

Express + Knex + Objection = Painless API with DB

How to not get mad at your code (and yourself) while connecting to a database



Nicola Dall'Asen

May 4 · 6 min read

TL;DR

In this article we'll build an API system on Node.js for a simple users/messages database.

The final code can be found [here](#).

Introduction

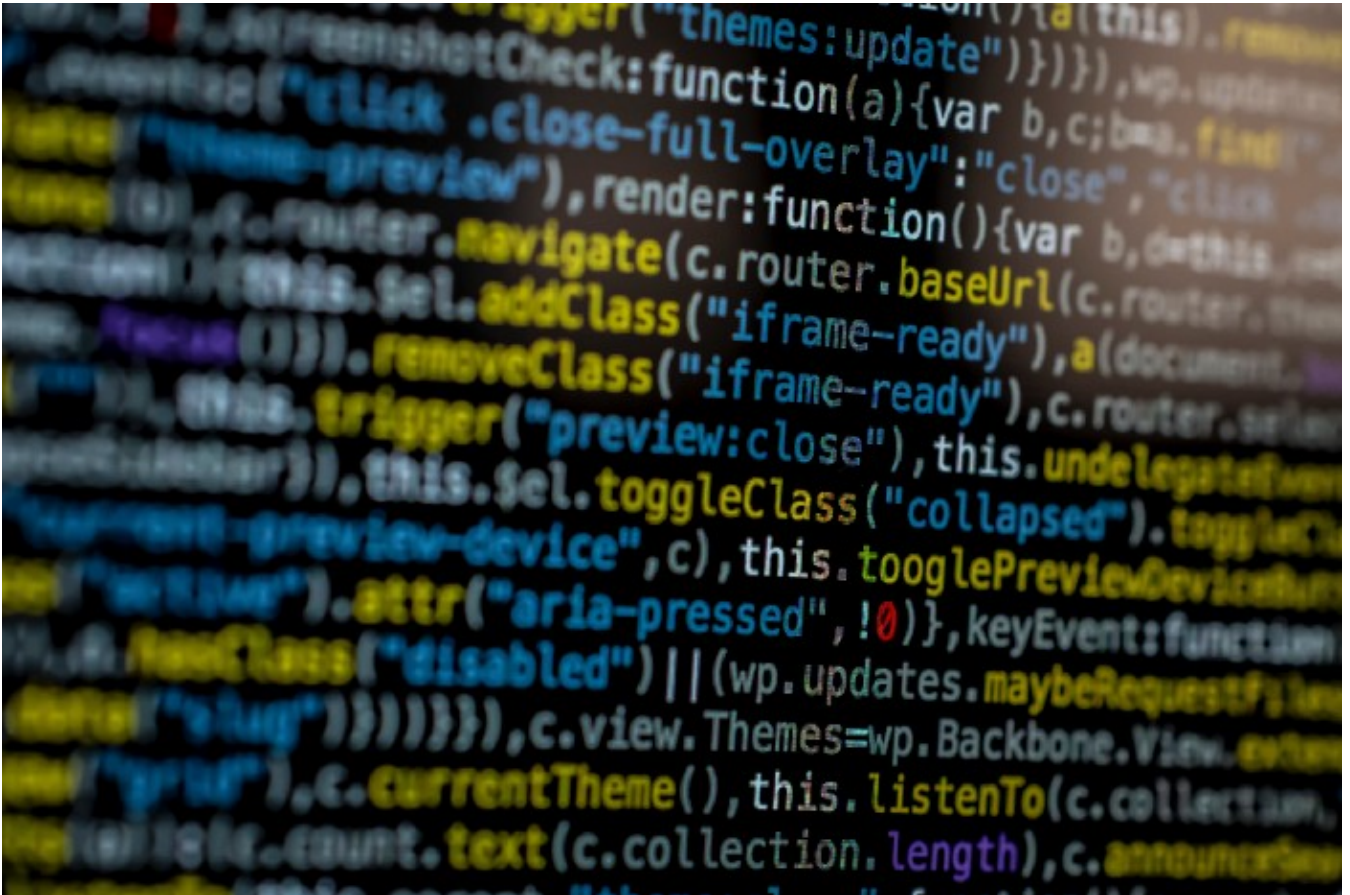
Node.js doesn't provide a simple way out of the box to connect to databases due to its general purpose nature, of course there are packages like `pg` or `mongoose` to interact directly to your database, but if your application gets big enough, managing connection and queries through these packages can become easily a pain.

