search

Objection + Knex = Painless PostgreSQL in your Node App

Ali Spittel Feb 20 '18 · 9 min read

#node #backend #postgres #showdev

It's no secret that I'm a total PostgreSQL fangirl -- I rarely see a use case for using a different database, especially with the support for array and JSON fields. I also love Node and Express for simple APIs (without auth). In the past, SQL support within Node and Express hasn't been perfect. I've been hearing great things about Objection, so I decided to try it out!

Objection, which is built on top of Knex, uses the new ES7 class features to build a nice ORM query language for Node. ORMs allow you to use whatever programming language you are using for your app to query a database rather than querying in the natie language of the database (here we will use JavaScript to interact with our database instead of SQL). Since Objection is still really new, I will be walking through all of my code step by step.

The Learning Process

For this project, I relied pretty much exclusively on the documentation. The Knex documentation was great, and there were examples on the Objection GitHub that were very

comfortable continuing with the project after skimming these resources.

The Final Project

I've been having trouble coming up with app ideas for this blog! So, I built an app idea app! The models were relatively simple: `ideas` and `comments`, but they still demonstrate one of the biggest use cases for Objection: relations between data. The `ideas` will be the "parents" with "child" comments attached to them. Essentially, users will be able to comment on various app ideas.

Knex Initialization

First, I initialized Knex, which will facilitate our database connection using `pg`, our migrations, and our seeds. After setting up my typical Express API boilerplate in my index file and installing the requirements in my package.json, I ran `knex init` in the root of my project. This created a `knexfile.js` that contains a boilerplate with example connections to databases. I decided to remove the `production`, `development`, and `staging` options in favor of just specifying a database connection string in my `.env` file. The `knexfile` ended up looking like:

```
require('dotenv').config()

const pg = require('pg')
pg.defaults.ssl = true
```

```
}
```

The `ssl` configuration is only necessary if you are using a database on Heroku or another provider that requires an SSL connection. `dotenv` allows us to retrieve environmental variables from a `.env` file! That variable is a standard PostgreSQL connection string:

```
DATABASE_URL=postgres://username:password@host:port/db_name
```

I created the database on my computer using `psql`, I created the production database using a Heroku add-on.

Migrations

Migrations are changes to a database's schema specified within your ORM, so we will be defining the tables and columns of our database straight in JavaScript rather than using SQL.

From there, I generated my migrations:

```
$ knex migrate:make create_ideas
$ knex migrate:make create_comments
```

Each migrate command created its own separate file in the

migrations/20180218215453_create_ideas.js. I created two separate migrations to keep things organized, and because I created the comments after the ideas. These could be combined, though.

The migration is generated with:

```
exports.up = function (knex, Promise) {
}

exports.down = function (knex, Promise) {
}
```

The migration itself goes within the body of the `exports.up` function and then whatever the opposite of that migration does goes within `exports.down`. The `exports.down` allows us to undo migrations that we no longer want. For the `create_ideas` migration, I added the following:

```
exports.up = function (knex, Promise) {
  return Promise.all([
    knex.schema.createTable('ideas', table => {
      table.increments('id').primary()
      table.string('idea')
      table.string('creator')
    })
  ])
}

exports.down = function (knex, Promise) {
  return Promise.all([
    knex.schema.dropTable('ideas')
```

Knex migration functions should "always return a promise" according to its documentation. We can use `Promise.all()` in order to return an array of promises to resolve. Even though each function only has one action in this case, I could have added more actions separated by `,`'s. The `exports.up` contains the table creation logic for the `ideas` table, including a primary key that is auto-incremented `table.increments('id').primary()`. It also has two other string columns called `idea` and `creator`. To undo the migration, we would drop the `ideas` table, as specified in the `exports.down` function.

The second migration to create the `comments` file is similar:

```
exports.up = function (knex, Promise) {
  return Promise.all([
    knex.schema.createTable('comments', table => {
      table.increments('id').primary()
      table.string('comment')
      table.string('creator')
      table.integer('ideas_id').references('ideas.id')
    })
  ])
}

exports.down = function (knex, Promise) {
  return Promise.all([
    knex.schema.dropTable('comments')
  ])
}
```

many ways to do this specified in the documentation; however, the Objection documentation does it this way so I did as well. Knex enforced the column name `ideas_id` rather than `idea_id` which was unsemantic. I am sure there is a way around that naming mandate; however, I didn't put much effort into looking it up!

