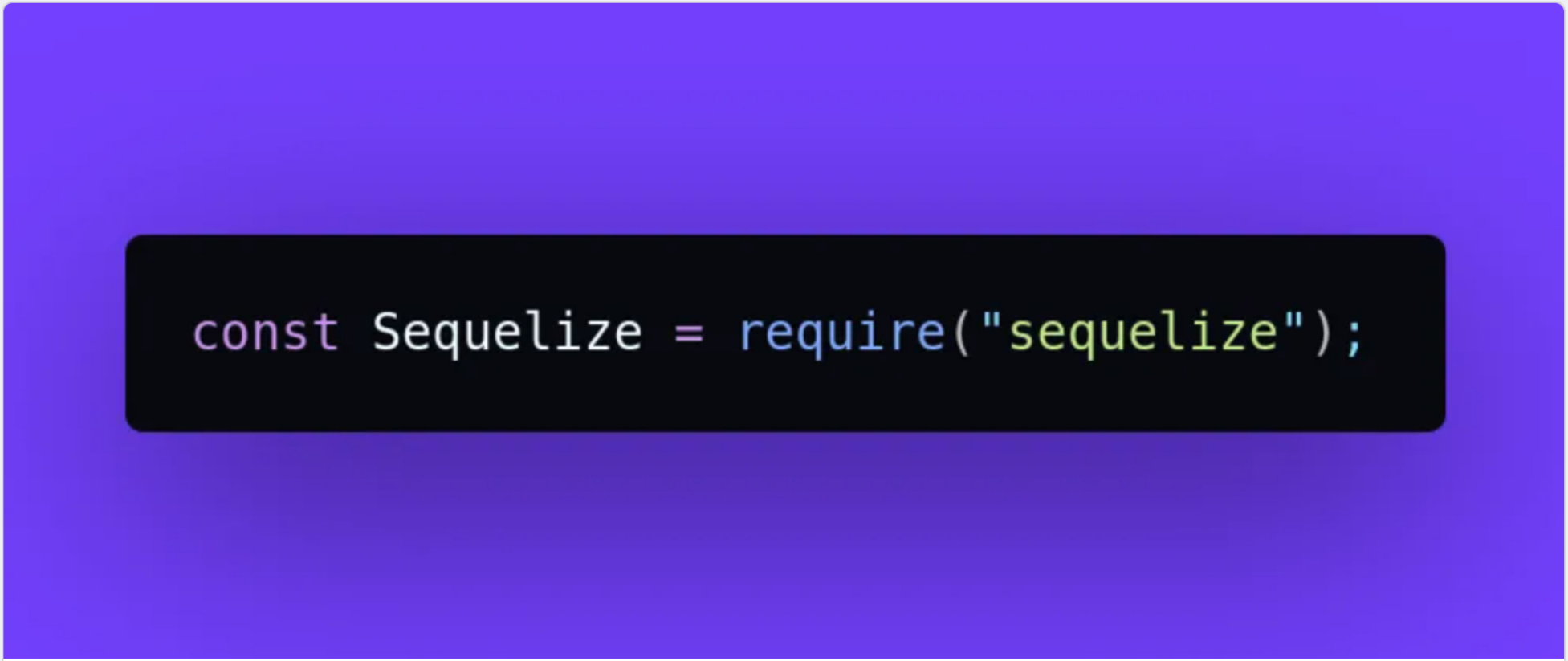


The Comprehensive Sequelize Cheatsheet

#database #node #webdev #javascript

Aniket May 8 · *Updated on Jul 24* · 12 min read

Sequelize is the most famous Node ORM and is quite feature-rich, but while using it I

Sequelize is the most famous Node ORM and is quite feature rich, but while using it I spend much of my time juggling around between the documentation and different google searches.
This Cheatsheet is the one that I always wanted but was never able to find.
See any error or anything missing? Comment below, or better, send a pull request to the repo linked in the end.

Table of Contents

    3. findOrCreate
    4. findAll
    5. findAndCountAll
    6. count
    7. max
    8. min
    9. sum
2. Filtering
    1. where
        1. Operators
    2. order
    3. Pagination and Limiting
7. Things I did not include in this Cheatsheet (with links to official docs)
    1. Hooks
    2. Transactions
    3. Scopes
    4. Raw Queries

5. Eager Loading

Installing Dependencies

Installing Sequelize

```
npm install --save sequelize
```

Installing Database Driver

You also need to install the driver for the database you're using.

```
# One of the following:
npm install --save pg pg-hstore # Postgres If node version < 14 use pg@7.12.1 instea
npm install --save mysql2
npm install --save mariadb
npm install --save sqlite3
npm install --save tedious # Microsoft SQL Server
```

Setting up a Connection

A Sequelize instance must be created to connect to the database. By default, this connection is kept open and used for all the queries but can be closed explicitly.

