```
create user recipe_box_app with createdb password 'password';
```

Press enter and you should see:

```
CREATE ROLE
```

```
alias psql='psql -h localhost'
alias sc='npx sequelize-cli'
alias sc-init='npx sequelize-cli init'
alias sc-makedb='npx sequelize-cli db:create'
alias sc-makemodel='npx sequelize-cli model:generate'
alias sc-migrate='npx sequelize-cli db:migrate'
alias sc-genseed='npx sequelize-cli seed:generate'
alias sc-seed='npm sequelize-cli db:seed:all'
```

Sequelize provides utilities for generating migrations, models, and seed files

Init Project

```
$ npx sequelize-cli init
```

```
npx sequelize-cli init
```

ome directories and a file should have been generated

1. config directory
   - config.json file
2. migrations directory
3. models directory
   - index.js file
4. seeders directory

You must create a database user, and update the `config/config.json` file to match your database settings to complete the initialization process.

Models and Migrations

Make sure that you start by generating the table that does not depend on anything else, meaning it does not contain any foreign keys.

NOTE: As you continue creating tables, make sure you generate models and migrations for tables that only contain foreign keys to models that have already been generated.

We will generate all the models and migrations before moving actually migrating, because with the timestamps on the migration files, they will be created in the right order as long as we do this right.

Our Recipes table does not depend on anything, it contains no foreign keys, so we will make it first [2]

Create Database

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

Generate a model and its migration

```
npx sequelize-cli model:generate --name <ModelName> --attributes <column1>:<type>,<column2>:<type>,...
```

Run pending migrations

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

Run all pending seeds

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

Rollback one migration

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

Rollback one seed

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

Rollback all migrations

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

Rollback all seeds

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

Generate a new seed file

```
npx sequelize-cli seed:generate --name <descriptiveName>
```

1. Change all of the usernames to the username from "root" to the username that you created before: `"username": "recipe_box_app"`
2. Change the password line to provide the password for the user you created for this app: `"password": "password"`
3. Change the database line for all three environments (development, test, and production) to `"database": "recipe_box_development"`, `"database": "recipe_box_test"`, and `"database": "recipe_box_production"`.
4. Change the dialect line to `"dialect": "postgres"`
5. Remove the `"operatorsAliases"` line
6. Add the line: `"seederStorage": "sequelize"`

```
{
  "development": {
    "username": "root",
    "password": null,
    "database": "database_development",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false
  },
  "test": {
    "username": "root",
    "password": null,
    "database": "database_test",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false
  },
  "production": {
    "username": "root",
    "password": null,
    "database": "database_production",
    "host": "127.0.0.1",
    "dialect": "mysql",
    "operatorsAliases": false
  }
}
```

```
{
  "development": {
    "username": "recipe_box_app",
    "password": "password",
    "database": "recipe_box_development",
    "host": "127.0.0.1",
    "dialect": "postgres",
    "seederStorage": "sequelize"
  },
  "test": {
    "username": "recipe_box_app",
    "password": "password",
    "database": "recipe_box_test",
    "host": "127.0.0.1",
    "dialect": "postgres",
    "seederStorage": "sequelize"
  },
  "production": {
    "username": "recipe_box_app",
    "password": "password",
    "database": "recipe_box_production",
    "host": "127.0.0.1",
    "dialect": "postgres",
    "seederStorage": "sequelize"
  }
}
```

Migrations

Create Table (usually used in the up() method)

```
// This uses the short form for references
return queryInterface.createTable(<TableName>, {
    <columnName>: {
        type: Sequelize.<type>,
        allowNull: <true|false>,
        unique: <true|false>,
        references: { model: <TableName> }, // This is the plural table name
                                            // that the column references.
    }
});
// This the longer form for references that is less confusing
return queryInterface.createTable(<TableName>, {
    <columnName>: {
        type: Sequelize.<type>,
        allowNull: <true|false>,
        unique: <true|false>,
        references: {
            model: {
                tableName: <TableName> // This is the plural table name
            }
        }
    }
});
```

Delete Table (usually used in the down() function)

```
return queryInterface.dropTable(<TableName>);
```

Adding a column

```
return queryInteface.addColumn(<TableName>, <columnName>: {
    type: Sequelize.<type>,
    allowNull: <true|false>,
    unique: <true|false>,
    references: { model: <TableName> }, // This is the plural table name
                                        // that the column references.
});
```

