JavaScript - ORMs - Sequelize

# General

## Overview

Sequelize uses models to abstract tables in the application database in a similar way to ActiveRecord.

## Setup

Add sequelize to the application using package manager:

yarn add sequelize

# Validations

## General

Validations are the checks to ensure only the correct format of data is inserted into a record, allowing the application to assume and design around the datatypes belonging to a model.

Validations are defined in the model and performed in JavaScrip on an application level in sequelize. The advantage of validations is that they can be arbitrarily complex, and if validations fail, no SQL query will be sent to the database.

To add a validations, add one of the available built in validations to the attribute to be validated, or create a custom validation. For example:

attribute: {

type: DataTypes.TEXT,

allowNull: false,

validate: {

is: /^[0-9]+$/i,

isEven(value) { // custom validator:

if (parseInt(value) % 2 !== 0) {

throw new Error('Only even values are allowed!')

},

}

}

},

Full list of built in validators: <https://github.com/validatorjs/validator.js>

To add custom messages to validations, add the value as a object with the options as args and the message as msg:

validate: {

is: {

args: /^[0-9]+$/i,

msg: 'Must be a number'

}

}

## Context Validations

Context validations are custom validations which apply to the whole model and therefore allow for validations of one or more attributes against eachother. For example, longitude and latitude attributes both having to be present for the model to be valid.

Context validations are defined in the model options object, for example:

model>> }, {

sequelize,

validate: {

bothCoordsOrNone() {

if ((this.latitude === null) !== (this.longitude === null)) {

throw new Error('Either both latitude and longitude, or neither!');

}

}

}

})

## Constraints

While constraints are similar to validations, they are rules defined at SQL level and stored in the database. Therefore, while a constraint will stop invalid data from entering the database, its complexity is limited to the database SQL language and if the data is invalid a query has been made the database and an error thrown, reducing application performance.

In sequelize, since some constraints can also be validations, they defined in the database migration and model. For example, the allowNull constraint:

username: {

type: DataTypes.TEXT,

allowNull: false,

unique: true

},

# Migrations

## General

Migrations can be used to keep track of changes to the application database, allowing different machines to ensure synchronousity between database states. Each migration defines a database state, with the actions to get to that state and reverse it.

## Setup

To create sequelize migrations, sequelize-cli is required. Add the package using the project package manager:

yarn add sequelize-cli

Initalize a new project using the init command:

yarn sequelize-cli init

Which will create the revelant folders:

- config - config file for connection to database

- models - models

- migrations - migration files

- seeders - seed files

Add commands to yarn:

"db:create": "sequelize-cli db:create",

"db:migrate": "yarn sequelize-cli db:migrate",

"db:reset": "yarn sequelize-cli db:migrate:undo:all && npm run migrate",

## Config

The configuration for how the ORM connects to the database is contained in config/config.js(on). Generally, there are three or four database environments:

* development
* test
* staging (if project requires)
* production

Add the configuration for each into the config file:

require('dotenv').config()

const fs = require('fs');

module.exports = {

development: {

username: process.env.DEV\_DB\_USER,

password: process.env.DEV\_DB\_PASSWORD,

database: process.env.DEV\_DB\_NAME,

host: process.env.DEV\_DB\_HOST,

dialect: "postgres"

},

test: {

username: process.env.TEST\_DB\_USER,

password: process.env.TEST\_DB\_PASSWORD,

database: process.env.TEST\_DB\_NAME,

host: process.env.TEST\_DB\_HOST,

dialect: "postgres",

logging: false

},

production: {

username: process.env.PROD\_DB\_USER,

password: process.env.PROD\_DB\_PASSWORD,

database: process.env.PROD\_DB\_NAME,

host: process.env.PROD\_DB\_HOST,

dialect: "postgres",

logging: false

}

}

Use .env environment variables to set dynamic database credentails.

The path to the configuration file and model, seeders, and migrations folders is set in the .sequelizerc file, which is placed in the root directory of the project. A typical .sequelizerc is:

const path = require('path');

module.exports = {

'config': path.resolve('config', 'database.json'),

'models-path': path.resolve('db', 'models'),

'seeders-path': path.resolve('db', 'seeders'),

'migrations-path': path.resolve('db', 'migrations')

};

## Create Migration

Use the model:generate command to create a model. Use the --name flag to define the model name, and the --attributes flag to comma seperated add attributes:

yarn sequelize-cli model:generate --name User --attributes email:string,name:string

All datatypes can be found here:

<https://sequelize.org/master/manual/model-basics.html#data-types>

Run the migration using the db:migrate command:

yarn sequelize-cli db:migrate

To undo the migration, use the undo command:

yarn sequelize-cli db:migrate:undo

To undo multiple migrations, use:

yarn sequelize-cli db:migrate:undo :all --to migration-file-name.js

## Migration Structure

Once migrations have been generated, they can be edited to specify exact changes to the database model. The typical structure of a migration is a module export, with an up and down method for migrating and undoing respectively. Each function gets passed the queryInterface and Sequelize object. The queryInterface can be used to modify the database using its various methods, for example, creating a model:

module.exports = {

up: (queryInterface, Sequelize) => {

return queryInterface.createTable('Person', {

name: Sequelize.DataTypes.STRING,

isBetaMember: {

type: Sequelize.DataTypes.BOOLEAN,

defaultValue: false,

allowNull: false

},

userId: {

type: Sequelize.DataTypes.INTEGER,

references: {

model: {

tableName: 'users',

schema: 'schema'

},

key: 'id'

},

allowNull: false

},

});

},

down: (queryInterface, Sequelize) => {

return queryInterface.dropTable('Person');

}

};

When running mutlitple migrations use automatically managed migrations to ensure all instructions are applied to the databsae, or rolled back if they fail:

module.exports = {

up: (queryInterface, Sequelize) => {

return queryInterface.sequelize.transaction(t => {

return Promise.all([

queryInterface.addColumn('Person', 'petName', {

type: Sequelize.DataTypes.STRING

}, { transaction: t }),

queryInterface.addColumn('Person', 'favoriteColor', {

type: Sequelize.DataTypes.STRING,

}, { transaction: t })

]);

});

},

down: (queryInterface, Sequelize) => {

return queryInterface.sequelize.transaction(t => {

return Promise.all([

queryInterface.removeColumn('Person', 'petName', { transaction: t }),

queryInterface.removeColumn('Person', 'favoriteColor', { transaction: t })

]);

});

}

};

## Create Seeds

Automatically add records to the database by running premade seeds. Add a seed using the seed:generate command:

yarn sequelize-cli seed:generate --name <seed-name>

Once generated, add the up and down details in the seed file:

module.exports = {

up: (queryInterface, Sequelize) => {

return queryInterface.bulkInsert('<model-name>', [{

email: 'example@example.com',

createdAt: new Date(),

updatedAt: new Date()

}]);

},

down: (queryInterface, Sequelize) => {

return queryInterface.bulkDelete('Users', null, {});

}

};

<https://sequelize.org/master/class/lib/dialects/abstract/query-interface.js~QueryInterface.html>

Run seeds using:

yarn sequelize-cli db:seed:all

Undo seeds:

yarn sequelize-cli db:seed:undo:all

# Confirmations

## General

Use middleware to verify confirmations and set values in request.

## Inputs

* no input, no confirmation => next(), allow validations to throw errors
* input or confirmation present =>
  + input == confirmation => apply confirmation to req => next()
  + input !== confirmation => return 422 error