

## 13.1.4 Scaffold the Application Architecture

As with most applications using Node.js, we'll start with initializing Node.js, adding the packages, and installing the dependencies we need. The following steps walk you through this process:

1. Use `cd` to navigate to the root of your cloned repository folder.
2. Run `npm init` or `npm init -y` from the command line to initialize a new Node.js package. Your entry point for the program should be `server.js`. If you use the `npm init -y` option, remember to manually update your `package.json` to `"main": "server.js"` instead of `"main": "index.js"`.
3. Once that's done, update `package.json` with the following script:

```
"start": "node server.js"
```

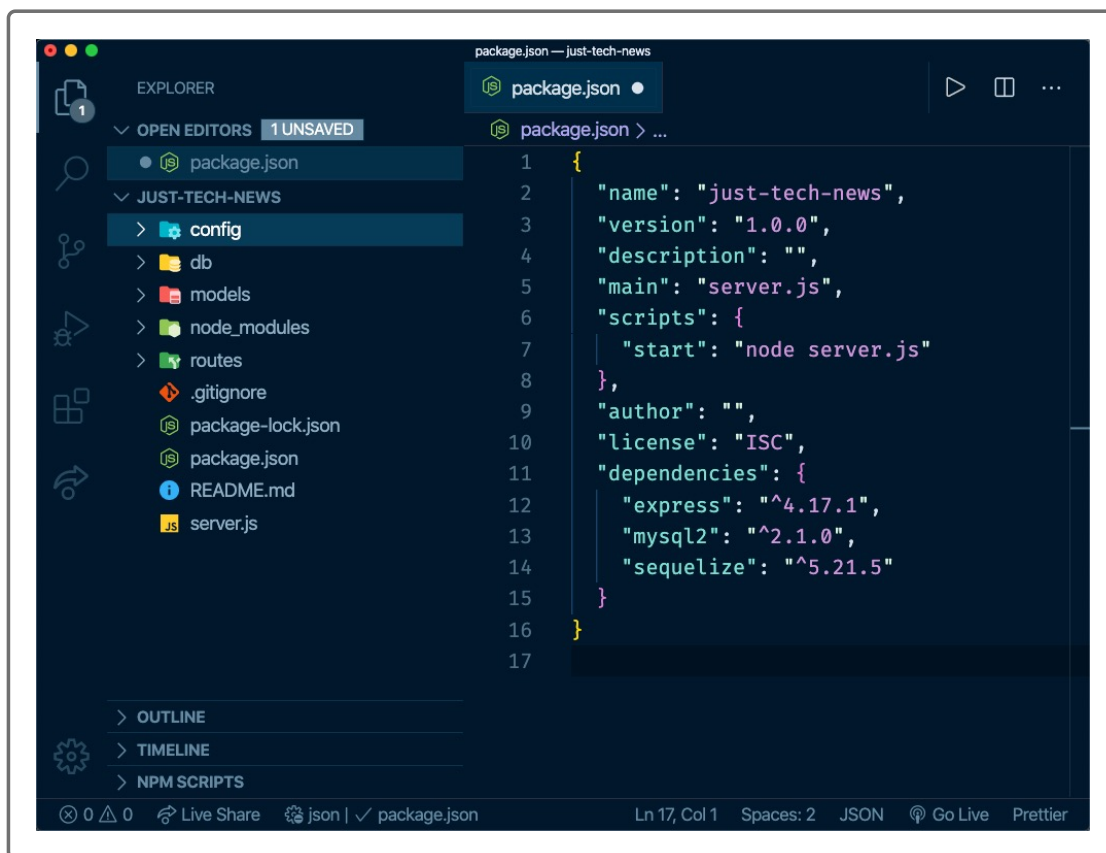
4. Create a `.gitignore` file, and add the following files and folders:
  - `node_modules`
  - `.DS_Store`

5. Create a `server.js` file.
6. Create folders called `models`, `routes`, `config`, and `db`.
7. Lastly, install the dependencies by using the following command:

```
npm install express sequelize mysql2
```

Note that in the last step, we installed multiple libraries using npm by putting a space between each library's name.

