Node.JS and SQL - Section 5

An Introduction to Sequelize

- ORM stands for Object Relationship Mapping, it's a piece of software that abstracts away the complexity of having to write queries and makes interacting with databases as natural as possible for us as developers.
- Sequelize is a really popular ORM that is widely used to interact with MySQL and can be installed using

```
$ npm install sequelize mysql2
```

Initiating a Connection

- A sequelize connection can be initiated using the Sequelize function imported from the package, you need to pass certain connection options and the connection is good to go.

```
const { Sequelize } = require("sequelize");
const sequelize = new Sequelize({
 dialect: "mysql", // the flavor, or dialect of SQL
 port: <port>, // Port the server is running on, default 3306
 username: <username>, // username with which to connect to the DB
 password: <password>, // password for the username provided
 database: <database> // database to use
});
```

- The connection can then be opened and checked if it works using the authenticate method.

```
await sequelize.authenticate().catch((err) /> {
 console.error(“unable to connect to SQL\n”, err);
});
```

Models

- A model is the core essence of Sequelize, it’s the abstraction for a table in the database, it allows us to define and update tables straight from our application’s code.
- A model can be defined using the define method on sequelize:

```
sequelize.define(“<model_name>”, attributes, options);

sequelize.define(“User”, {
 email: {
 primaryKey: true,
 allowNull: false,
 unique: true,
 type: DataTypes.STRING
 },
 name: {
 type: DataTypes.STRING,
 allowNull: false
 }
});
```

Inserting and Reading Data

- Data can be inserted into a table by using the “create” method on a model.

```
<Model_Name>.create({
 property1: value1,
 property2: value2
});
```

- We can use one of the many methods that exist on the Model to find the data we need.

1. `findAll`: returns all the entries which match the criteria.
2. `findOne`: returns the first entry to match the criteria specified.
3. `findByPk`: finds an entry by the primary key.

Updating and Deleting Data

- An instance of the model can be updated either by manipulating the attribute directly or by bulk updating the attributes using the set method.

```
const user = await User.findByPk(email);
user.firstName = “Jane”; // by manipulating attribute
user.set({ // by using set method
 age: 32,
 salary: 72000
});

await user.save();
```

- An instance of a model can be deleted using the “destroy” method

```
const user = await User.findByPk(email);

await user.destroy();
```

One to One Relationship

> https://vertabelo.com/blog/one-to-one-relationship-in-database/

- Say we have a Users Table and a Company table, each company has exactly one admin, to define that relationship we can use the association syntax in sequelize. The syntax needs us to specify the relation ship in both the DBs where, we state that the company “belongs to” a user and the user can “have one” company.

The foreign key option here is used to define the foreign key to be created in the table marked as the the table which will belong to the other.

```
const User = sequelize.define(“User”, <attributes>);
const Company = sequelize.define(“Company”, <attributes>);

User.hasOne(Company, { foreignKey: “owner” });
Company.belongsTo(User, { foreignKey: “owner” });
```

One to Many Relationship

- One to many relationship is when one model owns multiple entities of another type, let’s take the example where we have one user who can have multiple posts, and we define this relationship with the foreign key of “creator” in the posts table.

```
const User = sequelize.define(“User”, <attributes>);
const Post = sequelize.define(“Post”, <attributes>);

User.hasMany(Post, { foreignKey: “creator” });
Post.belongsTo(User, { foreignKey: “creator” });
```

Many to Many (M:N) Association

- Let's take the following case, we have a table of Projects and a Table of Employees, a Project can have many employees and an employee could be working on multiple projects at once. Or the case where one class can have many students and the student can be taking many classes, in such cases we have 2 tables which are associated to each other in a fashion know as M:N or Many to Many.

In a Many to Many association we use a Junction Table, or a Join table to describe the relationship so as to not mess up the structure of the data and turning it into a mess in one of the other tables which controls the relationship.

