**Introduction**

[Sequelize](https://sequelize.org/) is a [Node.js](https://nodejs.org/en/)-based Object Relational Mapper that makes it easy to work with [MySQL](https://www.mysql.com/), [MariaDB](https://mariadb.org/), [SQLite](https://www.sqlite.org/index.html), [PostgreSQL](https://www.postgresql.org/) databases, and more. An [*Object Relational Mapper*](https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping) performs functions like handling database records by representing the data as objects. Sequelize has a powerful migration mechanism that can transform existing database schemas into new versions. Overall, Sequelize provides excellent support for database synchronization, eager loading, associations, transactions, and database migrations while reducing development time and preventing SQL injections.

In this tutorial, you will install and configure Sequelize with MySQL on your local development environment. Next, you will use Sequelize to create databases and models, as well as perform the insert, select, and delete operations. Then, you will create Sequelize associations for *one-to-one*, *one-to-many*, and *many-to-many* relationships. Finally, you will create [Sequelize raw queries](https://sequelize.org/docs/v6/core-concepts/raw-queries/) for array and object replacements.

**Prerequisites**

To complete this tutorial, you will need:

* A development environment running Node.js. To set this up, follow the tutorial [How to Install Node.js and Create a Local Development Environment](https://www.digitalocean.com/community/tutorial_series/how-to-install-node-js-and-create-a-local-development-environment)for your operating system.
* MySQL installed in your working environment. Follow the [How To Install MySQL](https://www.digitalocean.com/community/tutorial_collections/how-to-install-mysql) tutorial for your environment, including the steps to create a user.
* Familiarity with JavaScript and MySQL queries, which you can develop with the [How To Code in JavaScript](https://www.digitalocean.com/community/tutorial_series/how-to-code-in-javascript) series and the tutorial [An Introduction to Queries in MySQL](https://www.digitalocean.com/community/tutorials/introduction-to-queries-mysql).
* A basic understanding of relational database concepts, which you can find in the tutorial, [Understanding Relational Databases](https://www.digitalocean.com/community/tutorials/understanding-relational-databases).

This tutorial was tested on Node.js version 14.17.6 and npm version 6.14.15 on macOS Catalina.

**Step 1 — Installing and Configuring Sequelize**

In this step, you will install Sequelize and create the connection to your MySQL database. To do that, first you will create a Node.js application. Then, you will install Sequelize, configure the MySQL database, and develop a simple application.

**Installing Sequelize**

Begin by creating a project folder. In this example, you can use hello-world. Once the folder is created, navigate to the folder using the terminal:

1. mkdir hello-world
2. cd hello-world

Copy

Then, create a sample Node.js application using the following command:

1. npm init

Copy

Next, you will be prompted to answer some set-up questions. Use the following output for your configuration. Press ENTER to use the displayed default value and be sure to set the main entry point as server.js. This creates a project structure that is easy to maintain.

The output will look as follows, which will populate the package.json file:

/hello-world/package.json

my

Copy

Next, create an empty server.js file inside the project folder:

1. touch server.js

Copy

After following the previous steps, your final folder structure will look like this:

hello-world/

├─ package.json

├─ server.js

Now you can install Sequelize with the following command:

1. npm i sequelize@6.11.0

Copy

**Note:** This command installs version 6.11.0. If you need to install the latest version, run npm i sequelize.

After these updates, the package.json file now looks like this:

/hello-world/package.json

{

"name": "hello-world",

"version": "1.0.0",

"description": "",

"main": "server.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"start": "node server.js"

},

"author": "",

"license": "ISC",

"dependencies": {

"sequelize": "^6.11.0"

}

}

Copy

In the dependencies section, you will now see a Sequelize dependency.

You have set up the project and installed Sequelize. Next, you’ll create a sample database to connect to.

**Creating a Sample Database**

As part of the prerequisites, you installed and configured MySQL, which included creating a user. Now you will create an empty database.

To do that, first, you need to log in to your MySQL instance. If you are running remotely, you can use your preferred tool. If you are using a locally running MySQL instance, you can use the following command, replacing your\_usernamewith your MySQL username:

1. mysql -u your\_username -p

Copy

-u is username and the -p option is passed if the account is secured with a password.

The MySQL server will ask for your database password. Type your password and press ENTER.

Once you’re logged in, create a database called hello\_world\_db using the following command:

1. CREATE DATABASE hello\_world\_db;

Copy

To verify whether you have created the database successfully, you can use this command:

1. SHOW DATABASES;

Copy

Your output will be similar to this:

+--------------------+

| Database |

+--------------------+

| hello\_world\_db |

| information\_schema |

| mysql |

| performance\_schema |

| sys |

+--------------------+

After creating the sample database, disconnect from the MySQL server:

1. mysql> QUIT

Copy

Now, you need to install a manual driver for your database of choice. As Sequelize provides ORM features only, it doesn’t include built-in database drivers. Therefore, you’ll need to install drivers according to your preference. To do that, navigate to the project directory using the terminal and install the MySQL driver to the project using the following command:

1. npm install --save mysql2

Copy

In this case, you are using the driver for MySQL.

**Note:** Since this tutorial uses MySQL as the database, you are using a driver for that. Depending on your database, you can manually install the driver like so:

* npm install --save pg pg-hstore # Postgres
* npm install --save mysql2
* npm install --save mariadb
* npm install --save sqlite3
* npm install --save tedious # Microsoft SQL Server

Now that you have a sample database, you can create your first Sequelize application with database connectivity.

**Connecting to the MySQL Database**

In this section, you will connect the Node.js application to the MySQL database using Sequelize.

