Desenvolvimento de Serviços e APIs

Faculdade Senac Pelotas

Escola de Tecnologia da Informação

Prof. Edécio Fernando Iepsen

Getters, Setters e Virtuais

O Sequelize permite definir getters e setters personalizados para os atributos de seus modelos.

O Sequelize também permite que você especifique os chamados *atributos virtuais* , que são atributos no modelo Sequelize que realmente não existem na tabela SQL subjacente, mas são preenchidos automaticamente pelo Sequelize. Eles são muito úteis para criar atributos personalizados que também podem simplificar seu código, por exemplo.

Um getter é uma `get()` função definida para uma coluna na definição do modelo:

```
const User = sequelize.define('user', {
  // Let's say we wanted to see every username in uppercase, even
  // though they are not necessarily uppercase in the database itself
  username: {
    type: DataTypes.STRING,
    get() {
      const rawValue = this.getDataValue('username');
      return rawValue ? rawValue.toUpperCase() : null;
    }
  }
});
```

Este getter, assim como um getter JavaScript padrão, é chamado automaticamente quando o valor do campo é lido:

```
const user = User.build({ username: 'SuperUser123' });
console.log(user.username); // 'SUPERUSER123'
console.log(user.getDataValue('username')); // 'SuperUser123'
```

Observe que, embora tenha `SUPERUSER123` sido registrado acima, o valor realmente armazenado no banco de dados ainda é `SuperUser123`. Costumávamos `this.getDataValue('username')` obter esse valor e convertê-lo em maiúsculas.

Um setter é uma `set()` função definida para uma coluna na definição do modelo. Ele recebe o valor que está sendo definido:

```
const User = sequelize.define('user', {
  username: DataTypes.STRING,
  password: {
    type: DataTypes.STRING,
    set(value) {
      // Storing passwords in plaintext in the database is terrible.
      // Hashing the value with an appropriate cryptographic hash function is better.
      this.setDataValue('password', hash(value));
    }
  }
});
```

```
const user = User.build({ username: 'someone', password: 'NotSo§tr0ngP4$SW0RD!' });
console.log(user.password); // '7cfc84b8ea898bb72462e78b4643cfccd77e9f05678ec2ce78754147ba947acc'
console.log(user.getDataValue('password')); // '7cfc84b8ea898bb72462e78b4643cfccd77e9f05678ec2ce78754147ba947acc'
```

Observe que o Sequelize chamou o setter automaticamente, antes mesmo de enviar os dados para o banco de dados. Os únicos dados que o banco de dados viu foram o valor já com hash.

Se quisermos envolver outro campo de nossa instância de modelo na computação, isso é possível e muito fácil!

```
const User = sequelize.define('user', {
  username: DataTypes.STRING,
  password: {
    type: DataTypes.STRING,
    set(value) {
      // Storing passwords in plaintext in the database is terrible.
      // Hashing the value with an appropriate cryptographic hash function is better.
      // Using the username as a salt is better.
      this.setDataValue('password', hash(this.username + value));
    }
  }
});
```

Nota: Os exemplos acima envolvendo manipulação de senha, embora muito melhores do que simplesmente armazenar a senha em texto simples, estão longe de ser uma segurança perfeita. Manusear senhas corretamente é difícil, tudo aqui é apenas um exemplo para mostrar a funcionalidade do Sequelize. Sugerimos envolver um especialista em segurança cibernética e/ou ler documentos OWASP e/ou visitar o InfoSec StackExchange .

#Campos virtuais

Campos virtuais são campos que o Sequelize preenche sob o capô, mas na realidade eles nem existem no banco de dados.

Por exemplo, digamos que temos os atributos `firstName` e para um usuário. `lastName`

Seria bom ter uma maneira simples de obter o *nome completo* diretamente! Podemos combinar a ideia de `getters` com o tipo de dados especial que o Sequelize fornece para esse tipo de situação: `DataTypes.VIRTUAL`:

```
const { DataTypes } = require("sequelize");

const User = sequelize.define('user', {
  firstName: DataTypes.TEXT,
  lastName: DataTypes.TEXT,
  fullName: {
    type: DataTypes.VIRTUAL,
    get() {
      return `${this.firstName} ${this.lastName}`;
    },
    set(value) {
      throw new Error('Do not try to set the `fullName` value!');
    }
  }
});
```

O `VIRTUAL` campo não faz com que uma coluna na tabela exista. Ou seja, o modelo acima não terá `fullName` coluna. No entanto, parecerá tê-lo!

```
const user = await User.create({ firstName: 'John', lastName: 'Doe' });
console.log(user.fullName); // 'John Doe'
```

Validações e Restrições

Neste tutorial você aprenderá como configurar validações e restrições para seus modelos no Sequelize.

