Version: v6 - stable

# Eager Loading

As briefly mentioned in the associations guide, eager Loading is the act of querying data of several models at once (one 'main' model and one or more associated models). At the SQL level, this is a query with one or more joins.

When this is done, the associated models will be added by Sequelize in appropriately named, automatically created field(s) in the returned objects.

In Sequelize, eager loading is mainly done by using the `include` option on a model finder query (such as `findOne`, `findAll`, etc).

## Basic example

Let's assume the following setup:

```
const User = sequelize.define('user', { name: DataTypes.STRING }, { timestamps: false });
const Task = sequelize.define('task', { name: DataTypes.STRING }, { timestamps: false });
const Tool = sequelize.define('tool', {
  name: DataTypes.STRING,
  size: DataTypes.STRING
}, { timestamps: false });
User.hasMany(Task);
Task.belongsTo(User);
User.hasMany(Tool, { as: 'Instruments' });
```

### Fetching a single associated element

OK. So, first of all, let's load all tasks with their associated user:

```
const tasks = await Task.findAll({ include: User });
console.log(JSON.stringify(tasks, null, 2));
```

Output:

```
[{
  "name": "A Task",
  "id": 1,
  "userId": 1,
  "user": {
    "name": "John Doe",
    "id": 1
  }
}]
```

Here, `tasks[0].user instanceof User` is `true`. This shows that when Sequelize fetches associated models, they are added to the output object as model instances.

Above, the associated model was added to a new field called `user` in the fetched task. The name of this field was automatically chosen by Sequelize based on the name of the associated model, where its pluralized form is used when applicable (i.e., when the association is `hasMany` or `belongsToMany`). In other words, since `Task.belongsTo(User)`, a task is associated to one user, therefore the logical choice is the singular form (which Sequelize follows automatically).

### Fetching all associated elements

Now, instead of loading the user that is associated to a given task, we will do the opposite - we will find all tasks associated to a given user.

The method call is essentially the same. The only difference is that now the extra field created in the query result uses the pluralized form (`tasks` in this case), and its value is an array of task instances (instead of a single instance, as above).

```
const users = await User.findAll({ include: Task });
console.log(JSON.stringify(users, null, 2));
```

Output:

```
[{
  "name": "John Doe",
  "id": 1,
  "tasks": [{
    "name": "A Task",
    "id": 1,
    "userId": 1
  }]
}]
```

Notice that the accessor (the `tasks` property in the resulting instance) is pluralized since the association is one-to-many.

### Fetching an Aliased association

If an association is aliased (using the `as` option), you must specify this alias when including the model. Instead of passing the model directly to the `include` option, you should instead provide an object with two options: `model` and `as`.

Notice how the user's `Tool`s are aliased as `Instruments` above. In order to get that right you have to specify the model you want to load, as well as the alias:

```
const users = await User.findAll({
  include: { model: Tool, as: 'Instruments' }
});
console.log(JSON.stringify(users, null, 2));
```

Output:

```
[{
  "name": "John Doe",
  "id": 1,
  "Instruments": [{
    "name": "Scissor",
    "id": 1,
    "userId": 1
  }]
}]
```

You can also include by alias name by specifying a string that matches the association alias:

```
User.findAll({ include: 'Instruments' }); // Also works
User.findAll({ include: { association: 'Instruments' } }); // Also works
```

## Required eager loading

When eager loading, we can force the query to return only records which have an associated model, effectively converting the query from the default `OUTER JOIN` to an `INNER JOIN`. This is done with the `required: true` option, as follows:

```
User.findAll({
  include: {
    model: Task,
    required: true
  }
});
```

This option also works on nested includes.

## Eager loading filtered at the associated model level

When eager loading, we can also filter the associated model using the `where` option, as in the following example:

```
User.findAll({
  include: {
    model: Tool,
    as: 'Instruments'
    where: {
      size: {
        [Op.ne]: 'small'
      }
    }
  }
});
```

Generated SQL:

```
SELECT
  `user`.`id`,
  `user`.`name`,
  `Instruments`.`id` AS `Instruments.id`,
  `Instruments`.`name` AS `Instruments.name`,
  `Instruments`.`size` AS `Instruments.size`,
  `Instruments`.`userId` AS `Instruments.userId`
FROM `users` AS `user`
INNER JOIN `tools` AS `Instruments` ON
  `user`.`id` = `Instruments`.`userId` AND
  `Instruments`.`size` != 'small';
```

Note that the SQL query generated above will only fetch users that have at least one tool that matches the condition (of not being `small`, in this case). This is the case because, when the `where` option is used inside an `include`, Sequelize automatically sets the `required` option to `true`. This means that, instead of an `OUTER JOIN`, an `INNER JOIN` is done, returning only the parent models with at least one matching children.

