Model definition

To define mappings between a model and a table, use the `define` method. Each column must have a datatype, see more about datatypes.

```
class Project extends Model {}
Project.init({
  title: Sequelize.STRING,
  description: Sequelize.TEXT
}, { sequelize, modelName: 'project' });

class Task extends Model {}
Task.init({
  title: Sequelize.STRING,
  description: Sequelize.TEXT,
  deadline: Sequelize.DATE
}, { sequelize, modelName: 'task' })
```

Apart from datatypes, there are plenty of options that you can set on each column.

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

 // default values for dates => current time
 myDate: { type: Sequelize.DATE, defaultValue: Sequelize.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: Sequelize.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: Sequelize.STRING,  unique: 'compositeIndex' },
 uniqueTwo: { type: Sequelize.INTEGER, unique: 'compositeIndex' },

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

 // It's exactly the same as creating the index in the model's options.
 { someUnique: { type: Sequelize.STRING } },
 { indexes: [ { unique: true, fields: [ 'someUnique' ] } ] },

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

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

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

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

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

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

      // This declares when to check the foreign key constraint. PostgreSQL only.
      deferrable: Sequelize.Deferrable.INITIALLY_IMMEDIATE
    }
  },

  // It is possible to add comments on columns for MySQL, PostgreSQL and MSSQL only
  commentMe: {
    type: Sequelize.INTEGER,

    comment: 'This is a column name that has a comment'
  }
}, {
  sequelize,
  modelName: 'foo'
});
```

The comment option can also be used on a table, see model configuration.

Timestamps

By default, Sequelize will add the attributes `createdAt` and `updatedAt` to your model so you will be able to know when the database entry went into the db and when it was updated last. Note that if you are using Sequelize migrations you will need to add the `createdAt` and `updatedAt` fields to your migration definition:

```
module.exports = {
  up(queryInterface, Sequelize) {
    return queryInterface.createTable('my-table', {
      id: {
        type: Sequelize.INTEGER,
        primaryKey: true,
        autoIncrement: true,
      },

      // Timestamps
      createdAt: Sequelize.DATE,
      updatedAt: Sequelize.DATE,
    })
  },
  down(queryInterface, Sequelize) {
    return queryInterface.dropTable('my-table');
  },
}
```

If you do not want timestamps on your models, only want some timestamps, or you are working with an existing database where the columns are named something else, jump straight on to configuration to see how to do that.

Deferrable

When you specify a foreign key column it is optionally possible to declare the deferrable type in PostgreSQL. The following options are available:

```
// Defer all foreign key constraint check to the end of a transaction
Sequelize.Deferrable.INITIALLY_DEFERRED

// Immediately check the foreign key constraints
Sequelize.Deferrable.INITIALLY_IMMEDIATE

// Don't defer the checks at all
Sequelize.Deferrable.NOT
```

The last option is the default in PostgreSQL and won't allow you to dynamically change the rule in a transaction. See the transaction section for further information.

Getters & setters

It is possible to define 'object-property' getters and setter functions on your models, these can be used both for 'protecting' properties that map to database fields and for defining 'pseudo' properties.

Getters and Setters can be defined in 2 ways (you can mix and match these 2 approaches):

- as part of a single property definition
- as part of a model options

N.B: If a getter or setter is defined in both places then the function found in the relevant property definition will always take precedence.

Defining as part of a property

```
class Employee extends Model {}
Employee.init({
  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());
    }
  }
}, { sequelize, modelName: 'employee' });
```

```
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
  })
```

Defining as part of the model options

Below is an example of defining the getters and setters in the model options.

The `fullName` getter, is an example of how you can define pseudo properties on your models - attributes which are not actually part of your database schema. In fact, pseudo properties can be defined in two ways: using model getters, or by using a column with the `VIRTUAL` datatype. Virtual datatypes can have validations, while getters for virtual attributes cannot.
Note that the `this.firstname` and `this.lastname` references in the `fullName` getter function will trigger a call to the respective getter functions. If you do not want that then use the `getDataValue()` method to access the raw value (see below).