Instance Creation

```
const Sequelize = require('sequelize');

// Option 1: Passing parameters separately
const sequelize = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: /* one of 'mysql' | 'mariadb' | 'postgres' | 'mssql' */
});

// Option 2: Passing a connection URI
const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname');

// For SQLite, use this instead
const sequelize = new Sequelize({
  dialect: 'sqlite',
  storage: 'path/to/database.sqlite'
});
```

For more detailed information about connecting to different dialects, check out the official docs

Testing Connection

`.authenticate()` can be used with the created instance to check whether the connection is working.

```
sequelize
  .authenticate()
  .then(() => {
    console.log("Connection has been established successfully.");
  })
  .catch((err) => {
    console.error("Unable to connect to the database:", err);
  });
```

Closing Connection

```
sequelize.close();
```

Defining Models

Basic Definition

To define mappings between Model and Table, we can use the `.define()` method
To set up a basic model with only attributes and their datatypes

```
const ModelName = sequelize.define("tablename", {
  // s will be appended automatically to the tablename
  firstColumn: Sequelize.INTEGER,
  secondColumn: Sequelize.STRING,
});
```

For getting a list of all the data types supported by Sequelize, check out the official docs

Extending Column Definition

Basic Extentions

Apart from datatypes, many other options can also be set on each column

```
const ModelName = sequelize.define("tablename", {
  firstColumn: {
    // REQUIRED
    type: Sequelize.INTEGER,
    // OPTIONAL
    allowNull: false, // true by default
    defaultValue: 1,
    primaryKey: true, // false by default
    autoIncrement: true, // false by default
    unique: true,
    field: "first_column", // To change the field name in actual table
  },
});
```

Composite Unique Key

To create a composite unique key, give the same name to the constraint in all the columns you want to include in the composite unique key

```
const ModelName = sequelize.define("tablename", {
  firstColumn: {
    type: Sequelize.INTEGER,
    unique: "compositeIndex",
  },
```

```
  secondColumn: {
    type: Sequelize.INTEGER,
    unique: "compositeIndex",
  },
});
```

They can also be created using indexes

```
const ModelName = sequelize.define(
  "tablename",
  {
    firstColumn: Sequelize.INTEGER,
    secondColumn: Sequelize.INTEGER,
  },
  {
    indexes: [
      {
        unique: true,
        fields: ["firstColumn", "secondColumn"],
      },
    ],
  }
);
```

Getters and Setters

Getters and Setters

Getters can be used to get the value of the column after some processing.
Setters can be used to process the value before saving it into the table.

```
const Employee = sequelize.define("employee", {
  name: {
    type: Sequelize.STRING,
    allowNull: false,
    get() {
      const title = this.getDataValue("title");
      // 'this' allows you to access attributes of the instance
      return this.getDataValue("name") + " (" + title + ")";
    },
  },
  title: {
    type: Sequelize.STRING,
    allowNull: false,
    set(val) {
      this.setDataValue("title", val.toUpperCase());
    },
  },
});

Employee.create({ name: "John Doe", title: "senior engineer" }).then(
  (employee) => {
    console.log(employee.get("name")); // John Doe (SENIOR ENGINEER)
```

```
    console.log(employee.get("title")); // SENIOR ENGINEER
  }
);
```

For more in-depth information about Getters and Setters, check out the official docs

Validations

Validations are automatically run on `create`, `update` and `save`

Per Attribute Validations

```
const ModelName = sequelize.define("tablename", {
  firstColumn: {
    type: Sequelize.STRING,
    validate: {
      is: ["^[a-z]+$", "i"], // will only allow letters
      is: /^[a-z]+$/i, // same as the previous example using real RegExp
      not: ["[a-z]", "i"], // will not allow letters
      isEmail: true, // checks for email format (foo@bar.com)
      isUrl: true, // checks for url format (http://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" wi
        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!");
        }
      },
    },
  },
});
```