Removing a column

```
return queryInterface.removeColumn(<TableName>, <columnName>);
```

One to One between Student and Scholarship

student.js

```
Student.hasOne(models.Scholarship, { foreignKey: 'studentId' });
```

scholarship.js

```
Scholarship.belongsTo(models.Student, { foreignKey: 'studentId' });
```

One to Many between Student and Class

student.js

```
Student.belongsTo(models.Class, { foreignKey: 'classId' });
```

class.js

```
Class.hasMany(models.Student, { foreignKey: 'classId' });
```

Many to Many between Student and Lesson through StudentLessons table

student.js

```
const columnMapping = {
    through: 'StudentLesson', // This is the model name referencing the join table.
    otherKey: 'lessonId',
    foreignKey: 'studentId'
}
Student.belongsToMany(models.Lesson, columnMapping);
```

lesson.js

```
const columnMapping = {
    through: 'StudentLesson', // This is the model name referencing the join table.
    otherKey: 'studentId',
    foreignKey: 'lessonId'
}
Lesson.belongsToMany(models.Student, columnMapping);
```

Deleting a single item

```
// Find the pet with id = 1
const pet = await Pet.findByPk(1);
// Notice this is an instance method
pet.destroy();
```

Deleting multiple items

```
// Notice this is a static class method
await Pet.destroy({
    where: {
        petTypeId: 1 // Destorys all the pets where the petType is 1
    }
});
```

Updating an item

```
// Find the pet with id = 1
const pet = await Pet.findByPk(1);
// Way 1
pet.name = "Fido, Sr."
await pet.save;
// Way 2
await pet.update({
    name: "Fido, Sr."
});
```

NOTE: *You **do not** want to create the Ingredients or Instructions table before the Recipes and MeasurementUnits tables*

For each model you generate, a new file will appear in your models and migrations folders.

To generate the Recipes model:[3]

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

To generate the MeasurementUnits model:[4]

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

After creating the tables with no dependencies, we can create the tables that depend on them

To generate the Instructions model:[6]

```
npx sequelize-cli model:generate \
  --name Instruction \
  --attributes specification:text,listOrder:integer,recipeId:integer
```

To generate the Ingredients model:

```
npx sequelize-cli model:generate \
  --name Ingredients \
  --attributes amount:numeric,measurementUnitId:integer,foodStuff:string,recipeId:integer
```

findAll

```
await <Model>.findAll({
    where: {
        <column>: {
            [Op.<operator>]: <value>
        }
    },
    include: <include_specifier>,
    offset: 10,
    limit: 2
});
```

findByPk

```
await <Model>.findByPk(<primary_key>, {
    include: <include_specifier>
});
```

Eager loading associations with `include`

Simple include of one related model.

```
    await Pet.findByPk(1,  {
        include: PetType
    })
```

Include can take an array of models if you need to include more than one.

```
    await Pet.findByPk(1, {
        include: [Pet, Owner]
    })
```

Include can also take an object with keys `model` and `include`.
This is in case you have nested associations.
In this case Owner doesn't have an association with PetType, but Pet does, so we want to include PetType onto the Pet Model.

```
    await Owner.findByPk(1, {
        include: {
            model: Pet
            include: PetType
        }
    });
```

