Notes – Database

By this point, you know how to build a basic web application with Express.js, routing and templating. This unit is about taking the application and connecting it to a database. you learn to connect and read from SQL and NoSQL databases, including MongoDB, a popular database for Node.js.

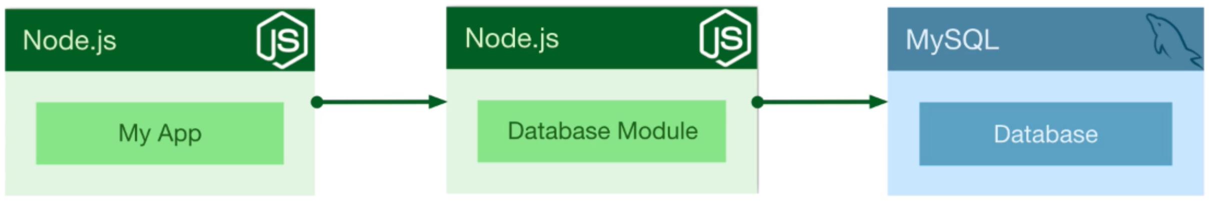
With Node.js, you can work with practically any common database, such as MySQL, and PostgreSQL. MongoDB, however, offers a unique style of data storage that resembles JSON—a JavaScript-friendly format that may make working with databases easier for you as you delve into saving data with Node.js for the first time. We already reviewed access to SQL databases. We will review that area again and will focus exploring NoSQL databases. If you are comfortable with the topics in SQL databases, you can skip the first part and go directly to NoSQL databases.

Let’s have a look at how Node talks to a database. Using the Node module system, we can install modules that will handle all the heavy lifting of talking to various different types of databases so your app can interface with a database module, but then talk to the database directly for you.

Later on, you will learn about technology called object-relational mappers or ORMs. ORM gives you a common language for talking to different SQL databases, because even though they're all SQL, they all vary in what data types they support and what commands they support. With ORMs you can communicate with data bases in a common language that is not SQL-based, and ORM takes care of communicating with other databases. This way, you can talk to different types of databases without changing much.

WORKING WITH DATABASES USING NODE MODULES

With Modules, the data processing look like the following diagram:



Let’s work with SQLite database as it doesn’t require servers and easier to understand.

The module that is required to communicate with a SQLite is called **sqlite3**. Just using sqlite3 because it is a file based SQL system, and doesn’t require a server, so there's nothing for you to install or configure to get these examples going. Here is the instruction for an example:

- From your working directory, let's create **trysqlite** directory and
- **npm init**
- **npm install sqlite3**
- Create **seed.js**

SQLite is not particularly a powerful database and it's not designed for multi-user use, However.

Let's type the following code, to create a "favourite cheese database". We use **serialize ( )** method to put together the data. The structure is as below:

seed.js

```
const sqlite3 = require('sqlite3');
const sqlite3 = require('sqlite3');
const db = new sqlite3.Database('example.sqlite3');

db.serialize(() => {
    db.run('DROP TABLE IF EXISTS authors');
    db.run("CREATE TABLE authors (name TEXT)");

    db.run("INSERT INTO authors VALUES ('Mark Twain')");
    db.run("INSERT INTO authors VALUES ('William Shakespeare')");
    db.run("INSERT INTO authors VALUES ('Agatha Christie')");
    db.run("INSERT INTO authors VALUES ('Barbara Cartland')");

    // You can also use prepared statements, which are more efficient
    // const stmt = db.prepare("INSERT INTO Cheeses VALUES (?)");
    // stmt.run("Mark Twain");
    // stmt.run("William Shakespeare");
    // stmt.run("Agatha Christie");
    // stmt.run("Barbara Cartland");

    // stmt.finalize();

    db.each('SELECT rowid AS id, name FROM authors', (err, row) => {
        console.log(`${row.id}: ${row.name}`);
    });

});

db.close;
```

run:

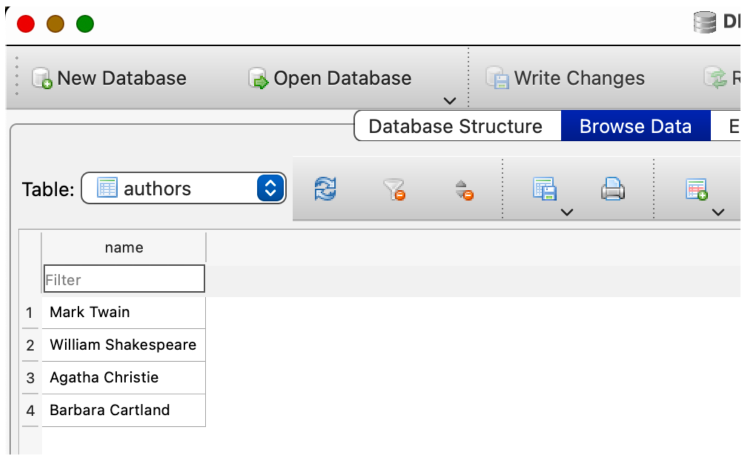
> node seed

It creates **example.sqlite**

If you want to remove it and run it, type: **rm example.sqlite**

This was for creating a database. In order to read the data in the database, you can use the Sqlite viewer that you can download from:

https://sqlitebrowser.org/dl/



	name
1	Mark Twain
2	William Shakespeare
3	Agatha Christie
4	Barbara Cartland

Or write a program to see the results. See the following code:

From index, this is what you need to have:

Index.js

```
const sqlite3 = require('sqlite3');
const db = new sqlite3.Database('example.sqlite');

    db.each("SELECT rowid AS id, name FROM authors", (err,row) => {
        console.log(`${row.id}: ${row.name}`);
    });
```

This just ouputs the cheese table into console.

> **node index**

Let’s try another example below with more functionalities.

Similarly you can apply CRUD operations using SQL statements. An example is provided in the following and in the folder **trycrud**. For more information, check the codes and evaluate them.

Applying CRUD

Let’s work with another example:
Let’s use the following table

Users		
name	Name of user	String
email	Email address	String
fav_pizza	Preferred pizza for when we order out	String
space_invaders	High Score on the office Space Invaders machine	Integer

Create a new directory, called “**d:/crud**”

> npm install sqlite3

Create **migrate.js**

Here is the code

```
const sqlite3 = require('sqlite3');
const db = new sqlite3.Database('users.sqlite');

db.serialize ( () => {
    db.run("DROP TABLE IF EXISTS users");
    db.run("CREATE TABLE users (name TEXT, email TEXT, fav_pizza TEXT, space_invaders INT)");

    //populate
    const stmt = db.prepare("INSERT INTO users VALUES (?,?,?,?)");

    stmt.run("PJ", "pj@company.org", "Pepperoni", 826488);
    stmt.run("Trish", "trish@company.org", "Spicy", 826478);
    stmt.run("Paddy", "paddy@company.org", "Ham", 826588);
    stmt.run("Bob", "bob@company.org", "Onion", 836488);
    stmt.run("Alice", "alice@company.org", "Everything", 826428);

    stmt.finalize();

});

db.close();
```

migrate.js

create **index.js**

From index, this is what you need to have:

```
const sqlite3 = require('sqlite3');
const db = new sqlite3.Database('users.sqlite');

    db.each("SELECT rowid AS id, * FROM users", (err,row) => {
        console.log(`${row.name} (${row.email}): pizza - ${row.fav_pizza}, Space Invaders - ${
row.space_invaders} `);
    });

db.close();
```

Index.js

Now let’s
> node migrate
> node index

For UPDATE … check the following in **update.js**

```
const sqlite3 = require("sqlite3");
const db = new sqlite3.Database("users.sqlite");

db.run(
  "UPDATE users SET space_invaders = 999999999 WHERE name = 'PJ'",
  (err, row) => {
    if (err) {
      console.log(`Error ${err} `);
    } else {
      console.log("Row updated");
    }
  }
);

db.close();
```

updagte.js

> node update

> node index

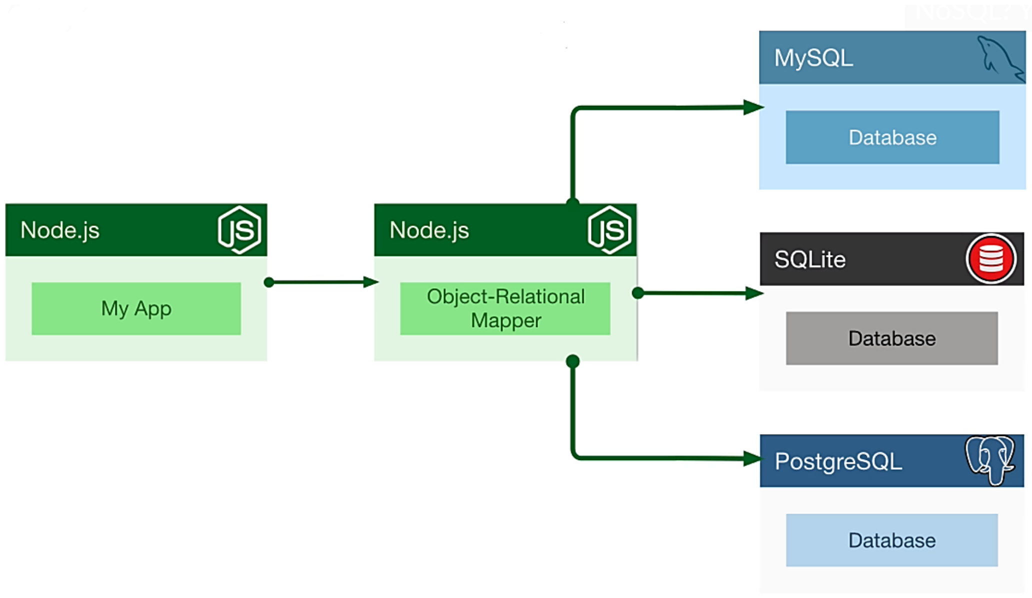
For DELETE is the same except that we change UPDATE with DELETE.