Model Wide Validations

```
const ModelName = sequelize.define(
  "tablename",
  {
    firstColumn: Sequelize.INTEGER,
    secondColumn: Sequelize.INTEGER,
  },
  {
    validate: {
      // Define your Model Wide Validations here
      checkSum() {
        if (this.firstColumn + this.secondColumn < 10) {
          throw new Error("Require sum of columns >=10");
        }
      },
    },
```

```
);
```

Timestamps

```
const ModelName = sequelize.define(
  "tablename",
  {
    firstColumn: Sequelize.INTEGER,
  },
  {
    timestamps: true, // Enable timestamps
    createdAt: false, // Don't create createdAt
    updatedAt: false, // Don't create updatedAt
    updatedAt: "updateTimestamp", // updatedAt should be called updateTimestamp
  }
);
```

Database Synchronization

Sequelize can automatically create the tables, relations and constraints as defined in the models

```
```

```
ModelName.sync(); // Create the table if not already present

// Force the creation
ModelName.sync({ force: true }); // this will drop the table first and re-create it

ModelName.drop(); // drop the tables
```

You can manage all models at once using sequelize instead

```
sequelize.sync(); // Sync all models that aren't already in the database

sequelize.sync({ force: true }); // Force sync all models

sequelize.sync({ force: true, match: /_test$/ }); // Run .sync() only if database na

sequelize.drop(); // Drop all tables
```

Expansion of Models

Sequelize Models are ES6 classes. We can easily add custom instance or class level methods.

```
const ModelName = sequelize.define("tablename", {
  firstColumn: Sequelize.STRING,
  secondColumn: Sequelize.STRING,
});
// Adding a class level method
ModelName.classLevelMethod = function () {
  return "This is a Class level method";
};

// Adding a instance level method
ModelName.prototype.instanceLevelMethod = function () {
  return [this.firstColumn, this.secondColumn].join(" ");
};
```

Indexes

```
const User = sequelize.define(
  "User",
  {
    /* attributes */
  },
  {
    indexes: [
      // Create a unique index on email
      {
```

```
    unique: true,
    fields: ["email"],
  },

  // Creates a gin index on data with the jsonb_path_ops operator
  {
    fields: ["data"],
    using: "gin",
    operator: "jsonb_path_ops",
  },

  // By default index name will be [table]_[fields]
  // Creates a multi column partial index
  {
    name: "public_by_author",
    fields: ["author", "status"],
    where: {
      status: "public",
    },
  },

  // A BTREE index with an ordered field
  {
    name: "title_index",
    using: "BTREE",
    fields: [
      "author",
```

```
            {
              attribute: "title",
              collate: "en_US",
              order: "DESC",
              length: 5,
            },
          ],
        },
      ],
    }
);
```

Associations

Defining Associations

There are four types of definitions. They are **used in pairs**.
For the example lets define two Models

```
const Foo = sequelize.define("foo" /* ... */);
const Bar = sequelize.define("bar" /* ... */);
```

The model whose function we will be calling is called the source model, and the model

which is passed as a parameter is called the target model.

hasOne

```
Foo.hasOne(Bar, {
  /* options */
});
```

This states that a One-to-One relationship exists between Foo and Bar with foreign key defined in Bar

belongsTo

```
Foo.belongsTo(Bar, {
  /* options */
});
```

This states that a One-to-One or One-to-Many relationship exists between Foo and Bar with foreign key defined in Foo

hasMany

```
Foo.hasMany(Bar, {
  /* options */
});
```

This states that a One-to-Many relationship exists between Foo and Bar with foreign key defined in Bar

belongsToMany

```
Foo.belongsToMany(Bar, {
  // REQUIRED
  through: "C", // Model can also be passed here
  /* options */
});
```

This states that a Many-to-Many relationship exists between Foo and Bar through a junction table C

Relations

One to One

To setup a One-to-One relationship, we have to simply write

```
Foo.hasOne(Bar);
Bar.belongsTo(Foo);
```

In the above case, no option was passed. This will auto create a foreign key column in Bar referencing to the primary key of Foo. If the column name of PK of Foo is email, the column formed in Bar will be fooEmail.

Options

The following options can be passed to customize the relation.

```
Foo.hasOne(Bar, {
  foreignKey: "customNameForFKColumn", // Name for new column added to Bar
  sourceKey: "email", // Column in Foo that FK will reference to
  // The possible choices are RESTRICT, CASCADE, NO ACTION, SET DEFAULT and SET NULL
  onDelete: "RESTRICT", // Default is SET NULL
  onUpdate: "RESTRICT", // Default is CASCADE
});
Bar.belongsTo(Foo, {
  foreignKey: "customNameForFKColumn", // Name for new column added to Bar
```

```
});
```

One to Many

To setup a One-to-One relationship, we have to simply write

```
Foo.hasMany(Bar);
Bar.belongsTo(Foo);
```

In the above case, no option was passed. This will auto create a foreign key column in Bar referencing to the primary key of Foo. If the column name of PK of Foo is email, the column formed in Bar will be fooEmail.

