Jul 26, 2021 · 8 min read

Using Sequelize with TypeScript

Ibiyemi Adewakun
Ibiyemi is a full-stack developer from Lagos. When she's not writing code, she likes to read, listen to music, and put cute outfits together.

Writing raw SQL in your API is so passé, or at best it's reserved for really complex queries. These are simpler times for developing, and for most APIs, using one of many object-relational mappers (ORMs) is sufficient.

ORMs also conveniently encapsulate the intricate details of communicating with a database and its query language.

This means you can use a single ORM on multiple database types like MySQL, PostgreSQL, or MongoDb, making it easy to switch between databases without rewriting your code! You can also connect different types of databases to your project while using the same code to access them.

In this article, you'll learn how to use the Sequelize ORM with TypeScript. So grab your laptops, open your IDE, and let's get started!

Prerequisites

To follow along with this article, install the following:

- `Node.js`
- JavaScript package manager; we'll use yarn
- An IDE or text editor of your choice, like Sublime Text or Visual Studio Code

Setting up the project

To begin our project, let's set up a simple Express.js API to create a virtual cookbook that stores recipes and ingredients, and tags our recipes with popular categories.

First, let's create our project directory by typing the following into our terminal:

```
$ mkdir cookbook
$ cd cookbook
```

Inside the new `cookbook` project directory, install the needed project dependencies using `yarn`. First, run `npm init` to initialize the Node.js project with a `package.json` file:

```
$ npm init
```

After the Node.js project initializes, install the dependencies starting with `express`:

```
$ yarn add express
```

Next, add TypeScript to the project by running the following:

```
$ yarn add -D typescript ts-node @types/express @types/node
```

💡 Note that we've added a flag, `-D`, to our installation command. This flag tells Yarn to add these libraries as dev dependencies, meaning these libraries are only needed when the project is in development. We also added type definitions for Express.js and Node.js.

With TypeScript added to our project, let's initialize it:

```
$ npx tsc --init
```

This creates our TypeScript configuration file `ts.config`, and sets the default values:

```
// ts.config
{
    "compilerOptions": {
      "target": "es5",
      "module": "commonjs",
      "sourceMap": true,
      "outDir": "dist",
      "strict": true,
      "esModuleInterop": true,
      "skipLibCheck": true,
      "forceConsistentCasingInFileNames": true
    }
}
```

Find more information about customizing `ts.config` here.

Finally, let's define a simple API structure for our project by creating project directories and files to match the outline below:

```
- dist # the name of our outDir set in tsconfig.json
- src
  - api
```

```
    - controllers
    - contracts
    - routes
    - services
  - db
    - dal
    - dto
    - models
    config.ts
    init.ts
  - errors
  index.ts
  ts.config
```

Now that we have defined our project structure in the `index.ts` file, which is our application's starting point, add the following code to create our Express.js server:

```
# src/index.ts

import express, { Application, Request, Response } from 'express'

const app: Application = express()
const port = 3000

// Body parsing Middleware
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.get('/', async(req: Request, res: Response): Promise<Response
    return res.status(200).send({ message: `Welcome to the cookboo
})

try {
    app.listen(port, () => {
        console.log(`Server running on http://localhost:${port}`)
    })
```

```
} catch (error) {
    console.log(`Error occurred: ${error.message}`)
}
```

We must also include some additional libraries to run the application easily and pass in environment variables. These additional libraries are nodemon using `yarn add -D nodemon`, eslint using `yarn add -D eslint`, and dotenv using `yarn add dotenv`.

Setting up the Sequelize ORM

At this point, the Express.js application is running, so it's time to bring in the fun stuff: Sequelize ORM!

Start by adding Sequelize to the project by running the following:

```
$ yarn add sequelize
$ yarn add mysql2
```

Although we added the database driver for MySQL, which is solely based on personal preference, you can install any driver of your preferred database instead. View here for other available database drivers.

