Express.js Web Application Tutorial

Part 3: Connecting to a Database

In this tutorial, you will learn how to create Nodejs application with database. We will start with SQLite.

What is SQLite?

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. The code for SQLite is free for use for any purpose, commercial or private. SQLite is the most widely deployed database with applications in many domains, including several high-profile projects. It is most widely used for Mobile Apps Development.

Sqlite3 Module

SQLite3 is a NodeJS module that implements the API for SQLite database engine. To understand how the sqlite3 module works, we will follow the following steps in sequence:

1- **Connecting to an SQLite3 database** – shows you how to connect to an in-memory database or a file-based database.
2- **Inserting data into a table** – shows you how to insert one or more rows into a table.
3- **Querying data from tables** – introduces you to various methods for querying data from tables.
4- **Controlling the execution flow of statements** – explains the steps of executing multiple SQL statements in serialized mode or in parallel mode.
5- **Updating data** – covers the steps of updating data in a table.
6- **Deleting data from a table** – show you how to delete data from a table.

1. Connecting to an SQLite3 database

To install SQLite3 module into your application, change directory to where Nodejs code is stored (See Part 1 of this tutorial) and use NPM to install the EJS module as follows:

```
D:\> cd www\dbapp1

D:\www\dbapp1> npm install sqlite3
```

After installing the sqlite3 module, you are ready to connect to a SQLite database from a Node.js application.

To connect to an SQLite database, you need to:

- First, import the sqlite3 module
- Second, call the Database() function of the sqlite3 module and pass the database information such as database file, opening mode, and a callback function.

Connecting to the in-memory database:

To open a database connection to an **in-memory** database, you use the following steps.

First, import the sqlite3 module:

```
const sqlite3 = require('sqlite3').verbose();
```

Notice that the execution mode is set to **verbose** to produce long stack traces.

Second, create a Database object:

```
let db = new sqlite3.Database(':memory:');
```

The sqlite3.Database() returns a Database object and opens the database connection automatically.

The sqlite3.Database() accepts a callback function that will be called when the database opened successfully or when an error occurred.

The callback function has the error object as the first parameter. If an error occurred, the error object is not null, otherwise, it is null.

If you don't provide the callback function and an error occurred during opening the database, an error event will be emitted. In case the database is opened successfully, the open event is emitted regardless of whether a callback is provided or not.

So you now can open an SQLite database and provide the detailed information if an error occurred as follows:

```
let db = new sqlite3.Database(':memory:', (err) => {

  if (err) {

    return console.error(err.message);

  }

  console.log('Connected to the in-memory SQlite database.');

});
```

It is a good practice to close a database connection when you are done with it. To close a database connection, you call the close() method of the Database object as follows:

```
db.close();
```

Similar to the Database(), the close() method also accepts a callback that indicates whether an error occurred during closing the database connection.

Now this is a full program for connecting to SQLite database.

connect.js

```
const sqlite3 = require('sqlite3').verbose();

// open database in memory

let db = new sqlite3.Database(':memory:', (err) => {

  if (err) { return console.error(err.message);   }

  console.log('Connected to the in-memory SQlite database.');

});

// close the database connection

db.close((err) => {

  if (err) {  return console.error(err.message);   }

  console.log('Close the database connection.');

});
```

Let’s run the program to see how it works:

```
D:\www\dbapp1> node connect.js

Connected to the in-memory SQlite database.

Close the database connection.
```

Connecting to a disk file database:

To connect to a disk file database, instead of passing the ':memory:' string, you pass the path to the database file.

For example, to connect to the mydb database file stored in the root folder, you use the following statement:

```
let db = new sqlite3.Database('./mydb.db', (err) => {

  if (err) {   console.error(err.message);  }

  console.log('Connected to the mydb database.');

});
```

There are three opening modes:

1- sqlite3.OPEN_READONLY: open the database for read-only.
2- sqlite3.OPEN_READWRITE : open the database for reading and writting.
3- sqlite3.OPEN_CREATE: open the database, if the database does not exist, create a new database.

The sqlite3.Database() accepts one or more mode as the second argument. By default, it uses the OPEN_READWRITE | OPEN_CREATE mode. It means that if the database does not exist, the new database will be created and is ready for read and write.

