 이고, **Team** 이 **target** 일 때:

Player.belongsTo(Team);

// 또는

Player.hasOne(Team);

**Team** 이 **source** 이고, **Player** 가 **target** 일 때:

Team.belongsTo(Player)

// 또는

Team.hasOne(Player)

HasOne과 BelongsTo는 관계 키를 삽입하는 모델이 서로 다릅니다. HasOne은 관계 키를 **target**모델에 삽입하지만, BelongsTo는 관계 키를 **source** 모델에 삽입합니다.

BelongsTo와 HasOne을 사용하는 경우의 보여드리죠.

const Player = this.sequelize.define('player', {/\* attributes \*/})

const Coach = this.sequelize.define('coach', {/\* attributes \*/})

const Team = this.sequelize.define('team', {/\* attributes \*/});

Player 모델이 자신이 속한 팀의 정보를 teamId 컬럼으로 가진다고 가정해봅시다. 또한 각 팀의 Coach 에 대한 정보는 Team 모델에서 coachId 컬럼으로 저장됩니다. 앞의 두 경우는 각각 다른 종류의 일대일 관계를 필요로 하는데, 외래 키 관계가 여러 모델에서 동시에 발생하기 때문입니다.

관계 정보가 **source** 모델에 존재하는 경우 belongsTo를 사용하면 됩니다. 예시에서는 Player가 teamId를 가지므로, belongsTo를 사용하는 것이 적절합니다.

Player.belongsTo(Team) // 'teamId' 는 Player(source 모델)에 추가됩니다.

관계 정보가 **target** 모델에 존재하는 경우 hasOne을 사용하면 됩니다. 예시에서는 Team이 coachId를 통하여 Coach에 대한 정보를 가지므로, Coach는 hasOne을 사용하는 것이 적절합니다.

Coach.hasOne(Team) // 'coachId' 는 Team(target 모델)에 추가됩니다.

## 1:다 관계 (hasMany)

일대다 관계는 단일 source를 복수의 target 모델에 연결합니다. 이때 target 모델은 반드시 단 하나의 source 모델에만 연결됩니다.

const User = sequelize.define('user', {/\* ... \*/})

const Project = sequelize.define('project', {/\* ... \*/})

// 이제 조금 복잡해졌습니다. (사용자는 이를 전혀 모르겠지만요 :))

// 우선, hasMany 관계를 정의합니다.

Project.hasMany(User, {as: 'Workers'})

이렇게 하면, User 모델에 projectId 또는 project\_id 특성이 추가됩니다. 또한 Project 인스턴스는 접근 유틸리티 메서드인 getWorkers와 setWorkers를 가지게 됩니다.

일대일 관계 때와는 달리, target이 복수가 되면 as 옵션이 **외래 키 이름에 영향을 주지 않습니다.** 유틸리티 메서드 이름에만 영향을 끼칩니다.

레코드가 별도의 컬럼을 통하여 외래 키 관계를 형성할 수도 있는데, 이 때는 sourceKey 옵션을 사용합니다.

const City = sequelize.define('city', {

countryCode: Sequelize.STRING

});

const Country = sequelize.define('country', { isoCode: Sequelize.STRING });

// Country와 City를 국가 코드(isoCode)를 바탕으로 연결할 수 있습니다.

Country.hasMany(City, {foreignKey: 'countryCode', sourceKey: 'isoCode'});

City.belongsTo(Country, {foreignKey: 'countryCode', targetKey: 'isoCode'});

위에서 [Target Key](https://velog.io/@cadenzah/sequelize-document-4) 예제에서 사용한 것과 동일한 방식입니다. 물론 이 예제는 MySQL 기반의 DB를 사용할 경우 동일한 이유에 의하여 오류를 발생시킵니다. 제대로 작동하도록 하려면, Country 모델을 정의할 때에 isoCode를 주요 키로 설정해줘야 합니다.  
위의 예시에서 foreignKey에 지정해준 컬럼 이름과 모델 정의시 만든 컬럼의 이름이 겹칩니다(countryCode). 이런 경우, Sequelize는 외래 키를 위한 컬럼을 새로 만들지 않고 해당 컬럼을 외래 키 컬럼으로 지정해줍니다. 만약 foreignKey 옵션이 정의되지 않으면 countryCode와 별개의 새로운 컬럼이 생성됩니다.

지금까지 일방향 관계만 다뤘지만, 그 이상도 할 수 있습니다! 다음 섹션에서는 다:다 관계를 다룹니다.

## N:M 관계

다대다 관계는 다수의 source를 다수의 target과 연결할 때에 사용합니다. 또한 다수의 target은 다수의 source와 연결할 수 있습니다.