- A many to many association in Sequelize can be defined using the belongs to many method.

```
const Company = sequelize.define("Company", <attributes>);
const Project = sequelize.define("Project", <attributes>);
const CompanyProjects = sequelize.define("CompanyProjects",{});

Company.belongsToMany(Project, { through: CompanyProjects });
Project.belongsToMany(Company, { through: CompanyProjects });
```

- Now we can define a User or a Project while creating one of the entities using the include option, or using the add relationship methods on any of the entities.

```
const company = await Company.create({ // creating using the include option
 companyName: "TalentLabs",
 Projects: [{
 title: "Backend CABE"
 }, {
 title: "JavaScript M1"
 }]
}, {
 include: [Project]
});
const company = await Company.findByPk(companyId); // using add relation method
const project = await Project.findByPk(projectId);

company.addProject(project, { through: CompanyProjects });
```

A dive into Models (Tutorial)

main.js:

```
// const { Sequelize, Op, Model, DataTypes } = require("sequelize");
// const sequelize = new Sequelize("sqlite::memory:");
const { Sequelize } = require("sequelize"); // 1) to  require Sequelize services
```

```
// 2) to connect to the database, you must create a Sequelize instance
const db = new Sequelize({
  dialect: "mysql2",
  host: "127.0.0.1",
  username: "root",
  password: "pass",
  port: 3306,
  database: "sequelize", // database name
});

// 5) Model define
const User = db.define('User', {
  // 6) Model attributes are defined here
  id: {
    // REQUIRED
    type: DataTypes.INTEGER,
    // OPTIONAL
    autoIncrement: true,
    primaryKey: true,
  },
  firstName: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  lastName: {
    type: DataTypes.STRING,
    allowNull: true,
    // defaultValue: "Doe",
  },
  email: {
    type: DataTypes.STRING,
    allowNull: false,
    unique: true,
  }
})

// 3) test connection
const run = async () => {
  try {
    // 3) .authenticate() function to test if the connection is OK
    await db.authenticate();
    // 4) .sync() method to create or update database tables based on your defined
models
    await db.sync();
    // 7) Model Instances - we create entities (values)
    const user = await User.create({
      firstName: "Jane",
      lastName: "Smith",
      email: "jane@smith.com"
    })

    const user = await User.findAll({})

    const user = await User.findAll({ where: { id: 12  } })
```

```
    const bob = await User.findByPk(1); // by Primary Key

    const bob = await User.findOne({ where: { firstName: "Bob" } })

    const users = await User.findAll({ where: {email: { [Op.like]: "%doe.com" } }
})
     // 8) debug
  console.log(user, "\n", user.toJSON());
  console.log(bob.toJSON());
    console.log("Connection has been established successfully.");
  } catch (error) {
    console.error("Unable to connect to the database:", error);
  }
};
run();
```

> Methods of changing the current working directory: `cd ..` or `cd ~`

1. install dependencies: `npm i sequelize mysql2 nodemon`.
2. run the main.js in integrated terminal: `nodemon main.js`.
3. in terminal to create a database, type `mysql -u root -p` will enter mysql terminal. enter mysql password. if term 'mysql' is not recognized shown, go here.
4. then, `create database <db_name>;`, example; `create database sequelize;`. `exit` mysql to quit. tip: `;` put semicolon symbol.
5. add database

```
// ...
  database: "sequelize" // database to use
// ...
```

6. enter mysql terminal, `use sequelize;` to use db. then, `desc users;` will show db properties.

```
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| id        | int          | NO   | PRI | NULL    | auto_increment |
| firstName | varchar(255) | YES  |     | NULL    |                |
| lastName  | varchar(255) | NO   |     | NULL    |                |
| email     | varchar(255) | NO   | UNI | NULL    |                |
| createdAt | datetime     | NO   |     | NULL    |                |
| updatedAt | datetime     | NO   |     | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+
6 rows in set (0.01 sec)
```

7. in main.js, add new user by using `create` method from sequelize. `User.create({ ... })`

> The term 'mysql' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.