Here is full code to connect to in-disk database:

Connect2.js

```
const sqlite3 = require('sqlite3').verbose();
// open database in memory
let db = new sqlite3.Database('./mydb.db', (err) => {
  if (err) { return console.error(err.message);   }
  console.log('Connected to mydb SQlite database.');
});
// close the database connection
db.close((err) => {
  if (err) {  return console.error(err.message);   }
  console.log('Close the database connection.');
});
```

Let’s run the program to see how it works:

```
D:\www\dbapp1> node connect2.js
Connected to mydb SQlite database.
Close the database connection.
```

2. Inserting data into database Tables

To insert data into an SQLite table from a Node.js application, you follow these steps:

1- Open a database connection.
2- Execute an INSERT statement.
3- Close the database connection.

For the demonstration, we will use a database named mydb.db in the root folder.

Note: When you open a database connection in the default mode, the database is created if it does not exist.

In the following code we will create one table named ‘langs’ and insert a row of data into the table:

Insert.js

```
const sqlite3 = require('sqlite3').verbose();

let db = new sqlite3.Database('./mydb.db');

let sql1 = 'CREATE TABLE IF NOT EXISTS langs(name text)';

db.run(sql1);

 // insert one row into the langs table

db.run(`INSERT INTO langs(name) VALUES(?)`, ['Nodejs'],
(err)=> {

    if (err) {  return console.log(err.message);    }

    // get the last insert id

    console.log(`A row has been inserted with rowid {this.lastID}`);

  });db.close();
```

Insert multiple rows into a table at a time:

To insert multiple rows at a time into a table, you use the following form of the INSERT statement:

INSERT INTO table_name(column_name) VALUES(value_1), (value_2), (value_3),...

The following insert-many.js program illustrates how to insert multiple rows into the langs table:

Insert_many.js

```
const sqlite3 = require('sqlite3').verbose();

let db = new sqlite3.Database('./mydb.db');

db.run('CREATE TABLE IF NOT EXISTS langs(name text)');

 // insert one row into the langs table

db.run(`INSERT INTO langs(name) VALUES (?),(?),(?),(?)` , ['Nodejs',
'Java', 'C++','Python'], (err)=> {

    if (err) {  return console.log(err.message);    }

    console.log(`4 row has been inserted with rowid ${this.lastID}`);

  });

// display data from database

let sql = 'SELECT * FROM langs';
  db.all(sql, [], (err, rows) => {
    if (err) {  throw err;  }
    rows.forEach((row) => {
      console.log(row.name);
    });
  }); db.close();
```

3. Querying data from database Tables

To query data in SQLite database from a Node.js application, you use these steps:

1- Open a database connection.
2- Execute a SELECT statement and process the result set.
3- Close the database connection.

The sqlite3 module provides you with some methods for querying data such as all(), each() and get().

Select1.js

```
const sqlite3 = require('sqlite3').verbose();

let db = new sqlite3.Database('./mydb.db' , (err)=>{
    if(err){
        return console.log(err);
    }
    console.log('Connected to mydb database!');
});

// display data from database

let sql = 'SELECT * FROM langs';
  db.all(sql, [], (err, rows) => {
    if (err) {  throw err;  }
    rows.forEach((row) => {
      console.log(row.name);
    });
  }); db.close();
```

The following code will select DITINCT names and order them:

Select1.js

```
const sqlite3 = require('sqlite3').verbose();

let db = new sqlite3.Database('./mydb.db' , (err)=>{
    if(err){
        return console.log(err);
    }
    console.log('Connected to mydb database!');
});

// display data from database

//let sql = 'SELECT * FROM langs';
let sql = `SELECT DISTINCT Name name FROM langs
           ORDER BY name`;

  db.all(sql, [], (err, rows) => {
    if (err) {  throw err;  }
    rows.forEach((row) => {
      console.log(row.name);
    });
  }); db.close();
```

Result will be:

```
D:\www\Express\week08> node select1

Connected to mydb database!

C++

Java

Nodejs
```

Query the first row in the result set:

When you know that the result set contains zero or one row e.g., querying a row based on the primary key or querying with only one aggregate function such as count, sum, max, min, etc., you can use the get() method of Database object.

