Module 17 Challenge

MongoDB is a popular choice for many social networks due to its speed with large amounts of data and flexibility with unstructured data. Over the last part of this course, you'll use several of the technologies that social networking platforms use in their full-stack applications. Because the foundation of these applications is data, it's important that you understand how to build and structure the API first.

Social Network API

Your Challenge is to build an API from scratch for a social network web application where users can share their thoughts, react to friends' thoughts, and create a friend list. You'll use Express.js for routing, a MongoDB database, and the Mongoose ODM. In addition to using the Express.js and Mongoose packages, you may also optionally use a JavaScript date library of your choice or the native JavaScript `Date` object to format timestamps.

Because this application won't be deployed, you'll also need to create a walkthrough video that demonstrates its functionality and all of the following acceptance criteria being met. You'll need to submit a link to the video and add it to the README of your project.

User Story

```
AS A social media startup
I WANT an API for my social network that uses a NoSQL database
SO THAT my website can handle large amounts of unstructured data
```

Acceptance Criteria

```
GIVEN a social network API
WHEN I enter the command to invoke the application
THEN my server is started and the Mongoose models are synced to the MongoDB database
WHEN I open API GET routes in Insomnia for users and thoughts
THEN the data for each of these routes is displayed in a formatted JSON
WHEN I test API POST, PUT, and DELETE routes in Insomnia
THEN I am able to successfully create, update, and delete users and thoughts in my database
WHEN I test API POST and DELETE routes in Insomnia
THEN I am able to successfully create and delete reactions to thoughts and add and remove friends to a user's friend list
```

Your walkthrough video when you submit should contain the following:

- GET routes to return all users and all thoughts being tested in Insomnia
- GET routes to return a single user and a single thought being tested in Insomnia
- the POST, PUT, and DELETE routes for users being tested in Insomnia
- the POST, PUT, and DELETE routes for thoughts being tested in Insomnia
- the POST and DELETE routes for a user's friend list being tested in Insomnia
- the POST and DELETE routes for reactions to thoughts being tested in Insomnia.

Getting Started

Use the following guidelines to set up your models and API routes:

Models

User

- `username`
    - String
    - Unique
    - Required

    - Trimmed
- `email`
    - String
    - Required
    - Unique
    - Must match a valid email address (look into Mongoose's matching validation)
- `thoughts`
    - Array of `_id` values referencing the `Thought` model
- `friends`
    - Array of `_id` values referencing the `User` model (self-reference)

Schema Settings

Create a virtual called `friendCount` that retrieves the length of the user's `friends` array field on query.

Thought

- `thoughtText`
    - String
    - Required
    - Must be between 1 and 280 characters
- `createdAt`
    - Date
    - Set default value to the current timestamp
    - Use a getter method to format the timestamp on query
- `username` (The user that created this thought)
    - String
    - Required
- `reactions` (These are like replies)
    - Array of nested documents created with the `reactionSchema`

Schema Settings

Create a virtual called `reactionCount` that retrieves the length of the thought's `reactions` array field on query.

Reaction (SCHEMA ONLY)

- `reactionId`
  - Use Mongoose's ObjectId data type
  - Default value is set to a new ObjectId
- `reactionBody`
  - String
  - Required
  - 280 character maximum
- `username`
  - String
  - Required
- `createdAt`
  - Date
  - Set default value to the current timestamp
  - Use a getter method to format the timestamp on query

Schema Settings

This will not be a model, but rather will be used as the `reaction` field's subdocument schema in the `Thought` model.

API Routes

`/api/users`

- `GET` all users
- `GET` a single user by its `_id` and populated thought and friend data
- `POST` a new user (note that the examples below are just sample data):

```
{
    "username": "lernantino",
    "email": "lernantino@gmail.com"
}
```

-
- `PUT` to update a user by its `_id`
- `DELETE` to remove user by its `_id`

BONUS: Remove a user's associated thoughts when deleted.