Note also that the `where` option used was converted into a condition for the `ON` clause of the `INNER JOIN`. In order to obtain a *top-level* `WHERE` clause, instead of an `ON` clause, something different must be done. This will be shown next.

#### Referring to other columns

If you want to apply a `WHERE` clause in an included model referring to a value from an associated model, you can simply use the `Sequelize.col` function, as show in the example below:

```
// Find all projects with a least one task where task.state === project.state
Project.findAll({
  include: {
    model: Task,
    where: {
      state: Sequelize.col('project.state')
    }
  }
})
```

## Complex where clauses at the top-level

To obtain top-level `WHERE` clauses that involve nested columns, Sequelize provides a way to reference nested columns: the `'$nested.column$'` syntax.

It can be used, for example, to move the where conditions from an included model from the `ON` condition to a top-level `WHERE` clause.

```
User.findAll({
  where: {
    '$Instruments.size$': { [Op.ne]: 'small' }
  },
  include: [{
    model: Tool,
    as: 'Instruments'
  }]
});
```

Generated SQL:

```
SELECT
  `user`.`id`,
  `user`.`name`,
  `Instruments`.`id` AS `Instruments.id`,
  `Instruments`.`name` AS `Instruments.name`,
  `Instruments`.`size` AS `Instruments.size`,
  `Instruments`.`userId` AS `Instruments.userId`
FROM `users` AS `user`
LEFT OUTER JOIN `tools` AS `Instruments` ON
  `user`.`id` = `Instruments`.`userId`
WHERE `Instruments`.`size` != 'small';
```

The `$nested.column$` syntax also works for columns that are nested several levels deep, such as `$some.super.deeply.nested.column$`. Therefore, you can use this to make complex filters on deeply nested columns.

For a better understanding of all differences between the inner `where` option (used inside an `include`), with and without the `required` option, and a top-level `where` using the `$nested.column$` syntax, below we have four examples for you:

```
// Inner where, with default `required: true`
await User.findAll({
  include: {
    model: Tool,
    as: 'Instruments',
    where: {
      size: { [Op.ne]: 'small' }
    }
  }
});

// Inner where, `required: false`
await User.findAll({
  include: {
    model: Tool,
    as: 'Instruments',
    where: {
      size: { [Op.ne]: 'small' }
    },
    required: false
  }
});

// Top-level where, with default `required: false`
await User.findAll({
  where: {
    '$Instruments.size$': { [Op.ne]: 'small' }
  },
  include: {
    model: Tool,
    as: 'Instruments'
  }
});

// Top-level where, `required: true`
await User.findAll({
  where: {
    '$Instruments.size$': { [Op.ne]: 'small' }
  },
  include: {
    model: Tool,
    as: 'Instruments',
    required: true
  }
});
```

Generated SQLs, in order:

```
-- Inner where, with default `required: true`
SELECT [...] FROM `users` AS `user`
INNER JOIN `tools` AS `Instruments` ON
  `user`.`id` = `Instruments`.`userId`
  AND `Instruments`.`size` != 'small';

-- Inner where, `required: false`
SELECT [...] FROM `users` AS `user`
LEFT OUTER JOIN `tools` AS `Instruments` ON
  `user`.`id` = `Instruments`.`userId`
  AND `Instruments`.`size` != 'small';

-- Top-level where, with default `required: false`
SELECT [...] FROM `users` AS `user`
LEFT OUTER JOIN `tools` AS `Instruments` ON
  `user`.`id` = `Instruments`.`userId`
WHERE `Instruments`.`size` != 'small';

-- Top-level where, `required: true`
SELECT [...] FROM `users` AS `user`
INNER JOIN `tools` AS `Instruments` ON
  `user`.`id` = `Instruments`.`userId`
WHERE `Instruments`.`size` != 'small';
```

### Fetching with `RIGHT OUTER JOIN` (MySQL, MariaDB, PostgreSQL and MSSQL only)

By default, associations are loaded using a `LEFT OUTER JOIN` - that is to say it only includes records from the parent table. You can change this behavior to a `RIGHT OUTER JOIN` by passing the `right` option, if the dialect you are using supports it.

