# Model Basics

In this tutorial you will learn what models are in Sequelize and how to use them.

## Concept

Models are the essence of Sequelize. A model is an abstraction that represents a table in your database. In Sequelize, it is a class that extends Model.

The model tells Sequelize several things about the entity it represents, such as the name of the table in the database and which columns it has (and their data types).

A model in Sequelize has a name. This name does not have to be the same name of the table it represents in the database. Usually, models have singular names (such as `User`) while tables have pluralized names (such as `Users`), although this is fully configurable.

## Model Definition

Models can be defined in two equivalent ways in Sequelize:

- Calling `sequelize.define(modelName, attributes, options)`
- Extending Model and calling `init(attributes, options)`

After a model is defined, it is available within `sequelize.models` by its model name.

To learn with an example, we will consider that we want to create a model to represent users, which have a `firstName` and a `lastName`. We want our model to be called `User`, and the table it represents is called `Users` in the database.

Both ways to define this model are shown below. After being defined, we can access our model with `sequelize.models.User`.

### Using `sequelize.define`:

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

const User = sequelize.define(
  'User',
  {
    // Model attributes are defined here
    firstName: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    lastName: {
      type: DataTypes.STRING,
      // allowNull defaults to true
    },
  },
  {
    // Other model options go here
  },
);

// `sequelize.define` also returns the model
console.log(User === sequelize.models.User); // true
```

## Extending Model

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

class User extends Model {}

User.init(
  {
    // Model attributes are defined here
    firstName: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    lastName: {
      type: DataTypes.STRING,
      // allowNull defaults to true
    },
  },
  {
```

```
    // Other model options go here
    sequelize, // We need to pass the connection instance
    modelName: 'User', // We need to choose the model name
  },
);

// the defined model is the class itself
console.log(User === sequelize.models.User); // true
```

Internally, `sequelize.define` calls `Model.init`, so both approaches are essentially equivalent.

**Caveat with Public Class Fields**

Adding a Public Class Field with the same name as one of the model's attribute is going to cause issues. Sequelize adds a getter & a setter for each attribute defined through `Model.init`. Adding a Public Class Field will shadow those getter and setters, blocking access to the model's actual data.

```
// Invalid
class User extends Model {
  id; // this field will shadow sequelize's getter & setter. It should be
removed.
  otherPublicField; // this field does not shadow anything. It is fine.
}

User.init(
  {
    id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true,
    },
  },
  { sequelize },
);

const user = new User({ id: 1 });
user.id; // undefined
```

```
// Valid
class User extends Model {
  otherPublicField;
```

```
}

User.init(
  {
    id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true,
    },
  },
  { sequelize },
);

const user = new User({ id: 1 });
user.id; // 1
```

In TypeScript, you can add typing information without adding an actual public class field by using the `declare` keyword:

```
// Valid
class User extends Model {
  declare id: number; // this is ok! The 'declare' keyword ensures this
field will not be emitted by TypeScript.
}

User.init(
  {
    id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true,
    },
  },
  { sequelize },
);

const user = new User({ id: 1 });
user.id; // 1
```

# Table name inference

Observe that, in both methods above, the table name (`Users`) was never explicitly defined. However, the model name was given (`User`).

By default, when the table name is not given, Sequelize automatically pluralizes the model name and uses that as the table name. This pluralization is done under the hood by a library called inflection, so that irregular plurals (such as `person -> people`) are computed correctly.

Of course, this behavior is easily configurable.

## Enforcing the table name to be equal to the model name

You can stop the auto-pluralization performed by Sequelize using the `freezeTableName: true` option. This way, Sequelize will infer the table name to be equal to the model name, without any modifications:

```
sequelize.define(
  'User',
  {
    // ... (attributes)
  },
  {
    freezeTableName: true,
  },
);
```

The example above will create a model named `User` pointing to a table also named `User`.

This behavior can also be defined globally for the sequelize instance, when it is created:

```
const sequelize = new Sequelize('sqlite::memory:', {
  define: {
    freezeTableName: true,
  },
});
```

This way, all tables will use the same name as the model name.

## Providing the table name directly

You can simply tell Sequelize the name of the table directly as well:

```
sequelize.define(
  'User',
```

```
  {
    // ... (attributes)
  },
  {
    tableName: 'Employees',
  },
);
```

# Model synchronization

When you define a model, you're telling Sequelize a few things about its table in the database. However, what if the table actually doesn't even exist in the database? What if it exists, but it has different columns, less columns, or any other difference?

This is where model synchronization comes in. A model can be synchronized with the database by calling `model.sync(options)`, an asynchronous function (that returns a Promise). With this call, Sequelize will automatically perform an SQL query to the database. Note that this changes only the table in the database, not the model in the JavaScript side.

- `User.sync()` - This creates the table if it doesn't exist (and does nothing if it already exists)
- `User.sync({ force: true })` - This creates the table, dropping it first if it already existed
- `User.sync({ alter: true })` - This checks what is the current state of the table in the database (which columns it has, what are their data types, etc), and then performs the necessary changes in the table to make it match the model.