What we'll need:

- `Node.js` (any recent version should be fine)
- `pg` for accessing PostgreSQL databases, or any equivalent for other DBMS
- `knex` SQL builder
- `objection` to define Models (see later)
- `express` to define our endpoints

Optional, to expand the project:

- `body-parser` to manage POST requests



You don't want your code to look like this right?

Knex.js

This is the right time to introduce the first package; Knex.js allows to easily define the connection properties in a separate file (called `knexfile.js`) and it can be configured differently for your development/testing/production server.

In this file you get to define also the migration (which tables are to be created) and seeds (DB population) folders, so you don't have to manually track your tables and data.

After installing the package (`npm install -g knex`) you can type `knex init` in your console to generate the default configuration file, after adapting it to your DB it should look like this:

```
1 module.exports = {
2   test: {
3     client: 'pg',
4     connection: 'postgres://localhost/test_db',
5     migrations: {
6       directory: __dirname + '/db/migrations'
```

```
7     },
8     seeds: {
9       directory: __dirname + '/db/seeds/test'
10    }
11  },
12  development: {
13    client: 'pg',
14    connection: 'postgres://localhost/development_db',
15    migrations: {
16      directory: __dirname + '/db/migrations'
17    },
18    seeds: {
19      directory: __dirname + '/db/seeds/development'
20    }
21  },
22  production: {
23    client: 'pg',
24    connection: process.env.DATABASE_URL,
25    migrations: {
26      directory: __dirname + '/db/migrations'
27    },
28    seeds: {
29      directory: __dirname + '/db/seeds/production'
30    }
31  }
32 }
```

knexfile.js hosted with ❤ by GitHub

[view raw](#)

An example of knex configuration file

Then you can create a module that exports the connection to the DB, I call it `knex.js`

```
1  var environment = process.env.NODE_ENV || 'development'
2  var config = require('../knexfile.js')[environment]
3
4  module.exports = require('knex')(config)
```

knex.js hosted with ❤ by GitHub

[view raw](#)

First we're getting in which environment we are, production servers usually export this variable and it can be found in `process.env.NODE_ENV`, otherwise we're in the development server. Next we're getting the relative configuration from `knexfile` defined before and then passing it to the library while exporting it, easy.

I usually place this file in a `db` folder, alongside migrations and seeds, which are the next topic.

Migrations

One of the big advantages of Knex is the migration tool which can make your life easier managing the tables of your DB. It creates sequential files with timestamps where you can define which table with which columns should be created; it manages also tables alterations, so you can add/drop columns without touching the original migration file.

If you installed Knex globally you can access the bin without entering the `.node_modules` folder, so if you type something like:

```
knex migrate:make users
```

it will create a new file in your migration folder exporting two functions, the first for the table creation/alteration, the second for reverting changes (if you need to). Knex is agnostic about which DB you're using (except for particular column types) so you can write the same columns with the same type independently if you're using PostgreSQL/MySQL/SQLite/[...]

A typical migration will look like this:

```
1
2 exports.up = function(knex, Promise) {
3   return knex.schema.createTable('users', t => {
4     t.increments('id')
5     t.string('username')
6     t.string('email')
7   })
8 };
9
10 exports.down = function(knex, Promise) {
11   return knex.schema.dropTable('users')
12 };
```

user_migration.js hosted with ❤ by GitHub

[view raw](#)

In the `up` function we're defining name and fields of the table, while in `down` we're saying that if we want to revert changes the table should be dropped.

Constraints are possible, like not null, uniqueness and similar but I won't cover them here, you can find them on knex website.

Now, if everything is configured correctly, running `knex migrate:latest` should create the table in your DB, hurray!

