- ⏱️ 21 minutes
- ✅ Completed

Data-Driven Websites - Part 2: Integrating Sequelize with Express

Welcome to part two of creating the data-driven Reading List website!

Over the course of three articles, you'll create a data-driven Reading List website that will allow you to view a list of books, add a book to the list, update a book in the list, and delete a book from the list. In the first article, you created the project. In this article, you'll learn how to integrate Sequelize with an Express application. In the last article, you'll create the routes and views to perform CRUD (create, read, update, and delete) operations using Sequelize.

When you finish this article, you should be able to:

- Install and configure Sequelize within an Express application; and
- Use Sequelize to test the connection to a database before starting the HTTP server on application startup.

You'll also review the following:

- Using the Sequelize CLI to create a model and migration;
- Using the Sequelize CLI to seed the database; and
- Using Sequelize to query data from the database.

Installing and configuring Sequelize

First things first, you need to install and configure Sequelize!

Use npm to install the following dependencies:

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

Then install the Sequelize CLI as a development dependency:

```
npm install sequelize-cli@^5.0.0 --save-dev
```

Configuring the Sequelize CLI

Before using the Sequelize CLI to initialize Sequelize within your project, add a file named `.sequelizerc` to the root of your project containing the following code:

```
const path = require('path');

module.exports = {
  'config': path.resolve('config', 'database.js'),
  'models-path': path.resolve('db', 'models'),
  'seeders-path': path.resolve('db', 'seeders'),
  'migrations-path': path.resolve('db', 'migrations')
};
```

The `.sequelizerc` file configures the Sequelize CLI so that it knows:

- Where your database configuration is located; and
- Where to generate the `models`, `seeders`, and `migrations` folders.

Initializing Sequelize

Now you're ready to initialize Sequelize by running the following command:

```
npx sequelize init
```

When the command completes, your project should now contain the following:

- The `config/database.js` file;
- The `db/migrations`, `db/models`, and `db/seeders` folders; and
- The `db/models/index.js` file.

Creating a new database and database user

To prepare for configuring Sequelize, you need to create a new database for the Reading List application to use and a new normal or limited user (i.e. a user without superuser privileges) that has permissions to access the new database.

Open psql by running the command `psql` (to use the currently logged in user) or `psql -U «super user username»` to specify the username of the super user to use. Then execute the following SQL statements:

```
create database reading_list;
create user reading_list_app with encrypted password '«a strong
password for the reading_list_app user»';
```

```
grant all privileges on database reading_list to
reading_list_app;
```

Make note of the password that you use as you'll need them for the next step in the configuration process!

To review how to create a new PostgreSQL database and user, see the "Database Management Walk-Through" and "User Management Walk-Through" readings in the SQL lesson.

Adding the database environment variables

Now you're ready to add the `DB_USERNAME`, `DB_PASSWORD`, `DB_DATABASE`, and `DB_HOST` environment variables to the `.env` and `.env.example` files:

```
PORT=8080
DB_USERNAME=reading_list_app
DB_PASSWORD=«the reading_list_app user password»
DB_DATABASE=reading_list
DB_HOST=localhost
```

Next, update the `config` module (the `config/index.js` file) with the following code:

```
// ./config/index.js

module.exports = {
  environment: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 8080,
  db: {
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE,
    host: process.env.DB_HOST,
  },
};
```

Remember that the `config` module is responsible for providing access to your application's environment variables. Any part of the application that needs access to the `DB_USERNAME`, `DB_PASSWORD`, `DB_DATABASE`, and `DB_HOST` environment variables can use the `username`, `password`, `database`, and `host` properties on the `config` module's `db` object.

Configuring the Sequelize database connection

Now you're ready to configure the database connection for Sequelize! Update the `config/database.js` file with the following code:

```
const {
  username,
  password,
  database,
  host,
} = require('./index').db;

module.exports = {
  development: {
    username,
    password,
    database,
    host,
    dialect: 'postgres',
  },
};
```

The first statement uses the `require()` function to import the `config/index` module. Destructuring is used to declare the `username`, `password`, `database`, and `host` variables and initialize them to the values of the corresponding property names on the `config` module's `db` property.

You could remove the destructuring by refactoring the above statement into the following statements:

```
const config = require('./index');

const db = config.db;
const username = db.username;
const password = db.password;
const database = db.database;
const host = db.host;
```

The `config/database` module then exports an object with a property named `development` set to an object literal with `username`, `password`, `database`, `host`, and `dialect` properties:

```
module.exports = {
  development: {
    username,
    password,
    database,
    host,
    dialect: 'postgres',
  },
};
```

The `development` property name indicates that these configuration settings are for the `development` environment. The `username`, `password`, `database`, `host`,

and `dialect` property names are the Sequelize options used to configure the database connection.
For a complete list of the available Sequelize options, see the official Sequelize API documentation.