Finally, I ran the migrations using the command:

```
$ knex migrate:latest
```

Even though the command implies it runs only the latest migration, it instead runs all migrations that haven't been run yet.

Database Seeding

Knex also has some built-in functionality to help us seed, or add initial test data, to our database.

```
$ knex seed:make ideas
```

The above command created a `seeds` directory with an `ideas.js` file within it. That file also had the following code in it:

```
exports.seed = function (knex, Promise) {
}
```

I added the following:

```
exports.seed = function (knex, Promise) {
  return knex('ideas').del().then(() => {
    return knex('ideas').insert([
        {creator: 'Ali', idea: 'A To Do List app!'},
        {creator: 'Ali', idea: 'A Blog!'},
        {creator: 'Ali', idea: 'A calculator'}
    ])
  })
}
```

This cleared the `ideas` table, so there wasn't any data in the table, and then it inserted three records into the database. It used the JSON keys and values to create those rows. I only seeded the `ideas` table, but you could definitely seed the `comments` table as well!

I then ran the following command to update the database:

```
$ knex seed:run
```

Models

Up until this point, we have been using Knex to interact with

created a `models` folder with a `schema.js` file within it. You could structure this pretty much anyway -- one good way would be to have each model in a different file. I kept everything together, though, for demonstration's sake!

First, let's take care of some administrative things at the top:

```
const Knex = require('knex')
const connection = require('../knexfile')
const { Model } = require('objection')

const knexConnection = Knex(connection)

Model.knex(knexConnection)
```

These lines of code connect us to the database using our `knexfile` from earlier. We are also attaching Objection to our database connection.

Now, let's create our model for our `Comment` data. The models will allow us to interact cleanly with the data we are retrieving from our database.

```
class Comment extends Model {
  static get tableName () {
    return 'comments'
  }

  static get relationMappings () {
    return {
      idea: {
```

```
          to: 'ideas.id'
        }
      }
    }
  }
}
```

Let's break this down. The static getter method `tableName` returns the name `comments` : the name of the database table we want our `Comment` class to model! We also have a second static getter method that defines the `Comment` model's relationships to other models. In this case, the key of the outside object `idea` is how we will refer to the parent class. The `relation` key within the child object has the value `Model.BelongsToOneRelation` which says that each comment is going to have one parent idea. The `modelClass` says that the `idea` is coming from the `Idea` model and then the `join` specifies the database table and column names to perform a SQL join on, in this case, the `ideas_id` column in the `comments` table to the `id` column in the `ideas` table. static and get were added in ES6!

The Idea class looks almost identical, though the relationships are inverted!

```
class Idea extends Model {
  static get tableName () {
    return 'ideas'
  }

  static get relationMappings () {
```

```
        join: {
          from: 'ideas.id',
          to: 'comments.ideas_id'
        }
      }
    }
  }
}

module.exports = { Idea, Comment }
```

In this case, our relationship is `Model.HasManyRelation` since one idea can have multiple comments! I also exported the models so they could be used within our other files.

Querying

The final file I worked with was `controllers/ideas.js` . I usually separate all my "controller" functions -- the routing functions that decide what each route renders -- into a file or files if there are lots of them! This week, I built an API that I will build a front-end for in the future.

First, some imports:

```
const express = require('express')

const { Idea, Comment } = require('../models/schema')

const router = express.Router()
```

all of the `ideas`:

```
router.get('/', async (req, res) => {
  const ideas = await Idea.query()
  res.json(ideas)
})
```

In the above example, we are making the arrow function callback that handles the request and response asynchronous using `async`, then we can "pause" the body of the function until the promise from our `Idea.query()` resolves. That query will return a JavaScript object with all of the items in our `ideas` table using our `res.json(ideas)` method. If we navigate to `localhost:3000/ideas` locally or `https://application-ideas.herokuapp.com/ideas` in production we see:

```
[
    {
        "id": 1,
        "idea": "A To Do List app!",
        "creator": "Ali"
    },
    {
        "id": 2,
        "idea": "A Blog!",
        "creator": "Ali"
    },
    {
        "id": 3,
        "idea": "A calculator",
        "creator": "Ali"
```

Note: The Objection documentation uses async and await to handle promises in JavaScript; however, we could rewrite the above function to look like the following and that would work equally as well!

```
router.get('/', (req, res) => {
  Idea.query().then(ideas => {
    res.json(ideas)
  })
})
```