To connect to the database, open server.js for editing using nano or your preferred code editor:

1. nano server.js

Copy

Here, you will create a database connection in your application using a Sequelize instance. In the new Sequelize() method, pass the MySQL server parameters and database credentials as follows, replacing DATABASE\_USERNAME and DATABASE\_PASSWORD with the credentials of your MySQL user:

/hello-world/server.js

const Sequelize = require("sequelize");

const sequelize = new Sequelize(

'hello\_world\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

Copy

host is where the MySQL server is hosted, so you’ll need to provide a server URL or an IP address. If you are using a locally installed MySQL server, you can replace DATABASE\_HOST with localhost or 127.0.0.1 as the value.

Similarly, if you are using a remote server, make sure to replace database connection values accordingly with the appropriate remote server details.

**Note:** If you are using any other database server software, you can replace the dialect parameter accordingly. `dialect: ‘mysql’, ‘mariadb’, ‘postgres’, ‘mssql’.

Next, call a [promise-based](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) authenticate() method to instantiate a database connection to the application. To do that, add the following code block to the your server.js file:

/hello-world/server.js

...

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

Copy

The authenticate() method is used to connect with the database and tests whether the given credentials are correct. Here, the database connection is open by default and the same connection can be used for all queries. Whenever you need to close the connection, call the sequelize.close() method after this authenticate() call. To learn more about Sequelize, please see their [getting started guide](https://sequelize.org/docs/v6/getting-started/).

Most of the methods provided by Sequelize are asynchronous. That means you can run processes in your application while an asynchronous code block is in its execution time. Also, after the successful asynchronous code block execution, it returns a *promise*, which is the value returned at the end of a process. Therefore, in asynchronous code blocks, you can use then(), catch(), and finally() to return the processed data.

At this point, the server.js file will look like the following:

/hello-world/server.js

const Sequelize = require("sequelize");

const sequelize = new Sequelize(

'hello\_world\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

Copy

Save and close your file.

In the project directory, run the server.js application by running the following command:

1. node server.js

Copy

Your output will look like this:

Output

Connection has been established successfully!

You have created the database connection successfully.

In this step, you installed Sequelize, created a sample database, and used Sequelize to connect with the database. Next, you will work with models in Sequelize.

**Step 2 — Creating a Database Table Using Sequelize**

Now that you have created a sample MySQL database, you can use Sequelize to create a table and populate it with data. In Sequelize, database tables are referred to as *models*. A model is an abstraction that represents a table of the database. Models define several things to Sequelize, such as the name of the table, column details, and data types. In this step, you will create a Sequelize model for book data.

To begin, create a new file called book.model.js in the project directory:

1. nano book.model.js

Copy

Similar to the the previous step, add a Sequelize code for database initiation with a new import for DataTypes at the top of the file:

/hello-world/book.model.js

const { Sequelize, DataTypes } = require("sequelize");

Copy

Sequelize contains many built-in data types. To access those data types, you add an import for DataTypes. This tutorial refers to some frequently used data types, such as STRING, INTEGER, and DATEONLY. To learn more about other supported data types, you can refer to the official [Sequelize documentation](https://sequelize.org/docs/v6/core-concepts/model-basics/" \l "data-types).

Then, include the lines you used previously to create a connection to your MySQL database, updating your MySQL credentials accordingly:

/hello-world/book.model.js

...

const sequelize = new Sequelize(

'hello\_world\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

Copy

Next, you will create a model called books, which includes title, author, release\_date, and subject ID. To do that, use the sequelize.define() method as shown:

/hello-world/book.model.js

...

const Book = sequelize.define("books", {

title: {

type: DataTypes.STRING,

allowNull: false

},

author: {

type: DataTypes.STRING,

allowNull: false

},

release\_date: {

type: DataTypes.DATEONLY,

},

subject: {

type: DataTypes.INTEGER,

}

});

Copy

The sequelize.define() method defines a new model, which represents a table in the database. This code block creates a table called books and stores the book records according to the title, author, release\_date, and subject.

In this code, allowNull shows that the model column value cannot be null. Likewise, if you need to set such a value, you can use defaultValue: "value".

Next, you’ll add the book model to your database. To do that, you’ll use the sync() method as follows:

/hello-world/book.model.js

...

sequelize.sync().then(() => {

console.log('Book table created successfully!');

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

In the sync() method, you’re asking Sequelize to do a few things to the database. With this call, Sequelize will automatically perform an SQL query to the database and create a table, printing the message Book table created successfully!.

As mentioned, the sync() method is a promise-based method, which means it can also perform error handling. In this code block, you’ll check whether the table is created successfully. If not, it will return an error via the catch method and print it on the output.

**Note:** You can manage model synchronization by passing force parameters to force the creation of a new table if it does not exist, or else use an existing one. Here are some examples, which may be helpful to you while working with Sequelize:

* model.sync(): This creates the table if it doesn’t exist already.
* model.sync({ force: true }): This creates the table by dropping it if the same table exists already.

The final code will look like this:

/hello-world/book.model.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'hello\_world\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

const Book = sequelize.define("books", {

title: {

type: DataTypes.STRING,

allowNull: false

},

author: {

type: DataTypes.STRING,

allowNull: false

},

release\_date: {

type: DataTypes.DATEONLY,

},

subject: {

type: DataTypes.INTEGER,

}

});

sequelize.sync().then(() => {

console.log('Book table created successfully!');

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

Save and close your file.

Run your application by using the following command:

1. node book.model.js

Copy

You will get the following output in your command line:

Output

Executing (default): SELECT 1+1 AS result

Executing (default): CREATE TABLE IF NOT EXISTS `books` (`id` INTEGER NOT NULL auto\_increment , `title` VARCHAR(255) NOT NULL, `author` VARCHAR(255) NOT NULL, `release\_date` DATE, `subject` INTEGER, `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB;

Connection has been established successfully.

Executing (default): SHOW INDEX FROM `books`

Book table created successfully!

In the output, you will see the return log contains the message, Book table created successfully!. You can verify this by checking your database to see the new books table created in the hello\_world\_db database.