Initiating Sequelize's connection

After installing Sequelize, we must initiate its connection to our database. Once initiated, this connection registers our models:

```
# db/config.ts

import { Dialect, Sequelize } from 'sequelize'
```

```
const dbName = process.env.DB_NAME as string
const dbUser = process.env.DB_USER as string
const dbHost = process.env.DB_HOST
const dbDriver = process.env.DB_DRIVER as Dialect
const dbPassword = process.env.DB_PASSWORD

const sequelizeConnection = new Sequelize(dbName, dbUser, dbPassw
  host: dbHost,
  dialect: dbDriver
})

export default sequelizeConnection
```

Creating and registering Sequelize models

Sequelize provides two ways to register models: using `sequelize.define` or extending the Sequelize model class. In this tutorial, we'll use the model extension method to register our `Ingredient` model.

We begin by creating the interfaces of the following:

- `IngredientAttributes` defines all the possible attributes of our model
- `IngredientInput` defines the type of the object passed to Sequelize's `model.create`
- `IngredientOuput` defines the returned object from `model.create`, `model.update`, and `model.findOne`

```
# db/models/Ingredient.ts

import { DataTypes, Model, Optional } from 'sequelize'
import sequelizeConnection from '../config'

interface IngredientAttributes {
```

```
  id: number;
  name: string;
  slug: string;
  description?: string;
  foodGroup?: string;
  createdAt?: Date;
  updatedAt?: Date;
  deletedAt?: Date;
}
export interface IngredientInput extends Optional<IngredientAttr
export interface IngredientOuput extends Required<IngredientAttr
```

Next, create an `Ingredient` class that extends, initializes, and exports the `import {Model} from 'sequelize'` Sequelize model class:

```
# db/models/Ingredient.ts

...

class Ingredient extends Model<IngredientAttributes, IngredientI
  public id!: number
  public name!: string
  public slug!: string
  public description!: string
  public foodGroup!: string

  // timestamps!
  public readonly createdAt!: Date;
  public readonly updatedAt!: Date;
  public readonly deletedAt!: Date;
}

Ingredient.init({
  id: {
    type: DataTypes.INTEGER.UNSIGNED,
```

```
      autoIncrement: true,
      primaryKey: true,
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false
    },
    slug: {
      type: DataTypes.STRING,
      allowNull: false,
      unique: true
    },
    description: {
      type: DataTypes.TEXT
    },
    foodGroup: {
      type: DataTypes.STRING
    }
  }, {
    timestamps: true,
    sequelize: sequelizeConnection,
    paranoid: true
  })

  export default Ingredient
```

💡 Note we added the option `paranoid: true` to our model; this imposes a soft delete on the model by adding a `deletedAt` attribute that marks records as `deleted` when invoking the `destroy` method.

To complete our model and create its target table in the connected database, run the model `sync` method:

```
  # db/init.ts
```

```
import { Recipe, RecipeTags, Tag, Review, Ingredient, RecipeIngr
const isDev = process.env.NODE_ENV === 'development'

const dbInit = () => {
  Ingredient.sync({ alter: isDev })
}
export default dbInit
```

💡 The `sync` method accepts the `force` and `alter` options. The `force` option forces the recreation of a table. The `alter` option creates the table if it does not exist or updates the table to match the attributes defined in the model.

💡 Pro tip: reserve using `force` or `alter` for development environments so you don't accidentally recreate your production database, losing all your data or applying changes to your database that might break your application.

Using our models in DAL and services

The data access layer (DAL) is where we implement our SQL queries, or in this case, where the Sequelize model queries run:

```
# db/dal/ingredient.ts

import {Op} from 'sequelize'
import {Ingredient} from '../models'
import {GetAllIngredientsFilters} from './types'
import {IngredientInput, IngredientOuput} from '../models/Ingred

export const create = async (payload: IngredientInput): Promise<
    const ingredient = await Ingredient.create(payload)
    return ingredient
}
```

```
export const update = async (id: number, payload: Partial<Ingredi
    const ingredient = await Ingredient.findByPk(id)
    if (!ingredient) {
        // @todo throw custom error
        throw new Error('not found')
    }
    const updatedIngredient = await (ingredient as Ingredient).up
    return updatedIngredient
}

export const getById = async (id: number): Promise<IngredientOupu
    const ingredient = await Ingredient.findByPk(id)
    if (!ingredient) {
        // @todo throw custom error
        throw new Error('not found')
    }
    return ingredient
}

export const deleteById = async (id: number): Promise<boolean> =>
    const deletedIngredientCount = await Ingredient.destroy({
        where: {id}
    })
    return !!deletedIngredientCount
}

export const getAll = async (filters?: GetAllIngredientsFilters)
    return Ingredient.findAll({
        where: {
            ...(filters?.isDeleted && {deletedAt: {[Op.not]: null}
        },
        ...((filters?.isDeleted || filters?.includeDeleted) && {pa
    })
}
```

Adding the `paranoid: true` option to the `findAll` model method includes the

soft-deleted records with `deletedAt` set in the result. Otherwise, the results exclude soft deleted records by default.

