WEEK-10 DAY-4
Sequelize ORM

Installing And Using Sequelize

Now that you have gained experience with SQL, it is time to learn how to access data stored in a SQL database using a JavaScript program. You will use a JavaScript library called Sequelize to do this. Sequelize is an example of an *Object Relational Mapping* (commonly abbreviated *ORM*). An ORM allows a JavaScript programmer to fetch and store data in a SQL database using JavaScript functions instead of writing SQL code.

When you finish this reading you will be able to:

- Describe what an Object Relational Mapping is and what it is used for.
- Install and configure the packages needed to use Sequelize.
- Use Sequelize to generate JavaScript code that fetches and stores data in a SQL database.
- Use those auto-generated methods to fetch and store data in a SQL database.

What Is An ORM?

An *Object Relational Mapping* is a library that allows you to access data stored in a SQL database through object-oriented, non-SQL code (such as JavaScript). You will write *object-oriented* code that accesses data stored in a *relational* SQL database like Postgres. The ORM is the *mapping* that will "translate" your object-oriented code into SQL code that the database can run. The ORM will automatically generate SQL code for common tasks like fetching and storing data.

You will learn how to use the Sequelize ORM.
Sequelize is the most widely used JavaScript ORM library.

How To Install Sequelize

After creating a new node project with `npm init` we are ready to install the Sequelize library.

```
npm install sequelize@^5.0.0
npm install sequelize-cli@^5.0.0
npm install pg@^8.0.0
```

We have installed not only the Sequelize library, but also a command line tool called `sequelize-cli` that will help us auto-generate and manage JavaScript files which will hold our Sequelize ORM code.

Last, we have also installed the pg library. This library allows Sequelize to access a Postgres database. If you were using a different database software (such as MySQL), you would need to install a different library.

How To Initialize Sequelize

We can run the command `npx sequelize init` to automatically setup the following directory structure for our project:

```
.
├── config
│   └── config.json
├── migrations
├── models
│   └── index.js
├── node_modules
├── package-lock.json
├── package.json
└── seeders
```

> Aside: the `npx` tool allows you to easily run scripts provided by packages like `sequelize-cli`. If you don't already have `npx`, you can install it with `npm install npx --global`. Without `npx` you would have to run the bash command: `./node_modules/.bin/sequelize init`. This directly runs the `sequelize` script provided by the installed `sequelize-cli` package.

Having run `npx sequelize init`, we must write our database login information into `config/config.json`.

By default this file contains different sections we call "environments". In a typical company you will have different database servers and configuration depending on where you app is running. Development is usually where you do

your development work. In our case this is our local computer. But test might be and environment where you run tests, and production is the environment where real users are interacting with your application.

Since we are doing development, we can just modify the "development" section to look like this:

```
{
  "development": {
    "username": "catsdbuser",
    "password": "catsdbpassword",
    "database": "catsdb",
    "host": "127.0.0.1",
    "dialect": "postgres"
  }
}...
```

Here we are supposing that we have already created a `catsdb` database owned by the user `catsdbuser`, with password `catsdbpassword`. By setting `host` to `127.0.0.1`, we are saying that the database will run on the same machine as my JavaScript application. Last, we specify that we are using a `postgres` database.

Verifying That Sequelize Can Connect To The Database

At the top level of our project, we should create an `index.js` file. From this file we will verify that Sequelize can connect to the SQL database. To do this, we use the `authenticate` method of the sequelize object.

```
// ./index.js

const { sequelize } = require("./models");

async function main() {
  try {
    await sequelize.authenticate();
  } catch (e) {
    console.log("Database connection failure.");
    console.log(e);
    return;
  }

  console.log("Database connection success!");
  console.log("Sequelize is ready to use!");

  // Close database connection when done with it.
  await sequelize.close();
}

main();

// Prints:
//
// Executing (default): SELECT 1+1 AS result
// Database connection success!
// Sequelize is ready to use!
```

You may observe that the `authenticate` method returns a JavaScript `Promise` object. We use `await` to wait for the database connection to be established. If `authenticate` fails to connect, the `Promise` will be rejected. Since we use `await`, an exception will be thrown.

Many Sequelize methods return `Promise`s. Using `async` and `await` lets us use Sequelize methods as if they were synchronous. This helps reduce code complexity significantly.

Note that I call `sequelize.close()`. This closes the connection to the database. A Node.js JavaScript program will not terminate until all open files and database connections are closed. Thus, to make sure the Node.js program doesn't "hang" at the end, we close the database connection. Otherwise we will be forced to kill the Node.js program with `CTRL-C`, which is somewhat annoying.

Our Preexisting Database Schema

We are assuming that we are working with a preexisting SQL database. Our `catsdb` has a single table: `Cats`. Using the `psql` command-line program, we can describe the pre-existing `Cats` table below.

```
catsdb=> \d "Cats"
                                    Table "public.Cats"
    Column    |           Type           | Collation | Nullable |              Default
--------------+--------------------------+-----------+----------+-----------------------------------
 id           | integer                  |           | not null | nextval('"Cats_id_seq"'::regclass)
 firstName    | character varying(255)   |           |          |
 specialSkill | character varying(255)   |           |          |
 age          | integer                  |           |          |
 createdAt    | timestamp with time zone |           | not null |
 updatedAt    | timestamp with time zone |           | not null |
Indexes:
    "Cats_pkey" PRIMARY KEY, btree (id)
```

Besides a primary key `id`, each `Cats` record has a `firstName`, a `specialSkill`, and an `age`. Each record also keeps track of two timestamps: the time when the cat was created (`createdAt`), and the most recent time when a column of the cat has been updated (`updatedAt`).

Using Sequelize To Generate The Model File

We will configure Sequelize to access the `Cats` table via a JavaScript class called `Cat`. To do this, we first use our trusty Sequelize CLI:

```
# Oops, forgot age:integer! (Don't worry we'll fix it later)
npx sequelize model:generate --name Cat --attributes "firstName:string,specialSkill:string"
```

This command generates two files: a *model* file (`./models/cat.js`) and a *migration* file (`./migrations/20200203211508-Cat.js`). We will ignore the migration file for now, and focus on the model file.

When using Sequelize's `model:generate` command, we specify two things. First: we specify the *singular* form of the `Cats` table name (`Cat`). Second: we list the columns of the `Cats` table after the `--attributes` flag: `firstName` and `specialSkill`. We tell Sequelize that these are both `string` columns (Sequelize calls SQL `character varying(255)` columns `string`s).

We do not need to list `id`, `createdAt`, or `updatedAt`. Sequelize will always presume those exist. Notice that we have **forgotten** to list `age:integer` -- we will fix that soon!

Examining (And Modifying) A Sequelize Model File

Let us examine the generated `./models/cat.js` file:

```
// ./models/cat.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: DataTypes.STRING,
    specialSkill: DataTypes.STRING
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

This file exports a function that defines a `Cat` class. When you use `Sequelize` to query the `Cats` table, each row retrieved will be transformed by Sequelize into an instance of the `Cat` class. A JavaScript class like `Cat` that corresponds to a SQL table is called a *model* class.

The `./models/cat.js` will not be loaded by us directly. Sequelize will load this file and call the exported function to define the `Cat` class. The exported function uses Sequelize's `define` method to auto-generate a new class (called `Cat`).

> Note: You may notice we aren't using the JavaScript's `class` keyword to define the Cat class. With Sequelize, it is going to do all that for us with the `define` method. This is because Sequelize was around way before the `class` keyword was added to JavaScript. It is possible to use the class keyword with Sequelize, but it's undocumented.

The first argument of `define` is the name of the class to define: `Cat`. Notice how the second argument is an `Object` of `Cats` table columns:

```
{
    firstName: DataTypes.STRING,
    specialSkill: DataTypes.STRING
}
```

This object tells Sequelize about each of the columns of `Cats`. It maps each column name (`firstName`, `specialSkill`) to the type of data stored in the corresponding column of the `Cats` table. It is unnecessary to list `id`, `createdAt`, `updatedAt`, since Sequelize will already assume those exist.

We can correct our earlier mistake of forgetting `age`. We update the definition as so:

```
const Cat = sequelize.define('Cat', {
  firstName: DataTypes.STRING,
  specialSkill: DataTypes.STRING,
  age: DataTypes.INTEGER,
}, {});
```

A complete list of Sequelize datatypes can be found in the documentation.

Using The `Cat` Model To Fetch And Update SQL Data

We are now ready to use our `Cat` model class. When Sequelize defines the `Cat` class, it will generate instance and class methods needed to interact with the `Cats` SQL table.

As we mentioned before we don't require our `cats.js` file directly. Instead we require `./models` which loads the file `./models/index.js`.

Inside this file it reads through all our models and attaches them to an object that it exports. So we can use destructuring to get a reference to our model class `Cat` like so:

```
const { sequelize, Cat } = require("./models");
```

Now let's update *our* `index.js` file to fetch a `Cat` from the `Cats` table:

```
const { sequelize , Cat } = require("./models");

async function main() {
  try {
    await sequelize.authenticate();
  } catch (e) {
    console.log("Database connection failure.");
    console.log(e);
    return;
  }

  console.log("Database connection success!");
  console.log("Sequelize is ready to use!");

  const cat = await Cat.findByPk(1);
  console.log(cat.toJSON());

  await sequelize.close();
}

main();

// This code prints:
//
// Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
// {
//   id: 1,
//   firstName: 'Markov',
//   specialSkill: 'sleeping',
//   age: 5,
//   createdAt: 2020-02-03T21:32:28.960Z,
//   updatedAt: 2020-02-03T21:32:28.960Z
// }
```

We use the `Cat.findByPk` static class method to fetch a single cat: the one with `id` equal to 1. This static method exists because our `Cat` model class *extends* `Sequelize.Model`.

"Pk" stands for *primary key*; the `id` field is the primary key for the `Cats` table. `findByPk` returns a `Promise`, so we must `await` the result. The result is an instance of the `Cat` model class.

The cleanest way to log a fetched database record is to first call the `toJSON` method. `toJSON` converts a `Cat` object to a *Plain Old JavaScript Object* (POJO). `Cat` instances have many private variables and methods that can be distracting when printed. When you call

`toJSON`, only the public data fields are copied into a JavaScript `Object`. Printing this raw JavaScript `Object` is thus much cleaner.

> The author has a pet-peeve about the `.toJSON()` method of Sequelize, it does not return JSON. It instead returns a POJO. If you needed it to be JSON you would still need to call `JSON.stringify(cat.toJSON())`. Perhaps they should have called it `.toObject` or `.toPOJO` instead.

Note that Sequelize has logged the SQL query it ran to fetch Markov's information. This logging information is often helpful when trying to figure out what Sequelize is doing.

You'll also notice that Sequelize puts double quotes around the table and field names. So if you are trying to look at your "Cats" table from the `psql` command you will need to quote them there as well. This is because PostgreSQL lowercases all identifiers like table and fields names before the query is run if they aren't quoted.

Reading And Changing Record Attributes

While `toJSON` is useful for logging a `Cat` object, it is not the simplest way to access individual column values. To read the `id`, `firstName`, etc of a `Cat`, you can directly access those attributes on the `Cat` instance itself:

```
async function main() {
  // Sequelize authentication code from above...

  const cat = await Cat.findByPk(1);
  console.log(`${cat.firstName} has been assigned id #${cat.id}.`);
  console.log(`They are ${cat.age} years old.`)
  console.log(`Their special skill is ${cat.specialSkill}.`);

  await sequelize.close();
}

main();

// This code prints:
//
// Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
// Markov has been assigned id #1.
// They are 5 years old.
// Their special skill is sleeping.
```

Accessing data directly through the `Cat` object is just like reading an attribute on any other JavaScript class. You may likewise *change* values in the database:

```
async function main() {
  // Sequelize authentication code from above...

  // Fetch existing cat from database.
  const cat = await Cat.findByPk(1);
  // Change cat's attributes.
  cat.firstName = "Curie";
  cat.specialSkill = "jumping";
  cat.age = 123;

  // Save the new name to the database.
  await cat.save();

  await sequelize.close();
}

// Prints:
//
// Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
// Executing (default): UPDATE "Cats" SET "firstName"=$1,"specialSkill"=$2,"age"=$3,"updatedAt"=$4

main();
```

Note that changing the `firstName` attribute value does not immediately change the stored value in the SQL database. Changing the `firstName` without calling `save` **has no effect** on the database. Only when we call `cat.save()` (and `await` the promise to resolve) will the changes to `firstName`, `specialSkill`, and `age` be saved to the SQL database. All these values are updated simultaneously.

Conclusion

Having completed this reading, you should be able to:

- Describe what an Object Relational Mapping is and what it is used for.
- Install the `sequelize`, `sequelize-cli`, `pg` packages.
- Configure Sequelize via the `config/config.json` file.
- Use Sequelize's `authenticate` method to verify that Sequelize can connect to the database.
- Use the Sequelize CLI `model:generate` command to generate a model file.
- Configure a model file to tell Sequelize about each database column.
- Use the `findByPk` class method to fetch data from a SQL table.
- Read data attributes from a model instance.
- Modify a model instance's attributes and save the changes back to the SQL database using the `save` method.

Using Database Migrations

We've seen how to use an ORM like Sequelize to fetch and store data in a SQL database using JavaScript classes and methods. Sequelize also lets you write JavaScript code that creates, modifies, or drops SQL tables. The JavaScript code that does this is called a *migration*. A migration "moves" the database from an old schema to a new schema.

When you finish this reading you will be able to:

- Describe advantages to using migrations over raw SQL commands to create and drop tables.
- Write migrations that create and drop tables.
- Undo incorrect migrations, fix them, and rerun them.

Sequelize Migration Files

In the prior reading we assumed that a `Cats` table already existed in our `catsdb` database. In this reading, we will presume that the `Cats` table does not exist, and that we have to create the table ourselves. This is the typical case when you aren't merely interacting with a preexisting database. When you develop your own application, the database will start out empty and with a blank schema.

We previously used the Sequelize CLI tool to autogenerate a `Cat` model file like so:

```
# Oops, forgot age:integer!
npx sequelize model:generate --name Cat --attributes "firstName:string,specialSkill:string"
```

We noted that this creates *two* files. We've already examined the model file `./models/cat.js`. We will now look at the auto-generated *migration* file `./migrations/20200203211508-create-cat.js`.

```
// ./migrations/20200203211508-create-cat.js

'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Cats', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      firstName: {
        type: Sequelize.STRING
      },
      specialSkill: {
        type: Sequelize.STRING
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Cats');
  }
};
```

The migration file exports two functions: `up` and `down`. The `up` function tells Sequelize how to create a `Cats` table. The `down` function tells Sequelize how to "undo" the `up` function. The `down` function drops the `Cats` table.

We will examine these functions more closely, but let's first see how to use a migration.

> Note: The timestamp `20200203211508` preceding `-create-cat.js` represents February 2, 2020. It gives the time and day that the migration was generated. (Your date should be when you generated your migration) By using the date and time as part of the filename, all migration files will have unique names. Also, alphabetical sorting will order the files from oldest to most recent migration.

Running A Migration

To create the `Cats` table, we must run our migration code. Having generated the `20200203211508-create-cat.js` migration file, we will use the Sequelize CLI tool to run the migration. We may do this like so:

```
# Run the migration's `up` method.
npx sequelize db:migrate
```

By giving Sequelize the `db:migrate` subcommand, it will know that we are asking it to run any new migrations. To run a migration, Sequelize will call the `up` method defined in the migration file. The `up` method will run the necessary `CREATE TABLE ...` SQL command for us. Sequelize will record (in a special `catsdb` table called `SequelizeMeta`) that the migration has been run. The next time we call `npx sequelize db:migrate`, Sequelize will not try to "redo" this already performed migration. It will do nothing the second time.

Having run the migration, we can verify that the `Cats` table looks like it should (with the exception of the `age` column):

Note that we we are using the table name in quotes here in `psql`.

```
catsdb=> \d "Cats";
                                          Table "public.Cats"
    Column    |           Type           | Collation | Nullable
--------------+--------------------------+-----------+----------
 id           | integer                  |           | not null
 firstName    | character varying(255)   |           |
 specialSkill | character varying(255)   |           |
 createdAt    | timestamp with time zone |           | not null
 updatedAt    | timestamp with time zone |           | not null
Indexes:
    "Cats_pkey" PRIMARY KEY, btree (id)
```

Rolling Back A Migration

We made a mistake when generating our `Cat` migration. We forgot to include the `age` column.

One way to fix this is to generate a *second* migration that adds the forgotten `age` column. If we have already pushed our migration code to

a remote git server, we should opt for this option.

If the migration has not yet been pushed, we can fix the migration directly. We will "undo" (AKA *rollback*) the migration that created the `Cats` table (dropping the table), fix the `up` method so that the `age` column is included, and finally rerun the migration.

> Note: this is not the same as the SQL command ROLLBACK.

To undo the migration, we run:

```
npx sequelize db:migrate:undo
```

Sequelize will call the `down` method for us, and the `Cats` table is dropped.

Why should you not use the `db:migrate:undo` way when the migration file has already been pushed to a remote git server? The reason is this: you can easily tell other developers to fix a broken migration by writing a second fixup migration (for instance, that adds the `age` column). All you need to do is check this new migration file into source control and push it. When another developer pulls your new migration code, the next time they run `npx sequelize db:migrate`, your fixup migration will be run on their local machine.

When rolling back already-checked-in migrations, there is no way to easily communicate to other developers that they should (1) rollback your migration and (2) rerun the newly corrected version of this migration. To avoid this communication problem, you should only rollback commits if you haven't already pushed them to a remote git server.

Editing A Migration File

Let's examine the `up` and `down` methods more closely. Let's start with the `up` method:

```
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Cats', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      firstName: {
        type: Sequelize.STRING
      },
      specialSkill: {
        type: Sequelize.STRING
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  // ...
};
```

The `up` method will be passed a QueryInterface (documentation) object. This object provides a number of commands for modifying the SQL database schema. The `createTable` method is amongst the most important.

We pass the table name (`'Cats'`) along with an object mapping column names to column attributes. Every column must have a specified `type`. This is similar to what we saw when we generated a model file. Note that we **do not** take `id`, `createdAt`, or `updatedAt` for granted. We need to include those columns. Luckily, everything has been auto-generated for us!

We will talk about `allowNull` and `primaryKey` in a later reading. These attributes ask Sequelize to add database constraints to a column. Likewise we will ignore `autoIncrement` for the moment (this allows a unique `id` to be auto-generated by the database for each saved row in the `Cats` table).

We fix the `up` method like so:

```
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Cats', {
      // ...
      firstName: {
        type: Sequelize.STRING
      },
      specialSkill: {
        type: Sequelize.STRING
      },
      // Here we add the `age` column.
      age: {
        type: Sequelize.INTEGER,
      },
      // ...
    });
  },
  // ...
};
```

Adding the `age` column to the migration is a lot like how we added `age` to our model file.

Having fixed our migration, we may now "rerun" it the same way we ran it the first time:

```
npx sequelize db:migrate
```

We may now behold the fixed table:

```
catsdb=> \d "Cats"
                                      Table "public.Cats"
    Column    |           Type           | Collation | Nullable
--------------+--------------------------+-----------+----------
 id           | integer                  |           | not null
 firstName    | character varying(255)   |           |
 specialSkill | character varying(255)   |           |
 age          | integer                  |           |
 createdAt    | timestamp with time zone |           | not null
 updatedAt    | timestamp with time zone |           | not null
Indexes:
    "Cats_pkey" PRIMARY KEY, btree (id)
```

`up` And `down` are Asynchronous

A final note about `up` (and also `down`). Sequelize expects `up` to be *asynchronous*. That is, Sequelize expects `up` to return a `Promise` object. Sequelize will wait for the `Promise` to be resolved. When the `Promise` is resolved, Sequelize will know the work of the `up` method is complete.

The `createTable` method is also asynchronous (returns a `Promise`). The promise resolves when `createTable` is done creating the table. This is why `up` is written as:

```
module.exports = {
  up: (queryInterface, Sequelize) => {
    // up returns Promise returned by `createTable`.
    return queryInterface.createTable('Cats', {
      // ...
    });
  },
  // ...
};
```

Sequelize is able to autogenerate a migration to create a `Cats` table, but many other migrations (for instance, to add an `age` column to our `Cats` table) must be written by hand. When writing your own migrations, you may prefer using `async` / `await`, which is clearer:

```
module.exports = {
  // Note the addition of the `async` keyword
  up: async (queryInterface, Sequelize) => {
    // await `createTable` to finish its work.
    await queryInterface.createTable('Cats', {
      // ...
    });

    // No need to return anything. An `async` method always returns a
    // Promise that waits for all `await`ed work to be performed.
  },
  // ...
};
```

Writing A `down` Method

A `down` method is written just like an `up` method. In the `down` method we "undo" what has been performed by the `up` method. We call `QueryInterface`'s `dropTable` method to drop the `Cats` table we created in `up`:

```
module.exports = {
  // ...
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Cats');
  }
};

// OR, async/await way:
module.exports = {
  // ...
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('Cats');
  }
};
```

Imagine we had forgotten to drop the `Cats` table in the `down` method. That is: imagine the `down` method was somehow left empty. If we rollback the migration nothing will be done by the empty `down` method. Thus the incorrect `Cats` table we created will not have been dropped. The wrong `Cats` table would still exist.

Imagine we next fix the migration's `up` method. We want to rerun the migration now and create the corrected `Cats` table. But when try to do this, Sequelize will hit an error! Rerunning the migration will try to `CREATE TABLE "Cats"` again, but SQL will complain because a `Cats` table already exists. It was created the first time we ran the migration, but never dropped when we tried to rollback the migration!

Inevitably all programmers will sometimes make mistakes like this. In these circumstances, you will probably have to open `psql` and write a SQL `DROP TABLE` command to fix things. Having manually corrected things, you can finally rerun the corrected migration.

You should **never** manually drop a table on a production database. That is incredibly dangerous, and typically cannot be undone. Even if database backups do not exist, recently inserted data will be lost forever. This is yet another reason why you ought not rollback migrations that have been pushed from your local development environment!

Advantages Of Migrations

Having seen how to *use* Sequelize migrations, we can discuss their benefits versus writing SQL commands like `CREATE TABLE ...` yourself.

The first advantage is that Sequelize migration code is written in JavaScript, which you may find simpler to write/read than the corresponding SQL code. Most programmers write more JavaScript than SQL, so they are typically better at remembering how to do things in JavaScript than in SQL.

A second advantage is that migration files store SQL schema change code permanently. The migration files can be checked into git, so that you don't ever forget how your database was configured.

A third (related) advantage comes when another developer wants to collaborate on your JavaScript program. By cloning your git repository, they get all the migration files, too. To setup their own copy of your database, a collaborator can run the migration files on their own computer, playing back the schema changes one-by-one. Because they apply the same migrations as you, they end up with the same schema as you.

Last, by using migrations you are able to rollback database changes to fix bugs. This can be helpful in a local development environment where it is typical to make mistakes. Remember though: you should **never** rollback migrations that have been run on a production server.

Conclusion

Having completed this reading, you now are able to:

- Describe advantages to using migrations over raw SQL commands to create, modify, and drop tables.
- Generate (and modify as needed)) migrations that create and drop tables.
- Run migrations to change the database schema.
- Undo incorrect migrations, fix them, and rerun them.

CRUD Operations Using Sequelize

There are four general ways to interact with a database. To illustrate these, recall our `Cats` table. We can:

1. Save a new cat to the database by *creating* a new row in the `Cats` table,
2. We can *read* previously stored cat data by fetching a row (or multiple rows) out of the `Cats` table,
3. We can *update* some of the column values for a pre-existing cat by modifying a row in the `Cats` table,
4. We can delete (*destroy*) the data for a cat by removing a row in the `Cats` table.

These four actions are sometimes abbreviated as *CRUD*. After this reading, you will be able to:

- Use Sequelize to create new records in a table,
- Use Sequelize to read/fetch existing records by primary key,
- Use Sequelize to update existing records with new attribute values,
- Use Sequelize to delete records from a table.

Creating A New Record

To save a new cat's data as a row in the `Cats` table, we do a two step process:

1. We call the static `build` method on the `Cat` class with the desired values.
2. We call the `save` method on the `cat` instance.

Let's see an example:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Constructs an instance of the JavaScript `Cat` class. **Does not
  // save anything to the database yet**. Attributes are passed in as a
  // POJO.
  const cat = Cat.build({
    firstName: "Markov",
    specialSkill: "sleeping",
    age: 5,
  });

  // This actually creates a new `Cats` record in the database. We must
  // wait for this asynchronous operation to succeed.
  await cat.save();

  console.log(cat.toJSON());

  await sequelize.close();
}

main();
```

Running the code:

```
Executing (default): INSERT INTO "Cats" ("id","firstName","specialSkill","age","createdAt","updatedAt") VALUES (DEF
{
  id: 1,
  firstName: 'Markov',
  specialSkill: 'sleeping',
  age: 5,
  updatedAt: 2020-02-11T19:04:23.116Z,
  createdAt: 2020-02-11T19:04:23.116Z
}
```

A new row has been inserted into the `Cats` table. We see that `id`, `updatedAt`, and `createdAt` were each autogenerated for us.

Reading A Record By Primary Key

Let's read an existing record from the database:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Fetch the cat with id #1.
  const cat = await Cat.findByPk(1);
  console.log(cat.toJSON());

  await sequelize.close();
}

main();
```

Running this code prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
{
  id: 1,
  firstName: 'Markov',
  specialSkill: 'sleeping',
  age: 5,
  createdAt: 2020-02-11T19:04:23.116Z,
  updatedAt: 2020-02-11T19:04:23.116Z
}
```

Fetching a record by primary key is the most common form of read operation from a database. In another reading we will learn other ways to fetch data. For instance: we will learn how to fetch all cats named "Markov" (there may be many).