To verify the creation of the new table, log into your MySQL instance:

1. mysql -u YOUR\_USERNAME -p

Copy

After inputting your password, change into the sample database:

1. USE hello\_world\_db;

Copy

And then run the command to show tables:

1. SHOW TABLES;

Copy

Your output will be similar to this:

+---------------------------+

| Tables\_in\_hello\_world\_db |

+---------------------------+

| books |

+---------------------------+

1 row in set (0.00 sec)

Finally, disconnect from the MySQL server:

1. mysql> QUIT

Copy

You have verified that the book model creation was successful. Using this process, you can create any number of models by following the same procedure.

In this step, you created a model in a database and initiated working with a model using built-in methods. You also used Sequelize-supported data types to define your model. Next, you will work with basic model queries.

**Step 3 — Using Sequelize for Database Queries**

In this step, you will use the Sequelize built-in queries for insertion, selection, selection with conditional clauses, and deletion.

**Inserting a New Record**

In the previous step, you created a book model inside the database. In this section, you’ll insert data into this model.

To get started, copy the contents of book.model.js from the previous step. Create a new file called book.controller.js to handle the query logic. Add the code from book.model.js to book.controller.js.

In book.controller.js, locate the sync() method. In the sync() method, add the following highlighted lines:

/hello-world/book.controller.js

...

sequelize.sync().then(() => {

console.log('Book table created successfully!');

Book.create({

title: "Clean Code",

author: "Robert Cecil Martin",

release\_date: "2021-12-14",

subject: 3

}).then(res => {

console.log(res)

}).catch((error) => {

console.error('Failed to create a new record : ', error);

});

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

Here, you insert a new book record into the books model you’ve already created using the sync() method, which supports adding new records to previously created models. Once the sync() method executes successfully, it runs the then() method. Inside the then() method, you call create() method to insert the new records to the model.

You use the create() method to pass the data you need to add to the database as an object. The highlighted section of code will insert a new entry to your existing books table. In this example, you add Clean Code by Robert Cecil Martin, which has been categorized with the subject ID of 3. You can use the same code, updated with information for other books, to add new records to your database.

Save and close the file.

Run the application using the following command:

1. node book.controller.js

Copy

Your output will look similar to the following:

Output

books {

dataValues:

{ id: 1,

title: 'Clean Code',

author: 'Robert Cecil Martin',

release\_date: '2021-12-14',

subject: 3,

updatedAt: 2021-12-14T10:12:16.644Z,

...

}

You inserted a new record to the model you created in the database. You can continue adding multiple records using the same process.

**Selecting All Records**

In this section, you will select and get all the book records from the database using the findAll() method. To do that, first open book.controller.js and remove the previous Book.create() method. In the sync() method, add the Book.findAll() method as shown:

/hello-world/book.controller.js

...

sequelize.sync().then(() => {

Book.findAll().then(res => {

console.log(res)

}).catch((error) => {

console.error('Failed to retrieve data : ', error);

});

}).catch((error) => {

console.error('Unable to create table : ', error);

});

...

Copy

Save and close the file.

Next, run the application again using the following command:

1. node book.controller.js

Copy

Your output will look similar to the following:

Output

[

books {

dataValues: {

id: 1,

title: 'Clean Code',

author: 'Robert Cecil Martin',

release\_date: '2020-01-01',

subject: 3,

createdAt: 2021-02-22T09:13:55.000Z,

updatedAt: 2021-02-22T09:13:55.000Z

},

\_previousDataValues: {

id: 1,

title: 'Clean Code',

author: 'Robert Cecil Martin',

release\_date: '2020-01-01',

subject: 3,

createdAt: 2021-02-22T09:13:55.000Z,

updatedAt: 2021-02-22T09:13:55.000Z

},

...

]

The output contains all book data as an array object. You successfully used the Sequelize findAll() method to return all book data from the database.

**Selecting with the where Clause**

In this section, you will select values with conditions using the [where](https://sequelize.org/docs/v6/core-concepts/model-querying-basics/#applying-where-clauses) clause. The where clause is used to specify a condition while fetching data. For this tutorial, you will get a book by a specific record ID from the database using the findOne() method.

To do that, open book.controller.js for editing, delete the findAll() method, and add the following lines:

/hello-world/book.controller.js

...

sequelize.sync().then(() => {

Book.findOne({

where: {

id : "1"

}

}).then(res => {

console.log(res)

}).catch((error) => {

console.error('Failed to retrieve data : ', error);

});

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

Here, you select a specific book record from the database using the findOne()method with the where option. In this example, you are retrieving the book data whose id is equal to 1.

Save and close the file.

Next, run the application:

1. node book.controller.js

Copy

Your output will look similar to the following:

Output

books {

dataValues: {

id: 1,

title: 'Clean Code',

author: 'Robert Cecil Martin',

release\_date: '2020-01-01',

subject: 'Science',

createdAt: 2021-02-22T09:13:55.000Z,

updatedAt: 2021-02-22T09:13:55.000Z

},

...

}

You have successfully used where clauses to get data from Sequelize models. You can use the where clause in the database application to capture conditional data.

**Deleting a Record**

To delete a specific record from the database model, you use the destroy()method with the where option. To do that, open book.controller.js, remove the findOne() method, and add the following highlighted lines:

/hello-world/book.controller.js

...

sequelize.sync().then(() => {

Book.destroy({

where: {

id: 2

}

}).then(() => {

console.log("Successfully deleted record.")

}).catch((error) => {

console.error('Failed to delete record : ', error);

});

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

Here, you remove a book record from the database by using the destroy()method with the where option and passing in the id of the book to remove. You are going to remove the book record whose id equals 2.

Save and close the file.

Next, run the application:

1. node book.controller.js

Copy

Your output will look like the following:

Output

Successfully deleted record.

The record has been deleted.