Example:

```
await User.sync({ force: true });
console.log('The table for the User model was just (re)created!');
```

## Synchronizing all models at once

You can use `sequelize.sync()` to automatically synchronize all models. Example:

```
await sequelize.sync({ force: true });
console.log('All models were synchronized successfully.');
```

## Dropping tables

To drop the table related to a model:

```
await User.drop();
console.log('User table dropped!');
```

To drop all tables:

```
await sequelize.drop();
console.log('All tables dropped!');
```

## Database safety check

As shown above, the `sync` and `drop` operations are destructive. Sequelize accepts a `match` option as an additional safety check, which receives a RegExp:

```
// This will run .sync() only if database name ends with '_test'
sequelize.sync({ force: true, match: /_test$/ });
```

## Synchronization in production

As shown above, `sync({ force: true })` and `sync({ alter: true })` can be destructive operations. Therefore, they are not recommended for production-level software. Instead, synchronization should be done with the advanced concept of Migrations, with the help of the Sequelize CLI.

# Timestamps

By default, Sequelize automatically adds the fields `createdAt` and `updatedAt` to every model, using the data type `DataTypes.DATE`. Those fields are automatically managed as well - whenever you use Sequelize to create or update something, those fields will be set correctly. The `createdAt` field will contain the timestamp representing the moment of creation, and the `updatedAt` will contain the timestamp of the latest update.

**Note:** This is done in the Sequelize level (i.e. not done with *SQL triggers*). This means that direct SQL queries (for example queries performed without Sequelize by any other means) will not

cause these fields to be updated automatically.

This behavior can be disabled for a model with the `timestamps: false` option:

```
sequelize.define(
  'User',
  {
    // ... (attributes)
  },
  {
    timestamps: false,
  },
);
```

It is also possible to enable only one of `createdAt`/`updatedAt`, and to provide a custom name for these columns:

```
class Foo extends Model {}
Foo.init(
  {
    /* attributes */
  },
  {
    sequelize,

    // don't forget to enable timestamps!
    timestamps: true,

    // I don't want createdAt
    createdAt: false,

    // I want updatedAt to actually be called updateTimestamp
    updatedAt: 'updateTimestamp',
  },
);
```

# Column declaration shorthand syntax

If the only thing being specified about a column is its data type, the syntax can be shortened:

```
// This:
sequelize.define('User', {
  name: {
    type: DataTypes.STRING,
  },
});

// Can be simplified to:
sequelize.define('User', { name: DataTypes.STRING });
```

# Default Values

By default, Sequelize assumes that the default value of a column is `NULL`. This behavior can be changed by passing a specific `defaultValue` to the column definition:

```
sequelize.define('User', {
  name: {
    type: DataTypes.STRING,
    defaultValue: 'John Doe',
  },
});
```

Some special values, such as `DataTypes.NOW`, are also accepted:

```
sequelize.define('Foo', {
  bar: {
    type: DataTypes.DATETIME,
    defaultValue: DataTypes.NOW,
    // This way, the current date/time will be used to populate this column
(at the moment of insertion)
  },
});
```

# Data Types

Every column you define in your model must have a data type. Sequelize provides a lot of built-in data types. To access a built-in data type, you must import `DataTypes`:

```
const { DataTypes } = require('sequelize'); // Import the built-in data
types
```

## Strings

```
DataTypes.STRING; // VARCHAR(255)
DataTypes.STRING(1234); // VARCHAR(1234)
DataTypes.STRING.BINARY; // VARCHAR BINARY
DataTypes.TEXT; // TEXT
DataTypes.TEXT('tiny'); // TINYTEXT
DataTypes.CITEXT; // CITEXT          PostgreSQL and SQLite only.
DataTypes.TSVECTOR; // TSVECTOR        PostgreSQL only.
```

## Boolean

```
DataTypes.BOOLEAN; // TINYINT(1)
```

## Numbers

```
DataTypes.INTEGER; // INTEGER
DataTypes.BIGINT; // BIGINT
DataTypes.BIGINT(11); // BIGINT(11)

DataTypes.FLOAT; // FLOAT
DataTypes.FLOAT(11); // FLOAT(11)
DataTypes.FLOAT(11, 10); // FLOAT(11,10)

DataTypes.REAL; // REAL            PostgreSQL only.
DataTypes.REAL(11); // REAL(11)        PostgreSQL only.
DataTypes.REAL(11, 12); // REAL(11,12)     PostgreSQL only.

DataTypes.DOUBLE; // DOUBLE
DataTypes.DOUBLE(11); // DOUBLE(11)
DataTypes.DOUBLE(11, 10); // DOUBLE(11,10)

DataTypes.DECIMAL; // DECIMAL
DataTypes.DECIMAL(10, 2); // DECIMAL(10,2)
```