In our DAL above, we defined some commonly needed CRUD queries using our `ModelInput` type definition and placing any additional types in `db/dal/types.ts`:

```
# db/dal/types.ts

export interface GetAllIngredientsFilters {
    isDeleted?: boolean
    includeDeleted?: boolean
}
```

💡 Sequelize ORM has some really cool model methods, including `findAndCountAll`, which returns a list of records and a count of all records matching the filter criteria. This is really useful for returning paginated list responses in an API.

Now we can create our service, which acts as an intermediary between our controller and DAL:

```
# api/services/ingredientService.ts

import * as ingredientDal from '../dal/ingredient'
import {GetAllIngredientsFilters} from '../dal/types'
import {IngredientInput, IngredientOuput} from '../models/Ingred

export const create = (payload: IngredientInput): Promise<Ingred
    return ingredientDal.create(payload)
}
export const update = (id: number, payload: Partial<IngredientInp
    return ingredientDal.update(id, payload)
}
```

```
export const getById = (id: number): Promise<IngredientOuput> =>
    return ingredientDal.getById(id)
}
export const deleteById = (id: number): Promise<boolean> => {
    return ingredientDal.deleteById(id)
}
export const getAll = (filters: GetAllIngredientsFilters): Promi
    return ingredientDal.getAll(filters)
}
```

Powering up the model with routes and controllers

We've come a long way! Now that we have services fetching our data from our database, it's time to bring all that magic to the public using routes and controllers.

Let's start by creating our `Ingredients` routes in `src/api/routes/ingredients.ts`:

```
# src/api/routes/ingredients.ts

import { Router } from 'express'

const ingredientsRouter = Router()
ingredientsRouter.get(':/slug', () => {
  // get ingredient
})
ingredientsRouter.put('/:id', () => {
  // update ingredient
})
ingredientsRouter.delete('/:id', () => {
  // delete ingredient
})
```

```
ingredientsRouter.post('/', () => {
  // create ingredient
})

export default ingredientsRouter
```

Our cookbook API will eventually have several routes, such as `Recipes` and `Tags`. So, we must create an `index.ts` file to register the different routes to their base paths and have one central export to connect to our Express.js server from earlier:

```
# src/api/routes/index.ts

import { Router } from 'express'
import ingredientsRouter from './ingredients'

const router = Router()

router.use('/ingredients', ingredientsRouter)

export default router
```

Let's update our `src/index.ts` to import our exported routes and register them to our Express.js server:

```
# src/index.ts

import express, { Application, Request, Response } from 'express'
import routes from './api/routes'

const app: Application = express()

...
```

```
app.use('/api/v1', routes)
```

After creating and connecting the routes, let's create a controller to link to our routes and call the service methods.

To support typing the parameters and results between the routes and controllers, let's add data transfer objects (DTOs) and mappers to transform the results:

```
# src/api/controllers/ingredient/index.ts

import * as service from '../../../db/services/IngredientService'
import {CreateIngredientDTO, UpdateIngredientDTO, FilterIngredie
import {Ingredient} from '../../interfaces'
import * as mapper from './mapper'

export const create = async(payload: CreateIngredientDTO): Promi
    return mapper.toIngredient(await service.create(payload))
}
export const update = async (id: number, payload: UpdateIngredier
    return mapper.toIngredient(await service.update(id, payload))
}
export const getById = async (id: number): Promise<Ingredient> =>
    return mapper.toIngredient(await service.getById(id))
}
export const deleteById = async(id: number): Promise<Boolean> =>
    const isDeleted = await service.deleteById(id)
    return isDeleted
}
export const getAll = async(filters: FilterIngredientsDTO): Prom
    return (await service.getAll(filters)).map(mapper.toIngredien
}
```