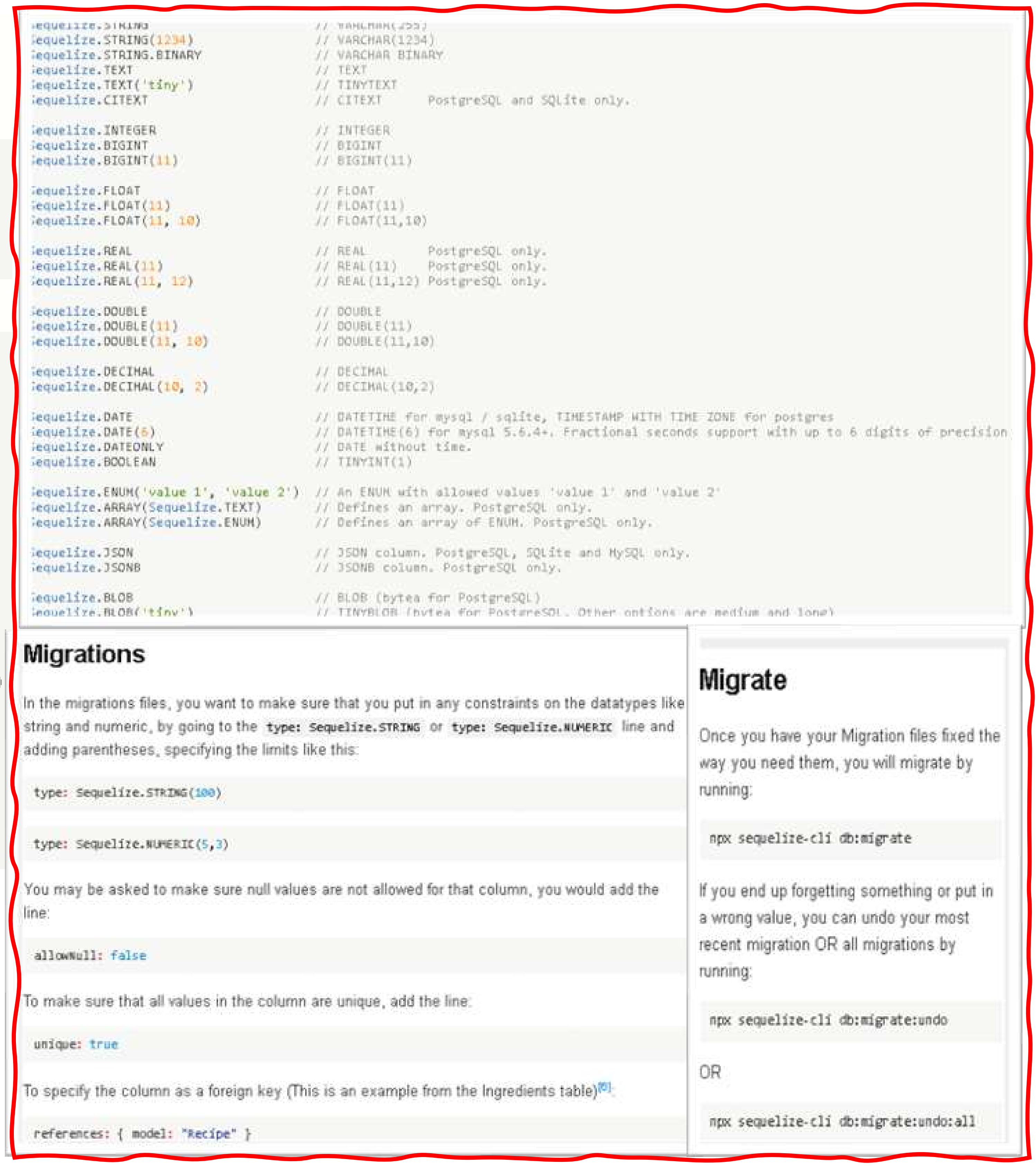
toJSON method

The confusingly named toJSON() method does **not** return a JSON string but instead returns a POJO for the instance.

```
// pet is an instance of the Pet class
const pet = await Pet.findByPk(1);
console.log(pet) // prints a giant object with
                 // tons of properties and methods
// petPOJO is now just a plain old Javascript Object
const petPOJO = pet.toJSON();
console.log(petPOJO); // { name: "Fido", petTypeId: 1 }
```