Instead of going through the other routes in paragraph form, I am going to put the annotated code below:

```
router.get('/:id', async (req, res) => {
  // gets one idea, found by id.
  //Also fetches the related comments using the .eager method
  const idea = await Idea.query().findById(req.params.id).eager('comments')
  res.json(idea)
})

router.post('/', async (req, res) => {
  // creates a new idea from the request body
  // only allows the idea and creator fields for safety
  const newIdea = req.body

  const idea = await Idea.query()
                         .allowInsert('[idea, creator]')
                         .insert(newIdea)

  res.send(idea)
})

router.post('/:id/comments', async (req, res) => {
```

```
            .allowInsert('[comment, creator]')
            .insert(req.body)

  res.send(idea)
})

router.delete('/:id', async (req, res) => {
  // deletes an idea
  await Idea.query().deleteById(req.params.id)

  res.redirect('/ideas')
})

router.delete('/:id/comments/:commentId', async (req, res) => {
  // deletes a comment
  await Comment.query().deleteById(req.params.commentId)

  res.redirect(`/ideas/${req.params.id}`)
})

module.exports = router
```

There's a bunch more you can do with Objection, like raw queries, interaction with JSON fields, and validations.

Next Steps

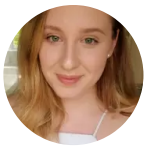
I had a really fun time working with Objection and Knex! It is honestly very similar to working with Mongoose and MongoDB from a configuration standpoint, but it makes hierarchical and related data so much easier to work with! I would definitely keep using these libraries in the future with Express apps! Definitely a must-try if you use Node frequently!

search

comment or tweet me with suggestions for a front-end tool to use for it!

Full Code
Deployed App
Objection Documentation
Knex Documentation

Part of my On Learning New Things Series



Ali Spittel + FOLLOW
Passionate about education, Python, JavaScript, and code art.
@aspittel ASpittel aspittel www.alispit.tel/links

Add to the discussion

PREVIEW SUBMIT

Sami Koskimäki

Apr 3 '18

Great post, thanks for writing it! You have a small error in the usage of `allowInsert` method though. It can only be used to limit relations when inserting/upserting graphs using `insertGraph` and `upsertGraph` methods. It has no effect with the normal insert method.

3 REPLY

Daniel HB

Feb 21 '18

How does Knex compare to Sequelize?

53 16 38

search

Like apples and oranges, basically. Sequelize, like its peers Objection and Bookshelf, is a full-fledged O/RM which gives you a well-defined, database-agnostic API at the cost of maintaining parallel data models, somewhat restricted flexibility (the stateful entity metaphor O/RMs encourage is a very limited view of what your database can actually do), lowest-common-denominator feature sets, and winding up writing SQL anyway when things get complicated, which they always do. As @rhymes mentioned below, I run a competing project with a different approach to data access, so take my opinion for what it's worth :)

Knex, meanwhile, is a query builder which generates SQL statements. It's lower-level and doesn't care about your data model; it's strictly a way to build SQL in JavaScript.

4 REPLY

Sami Koskimäki

Nov 25 '18

Based on this comment, I'm guessing you haven't actually used objection. The main design goal of objection is to NOT restrict flexibility or limit access to DB features.

2 THREAD

Dian Fay

Nov 25 '18

I have to admit I have not; the last time I had to search for a new data access framework for Node was a matter of days before you released 0.1.0 :) I did at least do the bare minimum of reading your documentation and looking at examples to see how it addressed the fundamental weaknesses of object-relational mapping before I mentioned it, though!

I consider models inherently limiting. To be completely fair to Objection, a minimally functional model is maybe half a dozen lines long, and `patch` takes care of one of the major restrictions of the active record pattern, but recapitulating your database architecture before you can use it necessarily adds a structural rigidity (if badly done, it can be outright brittle) and a certain inertia. This isn't something that can be laid at your or anyone else's doorstep; it's the price of working with record graphs in application code, as implicit in the object-relational mapper pattern. You can argue that it is or can be worth it, but that doesn't mean you don't have to pay it.

3 THREAD

Sami Koskimäki

Apr 11

First of all, objection is built on knex and every possible operation you can start with it

53 16 38

denominator.