In this step, you experimented with your database model and model querying. You initiated the database, created models, inserted records, retrieved records, retrieved records with conditions using the where clause, and deleted selected records. With this knowledge of Sequelize, you will now create associations in Sequelize. After that, you will be able to define and work with a variety of relationships using Sequelize models.

**Step 4 — Creating Associations Using Sequelize**

In this step, you will use the standard association types that Sequelize supports: *one-to-one*, *one-to-many*, and *many-to-many* associations. You’ll use sample data about students, courses, and grade levels.

Sequelize uses association types based on the following database relationships:

* [*one-to-one* relationship](https://sequelize.org/docs/v6/core-concepts/assocs/#one-to-one-relationships): A one-to-one relationship means a record in one table is associated with exactly one record in another table. In terms of Sequelize, you can use belongsTo() and hasOne() associations to create this type of relationship.
* [*one-to-many* relationship](https://sequelize.org/docs/v6/core-concepts/assocs/#one-to-many-relationships): A one-to-many relationship means a record in one table is associated with multiple records in another table. With Sequelize, you can use hasMany() associations methods to create this type of relationship.
* [*many-to-many* relationship](https://sequelize.org/docs/v6/core-concepts/assocs/#many-to-many-relationships): A many-to-many relationship means multiple records in one table are associated with multiple records in another table. With Sequelize, you can use belongsToMany() associations to create this type of relationship.

Before creating these associations, you will first create a new database called student\_db and add new models and some sample data for students, courses, and grade level.

To create the database, follow the same process in [Step 1 — Installing and Configuring Sequelize](https://www.digitalocean.com/community/tutorials/how-to-use-sequelize-with-node-js-and-mysql#step-1-installing-and-configuring-sequelize) to log into MySQL and create a database called student\_db. Once the new database has been created, log out of MySQL. Next, you’ll start creating database associations.

**Creating a One-to-One Relationship with belongsTo()**

In this section, you will create a one-to-one relationship using Sequelize models. Imagine you want to get one student’s details along with their grade level. Since one student can have only one grade level, this type of association is a one-to-one relationship and you can use the belongsTo() method.

**Note:** There is a difference between belongsTo() and hasOne(). belongsTo()will add the foreignKey on the source table, whereas hasOne() will add it to the target table. In any case, if both relationships are used at the same time, it will work as Sequelize bidirectional one-to-one relationships.

The belongsTo() method allows you to create a one-to-one relationship between two Sequelize models. In this example, you are using the Student and Grade models.

Create a new file called one\_to\_one.js. As you did in the previous section, [Connecting to the MySQL Database](https://www.digitalocean.com/community/tutorials/how-to-use-sequelize-with-node-js-and-mysql#connecting-to-the-mysql-database), include the lines to create a connection to the database and authenticate your MySQL user to the top of the file. Be sure to update the MySQL credentials as needed:

/hello-world/one\_to\_one.js

const { Sequelize, DataTypes } = require("sequelize");

const sequelize = new Sequelize(

'student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

Copy

In this section, you will create three models in the new student\_db database: Student, Grade, and Course. You’ll begin by creating the Student and Grademodels. Later in this step, you’ll create the Courses model.

For the Student model, add the following code block to one\_to\_one.js:

/hello-world/one\_to\_one.js

...

const Student = sequelize.define("students", {

student\_id: {

type: DataTypes.UUID,

defaultValue: DataTypes.UUIDV4,

primaryKey: true,

},

name: {

type: DataTypes.STRING,

allowNull: false

}

});

Copy

This student model contains two columns: student\_id and name.

Next, add a code block for the Grade model:

/hello-world/one\_to\_one.js

...

const Grade = sequelize.define("grades", {

grade: {

type: DataTypes.INTEGER,

allowNull: false

}

});

Copy

The Grade model contains the column grade.

To demonstrate the associations, you’ll need to add sample data to the database. For that, you’ll use the bulk() method. Rather than inserting data into the rows one by one, the bulkCreate() method allows you to insert multiple rows into your database models at once.

So now, import the Grade and Student data to their respective models in the database as shown:

/hello-world/one\_to\_one.js

...

const grade\_data = [{grade : 9}, {grade : 10}, {grade : 11}]

const student\_data = [

{name : "John Baker", gradeId: 2},

{name : "Max Butler", gradeId: 1},

{name : "Ryan Fisher", gradeId: 3},

{name : "Robert Gray", gradeId: 2},

{name : "Sam Lewis", gradeId: 1}

]

sequelize.sync({ force: true }).then(() => {

Grade.bulkCreate(grade\_data, { validate: true }).then(() => {

Student.bulkCreate(student\_data, { validate: true }).then(() => {

…

}).catch((err) => { console.log(err); });

}).catch((err) => { console.log(err); });

}).catch((error) => {

console.error('Unable to create the table : ', error);

});

Copy

Here, you provide sample data and import the data into the Student and Grademodels. With your database, models, and sample data in place, you’re ready to create associations.

In one-to-one.js, add the following line below the student\_data block:

/hello-world/one\_to\_one.js

...

Student.belongsTo(Grade);

Copy

Next, you will need to check whether the association is working properly. To do that, you can retrieve all students’ data with associated grade levels by passing the include parameter inside the findAll() method.

