Desenvolvimento de Serviços e APIs

Faculdade Senac Pelotas

Escola de Tecnologia da Informação

Prof. Edécio Fernando Iepsen

Associações

O Sequelize suporta as associações padrão: One-To-One , One-To-Many e Many-To-Many .

Para fazer isso, o Sequelize fornece **quatro** tipos de associações que devem ser combinadas para criá-las:

- a `HasOne` associação
- a `BelongsTo` associação
- a `HasMany` associação
- a `BelongsToMany` associação

O guia começará explicando como definir esses quatro tipos de associações e, em seguida, explicará como combiná-los para definir os três tipos de associação padrão (Um-para-um , Um-para-muitos e Muitos-para-muitos).

Definindo as #associações

Os quatro tipos de associação são definidos de maneira muito semelhante. Digamos que temos dois modelos, `A` e `B`. Dizer ao Sequelize que você deseja uma associação entre os dois precisa apenas de uma chamada de função:

```
const A = sequelize.define('A', /* ... */);
const B = sequelize.define('B', /* ... */);

A.hasOne(B); // A HasOne B
A.belongsTo(B); // A BelongsTo B
A.hasMany(B); // A HasMany B
A.belongsToMany(B, { through: 'C' }); // A BelongsToMany B through the junction table C
```

Todos eles aceitam um objeto de opções como segundo parâmetro (opcional para os três primeiros, obrigatório por `belongsToMany` conter pelo menos a `through` propriedade):

```
A.hasOne(B, { /* options */ });
A.belongsTo(B, { /* options */ });
A.hasMany(B, { /* options */ });
A.belongsToMany(B, { through: 'C', /* options */ });
```

A ordem em que a associação é definida é relevante. Em outras palavras, a ordem importa, para os quatro casos. Em todos os exemplos acima, `A` é chamado de modelo **de origem** e `B` é chamado de modelo **de destino** . Esta terminologia é importante.

A `A.hasOne(B)` associação significa que existe um relacionamento One-To-One entre `A` e `B`, com a chave estrangeira sendo definida no modelo de destino (`B`).

A `A.belongsTo(B)` associação significa que existe um relacionamento One-To-One entre `A` e `B`, com a chave estrangeira sendo definida no modelo de origem (`A`).

A `A.hasMany(B)` associação significa que existe um relacionamento One-To-Many entre `A` e `B`, com a chave estrangeira sendo definida no modelo de destino (`B`).

Essas três chamadas farão com que o Sequelize adicione automaticamente chaves estrangeiras aos modelos apropriados (a menos que já estejam presentes).

A `A.belongsToMany(B, { through: 'C' })` associação significa que existe uma relação Muitos-para-Muitos entre `A` e `B`, usando tabela `C` como tabela de junção , que terá as chaves estrangeiras (`aId` e `bId`, por exemplo). O Sequelize criará automaticamente este modelo `C` (a menos que já exista) e definirá as chaves estrangeiras apropriadas nele.

Nota: Nos exemplos acima para `belongsToMany`, uma string (`'C'`) foi passada para a opção through. Neste caso, o Sequelize gera automaticamente um modelo com este nome. No entanto, você também pode passar um modelo diretamente, se já o tiver definido.

Estas são as principais ideias envolvidas em cada tipo de associação. No entanto, esses relacionamentos geralmente são usados em pares, para permitir um melhor uso com o Sequelize. Isso será visto mais adiante.

Várias opções podem ser passadas como um segundo parâmetro da chamada de associação.

`onDelete` `onUpdate`

Por exemplo, para configurar os comportamentos `ON DELETE` e `ON UPDATE`, você pode fazer:

```
Foo.hasOne(Bar, {
  onDelete: 'RESTRICT',
  onUpdate: 'RESTRICT'
});
Bar.belongsTo(Foo);
```

As opções possíveis são `RESTRICT`, `CASCADE`, e .`NO ACTION` `SET DEFAULT` `SET NULL`

Os padrões para as associações One-To-One são `SET NULL` for `ON DELETE` e `CASCADE` for `ON UPDATE`.

Personalizando a #chave

Ambas as chamadas `hasOne` e `belongsTo` mostradas acima inferirão que a chave estrangeira a ser criada deve ser chamada `fooId`. Para usar um nome diferente, como `myFooId`:

```
// Option 1
Foo.hasOne(Bar, {
  foreignKey: 'myFooId'
});
Bar.belongsTo(Foo);

// Option 2
Foo.hasOne(Bar, {
  foreignKey: {
    name: 'myFooId'
  }
});
Bar.belongsTo(Foo);

// Option 3
Foo.hasOne(Bar);
Bar.belongsTo(Foo, {
  foreignKey: 'myFooId'
});

// Option 4
Foo.hasOne(Bar);
Bar.belongsTo(Foo, {
  foreignKey: {
    name: 'myFooId'
  }
});
```

#Relacionamentos um-para-muitos

As associações um-para-muitos conectam uma origem com vários destinos, enquanto todos esses destinos são conectados apenas a essa única origem.