```
class Foo extends Model {
  get fullName() {
    return this.firstname + ' ' + this.lastname;
  }

  set fullName(value) {
    const names = value.split(' ');
    this.setDataValue('firstname', names.slice(0, -1).join(' '));
    this.setDataValue('lastname', names.slice(-1).join(' '));
  }
}
Foo.init({
  firstname: Sequelize.STRING,
  lastname: Sequelize.STRING
}, {
  sequelize,
  modelName: 'foo'
});

// Or with `sequelize.define`
sequelize.define('Foo', {
  firstname: Sequelize.STRING,
  lastname: Sequelize.STRING
}, {
  getterMethods: {
    fullName() {
      return this.firstname + ' ' + this.lastname;
    }
  },

  setterMethods: {
    fullName(value) {
      const names = value.split(' ');

      this.setDataValue('firstname', names.slice(0, -1).join(' '));
      this.setDataValue('lastname', names.slice(-1).join(' '));
    }
```

```
  }
});
```

Helper functions for use inside getter and setter definitions

- retrieving an underlying property value - always use `this.getDataValue()`
```
/* a getter for 'title' property */
get() {
  return this.getDataValue('title')
}
```
- setting an underlying property value - always use `this.setDataValue()`
```
/* a setter for 'title' property */
set(title) {
  this.setDataValue('title', title.toString().toLowerCase());
}
```
N.B: It is important to stick to using the `setDataValue()` and `getDataValue()` functions (as opposed to accessing the underlying "data values" property directly) - doing so protects your custom getters and setters from changes in the underlying model implementations.

Validations

Model validations 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, as shown below.

```
class ValidateMe extends Model {}
ValidateMe.init({
  bar: {
    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"
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.');
        }
      }
    }
  }
}, { sequelize });
```

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 `[['one', 'two']]` 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.

Per-attribute validators and `allowNull`

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: Sequelize.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: Sequelize.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: Sequelize.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 Pub extends Model {}
Pub.init({
  name: { type: Sequelize.STRING },
  address: { type: Sequelize.STRING },
  latitude: {
    type: Sequelize.INTEGER,
    allowNull: true,
    defaultValue: null,
    validate: { min: -90, max: 90 }
  },
  longitude: {
    type: Sequelize.INTEGER,
    allowNull: true,
    defaultValue: null,
    validate: { min: -180, max: 180 }
  },
}, {
  validate: {
    bothCoordsOrNone() {
      if ((this.latitude === null) !== (this.longitude === null)) {
        throw new Error('Require either both latitude and longitude or neither')
      }
    }
  },
  sequelize,
})
```

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, `raging_bullock_arms.validate()` might return

```
{
  'latitude': ['Invalid number: latitude'],
  'bothCoordsOrNone': ['Require 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.

You can also influence the way Sequelize handles your column names:

```
class Bar extends Model {}
Bar.init({ /* bla */ }, {
  // The name of the model. The model will be stored in `sequelize.models` under this
name.
  // This defaults to class name i.e. Bar in this case. This will control name of
auto-generated
  // foreignKey and association naming
  modelName: 'bar',

  // don't add the timestamp attributes (updatedAt, createdAt)
  timestamps: false,

  // don't delete database entries but set the newly added attribute deletedAt
  // to the current date (when deletion was done). paranoid will only work if
  // timestamps are enabled
  paranoid: true,

  // Will automatically set field option for all attributes to snake cased name.
  // Does not override attribute with field option already defined
  underscored: true,

  // disable the modification of table names; By default, sequelize will
automatically
  // transform all passed model names (first parameter of define) into plural.
  // if you don't want that, set the following
  freezeTableName: true,

  // define the table's name
  tableName: 'my_very_custom_table_name',

  // Enable optimistic locking.  When enabled, sequelize will add a version count
attribute
  // to the model and throw an OptimisticLockingError error when stale instances are
saved.
  // Set to true or a string with the attribute name you want to use to enable.
  version: true,

  // Sequelize instance
  sequelize,
})
```

If you want sequelize to handle timestamps, but only want some of them, or want your timestamps to be called something else, you can override each column individually:

```
class Foo extends Model {}
Foo.init({ /* bla */ }, {
  // 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',
```

```
  // And deletedAt to be called destroyTime (remember to enable paranoid for this to
work)
  deletedAt: 'destroyTime',
  paranoid: true,

  sequelize,
})
```

You can also change the database engine, e.g. to MyISAM. InnoDB is the default.