Options

The following options can be passed to customize the relation.

```
Foo.hasMany(Bar, {
  foreignKey: "customNameForFKColumn", // Name for new column added to Bar
  sourceKey: "email", // Column in Foo that FK will reference to
  // The possible choices are RESTRICT, CASCADE, NO ACTION, SET DEFAULT and SET NULL
  onDelete: "RESTRICT", // Default is SET NULL
  onUpdate: "RESTRICT", // Default is CASCADE
```

```
  onUpdate: "RESTRICT", // Default is CASCADE
});
Bar.belongsTo(Foo, {
  foreignKey: "customNameForFKColumn", // Name for new column added to Bar
});
```

Many to Many

To setup a Many-to-Many relationship, we have to simply write

```
// This will create a new table rel referencing the PK(by default) of both the table
Foo.belongsToMany(Bar, { through: "rel" });
Bar.belongsToMany(Foo, { through: "rel" });
```

Options

The following options can be passed to customize the relation.

```
Foo.belongsToMany(Bar, {
  as: "Bar",
  through: "rel",
  foreignKey: "customNameForFoo", // Custom name for column in rel referencing to Fo
  sourceKey: "name", // Column in Foo which rel will reference to
```

```
});
Bar.belongsToMany(Foo, {
  as: "Foo",
  through: "rel",
  foreignKey: "customNameForBar", // Custom name for column in rel referencing to Ba
  sourceKey: "name", // Column in Foo which rel will reference to
});
```

Instances

Creating Instances

There are two ways to create instances

build

We can use build method to create non-persistent(not saved to table) instances. They will automatically get the default values as stated while defining the Model.
To save to the table we need to save these instances explicitly.

```
const instance = ModelName.build({
  firstColumn: "Lorem Ipsum",
```

```
  secondColumn: "Dotor",
});
// To save this instance to the db
instance.save().then((savedInstance) => {});
```

create

We can create a method to create persistent(saved to table) instances

```
const instance = ModelName.create({
  firstColumn: "Lorem Ipsum",
  secondColumn: "Dotor",
});
```

Mutating Instances

Update

There are two ways to update any instance

```
// Way 1
```

```
instance.secondColumn = "Updated Dotor";
instance.save().then(() => {});
// To update only some of the modified fields
instance.save({ fields: ["secondColumn"] }).then(() => {});

// Way 2
instance
  .update({
    secondColumn: "Updated Dotor",
  })
  .then(() => {});
// To update only some of the modified fields
instance
  .update(
    {
      secondColumn: "Updated Dotor",
    },
    { fields: ["secondColumn"] }
  )
  .then(() => {});
```

Delete

To delete/destroy any instance

```
instance.destroy().then(() => {});
```

Using Models

Methods

findByPk

Returns the row with the given value of Primary Key.

```
ModelName.findByPK(PKvalue).then((foundResult) => {});
```

findOne

Returns the first row with the given conditions.

```
ModelName.findOne({
  // Optional options
  // Filtering results using where
  where: { firstColumn: "value" },
```

```
  // Returning only specified columns
  attributes: ["firstColumn", "secondColumn"],
}).then((foundResult) => {});
```

findOrCreate

Returns the row found with given conditions. If no such row exists, creates one and returns that instead

```
ModelName.findOrCreate({
  // Conditions that must be met
  where: { firstColumn: "lorem ipsum" },
  // Value of other columns to be set if no such row found
  defaults: { secondColumn: "dotor" },
}).then(([result, created]) => {}); //Created is a bool which tells created or not
```

findAll

Returns all the rows satisfying the conditions

```
ModelName.findAll({
  // Optional Options
```

```
  where: {
    firstColumn: "lorem ipsum",
  },
  offset: 10,
  limit: 2,
}).then((results) => {});
```

findAndCountAll

Returns all the rows satisfying the conditions along with their count

```
ModelName.findAndCountAll({
  where: {
    firstColumn: "lorem ipsum",
  },
}).then((results) => {
  console.log(results.count);
  console.log(results.rows);
});
```