Common Where Operators

```
const Op = Sequelize.Op
[Op.and]: [{a: 5}, {b: 6}] // (a = 5) AND (b = 6)
[Op.or]: [{a: 5}, {a: 6}]  // (a = 5 OR a = 6)
[Op.gt]: 6,                // > 6
[Op.gte]: 6,               // >= 6
[Op.lt]: 10,               // < 10
[Op.lte]: 10,              // <= 10
[Op.ne]: 20,               // != 20
[Op.eq]: 3,                // = 3
[Op.is]: null              // IS NULL
[Op.not]: true,            // IS NOT TRUE
[Op.between]: [6, 10],     // BETWEEN 6 AND 10
[Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15
[Op.in]: [1, 2],           // IN [1, 2]
[Op.notIn]: [1, 2],        // NOT IN [1, 2]
[Op.like]: '%hat',         // LIKE '%hat'
[Op.notLike]: '%hat'       // NOT LIKE '%hat'
[Op.iLike]: '%hat'         // ILIKE '%hat' (case insensitive) (PG only)
[Op.notILike]: '%hat'      // NOT ILIKE '%hat'  (PG only)
[Op.startsWith]: 'hat'     // LIKE 'hat%'
[Op.endsWith]: 'hat'       // LIKE '%hat'
[Op.substring]: 'hat'      // LIKE '%hat%'
[Op.regexp]: '^[h|a|t]'    // REGEXP/~ '^[h|a|t]' (MySQL/PG only)
[Op.notRegexp]: '^[h|a|t]' // NOT REGEXP/!~ '^[h|a|t]' (MySQL/PG only)
[Op.iRegexp]: '^[h|a|t]'    // ~* '^[h|a|t]' (PG only)
[Op.notIRegexp]: '^[h|a|t]' // !~* '^[h|a|t]' (PG only)
[Op.like]: { [Op.any]: ['cat', 'hat']}
```



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

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

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

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

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

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

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

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

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

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

Migrations

In the migrations files, you want to make sure that you put in any constraints on the datatypes like string and numeric, by going to the `type: Sequelize.STRING` or `type: Sequelize.NUMERIC` line and adding parentheses, specifying the limits like this:

```
type: Sequelize.STRING(100)
```

```
type: Sequelize.NUMERIC(5,3)
```

You may be asked to make sure null values are not allowed for that column, you would add the line:

```
allowNull: false
```

To make sure that all values in the column are unique, add the line:

```
unique: true
```

To specify the column as a foreign key (This is an example from the Ingredients table)[8]:

```
references: { model: "Recipe" }
```

Migrate

Once you have your Migration files fixed the way you need them, you will migrate by running:

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

If you end up forgetting something or put in a wrong value, you can undo your most recent migration OR all migrations by running:

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

OR

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

Models and Associations

In your model’s files, you will specify the associations. You can have one-to-one, one-to-many, or many-to-many.

For our example, the Recipes and Instructions table have a one-to-many association, the Recipes and Instructions table have a one-to-many association, and the MeasurementUnits and Ingredients have a on-to-many relationship.

Let's associate the Recipes and Instructions first. We will go into the Recipes model file at `models/recipe.js` and in the `Recipe.associate` file. Since the Recipes table does not contain a foreign key, but is referenced by the Instructions table, we will call the `hasMany` function here:

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

In plain English, you can read the above code as: "Each recipe has many instructions, and each instruction references the recipe with the foreign key 'recipeId'."

Since the Instruction model contains a foreign key referencing the Recipe model, it *belongs to* the Recipe model. So, in the `models/instruction.js` we will associate the Instruction model to the Recipe model using the `belongsTo` function:

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

In plain English, you can read the above code as: "Each Instruction belongs to a recipe that is referenced by the foreign key 'recipeId'."

To set up the Recipes to Ingredients association,

In the `models/recipe.js` file, define the association as:

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

and in the `models/ingredient.js` file, define the association as:

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

Finally, for the Ingredients to MeasurementUnits associations,

In the `models/ingredient.js` file, define the association as:

```
Ingredient.belongsTo(models.MeasurementUnit, { foreignKey: 'measurementUnitId' });
```

and in the `models/measurementunit.js` file, define the association as:

```
MeasurementUnit.hasMany(models.Ingredient, { foreignKey: 'measurementUnitId' });
```

