Creating a REST API with Express.js and MongoDB

APRIL 27, 2020 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter 17k Follow on Facebook

This tutorial is part 5 of 5 in this series.

Part 1: The minimal Node.js with Babel Setup
Part 2: How to setup Express.js in Node.js
Part 3: How to create a REST API with Express.js in Node.js
Part 4: Setup MongoDB with Mongoose in Express

Node + Express + MongoDB is a powerful tech stack for backend applications to offer CRUD operations. It gives you everything to expose an API (Express routes), to add business logic (Express middleware and logic within Express routes), and to use real data with a database (MongoDB). It's perfect for establishing a MERN (MongoDB, Express, React, Node), MEAN (MongoDB, Express, Angular, Node), or MEVN (MongoDB, Express, Vue, Node) tech stack. Everything that would be missing is the frontend

application with React, Angular, Vue or something else. But that's up to another section.

This section focuses first on connecting MongoDB to Express for our REST API. Previously we have set up MongoDB in our Express.js application and seeded the database with initial data, but didn't use it in Express for the RESTful API yet. Now we want to make sure that every CRUD operation going through this REST API reads or writes from/to the MongoDB database rather than using sample data as we did before for our Express routes. That's why we need to wire our Express routes to MongoDB via Mongoose to marry both worlds.

In our *src/index.js* where we set up and start the Express application with the MongoDB database, we already have a Express middleware in place which passes the models as context to all of our Express routes. Previously, these models have been sample data. Now we are using the Mongoose models that connect us to the MongoDB database. Since the folder/file data structure is the same as before, nothing changes for passing the models as context to the Express routes.

```
...

import models from './models';

const app = express();

...

app.use((req, res, next) => {
  req.context = {
    models,
    me: models.users[1],
  };
  next();
});

...
```

However, the me user (authenticated user) can be retrieved from the seeded data from the database. There is no `users` array available anymore as sample data on the models object, because the models are our interface to the MongoDB database now.

```
...

import models from './models';

const app = express();

...

app.use(async (req, res, next) => {
```

```
  req.context = {
    models,

    me: await models.User.findByLogin('rwieruch'),
  };
  next();
});

...
```

Even though we don't know the authenticated user yet, because we are not passing any data to the REST API for it from the outside, we just take any user that we know exists in our database because of the previous MongoDB database seeding. The `findByLogin` method is available on our model, because we have implemented it previously as custom method for it to retrieve users by username or email.

Let's dive into our Express routes now. We have routes for the session, the user, and the message entity. The session entity comes first. Again, instead of using the sample data which was available previously on the models, we can use the models' interface -- powered by Mongoose -- to interact with the database now. In the *src/routes/session.js* change the following lines of code:

```
import { Router } from 'express';

const router = Router();

router.get('/', async (req, res) => {
  const user = await req.context.models.User.findById(
    req.context.me.id,
  );
  return res.send(user);
});

export default router;
```

The route function becomes an asynchronous function, because we are dealing with an asynchronous request to the MongoDB database now. We handle the asynchronous nature of the function with async/await.

Since we passed the models conveniently via the context object to every Express route with an application-wide Express middleware before, we can make use of it here. The authenticated user, which we have taken arbitrarily from the MongoDB database before, can be used to retrieve the current session user from the database.

Let's tackle the user routes in the *src/routes/user.js* file which offer RESTful API endpoints for fetching users or a single user by id. Both API requests should lead into read operations for the MongoDB database:

```
import { Router } from 'express';

const router = Router();

router.get('/', async (req, res) => {
  const users = await req.context.models.User.find();
  return res.send(users);
});

router.get('/:userId', async (req, res) => {
  const user = await req.context.models.User.findById(
    req.params.userId,
  );
  return res.send(user);
});

export default router;
```

The first API endpoint that fetches a list of users doesn't get any input parameters from the request. But the second API endpoint has access to the user identifier to read the correct user from the MongoDB database.

Last but not least, the message routes in the *src/routes/message.js* file. Apart from reading messages and a single message by identifier, we also have API endpoints for creating a message and deleting a message. Both operations should lead to write operations for the MongoDB database:

```
import { Router } from 'express';

const router = Router();

router.get('/', async (req, res) => {
  const messages = await req.context.models.Message.find();
  return res.send(messages);
});

router.get('/:messageId', async (req, res) => {
  const message = await req.context.models.Message.findById(
    req.params.messageId,
  );
  return res.send(message);
});

router.post('/', async (req, res) => {
  const message = await req.context.models.Message.create({
    text: req.body.text,
    user: req.context.me.id,
  });

  return res.send(message);
});
```

```
router.delete('/:messageId', async (req, res) => {
  const message = await req.context.models.Message.findById(
    req.params.messageId,
  );

  if (message) {
    await message.remove();
  }

  return res.send(message);
});

export default router;
```

There are shorter ways to accomplish the remove of a message in the database with Mongoose. However, by going this way, you make sure to trigger the database hooks which can be set up in the models. You have set up one of these hooks, a remove hook, in the *src/models/user.js* file previously:

```
...

userSchema.pre('remove', function(next) {
  this.model('Message').deleteMany({ user: this._id }, next);
});

...
```

Every time a user is deleted, this hook makes sure that all messages that belong to this user are deleted as well. That's how you don't have to deal to clean up the database properly on every delete operation of an entity.

Basically that's it for connecting MongoDB to Express routes with Mongoose. All the models set up with Mongoose can be used as interface to your MongoDB database. Once a user hits your REST API, you can do read or write operations in the Express routes to your MongoDB database.

Exercises

- Confirm your source code for the last section. Be aware that the project cannot run properly in the Sandbox, because there is no database.
  - Confirm your changes from the last section.
- Check the source code of the alternative PostgreSQL with Sequelize implementation.
- Experiment with your REST API with cURL operations.

This tutorial is part 1 of 2 in this series.

Part 2: How to handle errors in Express

Show Comments

KEEP READING ABOUT DOCKER›

HOW TO DOCKER WITH NODE.JS

Just recently I had to use Docker for my Node.js web application development. Here I want to give you a brief walkthrough on how to achieve it. First of all, we need a Node.js application. Either take...

CREATING A REST API WITH EXPRESS.JS AND POSTGRESQL

Node + Express + PostgreSQL is a powerful tech stack for backend applications to offer CRUD operations. It gives you everything to expose an API (Express routes), to add business logic (Express...

THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK >

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✔ Join 50.000+ Developers
- ✔ Learn Web Development
- ✔ Learn JavaScript
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

SUBSCRIBE >

View our Privacy Policy.

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

© Robin Wieruch

Contact Me Privacy & Terms