Testing the connection to the database

You've configured Sequelize—specifically the database connection—but how do you know if Sequelize can actually connect to the database? The Sequelize instance method `authenticate()` can be used to test the connection to the database by attempting to execute the query `SELECT 1+1 AS result` against the database specified in the `config/database` module.
The Reading List application is a data-driven website, so it'll be heavily dependent on its database. Given that, it's best to test the connection to the database as early as possible. You can do that by updating the `./bin/www` file to test the connection to the database before starting the application listening for HTTP connections.
Update the `./bin/www` file with the following code:

```
#!/usr/bin/env node

const { port } = require('../config');

const app = require('../app');
const db = require('../db/models');

// Check the database connection before starting the app.
db.sequelize.authenticate()
  .then(() => {
    console.log('Database connection success! Sequelize is
ready to use...');

    // Start listening for connections.
    app.listen(port, () => console.log(`Listening on port
${port}...`));
  })
  .catch((err) => {
    console.log('Database connection failure.');
    console.error(err);
  });
```

In addition to importing the `app` module, the `require()` function is called to import the `./db/models` module—a module that was generated by the Sequelize CLI when you initialized your project to use Sequelize.
The `./db/models` module provides access to the Sequelize instance via the `sequelize` property. The `authenticate()` method is called on the Sequelize

instance. The `authenticate()` method is asynchronous, so it returns a Promise that will resolve if the connection to the database is successful, otherwise it will be rejected:

```
// Check the database connection before starting the app.
db.sequelize.authenticate()
  .then(() => {
    // The connection to the database succeeded.
  })
  .catch((err) => {
    // The connection to the database failed.
  });
```

Inside of the `then()` method callback, a message is logged to the console and the application is started listening for HTTP connections. Inside of the `catch()` method callback, an error message and the `err` object are logged to the console:

```
// Check the database connection before starting the app.
db.sequelize.authenticate()
  .then(() => {
    console.log('Database connection success! Sequelize is
ready to use...');

    // Start listening for connections.
    app.listen(port, () => console.log(`Listening on port
${port}...`));
  })
  .catch((err) => {
    console.log('Database connection failure.');
    console.error(err);
  });
```

To test the connection to the database, run the command `npm start` in the terminal to start your application. In the console, you should see the success message if the database connection succeeded, otherwise you'll see the error message.

Creating the Book model

Now that you've confirmed that the application can successfully connect to the database, it's time to create the application's first model.

As a reminder, the Reading List website—when it's completed—will allow you to view a list of books, add a book to the list, update a book in the list, and delete a book from the list. At the heart of all of these features are books, so let's use the Sequelize CLI to generate a `Book` model.

The `Book` model should include the following properties:

- `title` - A string representing the title;
- `author` - A string representing the the author;
- `releaseDate` - A date representing the release date;

- `pageCount` - An integer representing the page count; and
- `publisher` - A string representing the publisher.

From the terminal, run the following command to use the Sequelize CLI to generate the `Book` model:

```
npx sequelize model:generate --name Book --attributes "title:string,
author:string, releaseDate:dateonly, pageCount:integer,
publisher:string"
```

If the command succeeds, you'll see the following output in the console:

```
New model was created at [path to the project
folder]/db/models/book.js .
New migration was created at [path to the project
folder]/db/migrations/[timestamp]-Book.js .
```

This confirms that two files were generated: a file for the `Book` model and a file for a database migration to add the `Books` table to the database.

Updating the generated `Book` model and migration files

The `Book` model and migration files generated by the Sequelize CLI are close to what is needed, but some changes are required. Two things in particular need to be addressed: column string lengths and column nullability (i.e. the ability for a column to accept `null` values).

For your reference, here are the generated model and migration files:

```
// ./db/models/book.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const Book = sequelize.define('Book', {
    title: DataTypes.STRING,
    author: DataTypes.STRING,
    releaseDate: DataTypes.DATEONLY,
    pageCount: DataTypes.INTEGER,
    publisher: DataTypes.STRING
  }, {});
  Book.associate = function(models) {
    // associations can be defined here
  };
  return Book;
};
// ./db/migrations/[timestamp]-create-book.js

'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Books', {
```

```
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      title: {
        type: Sequelize.STRING
      },
      author: {
        type: Sequelize.STRING
      },
      releaseDate: {
        type: Sequelize.DATEONLY
      },
      pageCount: {
        type: Sequelize.INTEGER
      },
      publisher: {
        type: Sequelize.STRING
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Books');
  }
};
```

