Render and Sequelize

While using Render to deploy our Sequelize applications, some changes to our migration and seeder files are necessary.

Why? Well, Render's free tier only allows for one database, which we will need to share across each of your projects. But we need each project to have it's own set of tables and data. Postgres, the RDBMS you'll use in production, allows for the use of "schemas", which in this context is essentially a subdivision of a database. We can use schemas to isolate the tables and data into discrete sections.

Due to this separation of pieces of the database, when we create tables or insert data into those tables, we will need to tell Postgres which "schema" that table or data should be created in.

Every migration and seeder file will need to include the following block of code at the top of the file (outside the up/down functions):

```
let options = {};
if (process.env.NODE_ENV === 'production') {
  options.schema = process.env.SCHEMA; // define your schema in options object
}
```

When in production, this object will be used by Sequelize to determine which subdivision of your single database to execute the file on.

There are two ways we will apply this options object, and which method you use depends entirely on the `queryInterface` method being invoked.

createTable

When using the `queryInterface.createTable()` method, the options object will be passed in as the third argument. It should look like this:

```
return queryInterface.createTable("Users", {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      username: {
        type: Sequelize.STRING(30),
        allowNull: false,
        unique: true
      },
      email: {
        type: Sequelize.STRING(256),
        allowNull: false,
        unique: true
      },
      hashedPassword: {
        type: Sequelize.STRING.BINARY,
        allowNull: false
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE,
        defaultValue: Sequelize.literal('CURRENT_TIMESTAMP')
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE,
        defaultValue: Sequelize.literal('CURRENT_TIMESTAMP')
      }
    }, options);
```

All other queryInterface methods

For ALL other `queryInterface` method calls, you will instead pass the options object in as the first argument to the method call, replacing the table name string, like so:

```
await queryInterface.addColumn(options, 'firstName', {
  type: Sequelize.STRING(30)
});
```

```
await queryInterface.removeColumn(options, 'firstname')
```

Additionally, because we're replacing the table name argument in these method calls, we need to add a `tableName` property to the options object, with the table name we're dealing with as the value. You can place this at the top of the file (but outside the environment conditional), or individually in each up and down function.

For example, the file that contains the above up/down functions

```
'use strict';
/** @type {import('sequelize-cli').Migration} */
```

```
let options = {};
if (process.env.NODE_ENV === 'production') {
  options.schema = process.env.SCHEMA;  // define your schema in options object
}

module.exports = {
  async up(queryInterface, Sequelize) {
    options.tableName = "Users";
    await queryInterface.addColumn(options, 'firstName', {
      type: Sequelize.STRING(30)
    });

  },

  async down(queryInterface, Sequelize) {
    options.tableName = "Users";
    await queryInterface.removeColumn(options, 'firstname')
  }
};
```

Or:

```
'use strict';
/** @type {import('sequelize-cli').Migration} */

let options = {};
if (process.env.NODE_ENV === 'production') {
  options.schema = process.env.SCHEMA;  // define your schema in options object
}
options.tableName = "Users";

module.exports = {
  async up(queryInterface, Sequelize) {
    await queryInterface.addColumn(options, 'firstName', {
      type: Sequelize.STRING(30)
    });

  },

  async down(queryInterface, Sequelize) {
    await queryInterface.removeColumn(options, 'firstname')
  }
};
```

Note that seeder files will require the same. An example of a seeder file would look like this:

```
'use strict';
const bcrypt = require("bcryptjs");

let options = {};
if (process.env.NODE_ENV === 'production') {
  options.schema = process.env.SCHEMA;  // define your schema in options object
}

module.exports = {
  up: async (queryInterface, Sequelize) => {
    options.tableName = 'Users';
    return queryInterface.bulkInsert(options, [
      {
        email: 'demo@user.io',
        username: 'Demo-lition',
        hashedPassword: bcrypt.hashSync('password')
      },
    ], {});
  },

  down: async (queryInterface, Sequelize) => {
    options.tableName = 'Users';
    const Op = Sequelize.Op;
    return queryInterface.bulkDelete(options, {
      username: { [Op.in]: ['Demo-lition'] }
    }, {});
  }
};
```