```
class Person extends Model {}
Person.init({ /* attributes */ }, {
  engine: 'MYISAM',
  sequelize
})

// or globally
const sequelize = new Sequelize(db, user, pw, {
  define: { engine: 'MYISAM' }
})
```

Finally you can specify a comment for the table in MySQL and PG

```
class Person extends Model {}
Person.init({ /* attributes */ }, {
  comment: "I'm a table comment!",
  sequelize
})
```

Import

You can also store your model definitions in a single file using the `import` method. The returned object is exactly the same as defined in the imported file's function. Since `v1:5.0` of Sequelize the import is cached, so you won't run into troubles when calling the import of a file twice or more often.

```
// in your server file - e.g. app.js
const Project = sequelize.import(__dirname + "/path/to/models/project")

// The model definition is done in /path/to/models/project.js
// As you might notice, the DataTypes are the very same as explained above
module.exports = (sequelize, DataTypes) => {
  class Project extends sequelize.Model { }
  Project.init({
    name: DataTypes.STRING,
    description: DataTypes.TEXT
  }, { sequelize });
  return Project;
}
```

The `import` method can also accept a callback as an argument.

```
sequelize.import('project', (sequelize, DataTypes) => {
  class Project extends sequelize.Model {}
  Project.init({
    name: DataTypes.STRING,
    description: DataTypes.TEXT
  }, { sequelize })
  return Project;
})
```

This extra capability is useful when, for example, `Error: Cannot find module` is thrown even though `/path/to/models/project` seems to be correct. Some frameworks, such as Meteor, overload `require`, and spit out "surprise" results like :

```
Error: Cannot find module
'/home/you/meteorApp/.meteor/local/build/programs/server/app/path/to/models/project.j
s'
```

This is solved by passing in Meteor's version of `require`. So, while this probably fails ...

```
const AuthorModel = db.import('./path/to/models/project');
```

... this should succeed ...

```
const AuthorModel = db.import('project', require('./path/to/models/project'));
```

Optimistic Locking

Sequelize has built-in support for optimistic locking through a model instance version count. Optimistic locking is disabled by default and can be enabled by setting the `version` property to true in a specific model definition or global model configuration. See model configuration for more details.

Optimistic locking allows concurrent access to model records for edits and prevents conflicts from overwriting data. It does this by checking whether another process has made changes to a record since it was read and throws an OptimisticLockError when a conflict is detected.

Database synchronization

When starting a new project you won't have a database structure and using Sequelize you won't need to. Just specify your model structures and let the library do the rest. Currently supported is the creation and deletion of tables:

```
// Create the tables:
Project.sync()
Task.sync()

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

// drop the tables:
Project.drop()
Task.drop()

// event handling:
Project.[sync|drop]().then(() => {
  // ok ... everything is nice!
}).catch(error => {
  // oooh, did you enter wrong database credentials?
})
```

Because synchronizing and dropping all of your tables might be a lot of lines to write, you can also let Sequelize do the work for you:

```
// Sync all models that aren't already in the database
sequelize.sync()
// Force sync all models
sequelize.sync({force: true})
// Drop all tables
sequelize.drop()
// emit handling:
sequelize.[sync|drop]().then(() => {
  // woot woot
}).catch(error => {
  // whooops
})
```

Because `.sync({ force: true })` is destructive operation, you can use `match` option as an additional safety check. `match` option tells sequelize to match a regex against the database name before syncing - a safety check for cases where `force: true` is used in tests but not live code.

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

Expansion of models

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

```
class User extends Model {
  // Adding a class level method
  static classLevelMethod() {
    return 'foo';
  }

  // Adding an instance level method
  instanceLevelMethod() {
    return 'bar';
  }
}
User.init({ firstname: Sequelize.STRING }, { sequelize });
```

Of course you can also access the instance's data and generate virtual getters:

```
class User extends Model {
  getFullname() {
    return [this.firstname, this.lastname].join(' ');
  }
}
User.init({ firstname: Sequelize.STRING, lastname: Sequelize.STRING }, { sequelize
});

// Example:
User.build({ firstname: 'foo', lastname: 'bar' }).getFullname() // 'foo bar'
```

Indexes

Sequelize supports adding indexes to the model definition which will be created during `Model.sync()` or `sequelize.sync`.