Now, update the router with the calls to the controller:

```
# src/api/routes/ingredients.ts
```

```
import { Router, Request, Response} from 'express'
import * as ingredientController from '../controllers/ingredient'
import {CreateIngredientDTO, FilterIngredientsDTO, UpdateIngredi

const ingredientsRouter = Router()

ingredientsRouter.get('/:id', async (req: Request, res: Response)
    const id = Number(req.params.id)
    const result = await ingredientController.getById(id)
    return res.status(200).send(result)
})
ingredientsRouter.put('/:id', async (req: Request, res: Response)
    const id = Number(req.params.id)
    const payload:UpdateIngredientDTO = req.body

    const result = await ingredientController.update(id, payload)
    return res.status(201).send(result)
})
ingredientsRouter.delete('/:id', async (req: Request, res: Respon
    const id = Number(req.params.id)

    const result = await ingredientController.deleteById(id)
    return res.status(204).send({
        success: result
    })
})
ingredientsRouter.post('/', async (req: Request, res: Response) =
    const payload:CreateIngredientDTO = req.body
    const result = await ingredientController.create(payload)
    return res.status(200).send(result)
})
ingredientsRouter.get('/', async (req: Request, res: Response) =>
    const filters:FilterIngredientsDTO = req.query
    const results = await ingredientController.getAll(filters)
    return res.status(200).send(results)
```

```
})
export default ingredientsRouter
```

At this point, we can add a build script to run our API:

```
# package.json

...
"scripts": {
  "dev": "nodemon src/index.ts",
  "build": "npx tsc"
},
...
```

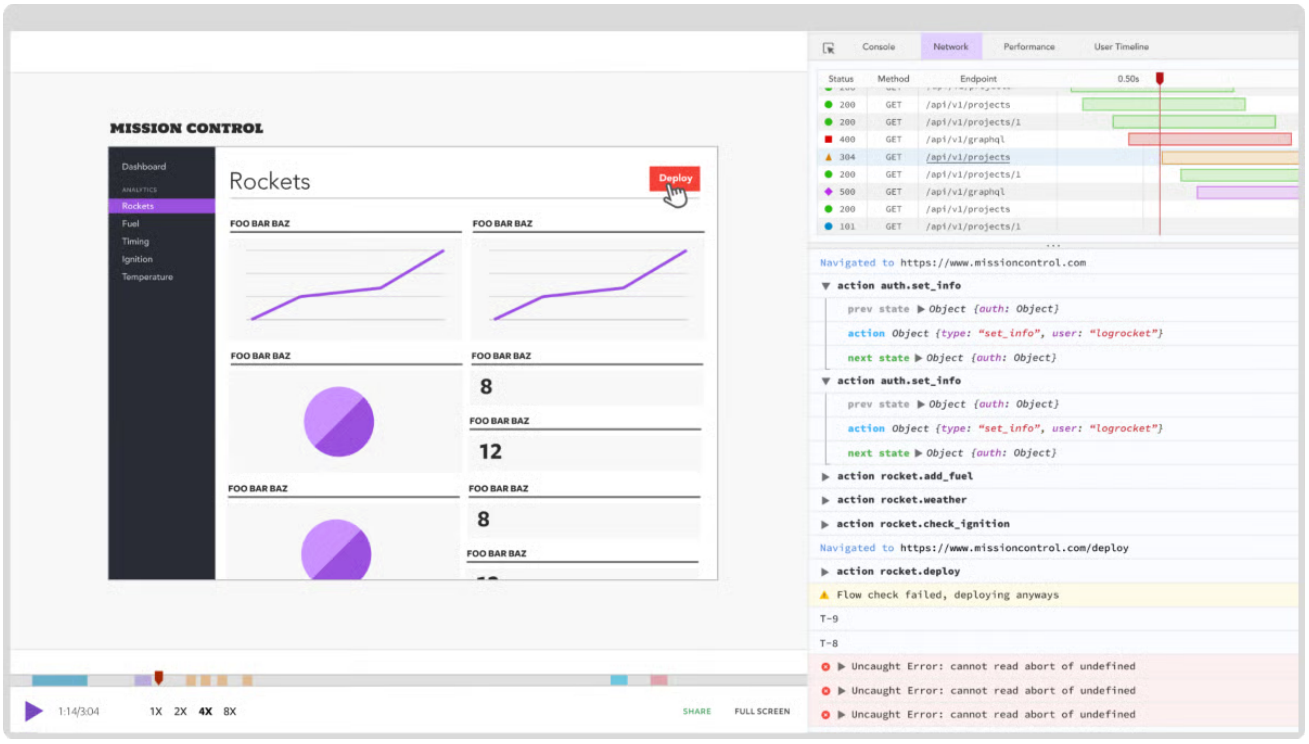
To see the final product, run the API using `yarn run dev` and visit our ingredient endpoints at http://localhost:3000/api/v1/ingredients.