Currently, SQLite does not support right joins.

*Note:* `right` is only respected if `required` is false.

```
User.findAll({
  include: [{
    model: Task // will create a left join
  }]
});
User.findAll({
```

```
  include: [{
    model: Task,
    right: true // will create a right join
  }]
});
User.findAll({
  include: [{
    model: Task,
    required: true,
    right: true // has no effect, will create an inner join
  }]
});
User.findAll({
  include: [{
    model: Task,
    where: { name: { [Op.ne]: 'empty trash' } },
    right: true // has no effect, will create an inner join
  }]
});
User.findAll({
  include: [{
    model: Tool,
    where: { name: { [Op.ne]: 'empty trash' } },
    required: false // will create a left join
  }]
});
User.findAll({
  include: [{
    model: Tool,
    where: { name: { [Op.ne]: 'empty trash' } },
    required: false
    right: true // will create a right join
  }]
});
```

## Multiple eager loading

The `include` option can receive an array in order to fetch multiple associated models at once:

```
Foo.findAll({
  include: [
    {
      model: Bar,
      required: true
    },
    {
      model: Baz,
      where: /* ... */
    },
    Qux // Shorthand syntax for { model: Qux } also works here
  ]
})
```

## Eager loading with Many-to-Many relationships

When you perform eager loading on a model with a Belongs-to-Many relationship, Sequelize will fetch the junction table data as well, by default. For example:

```
const Foo = sequelize.define('Foo', { name: DataTypes.TEXT });
const Bar = sequelize.define('Bar', { name: DataTypes.TEXT });
Foo.belongsToMany(Bar, { through: 'Foo_Bar' });
Bar.belongsToMany(Foo, { through: 'Foo_Bar' });

await sequelize.sync();
const foo = await Foo.create({ name: 'foo' });
const bar = await Bar.create({ name: 'bar' });
await foo.addBar(bar);
const fetchedFoo = await Foo.findOne({ include: Bar });
console.log(JSON.stringify(fetchedFoo, null, 2));
```

Output:

```
{
  "id": 1,
  "name": "foo",
  "Bars": [
    {
      "id": 1,
      "name": "bar",
      "Foo_Bar": {
        "FooId": 1,
        "BarId": 1
      }
    }
  ]
}
```

Note that every bar instance eager loaded into the `"Bars"` property has an extra property called `Foo_Bar` which is the relevant Sequelize instance of the junction model. By default, Sequelize fetches all attributes from the junction table in order to build this extra property.

However, you can specify which attributes you want fetched. This is done with the `attributes` option applied inside the `through` option of the include. For example:

```
Foo.findAll({
  include: [{
    model: Bar,
    through: {
      attributes: [/* list the wanted attributes here */]
    }
  }]
});
```

If you don't want anything from the junction table, you can explicitly provide an empty array to the `attributes` option inside the `through` option of the `include` option, and in this case nothing will be fetched and the extra property will not even be created:

through option of the include option, and in this case nothing will be fetched and the extra property will not even be created.

```
Foo.findOne({
  include: {
    model: Bar,
    through: {
      attributes: []
    }
  }
});
```

Output:

```
{
  "id": 1,
  "name": "foo",
  "Bars": [
    {
      "id": 1,
      "name": "bar"
    }
  ]
}
```

Whenever including a model from a Many-to-Many relationship, you can also apply a filter on the junction table. This is done with the `where` option applied inside the `through` option of the include. For example:

```
User.findAll({
  include: [{
    model: Project,
    through: {
      where: {
        // Here, `completed` is a column present at the junction table
        completed: true
      }
    }
  }]
});
```

Generated SQL (using SQLite):

```
SELECT
  `User`.`id`,
  `User`.`name`,
  `Projects`.`id` AS `Projects.id`,
  `Projects`.`name` AS `Projects.name`,
  `Projects->User_Project`.`completed` AS `Projects.User_Project.completed`,
  `Projects->User_Project`.`UserId` AS `Projects.User_Project.UserId`,
  `Projects->User_Project`.`ProjectId` AS `Projects.User_Project.ProjectId`
FROM `Users` AS `User`
LEFT OUTER JOIN `User_Projects` AS `Projects->User_Project` ON
  `User`.`id` = `Projects->User_Project`.`UserId`
LEFT OUTER JOIN `Projects` AS `Projects` ON
  `Projects`.`id` = `Projects->User_Project`.`ProjectId` AND
  `Projects->User_Project`.`completed` = 1;
```

