Getting Started

In this tutorial, you will learn to make a simple setup of Sequelize.

Installing

Sequelize is available via npm (or yarn).

```
npm install --save sequelize
```

You'll also have to manually install the driver for your database of choice:

```
# One of the following:
$ 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
$ npm install --save oracledb # Oracle Database
```

Connecting to a database

To connect to the database, you must create a Sequelize instance. This can be done by either passing the connection parameters separately to the Sequelize constructor or by passing a single connection URI:

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

// Option 1: Passing a connection URI
const sequelize = new Sequelize('sqlite::memory:') // Example for sqlite
const sequelize = new
Sequelize('postgres://user:pass@example.com:5432/dbname') // Example for
postgres
```

```
// Option 2: Passing parameters separately (sqlite)
const sequelize = new Sequelize({
  dialect: 'sqlite',
  storage: 'path/to/database.sqlite'
});

// Option 3: Passing parameters separately (other dialects)
const sequelize = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: /* one of 'mysql' | 'postgres' | 'sqlite' | 'mariadb' | 'mssql' |
'db2' | 'snowflake' | 'oracle' */
});
```

The Sequelize constructor accepts a lot of options. They are documented in the API Reference.

Testing the connection

You can use the `.authenticate()` function to test if the connection is OK:

```
try {
  await sequelize.authenticate();
  console.log('Connection has been established successfully.');
} catch (error) {
  console.error('Unable to connect to the database:', error);
}
```

Closing the connection

Sequelize will keep the connection open by default, and use the same connection for all queries. If you need to close the connection, call `sequelize.close()` (which is asynchronous and returns a Promise).

> **NOTE**
>
> Once `sequelize.close()` has been called, it's impossible to open a new connection. You will need to create a new Sequelize instance to access your database again.

Terminology convention

Observe that, in the examples above, `Sequelize` refers to the library itself while `sequelize` refers to an instance of Sequelize, which represents a connection to one database. This is the recommended convention and it will be followed throughout the documentation.

Tip for reading the docs

You are encouraged to run code examples locally while reading the Sequelize docs. This will help you learn faster. The easiest way to do this is using the SQLite dialect:

```
const { Sequelize, Op, Model, DataTypes } = require('sequelize');
const sequelize = new Sequelize('sqlite::memory:');

// Code here! It works!
```

To experiment with the other dialects, which are harder to set up locally, you can use the Sequelize SSCCE GitHub repository, which allows you to run code on all supported dialects directly from GitHub, for free, without any setup!

New databases versus existing databases

If you are starting a project from scratch, and your database is still empty, Sequelize can be used from the beginning in order to automate the creation of every table in your database.

Also, if you want to use Sequelize to connect to a database that is already filled with tables and data, that works as well! Sequelize has got you covered in both cases.

Logging

By default, Sequelize will log into the console for every SQL query it performs. The `options.logging` option can be used to customize this behavior, by defining the function that gets executed every time Sequelize logs something. The default value is `console.log` and when using that only the first log parameter of a log function call is displayed. For example, for query logging the first parameter is the raw query and the second (hidden by default) is the Sequelize object.

Common useful values for `options.logging`:

```
const sequelize = new Sequelize('sqlite::memory:', {
  // Choose one of the logging options
  logging: console.log, // Default, displays the first parameter of the log function call
  logging: (...msg) => console.log(msg), // Displays all log function call parameters
  logging: false, // Disables logging
  logging: msg => logger.debug(msg), // Use custom logger (e.g. Winston or Bunyan), displays the first parameter
  logging: logger.debug.bind(logger), // Alternative way to use custom logger, displays all messages
});
```

Promises and async/await

Most of the methods provided by Sequelize are asynchronous and therefore return Promises. They are all Promises, so you can use the Promise API (for example, using `then`, `catch`, `finally`) out of the box.

Of course, using `async` and `await` works fine as well.