# ORM with Sequelize

## Learning Objectives

- Understand what Object-Relational mapping is
- Understand the advantage of following an ORM pattern
- Understand the basics of using the Sequelize library

## Material

### What is ORM?

Classes and tables both represent a type of entity in our system. For example, if we were creating an application which handled users, we'd likely implement a User class in our code and we'd persist their data in a users table in the database. Individual instances of the User class (objects) would correspond to individual rows in the users table. The User class would encapsulate the logic for saving itself to and loading itself from the database. This mapping between class instances (objects) and SQL tables (relations) is called Object-Relational mapping (ORM).

### Why ORM?

Object-relational mapping allows the rest of our code to interact with objects and classes, rather than rows and tables. It abstracts away the implementation details of how exactly the entities are stored on disk.

### Sequelize

Sequelize is an ORM library. By including it in our projects, we can simply write JavaScript code as normal and, behind the scenes, Sequelize will convert our method calls to SQL queries.

### Creating a Database Connection

First, we need to install sequelize and sqlite3 (which sequelize depends on when using sqlite as the RDBMS).

```
npm i sequelize sqlite3
```

Next, we import sequelize and configure a new database connection. We then export this connection.

```
// db.js
const { Sequelize } = require("sequelize");
const path = require("path");

const dbPath = path.join(__dirname, "db.sqlite");

const sequelize = new Sequelize({
    dialect: "sqlite",
    storage: dbPath,
});

module.exports = sequelize;
```

Whenever we deal with the file-system in Node.js, we should use its in-built path module. `path.join(__dirname, "db.sqlite")` returns a path to the same directory as `db.js` but with `db.sqlite` as the filename.

Notice also how we export small-s `sequelize` - this is our connection to the database not the Sequelize module itself.

## Defining Models

Models are the entities in our system - they're the tables and classes we're going to work with. Sequelize provides us with a base class Model which we can extend. For now, let's assume our database only needs one table to hold the names and population counts of cities:

```
// city.js
const sequelize = require("./db");
const { DataTypes, Model } = require("sequelize");

class City extends Model {}

City.init(
    {
        name: DataTypes.STRING,
        population: DataTypes.INTEGER,
    },
    {
        sequelize,
        modelName: "city",
        timestamps: false,
    }
);

module.exports = City;
```

Let's unpack this.

First, we extend Sequelize's base `Model` to create a `City` model; this will give `City` access to `Model`'s functionality.

Next, we initialize City with its particular data - this is what distinguishes City from any other Sequelize model. The first argument to `init` are the table columns: name and population. We also tell Sequelize what DataType each should be, using its in-built DataTypes object.

The next argument to init is the options object for this model. `sequelize` is the connection to the database we created before; this tells the model which database it belongs to. `modelName` defines what Sequelize will call the model - it is not strictly necessary but I'd recommend setting it to the lower-case version of the class' name. `timestamps: false` stops Sequelize's default behaviour of adding `created_at` and `updated_at` columns to your table.

## Creating and Using our Database

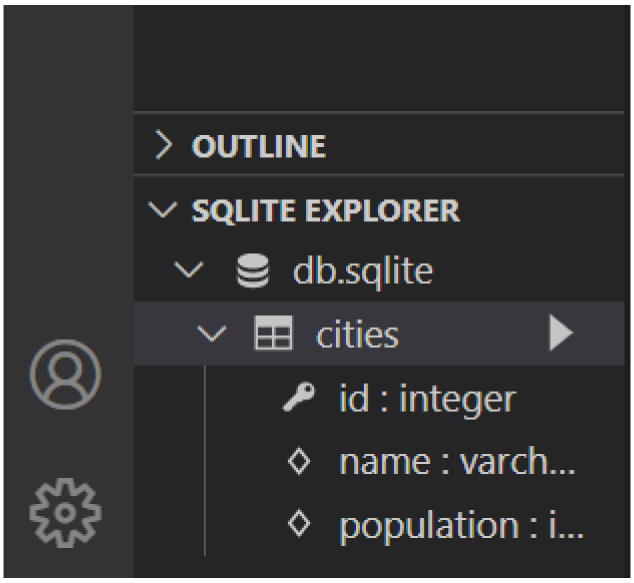
Our database is now configured so let's create it and start using it. We can create a sandbox.js file to experiment with sequelize:

```
// sandbox.js
const sequelize = require("./db");
const City = require("./city");

async function sandbox() {
    await sequelize.sync();
}
sandbox();
```

Sequelize is a promise-based library (remember, database queries are slow so we want to handle them using callbacks or promises) so we create an async function to hold all our code so we can use the `await` keyword. It's a pretty safe bet that, whenever you see `await` before a sequelize method, it's doing something with the database.

The `sync()` method will create the database for us if it doesn't already exist. If you run this, you'll see a db.sqlite file will be created in our directory. This is our database. If you're using VSCode, you can search the extensions tab for "SQLite" and install the extension created by "alexcvzz". Once installed, you can right-click your db.sqlite file and select "Open Database". This will create a new tab in your Explorer:

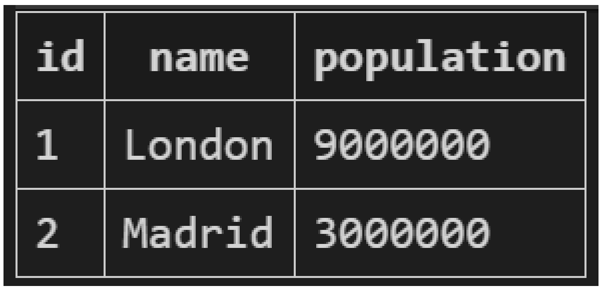


We can see that Sequelize has pluralised our "city" model name to a "cities" table. It has also automatically added an "id" column as a primary key.