Conclusion

In this article, we set up a simple TypeScript application with Express.js to use the Sequelize ORM and walked through initializing Sequelize, creating our models, and running queries through the ORM.

Using Sequelize with TypeScript in our project helps us write less code and abstract the database engine while defining strict types for model input and output. This makes our code is more consistent, even if we change database types, and can prevent the occurrence of SQL injection to our tables.

The entire code from this article is available on Github. I hope you found this article easy to follow and I would love to hear any ideas you have on cool ways to use Sequelize in your application or any questions you have in the comment section!



LogRocket lets you replay user sessions, eliminating guesswork by showing exactly what users experienced. It captures console logs, errors, network requests, and pixel-perfect DOM recordings — compatible with all frameworks, and with plugins to log additional context from Redux, Vuex, and @ngrx/store.

With Galileo AI, you can instantly identify and explain user struggles with automated monitoring of your entire product experience.

Modernize how you understand your web and mobile apps — start monitoring for free.

#typescript

Recent posts:

How to create fancy corners using CSS `corner-shape`

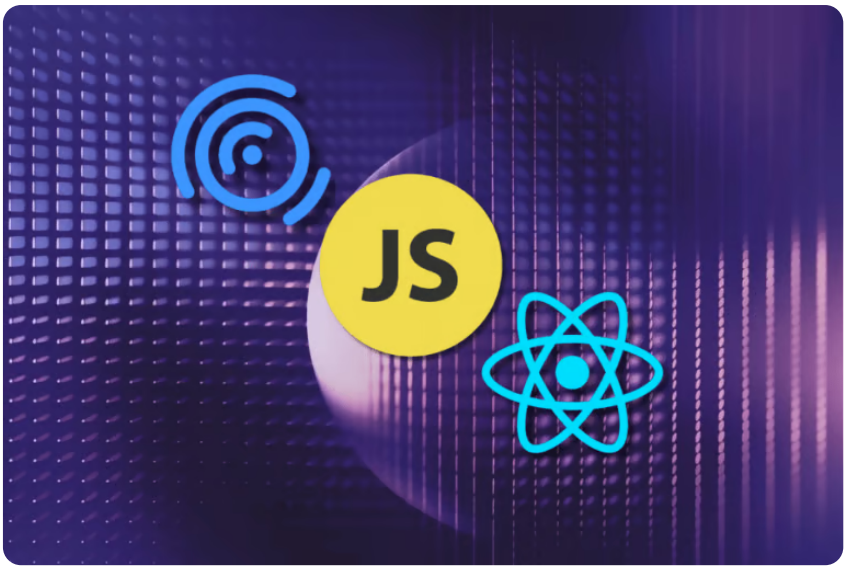
Learn about CSS's `corner-shape` property and how to use it, as well as the more advanced side of `border-radius` and why it's crucial to using `corner-shape` effectively.

Daniel Schwarz
Nov 26, 2025 · 7 min read

The Replay (11/26/25): An AI reality check, Prisma v7, and more

An AI reality check, Prisma v7, and “caveman compression”: discover what’s new in The Replay, LogRocket’s newsletter for dev and engineering leaders, in the November 26th issue.