Updating A Record

Let's tweak our reading code to change (*update*) an attribute of Markov:

```
const { sequelize, Cat }  = require("./models");

async function main() {
  const cat = await Cat.findByPk(1);

  console.log("Old Markov: ");
  console.log(cat.toJSON());

  // The Cat object is modified, but the corresponding record in the
  // database is *not* yet changed at all.
  cat.specialSkill = "super deep sleeping";
  // Only by calling `save` will the data be saved.
  await cat.save();

  console.log("New Markov: ");
  console.log(cat.toJSON());

  await sequelize.close();
}

main();
```

Running this code prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
Old Markov:

{
  id: 1,
  firstName: 'Markov',
  specialSkill: 'sleeping',
  age: 5,
  createdAt: 2020-02-11T19:04:23.116Z,
  updatedAt: 2020-02-11T19:04:23.116Z
}
Executing (default): UPDATE "Cats" SET "specialSkill"=$1,"updatedAt"=$2 WHERE "id" = $3
New Markov:
{
  id: 1,
  firstName: 'Markov',
  specialSkill: 'super deep sleeping',
  age: 5,
  createdAt: 2020-02-11T19:04:23.116Z,
  updatedAt: 2020-02-11T19:15:08.668Z
}
```

Important note: changing an attribute of a `Cat` object does not immediately change any data in the `Cats` table. To change data in the

`Cats` table, you must also call `save`. If you forget to call `save`, no data will be changed. `save` is asynchronous, so you must also `await` for it to complete.

If you look carefully, you can see that the `updatedAt` attribute was changed for us when we updated Markov!

Destroying A Record

We can also destroy records and remove them from the database:

```
const process = require("process");

const { sequelize , Cat } = require("./models");

async function main() {
  const cat = await Cat.findByPk(1);
  // Remove the Markov record.
  await cat.destroy();

  await sequelize.close();
}

main();
```

This code prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
Executing (default): DELETE FROM "Cats" WHERE "id" = 1
```

Class Methods For CRUD

When creating a record, you can avoid the two step process of (1) creating a `Cat` instance and (2) calling the `save` instance method. You can do a one step process of calling the `create` **class method**:

```
const { sequelize, Cat } = require("./models");

async function main() {
  const cat = await Cat.create({
    firstName: "Curie",
    specialSkill: "jumping",
    age: 4,
  });

  console.log(cat.toJSON());

  await sequelize.close();
}

main();
```

The `create` class method does both steps in one. It is just a convenience. Similar to before, this code prints:

```
Executing (default): INSERT INTO "Cats" ("id","firstName","specialSkill","age","createdAt","updatedAt") VALUES (DEF
{
  id: 3,
  firstName: 'Curie',
  specialSkill: 'jumping',
  age: 4,
  updatedAt: 2020-02-11T19:36:03.858Z,
  createdAt: 2020-02-11T19:36:03.858Z
}
```

When destroying, we also did a two step process: (1) fetch the record, (2) call the `destroy` instance method. Instead, we could just call the `destroy` **class method** directly:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Destroy the Cat record with id #3.
  await Cat.destroy({ where: { id: 3 } });

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): DELETE FROM "Cats" WHERE "id" = 3
```

An advantage to the class method form of destroying is that we avoid an unnecessary fetch of `Cat.findByPk(3)`. Database queries can sometimes be slow, though typically a few extra queries won't make a big difference. Choosing between the instance and class methods of destroying usually comes down to which you consider easier to read/understand.

Conclusion

As ever, the best resource for learning about Sequelize model methods is the documentation. The documentation explains the `create`, `destroy`, `findByPk`, and `save` methods in depth.

Having completed this reading, you now know how to:

- Use Sequelize to create new records in a table (using both instance and class methods),
- Use Sequelize to read/fetch existing records by primary key,
- Use Sequelize to update existing records with new attribute values,
- Use Sequelize to delete records from a table (using both instance and class methods).

Querying Using Sequelize

We have already seen how to find a single record by primary key: `findByPk`. In this reading we will learn about more advanced ways to query a table. We will learn how to:

- Fetch all `Cats` whose name is `"Markov"`,
- Fetch all `Cats` whose name is `"Markov"` **OR** `"Curie"`,
- Fetch all `Cats` whose age is **not** 5,
- Fetch all `Cats` whose name is `"Markov"` **AND** whose age is 5,
- Fetch all `Cats` whose age is **less than** 5,

We will also learn how to:

- Order `Cats` results by age (descending or ascending),
- Limit `Cats` results to a finite number.

Basic Usage Of `findAll` To Retrieve Multiple Records

Let's consider a simple example where we want to retrieve all the `Cats` in the database:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // `findAll` asks to retrieve _ALL_ THE CATS!!  An array of `Cat`
  // objects will be returned.
  const cats = await Cat.findAll();

  // Log the fetched cats.
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

Since this is an array we can't use that `.toJSON()` method we learned earlier, so we can instead use `JSON.stringify` on the Array.

Pro tip: giving a 3rd argument to `JSON.stringify` will pretty-print the result with the specified spacing. (We pass `null` as the 2nd argument to skip it.) You can read more at the `JSON.stringify` docs.

Running this code prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  },
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

It isn't typical to want to fetch *every* record. We typically want to get only those records that match some criterion. In SQL, we use a `WHERE` clause to do this. With Sequelize, we issue a `WHERE` query like so:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Fetch all cats named Markov.
  const cats = await Cat.findAll({
    where: {
      firstName: "Markov",
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

Which prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  }
]
```

We've passed the `findAll` class method the `where` option. The `where` option tells Sequelize to use a `WHERE` clause. The option value passed is `{ firstName: "Markov" }`. This tells Sequelize to only return those `Cats` where `firstName` is equal to `"Markov"`.

If we wanted to select those `Cats` named Markov **OR** Curie, we can map `firstName` to an array of `["Markov", "Curie"]`. For example:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Fetch all cats named either Markov or Curie.
  const cats = await Cat.findAll({
    where: {
      firstName: ["Markov", "Curie"],
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  },
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

The difference is that we've passed `{ firstName: ["Markov", "Curie" ]}`. Sequelize will return all `Cats` whose `firstName` matches either `"Markov"` or `"Curie"`.

Using `findAll` To Find Objects Not Matching A Criterion

We can also find all the `Cats` whose names are **NOT** Markov, but we will need to require in the `Op` object from Sequelize so we can use the "not equal" operator from it:

```
const { Op } = require("sequelize");
const { sequelize, Cat } = require("./db/models");

async function main() {
  const cats = await Cat.findAll({
    where: {
      firstName: {
        // Op.ne means the "not equal" operator.
        [Op.ne]: "Markov",
      },
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

Prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
[
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

This is our first example of a Sequelize *operator*: `Op.ne`. `ne` stands for "not equal." Instead of mapping `firstName` to a single value like `"Markov"` or an array of values like `["Markov", "Curie"]`, we have mapped it to:

```
{ [Op.ne]: "Markov" }
```

How does this work? `Op.ne` is a JavaScript *symbol*: `Op.ne === Symbol.for('ne')`. To simplify, let's just imagine that `Op.ne === "ne"`.

When we write `{ [Op.ne]: "Markov" }`, the `[]` brackets perform key interpolation. So this is equal to `{ "ne": "Markov" }`. So overall, we are effectively writing:

```
db.Cat.findAll({
  where: {
    // Won't exactly work (you need to use `[Op.ne]` after all). Does
    // illustrate the concept though.
    firstName: { "ne": "Markov" },
  },
})
```

This perhaps makes it clearer how Sequelize understands what we want. Sequelize is being passed an *object* as the `firstName` value. The object is specifying that we want to do a `!=` SQL operation by using the `"ne"` ("not equal") key. The value to "not equal" is specified as `"Markov"`.

Combining Criteria with `Op.and`

We've seen one way to do an `OR` operation above (by mapping a column name to an array of values). Let's see how to do an `AND` operation:

```
const { Op } = require("sequelize");
const { sequelize , Cat } = require("./models");

async function main() {
  // fetch cats with name != Markov AND age = 4.
  const cats = await Cat.findAll({
    where: {
      firstName: {
        [Op.ne]: "Markov",
      },
      age: 4,
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
[
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

Simply by listing more key/value pairs in the `where` object, we ask Sequelize to "AND" together multiple criteria.

Another way to do the same thing is like so:

```
const { Op } = require("sequelize");
const { sequelize , Cat } = require("./models");

async function main() {
  const cats = await db.Cat.findAll({
    where: {
      [Op.and]: [
        { firstName: { [Op.ne]: "Markov" } },
        { age: 4 },
      ],
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

The use of the `Op.and` operator is somewhat similar to `Op.ne`. This time we map `Op.and` to an *array* of criteria. Returned records must match all the criteria.

Combining Criteria with `Op.or`

We've already seen how to do an `OR` to match a *single column* against *multiple values*. You can use `Op.or` for even greater flexibility:

```
const { Op } = require("sequelize");
const { sequelize, Cat } = require("./models");

async function main() {
  // fetch cats with name == Markov OR age = 4.
  const cats = await Cat.findAll({
    where: {
      [Op.or]: [
        { firstName: "Markov" },
        { age: 4 },
      ],
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  },
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

Our query is to find all cats whose names are "Markov" and whose age is 4. Therefore both cats are returned: Markov and Curie (whose age is 4).