Seeds

In the same way as migrations we can create seeds, so we can populate the DB consistently. The command is similar to the migration one, we can type

```
knex seed:make users
```

And a standard seed file will be created in the seeds folder specified in the knexfile. Eventually it will look like this, with each entry defined as a JSON object with fields with the same names as those in the migration:

```
1
2  exports.seed = function(knex, Promise) {
3    // Deletes ALL existing entries
4    return knex('users').del()
5      .then(function () {
6        // Inserts seed entries
7        return knex('users').insert([
8          {username: 'fodark', email: 'mock@email.com'},
9          {username: 'john', email: 'mock2@email.com'},
10         {username: 'david', email: 'mock3@email.com'}
11       ]);
12     });
13  };
```

user_seed.js hosted with ❤ by GitHub

[view raw](#)

What does the function do? First it deletes every entry in the table and then the table gets populated with our data.

How to run it? Simply type `knex seed:run` and knex will do it!

Extra

In the repo you can find migration and seed file for the messages table, and you can also see how to make foreign keys in knex; this table will be used later to show Objection features.

Side note

Seed files are executed sequentially so if you have constraints like `author_id` in table `messages` you should ensure an order, I usually put a number in seed files, like `01_users.js` , `02_messages.js` and so on.

Objection.js

Objection is an ORM built upon Knex, and allows to define models for our DB, which we'll see later, and other features like eager loading, schema validation, etc...

We'll use it for models and eager loading, which allows us to not write complex join queries to get related rows between tables.

Models

The first thing we want to define is our models which represent the tables in our DB. I usually store them in `models` folder to organize my code.

The minimum structure required to use Objection models is this:

```
1  const { Model } = require('objection');
2
3  class User extends Model {
4    static get tableName() {
5      return 'users';
6    }
7  }
8
9  module.exports = User;
```

user_model_min.js hosted with ❤ by GitHub

[view raw](#)

which simply tells to Objection the name of the table in our DB.

But the cool stuff is eager loading, suppose we want to get every message written by an user, usually you'd write a query like this:

```
1  SELECT *
2  FROM messages
3  WHERE user_id=your_id
```

message_query.sql hosted with ❤ by GitHub

[view raw](#)

which is not really complex you may say, but suppose you'd want to have them inside the user object when responding to the user, it can become easily a mess!

This is where Objection models show their strength, we can easily define relation between tables and then use them to get subsets of rows related.

So, if we want to get the messages of a user we could define a relation like this:

```
1  const { Model } = require('objection');
2  const knex = require('../db/knex')
3
4  Model.knex(knex)
5
6  class User extends Model {
7    static get tableName() {
8      return 'users';
9    }
10
11    static get relationMappings() {
12      const Message = require('../Message')
13      return {
14        messages: {
15          relation: Model.HasManyRelation,
16          modelClass: Message,
17          join: {
18            from: 'users.id',
19            to: 'messages.user_id'
20          }
21        }
22      }
23    }
24  }
25
26  module.exports = User;
```

user_model_complete.js hosted with ❤ by GitHub

[view raw](#)

We're basically saying which relation there is between the two tables (one User has many Messages), which is the module that holds the other table and which fields make the relation possible.

Line 2–4 are binding the Model to the DB connection we defined in the previous section.

Analogous the Message model:

```
1  const { Model } = require('objection');
2  const knex = require('../db/knex')
```

```
3
4  Model.knex(knex)
5
6  class Message extends Model {
7    static get tableName() {
8      return 'messages';
9    }
10
11    static get relationMappings() {
12      const User = require('./User')
13      return {
14        writer: {
15          relation: Model.BelongsToOneRelation,
16          modelClass: User,
17          join: {
18            from: 'messages.user_id',
19            to: 'users.id'
20          }
21        }
22      }
23    }
24  }
25
26  module.exports = Message;
```

message_model_complete.js hosted with ❤ by GitHub

[view raw](#)

It's not mandatory to define the reverse relation but it could be useful to have it.

Yeah, now we have our database correctly setup and we have our models defined, last thing is setting up the API for these resources!

API

For this last part we'll be using Express, which is the most known framework for Node.

