Model usage

Data retrieval / Finders

Finder methods are intended to query data from the database. They do *not* return plain objects but instead return model instances. Because finder methods return model instances you can call any model instance member on the result as described in the documentation for *instances*.

In this document we'll explore what finder methods can do:

`find` - Search for one specific element in the database

```
// search for known ids
Project.findById(123).then(project => {
  // project will be an instance of Project and
stores the content of the table entry
  // with id 123. if such an entry is not
defined you will get null
})

// search for attributes
Project.findOne({ where: {title: 'aProject'}
}).then(project => {
  // project will be the first entry of the
Projects table with the title 'aProject' ||
null
})


Project.findOne({
  where: {title: 'aProject'},
  attributes: ['id', ['name', 'title']]
}).then(project => {
  // project will be the first entry of the
Projects table with the title 'aProject' ||
null
  // project.title will contain the name of the
project
})
```

`findOrCreate` - Search for a specific element or create it if not

The method `findOrCreate` can be used to check if a certain element already exists in the database. If that is the case the method will result in a respective instance. If the element does not yet exist, it will be created.

Let's assume we have an empty database with a `User` model which has a `username` and a `job`.

```
User
  .findOrCreate({where: {username: 'sdepold'},
defaults: {job: 'Technical Lead JavaScript'}})
  .spread((user, created) => {
    console.log(user.get({
      plain: true
    }))
    console.log(created)

    /*
     findOrCreate returns an array containing
the object that was found or created and a
boolean that will be true if a new object was
created and false if not, like so:

    [ {
        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 ]

 In the example above, the "spread" on line 39
divides the array into its 2 parts and passes
them as arguments to the callback function
defined beginning at line 39, which treats them
as "user" and "created" in this case. (So
"user" will be the object from index 0 of the
returned array and "created" will equal
"true".)
    */
  })
```

The code created a new instance. So when we already have an instance ...

```
User.create({ username: 'fnord', job:
'omnomnom' })
  .then(() => User.findOrCreate({where:
{username: 'fnord'}, defaults: {job: 'something
else'}}))
  .spread((user, created) => {
    console.log(user.get({
      plain: true
    }))
    console.log(created)
```

```
    In this example, findOrCreate returns an
array like this:
    [ {
        username: 'fnord',
        job: 'omnomnom',
        id: 2,
        createdAt: Fri Mar 22 2013 21: 28: 34
GMT + 0100(CET),
        updatedAt: Fri Mar 22 2013 21: 28: 34
GMT + 0100(CET)
      },
      false
    ]
    The array returned by findOrCreate gets
spread into its 2 parts by the "spread" on line
69, and the parts will be passed as 2 arguments
to the callback function beginning on line 69,
which will then treat them as "user" and
"created" in this case. (So "user" will be the
object from index 0 of the returned array and
"created" will equal "false".)
    */
  })
```

... the existing entry will not be changed. See the `job` of the second user, and the fact that created was false.

`findAndCountAll` - Search for multiple elements in the database, returns both data and total count

This is a convenience method that combines `findAll` and `count` (see below) this is useful when dealing with queries related to pagination where you want to retrieve data with a `limit` and `offset` but also need to know the total number of records that match the query:

The success handler will always receive an object with two properties:

- `count` - an integer, total number records matching the where clause and other filters due to associations
- `rows` - an array of objects, the records matching the where clause and other filters due to associations, within the limit and offset range

```
Project
  .findAndCountAll({
     where: {
        title: {
          [Op.like]: 'foo%'
        }
     },
```

```
  });
  .then(result => {
    console.log(result.count);
    console.log(result.rows);
  });
```

It support includes. Only the includes that are marked as `required` will be added to the count part:

Suppose you want to find all users who have a profile attached:

```
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.

The options object that you pass to `findAndCountAll` is the same as for `findAll` (described below).

`findAll` - Search for multiple elements in the database

```
// find multiple entries
Project.findAll().then(projects => {
  // projects will be an array of all Project
instances
})

// also possible:
Project.all().then(projects => {
  // projects will be an array of all Project
```

```
// search for specific attributes - hash usage
Project.findAll({ where: { name: 'A Project' }
}).then(projects => {
  // projects will be an array of Project
instances with the specified name
})

// search within a specific range
Project.findAll({ where: { id: [1,2,3] }
}).then(projects => {
  // projects will be an array of Projects
having the id 1, 2 or 3
  // this is actually doing an IN query
})

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
    }
  }
})
```

Complex filtering / OR / NOT queries

that you can use `or`, `and` or `not` Operators:

```
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 }
      ]
    }
  }
})
```

Both pieces of code will generate the following:

```
SELECT *
FROM `Projects`
WHERE (
  `Projects`.`name` = 'a project'
   AND (`Projects`.`id` IN (1,2,3) OR
`Projects`.`id` > 10)
)
LIMIT 1;
```

`not` example:

```
Project.findOne({
  where: {
    name: 'a project',
    [Op.not]: [
      { id: [1,2,3] },
      { array: { [Op.contains]: [3,4,5] } }
    ]
  }
});
```

Will generate:

```
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;
```

offset, order and group

To get more relevant data, you can use limit, offset, order and grouping:

```
// limit the results of the query
Project.findAll({ limit: 10 })

// step over the first 10 elements
Project.findAll({ offset: 10 })

// step over the first 10 elements, and take 2
Project.findAll({ offset: 10, limit: 2 })
```

The syntax for grouping and ordering are equal, so below it is only explained with a single example for group, and the rest for order. Everything you see below can also be done for group