Querying With Comparisons

We can use operators like `Op.gt` (greater than) and `Op.lt` (less than) to select by comparing values. We use these just like `Op.ne`:

```
const { Op } = require("sequelize");
const { sequelize, Cat } = require("./models");

async function main() {
  // Fetch all cats whose age is > 4.
  const cats = await Cat.findAll({
    where: {
      age: { [Op.gt]: 4 },
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  }
]
```

Ordering Results

We've seen how to use a `where` query option to filter results with a SQL `WHERE` clause. We can use the `order` query option to perform a SQL `ORDER BY`:

```
const { sequelize, Cat } db = require("./models");

async function main() {
  const cats = await Cat.findAll({
    order: [["age", "DESC"]],
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  },
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

We've specified `{ order: [["age", "DESC"]] }`. Notice how we specify the sort order with a doubly-nested array. If we wanted to sort ascending we could more simply write: `{ order: ["age"] }`.

What if we wanted to sort by *two* columns? For instance, say we wanted to `SORT BY age DESC, firstName`. We would write: `{ order: [["age", "DESC"], "firstName"] }`. That would sort descending by `age`, and then ascending by `firstName` for cats with the same age.

Limiting Results and `findOne`

If we want only the oldest cat we can use the `limit` query option:

```
const { sequelize, Cat } = require("./models");

async function main() {
  const cats = await Cat.findAll({
    order: [["age", "DESC"]],
    limit: 1,
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This selects only one (the oldest) cat:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  }
]
```

Since we know that there will be only one result, it is pointless to return an array. In cases when we want a maximum of one result, we can use `findOne`:

```
const { sequelize, Cat } = require("./models");

async function main() {
  const cat = await Cat.findOne({
    order: [["age", "DESC"]],
  });
  console.log(JSON.stringify(cat, null, 2));

  await sequelize.close();
}

main();
```

Which prints:

```
>> node index.js
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
{
  "id": 4,
  "firstName": "Markov",
  "specialSkill": "sleeping",
  "age": 5,
  "createdAt": "2020-02-11T23:03:25.388Z",
  "updatedAt": "2020-02-11T23:03:25.388Z"
}
```

This returned the `Cat` object directly, not wrapped in an array.

If there is no record matching the criteria passed to `findOne`, it will return `null` (rather than an empty array):

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Try to find a non-existant cat.
  const cat = await Cat.findOne({
    where: {
      firstName: "Franklin Delano Catsevelt",
    },
  });
  console.log(JSON.stringify(cat, null, 2));

  await sequelize.close();
}

main();
```

No such cat exists:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt" FROM "Cats" AS "Cat"
null
```

Conclusion

We've scratched the surface of the many query options supported by Sequelize. You may find more information as necessary by reading the Sequelize querying documentation. You can in particular review the list of Sequelize query operators.

Now that you've completed this reading you should know how to:

- Use the `where` query option,
- Use the `Op.and` operator to match **all** of multiple criteria,
- Use the `Op.or` operator to match **any** of multiple criteria,
- Use the `Op.ne` to match rows where the value **does not equal** the specified value,
- Use the `Op.gt`, `Op.lt` operators to **compare** values,
- Use the `order` query option to **order** results,
- Use the `limit` query option to **limit** the number of returned results,
- Use `findOne` when only one result is expected or desired.

Model Validations With Sequelize

It's important to make sure that data stored to a database is not erroneous or incomplete. Imagine the following forms of "garbage data:"

- A `Cats` record with `firstName` set to `NULL`. All `Cats` ought to have a name.
- A `Cats` record with `firstName` set to the empty string: `""`.
- A `Cats` record with an `age` less than `0`. `age` must always be non-negative.

- Perhaps the `specialSkill` should come from a pre-defined limited list of `["jumping", "sleeping", "purring"]`. A `Cats` record with a `specialSkill` of `"pearl diving"` would thus be invalid.

Sequelize lets us write JavaScript code that will check that these data requirements are satisfied before saving a record to the database. The JavaScript code that does this is called a *validation*. A validation is code that makes sure that data is valid.

In this reading you will learn how to:

1. Validate that an attribute is not set to `NULL`.
2. Validate that a string attribute is not set to the empty string `""`.
3. Validate that a string attribute is not too long (has too many characters).
4. Validate that a numeric attribute meets minimum or maximum thresholds.
5. Validate that an attribute is within a limited set of options.

Validating That An Attribute Is Not `NULL`

We should not allow a `Cat` to be saved to the database if it lacks

1. a `firstName`,
2. an `age`, or
3. a `specialSkill`.

None of these should be set to `NULL`.

Before adding validations to check these requirements, let's review what our `Cat` model code currently looks like:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: DataTypes.STRING,
    specialSkill: DataTypes.STRING,
    age: DataTypes.INTEGER,
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

We will modify our model definition to give more specific instructions to Sequelize about the `firstName`, `specialSkill`, and `age` attributes:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "firstName must not be null",
        },
      },
    },
    specialSkill: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "specialSkill must not be null",
        },
      },
    },
    age: {
      type: DataTypes.INTEGER,
      allowNull: false,
      validate: {
        notNull: {
          msg: "age must not be null",
        },
      },
    },
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

What has changed? We now map each attribute name (`firstName` , `specialSkill` , `age`) to a POJO that tells Sequelize how to configure that attribute. Here is the POJO for `firstName` :

```
{
  type: DataTypes.STRING,
  allowNull: false,
  validate: {
    notNull: {
      msg: "firstName must not be null",
    },
  },
}
```

The `type` attribute is of course vital: this used to be the only thing we specified. We've added two new attributes. The first is `allowNull: false` . This tells Sequelize not to let us set the `firstName` attribute to `NULL` .

The second attribute is `validate` . We will spend a lot of time examining this attribute in this reading. Validation logic for `firstName` is configured inside the `validate` attribute. Our `validate` configuration is:

```
{
  notNull: {
    msg: "firstName must not be null",
  },
}
```

This configuration tells Sequelize what error message to give if we try to set the `firstName` attribute to `NULL` . It's odd that we have to set both `allowNull: false` and `notNull: { msg: ... }` . This feels like unnecessary duplication. Regardless, that's what Sequelize wants us to do. On the other hand, we do get a chance to specify the error message to print if the validation fails (`"firstName must not be null"`).

Let's see how the validation logic helps us avoid saving junk data to our database:

```
// index.js
const { sequelize, Cat } = require("./models");

async function main() {
  const cat = Cat.build({
    // Empty cat. All fields set to `null`.
  });

  try {
    // Try to save cat to the database.
    await cat.save();

    console.log("Save success!");
    console.log(JSON.stringify(cat, null, 2));
  } catch (err) {
    console.log("Save failed!");

    // Print list of errors.
    for (const validationError of err.errors) {
      console.log("*", validationError.message);
    }
  }

  await sequelize.close();
}

main()
```

Running this code prints:

```
Save failed!
* firstName must not be null
* specialSkill must not be null
* age must not be null
```

What happened? When we call the `save` method on a `Cat`, Sequelize will check that all the specified validations are satisfied. In this case none of them are! The `save` method will throw an exception, which we handle using `try { ... } catch (err) { ... }`.

What kind of exception? The thrown error is a `ValidationError`. This has an `errors` attribute, which stores an array of `ValidationErrorItem`s. We print out the message for each item error.

Because there were validation failures, Sequelize **will not save** the invalid `Cats` record to the database. Sequelize thus keeps us from inserting junk data into the database.

If we want to save our `Cat` object, we would have to change its attributes to meet the validations (i.e., set them to something other than `NULL`) and call `save` a second time. For example:

```
// index.js
const { sequelize, Cat } = require("./models");

async function main() {
  const cat = Cat.build({
    // Empty cat. All fields set to `null`.
  });

  try {
    await cat.save();
  } catch (err) {
    // The save will not succeed!
    console.log("We will fix and try again!");
  }

  // Fix the various validation problems.
  cat.firstName = "Markov";
  cat.specialSkill = "sleeping";
  cat.age = 4;

  try {
    // Trying to save a second time!
    await cat.save();

    console.log("Success!");
  } catch (err) {
    // The save *should* succeed!
    console.log(err);
  }

  await sequelize.close();
}

main()
```

The `notEmpty` Validation

Even though we are not allowed to set `firstName` and `specialSkill` to `NULL`, we could still set them to the empty string `""`:

```
// index.js
const { sequelize, Cat } = require("./models");

async function main() {
  const cat = Cat.build({
    firstName: "",
    specialSkill: "",
    age: 5,
  });

  try {
    // Try to save cat to the database.
    await cat.save();

    console.log("Save success!");
    console.log(JSON.stringify(cat, null, 2));
  } catch (err) {
    console.log("Save failed!");

    // Print list of errors.
    for (const validationError of err.errors) {
      console.log("*", validationError.message);
    }
  }

  await sequelize.close();
}

main();
```

```
Executing (default): INSERT INTO "Cats" ("id","firstName","specialSkill","age","createdAt","updatedAt") VALUES (DEF
Save success!
{
  "id": 8,
  "firstName": "",
  "specialSkill": "",
  "age": 5,
  "updatedAt": "2020-02-12T21:34:49.250Z",
  "createdAt": "2020-02-12T21:34:49.250Z"
}
```

This is bogus: `Cats` records should have a non-empty `firstName` and `specialSkill`. We will therefore add a second validation for both `firstName` and `specialSkill`:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "firstName must not be null",
        },
        notEmpty: {
          msg: "firstName must not be empty",
        },
      },
    },
    specialSkill: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "specialSkill must not be null",
        },
        notEmpty: {
          msg: "specialSkill must not be empty",
        },
      },
    },
    // ...
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

When we run the same `index.js` that tries to save the `Cats` record with the empty `firstName` and `specialSkill`, we now print:

```
Save failed!
* firstName must not be empty
* specialSkill must not be empty
```

Excellent! We've added the new validation by adding a `notEmpty` key to the `validate` POJO. Just like with `notNull`, we specify a message to print.

This is the typical story: we add new validations by adding new key/value pairs to the `validate` POJO. Sequelize provides many different kinds of validations for us, but we configure all of them in the same general manner.

Forbidding Long String Values

We don't want our cats to have names that are too long. We add a `len` validation like so:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "firstName must not be null",
        },
        notEmpty: {
          msg: "firstName must not be empty",
        },
        len: {
          args: [0, 8],
          msg: "firstName must not be more than eight letters long",
        },
      },
    },
    // ...
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

If we try to run:

```
// index.js
const { sequelize, Cat } = require("./models");

async function main() {
  const cat = Cat.build({
    firstName: "Markov The Magnificent",
    specialSkill: "sleeping",
    age: 5,
  });

  try {
    // Try to save cat to the database.
    await cat.save();

    console.log("Save success!");
    console.log(JSON.stringify(cat, null, 2));
  } catch (err) {
    console.log("Save failed!");

    // Print list of errors.
    for (const validationError of err.errors) {
      console.log("*", validationError.message);
    }
  }

  await sequelize.close();
}

main();
```

We will be told:

```
Save failed!
* firstName must not be more than eight letters long
```

The `len` validation gets a `msg` attribute as usual. We also configure `args: [0, 8]`. These are the "arguments" to the `len` validation. We are telling Sequelize to trigger a validation error if the `firstName` property has a length less than zero (impossible) or greater than eight.

Note that even though the `len` validation is not triggered for a length of zero, the `notEmpty` validation still will be.

If desired, we could use the `len` validation to set a true minimum length for a string. If we wanted a minimum length of two letters, we

would just change `args: [2, 8]`. (We ought also update the `msg` appropriately.)

Validating That A Numeric Value Is Within A Specified Range

A `Cat` should never have a negative age. Perhaps, also, a `Cat` should have a theoretical maximum age of 99 years. We can add validations to enforce these requirements:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    // ...
    age: {
      type: DataTypes.INTEGER,
      allowNull: false,
      validate: {
        notNull: {
          msg: "age must not be null",
        },
        min: {
          args: [0],
          msg: "age must not be less than zero",
        },
        max: {
          args: [99],
          msg: "age must not be greater than 99",
        },
      },
    },
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

You can see that the `min` and `max` validations are configured in the same sort of way that the `len` validation is.

If we try to save a `Cat` with an `age` of `-1`, we are printed:

```
Save failed!
* age must not be less than zero
```

Likewise, if we try to save a `Cat` with an `age` of `123` we are printed:

```
Save failed!
* age must not be greater than 99
```

(Note: I've stopped repeating our `index.js` file, since there are only trivial modifications to a `Cat`'s attributes each time.)

Validating That An Attribute Is Among A Finite Set Of Values

Let's say that a `Cat`'s `specialSkill` should be restricted to a pre-defined list of `["jumping", "sleeping", "purring"]`. That is: a `Cat` should not be allowed to have just any `specialSkill`. The `specialSkill` must be on the list.

We can enforce this requirement like so:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    // ...
    specialSkill: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "specialSkill must not be null",
        },
        notEmpty: {
          msg: "specialSkill must not be empty",
        },
        isIn: {
          args: [["jumping", "sleeping", "purring"]],
          msg: "specialSkill must be either jumping, sleeping, or purring",
        },
      },
    },
    // ...
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

Notice how we **doubly-nest** the list of special skills (`["jumping, "sleeping", "purring"]`) when specifying the `args` for the `isIn` validation. This is because we want to pass **one** argument: an array of three possible special skills.

Now when we try to save a `Cat` with `specialSkill` set to `"pearl diving"`, our code will print:

```
Save failed!
* specialSkill must be either jumping, sleeping, or purring
```

Conclusion

There is a very large variety of validations that are provided by Sequelize. You can find many more in the Sequelize documentation for validations.

Having completed this reading, you now know how to:

1. Validate that an attribute is not set to `NULL`.
2. Validate that a string attribute is not set to the empty string `""`.
3. Validate that a string attribute is not too long (has too many characters).
4. Validate that a numeric attribute meets minimum or maximum thresholds.
5. Validate that an attribute is within a limited set of options.

Recipe Box With Sequelize Project

In this project, you will build the Data Access Layer to power a Web application. Unlike previously, you will use the Sequelize library and tools to do this to build a more maintainable application.

It has more steps than the SQL version, but it's more maintainable in the long run. Also, the SQL version hid a lot of complexity from you with respect to the running of the SQL. Go look at the SQL version of the files in the **controllers** directory to see what we had to do to load the SQL and execute it.

Now, compare the *simplicity* of those with the simplicity of the files in the **controllers** directory for *this* version of the application. It's easier to understand *this* version. You want to know where to add a column to a table? Go to the migrations. You want to know where to fix a query? Go to the proper repository file.

It's just *so* much better organized.

Quite often, you will see that you will have more files and, overall, more lines of code in well-organized, highly-maintainable software project. Remembering where code is *is hard*. That's why having clearly-named files and directories is so very important.

The data model analysis

This looks no different because it's the same application.

What goes into a recipe box? Why, recipes, of course! Here's an example recipe card.

You can see that a recipe is made up of three basic parts:

- A title,
- A list of ingredients, and
- A list of instructions.

You're going to add a little more to that, too. It will also have

- The date/time that it was entered into the recipe box, and
- The date/time it was last updated in the recipe box.

These are good pieces of data to have so that you can show them "most recent" for example.

Ingredients themselves are complex data types and need their own structure. They "belong" to a recipe. That means they'll need to reference that recipe. That means an ingredient is made up of:

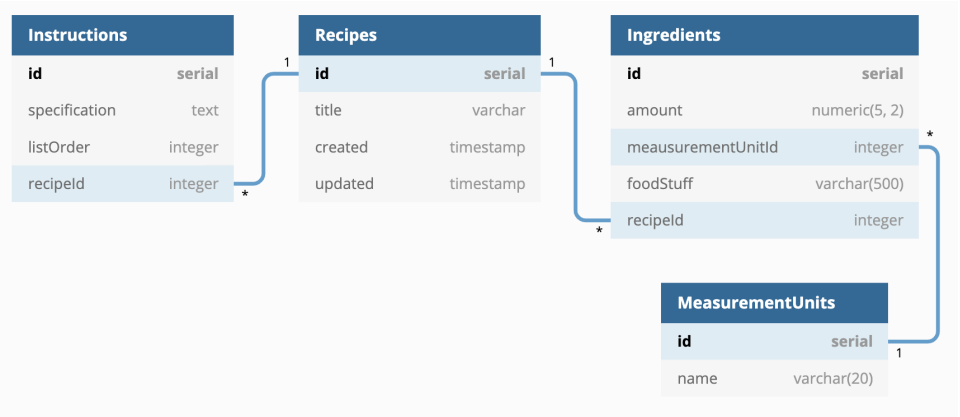
- An amount (optional),
- A unit of measure (optional),
- The actual food stuff, and
- The id of the recipe that it belongs to.

That unit of measure is a good candidate for normalization, don't you think? It's a predefined list of options that should not change and that you don't want people just typing whatever they want in there, not if you want to maintain data integrity. Otherwise, you'll end up with "C", "c", "cup", "CUP", "Cup", and all the other permutations, each of which is a distinct value but means the same thing.

Instructions are also complex objects, but not by looking at them. Initially, one might only see text that comprises an instruction. But, very importantly, instructions have *order*. They also *belong* to the recipe. With that in mind, an instruction is made up of:

- The text of the instruction,
- The order that it appears in the recipe, and
- The id of the recipe that it belongs to.

That is enough to make a good model for the recipe box.



The application

The application is a standard express.js application using the pug library to generate the HTML and the node-postgres library to connect to the database.

It already has sequelize and sequelize-cli installed.

Getting started

- Clone the starter project from https://github.com/appacademy-starters/sql-orm-recipe-box
- Run `npm install` to install the packages
- Run `npm run dev` to start the server on port 3000

You'll do all of your work in the **data-access-layer** directory. In there, you will find a series of JS files. Each of these will hold your JavaScript code rather than SQL code.

Your code

You're going to be using JavaScript and the tools of Sequelize. Keep the Sequelize documentation open and handy. Even developers that use ORMs every day will keep the documentation open because there's so much to know about them.

Phase 1: Initialize the Sequelize project

Because this project already has sequelize-cli installed, you can initialize the project by typing `npx sequelize-cli init`. The `npx` command runs locally-installed tools. That will create the project structure that Sequelize expects for us to continue to use its tools.

Phase 2: Create a database user for the project

Using a PostgreSQL client like `psql` or Postbird, create a new user for this application named "sequelize_recipe_box_app" with the password "HfKfK79k" *and* the ability to create a database. Here's the link to the CREATE USER documentation so that you can determine which options to give.

Phase 2: Change the connection configuration

The project contains a directory named **config**. Inside there, you will find a file named **config.json**. You need to make some configuration changes.

- Change all the "user" and "password" values to the information for the user that you created in Phase 2.
- Change the "database" values to be "recipe_box_development", "recipe_box_test", and "recipe_box_production".
- Change all of the "dialect" values from "mysql" to "postgres".
- Delete all of the "operatorAliases" entries. It's to support earlier versions of the Sequelize library. Make sure to remove the comma from the preceding line so that it's valid JSON.
- Because you'll be using seed data in this project, add `"seederStorage": "sequelize"` to each of the different blocks so that Sequelize CLI won't run a seeder more than once causing duplicate entries in the database.

That will configure the application and the Sequelize tools to properly connect to your development database.

Phase 3: Create your database

Rather than writing SQL to do this, you will use the tools. Run