As it is, the generated migration file would create the following `Books` table in the database:

```
reading_list=# \d "Books"
                                        Table "public.Books"
   Column    |           Type           | Collation | Nullable |
Default
-------------+--------------------------+-----------+----------+----
--------------------------------
 id          | integer                  |           | not null |
nextval('"Books_id_seq"'::regclass)
 title       | character varying(255)   |           |          |
 author      | character varying(255)   |           |          |
 releaseDate | date                     |           |          |
 pageCount   | integer                  |           |          |
```

```
 publisher   | character varying(255)   |           |          |
 createdAt   | timestamp with time zone |           | not null |
 updatedAt   | timestamp with time zone |           | not null |
Indexes:
    "Books_pkey" PRIMARY KEY, btree (id)
```

Notice that all of the `Book` `string` based properties (i.e. `title`, `author`, and `publisher`) resulted in columns with a data type of `character varying(255)`, which is a variable length text based column up to 255 characters in length. Allowing for 255 characters for the `title` column seems about right, but for the `author` and `publisher` columns, it seems excessive.

Also notice that the `title`, `author`, `releaseDate`, `pageCount`, and `publisher` columns all allow `null` values (a value of `not null` in the "Nullable" column means that the column doesn't allow `null` values, otherwise the column allows `null` values). Ideally, each book in the database would have values for all of those columns.

We can address both of these issues by updating the `./db/models/book.js` file to the following code:

```
// ./db/models/book.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const Book = sequelize.define('Book', {
    title: {
      type: DataTypes.STRING,
      allowNull: false
    },
    author: {
      type: DataTypes.STRING(100),
      allowNull: false
    },
    releaseDate: {
      type: DataTypes.DATEONLY,
      allowNull: false
    },
    pageCount: {
      type: DataTypes.INTEGER,
      allowNull: false
    },
    publisher: {
      type: DataTypes.STRING(100),
      allowNull: false
    }
  }, {});
  Book.associate = function(models) {
    // associations can be defined here
  };
  return Book;
};
```

The migration file `./db/migrations/[timestamp]-create-book.js` also needs to be updated to the following code:

```
// ./db/migrations/[timestamp]-create-book.js

'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Books', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      title: {
        type: Sequelize.STRING,
        allowNull: false
      },
      author: {
        type: Sequelize.STRING(100),
        allowNull: false
      },
      releaseDate: {
        type: Sequelize.DATEONLY,
        allowNull: false
      },
      pageCount: {
        type: Sequelize.INTEGER,
        allowNull: false
      },
      publisher: {
        type: Sequelize.STRING(100),
        allowNull: false
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Books');
  }
};
```

Applying the pending migration

After resolving the column data type and nullability issues in the model and migration files, you're ready to apply the pending migration to create the `Books` table in the database. In the terminal, run the following command:

```
npx dotenv sequelize db:migrate
```

Notice that you're using npx to invoke the `dotenv` tool which loads your environment variables from the `.env` file and then invokes the `sequelize db:migrate` command. In the console, you should see something similar to the following output:

```
Loaded configuration file "config/database.js".
Using environment "development".
== [timestamp]-create-book: migrating =======
== [timestamp]-create-book: migrated (0.021s)
```

To confirm the creation of the `Books` table, you can run the following command from within psql:

```
\d "Books"
```

Be sure that you're connected to the `reading_list` database in psql. If you are, the cursor should read `reading_list=#`. If you're not connected to the correct database, you can run the command `\c reading_list` to connect to the `reading_list` database.

After running the `\d "Books"` command, you should see the following output within psql:

```
                                      Table "public.Books"
   Column    |           Type           | Collation | Nullable |
Default
-------------+--------------------------+-----------+----------+----
--------------------------------
 id          | integer                  |           | not null |
nextval('"Books_id_seq"'::regclass)
 title       | character varying(255)   |           | not null |
 author      | character varying(100)   |           | not null |
 releaseDate | date                     |           | not null |
 pageCount   | integer                  |           | not null |
 publisher   | character varying(100)   |           | not null |
 createdAt   | timestamp with time zone |           | not null |
 updatedAt   | timestamp with time zone |           | not null |
Indexes:
    "Books_pkey" PRIMARY KEY, btree (id)
```

Seeding the database

With the `Books` table created in the database, you're ready to seed the table with some test data!