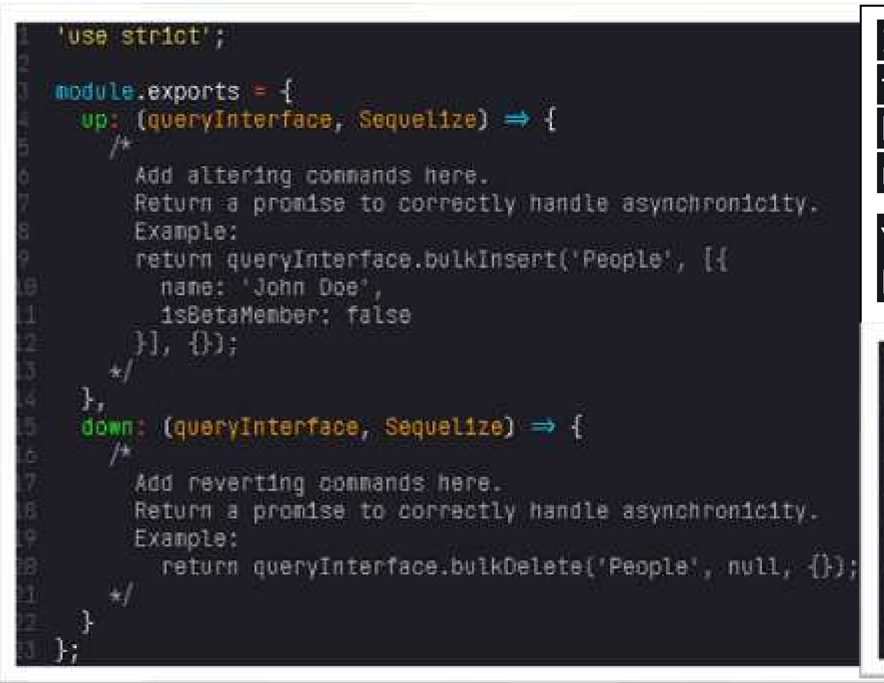
The many to many relationship is the only one that is ***really*** different

Seeding Tables

Now we need to seed the tables, so we will go back to the terminal and generate the seed files. You will run:

```
npx sequelize-cli seed:generate --name recipe-seeder
```

You can name the seeder file anything you would like. The file with the name you gave it will appear in your `seeders/` folder. We need to open it up and put in information for each item that we want to seed into the table. **NOTE:** *Make sure you provide the createdAt and updatedAt values, otherwise they will be considered null, which is not allowed.*

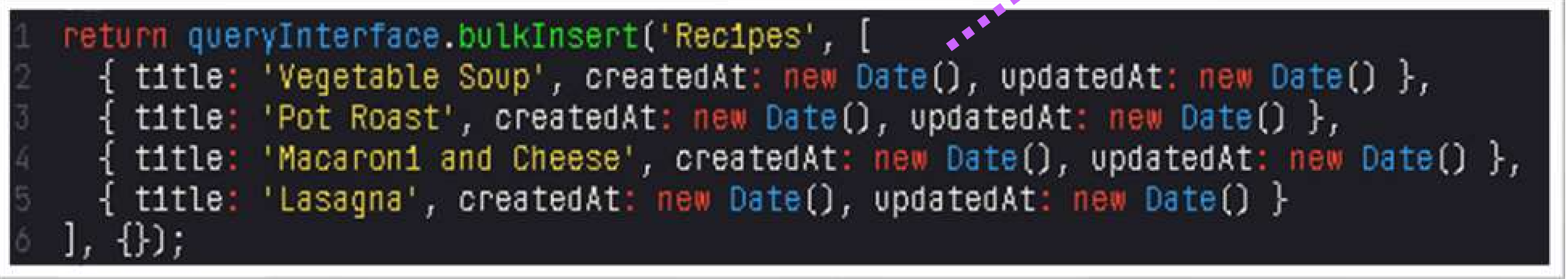


```
'use strict';

module.exports = {
  up: (queryInterface, Sequelize) => {
    /*
      Add altering commands here.
      Return a promise to correctly handle asynchronicity.
      Example:
      return queryInterface.bulkInsert('People', [{
        name: 'John Doe',
        isBetaMember: false
      }], {});
    */
  },
  down: (queryInterface, Sequelize) => {
    /*
      Add reverting commands here.
      Return a promise to correctly handle asynchronicity.
      Example:
        return queryInterface.bulkDelete('People', null, {});
    */
  }
};
```