```
class User extends Model {}
User.init({}, {
  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}]
    }
  ],
  sequelize
});
```

Data Types

Below are some of the datatypes supported by sequelize. For a full and updated list, see DataTypes.

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

Sequelize.INTEGER                     // INTEGER
Sequelize.BIGINT                      // BIGINT
Sequelize.BIGINT(11)                  // BIGINT(11)

Sequelize.FLOAT                       // FLOAT
Sequelize.FLOAT(11)                   // FLOAT(11)
Sequelize.FLOAT(11, 10)               // FLOAT(11,10)

Sequelize.REAL                        // REAL        PostgreSQL only.
Sequelize.REAL(11)                    // REAL(11)    PostgreSQL only.
Sequelize.REAL(11, 12)                // REAL(11,12) PostgreSQL only.

Sequelize.DOUBLE                      // DOUBLE
Sequelize.DOUBLE(11)                  // DOUBLE(11)
Sequelize.DOUBLE(11, 10)              // DOUBLE(11,10)

Sequelize.DECIMAL                     // DECIMAL
Sequelize.DECIMAL(10, 2)              // DECIMAL(10,2)

Sequelize.DATE                        // DATETIME for mysql / sqlite, TIMESTAMP WITH TIME ZONE
for postgres
Sequelize.DATE(6)                     // DATETIME(6) for mysql 5.6.4+. Fractional seconds
support with up to 6 digits of precision
Sequelize.DATEONLY                    // DATE without time.
Sequelize.BOOLEAN                     // TINYINT(1)

Sequelize.ENUM('value 1', 'value 2')  // An ENUM with allowed values 'value 1' and 'value 2'
Sequelize.ARRAY(Sequelize.TEXT)       // Defines an array. PostgreSQL only.
Sequelize.ARRAY(Sequelize.ENUM)       // Defines an array of ENUM. PostgreSQL only.

Sequelize.JSON                        // JSON column. PostgreSQL, SQLite and MySQL only.
Sequelize.JSONB                       // JSONB column. PostgreSQL only.

Sequelize.BLOB                        // BLOB (bytea for PostgreSQL)
Sequelize.BLOB('tiny')                // TINYBLOB (bytea for PostgreSQL. Other options are
medium and long)

Sequelize.UUID                        // UUID datatype for PostgreSQL and SQLite, CHAR(36)
BINARY for MySQL (use defaultValue: Sequelize.UUIDV1 or Sequelize.UUIDV4 to make sequelize
generate the ids automatically)

Sequelize.CIDR                        // CIDR datatype for PostgreSQL
Sequelize.INET                        // INET datatype for PostgreSQL
Sequelize.MACADDR                     // MACADDR datatype for PostgreSQL

Sequelize.RANGE(Sequelize.INTEGER)    // Defines int4range range. PostgreSQL only.
Sequelize.RANGE(Sequelize.BIGINT)     // Defined int8range range. PostgreSQL only.
Sequelize.RANGE(Sequelize.DATE)       // Defines tstzrange range. PostgreSQL only.
Sequelize.RANGE(Sequelize.DATEONLY)   // Defines daterange range. PostgreSQL only.
Sequelize.RANGE(Sequelize.DECIMAL)    // Defines numrange range. PostgreSQL only.

Sequelize.ARRAY(Sequelize.RANGE(Sequelize.DATE)) // Defines array of tstzrange ranges.
PostgreSQL only.

Sequelize.GEOMETRY                    // Spatial column.  PostgreSQL (with PostGIS) or MySQL
only.
```

```
Sequelize.GEOMETRY('POINT')           // Spatial column with geometry type. PostgreSQL (with
PostGIS) or MySQL only.
Sequelize.GEOMETRY('POINT', 4326)     // Spatial column with geometry type and SRID.
PostgreSQL (with PostGIS) or MySQL only.
```

The BLOB datatype allows you to insert data both as strings and as buffers. When you do a find or findAll on a model which has a BLOB column, that data will always be returned as a buffer.