```
npx sequelize-cli db:create
```

That runs the Sequelize CLI with the command `db:create`.

When you run this, it will default to the "development" setting and read the information from the configuration file to create your database for you! It should print out something like

```
Sequelize CLI [Node: 10.19.0, CLI: 5.5.1, ORM: 5.21.5]

Loaded configuration file "config/config.json".
Using environment "development".
Database recipe_box_development created.
```

You can also drop the database by typing ... you guessed it! The Sequelize CLI with the command `db:drop`!

```
npx sequelize-cli db:drop
```

If you run that, run the "create" command, again, so the database exists.

Phase 4: The units of measurement data

Just as a review, here is the specification for the table that holds units of measurement.

Column Name	Column Type	Constraints
id	SERIAL	PK
name	VARCHAR(20)	NOT NULL

Luckily, the Sequelize models and migrations take care of the "id" property for you without you having to do anything. So, you can just focus on that "name" property.

Create a migration

It's time to create the first migration, the one that defines the table that will hold units of measure. You can use the Sequelize CLI to generate the migration for you. You can *also* tell it to create a model for you, and it will create a migration along *with* the model. You should do that to get the biggest return on investment for the characters that you will type.

The command is `model:generate` and it takes a couple of arguments, "--name" which contains the name of the model (as a singular noun) to generate, and "--attributes" which has a comma-separated list of "property-name:data-type" pairs.

Learning Tip: It is *so very important* that you don't copy and paste this. Type these things out so it has a better chance of creating durable knowledge.

```
npx sequelize-cli model:generate \
  --name MeasurementUnit \
  --attributes name:string
```

That will create two files, if everything works well. (The name of your migration file will be different because it's time-based.)

```
New model was created at models/measurementunit.js
New migration was created at migrations/20200101012349-MeasurementUnit.js
```

The **model** file will be used by the application to query the database. It will be used by the express.js application. It is part of the running software.

The **migration** file is used to construct the database. It is only used by the Sequelize CLI tool to build the database. Unlike those schema and seed files that you had in the SQL version of this project which destroyed *everything* when run, migrations are designed to change your database as your application grows. This is a much better strategy so that existing data in the databases that other people use aren't damaged.

Because the data model requires the "name" column to be both non-null *and* unique, you have to add some information to the migration file. Open it and, for the "name" property, make non-nullable by looking at how the other properties are configured. Then, add the "unique" property set to `true` to the "name" configuration, as well. That should be enough for Sequelize to create the table for you.

The last thing to do is to change the length of the "name" property. By default, Sequelize will make it 255 characters long. The specification for the table says it should really only be 20 characters. To tell the migration that, change the type for "name" from `Sequelize.STRING` to `Sequelize.STRING(20)`.

Run your migration

If you now run your migration with the Sequelize CLI, it will create the table for you.

```
npx sequelize-cli db:migrate
```

That should give you some output that looks similar to this.

```
Loaded configuration file "config/config.json".
Using environment "development".
== 20200101012349-create-measurement-unit: migrating =======
== 20200101012349-create-measurement-unit: migrated (0.021s)
```

You can confirm that the table "MeasurementUnits" is created by using your PostgreSQL client. You'll also see that another table is created, "SequelizeMeta", which contains information about which migration has most recently been run. It contains a single column, "name". Each row contains an entry of which migration file has run. Now that you've run your migration file, the table contains one entry, the name of your migration file. When you run more migrations, you will see more rows, each containing the name of the file that you've run.

psql Note: If you are using `psql` as you PostgreSQL command, be aware that it will lowercase any entity and column names you type in there. If you type `SELECT * FROM MeasurementUnits`, it converts that to `SELECT * FROM measurementunits` before running it. To prevent that from happening, use quotation marks around the table name. `SELECT * FROM "MeasurementUnits"` will do the trick.

It's important that you *never* change the name of a migration file after it's been run.

In the real world, you should *never* change the content of a migration file after it's been committed and shared in your Git repository. Asking others to rollback their migrations just because you changed one of yours is bad manners. Instead, you should add a new migration that makes the change that you want.

Create the seed data

You can create the seed data for the unit of measurements by creating a **seeder** as the Sequelize CLI calls them. You can create one using the Sequelize CLI tool. Run the following and make sure you don't get any errors.

```
npx sequelize-cli seed:generate --name default-measurement-units
```

Now, you want to insert the seed data. You will do this by using the `bulkInsert` method of the object passed in through the `queryInterface` parameter of the `up` method. Feel free to delete the comment in the `up` method and replace it with this.

```
return queryInterface.bulkInsert('MeasurementUnits', [
  { name: 'cups', createdAt: new Date(), updatedAt: new Date() },
]);
```

The `bulkInsert` method takes two parameters:

- The name of the table to insert into, and
- An array of objects that have property names that match the column names in the table.

You can see that the first object has been provided by the example. Now, create objects for all of these values, as well. (The empty item in the list is an empty string and is intentional) Make sure you do them **in this order,** or when we get to the seed data for the other tables it won't work. (We've supplied you with files for the seed data for the other tables because there is a lot of it)

- "fluid ounces"
- "gallons"
- "grams"
- "liters"
- "milliliters"
- "ounces"
- "pinch"
- "pints"
- "pounds"
- "quarts"
- "tablespoons"
- "teaspoons"
- ""
- "cans"
- "slices"
- "splash"

Now, run the Sequelize CLI with the command `db:seed:all`.

After you get that done, you can confirm that all of the records (rows) were created in the "MeasurementUnits" table.

Phase 5: The recipe table model

This will go much like the last one, except there's no seed data. Just to refresh your memory, here's the specification for the "recipes" table.

Column Name	Column Type	Constraints
id	SERIAL	PK
title	VARCHAR(200)	NOT NULL
created	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP
updated	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP

As you've discovered, Sequelize takes care of the "id" for you *and* the columns to track when the recipe has been created and updated! Your job is to

- Generate a model for the "recipe"
- Customize the migration so the "title" column is not nullable

Run your migration and confirm that you defined it correctly by checking the attributes in the description of the table. The important parts to check are that the "title" column is a VARCHAR(200) and is non-nullable. (The "Collation" column has been removed for brevity.)

```
                Table "public.Recipes"
  Column   |           Type           | Nullable |  Default
-----------+--------------------------+----------+------------
 id        | integer                  | not null | nextval(...
 title     | character varying(200)   | not null |
 createdAt | timestamp with time zone | not null |
 updatedAt | timestamp with time zone | not null |
Indexes:
    "Recipes_pkey" PRIMARY KEY, btree (id)
```

Phase 6: The instruction table model

Now, things get a little trickier because this model will reference the recipe model. Here's the specification for the "instructions" table.

Column Name	Column Type	Constraints
id	SERIAL	PK
specification	TEXT	NOT NULL
listOrder	INTEGER	NOT NULL
recipeId	INTEGER	FK, NOT NULL

When you type out your migration generation command, the "--attributes" parameter will look like this:

```
--attributes column1:type1,column2:type2,column3:type3
```

Instead of using "string" for the "specification" column of the table, use "text" to generate a TEXT column.

After it generates the migration file, modify each of the column descriptors in the migration so that the columns are not nullable. Then, add a new property to the one for "recipeId" called "references" that is an object that contains a "model" property set to "Recipes". It should look like this.

```
recipeId: {
  allowNull: false,
  references: { model: "Recipes" },
  type: Sequelize.INTEGER,
},
```

With that in place, run the migration. Then, check the table definition in your PostgreSQL client.