Your other point was that models are limiting. Let's test that theory. Let's say you have this model in objection

```
class Person extends Model {
  static tableName = 'persons'
}
```

Then you have a bunch of queries all over your database

```
const { Person }  = require('./models/person')

...

Person.query().select('...').where('...')
```

How is that any different from using a query builder? Instead of mentioning the name of the table in the query, you use the model. The table name and the model class reference are just as easy to change.

Let's say the name of the table changes. All you need to do is to change the `tableName` property. What if the columns names change? You change the queries, just like you would with a query builder. What if you want to change the model name? Well, search and replace the model name. How about relations? Again, change the model. If you use a plain query builder, you need to update the foreign keys in million places instead.

Using objection's models adds no overhead on top of using a query builder whatsoever. If anything, it removes a great deal of it.

I really don't see what you mean here. You still seem to be talking about your experiences with other ORMs.

2 THREAD

krckyboy

May 30

I have to say that Objection.js has by far the best docs that I've ever seen so far. Good job!

53 16 38

search

This is really cool! I actually use Postgres/Knex/Objection at work in production, and have for about a year and a half and it's an amazing tool. One of my favorite features of Objection is the use of the `get` and `set` in the class which you mentioned.

Combined with `lodash`, it's really easy to do something like:

```
res.send(_.pick(post, [
  ...post,
  computedProperty,
  computedProperty2,
  whateverElse
]);
```

I wanted to ask, why did you end up trying out Objection over, let's say, Bookshelf which was built by the same author as Knex and at around the same time as Knex? We migrated from Bookshelf to Objection so I think Objection is the right choice, just wondering though.

1 REPLY

Ali Spittel

Mar 2 '18

Ah awesome! I had a friend recommend Objection, so that's why I went with that one! I hadn't heard great things about Bookshelf.

2 REPLY

rhymes

Feb 20 '18

Now that you tested an ORM library I guess you have to test @dmfay Massive.js based on the Data Mapper pattern as well :-D

Great post, as usual!

3 REPLY

Ali Spittel

Mar 2 '18

Ah! Will look into that!

REPLY

53 16 38

search

Perfect timing! I like to add ORM to my node apps in the next days as well. Thanks

3 REPLY

 Gleb Bahmutov Feb 20 '18

I highly recommend using "knex-migrate" github.com/sheerun/knex-migrate#re... to do Knex migrations - a very nice tool with better commands for running / rolling / seeing migrations.

2 REPLY

 Ali Spittel Mar 2 '18

Oh cool! Didn't know this existed!

1 REPLY

 Belhassen Chelbi Feb 20 '18

loved it

2 REPLY

 Manuel Romero Feb 21 '18

Great post, Ali!

2 REPLY

 Sholto Maud Feb 21 '18

Nice Post Ali,

Do you have a link to how best integrate "auth"?

REPLY

53 16 38

search

Ah that would be awesome! I looked a little bit but didn't see anything. I think you could use Passport, though (kinda like this: code.tutsplus.com/tutorials/using-...)

2 REPLY


Arlei F. Farnetani Junior

Nov 6 '18

Excellent post! Thanks a lot!

2 REPLY


MrScX

Aug 10

BTW in exports.up and exports.down you pass in Promise as an argument. That throws an error. Removing that fixes it. Also, knex migrate:make doesn't pass that argument in its auto-generated file.

1 REPLY


Lazarus Lazaridis

Mar 7 '18

Great post, thank you!

1 REPLY

code of conduct - report abuse

Classic DEV Post from Jul 26

JavaScript Enhanced Scss mixins! concepts explained


Adam Crockett

65 7

53 16 38

search

Wizards Use Vim! My New Book on Vim

 Jaime González

Announcing my new book, giving some insider info and sharing a bunch of free versions of the book

144 24

Another Post You Might Like

Ultra List: One List to Rule Them All. March, 19

 Sarthak Sharma

This is a new series my team and I are starting on DEV.to in which we will be showcasing, at the end of every month, a huge list of resources that we find on the web.

729 27

6 points you need to know about async/await in JavaScript

Yaser Adel Mehraban - Aug 18

Nestjs, External EventBus

Sergey Telpuk - Aug 18

Vanilla JS equivalent for counting number of child elements using className

Paramanantham Harrison - Aug 18

Objects for mapping

53 16 38

search

Home About Privacy Policy Terms of Use Contact Code of Conduct

53 16 38