Project.belongsToMany(User, {through: 'UserProject'});

User.belongsToMany(Project, {through: 'UserProject'});

이렇게 하면, UserProject라는 모델이 생성되며, 여기에는 서로 동등한 자격의 외래 키 projectId와 userId가 생성됩니다. 컬럼이 camelcase인지 snail\_case인지는 두 모델의 설정에 따릅니다.

다대다 관계에서는 각 모델이 아니라 새로 생성된 모델에 특성이 추가된다는 점에 유의하세요.

여기서 through는 **필수 옵션** 입니다. 앞서 다룬 경우에서는 Sequelize가 알아서 모든 것을 처리해줬지만, 이것이 항상 논리적으로 가장 좋은 구성 방식은 아닙니다.

또한 이렇게 하고 나면 Project 객체에는 getUsers, setUsers, addUser, addUsers가 추가되고, User 객체에는 getProjects, setProjects, addProject, addProjects가 추가됩니다.

다대다 관계가 형성되면, 단수형 메서드와 복수형 메서드가 동시에 생성됩니다.

연결 관계 내에서 모델을 부를 때 원본과 다르게 부르고 싶을 수도 있죠? as 옵션을 사용해서 users를 workers로, projects를 tasks로 새로운 이름을 지어줄 수도 있습니다. 또한 직접 외래 키를 지정할 수도 있습니다.

User.belongsToMany(Project, { as: 'Tasks', through: 'worker\_tasks', foreignKey: 'userId' })

Project.belongsToMany(User, { as: 'Workers', through: 'worker\_tasks', foreignKey: 'projectId' })

foreignKey를 사용하면 **through** 로 정의된 새로운 모델 내에서 **source 모델** 을 가리키는 외래 키 컬럼의 이름을 지정해줄 수 있습니다. otherKey를 사용하면 **through** 로 정의된 새로운 모델 내에서 **target 모델** 을 가리키는 외래 키 컬럼의 이름을 지정해줄 수 있습니다.

// 아래 코드와 위의 코드는 동일한 결과를 만들어낸다

User.belongsToMany(Project, { as: 'Tasks', through: 'worker\_tasks', foreignKey: 'userId', otherKey: 'projectId'})

다대다 관계 정의에서는 **관계 설정하는 메서드를 호출하는 모델** 과 **through 모델** 을 중심으로 생각하면 됩니다. foreignKey가 가리키는 모델은 전자입니다. otherKey는 그 반대편의 모델(위의 코드에서는 Project a.k.a. Tasks)을 가리키는 것이겠죠?

belongsToMany로 재귀 관계를 정의할 수도 있습니다.

Person.belongsToMany(Person, { as: 'Children', through: 'PersonChildren' })

// PersonChildren 모델이 생성되고, Person 레코드의 id 2개로 이루어진 데이터를 갖게 된다

자기 참조를 하게 되면 through 테이블에 생기는 2개 외래 키를 구분할 수 있어야겠죠? 이때 as 옵션이 사용됩니다. 한 Person 인스턴스는 personId, Children 별칭의 인스턴스는 ChildId로 구분됩니다.

조인 테이블(through를 통하여 생성된 테이블)에 별도의 컬럼을 추가하고 싶다면, 관계를 정의하기 앞서서 조인 테이블을 먼저 정의하면 된다. 그 후 해당 테이블을 belongsToMany 관계 형성을 위하여 사용하면 된다.

const User = sequelize.define('user', {})

const Project = sequelize.define('project', {})

const UserProjects = sequelize.define('userProjects', {

status: DataTypes.STRING

})

User.belongsToMany(Project, { through: UserProjects })

Project.belongsToMany(User, { through: UserProjects })

through 키의 값이 '문자열'이 아니라 변수인 것에 주목해주세요.

user 인스턴스에 새로운 project 인스턴스를 추가하면서 status 값을 설정하려면, setter 함수(여기서는 addProject)에 option.through라는 추가적인 값을 전달하면 된다. 여기서 option.through는 through 테이블의 컬럼을 위한 값을 제공해준다.

// 각 테이블에서 user 레코드와 project 레코드를 찾고, 'started' status를 중심으로 연결해준다

user.addProject(project, { through: { status: 'started' }})

위의 코드를 실행하면, JOIN 연산에 기반하여 userProjects 테이블에 항목을 추가하는데, 이때 status 특성의 값도 'started'로 함께 변경해줍니다.