```
                  Table "public.Instructions"
    Column     |           Type           | Nullable |     Default
---------------+--------------------------+----------+-----------------
 id            | integer                  | not null | nextval('"Ins...
 specification | text                     | not null |
 listOrder     | integer                  | not null |
 recipeId      | integer                  | not null |
 createdAt     | timestamp with time zone | not null |
 updatedAt     | timestamp with time zone | not null |
Indexes:
    "Instructions_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "Instructions_recipeId_fkey" FOREIGN KEY ("recipeId")
                                 REFERENCES "Recipes"(id)
```

You should see all non-null columns and a foreign key between the "Instructions" table and the "Recipes" table.

Phase 7: The ingredients model

The model for ingredients has *two* foreign keys. Create the model and migration for it. Here's the table specification.

Column Name	Column Type	Constraints
id	SERIAL	PK
amount	NUMERIC(5, 2)	NOT NULL
measurementUnitId	INTEGER	FK, NOT NULL

Column Name	Column Type	Constraints
foodStuff	VARCHAR(500)	NOT NULL
recipeId	INTEGER	FK, NOT NULL

After you modify and run your migration, you should have a table in your database that looks like this, with two foreign keys, one to the "Recipes" table and the other to the "MeasurementUnits" table.

```
                    Table "public.Ingredients"
      Column       |           Type           | Nullable |      Default
-------------------+--------------------------+----------+-------------------
 id                | integer                  | not null | nextval('"Ing...
 amount            | numeric(5,2)             | not null |
 measurementUnitId | integer                  | not null |
 foodStuff         | character varying(500)   | not null |
 recipeId          | integer                  | not null |
 createdAt         | timestamp with time zone | not null |
 updatedAt         | timestamp with time zone | not null |
Indexes:
    "Ingredients_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "Ingredients_measurementUnitId_fkey"
        FOREIGN KEY ("measurementUnitId")
        REFERENCES "MeasurementUnits"(id)
    "Ingredients_recipeId_fkey"
        FOREIGN KEY ("recipeId")
        REFERENCES "Recipes"(id)
```

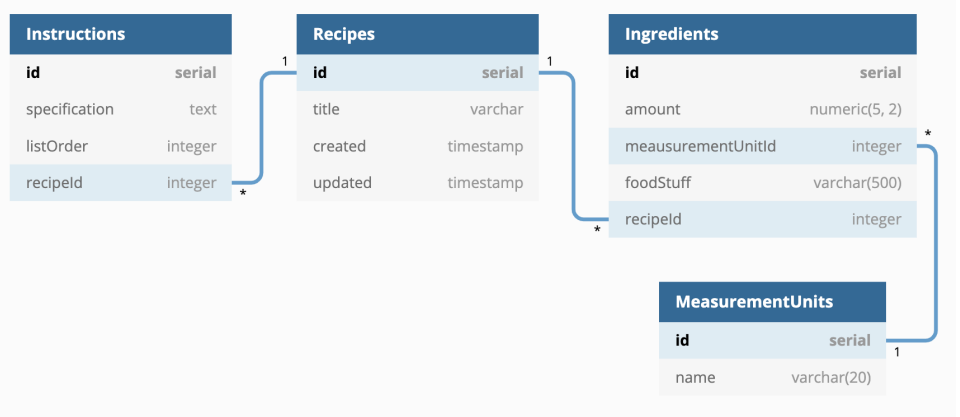
Phase 8: Seed data for all of the tables

Now that you have tables in the database, it's time to create some seed data for all of them. In the **data-access-layer** directory, you will find three text files each containing JavaScript objects on each row that match the tables in the previous three sections.

If you didn't seed the MeasurementUnits data in the correct order listed in the section above, you may have to redo that seed file, because the data from the text files depends on the ids of the data in the `MeasurementUnits` table being correct.

There are three tables to seed: Ingredients, Instructions, and Recipes. It is important to note that you will need to seed them in the correct order due to foreign key dependencies.

Look at the data model for the application, again.



You can see that the Instructions depends on Recipes because it has the foreign key "recipeId" to the Recipes table. You can also see that the Ingredients table has dependencies on the Recipes and MeasurementUnits tables because of its foreign keys "measurementUnitId" and "recipeId". (You've already seeded the MeasurementUnits table in Phase 4, so that data exists for use by the Ingredients table.) Recipes does not have any foreign keys. You need to seed Recipes, first, because it does not have any foreign keys and, therefore, does not have any data dependencies. Then, you can seed the Instructions and Ingredients tables in either order because their data dependencies will have been met.

Create seeder files for them in that order: Recipes, first, then Ingredients and Instructions. Use the contents of each of the text files in **data-access-layer** to do bulk inserts.

After you create each seed file, run

```
npx sequelize-cli db:seed:all
```

to make sure you don't have any errors. If you do, fix them before moving onto the next seed file.

If you end up seeding the data in the wrong order and getting a foreign key constraint error, just use the CLI to drop the database, create the database, migrate the database, and then you can try running your seeders, again. You may need to rename your migration filenames to get your seeds running in the correct order.

Phase 9: Updating models with references

Now that you have all of the migrations set up correctly and a database defined, it is time for you to turn your attention to the model files that were generated in the previous phases.

Consider the relationship between an Instruction and a Recipe. A Recipe *has many* Instructions. In the other direction, you would say that an Instruction *has one* Recipe, or that Instruction *belongs to* the Recipe. To set that up in your model, open the file **models/recipe.js**. In there, you will see the following.

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Recipe = sequelize.define('Recipe', {
    title: DataTypes.STRING
  }, {});
  Recipe.associate = function(models) {
    // associations can be defined here
  };
  return Recipe;
};
```

In the `associate` function is where you can define the association between the Recipe and the Instruction. Replace the comment with the following statement.

```
Recipe.hasMany(models.Instruction, { foreignKey: 'recipeId' });
```

This instructs Sequelize that Recipe should have a collection of Instruction objects associated with it. To insure that Sequelize uses the foreign key column that you created on the "Instructions" table in your migration, you must specify it as part of the collection definition.

In the file **models/instruction.js**, replace the comment with the following to define the other side of the relationship.

```
Instruction.belongsTo(models.Recipe, { foreignKey: 'recipeId' });
```

This instructs Sequelize that Instruction has a single Recipe object associated with it. Again, because of inconsistent naming conventions used by Sequelize, you must specify the foreign key column name in the "Instructions" table.

Think about the many-to-one and one-to-many relationships between Ingredient, MeasurementUnit, and Recipe. Then, modify those model files accordingly with the `hasMany` and `belongsTo` associations, always specifying the name of the foreign key column that binds the two tables together.

Phase 10: Updating models with validations

Now that you have seed data created, it will be important to prevent users from entering data that does not meet the expectations of the data model.

Consider the content of **models/instruction.js**

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Instruction = sequelize.define('Instruction', {
    specification: DataTypes.TEXT,
    listOrder: DataTypes.INTEGER,
    recipeId: DataTypes.INTEGER
  }, {});
  Instruction.associate = function(models) {
    Instruction.belongsTo(models.Recipe, { foreignKey: 'recipeId' });
  };
  return Instruction;
};
```

It would be nice if the model could validate each of those properties to make sure that no one sets them to null and that `listOrder` is greater than 0, for example. You can do that with per-attribute validations.

For example, you can change the above code to the following to make sure that the "specification" property won't get set to an empty string when someone tries to save the object.

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Instruction = sequelize.define('Instruction', {
    specification: {
      type: DataTypes.TEXT,
      validate: {
        notEmpty: true,
      },
    },
    listOrder: DataTypes.INTEGER,
    recipeId: DataTypes.INTEGER
  }, {});
  Instruction.associate = function(models) {
    Instruction.belongsTo(models.Recipe, { foreignKey: 'recipeId' });
  };
  return Instruction;
};
```

Make sure all of the other string properties in the models won't allow the empty string to be set on them.

Phase 11: Cascade delete for recipes

The Recipe model has dependencies: the Instruction and the Ingredient both have *belongs to* relationships. This means that the row in the "Recipes" table must exist to have records in the "Ingredients" and "Instructions" table. If you try to delete a Recipe row from the database that has either Instructions or Ingredients, it won't work due to referential integrity. You would have to delete all of the Ingredients and Instructions *before* being able to delete the Recipe.

Sequelize provides a handy shortcut for that and will manage deleting the associated records for you when you delete a row from the Recipes table. It's called a *cascading delete*. Open the **models/recipe.js** file. In there, modify the second argument of each of the `hasMany` calls to include two new property/value pairs:

- `onDelete: 'CASCADE'`
- `hooks: true`

Refer to the documentation on Associations to see an example. But, don't delete the `foreignKey` property that you put there in Phase 9.

Phase 12: Building the repositories

Now that you have the seeds, models, and migrations out of the way, you can build the data access layer with a lot of speed. Sequelize will now handle all of the SQL generation for you. You can just use the models that you've painstakingly crafted.

Because you are writing JavaScript files, you want the server to restart because it won't automatically reload the changed JavaScript that you're writing. To that end, you will use a different command while developing.

```
npm run dev
```

This runs a special script that will reload the JavaScript in the data access layer every time you make a change. You can see what's run in the **package.json** file in this project in the "scripts" section for the "dev" property.

You will work in the three files named

- **recipes-repository.js**: The collection of functions needed to interact with recipes for the application
- **instructions-repository.js**: The collection of functions needed to interact with the instructions for the application
- **ingredients-repository.js**: The collection of functions needed to interact with the ingredients for the application

Each of the files imports your models and makes them available to you. Then, you can use them in your querying. Follow the hints in each of the repository functions.