The get() method executes an SQL query and calls the callback function on the first result row. In case the result set is empty, the row argument is undefined.

The following get.js program demonstrates how to query a the langs table by name:

get1.js

```
const sqlite3 = require('sqlite3').verbose();

let db = new sqlite3.Database('./mydb.db' , (err)=>{
    if(err){
        return console.log(err);
    }
    console.log('Connected to mydb database!');
});
let sql = `SELECT Name name  FROM langs
           WHERE Name  = ?`;
let name1 = 'Nodejs';
// first row only
db.get(sql, [name1], (err, row) => {
  if (err) {
    return console.error(err.message);
  }   return row
    ? console.log(row.id, row.name)
    : console.log(`No playlist found with the id ${name1}`);
});
```

```
db.close();
```

Query rows with each() method

The each() method executes an SQL query with specified parameters and calls a callback for every row in the result set.

The following illustrates the each() method:

each1.js

```
const sqlite3 = require('sqlite3').verbose();

let db = new sqlite3.Database('./mydb.db' , (err)=>{
    if(err){
        return console.log(err);
    }
    console.log('Connected to mydb database!');
});
let sql = `SELECT Name name  FROM langs
        WHERE Name  = ?`;
// first row only
db.each(sql, ['Nodejs'], (err, row) => {
  if (err) {
   throw err;  }
     console.log(row.name);
});
db.close();
```

```
D:\www\Express\week08> node each1

Connected to mydb database!

Nodejs

Nodejs

Nodejs
```

4. Controlling the Execution Flow

The sqlite3 module provides you with two methods for controlling the execution flow of statements. The serialize() method allows you to execute statements in serialized mode, while the parallelize() method executes the statements in parallel.

Let's look into each method in detail to understand how it works.

Executing statement in serialized mode with Database.serialize

The serialize() method puts the execution mode into serialized mode. It means that only one statement can execute at a time. Other statements will wait in a queue until all the previous statements are executed.

After the serialize() method returns, the execution mode is set to the original mode again.

Suppose, you want to execute the following three statements in sequence:

1- Create a new table.
2- Insert data into the table.
3- Query data from the table.

To do this, you place these statements in the serialize() method as follows:

Serialize1.js

```
const sqlite3 = require('sqlite3').verbose();

// open the database connection
let db = new sqlite3.Database(':memory:', (err) => {
  if (err) {
    console.error(err.message);
  }
});
db.serialize(() => {
  // Queries scheduled here will be serialized.
  db.run('CREATE TABLE greetings(message text)')
    .run(`INSERT INTO greetings(message)
          VALUES('Hi'),
                ('Hello'),
                ('Welcome')`)
    .each(`SELECT message FROM greetings`, (err, row) => {
      if (err){
        throw err;
      }
      console.log(row.message);
    });
});

// close the database connection
db.close((err) => {
  if (err) {
    return console.error(err.message);
  }
});
```

5. Updating Data in Database Tables

To update data in the SQLite database from a Node.js application, you use these steps:

1- Open a database connection.
2- Execute an UPDATE statement.
3- Close the database connection.

For the demonstration, we will use the langs table in the mydb.db database that we created above.

Serialize1.js

```
const sqlite3 = require('sqlite3').verbose();

// open a database connection
let db = new sqlite3.Database('./mydb.db');

//
let data = ['Expressjs', 'C++'];
let sql = `UPDATE langs
            SET name = ?
            WHERE name = ?`;

db.run(sql, data, function(err) {
  if (err) {
    return console.error(err.message);
  }
  console.log(`Row(s) updated: ${this.changes}`);

});
// close the database connection
db.close();
```

6. Deleting Data from Database Tables

To delete data from the SQLite database from a Node.js application, you use these steps:

1- Open a database connection.
2- Execute an Delete statement.
3- Close the database connection.

For the demonstration, we will use the langs table in the mydb.db database that we created above.

Delete1.js

```
const sqlite3 = require('sqlite3').verbose();

// open a database connection
let db = new sqlite3.Database('./mydb.db');
db.run(`DELETE FROM langs WHERE name=?`, 'Nodejs', (err)=> {
  if (err) {
    return console.error(err.message);
  }
  console.log(`Row(s) deleted ${this.changes}`);
});
// close the database connection
db.close();
```