Version: v6 - stable

Validations & Constraints

In this tutorial you will learn how to setup validations and constraints for your models in Sequelize.

For this tutorial, the following setup will be assumed:

```
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
    }
  }
});

(async () => {
  await sequelize.sync({ force: true });
  // Code here
})();
```

Difference between Validations and Constraints

Validations are checks performed in the Sequelize level, in pure JavaScript. They can be arbitrarily complex if you provide a custom validator function, or can be one of the built-in validators offered by Sequelize. If a validation fails, no SQL query will be sent to the database at all.

On the other hand, constraints are rules defined at SQL level. The most basic example of constraint is an Unique Constraint. If a constraint check fails, an error will be thrown by the database and Sequelize will forward this error to JavaScript (in this example, throwing a `SequelizeUniqueConstraintError`). Note that in this case, the SQL query was performed, unlike the case for validations.

Unique Constraint

Our code example above defines a unique constraint on the `username` field:

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

When this model is synchronized (by calling `sequelize.sync` for example), the `username` field will be created in the table as `` `username` TEXT UNIQUE ``, and an attempt to insert an username that already exists there will throw a `SequelizeUniqueConstraintError`.

Allowing/disallowing null values

By default, `null` is an allowed value for every column of a model. This can be disabled setting the `allowNull: false` option for a column, as it was done in the `username` field from our code example:

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

Without `allowNull: false`, the call `User.create({})` would work.

Note about `allowNull` implementation

The `allowNull` check is the only check in Sequelize that is a mix of a *validation* and a *constraint* in the senses described at the beginning of this tutorial. This is because:

- If an attempt is made to set `null` to a field that does not allow null, a `ValidationError` will be thrown *without any SQL query being performed.*
- In addition, after `sequelize.sync`, the column that has `allowNull: false` will be defined with a `NOT NULL` SQL constraint. This way, direct SQL queries that attempt to set the value to `null` will also fail.

Validators

Model validators allow you to specify format/content/inheritance validations for each attribute of the model. Validations are automatically run on `create`, `update` and `save`. You can also call `validate()` to manually validate an instance.

Per-attribute validations

You can define your custom validators or use several built-in validators, implemented by validator.js (10.11.0), as shown below.

```
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
      notNull: true,            // won't allow null
      isNull: true,             // only allows null
      notEmpty: true,           // don't allow empty strings
      equals: 'specific value', // only allow a specific value
      contains: 'foo',          // force specific substrings
      notIn: [['foo', 'bar']],  // check the value is not one of these
      isIn: [['foo', 'bar']],   // check the value is one of these
      notContains: 'bar',       // don't allow specific substrings
      len: [2,10],              // only allow values with length between 2 and 10
      isUUID: 4,                // only allow uuids
      isDate: true,             // only allow date strings
      isAfter: "2011-11-05",    // only allow date strings after a specific date
      isBefore: "2011-11-05",   // only allow date strings before a specific date
      max: 23,                  // only allow values <= 23
      min: 23,                  // only allow values >= 23
      isCreditCard: true,       // check for valid credit card numbers

      // Examples of custom validators:
      isEven(value) {
        if (parseInt(value) % 2 !== 0) {
          throw new Error('Only even values are allowed!');
        }
      }
      isGreaterThanOtherField(value) {
        if (parseInt(value) <= parseInt(this.otherField)) {
          throw new Error('Bar must be greater than otherField.');
        }
      }
    }
  }
});
```

Note that where multiple arguments need to be passed to the built-in validation functions, the arguments to be passed must be in an array. But if a single array argument is to be passed, for instance an array of acceptable strings for `isIn`, this will be interpreted as multiple string arguments instead of one array argument. To work around this pass a single-length array of arguments, such as `[['foo', 'bar']]` as shown above.

To use a custom error message instead of that provided by validator.js, use an object instead of the plain value or array of arguments, for example a validator which needs no argument can be given a custom message with

```
isInt: {
  msg: "Must be an integer number of pennies"
}
```

or if arguments need to also be passed add an `args` property:

```
isIn: {
  args: [['en', 'zh']],
  msg: "Must be English or Chinese"
}
```

When using custom validator functions the error message will be whatever message the thrown `Error` object holds.

See the validator.js project for more details on the built in validation methods.

Hint: You can also define a custom function for the logging part. Just pass a function. The first parameter will be the string that is logged.

`allowNull` interaction with other validators

If a particular field of a model is set to not allow null (with `allowNull: false`) and that value has been set to `null`, all validators will be skipped and a `ValidationError` will be thrown.

On the other hand, if it is set to allow null (with `allowNull: true`) and that value has been set to `null`, only the built-in validators will be skipped, while the custom validators will still run.

This means you can, for instance, have a string field which validates its length to be between 5 and 10 characters, but which also allows `null` (since the length validator will be skipped automatically when the value is `null`):

```
class User extends Model {}
User.init({
  username: {
    type: DataTypes.STRING,
    allowNull: true,
    validate: {
      len: [5, 10]
    }
  }
}, { sequelize });
```

You also can conditionally allow `null` values, with a custom validator, since it won't be skipped:

```
class User extends Model {}
User.init({
  age: Sequelize.INTEGER,
  name: {
    type: DataTypes.STRING,
    allowNull: true,
    validate: {
      customValidator(value) {
        if (value === null && this.age !== 10) {
          throw new Error("name can't be null unless age is 10");
        }
      }
    }
  }
}, { sequelize });
```

You can customize `allowNull` error message by setting the `notNull` validator:

```
class User extends Model {}
User.init({
  name: {
    type: DataTypes.STRING,
    allowNull: false,
    validate: {
      notNull: {
        msg: 'Please enter your name'
      }
    }
  }
}, { sequelize });
```

Model-wide validations

Validations can also be defined to check the model after the field-specific validators. Using this you could, for example, ensure either neither of `latitude` and `longitude` are set or both, and fail if one but not the other is set.

Model validator methods are called with the model object's context and are deemed to fail if they throw an error, otherwise pass. This is just the same as with custom field-specific validators.

Any error messages collected are put in the validation result object alongside the field validation errors, with keys named after the failed validation method's key in the `validate` option object. Even though there can only be one error message for each model validation method at any one time, it is presented as a single string error in an array, to maximize consistency with the field errors.

An example:

```
class Place extends Model {}
Place.init({
  name: Sequelize.STRING,
  address: Sequelize.STRING,
  latitude: {
    type: DataTypes.INTEGER,
    validate: {
      min: -90,
      max: 90
    }
  },
  longitude: {
    type: DataTypes.INTEGER,
    validate: {
      min: -180,
      max: 180
    }
  },
}, {
  sequelize,
  validate: {
    bothCoordsOrNone() {
      if ((this.latitude === null) !== (this.longitude === null)) {
        throw new Error('Either both latitude and longitude, or neither!');
      }
    }
  }
})
```

In this simple case an object fails validation if either latitude or longitude is given, but not both. If we try to build one with an out-of-range latitude and no longitude, `somePlace.validate()` might return:

```
{
  'latitude': ['Invalid number: latitude'],
  'bothCoordsOrNone': ['Either both latitude and longitude, or neither!']
}
```

Such validation could have also been done with a custom validator defined on a single attribute (such as the `latitude` attribute, by checking `(value === null) !== (this.longitude === null)`), but the model-wide validation approach is cleaner.

Edit this page

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

Previous
« Getters, Setters & Virtuals

Next
Raw Queries »