Basics

First, let's create a simple `app.js` file which launches a server listening on a port.

```
1  const express = require('express')
2  const port = process.env.PORT || 3000
3
4  const app = express()
5
6  app.listen(port, () => {
7    console.log('Listening on port: ' + port)
```



```
8  })
```

app_v1.js hosted with ❤ by GitHub

[view raw](#)

We're importing `express` and defining the port where the server should listen for incoming requests, then creating a new application where we'll attach our endpoints and eventually make it listen on the port.

Yay, if you start the program with `node app.js` you should see `Listening on port: 3000` or similar, nice!

Endpoints

Next we want to have some endpoints serving our resources, so let's create a folder `api` where we'll put our files, generally one per resource. For example in `users.js` we could write something like this to get every user in the DB:

```
1  const express = require('express')
2  const router = express.Router()
3
4  const User = require('../models/User')
5
6  router.get('/users', (req, res) => {
7    User.query()
8      .then(users => {
9        res.json(users)
10      })
11  })
12
13 module.exports = {
14   router: router
15 }
```

users_api_v1.js hosted with ❤ by GitHub

[view raw](#)

So, what's going on? We're using Express router to answer to `GET` requests on `/users`, and we're using the Objection model to easily retrieve every row in the table, the JSON object returning from Objection can be served directly to the user.

To test it we need one more thing, to include the endpoints in our app, our `app.js` will look like this:

```
1  const express = require('express')
2  const port = process.env.PORT || 3000
```

```
3
4  const app = express()
5
6  // Endpoints
7  app.use('/api', require('./api/users').router)
8
9  app.listen(port, () => {
10     console.log('Listening on port: ' + port)
11  })
```

app_v2.js hosted with ❤ by GitHub

[view raw](#)

Good, we're saying: attach to the base url `/api` the routes defined in `/api/users` ,
launch `node app.js` and point your browser to

`localhost:3000/api/users`

You'll see the JSON with the users, hurray again!

Eager loading

So far so good, but we haven't used yet the eager loading we talked about before, let's suppose we want to setup an endpoint to get a single user, and when we get a single user we want to see even his messages: normally you'd set a complex union query, joining the user with the messages but with Objection we need only two more lines! Let's see how:

```
1  const express = require('express')
2  const router = express.Router()
3
4  const User = require('../models/User')
5
6  router.get('/users', (req, res) => {
7     User.query()
8       .then(users => {
9         res.json(users)
10      })
11  })
12
13  router.get('/users/:id', (req, res) => {
14     let id = parseInt(req.params.id)
15     User.query()
16       .where('id', id)
17       .eager('messages')
18       .then(user => {
19         res.json(user)
20      })
21  })
```

```
17         res.json(user)
20     })
21 })
22
23 module.exports = {
24     router: router
25 }
```

users_api_v2.js hosted with ❤ by GitHub

[view raw](#)

Yeah! First we're filtering with the `where` clause with the ID passed in the URL, and then, with the keyword `eager` we're using the relation name defined in the Model to get the user's messages, give it a try, point to:

`localhost:3000/api/users/1`

And you'll see a sub-field messages with the messages' objects, easy!

The End (?)

We're done, but is it really finished? Not really, there are no controls on the ID passed to the URL, and we cannot handle the other 3 actions (POST, PUT and DELETE) but this would have been too much for this article.

This example could be expanded with more tables, endpoints, tests and much more. Objection also allows to filter out fields when using eager loading, so if you have fields like `password` that obviously you don't want to return to the user you can remove them from the query.

Last cool thing I'll tell about Objection is the possibility to chain multiple eager loading together.

Let's suppose we have pictures attached to users, and comments to messages, you could easily retrieve everything with a single eager loading like this:


```
1 User.query()
2   .where('id', id)
3   .eager(['pictures, messages.comments'])
```

user_eager_big.js hosted with ❤ by GitHub

[view raw](#)

Okay, now it is really over!

I hope this overview is useful to get your head around DB management in Node! Clap if you liked it and share!

Some rights reserved 

[Nodejs](#) [Expressjs](#) [Postgres](#) [Database](#) [API](#)

[About](#) [Help](#) [Legal](#)