## Including everything

To include all associated models, you can use the `all` and `nested` options:

```
// Fetch all models associated with User
User.findAll({ include: { all: true }});

// Fetch all models associated with User and their nested associations (recursively)
User.findAll({ include: { all: true, nested: true }});
```

## Including soft deleted records

In case you want to eager load soft deleted records you can do that by setting `include.paranoid` to `false`:

```
User.findAll({
  include: [{
    model: Tool,
    as: 'Instruments',
    where: { size: { [Op.ne]: 'small' } },
    paranoid: false
  }]
});
```

## Ordering eager loaded associations

When you want to apply `ORDER` clauses to eager loaded models, you must use the top-level `order` option with augmented arrays, starting with the specification of the nested model you want to sort.

This is better understood with examples.

```
Company.findAll({
  include: Division,
  order: [
    // We start the order array with the model we want to sort
    [Division, 'name', 'ASC']
  ]
});
Company.findAll({
  include: Division,
  order: [
    [Division, 'name', 'DESC']
  ]
});
Company.findAll({
```

```
  // If the include uses an alias...
  include: { model: Division, as: 'Div' },
  order: [
    // ...we use the same syntax from the include
    // in the beginning of the order array
    [{ model: Division, as: 'Div' }, 'name', 'DESC']
  ]
});

Company.findAll({
  // If we have includes nested in several levels...
  include: {
    model: Division,
    include: Department
  },
  order: [
    // ... we replicate the include chain of interest
    // at the beginning of the order array
    [Division, Department, 'name', 'DESC']
  ]
});
```

In the case of many-to-many relationships, you are also able to sort by attributes in the through table. For example, assuming we have a Many-to-Many relationship between `Division` and `Department` whose junction model is `DepartmentDivision`, you can do:

```
Company.findAll({
  include: {
    model: Division,
    include: Department
  },
  order: [
    [Division, DepartmentDivision, 'name', 'ASC']
  ]
});
```

In all the above examples, you have noticed that the `order` option is used at the top-level. The only situation in which `order` also works inside the include option is when `separate: true` is used. In that case, the usage is as follows:

```
// This only works for `separate: true` (which in turn
// only works for has-many relationships).
User.findAll({
  include: {
    model: Post,
    separate: true,
    order: [
      ['createdAt', 'DESC']
    ]
  }
});
```

### Complex ordering involving sub-queries

Take a look at the guide on sub-queries for an example of how to use a sub-query to assist a more complex ordering.

## Nested eager loading

You can use nested eager loading to load all related models of a related model:

```
const users = await User.findAll({
  include: {
    model: Tool,
    as: 'Instruments',
    include: {
      model: Teacher,
      include: [ /* etc */ ]
    }
  }
});
console.log(JSON.stringify(users, null, 2));
```

Output:

```
[{
  "name": "John Doe",
  "id": 1,
  "Instruments": [{ // 1:M and N:M association
    "name": "Scissor",
    "id": 1,
    "userId": 1,
    "Teacher": { // 1:1 association
      "name": "Jimi Hendrix"
    }
  }]
}]
```

This will produce an outer join. However, a `where` clause on a related model will create an inner join and return only the instances that have matching sub-models. To return all parent instances, you should add `required: false`.

```
User.findAll({
  include: [{
    model: Tool,
    as: 'Instruments',
    include: [{
      model: Teacher,
      where: {
        school: "Woodstock Music School"
      },
      required: false
    }]
  }]
});
```

The query above will return all users, and all their instruments, but only those teachers associated with `Woodstock Music School`.

## Using `findAndCountAll` with includes

The `findAndCountAll` utility function supports includes. Only the includes that are marked as `required` will be considered in `count`. For example, if you want to find and count all users who have a profile:

```
User.findAndCountAll({
  include: [
    { model: Profile, required: true }
  ],
  limit: 3
});
```

Because the include for `Profile` has `required` set it will result in an inner join, and only the users who have a profile will be counted. If we remove `required` from the include, both users with and without profiles will be counted. Adding a `where` clause to the include automatically makes it required:

```
User.findAndCountAll({
  include: [
    { model: Profile, where: { active: true } }
  ],
  limit: 3
});
```

The query above will only count users who have an active profile, because `required` is implicitly set to true when you add a where clause to the include.

Edit this page

*Last updated on **Oct 13, 2022** by **Rik Smale***

Previous
« Advanced Association Concepts

Next
Creating with Associations »