Please try it and make sure your code works!

OBJECT-RELATIONAL MAPPING (ORM)

So far you learned the traditional way of talking to a database. Your application talks to a dedicated database module, which knows how to handle the database directly, so you don't need to worry about it; you just throw SQL statements at it. In the object-relational mapping world, you have a similar setup, except you don't throw SQL statements at it.

It knows how to talk to several different databases, but takes the SQL input and output and turns it into a much more object-orientated architecture. It is an interesting approach as you can prototype code on SQLite and then, later on, port it to something like MySQL or SQL Server.



One of the most popular ORMs is "**Sequelize**".

Create a directory called **trysequelize**

> npm init

> npm install sequelize sqlite3

Let's create and start from migration.js

requiring Sequelize. Note the capital S because this is the class we'll be making instances from.

Create our connection, and because we're using SQLite, we've got to tell it which file to use—that's storage.

```
const Sequelize = require('sequelize');

// initialize connection
const sequelize = new Sequelize ({
    dialect: 'sqlite',
    storage: 'users.sqlite',
});
```

Once we have our connection, we can define our model, which is called User. Using the define method, we give it a table name, and then the fields within that table. We've got some constants here that we can use to create our name, email, fav_pizza, and space_invaders. Quite similar to what we did before.

```
const user = sequelize.define('users' , {
    name: Sequelize.STRING,
    email: Sequelize.STRING,
    fav_pizza: Sequelize.STRING,
    space_invaders: Sequelize.INTEGER
});
```

Now, we have this object, we can perform operations on it.

Let's use **async** and **await** by wrapping them in an instant function (a defined anonymous function that runs immediately). **Sync ()** method makes sure that the model and the table on the actual database are in sync with each other. What this does, in effect, is create the table. Once we've done that, there is a very useful method called bulkCreate that allows to enter several records at once just by wrapping them in an array. Notice that each record is an object with the field names and the values one after the other. Quite simple. By using **await**, we know when we get to the end of this code that all our records have been created.

```
(async() => {
    await User.sync( {force: true});

    await User.bulkCreate([
       { name: 'PJ', email: 'pj@company.org', fav_pizza: 'Pepperoni', space_invaders: 826488},
       { name: 'Trish', email: 'trish@company.org', fav_pizza: 'Spicy Veg', space_invaders: 826388},
       { name: 'Paddy', email: 'paddy@company.org', fav_pizza: 'Ham', space_invaders: 826489},
       { name: 'Bob', email: 'bob@company.org', fav_pizza: 'Onion', space_invaders: 826228},
       { name: 'Alice', email: 'alice@company.org', fav_pizza: 'Everything', space_invaders: 8244488},
    ])
})();
```

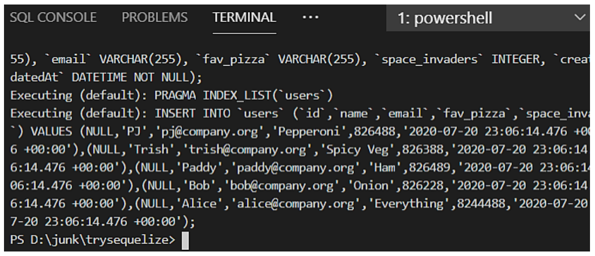
Here is the whole code together for **migration.js**

```

const Sequelize = require('sequelize');

// Initialise connection
const sequelize = new Sequelize({
    dialect: 'sqlite',
    storage: 'users.sqlite',
//    operatorsAliases: false
});

// Define our Model
const User = sequelize.define('users', {
    name: Sequelize.STRING,
    email: Sequelize.STRING,
    fav_pizza: Sequelize.STRING,
    space_invaders: Sequelize.INTEGER
});

// Create the table (force drops any existing table)
(async () => {

    await User.sync( {force: true} );

    await User.bulkCreate([
       { name: 'PJ', email: 'pj@company.org', fav_pizza: 'Pepperoni', space_invaders: 826488 },
       { name: 'Trish', email: 'trish@company.org', fav_pizza: 'Spicy Veg', space_invaders: 674588 },
       { name: 'Paddy', email: 'paddy@company.org', fav_pizza: 'Ham', space_invaders: 998988 },
       { name: 'Bob', email: 'bob@company.org', fav_pizza: 'Onion', space_invaders: 6577 },
       { name: 'Alice', email: 'alice@company.org', fav_pizza: 'Everything', space_invaders: 929848 }
    ]);

})();
```