위의 코드는 기본값으로 projectId와 userId를 userProjects 테이블에 추가하되, 만약 이미 동일한 주요 키로 이루어진 레코드가 존재한다면 이를 덮어쓴다. UserProjects 테이블의 항목은 원본 테이블(이 예시에서는 User와 Project)의 주요 키 쌍으로서 구분되므로, 그 밖의 주요 키가 또 존재할 필요가 없기 때문이다. UserProjects 모델에 대하여 주요 키를 별도로 정의할 수도 있다.

const UserProjects = sequelize.define('userProjects', {

id: {

type: Sequelize.INTEGER,

primaryKey: true,

autoIncrement: true

},

status: DataTypes.STRING

})

원래는 원본 테이블의 두 주요 키의 조합이 userProjects의 주요 키였지만, 이제는 id가 별도로 정의되었으므로 두 주요 키 조합은 외래 키의 역할을 갖게 됩니다. 이제는 두 주요 키 조합의 값이 새로 추가되더라도 덮어쓰기 되는 일 없이 새로운 항목이 추가될 겁니다.

belongsToMany 관계에서는 **through 테이블** 에 기반한 쿼리로 특정 컬럼값만을 취할 수도 있습니다. 아래의 예에서는 through와 함께 findAll를 사용합니다.

User.findAll({

include: [{

model: Project,

through: {

attributes: ['createdAt', 'startedAt', 'finishedAt'],

where: {completed: true}

}

}]

});

N:M 관계인 User와 Project, 그리고 그 가운데 through 테이블인 userProjects까지 3개의 테이블에 대한 쿼리입니다. 여러 가지 실습을 해볼 수 있도록 코드를 재작성해봤습니다.

const User = sequelize.define('user',{})

const Project = sequelize.define('project')

User.belongsToMany(Project, { through: 'userProjects'})

Project.belongsToMany(User, { through: 'userProjects'})

sequelize.sync()

.then(() => {

return User.findAll({

include: [{

model: Project,

through: {

attributes: ['createdAt']

}}]})})

.then((users) => {

users.map((user) => {

console.log(JSON.stringify(user))

console.log()

})})

/\*

{

"id":2,

"createdAt":"2019-01-06T08:52:23.000Z",

"updatedAt":"2019-01-06T08:52:23.000Z",

"projects": [

{

"id":1,

"createdAt":"2019-01-06T08:52:32.000Z",

"updatedAt":"2019-01-06T08:52:32.000Z",

"userProjects": {

"createdAt":"2019-01-06T08:54:20.000Z"

}

}

]

}

\*/

through 속성 내에서 정의된 내용은 through 테이블의 데이터에만 영향을 주었다는 점을 주목해주세요. 즉, attributes 속성에는 'createdAt' 만이 포함되었는데, JSON 결과를 보면 다른 데이터와는 달리 "userProjects" 에 대한 데이터는 "createdAt"만이 포함되어있다는 것을 알 수 있습니다. 그 외에도 옵션값을 수정하며 다양하게 실습하며 개념을 확인해보세요.

// 또다른 예시 쿼리

User.findAll({

attributes: ['id'],

where: {

id: 2

},

include: [{

model: Project,

attributes: ['updatedAt'],

through: {

attributes: ['createdAt']

}

}]

})

through 테이블에 주요 키가 존재하지 않을 경우, 고유 키(unique key)가 자동으로 생성됩니다. 고유 키의 이름은 uniqueKey 옵션으로 지정해줄 수 있습니다.

Project.belongsToMany(User, { through: UserProjects, uniqueKey: 'my\_custom\_unique' })

## 코멘트

며칠동안 저를 고민에 빠트렸던 as와 연결 관계에 대한 수수께끼가 점점 풀려가고 있습니다! 이제는 고구마먹은 것마냥 답답한 심정 없이 Sequelize를 쓸 수 있게 되었...겠죠? 대체 왜 이런 것들을 도큐먼트는 잘 알려주지 않는 걸까요... 이렇게 스스로 깨우치라는 개발자들의 깊은 통찰인 걸까요? 😂😂

다음 편은 **7. Association (하)**로 이어집니다.

[ORM](https://velog.io/tags/ORM)[Sequelize](https://velog.io/tags/Sequelize)[db](https://velog.io/tags/db)[javascript](https://velog.io/tags/javascript)[mysql](https://velog.io/tags/mysql)[sql](https://velog.io/tags/sql)[시퀄라이즈](https://velog.io/tags/%EC%8B%9C%ED%80%84%EB%9D%BC%EC%9D%B4%EC%A6%88)[자바스크립트](https://velog.io/tags/%EC%9E%90%EB%B0%94%EC%8A%A4%ED%81%AC%EB%A6%BD%ED%8A%B8)

### 0개의 댓글