To start, you need to create a seed file by running the following command in the terminal from the root of your project:

```
npx sequelize seed:generate --name test-data
```

If the command succeeds, you'll see the following output in the console:

```
seeders folder at "[path to the project folder]/db/seeders" already
exists.
New seed was created at [path to the project
folder]/db/seeders/[timestamp]-test-data.js .
```

This confirms that the seed file was generated. Go ahead and replace the contents of the `./db/seeders/[timestamp]-test-data.js` with the following code:

```
// ./db/seeders/[timestamp]-test-data.js

'use strict';

module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.bulkInsert('Books', [
      {
        title: 'The Martian',
        author: 'Andy Weir',
        releaseDate: new Date('2014-02-11'),
        pageCount: 384,
        publisher: 'Crown',
        createdAt: new Date(),
        updatedAt: new Date()
      },
      {
        title: 'Ready Player One',
        author: 'Ernest Cline',
        releaseDate: new Date('2011-08-16'),
        pageCount: 384,
        publisher: 'Crown',
        createdAt: new Date(),
        updatedAt: new Date()
      },
      {
        title: 'Harry Potter and the Sorcerer\'s Stone',
        author: 'J.K. Rowling',
        releaseDate: new Date('1998-10-01'),
        pageCount: 309,
        publisher: 'Scholastic Press',
        createdAt: new Date(),
        updatedAt: new Date()
      },
    ], {});
  },
```

```
  down: (queryInterface, Sequelize) => {
    return queryInterface.bulkDelete('Books', null, {});
  }
};
```

The `up` property references an anonymous method that uses the `queryInterface.bulkInsert()` method to insert an array of books into the `Book` table while the `down` property references an anonymous method that uses the `queryInterface.bulkDelete()` method to delete all of the data in the `Books` table.

Feel free to add to the array of books... have fun with it!

To seed your database with your test data, run the following command:

```
npx dotenv sequelize db:seed:all
```

In the console, you should see something similar to the following output:

```
Loaded configuration file "config/database.js".
Using environment "development".
== [timestamp]-test-data: migrating =======
== [timestamp]-test-data: migrated (0.009s)
```

Then you can use psql to check if the `Books` table contains the test data:

```
select * from "Books";
```

Which should produce the following output:

```
 id |                 title                 |    author    |
releaseDate | pageCount |    publisher     |         createdAt
|         updatedAt
----+---------------------------------------+--------------+--------
-----+-----------+------------------+----------------------------+--
-------------------------
  2 | The Martian                           | Andy Weir    | 2014-
02-11  |       384 | Crown            | 2020-03-31 19:06:32.452-07 |
2020-03-31 19:06:32.452-07
  3 | Ready Player One                      | Ernest Cline | 2011-
08-16  |       384 | Crown            | 2020-03-31 19:06:32.452-07 |
2020-03-31 19:06:32.452-07
  4 | Harry Potter and the Sorcerer's Stone | J.K. Rowling | 1998-
10-01  |       309 | Scholastic Press | 2020-03-31 19:06:32.452-07 |
2020-03-31 19:06:32.452-07
(3 rows)
```

Querying and rendering a temporary list of books

Now that you've installed and configured Sequelize, created the `Book` model and associated migration, and seeded the `Books` table, you're ready to update your application's default route to query the for a list of books and render the data in the `index` view!

In the `routes` module (the `./routes.js` file), use the `require()` function to import the `models` module:

```
const db = require('./db/models');
```

Then update the default route (`/`) to this:

```
router.get('/', async (req, res, next) => {
  try {
    const books = await db.Book.findAll({ order: [['title',
'ASC']] });
    res.render('index', { title: 'Home', books });
  } catch (err) {
    next(err);
  }
});
```

The `async` keyword was added to make the route handler an asynchronous function and the `db.Book.findAll()` method is used to retrieve a list of books from the database.

For your reference, here's what the complete `./routes.js` file should look like:

```
// ./routes.js

const express = require('express');

const db = require('./db/models');

const router = express.Router();

router.get('/', async (req, res, next) => {
  try {
    const books = await db.Book.findAll({ order: [['title',
'ASC']] });
    res.render('index', { title: 'Home', books });
  } catch (err) {
    next(err);
  }
});

module.exports = router;
```

Now you can update the `./views/index.pug` view to render the array of books:

```
extends layout.pug
```

```
block content
  p Hello from the Reading List app!
  h3 Books
  ul
    each book in books
      li= book.title
```

For now, the formatting of the book list is very simple. In the next article, you'll see how to use Bootstrap to improve the look and feel of the book list table.

Run the command `npm start` to start your application (if it's not already started) and browse to `http://localhost:8080/`. You should see the list of books from the database rendered to the page in an unordered list!

What you learned

In this article, you learned how to:

- Install and configure Sequelize within an Express application; and
- Use Sequelize to test the connection to a database before starting the HTTP server on application startup.

You also reviewed the following:

- Using the Sequelize CLI to create a model;
- Using the Sequelize CLI to seed the database; and
- Using Sequelize to query data from the database.

Next up: creating the routes and views to perform CRUD (create, read, update, and delete) operations using Sequelize!