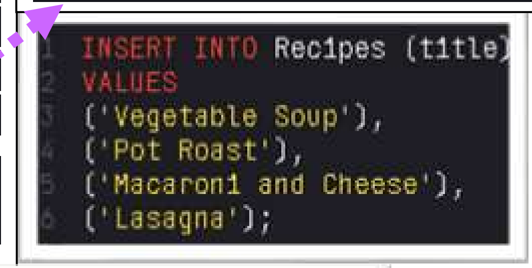
```
Inside of the up and down methods here, you see a comment with an "Example" in it. These are extremely helpful so you don't have to memorize exactly what to do here. You can take these out of the comments and use them, but make sure you change 'People' to the table name, which should be _plural_ here.
```

```
You will place your seed data as an array of POJOs in the bulkInsert method in the up section. The up method of the Recipes seeder will look a little like this:
```



```
return queryInterface.bulkInsert('Recipes', [
  { title: 'Vegetable Soup', createdAt: new Date(), updatedAt: new Date() },
  { title: 'Pot Roast', createdAt: new Date(), updatedAt: new Date() },
  { title: 'Macaroni and Cheese', createdAt: new Date(), updatedAt: new Date() },
  { title: 'Lasagna', createdAt: new Date(), updatedAt: new Date() }
], {});
```

```
This will translate into SQL as:
```



```
INSERT INTO Recipes (title)
VALUES
('Vegetable Soup'),
('Pot Roast'),
('Macaroni and Cheese'),
('Lasagna');
```

Accessing the Data

You can access and query the data using the `findByPk`, `findOne`, and `findAll` methods. First, make sure you import the models in your JavaScript file. In this case, we are assuming your JavaScript file is in the root of your project and so is the models folder.

```
const { Recipe, Ingredient, Instruction, MeasurementUnit } = require('./models');
```

The models folder exports each of the models that you have created. We have these four in our data model, so we will destructure the models to access each table individually. The associations that you have defined in each of your models will allow you to access data of related tables when you query your database using the `include` option.

If you want to find all recipes, for the recipe list, you would use the `findAll` method. You need to await this, so make sure your function is async.

```
async function findAllRecipes() {

  return await Recipe.findAll();

}
```

If you would like to include all the ingredients so you can create a shopping list for all the recipes, you would use `include`. This is possible because of the association you have defined in your Recipe and Ingredient models.

```
async function getShoppingList() {

  return await Recipe.findAll({ include: [ Ingredient ] });

}
```

If you only want to find one where there is chicken in the ingredients list, you would use `findOne` and `findByPk`.

```
async function findAChickenRecipe() {

  const chickenRecipe = await Ingredient.findOne({

    where: {

      foodStuff: 'chicken'

    }

  });

  return await Recipe.findByPk(chickenRecipe.recipeId);

}
```

Data Access to Create/Update/Delete Rows

You have two options when you want to create a row in a table (where you are saving one record into the table). You can either `.build` the row and then `.save` it, or you can `.create` it. Either way it does the same thing. Here are some examples:

Let's say we have a form that accepts the name of the recipe (for simplicity). When we get the results of the form, we can:

```
const newRecipe = await Recipe.build({ title: 'Chicken Noodle Soup' });

await newRecipe.save();
```

This just created our new recipe and added it to our Recipes table. You can do the same thing like this:

```
await Recipe.create({ title: 'Chicken Noodle Soup' });
```

If you want to modify an item in your table, you can use `update`. Let's say we want to change the chicken noodle soup to chicken noodle soup with extra veggies, first we need to get the recipe, then we can update it.

```
const modRecipe = await Recipe.findOne({ where: { title: 'Chicken Noodle Soup' } });

await modRecipe.update({ title: 'Chicken Noodle Soup with Extra Veggies' });
```

To delete an item from your table, you will do the same kind of process. Find the recipe you want to delete and `destroy` it, like this:

```
const deleteThis = await Recipe.findOne({ where: { title: 'Chicken Noodle Soup with Extra Veggies' } });

await deleteThis.destroy();
```

NOTE: If you do not await these, you will receive a promise, so you will need to use `.then` and `.catch` to do more with the items you are accessing and modifying.

Documentation

For the data types and validations in your models, here are the official docs. The sequelize docs are hard to look at, so these are the specific sections with just the lists:
Sequelize Data Types: *https://sequelize.org/v5/manual/data-types.html*
Validations: *https://sequelize.org/v5/manual/models-definition.html#validations*
When you access the data in your queries, here are the operators available, again because the docs are hard to navigate, this is the specific section with the list of operators.
Operators: *https://sequelize.org/v5/manual/querying.html#operators*
The documentation for building, saving, creating, updating and destroying is linked here, it does a pretty good job of explaining in my opinion, it just has a title that we have not been using in this course. When they talk about an instance, they mean an item stored in your table.
Create/Update/Destroy: *https://sequelize.org/v5/manual/instances.html*