Since you need to get the student grade level, you’ll pass Grade as the model. In the sequelize.sync() method, add the highlighted lines as shown:

/hello-world/one\_to\_one.js

...

sequelize.sync({ force: true }).then(() => {

Grade.bulkCreate(grade\_data, { validate: true }).then(() => {

Student.bulkCreate(student\_data, { validate: true }).then(() => {

Student.findAll({

include: [{

model: Grade

}]

}).then(result => {

console.log(result)

}).catch((error) => {

console.error('Failed to retrieve data : ', error);

});

}).catch((err) => { console.log(err); });

}).catch((err) => { console.log(err); });

}).catch((error) => {

console.error('Unable to create the table : ', error);

});

Copy

The complete code looks like the following:

/hello-world/one\_to\_one.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

const Student = sequelize.define("students", {

student\_id: {

type: DataTypes.UUID,

defaultValue: DataTypes.UUIDV4,

primaryKey: true,

},

name: {

type: DataTypes.STRING,

allowNull: false

}

});

const Grade = sequelize.define("grades", {

grade: {

type: DataTypes.INTEGER,

allowNull: false

}

});

const grade\_data = [{grade : 9}, {grade : 10}, {grade : 11}]

const student\_data = [

{name : "John Baker", gradeId: 2},

{name : "Max Butler", gradeId: 1},

{name : "Ryan Fisher", gradeId: 3},

{name : "Robert Gray", gradeId: 2},

{name : "Sam Lewis", gradeId: 1}

]

// One-To-One association

Student.belongsTo(Grade);

sequelize.sync({ force: true }).then(() => {

Grade.bulkCreate(grade\_data, { validate: true }).then(() => {

Student.bulkCreate(student\_data, { validate: true }).then(() => {

Student.findAll({

include: [{

model: Grade

}]

}).then(result => {

console.log(result)

}).catch((error) => {

console.error('Failed to retrieve data : ', error);

});

}).catch((err) => { console.log(err); });

}).catch((err) => { console.log(err); });

}).catch((error) => {

console.error('Unable to create the table : ', error);

});

Copy

Save and close your file.

Run the file by using the following command:

1. node one\_to\_one.js

Copy

The output will be long, and you will see all students’ data with grade levels. Here is a snippet of the output showing student data:

Output

students {

dataValues:

{ student\_id: '3e786a8f-7f27-4c59-8e9c-a8c606892288',

name: 'Sam Lewis',

createdAt: 2021-12-16T08:49:38.000Z,

updatedAt: 2021-12-16T08:49:38.000Z,

gradeId: 1,

grade: [grades] },

\_previousDataValues:

...

Depending on the command line tools you are using, the output may print as an expanded view or not. If it is an expanded view, it prints the expanded gradeobject as the output.

In this section, you created a one-to-one relationship using the Student.belongsTo(Grade); method call and got the details according to the association you created.

**Creating a One-to-Many Relationship with hasMany()**

In this section, you will create a one-to-many relationship using Sequelize models. Imagine you’d like to get all the students associated with a selected grade level. Since one specific grade level can have multiple students, this is a one-to-many relationship.

To get started, copy the contents of one\_to\_one.js into a new file called one\_to\_many.js. In one\_to\_many.js, remove the lines after the student\_datablock. Your one\_to\_many.js file will look like this:

/hello-world/one\_to\_many.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

const Student = sequelize.define("students", {

student\_id: {

type: DataTypes.UUID,

defaultValue: DataTypes.UUIDV4,

primaryKey: true,

},

name: {

type: DataTypes.STRING,

allowNull: false

}

});

const Grade = sequelize.define("grades", {

grade: {

type: DataTypes.INTEGER,

allowNull: false

}

});

const grade\_data = [ {grade : 9}, {grade : 10}, {grade : 11}]

const student\_data = [

{name : "John Baker", gradeId: 2},

{name : "Max Butler", gradeId: 1},

{name : "Ryan Fisher", gradeId: 3},

{name : "Robert Gray", gradeId: 2},

{name : "Sam Lewis", gradeId: 1}

]

Copy

After the student\_data block, use the hasMany() method to create a new relationship:

/hello-world/one\_to\_many.js

...

Grade.hasMany(Student)

Copy

The hasMany() method allows you to create a one-to-many relationship between two Sequelize models. Here, you are using the Grade and Studentmodels.

Next, add the sequelize.sync() method with the findAll() method below the hasMany() line:

/hello-world/one\_to\_many.js

...

sequelize.sync({ force: true }).then(() => {

Grade.bulkCreate(grade\_data, { validate: true }).then(() => {

Student.bulkCreate(student\_data, { validate: true }).then(() => {

Grade.findAll({

where: {

grade: 9

},

include: [{

model: Student

}]

}).then(result => {

console.dir(result, { depth: 5 });

}).catch((error) => {

console.error('Failed to retrieve data : ', error);

});

}).catch((err) => { console.log(err); });

}).catch((err) => { console.log(err); });

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

Here you are trying to access all the students in a particular grade level—in this case, all the students in grade 9. You also added the Student model in the include option.

Here is the complete code:

/hello-world/one\_to\_many.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

const Student = sequelize.define("students", {

student\_id: {

type: DataTypes.UUID,

defaultValue: DataTypes.UUIDV4,

primaryKey: true,

},

name: {

type: DataTypes.STRING,

allowNull: false

}

});

const Grade = sequelize.define("grades", {

grade: {

type: DataTypes.INTEGER,

allowNull: false

}

});

const grade\_data = [ {grade : 9}, {grade : 10}, {grade : 11}]

const student\_data = [

{name : "John Baker", gradeId: 2},

{name : "Max Butler", gradeId: 1},

{name : "Ryan Fisher", gradeId: 3},

{name : "Robert Gray", gradeId: 2},

{name : "Sam Lewis", gradeId: 1}

]

// One-To-Many relationship

Grade.hasMany(Student);

sequelize.sync({ force: true }).then(() => {

Grade.bulkCreate(grade\_data, { validate: true }).then(() => {

Student.bulkCreate(student\_data, { validate: true }).then(() => {

Grade.findAll({

where: {

grade: 9

},

include: [{

model: Student

}]

}).then(result => {

console.dir(result, { depth: 5 });

}).catch((error) => {

console.error('Failed to retrieve data : ', error);

});

}).catch((err) => { console.log(err); });

}).catch((err) => { console.log(err); });

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

Save and close your file.