`/api/users/:userId/friends/:friendId`

- `POST` to add a new friend to a user's friend list
- `DELETE` to remove a friend from a user's friend list

`/api/thoughts`

- `GET` to get all thoughts
- `GET` to get a single thought by its `_id`

`POST` to create a new thought. Don't forget to push the created thought's `_id` to the associated user's `thoughts` array field. (note that the examples below are just sample data):

```
{
    "thoughtText": "Here's a cool thought...",
    "username": "lernantino",
    "userId": "5edff358a0fcb779aa7b118b"
}
```

-
- `PUT` to update a thought by its `_id`
- `DELETE` to remove a thought by its `_id`

`/api/thoughts/:thoughtId/reactions`

- `POST` to create a reaction stored in a single thought's `reactions` array field
- `DELETE` to pull and remove a reaction by the reaction's `reactionId` value

---

Grading Requirements

note

If a Challenge assignment submission is marked as “0”, it is considered incomplete and will not count towards your graduation requirements. Examples of incomplete submissions include the following:

- A repository that has no code
- A repository that includes a unique name but nothing else
- A repository that includes only a README file but nothing else
- A repository that only includes starter code

This Challenge is graded based on the following criteria:

Deliverables: 10%

- Your GitHub repository containing your application code.

Walkthrough Video: 37%

- A walkthrough video that demonstrates the functionality of the social media API must be submitted, and a link to the video should be included in your README file.

- The walkthrough video must show all of the technical acceptance criteria being met.
- The walkthrough video must demonstrate how to start the application's server.
- The walkthrough video must demonstrate GET routes for all users and all thoughts being tested in Insomnia.
- The walkthrough video must demonstrate GET routes for a single user and a single thought being tested in Insomnia.
- The walkthrough video must demonstrate POST, PUT, and DELETE routes for users and thoughts being tested in Insomnia.
- Walkthrough video must demonstrate POST and DELETE routes for a user's friend list being tested in Insomnia.
- Walkthrough video must demonstrate POST and DELETE routes for reactions to thoughts being tested in Insomnia.

Technical Acceptance Criteria: 40%

- Satisfies all of the preceding acceptance criteria plus the following:
  - Uses the Mongoose package
  - Links to an external site.
  - to connect to a MongoDB database.
  - Includes User and Thought models outlined in the Challenge instructions.
  - Includes schema settings for User and Thought models as outlined in the Challenge instructions.
  - Includes Reactions as the `reaction` field's subdocument schema in the Thought model.
  - Uses functionality to format queried timestamps properly.

Repository Quality: 13%

- Repository has a unique name.

- Repository follows best practices for file structure and naming conventions.
- Repository follows best practices for class/id naming conventions, indentation, quality comments, etc.
- Repository contains multiple descriptive commit messages.
- Repository contains a high-quality README with description and a link to a walkthrough video.

Bonus

Fulfilling the following can add 10 points to your grade. Note that the highest grade you can achieve is still 100:

- Application deletes a user's associated thoughts when the user is deleted.

Express Guide:

Express

Fast, unopinionated, minimalist web framework for Node.js.

```
import express from 'express'

const app = express()

app.get('/', (req, res) => {
  res.send('Hello World')
})

app.listen(3000)
```

Installation

This is a Node.js module available through the npm registry.

Before installing, download and install Node.js. Node.js 18 or higher is required.

If this is a brand new project, make sure to create a `package.json` first with the `npm init` command.

Installation is done using the `npm install` command:

```
npm install express
```

Follow our installing guide for more information.

Features

- Robust routing
- Focus on high performance
- Super-high test coverage
- HTTP helpers (redirection, caching, etc)
- View system supporting 14+ template engines
- Content negotiation
- Executable for generating applications quickly

Docs & Community

- Website and Documentation - [website repo]
- GitHub Organization for Official Middleware & Modules
- Github Discussions for discussion on the development and usage of Express

PROTIP Be sure to read the migration guide to v5

Quick Start

The quickest way to get started with express is to utilize the executable `express(1)` to generate an application as shown below:

Install the executable. The executable's major version will match Express's:

```
npm install -g express-generator@4
```

Create the app:

```
express /tmp/foo && cd /tmp/foo
```

Install dependencies:

```
npm install
```

Start the server:

```
npm start
```

View the website at: http://localhost:3000

Philosophy

The Express philosophy is to provide small, robust tooling for HTTP servers, making it a great solution for single page applications, websites, hybrids, or public HTTP APIs.