Isso significa que, ao contrário da associação One-To-One, em que tínhamos que escolher onde seria colocada a chave estrangeira, nas associações One-To-Many só existe uma opção. Por exemplo, se um Foo tem muitos Bars (e desta forma cada Bar pertence a um Foo), então a única implementação sensata é ter uma `fooId` coluna na `Bar` tabela. O contrário é impossível, pois um Foo tem muitas Barras.

Neste exemplo, temos os modelos `Team` e `Player`. Queremos dizer ao Sequelize que existe uma relação Um-para-Muitos entre eles, o que significa que um Time tem muitos Jogadores, enquanto cada Jogador pertence a um único Time.

A principal maneira de fazer isso é a seguinte:

```
Team.hasMany(Player);
Player.belongsTo(Team);
```

Novamente, conforme mencionado, a principal maneira de fazer isso é usar um par de associações Sequelize (`hasMany` e `belongsTo`).

Novamente, conforme mencionado, a principal maneira de fazer isso é usar um par de associações Sequelize (`hasMany` e `belongsTo`).

Por exemplo, no PostgreSQL, a configuração acima resultará no seguinte SQL `sync()`:

```
CREATE TABLE IF NOT EXISTS "Teams" (
  /* ... */
);
CREATE TABLE IF NOT EXISTS "Players" (
  /* ... */
  "TeamId" INTEGER REFERENCES "Teams" ("id") ON DELETE SET NULL ON UPDATE CASCADE,
  /* ... */
);
```

#Opções

Link direto para opções

As opções e caso são as mesmas do caso One-To-One. Por exemplo, para alterar o nome da chave estrangeira e garantir que o relacionamento seja obrigatório, podemos fazer:

```
Team.hasMany(Player, {
  foreignKey: 'clubId'
});
Player.belongsTo(Team);
```

Assim como os relacionamentos um-para-um, `ON DELETE` o padrão é `SET NULL` e `ON UPDATE` o padrão é `CASCADE`.

Noções básicas de consultas envolvendo #associações

Com os fundamentos da definição de associações cobertos, podemos examinar as consultas que envolvem associações. As consultas mais comuns sobre este assunto são as consultas *de leitura* (ou seja, SELECTs). Posteriormente, outros tipos de consultas serão mostrados.

Para estudar isso, vamos considerar um exemplo em que temos Navios e Capitães, e uma relação um-para-um entre eles. Permitiremos null em chaves estrangeiras (o padrão), o que significa que um navio pode existir sem um capitão e vice-versa.

```
// This is the setup of our models for the examples below
const Ship = sequelize.define('ship', {
  name: DataTypes.TEXT,
  crewCapacity: DataTypes.INTEGER,
  amountOfSails: DataTypes.INTEGER
}, { timestamps: false });
const Captain = sequelize.define('captain', {
  name: DataTypes.TEXT,
  skillLevel: {
    type: DataTypes.INTEGER,
    validate: { min: 1, max: 10 }
  }
}, { timestamps: false });
Captain.hasOne(Ship);
Ship.belongsTo(Captain);
```

Buscando associações - Carregamento rápido vs

Os conceitos de Eager Loading e Lazy Loading são fundamentais para entender como as associações de busca funcionam no Sequelize. Lazy Loading refere-se à técnica de buscar os dados associados somente quando você realmente deseja; O Eager Loading, por outro lado, refere-se à técnica de buscar tudo de uma vez, desde o início, com uma consulta maior.

#Exemplo de carregamento lento

```
const awesomeCaptain = await Captain.findOne({
  where: {
    name: "Jack Sparrow"
  }
});
// Do stuff with the fetched captain
console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);
// Now we want information about his ship!
const hisShip = await awesomeCaptain.getShip();
// Do stuff with the ship
console.log('Ship Name:', hisShip.name);
console.log('Amount of Sails:', hisShip.amountOfSails);
```

Observe que no exemplo acima, fizemos duas consultas, buscando apenas o ship associado quando quiséssemos utilizá-lo. Isso pode ser especialmente útil se precisarmos ou não do navio, talvez desejemos buscá-lo condicionalmente, apenas em alguns casos; dessa forma, podemos economizar tempo e memória, buscando-o apenas quando necessário.

Observação: o `getShip()` método de instância usado acima é um dos métodos que o Sequelize adiciona automaticamente às `Captain` instâncias. Há outros. Você aprenderá mais sobre eles mais adiante neste guia.