Run the file with the following command:

1. node one\_to\_many.js

Copy

The output will look similar to the following. It will be quite long, but all students in grade 9 will be returned as follows:

Output

[ grades {

dataValues:

{ id: 1,

grade: 9,

createdAt: 2021-12-20T05:12:31.000Z,

updatedAt: 2021-12-20T05:12:31.000Z,

students:

[ students {

dataValues:

{ student\_id: '8a648756-4e22-4bc0-8227-f590335f9965',

name: 'Sam Lewis',

createdAt: 2021-12-20T05:12:31.000Z,

updatedAt: 2021-12-20T05:12:31.000Z,

gradeId: 1 },

...

students {

dataValues:

{ student\_id: 'f0304585-91e5-4efc-bdca-501b3dc77ee5',

name: 'Max Butler',

createdAt: 2021-12-20T05:12:31.000Z,

updatedAt: 2021-12-20T05:12:31.000Z,

gradeId: 1 },

...

In this section, you created a one-to-many relationship using the Grade.hasMany(Student); method call. In the output, you retrieved the details according to the association you created.

**Creating Many-to-Many Relationships with belongsToMany()**

In this section, you will create many-to-many relationships using Sequelize models. As an example, imagine a situation where students are enrolled in courses. One student can enroll in many courses and one course can have many students. This is a many-to-many relationship. To implement this using Sequelize, you will use the models Student, Course, and StudentCourse with the belongsToMany() method.

To get started, create a file called many\_to\_many.js and add the database initiation and authentication code blocks as follows. (You can reuse the code blocks from the previous one\_to\_many.js example.) Make sure to update the highlighted database connection values as needed.

/hello-world/many\_to\_many.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

Copy

Next, you’ll create the database models for many-to-many relationships: Student and Course. Then you’ll add some sample data to those models.

/hello-world/many\_to\_many.js

...

const Student = sequelize.define("students", {

student\_id: {

type: DataTypes.UUID,

defaultValue: DataTypes.UUIDV4,

},

name: {

type: DataTypes.STRING,

allowNull: false

}

});

const Course = sequelize.define("courses", {

course\_name: {

type: DataTypes.STRING,

allowNull: false

}

});

const StudentCourse = sequelize.define('StudentCourse', {

id: {

type: DataTypes.INTEGER,

primaryKey: true,

autoIncrement: true,

allowNull: false

}

});

const course\_data = [

{course\_name : "Science"},

{course\_name : "Maths"},

{course\_name : "History"}

]

const student\_data = [

{name : "John Baker", courseId: 2},

{name : "Max Butler", courseId: 1},

{name : "Ryan Fisher", courseId: 3},

{name : "Robert Gray", courseId: 2},

{name : "Sam Lewis", courseId: 1}

]

const student\_course\_data = [

{studentId : 1, courseId: 1},

{studentId : 2, courseId: 1},

{studentId : 2, courseId: 3},

{studentId : 3, courseId: 2},

{studentId : 1, courseId: 2},

]

Copy

Here, you create the Student and Course models and provide some sample data. You also set a courseID, which you will use to retrieve students according to this relationship type.

Finally, you defined a new model called StudentCourse, which manages the relationship data between Student and Course. In this example, studentId 1 is enrolled in courseId 1 and courseId 2.

You have completed the database initiation and added sample data to the database. Next, create many-to-many relationships using the belongsToMany()method as shown:

/hello-world/many\_to\_many.js

...

Course.belongsToMany(Student, { through: 'StudentCourse'})

Student.belongsToMany(Course, { through: 'StudentCourse'})

Copy

Within the belongsToMany() method, you pass the through configuration with the name of the model as the configuration option. In this case, it is StudentCourse. This is the table that manages the many-to-many relationships.

Finally, you can check whether the association is working properly by retrieving all course data with associated students. You’ll do that by passing the includeparameter inside the findAll() method. Add the following lines to many\_to\_many.js:

/hello-world/many\_to\_many.js

...

sequelize.sync({ force: true }).then(() => {

Course.bulkCreate(course\_data, { validate: true }).then(() => {

Student.bulkCreate(student\_data, { validate: true }).then(() => {

StudentCourse.bulkCreate(student\_course\_data, { validate: true }).then(() => {

Course.findAll({

include: {

model: Student,

},

}).then(result => {

console.log(result);

}).catch((error) => {

console.error('Failed to retrieve data : ', error);

});

}).catch((error) => {

console.log(error);

});

}).catch((error) => {

console.log(error);

});

}).catch((error) => {

console.log(error);

});

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

The complete code looks like the following:

/hello-world/many\_to\_many.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

const Student = sequelize.define("students", {

student\_id: {

type: DataTypes.UUID,

defaultValue: DataTypes.UUIDV4,

},

name: {

type: DataTypes.STRING,

allowNull: false

}

});

const Course = sequelize.define("courses", {

course\_name: {

type: DataTypes.STRING,

allowNull: false

}

});

const StudentCourse = sequelize.define('StudentCourse', {

id: {

type: DataTypes.INTEGER,

primaryKey: true,

autoIncrement: true,

allowNull: false

}

});

const course\_data = [

{course\_name : "Science"},

{course\_name : "Maths"},

{course\_name : "History"}

]

const student\_data = [

{name : "John Baker", courseId: 2},

{name : "Max Butler", courseId: 1},

{name : "Ryan Fisher", courseId: 3},

{name : "Robert Gray", courseId: 2},

{name : "Sam Lewis", courseId: 1}

]

const student\_course\_data = [

{studentId : 1, courseId: 1},

{studentId : 2, courseId: 1},

{studentId : 2, courseId: 3},

{studentId : 3, courseId: 2},

{studentId : 1, courseId: 2},

]

Course.belongsToMany(Student, { through: 'StudentCourse'})

Student.belongsToMany(Course, { through: 'StudentCourse'})

sequelize.sync({ force: true }).then(() => {

Course.bulkCreate(course\_data, { validate: true }).then(() => {

Student.bulkCreate(student\_data, { validate: true }).then(() => {

StudentCourse.bulkCreate(student\_course\_data, { validate: true }).then(() => {

Course.findAll({

include: {

model: Student,

},

}).then(result => {

console.log(result);

}).catch((error) => {

console.error('Failed to retrieve data : ', error);

});

}).catch((error) => {

console.log(error);

});

}).catch((error) => {

console.log(error);

});

}).catch((error) => {

console.log(error);

});

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

Save and close the file.