To confirm that everything is in place, compare your setup to the following image:



If your setup doesn't match, make sure you were in the right location when you ran the setup steps at the command line.

Notice that we installed a package called `mysql2`. Why didn't we install `mysql`? While there is a `mysql` library that can connect to a MySQL database through the Node.js application, Sequelize prefers to work with the `mysql2` library, which appears to be the successor of the native `mysql` library for Node.js.

## DEEP DIVE ▼

We also installed Sequelize, so let's start by learning more about it.

---

## Hello, Sequelize

When using SQL queries in a Node.js application, you need to do a bit of context switching because you'll be intermixing two different programming languages. When working with more involved SQL queries, having this type of connection between the JavaScript code and the SQL code can get tricky.

There's also the issue of validating data. Do we really want potentially harmful or incorrect data getting to the SQL database?

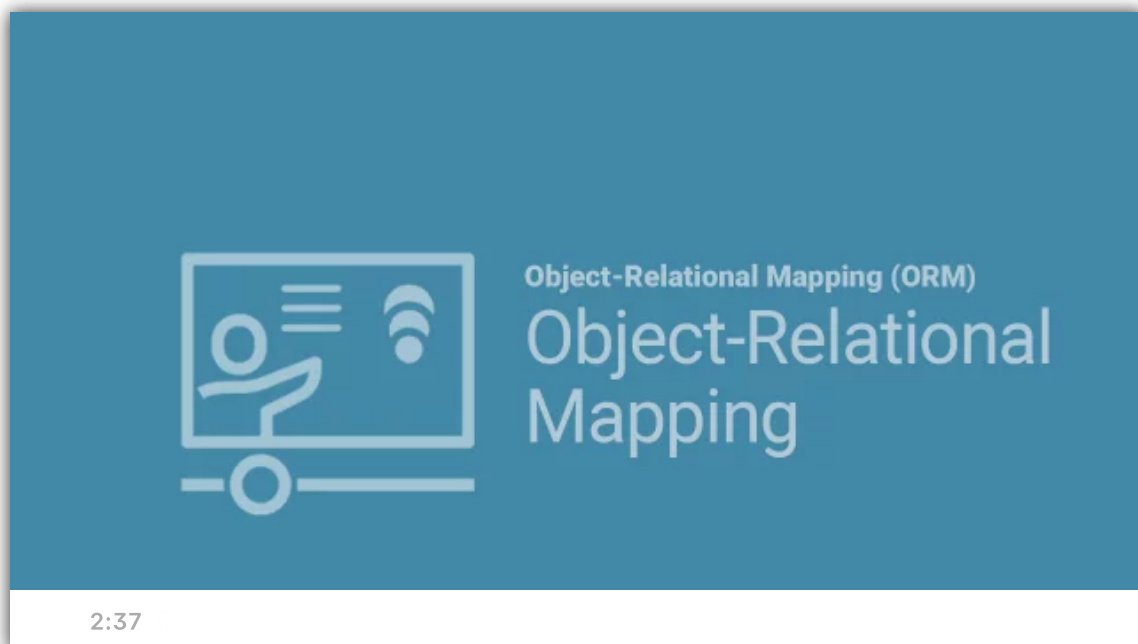
This is where Sequelize comes in. Take a minute to familiarize yourself with the [Sequelize version 5 documentation](https://sequelize.org/v5/) [\(https://sequelize.org/v5/\)](https://sequelize.org/v5/). This document has more information than we need at the moment, so just focus on the main page and the [Getting Started](https://sequelize.org/v5/manual/getting-started.html) [\(https://sequelize.org/v5/manual/getting-started.html\)](https://sequelize.org/v5/manual/getting-started.html) page for now.