#Exemplo de carregamento ansioso

```
const awesomeCaptain = await Captain.findOne({
  where: {
    name: "Jack Sparrow"
  },
  include: Ship
});
// Now the ship comes with it
console.log('Name:', awesomeCaptain.name);
console.log('Skill Level:', awesomeCaptain.skillLevel);
console.log('Ship Name:', awesomeCaptain.ship.name);
console.log('Amount of Sails:', awesomeCaptain.ship.amountOfSails);
```

Conforme mostrado acima, o Eager Loading é executado no Sequelize usando a `include` opção. Observe que aqui foi realizada apenas uma consulta ao banco de dados (que traz os dados associados junto com a instância).

Criar, atualizar e #excluir

O acima mostrou o básico sobre consultas para buscar dados envolvendo associações. Para criar, atualizar e excluir, você pode:

- Use as consultas de modelo padrão diretamente:

```
// Example: creating an associated model using the standard methods
Bar.create({
  name: 'My Bar',
  fooId: 5
});
// This creates a Bar belonging to the Foo of ID 5 (since fooId is
// a regular column, after all). Nothing very clever going on here.
```

- Ou use os *métodos/mixins especiais* disponíveis para modelos associados, que são explicados mais adiante nesta página.

Observação: o `save()` método de instância não reconhece associações. Em outras palavras, se você alterar um valor de um objeto *filho* que foi carregado antecipadamente junto com um objeto *pai* , chamar `save()` o pai ignorará completamente a alteração que ocorreu no filho.

Por que as associações são definidas em pares

Conforme mencionado anteriormente e mostrado na maioria dos exemplos acima, geralmente as associações no Sequelize são definidas em pares:

- Para criar um relacionamento **um-para-um** , as associações `hasOne` e `belongsTo` são usadas juntas;
- Para criar um relacionamento **um-para-muitos** , as associações `hasMany` e `belongsTo` são usadas juntas;
- Para criar um relacionamento **muitos-para-muitos** , duas `belongsToMany` chamadas são usadas juntas.

Quando uma associação Sequelize é definida entre dois modelos, apenas o modelo *de origem sabe disso* . Assim, por exemplo, ao usar `Foo.hasOne(Bar)` (assim `Foo` é o modelo de origem e `Bar` é o modelo de destino), só `Foo` se sabe da existência dessa associação. É por isso que neste caso, conforme mostrado acima, `Foo` as instâncias ganham os métodos `getBar()` , `setBar()` e `createBar()` , enquanto por outro lado `Bar` as instâncias não recebem nada.

Da mesma forma, para `Foo.hasOne(Bar)` , uma vez que `Foo` conhece o relacionamento, podemos realizar o carregamento antecipado como em `Foo.findOne({ include: Bar })` , mas não podemos fazer `Bar.findOne({ include: Foo })` .

Portanto, para trazer todo o poder para o uso do Sequelize, geralmente configuramos o relacionamento em pares, para que ambos os modelos *o conheçam* .

Demonstração prática:

- Se não definirmos o par de associações, chamando por exemplo apenas `Foo.hasOne(Bar)`:

```
// This works...
await Foo.findOne({ include: Bar });

// But this throws an error:
await Bar.findOne({ include: Foo });
// SequelizeEagerLoadingError: foo is not associated to bar!
```

- Se definirmos o par como recomendado, ou seja, ambos `Foo.hasOne(Bar)` e `Bar.belongsTo(Foo)`:

```
// This works!
await Foo.findOne({ include: Bar });

// This also works!
await Bar.findOne({ include: Foo });
```

Exercícios

- Criar um novo projeto, com as models Marca (id, nome e cidade) e Vinho (id, tipo, preco, teor, marca_id) – definir o relacionamento entre as tabelas.
- Criar os métodos para realizar a inclusão e listagem das marcas.
- Criar os métodos para realizar o CRUD na tabela de vinhos, sendo que a listagem deve retornar dados do vinho e o nome da marca.
- No controller das marcas, criar um método que retorne todas as marcas, com os vinhos de cada marca.
- Criar os seguintes métodos de filtro (para vinhos):
  - Que o tipo contenha a string passada como parâmetro
  - Que sejam da marca passada como parâmetro.
- Criar métodos que retornem a quantidade de vinhos agrupados por:
  - tipo
  - marca
- Testar todos os métodos a partir do Insomnia.

Berçário

Criar uma nova aplicação e as models para definir as tabelas e relacionamentos deste projeto (Banco de Dados I)

Um berçário deseja informatizar suas operações.

Quando um bebê nasce, algumas informações são armazenadas sobre ele, tais como: nome, data do nascimento, peso do nascimento, altura, a mãe deste bebê e o médico que fez seu parto.

Para as mães, o berçário também deseja manter um controle, guardando informações como: nome, endereço, telefone e data de nascimento.

Para os médicos, é importante saber: CRM, nome, telefone celular e especialidade.