Run the file using the following command:

1. node many\_to\_many.js

Copy

The output will be long, but will look something similar to the following:

Output

[ courses {

dataValues:

{ id: 1,

course\_name: 'Science',

createdAt: 2022-05-11T04:27:37.000Z,

updatedAt: 2022-05-11T04:27:37.000Z,

students: [Array] },

\_previousDataValues:

{ id: 1,

course\_name: 'Science',

createdAt: 2022-05-11T04:27:37.000Z,

updatedAt: 2022-05-11T04:27:37.000Z,

students: [Array] },

\_changed: Set {},

\_options:

{ isNewRecord: false,

\_schema: null,

\_schemaDelimiter: '',

include: [Array],

includeNames: [Array],

includeMap: [Object],

includeValidated: true,

attributes: [Array],

raw: true },

isNewRecord: false,

students: [ [students], [students] ] },

courses {

dataValues:

{ id: 2,

course\_name: 'Maths',

createdAt: 2022-05-11T04:27:37.000Z,

updatedAt: 2022-05-11T04:27:37.000Z,

students: [Array] },

\_previousDataValues:

...

As you can see in this output, the courses with associated students were retrieved. Within the courses block, you will see separate id values that indicate each course. For example, id: 1 is connected to the course\_name: Science for the Science class, whereas id: 2 is the Maths class, and so on.

In the database, you can see the three generated tables with the sample data you inserted.

In this step, you used Sequelize to create one-to-one, one-to-many, and many-to-many associations. Next, you will work with raw queries.

**Step 5 — Working with Raw Queries**

In this step, you will work with [raw queries](https://sequelize.org/docs/v6/core-concepts/raw-queries/) in Sequelize. In previous steps, you used Sequelize built-in methods, such as insert() and findAll(), to handle data insertion and selection from the database. You may have noticed that those methods follow a specific pattern for writing a query. However, with the use of raw queries, you don’t need to worry about Sequelize built-in methods and patterns. Using your knowledge of SQL queries, you can conduct a range of queries in Sequelize from simple to more advanced.

Here is an example of raw queries that perform the action of selecting all values from a particular table, deleting the selected values according to the condition, and updating the table with the given values.

SELECT \* FROM table\_name;

DELETE FROM table\_name WHERE condition;

UPDATE table\_name SET y = 42 WHERE x = 12;

In Sequelize, raw queries can be used with primarily two methodologies: array replacement and object replacement. When you are passing values to the SQL query, you can use either an array or an object to do that replacement.

Before writing a raw query, you will first need to supply student data in a sample database. Following the previous section, [Creating a Sample Database](https://www.digitalocean.com/community/tutorials/how-to-use-sequelize-with-node-js-and-mysql#creating-a-sample-database), log in to MySQL, create a database called sample\_student\_db, and log out of MySQL.

Next, you’ll add some raw data to start working with raw queries. Create a new file called add\_student\_records.js and add the following code blocks, which contain the previously discussed Sequelize methods of authenticate(), sync(), and bulkCreate().

/hello-world/add\_student\_records.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'sample\_student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

const Student = sequelize.define("students", {

student\_id: {

type: DataTypes.UUID,

defaultValue: DataTypes.UUIDV4,

primaryKey: true,

},

name: {

type: DataTypes.STRING,

allowNull: false

}

});

const student\_data = [

{name : "John Baker"},

{name : "Max Butler"},

{name : "Ryan Fisher"},

{name : "Robert Gray"},

{name : "Sam Lewis"}

]

sequelize.sync({ force: true }).then(() => {

Student.bulkCreate(student\_data, { validate: true }).then((result) => {

console.log(result);

}).catch((error) => {

console.log(error);

});

}).catch((error) => {

console.error('Unable to create table : ', error);

});

Copy

Here, you initiate the database connection, create the model, and insert a few student records inside the new database.

Save and close the file.

Next, run this script using the following command:

1. node add\_student\_records.js

Copy

The output will be something similar to the following. It will be quite long, but all the student records which you inserted will be returned as follows. Note that since the student\_id is an auto-generated UUID ([Universally Unique Identifiers](https://en.wikipedia.org/wiki/Universally_unique_identifier)) value, it will be different depending on the user.

Output

Executing (default): SELECT 1+1 AS result

Executing (default): DROP TABLE IF EXISTS `students`;

Connection has been established successfully.

Executing (default): DROP TABLE IF EXISTS `students`;

Executing (default): CREATE TABLE IF NOT EXISTS `students` (`student\_id` CHAR(36) BINARY , `name` VARCHAR(255) NOT NULL, `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, PRIMARY KEY (`student\_id`)) ENGINE=InnoDB;

Executing (default): SHOW INDEX FROM `students`

Executing (default): INSERT INTO `students` (`student\_id`,`name`,`createdAt`,`updatedAt`) VALUES ('45d1f26c-ba76-431f-ac5f-f41282351710','John Baker','2022-06-03 07:27:49','2022-06-03 07:27:49'),('1cb4e34d-bfcf-4a97-9624-e400b9a1a5f2','Max Butler','2022-06-03 07:27:49','2022-06-03 07:27:49'),('954c576b-ba1c-4dbc-a5c6-8eaf22bbbb04','Ryan Fisher','2022-06-03 07:27:49','2022-06-03 07:27:49'),('e0f15cd3-0025-4032-bfe8-774e38e14c5f','Robert Gray','2022-06-03 07:27:49','2022-06-03 07:27:49'),('826a0ec9-edd0-443f-bb12-068235806659','Sam Lewis','2022-06-03 07:27:49','2022-06-03 07:27:49');

[

students {

dataValues: {

student\_id: '45d1f26c-ba76-431f-ac5f-f41282351710'`,

name: 'John Baker',

createdAt: 2022-06-03T07:27:49.453Z,

updatedAt: 2022-06-03T07:27:49.453Z

},

\_previousDataValues: {

name: 'John Baker',

student\_id: '45d1f26c-ba76-431f-ac5f-f41282351710',

createdAt: 2022-06-03T07:27:49.453Z,

updatedAt: 2022-06-03T07:27:49.453Z

},

…

In the next section, you will apply raw queries using one of the student\_idoutputs in the code block above. Copy it down so that you have it for the next sections, where you will use the query() method for array and object replacements.