Create and findAll

```
const user = await User.create({
      firstName: "Jane",
      lastName: "Smith",
      email: "jane@smith.com"
    })

    const user = await User.findAll({})

    const user = await User.findAll({ where: { id: 12  } })

    const bob = await User.findByPk(1); // by Primary Key

    const bob = await User.findOne({ where: { firstName: "Bob" } })

    const users = await User.findAll({ where: {email: { [Op.like]: "%doe.com" } }
})
     // 8) debug
  console.log(user, "\n", user.toJSON());
  console.log(bob.toJSON()),
```

Update and Delete Data

- to update either directly manipulate property attributes or using `set`. to delete using `destroy`.

1. update: `set`. we grab the data that wants to update

```
// edit jane's lastName and email:
const jane = await User.findByPk(2);

jane.set({
  lastName: "Smith",
  email: "jane@smith.com",
});
await jane.save()

console.log(jane.toJSON())
```

2. delete: `destroy`. we find the entity and then destroy them

```
const john = await User.findByPk(1);
await john.destroy();
```

Relationship

> foreign key always exist on the child. so, parent is the User model. a user has one company and the foreignKey is owner. so, a foreignKey is created inside of Company model called owner.

```
sourceModel.association(targetModel, {foreignKey: "fkName"});
targetModel.association(sourceModel, {foreignKey: "fkName"});
```

main.js

```
// const User = sequelize.define("User", <attributes>);

// create another database called Company
const Company = db.define("Company", {
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true,
  },
  name: {
    type: DataTypes.STRING,
    allowNull: false,
  },
});

// now, have 2 models defined: User and Company
```

```
// One-to-One
User.hasOne(Company, { foreignKey: "owner" }); // This states that a One-to-One relationship exists between User and Company with foreign key defined in Company.
Company.belongsTo(User, { foreignKey: "owner" }); // This states that a One-to-One or One-to-Many relationship exists between Company and User with foreign key defined in Company.

const run = async () => {
  // ...
  // One-to-One
    // ex; Foo Company belongs to Bob with id 1
    const fooCompany = await Company.create({
      name: "Foo Company",
      owner: 1,
    });
    const user = await User.findByPk(1, { include: Company });

    console.log(user.toJSON());

  // One-to-Many (most common)
    // ex;
```

```
  // ...
};
```

Association

Creating associations in sequelize is done by calling one of the `belongsTo` / `hasOne` / `hasMany` / `belongsToMany` functions on a model (the source), and providing another model as the first argument to the function (the target).

> association: https://sequelize.org/docs/v6/core-concepts/assocs/
> https://sequelize.org/api/v6/class/src/associations/base.js~association
> https://dev.to/projectescape/the-comprehensive-sequelize-cheatsheet-3m1m#associations junction table: https://en.m.wikipedia.org/wiki/Associative_entity

The model whose function we will be calling is called the source model, and the model which is passed as a parameter is called the target model.

Sequelize provides **four** types of associations:

1. `hasOne` - adds a foreign key to the target and singular association mixins to the source.
2. `belongsTo` - add a foreign key and singular association mixins to the source.
3. `hasMany` - adds a foreign key to target and plural association mixins to the source.
4. `belongsToMany` - creates an N:M association with a join table and adds plural association mixins to the source. The junction table is created with sourceId and targetId.

```
const A = sequelize.define('A', /* ... */);
const B = sequelize.define('B', /* ... */);

A.hasOne(B, { /* options */ }); // A HasOne B
A.belongsTo(B, { /* options */ }); // A BelongsTo B
A.hasMany(B, { /* options */ }); // A HasMany B
A.belongsToMany(B, { through: 'C', /* options */ }); // A BelongsToMany B through
the junction table C
```

The order matters, for the four cases. In all examples above, A is called the **source** model and B is called the **target** model.