If you are working with the PostgreSQL TIMESTAMP WITHOUT TIME ZONE and you need to parse it to a different timezone, please use the pg library's own parser:

```
require('pg').types.setTypeParser(1114, stringValue => {
  return new Date(stringValue + '+0000');
  // e.g., UTC offset. Use any offset that you would like.
});
```

In addition to the type mentioned above, integer, bigint, float and double also support unsigned and zerofill properties, which can be combined in any order: Be aware that this does not apply for PostgreSQL!

```
Sequelize.INTEGER.UNSIGNED              // INTEGER UNSIGNED
Sequelize.INTEGER(11).UNSIGNED          // INTEGER(11) UNSIGNED
Sequelize.INTEGER(11).ZEROFILL          // INTEGER(11) ZEROFILL
Sequelize.INTEGER(11).ZEROFILL.UNSIGNED // INTEGER(11) UNSIGNED ZEROFILL
Sequelize.INTEGER(11).UNSIGNED.ZEROFILL // INTEGER(11) UNSIGNED ZEROFILL
```

The examples above only show integer, but the same can be done with bigint and float

Usage in object notation:

```
// for enums:
class MyModel extends Model {}
MyModel.init({
  states: {
    type: Sequelize.ENUM,
    values: ['active', 'pending', 'deleted']
  }
}, { sequelize })
```

Array(ENUM)

Its only supported with PostgreSQL.

Array(Enum) type require special treatment. Whenever Sequelize will talk to database it has to typecast Array values with ENUM name.

So this enum name must follow this pattern `enum_<table_name>_<col_name>`. If you are using `sync` then correct name will automatically be generated.

Range types

Since range types have extra information for their bound inclusion/exclusion it's not very straightforward to just use a tuple to represent them in javascript.

When supplying ranges as values you can choose from the following APIs:

```
// defaults to '["2016-01-01 00:00:00+00:00", "2016-02-01 00:00:00+00:00")'
// inclusive lower bound, exclusive upper bound
Timeline.create({ range: [new Date(Date.UTC(2016, 0, 1)), new Date(Date.UTC(2016, 1,
1))] });

// control inclusion
const range = [
  { value: new Date(Date.UTC(2016, 0, 1)), inclusive: false },
  { value: new Date(Date.UTC(2016, 1, 1)), inclusive: true },
];
// '("2016-01-01 00:00:00+00:00", "2016-02-01 00:00:00+00:00"]'

// composite form
const range = [
  { value: new Date(Date.UTC(2016, 0, 1)), inclusive: false },
  new Date(Date.UTC(2016, 1, 1)),
];
// '("2016-01-01 00:00:00+00:00", "2016-02-01 00:00:00+00:00")'

Timeline.create({ range });
```

However, please note that whenever you get back a value that is range you will receive:

```
// stored value: ("2016-01-01 00:00:00+00:00", "2016-02-01 00:00:00+00:00"]
range // [{ value: Date, inclusive: false }, { value: Date, inclusive: true }]
```

You will need to call reload after updating an instance with a range type or use `returning: true` option.

Special Cases

```
// empty range:
Timeline.create({ range: [] }); // range = 'empty'

// Unbounded range:
Timeline.create({ range: [null, null] }); // range = '[,)'
// range = '[,"2016-01-01 00:00:00+00:00")'
Timeline.create({ range: [null, new Date(Date.UTC(2016, 0, 1))] });

// Infinite range:
// range = '[-infinity,"2016-01-01 00:00:00+00:00")'
Timeline.create({ range: [-Infinity, new Date(Date.UTC(2016, 0, 1))] });
```

Extending datatypes

Most likely the type you are trying to implement is already included in DataTypes. If a new datatype is not included, this manual will show how to write it yourself.
Sequelize doesn't create new datatypes in the database. This tutorial explains how to make Sequelize recognize new datatypes and assumes that those new datatypes are already created in the database.