count

Returns number of rows satisfying the conditions

```
ModelName.count({
  where: {
    firstColumn: "lorem ipsum",
  },
}).then((c) => {});
```

max

Returns the value of the column with max value with given conditions

```
ModelName.max("age", {
  where: {
    firstColumn: "lorem ipsum",
  },
}).then((maxAge) => {});
```

min

Returns the value of the column with min value with given conditions

```
ModelName.min("age", {
```

```
  where: {
    firstColumn: "lorem ipsum",
  },
}).then((minAge) => {});
```

sum

Returns the sum of all the values of the columns with given conditions

```
ModelName.sum({
  where: {
    firstColumn: "lorem ipsum",
  },
}).then((sumAge) => {});
```

Filtering

where

where can be used to filter the results we work on

We can directly pass the values

```
ModelName.findAll({
  where: {
    firstColumn: "lorem ipsum",
  },
});
```

We can use AND and OR

```
const Op = Sequelize.Op;
ModelName.findAll({
  where: {
    [Op.and]: [{ secondColumn: 5 }, { thirdColumn: 6 }],
    [Op.or]: [{ secondColumn: 5 }, { secondColumn: 6 }],
  },
});
```

We can use various other operators

```
const Op = Sequelize.Op;
ModelName.findAll({
  where: {
    firstColumn: {
      [Op.ne]: "lorem ipsum dotor", // Not equal to
```

```
    },
  },
});
```

We can mix and match too

```
const Op = Sequelize.Op;
ModelName.findAll({
  where: {
    [Op.or]: {
      [Op.lt]: 1000,
      [Op.eq]: null,
    },
  },
});
```

Operators

Here is the full list of Operators

```
const Op = Sequelize.Op

[Op.and]: [{a: 5}, {b: 6}] // (a = 5) AND (b = 6)
[Op.or]: [{a: 5}, {a: 6}]  // (a = 5 OR a = 6)
```

```
[Op.gt]: 6,                  // > 6
[Op.gte]: 6,                 // >= 6
[Op.lt]: 10,                 // < 10
[Op.lte]: 10,                // <= 10
[Op.ne]: 20,                 // != 20
[Op.eq]: 3,                  // = 3
[Op.is]: null                // IS NULL
[Op.not]: true,              // IS NOT TRUE
[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.startsWith]: 'hat'       // LIKE 'hat%'
[Op.endsWith]: 'hat'         // LIKE '%hat'
[Op.substring]: 'hat'        // LIKE '%hat%'
[Op.regexp]: '^[h|a|t]'      // REGEXP/~ '^[h|a|t]' (MySQL/PG only)
[Op.notRegexp]: '^[h|a|t]' // NOT REGEXP/!~ '^[h|a|t]' (MySQL/PG only)
[Op.iRegexp]: '^[h|a|t]'     // ~* '^[h|a|t]' (PG only)
[Op.notIRegexp]: '^[h|a|t]' // !~* '^[h|a|t]' (PG only)
[Op.like]: { [Op.any]: ['cat', 'hat']}
                             // LIKE ANY ARRAY['cat', 'hat'] - also works for iLike ar
[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)

[Op.col]: 'user.organization_id' // = "user"."organization_id", with dialect specifi
[Op.gt]: { [Op.all]: literal('SELECT 1') }
                           // > ALL (SELECT 1)
[Op.contains]: 2            // @> '2'::integer (PG range contains element operator)
[Op.contains]: [1, 2]       // @> [1, 2) (PG range contains range operator)
[Op.contained]: [1, 2]      // <@ [1, 2) (PG range is contained by operator)
[Op.overlap]: [1, 2]        // && [1, 2) (PG range overlap (have points in common) op
[Op.adjacent]: [1, 2]       // -|- [1, 2) (PG range is adjacent to operator)
[Op.strictLeft]: [1, 2]     // << [1, 2) (PG range strictly left of operator)
[Op.strictRight]: [1, 2]    // >> [1, 2) (PG range strictly right of operator)
[Op.noExtendRight]: [1, 2] // &< [1, 2) (PG range does not extend to the right of op
[Op.noExtendLeft]: [1, 2]  // &> [1, 2) (PG range does not extend to the left of ope
```

order

```
ModelName.findAll({
  order: [
    ["firstColumn", "DESC"],
    ["secondColumn", "ASC"],
  ],
});
```