This terminology is important:

1. The `A.hasOne(B)` association means that a One-To-One relationship exists between A and B, with the foreign key being defined in the target model (`B`).
2. The `A.belongsTo(B)` association means that a One-To-One relationship exists between `A` and `B`, with the foreign key being defined in the source model (`A`).
3. The `A.hasMany(B)` association means that a One-To-Many relationship exists between `A` and `B`, with the foreign key being defined in the target model (`B`).

These three calls will cause Sequelize to automatically add foreign keys to the appropriate models (unless they are already present).

4. The `A.belongsToMany(B, { through: 'C' })` association means that a Many-To-Many relationship exists between `A` and `B`, using table `C` as junction table, which will have the foreign keys (`aId` and `bId`, for example). Sequelize will automatically create this model `C` (unless it already exists) and define the appropriate foreign keys on it.

> Note: In the examples above for `belongsToMany`, a string (`'C'`) was passed to the through option. In this case, Sequelize automatically generates a model with this name. However, you can also pass a model directly, if you have already defined it.

In summary:

- To create a **One-To-One** relationship, the `hasOne` and `belongsTo` associations are used together;
- To create a **One-To-Many** relationship, the `hasMany` and `belongsTo` associations are used together;
- To create a **Many-To-Many** relationship, two `belongsToMany` calls are used together.
    - Note: there is also a Super Many-To-Many relationship, which uses six associations at once, and will be discussed in the Advanced Many-to-Many relationships guide.

One-To-One

```
const User = sequelize.define("User", <attributes>);
const Company = sequelize.define("Company", <attributes>);

User.hasOne(Company, { foreignKey: "owner" });
Company.belongsTo(User, { foreignKey: "owner" });
```

1. open terminal, `mysql -u root -p`. then, `desc users` and `desc companies` will have `owner` field in companies db.

```
User.hasOne(Company, { foreignKey: "owner" });
Company.belongsTo(User, { foreignKey: "owner" });

const run = async () => {
  // ...
    // ex; Foo Company belongs to Bob with id 1
    const fooCompany = await Company.create({
      name: "Foo Company",
      owner: 1,
    });
    const user = await User.findByPk(1, { include: Company });

    console.log(user.toJSON());
  // ...
};
```

One-To-Many (most common)

```
const Company = sequelize.define(“Company”, <attributes>);
const Project = sequelize.define(“Project”, <attributes>);
const CompanyProjects = sequelize.define(“CompanyProjects”, {});

Company.belongsToMany(Project, { through: CompanyProjects });
Project.belongsToMany(Company, { through: CompanyProjects });
```

Example, one user can has multiple posts uploaded:

```
// define another Model called Post
const Post = db.define("Post", {
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true,
  },
  description: {
    type: DataTypes.TEXT,
    allowNull: false,
  },
  media: {
    type: DataTypes.STRING,
    allowNull: false,
  }
})

User.hasMany(Post, { foreignKey: “creatorId” });
Post.belongsTo(User, { foreignKey: “creatorId” });

const run = async () => {
  // ...
    // ex; Bob create post
    for (let i = 0; i < 5; i++) {
        const post = await Post.create({
          description: `Hello World ${i}`,
          media: `https://foo.com/${i}.png`,
          creatorId: 1,
      });
      console.log(post);
    }

    const user = await User.findByPk(1, { include: Post });

    console.log(user.toJSON());
  // ...
};
```

Many-to-Many (M:N - tricky)

> Many-to-Many: https://sequelize.org/docs/v6/core-concepts/assocs/#many-to-many-relationships

The concept of a Junction Model is used. This will be an extra model (and extra table in the database) which will have two foreign key columns and will keep track of the associations. The junction table is also sometimes called join table or through table.

