Datatypes

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 t
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

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 My
Sequelize.GEOMETRY('POINT', 4326)   // Spatial column with geometry type and SRID.  PostgreSQL (with Post
```

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:

```
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')
```

```
// 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.

PostgreSQL

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']
```

```
  // 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 suport 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)`.