Express does not force you to use any specific ORM or template engine. With support for over 14 template engines via @ladjs/consolidate, you can quickly craft your perfect framework.

Examples

To view the examples, clone the Express repository:

```
git clone https://github.com/expressjs/express.git --depth 1 && cd express
```

Then install the dependencies:

```
npm install
```

Then run whichever example you want:

```
node examples/content-negotiation
```

Running Tests

To run the test suite, first install the dependencies:

```
npm install
```

Then run `npm test`:

```
npm test
```

Mongoose Guide:

Mongoose

Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment. Mongoose supports Node.js and Deno (alpha).

Documentation

The official documentation website is mongoosejs.com.

Mongoose 8.0.0 was released on October 31, 2023. You can find more details on backwards breaking changes in 8.0.0 on our docs site.

Plugins

Check out the plugins search site to see hundreds of related modules from the community. Next, learn how to write your own plugin from the docs or this blog post.

Contributors

Pull requests are always welcome! Please base pull requests against the `master` branch and follow the contributing guide.

If your pull requests makes documentation changes, please do not modify any `.html` files. The `.html` files are compiled code, so please make your changes in `docs/*.pug`, `lib/*.js`, or `test/docs/*.js`.

View all 400+ contributors.

Installation

First install Node.js and MongoDB. Then:

```
npm install mongoose
```

Mongoose 6.8.0 also includes alpha support for Deno.

Importing

```
// Using Node.js `require()`
const mongoose = require('mongoose');

// Using ES6 imports
import mongoose from 'mongoose';
```

Or, using Deno's `createRequire()` for CommonJS support as follows.

```
import { createRequire } from 'https://deno.land/std@0.177.0/node/module.ts';
const require = createRequire(import.meta.url);

const mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/test')
  .then(() => console.log('Connected!'));
```

You can then run the above script using the following.

```
deno run --allow-net --allow-read --allow-sys --allow-env mongoose-test.js
```

Mongoose for Enterprise

Available as part of the Tidelift Subscription

The maintainers of mongoose and thousands of other packages are working with Tidelift to deliver commercial support and maintenance for the open source dependencies you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact dependencies you use. Learn more.

Overview

Connecting to MongoDB

First, we need to define a connection. If your app uses only one database, you should use `mongoose.connect`. If you need to create additional connections, use `mongoose.createConnection`.

Both `connect` and `createConnection` take a `mongodb://` URI, or the parameters `host, database, port, options`.

```
await mongoose.connect('mongodb://127.0.0.1/my_database');
```

Once connected, the `open` event is fired on the `Connection` instance. If you're using `mongoose.connect`, the `Connection` is `mongoose.connection`. Otherwise, `mongoose.createConnection` return value is a `Connection`.

Note: *If the local connection fails then try using 127.0.0.1 instead of localhost. Sometimes issues may arise when the local hostname has been changed.*

Important! Mongoose buffers all the commands until it's connected to the database. This means that you don't have to wait until it connects to MongoDB in order to define models, run queries, etc.

Defining a Model

Models are defined through the `Schema` interface.

```
const Schema = mongoose.Schema;
const ObjectId = Schema.ObjectId;

const BlogPost = new Schema({
  author: ObjectId,
  title: String,
  body: String,
  date: Date
});
```

Aside from defining the structure of your documents and the types of data you're storing, a Schema handles the definition of:

- Validators (async and sync)
- Defaults
- Getters
- Setters
- Indexes
- Middleware
- Methods definition
- Statics definition
- Plugins
- pseudo-JOINs

The following example shows some of these features:

```
const Comment = new Schema({
  name: { type: String, default: 'hahaha' },
  age: { type: Number, min: 18, index: true },
  bio: { type: String, match: /[a-z]/ },
  date: { type: Date, default: Date.now },
  buff: Buffer
});

// a setter
Comment.path('name').set(function(v) {
  return capitalize(v);
});

// middleware
Comment.pre('save', function(next) {
  notify(this.get('email'));
  next();
});
```

Take a look at the example in `examples/schema/schema.js` for an end-to-end example of a typical setup.