To extend Sequelize datatypes, do it before any instance is created. This example creates a dummy `NEWTYPE` that replicates the built-in datatype `Sequelize.INTEGER(11).ZEROFILL.UNSIGNED`.

```
// myproject/lib/sequelize.js

const Sequelize = require('Sequelize');
const sequelizeConfig = require('../config/sequelize')
const sequelizeAdditions = require('./sequelize-additions')

// Function that adds new datatypes
sequelizeAdditions(Sequelize)

// In this exmaple a Sequelize instance is created and exported
const sequelize = new Sequelize(sequelizeConfig)

module.exports = sequelize
// myproject/lib/sequelize-additions.js

module.exports = function sequelizeAdditions(Sequelize) {

  DataTypes = Sequelize.DataTypes

  /*
   * Create new types
   */
  class NEWTYPE extends DataTypes.ABSTRACT {
    // Mandatory, complete definition of the new type in the database
    toSql() {
      return 'INTEGER(11) UNSIGNED ZEROFILL'
    }

    // Optional, validator function
    validate(value, options) {
      return (typeof value === 'number') && (! Number.isNaN(value))
    }

    // Optional, sanitizer
    _sanitize(value) {
      // Force all numbers to be positive
      if (value < 0) {
        value = 0
      }

      return Math.round(value)
    }
```

```
    // Optional, value stringifier before sending to database
    _stringify(value) {
      return value.toString()
    }

    // Optional, parser for values received from the database
    static parse(value) {
      return Number.parseInt(value)
    }
  }

  DataTypes.NEWTYPE = NEWTYPE;

  // Mandatory, set key
  DataTypes.NEWTYPE.prototype.key = DataTypes.NEWTYPE.key = 'NEWTYPE'

  // Optional, disable escaping after stringifier. Not recommended.
  // Warning: disables Sequelize protection against SQL injections
  // DataTypes.NEWTYPE.escape = false

  // For convenience
  // `classToInvokable` allows you to use the datatype without `new`
  Sequelize.NEWTYPE = Sequelize.Utils.classToInvokable(DataTypes.NEWTYPE)

}
```

After creating this new datatype, you need to map this datatype in each database dialect and make some adjustments.

Let's say the name of the new datatype is `pg_new_type` in the postgres database. That name has to be mapped to `DataTypes.NEWTYPE`. Additionally, it is required to create a child postgres-specific datatype.

```
// myproject/lib/sequelize-additions.js

module.exports = function sequelizeAdditions(Sequelize) {

  DataTypes = Sequelize.DataTypes

  /*
   * Create new types
   */

  ...

  /*
   * Map new types
   */

  // Mandatory, map postgres datatype name
  DataTypes.NEWTYPE.types.postgres = ['pg_new_type']

  // Mandatory, create a postgres-specific child datatype with its own parse
  // method. The parser will be dynamically mapped to the OID of pg_new_type.
  PgTypes = DataTypes.postgres

  PgTypes.NEWTYPE = function NEWTYPE() {
    if (!(this instanceof PgTypes.NEWTYPE)) return new PgTypes.NEWTYPE();
    DataTypes.NEWTYPE.apply(this, arguments);
  }
  inherits(PgTypes.NEWTYPE, DataTypes.NEWTYPE);

  // Mandatory, create, override or reassign a postgres-specific parser
  //PgTypes.NEWTYPE.parse = value => value;
  PgTypes.NEWTYPE.parse = DataTypes.NEWTYPE.parse;

  // Optional, add or override methods of the postgres-specific datatype
  // like toSql, escape, validate, _stringify, _sanitize...

}
```

Ranges

After a new range type has been defined in postgres, it is trivial to add it to Sequelize.
In this example the name of the postgres range type is `newtype_range` and the name of the underlying postgres datatype is `pg_new_type`. The key of `subtypes` and `castTypes` is the key of the Sequelize datatype `DataTypes.NEWTYPE.key`, in lower case.

```
// myproject/lib/sequelize-additions.js

module.exports = function sequelizeAdditions(Sequelize) {

  DataTypes = Sequelize.DataTypes

  /*
   * Create new types
   */

  ...

  /*
   * Map new types
   */

  ...

  /*
   * Add support for ranges
   */

  // Add postgresql range, newtype comes from DataType.NEWTYPE.key in lower case
  DataTypes.RANGE.types.postgres.subtypes.newtype = 'newtype_range';
  DataTypes.RANGE.types.postgres.castTypes.newtype = 'pg_new_type';

}
```

The new range can be used in model definitions
as `Sequelize.RANGE(Sequelize.NEWTYPE)` or `DataTypes.RANGE(DataTypes.NEWTYPE)`.