If we click the play icon, it will open the table in a tab. Currently it's empty so let's add some rows:

```
async function sandbox() {
    await sequelize.sync();
    const london = await City.create({
        name: "London",
        population: 9000000,
    });
    const madrid = await City.create({
        name: "Madrid",
        population: 3000000,
    });
}
sandbox();
```

Notice that, although City is a class, Sequelize recommends we do not use the `new` keyword to create an instance. Instead we use the static `create` method which builds a new instance and saves it as a row in the database. If we hit the play button on cities, we can see two rows in our database:



| id | name | population |
|---|---|---|
| 1 | London | 9000000 |
| 2 | Madrid | 3000000 |

Notice how Sequelize has auto-incremented the “id” primary key for us.

## Relationships

Let’s imagine we want to add landmarks to our cities database. For now, we’ll just store the name of our landmarks. To do this, we can create a new Landmark model and export it:

```
// landmark.js
const sequelize = require("./db");
const { DataTypes, Model } = require("sequelize");

class Landmark extends Model {}

Landmark.init(
    {
        name: DataTypes.STRING,
    },
    {
        sequelize,
        modelName: "landmark",
        timestamps: false,
    }
);

module.exports = Landmark;
```

We now need to tell Sequelize how landmarks are related to cities. We can create an additional function to define all our relationships and sync the database:

```
// setupDb.js
const City = require("./city");
const Landmark = require("./landmark");
const db = require("./db");

async function setupDb() {
    City.hasMany(Landmark);
    Landmark.belongsTo(City);
    await db.sync();
}

module.exports = setupDb;
```

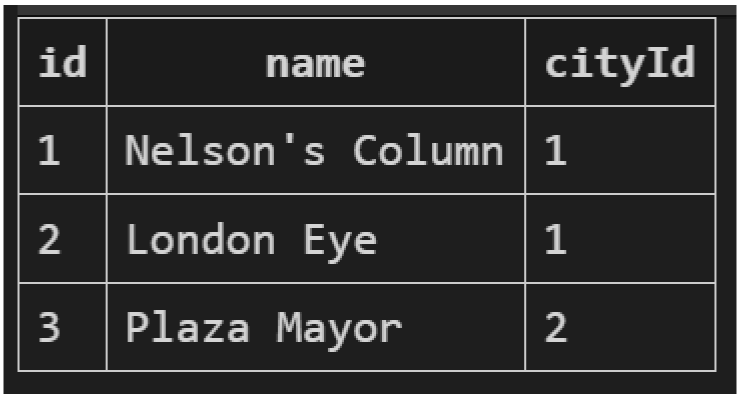
The `hasMany` and `belongsTo` methods tell Sequelize that a city has multiple landmarks, but a landmark belongs to only one city. Under the hood, Sequelize will create an extra column on a landmark called `cityId` - a foreign key mapping to a particular city.

Sequelize will also create an extra method on `City` called `createLandmark` - this creates a new landmark in the database and sets its `cityId` to that of the city instance the method was called on:

```
// sandbox.js
const City = require("./city");
const setupDb = require("./setupDb");

async function sandbox() {
    await setupDb();
    const london = await City.create({
        name: "London",
        population: 9000000,
    });
    const madrid = await City.create({
        name: "Madrid",
        population: 3000000,
    });
    const nelson = await london.createLandmark({
        name: "Nelson's Column",
    });
    const eye = await london.createLandmark({
        name: "London Eye",
    });
    const plaza = await madrid.createLandmark({
        name: "Plaza Mayor",
    });
}
sandbox();
```

After running this, the landmarks table is as follows:



| id | name | cityId |
|---|---|---|
| 1 | Nelson's Column | 1 |
| 2 | London Eye | 1 |
| 3 | Plaza Mayor | 2 |

Notice that `City.create` is a static method: we call it on the `City` class itself. `createLandmark`, on the other hand, is not static; it is called on an instance of a `City`, since the landmark belongs to a particular city.

## Core Assignments

### *Cinema Sequel*

Imagine you're a developer working on the online booking system for a chain of cinemas. Your system's database should include the following tables:

- Cinemas: their location and number of screens.
- Movies: their title and duration.
- Screenings: the movie, cinema, start time and screen number.

You will need to decide what you think the appropriate data-type is for each column.

Implement this database by creating the appropriate Sequelize models and relationships. Confirm it works as expected by creating cinemas, movies and screenings and checking they appear correctly in the database.

## Extension Assignments

### *Cinema Sequel++*

Complete the same assignment without using the Sequelize library. Instead, use a library like sqlite3 and write classes for the cinemas, movies and screenings which contain methods such as `save`, `update` etc to write raw SQL.

## Additional Resources

- Cities and landmarks example on GitHub