#### Unsigned & Zerofill integers - MySQL/MariaDB only

In MySQL and MariaDB, the data types `INTEGER`, `BIGINT`, `FLOAT` and `DOUBLE` can be set as unsigned or zerofill (or both), as follows:

```
DataTypes.INTEGER.UNSIGNED;
DataTypes.INTEGER.ZEROFILL;
DataTypes.INTEGER.UNSIGNED.ZEROFILL;
// You can also specify the size i.e. INTEGER(10) instead of simply INTEGER
// Same for BIGINT, FLOAT and DOUBLE
```

### Dates

```
DataTypes.DATE; // DATETIME for mysql / sqlite, TIMESTAMP WITH TIME ZONE for postgres
DataTypes.DATE(6); // DATETIME(6) for mysql 5.6.4+. Fractional seconds support with up to 6 digits of precision
DataTypes.DATEONLY; // DATE without time
```

### UUIDs

For UUIDs, use `DataTypes.UUID`. It becomes the `UUID` data type for PostgreSQL and SQLite, and `CHAR(36)` for MySQL. Sequelize can generate UUIDs automatically for these fields, simply use `DataTypes.UUIDV1` or `DataTypes.UUIDV4` as the default value:

```
{
  type: DataTypes.UUID,
  defaultValue: DataTypes.UUIDV4 // Or DataTypes.UUIDV1
}
```

### Others

There are other data types, covered in a separate guide.

## Column Options

When defining a column, apart from specifying the `type` of the column, and the `allowNull` and `defaultValue` options mentioned above, there are a lot more options that can be used. Some examples are below.

```
const { Model, DataTypes, Deferrable } = require('sequelize');

class Foo extends Model {}
Foo.init(
  {
    // instantiating will automatically set the flag to true if not set
    flag: { type: DataTypes.BOOLEAN, allowNull: false, defaultValue: true },

    // default values for dates => current time
    myDate: { type: DataTypes.DATE, defaultValue: DataTypes.NOW },

    // setting allowNull to false will add NOT NULL to the column, which
means an error will be
    // thrown from the DB when the query is executed if the column is null.
If you want to check that a value
    // is not null before querying the DB, look at the validations section
below.
    title: { type: DataTypes.STRING, allowNull: false },

    // Creating two objects with the same value will throw an error. The
unique property can be either a
    // boolean, or a string. If you provide the same string for multiple
columns, they will form a
    // composite unique key.
    uniqueOne: { type: DataTypes.STRING, unique: 'compositeIndex' },
    uniqueTwo: { type: DataTypes.INTEGER, unique: 'compositeIndex' },

    // The unique property is simply a shorthand to create a unique
constraint.
    someUnique: { type: DataTypes.STRING, unique: true },

    // Go on reading for further information about primary keys
    identifier: { type: DataTypes.STRING, primaryKey: true },

    // autoIncrement can be used to create auto_incrementing integer columns
    incrementMe: { type: DataTypes.INTEGER, autoIncrement: true },

    // You can specify a custom column name via the 'field' attribute:
    fieldWithUnderscores: {
      type: DataTypes.STRING,
      field: 'field_with_underscores',
    },

    // It is possible to create foreign keys:
    bar_id: {
      type: DataTypes.INTEGER,
```

```
      references: {
        // This is a reference to another model
        model: Bar,

        // This is the column name of the referenced model
        key: 'id',

        // With PostgreSQL, it is optionally possible to declare when to
check the foreign key constraint, passing the Deferrable type.
        deferrable: Deferrable.INITIALLY_IMMEDIATE,
        // Options:
        // - `Deferrable.INITIALLY_IMMEDIATE` - Immediately check the
foreign key constraints
        // - `Deferrable.INITIALLY_DEFERRED` - Defer all foreign key
constraint check to the end of a transaction
        // - `Deferrable.NOT` - Don't defer the checks at all (default) -
This won't allow you to dynamically change the rule in a transaction
      },
    },

    // Comments can only be added to columns in MySQL, MariaDB, PostgreSQL
and MSSQL
    commentMe: {
      type: DataTypes.INTEGER,
      comment: 'This is a column name that has a comment',
    },
  },
  {
    sequelize,
    modelName: 'foo',

    // Using `unique: true` in an attribute above is exactly the same as
creating the index in the model's options:
    indexes: [{ unique: true, fields: ['someUnique'] }],
  },
);
```

# Taking advantage of Models being classes

The Sequelize models are ES6 classes. You can very easily add custom instance or class level methods.

```
class User extends Model {
  static classLevelMethod() {
    return 'foo';
  }
  instanceLevelMethod() {
    return 'bar';
  }
  getFullname() {
    return [this.firstname, this.lastname].join(' ');
  }
}
User.init(
  {
    firstname: Sequelize.TEXT,
    lastname: Sequelize.TEXT,
  },
  { sequelize },
);

console.log(User.classLevelMethod()); // 'foo'
const user = User.build({ firstname: 'Jane', lastname: 'Doe' });
console.log(user.instanceLevelMethod()); // 'bar'
console.log(user.getFullname()); // 'Jane Doe'
```



*Last updated on **Feb 28, 2025** by **renovate[bot]***