Migration.js

> node migration

The output generates SQL statement.



```
55), `email` VARCHAR(255), `fav_pizza` VARCHAR(255), `space_invaders` INTEGER, `creat
datedAt` DATETIME NOT NULL);
Executing (default): PRAGMA INDEX_LIST(`users`)
Executing (default): INSERT INTO `users` (`id`,`name`,`email`,`fav_pizza`,`space_inva
`) VALUES (NULL,'PJ','pj@company.org','Pepperoni',826488,'2020-07-20 23:06:14.476 +0
6 +00:00'),(NULL,'Trish','trish@company.org','Spicy Veg',826388,'2020-07-20 23:06:14
6:14.476 +00:00'),(NULL,'Paddy','paddy@company.org','Ham',826489,'2020-07-20 23:06:1
06:14.476 +00:00'),(NULL,'Bob','bob@company.org','Onion',826228,'2020-07-20 23:06:14
6:14.476 +00:00'),(NULL,'Alice','alice@company.org','Everything',8244488,'2020-07-20
7-20 23:06:14.476 +00:00');
PS D:\junk\trysequelize>
```

The users.sqlite has been created!

The next file to create is **index.js**.

It starts the same as our migration.js, requiring sequelize, and then creating our connection. And, in fact, we also need to define the model, so we can act upon it. So, that code needs to be repeated as well. In index.js, we're going to get the content of the table and output it. And here's how you do it.

There's a really useful method called **findAll ( )** that we can perform on our model, and that has a callback, which allows us to operate on each row as it comes in. So, when all our rows arrive, we can have a **forEach** loop around it, which outputs our string, just like in the previous module.

We can change **findAll** to use await because it returns a promise. So, we can wrap that in an async function, just like before, and now we've got something a bit neater. In update.js, you can cut and paste in exactly what we've done before—the requirement, the connection, and then the definition of the model. So, it needs that every time so it knows how to act.

```
const Sequelize = require("sequelize");

//initialize connection
const sequelize = new Sequelize({
  dialect: "sqlite",
  storage: "users.sqlite",
//   operatorsAliases: false,
//   logging:false,
});

const User = sequelize.define("users", {
  name: Sequelize.STRING,
  email: Sequelize.STRING,
  fav_pizza: Sequelize.STRING,
  space_invaders: Sequelize.INTEGER,
});


// try this first as it is a bit easier then we add async function
// User.findAll().then((rows) => {
//   rows.forEach((row) => {
//     console.log(
//       `${row.name} (${row.email}): pizza - ${row.fav_pizza}, Space Invaders - ${row.space_i
nvaders} `
//     );
//   });
// });

// a better way is
(async () => {
    const rows = await User.findAll().then((rows) => {
      rows.forEach((row) => {
        console.log(
        `${row.name} (${row.email}): pizza - ${row.fav_pizza}, Space Invaders - ${row.space_invaders}`
        );
      });
    });
  })();
```

Index.js

For update and delete, see the code below:

```
const Sequelize = require("sequelize");

//initialize connection
const sequelize = new Sequelize({
  dialect: "sqlite",
  storage: "users.sqlite",
//   operatorsAliases: false,
//   logging:false,
});

const User = sequelize.define("users", {
  name: Sequelize.STRING,
  email: Sequelize.STRING,
  fav_pizza: Sequelize.STRING,
  space_invaders: Sequelize.INTEGER,
});
```

```
(async () => {
  await User.update({
    space_invaders: 99999999
  }, {
      where: {
          name: 'PJ'
      }
  });
})();
```

Update.js

```
const Sequelize = require("sequelize");

//initialize connection
const sequelize = new Sequelize({
  dialect: "sqlite",
  storage: "users.sqlite",
//    operatorsAliases: false,
//    logging:false,

});

const User = sequelize.define("users", {
  name: Sequelize.STRING,
  email: Sequelize.STRING,
  fav_pizza: Sequelize.STRING,
  space_invaders: Sequelize.INTEGER,
});

(async () => {
  await User.destroy({
    where: {
      name: "Bob",
    },
  });
})();
```

delete.js

try the CRUD, above.

Implementing an API using a SQL databse (sqlite)

Let’s create a REST API that uses a database instead of a local file. Similar to the concepts of the REST API Introduction, a similar API is built that uses a Sqlite dataset. You can find the files in the **Complete Simple API Sqlite** folder.