The beginning of the document contains the following explanation: "Sequelize is a Promise-based Node.js ORM for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server. It features solid transaction support, relations, eager and lazy loading, read replication and more."

Sequelize is a JavaScript library that works with any dialect of SQL but we'll use it with MySQL as the dialect. This is why we installed the `mysql2` library with Sequelize. Because it doesn't know which dialect of SQL we'll use, we have to install that dialect and then instruct Sequelize that we're using it.

The biggest benefit that an ORM like Sequelize offers is sparing us from spending a lot of time writing SQL to define our tables and queries. Instead, we can use object-oriented concepts to model our database tables using JavaScript classes. This lets us add validators and custom rules to the SQL data where SQL sometimes falls short.

Sequelize's benefits are hard to understand without experiencing them firsthand, which we'll get to shortly. First, watch the following video explaining ORM and how it can improve our work:



Now that we have a better understanding of ORMs, let's create a database for the application to synchronize with, and write the accompanying JavaScript code.

---

## Create and Connect to the Database

With Sequelize, we no longer have to bother with creating the entire SQL table schema and running it through the SQL shell. All we need to do is create a database; then when we start the app, Sequelize will create the tables for us!

It's a good practice to keep the database information with the application's code, so let's add it to a file. Create a file in the `db` directory called `schema.sql` and add the following code to it:

```
DROP DATABASE IF EXISTS just_tech_news_db;  
  
CREATE DATABASE just_tech_news_db;
```

Now that the database setup code is created, let's navigate to the MySQL shell and get the database up and running by following these steps:

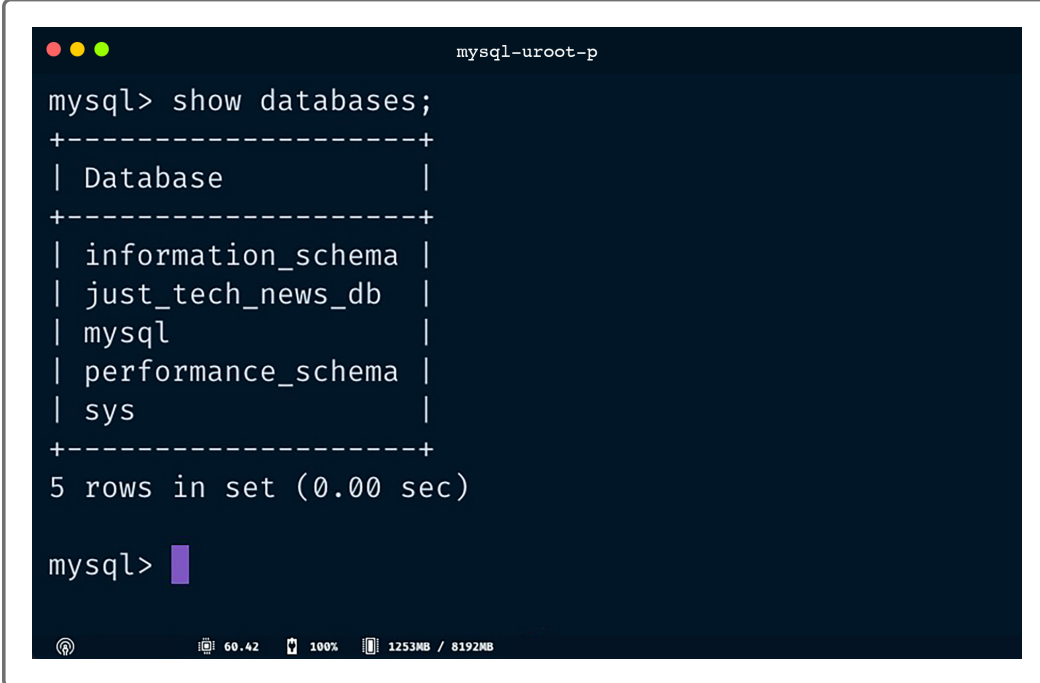
### IMPORTANT

Remember, if you use a PC, you'll use the Command Prompt application (not GitBash). If you're using macOS, these commands will work right in the Terminal command-line application.