Matt MacCormack
Nov 26, 2025 · 35 sec read

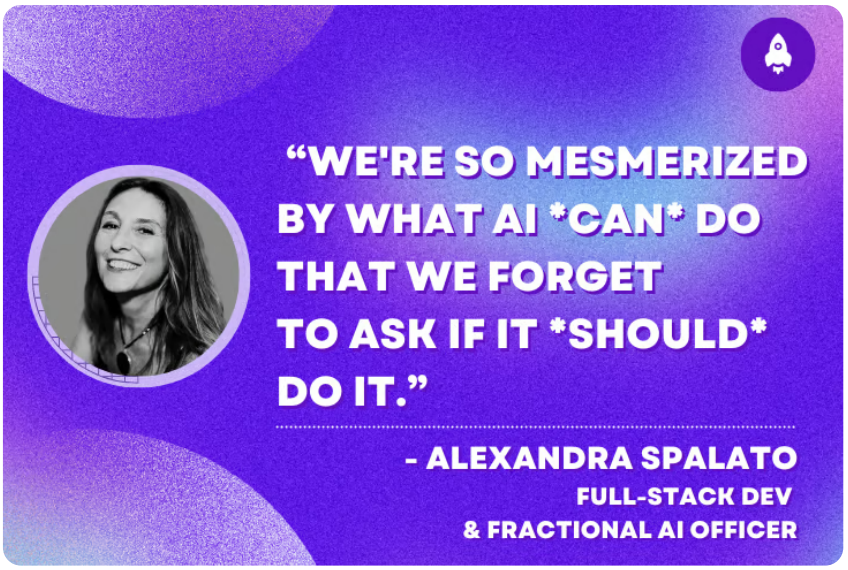


Ripple over React? Evaluating the newest JS framework

RippleJS takes a fresh approach to UI development with no re-renders and TypeScript built in. Here's why it's gaining attention.

Chizaram Ken

Nov 26, 2025 · 15 min read



You don’t need AI for everything: A reality check for developers

As a developer, it’s easy to feel like you need to integrate AI into every feature and deploy agents for every task. But what if the smartest move isn’t to use AI, but to know when not to?

Alexandra Spalato
Nov 26, 2025 · 6 min read

View all posts

8 Replies to "Using Sequelize with TypeScript"

juka1994 says: Reply

March 15, 2022 at 5:43 pm

Hi I have a doubt. I check the proyect git. folder /lib two files. local-cache.ts and check-cache.ts when would the check-cache be used. thank you. i wait your answer

Ronald Paniagua says: Reply

June 30, 2022 at 8:56 am

Hello, excellent contribution for those of us who are just starting out, I would like to know what benefit the use of the service brings to communicate with the Controller with the DAL layer, thank you.

Faizal says: Reply

August 15, 2022 at 10:05 pm

so excellent tutorial about typescript expressjs. i appreciate bro!

Nawlbergs says: Reply

September 10, 2022 at 9:39 am

Interesting.. I checked out your repo… its pretty different from how I have implemented my server. Main issues I have is there is a lot of boilerplate and it is not very DRY..(ex, there are multiple places the attributes on a model are defined .. ie.. defined in the model, in the dto, in the interface, then again in the mappers). Also, your cache would be very dangerous in any server behind a load balancer.

Arun Thomas says: Reply

May 19, 2023 at 9:41 am

Could you please tell me which boilerplate is the best?

Now I'm seeking to start a new project in Node with sequelize, and I'd appreciate it if you could share your repo.

hellues says: Reply

February 14, 2023 at 9:59 am

i couldnt see ingredients.dto where is that file?

Jayant Seth says: Reply

June 3, 2023 at 8:11 am

It's mentioned at the end, here it is : https://github.com/ibywaks/cookbook/blob/master/src/api/dto/ingredient.dto.ts

sergiously says: Reply

July 10, 2024 at 8:53 pm

how do you implement model associations? belongsTo, hasMany, etc.

Leave a Reply

This Page Cannot Be Displayed

Based on your organization's access policies, access to application Wordpress of type Blogging has been blocked.

Web Page Blocked as per BEL Internet Access Policy

If you have questions, please contact Please contact at IP-Ph:27776/27777
(websupport@bel.co.in) and provide the codes shown below.

Date: Thu, 27 Nov 2025 10:45:32 IST
Username: VDI\swsbubusiness@VDI_AD
Source IP: 172.19.9.12
URL: GET https://jetpack.wordpress.com/jetpack-comment/?
blogid=217016018&postid=59700&comment_registration=0&require_name_email=1&stc_enabled=1&s
%25s&color_scheme=light&lang=en_US&jetpack_version=15.1.1&iframe_unique_id=1&show_cookie_
Category: Computers and Internet
Reason: BLOCK-AVC
Notification: AVC