Accessing a Model

Once we define a model through `mongoose.model('ModelName', mySchema)`, we can access it through the same function

```
const MyModel = mongoose.model('ModelName');
```

Or just do it all at once

```
const MyModel = mongoose.model('ModelName', mySchema);
```

The first argument is the *singular* name of the collection your model is for. Mongoose automatically looks for the *plural* version of your model name. For example, if you use

```
const MyModel = mongoose.model('Ticket', mySchema);
```

Then `MyModel` will use the tickets collection, not the ticket collection. For more details read the model docs.

Once we have our model, we can then instantiate it, and save it:

```
const instance = new MyModel();
instance.my.key = 'hello';
await instance.save();
```

Or we can find documents from the same collection

```
await MyModel.find({});
```

You can also `findOne`, `findById`, `update`, etc.

```
const instance = await MyModel.findOne({ /* ... */ });
console.log(instance.my.key); // 'hello'
```

For more details check out the docs.

Important! If you opened a separate connection using `mongoose.createConnection()` but attempt to access the model through `mongoose.model('ModelName')` it will not work as expected since it is not hooked up to an active db connection. In this case access your model through the connection you created:

```
const conn = mongoose.createConnection('your connection string');
const MyModel = conn.model('ModelName', schema);
const m = new MyModel();
await m.save(); // works
```

vs

```
const conn = mongoose.createConnection('your connection string');
const MyModel = mongoose.model('ModelName', schema);
const m = new MyModel();
await m.save(); // does not work b/c the default connection object was never
connected
```

Embedded Documents

In the first example snippet, we defined a key in the Schema that looks like:

```
comments: [Comment]
```

Where `Comment` is a `Schema` we created. This means that creating embedded documents is as simple as:

```
// retrieve my model
const BlogPost = mongoose.model('BlogPost');

// create a blog post
const post = new BlogPost();

// create a comment
post.comments.push({ title: 'My comment' });

await post.save();
```

The same goes for removing them:

```
const post = await BlogPost.findById(myId);
post.comments[0].deleteOne();
await post.save();
```

Embedded documents enjoy all the same features as your models. Defaults, validators, middleware.

Middleware

See the docs page.

Intercepting and mutating method arguments

You can intercept method arguments via middleware.

For example, this would allow you to broadcast changes about your Documents every time someone `set`s a path in your Document to a new value:

```
schema.pre('set', function(next, path, val, typel) {
  // `this` is the current Document
  this.emit('set', path, val);

  // Pass control to the next pre
  next();
});
```

Moreover, you can mutate the incoming `method` arguments so that subsequent middleware see different values for those arguments. To do so, just pass the new values to `next`:

```
schema.pre(method, function firstPre(next, methodArg1, methodArg2) {
  // Mutate methodArg1
  next('altered-' + methodArg1.toString(), methodArg2);
});

// pre declaration is chainable
schema.pre(method, function secondPre(next, methodArg1, methodArg2) {
  console.log(methodArg1);
  // => 'altered-originalValOfMethodArg1'

  console.log(methodArg2);
  // => 'originalValOfMethodArg2'

  // Passing no arguments to `next` automatically passes along the current argument values
  // i.e., the following `next()` is equivalent to `next(methodArg1, methodArg2)`
  // and also equivalent to, with the example method arg
  // values, `next('altered-originalValOfMethodArg1', 'originalValOfMethodArg2')`
  next();
});
```

Schema gotcha

`type`, when used in a schema has special meaning within Mongoose. If your schema requires using `type` as a nested property you must use object notation:

```
new Schema({
  broken: { type: Boolean },
  asset: {
    name: String,
    type: String // uh oh, it broke. asset will be interpreted as String
  }
});

new Schema({
  works: { type: Boolean },
  asset: {
    name: String,
    type: { type: String } // works. asset is an object with a type property
  }
});
```

Driver Access

Mongoose is built on top of the official MongoDB Node.js driver. Each mongoose model keeps a reference to a native MongoDB driver collection. The collection object can be accessed using `YourModel.collection`. However, using the collection object directly bypasses all mongoose features,

including hooks, validation, etc. The one notable exception that `YourModel.collection` still buffers commands. As such, `YourModel.collection.find()` will not return a cursor.