1. From the root directory of your project, type `mysql -u root -p` and press Return.
2. Enter your MySQL password and press Return again to enter the MySQL shell environment.
3. To create the database, execute the following command:

```
source db/schema.sql
```

4. To ensure it worked, follow that command with `show databases;`. It should print a table like the following image shows:



```
mysql-uroot-p
mysql> show databases;
+-----+
| Database                |
+-----+
| information_schema      |
| just_tech_news_db      |
| mysql                   |
| performance_schema      |
| sys                     |
+-----+
5 rows in set (0.00 sec)

mysql>
```

5. Lastly, let's quit the shell environment by simply entering `quit` and pressing Return.

Now that we've set up the database, we can focus on connecting the application to it!

## Create the Database Connection

Set up the application to connect to the database using Sequelize. Create a file in the `config` directory called `connection.js`. Once that's created, we'll call the `Sequelize` connection constructor function by adding the following code to `connection.js`:

```
// import the Sequelize constructor from the library
const Sequelize = require('sequelize');

// create connection to our database, pass in your MySQL information for use
```

```
const sequelize = new Sequelize('just_tech_news_db', 'username', 'password',  
  host: 'localhost',  
  dialect: 'mysql',  
  port: 3306  
});  
  
module.exports = sequelize;
```

All we're doing here is importing the base Sequelize class and using it to create a new connection to the database. The `new Sequelize()` function accepts the database name, MySQL username, and MySQL password (respectively) as parameters, then we also pass configuration settings. Once we're done, we simply export the connection.

## DEEP DIVE ▼

Before we start thinking about the database tables, we need to consider one thing. If we were to push this to GitHub right now, does that mean everyone who sees the app can see the MySQL username and password? Unfortunately, yes. If you use a sensitive password instead of the default "root" password, someone else now knows it. We should probably fix that before we push our work!

## Set Up Environment Variables

Think back to your first Express.js application. To deploy the application, you had to set the server's port configuration to look like the following code:

```
const PORT = process.env.PORT || 3001;
```

This is because Heroku, and similar production environments, can't reserve the port 3001 for us and will provide a port dynamically. So we use the `process.env.PORT` variable to instruct the app that if the production environment provides a port for us, to use that one.

Well, we can do the exact same thing locally! As we'll see later, the database credentials we'll use for deploying to production won't be the same as the local credentials. So what we can do is set up local environment variables for development, then use the production environment variables when we deploy to Heroku. This means that our sensitive data will not be exposed when we push to GitHub!

To do this, we'll need to install another NPM package called `dotenv`. From the command line, run the following code:

```
npm install dotenv
```

## DEEP DIVE ▼

Once that's installed, at the root of your application create a file called `.env` and add the following code:

```
DB_NAME='just_tech_news_db'  
DB_USER='your-mysql-username'  
DB_PW='your-mysql-password'
```

That's all you have to do to set up these environment variables. Next, load them into the `connection.js` file. Let's update that file to look like the following code:



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

require('dotenv').config();

// create connection to our db
const sequelize = new Sequelize(process.env.DB_NAME, process.env.DB_USER, process.env.DB_PASSWORD, {
  host: 'localhost',
  dialect: 'mysql',
  port: 3306
});

module.exports = sequelize;
```

Notice how we don't have to save the `require('dotenv')` to a variable? All we need it to do here is execute when we use `connection.js` and all of the data in the `.env` file will be made available at `process.env.<ENVIRONMENT-VARIABLE-NAME>`.

Remember to add `.env` to your `.gitignore` file. It would defeat the purpose if we pushed up the data we were trying to hide!

Great job setting up this project! We have a lot of work to do, but having the application's structure properly in place will make building it out easier as we go.