**Array Replacement**

In this section, you’ll use the query() method for an array replacement. With this method, Sequelize can execute raw or already prepared SQL queries.

To get started, copy the contents of the server.js file from [Step 1](https://www.digitalocean.com/community/tutorials/how-to-use-sequelize-with-node-js-and-mysql#connecting-to-the-mysql-database), as that includes the initiate Sequelize() method and database initiation. Paste the contents into a new file called array\_raw\_query.js. Update the database name to sample\_student\_db:

/hello-world/array\_raw\_query.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'sample\_student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

Copy

At the end of the file, add the following code block for an array replacement, making sure to replace REPLACE\_STUDENT\_ID with the student\_id value that you copied in the previous section.

/hello-world/array\_raw\_query.js

...

sequelize.query(

'SELECT \* FROM students WHERE student\_id = ?',

{

replacements: ['REPLACE\_STUDENT\_ID'],

type: sequelize.QueryTypes.SELECT

}

).then(result => {

console.log(result);

}).catch((error) => {

console.error('Failed to insert data : ', error);

});

Copy

For array replacement, you pass the query() method with the SQL query and the configuration object. It contains the replacements value and type. To replacements, you pass data as an array and catch those values using the question mark (?) symbol.

Next, since you need to get data about a specific student, the student\_id is passed as the second parameter. After that, you pass the type: sequelize.QueryTypes.SELECT key-value pair, which you can use to select data from the database.

There are some other types as well, such as QueryTypes.UPDATE and QueryTypes.DELETE. Depending on the requirement, you can select the type that suits your purpose.

The following shows the full code block. Here you connect to the database and retrieve the selected student data using a raw query.

/hello-world/array\_raw\_query.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'sample\_student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

sequelize.query(

'SELECT \* FROM students WHERE student\_id = ?',

{

replacements: ['REPLACE\_STUDENT\_ID'],

type: sequelize.QueryTypes.SELECT

}

).then(result => {

console.log(result);

}).catch((error) => {

console.error('Failed to insert data : ', error);

});

Copy

Save and close your file.

Next, you can run this script using the following command:

1. node array\_raw\_query.js

Copy

You will see output similar to the following:

Output

Connection has been established successfully.

[ { student\_id: 'STUDENT\_ID\_YOU\_RETRIEVED',

name: 'Robert Gray',

createdAt: 2022-05-06T13:14:50.000Z,

updatedAt: 2022-05-06T13:14:50.000Z } ]

Due to the selected student\_id, your output values may differ.

**Object Replacement**

On the surface, object replacement is similar to array replacement, but the pattern of passing data to the raw query is different. In the replacement option, you pass data as an object, and in the query option, you use values like :key.

To get started, create a new file called object\_raw\_query.js and paste the complete code blocks from the server.js file, updating the database to sample\_student\_db.

/hello-world/array\_raw\_query.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'sample\_student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

Copy

Then, add the following code block to the end of the new object\_raw\_query.jsfile:

/hello-world/object\_raw\_query.js

...

sequelize.query(

'SELECT \* FROM students WHERE student\_id = :id',

{

replacements: { id: 'REPLACE\_STUDENT\_ID' },

type: sequelize.QueryTypes.SELECT

}

).then(result => {

console.log(result);

}).catch((error) => {

console.error('Failed to insert data : ', error);

});

Copy

Here, you get selected student data using the object replacement method. You create a replacement object, setting the id as the student information you wish to retrieve: { id: 'REPLACE\_STUDENT\_ID' }.

In the query(), you indicate: 'SELECT \* FROM students WHERE student\_id = :id'. Using the query() method, you pass the replacement value as an object, which is why this method is known as object replacement.

Here is the complete code:

/hello-world/object\_raw\_query.js

const {Sequelize, DataTypes} = require("sequelize");

const sequelize = new Sequelize(

'sample\_student\_db',

'DATABASE\_USERNAME',

'DATABASE\_PASSWORD',

{

host: 'DATABASE\_HOST',

dialect: 'mysql'

}

);

sequelize.authenticate().then(() => {

console.log('Connection has been established successfully.');

}).catch((error) => {

console.error('Unable to connect to the database: ', error);

});

sequelize.query(

'SELECT \* FROM students WHERE student\_id = :id',

{

replacements: { id: 'REPLACE\_STUDENT\_ID' },

type: sequelize.QueryTypes.SELECT

}

).then(result => {

console.log(result);

}).catch((error) => {

console.error('Failed to insert data : ', error);

});

Copy

Save and close the file.

Next, run this script using the following command:

1. node object\_raw\_query.js

Copy

The output will look similar to the following:

Output

Connection has been established successfully.

[ { student\_id: 'STUDENT\_ID\_YOU\_RETRIEVED',

name: 'Robert Gray',

createdAt: 2022-05-06T13:14:50.000Z,

updatedAt: 2022-05-06T13:14:50.000Z } ]

Due to the selected student\_id, your output values may differ.

In this step, you worked with Sequelize raw queries using two different methodologies: array replacement and object replacement.