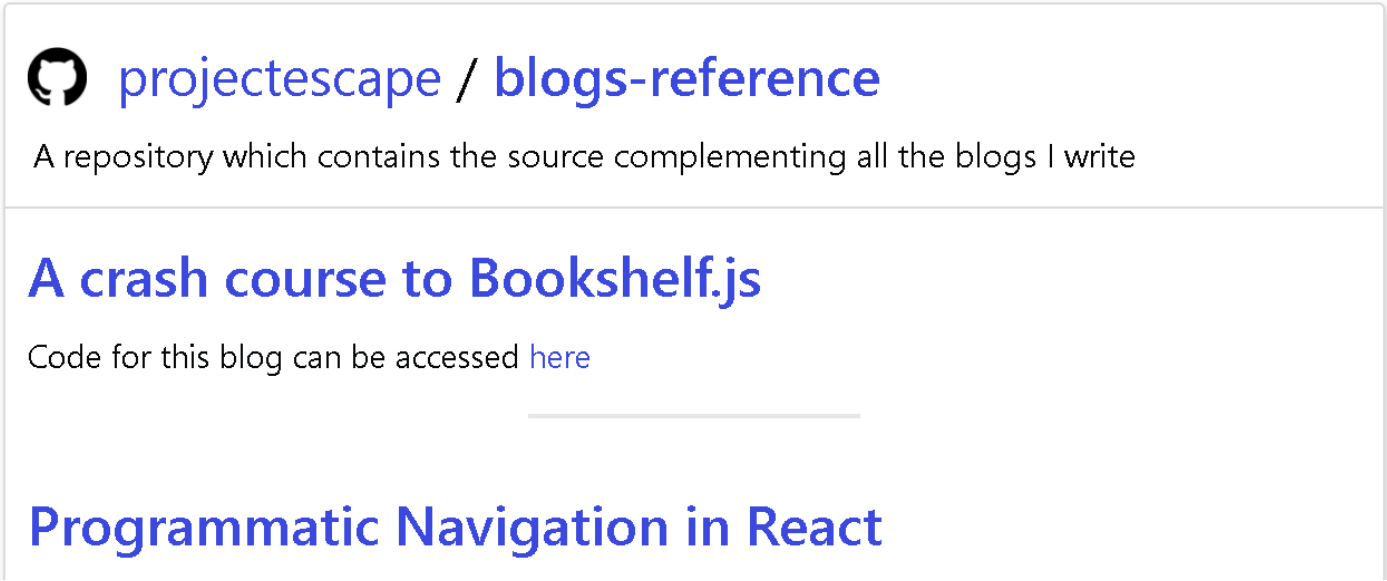
For much more detailed information on ordering, check out the official docs

Pagination and Limiting

```
ModelName.findAll({
  offset: 5, // Skip the first five results
  limit: 5, // Return only five results
});
```

Checkout my blogs repo

Send pull requests to add/modify to this post.



Code for this blog can be accessed here

The Comprehensive Sequelize Cheatsheet

Markdown for this blog can be accessed here

View on GitHub

Discussion

Subscribe

Add to the discussion

Chinedum Onyema

Jul 30 •••

Great summary. Thank you.

2

Reply

Sami Hamaizi

Jun 7 •••

that was so helpfull, thank you

2

Reply

Sm0ke

May 8 •••

Great post. Thank you!

2

Reply

Aniket Author

May 8 •••

This took way more time than I expected, but life will be much easier moving forward 😌

1

Reply

PhamMinhHaiAu-12035071

May 11 •••

great post. vote Sequelize and typeorm

1

Reply

Aniket Author

May 16

Haven't used typeorm yet!

1

Reply

Code of Conduct • Report abuse

Read next

How To Make Landing page Using Html, CSS, and JavaScript

backlinkn - Nov 5

How I developed and deployed my optimized website within a day

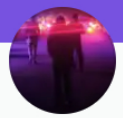
gmlunesa - Oct 30

How to become a Kickass Web Developer in 2021 [Frontend & Backend Tips]

Sunil Joshi - Oct 26

Fetch vs. Axios - comparison

Duomly - Nov 2



Aniket

A Forking Idiot!

Follow

WORK STATUS

I'm looking for work!

JOINED

Sep 25, 2019

More from Aniket

Major improvements made to monetize-npm-cli 🔥

#gftwhackathon #node #webdev #javascript

Web Monetization for NPM packages!!

#gftwhackathon #node #webdev #showdev

Unable to use global variables with es6 imports in node

#help #javascript #node #webdev