Instead of a string, passing a model directly is also supported, and in that case the given model will be used as the junction model (and no model will be created automatically). For example:

```
const Movie = sequelize.define('Movie', { name: DataTypes.STRING });
const Actor = sequelize.define('Actor', { name: DataTypes.STRING });
const ActorMovies = sequelize.define('ActorMovies', {
  MovieId: {
    type: DataTypes.INTEGER,
    references: {
      model: Movie, // 'Movies' would also work
      key: 'id'
    }
  },
  ActorId: {
    type: DataTypes.INTEGER,
    references: {
      model: Actor, // 'Actors' would also work
      key: 'id'
    }
  }
});
Movie.belongsToMany(Actor, { through: ActorMovies });
Actor.belongsToMany(Movie, { through: ActorMovies });
```

```
const Company = sequelize.define(“Company”, <attributes>);
const Project = sequelize.define(“Project”, <attributes>);
const CompanyProjects = sequelize.define(“CompanyProjects”,{});

Company.belongsToMany(Project, { through: CompanyProjects });
Project.belongsToMany(Company, { through: CompanyProjects });
```

Example; one actor can have many movies, and one movie can have many actors. a company can have many projects, and a project can have many companies contractors. M:N use a Junction Table, or a Join table to describe the relationship.

main.js

```
// define another Model called Company
const Company = db.define("Company", {
```

```
    id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true,
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false,
    },
  });

  const Project = db.define("Project", {
    id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true,
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false,
    }
  });

  // define model also as join table or junction table
  const CompanyProjects = db.define("CompanyProject", {});

  // define associations
  Company.belongsToMany(Project, { through: CompanyProjects });
  Project.belongsToMany(Company, { through: CompanyProjects });

  // ex; 2 companies and 3 projects
  const run = async () => {
    // ...
      // project 1 -> company 1, 2
      // project 2 -> company 3
      // project 3 -> company 2, 3

      // first method
      const project1 = await Project.create(
        {
        name: "Project One",
        Companies: [
            {
              name: "Company 1",
            },
            {
              name: "Company 2",
            }
          ]
        },
        {
          include: Company,
        }
      );
```

```
    // second method @ special method, no `include` -> fooInstance.addBar()
    const project2 = await Project.create({ name: "Project Two" });
    const company3 = await Company.create({ name: "Company Three"});
    await project2.addCompany3(company3, { through: CompanyProjects });

    // project 3 -> company 2, 3 with special method
    const project3 = await Project.create({ name: "Project Three" });
    const findCompany2 = await Company.findByPk(9);
    const findCompany3 = await Company.findByPk(10);

    await project3.addCompany2(company2, { through: CompanyProjects });
    await project3.addCompany3(company3, { through: CompanyProjects });

  // debug
    console.log(project1.toJSON());
    console.log(project2.toJSON());
    console.log(project3.toJSON());

    // ..................................................... //

    // looking all projects for one company, from company's perspective (company2
has project1 and project3)
    const findCompany = await Company.findByPk(9, { include: Project });
    console.log(findCompany.toJSON());

    // from project's perspective (project3, id 4 has company2 and company3)
    const findProject = await Project.findByPk(4, { include: Company });
    console.log(findProject.toJSON());
  // ...
};
```

> second method (or Special methods/mixins added to instances): https://sequelize.org/docs/v6/core-concepts/assocs/#foobelongstomanybar--through-baz-

- `Foo.belongsToMany(Bar, { through: Baz })`

The same ones from `Foo.hasMany(Bar)`:

- `fooInstance.getBars()`
- `fooInstance.countBars()`
- `fooInstance.hasBar()`
- `fooInstance.hasBars()`
- `fooInstance.setBars()`
- `fooInstance.addBar()`
- `fooInstance.addBars()`
- `fooInstance.removeBar()`
- `fooInstance.removeBars()`
- `fooInstance.createBar()`

1. start fresh, in mysql terminal `delete from companies;`.

2. `select * from CompanyProjects;`