```
Project.findAll({order: 'title DESC'})
// yields ORDER BY title DESC

Project.findAll({group: 'name'})
// yields GROUP BY name
```

Notice how in the two examples above, the string provided is inserted verbatim into the query, i.e. column names are not escaped. When you provide a string to order/group, this will always be the case. If you want to escape column names, you should provide an array of arguments, even though you only want to order/group by a single column

```
something.findOne({
  order: [
    // will return `name`
    ['name'],
    // will return `username` DESC
    ['username', 'DESC'],
    // will return max(`age`)
    sequelize.fn('max', sequelize.col('age')),
    // will return max(`age`) DESC
    [sequelize.fn('max', sequelize.col('age')),
'DESC'],
    // will return otherfunction(`col1`, 12,
'lalala') DESC
    [sequelize.fn('otherfunction',
sequelize.col('col1'), 12, 'lalala'), 'DESC'],
    // will return
otherfunction(awesomefunction(`col`)) DESC,
This nesting is potentially infinite!
    [sequelize.fn('otherfunction',
sequelize.fn('awesomefunction',
sequelize.col('col'))), 'DESC']
  ]
})
```

following:

- String - will be quoted
- Array - first element will be quoted, second will be appended verbatim
- Object -
  - Raw will be added verbatim without quoting
  - Everything else is ignored, and if raw is not set, the query will fail
- Sequelize.fn and Sequelize.col returns functions and quoted column names

Raw queries

Sometimes you might be expecting a massive dataset that you just want to display, without manipulation. For each row you select, Sequelize creates an instance with functions for update, delete, get associations etc. If you have thousands of rows, this might take some time. If you only need the raw data and don't want to update anything, you can do like this to get the raw data.

```
// Are you expecting a massive dataset from the DB,
// and don't want to spend the time building DAOs for each entry?
// You can pass an extra query option to get the raw data instead:
Project.findAll({ where: { ... }, raw: true })
```

`count` - Count the occurrences of elements in the database

There is also a method for counting database objects:

```
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` - Get the greatest value of a specific attribute within a specific

And here is a method for getting the max value of an attribute:f

```
/*
  Let's assume 3 person objects with an
attribute age.
  The first one is 10 years old,
  the second one is 5 years old,
  the third one is 40 years old.
*/
Project.max('age').then(max => {
  // this will return 40
})

Project.max('age', { where: { age: { [Op.lt]:
20 } } }).then(max => {
  // will be 10
})
```

`min` - Get the least value of a specific attribute within a specific table

And here is a method for getting the min value of an attribute:

```
/*
  Let's assume 3 person objects with an
attribute age.
  The first one is 10 years old,
  the second one is 5 years old,
  the third one is 40 years old.
*/
Project.min('age').then(min => {
  // this will return 5
})

Project.min('age', { where: { age: { [Op.gt]: 5
} } }).then(min => {
  // will be 10
})
```

`sum` - Sum the value of specific attributes

In order to calculate the sum over a specific column of a table, you can use the `sum` method.

```
/*
  Let's assume 3 person objects with an
attribute age.
  The first one is 10 years old,
  the second one is 5 years old,
  the third one is 40 years old.
```

```
  // this will return 55
})

Project.sum('age', { where: { age: { [Op.gt]: 5
} } }).then(sum => {
  // will be 50
})
```

Eager loading

When you are retrieving data from the database there is a fair chance that you also want to get associations with the same query - this is called eager loading. The basic idea behind that, is the use of the attribute `include` when you are calling `find` or `findAll`. Lets assume the following setup:

```
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(() => {
  // this is where we continue ...
})
```

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

```
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"
      }
    }]
  */
})
```

resulting instance) is singular because the association is one-to-something.

Next thing: Loading of data with many-to-something associations!

```
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
      }]
    }]
  */
})
```

Notice that the accessor (the `Tasks` property in the resulting instance) is plural because the association is many-to-something.

If an association is aliased (using the `as` option), you must specify this alias when including the model. 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:

```
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": [{
        "name": "Toothpick",
        "id": 1,
        "createdAt": null,
        "updatedAt": null,
        "userId": 1
      }]
```

```
});
```

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

```
User.findAll({ include: ['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
      }]
    }]
  */
})

User.findAll({ include: [{ association:
'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
      }]
    }]
  */
})
```

When eager loading we can also filter the associated model using `where`. This will return all `User`s in which the `where` clause of `Tool` model matches rows.

```
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
        }]
      }],
    */
  })
```

When an eager loaded model is filtered using `include.where` then `include.required` is implicitly set to `true`. This means that an inner join is done returning parent models with any matching children.

Top level where with eagerly loaded models

To move the where conditions from an included model from the `ON` condition to the top level `WHERE` you can use the `'$nested.column$'` syntax:

```
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
        }]
      }],
    */
```

Including everything

To include all attributes, you can pass a single object with `all: true`:

```
User.findAll({ include: [{ all: 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,
        where: { name: { [Op.like]: '%ooth%' }
},
        paranoid: false // query and loads the
soft deleted records
    }]
});
```

In the case of a one-to-many relationship.

```
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' ] ]
});
```

In the case of many-to-many joins, you are also able to sort by attributes in the through table.

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

Nested eager loading

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

```
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,
```

```
          "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
    }]
  }]
}).then(users => {
  /* ... */
})
```

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

Include all also supports nested loading:

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