Para este tutorial, a seguinte configuração será assumida:

```
const { Sequelize, Op, Model, DataTypes } = require("sequelize");
const sequelize = new Sequelize("sqlite::memory:");

const User = sequelize.define("user", {
  username: {
    type: DataTypes.TEXT,
    allowNull: false,
    unique: true
  },
  hashedPassword: {
    type: DataTypes.STRING(64),
    validate: {
      is: /^[0-9a-f]{64}$/i
    }
  }
});
```

Diferença entre validações e #restrições

As validações são verificações realizadas no nível Sequelize, em JavaScript puro. Eles podem ser arbitrariamente complexos se você fornecer uma função de validador personalizada ou podem ser um dos validadores integrados oferecidos pelo Sequelize. Se uma validação falhar, nenhuma consulta SQL será enviada ao banco de dados.

Por outro lado, as restrições são regras definidas no nível SQL. O exemplo mais básico de restrição é uma restrição única. Se uma verificação de restrição falhar, um erro será lançado pelo banco de dados e o Sequelize encaminhará esse erro para o JavaScript (neste exemplo, lançando um `SequelizeUniqueConstraintError`). Observe que neste caso foi realizada a consulta SQL, diferentemente do caso das validações.

#Restrição única

Link direto para restrição única

Nosso exemplo de código define uma restrição única no `username` campo:

```
/* ... */ {
  username: {
    type: DataTypes.TEXT,
    allowNull: false,
    unique: true
  },
```

Quando este modelo for sincronizado (chamando `sequelize.sync` por exemplo), o `username` campo será criado na tabela como `` `username` TEXT UNIQUE ``, e uma tentativa de inserir um nome de usuário que já existe lá lançará um `SequelizeUniqueConstraintError`.

Permitir/desautorizar

Por padrão, `null` é um valor permitido para cada coluna de um modelo. Isso pode ser desabilitado definindo a `allowNull: false` opção para uma coluna, como foi feito no `username` campo do nosso exemplo de código:

```
/* ... */ {
  username: {
    type: DataTypes.TEXT,
    allowNull: false,
    unique: true
  },
} /* ... */
```

Sem `allowNull: false`, a chamada `User.create({})` funcionaria.

#Validações por atributo

Você pode definir seus validadores personalizados ou usar vários validadores integrados, implementados por validator.js (10.11.0) , conforme mostrado abaixo.

```
sequelize.define('foo', {
  bar: {
    type: DataTypes.STRING,
    validate: {
      is: /^[a-z]+$/i,          // matches this RegExp
      is: ["^[a-z]+$",'i'],     // same as above, but constructing the RegExp from a string
      not: /^[a-z]+$/i,         // does not match this RegExp
      not: ["^[a-z]+$",'i'],    // same as above, but constructing the RegExp from a string
      isEmail: true,            // checks for email format (foo@bar.com)
      isUrl: true,              // checks for url format (https://foo.com)
      isIP: true,               // checks for IPv4 (129.89.23.1) or IPv6 format
      isIPv4: true,             // checks for IPv4 (129.89.23.1)
      isIPv6: true,             // checks for IPv6 format
      isAlpha: true,            // will only allow letters
      isAlphanumeric: true,     // will only allow alphanumeric characters, so "_abc" will fail
      isNumeric: true,          // will only allow numbers
      isInt: true,              // checks for valid integers
      isFloat: true,            // checks for valid floating point numbers
      isDecimal: true,          // checks for any numbers
      isLowercase: true,        // checks for lowercase
      isUppercase: true,        // checks for uppercase
```

Exercícios

- Criar um novo projeto, com a model Carro, contendo os atributos id, modelo, marca, ano, preco e placa. Os campos não podem conter valores nulos.
  - A marca do veículo deve ser armazenada com letras maiúsculas (set).
  - Validar o campo ano - que deve ser inteiro, com 4 dígitos e o valor não pode ser superior ao ano atual.
  - O campo placa deve ter conteúdo único e possuir tamanho de 7 caracteres.
  - Criar também o campo virtual status, que deve retornar 'Novo' (ano = atual), 'Semi-novo' (ano = atual-1 ou ano=atual-2) ou 'Usado'.
- Criar os métodos para realizar o CRUD básico dos dados.
- Criar um método que retorne os veículos em ordem de ano, decrescente.
- Criar um método que contenha um filtro para o campo virtual status.
- Criar um método que retorne marca e quantidade de veículos por marca (agrupamento).
- Testar todos os métodos a partir do Insomnia.