Setup MongoDB with Mongoose in Express

APRIL 27, 2020 BY ROBIN WIERUCH - EDIT THIS POST

This tutorial is part 4 of 4 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

Eventually every Node.js project running with Express.js as web application will need a database. Since most server applications are stateless, in order to scale them horizontally with multiple server instances, there is no way to persist data without another third-party (e.g. database). That's why it is fine to develop an initial application with sample data, where it is possible to read and write data without a database, but at some point you want to introduce a database to manage the data. The database would keep the data persistence across servers or even though one of your servers is not running.

The following sections will show you how to connect your Express application to a MongoDB database with Mongoose as ORM. If you haven't installed MongoDB on your machine yet,

head over to this guide on how to install MongoDB for your machine. It comes with a MacOS and a Windows setup guide. Afterward come back to the next section of this guide to learn more about using MongoDB in Express.

MONGODB WITH MONGOOSE IN EXPRESS INSTALLATION

To connect MongoDB to your Express application, we will use an ORM to convert information from the database to a JavaScript application without SQL statements. ORM is short for Object Related Mapping, a technique that programmers use to convert data among incompatible types. More specifically, ORMs mimic the actual database so a developer can operate within a programming language (e.g. JavaScript) without using a database query language (e.g. SQL) to interact with the database. The downside is the extra code abstraction, that's why there are developers who advocate against an ORM, but this shouldn't be a problem for many JavaScript applications without complex database queries.

For this application, we'll use Mongoose as ORM. Mongoose provides a comfortable API to work with MongoDB databases from setup to execution. Before you can implement database usage in your Node.js application, install mongoose on the command line for your Node.js application:

```
npm install mongoose --save
```

After you have installed the library as node packages, we'll plan and implement our database entities with models and schemas.

DATABASE MODELS, SCHEMAS AND ENTITIES

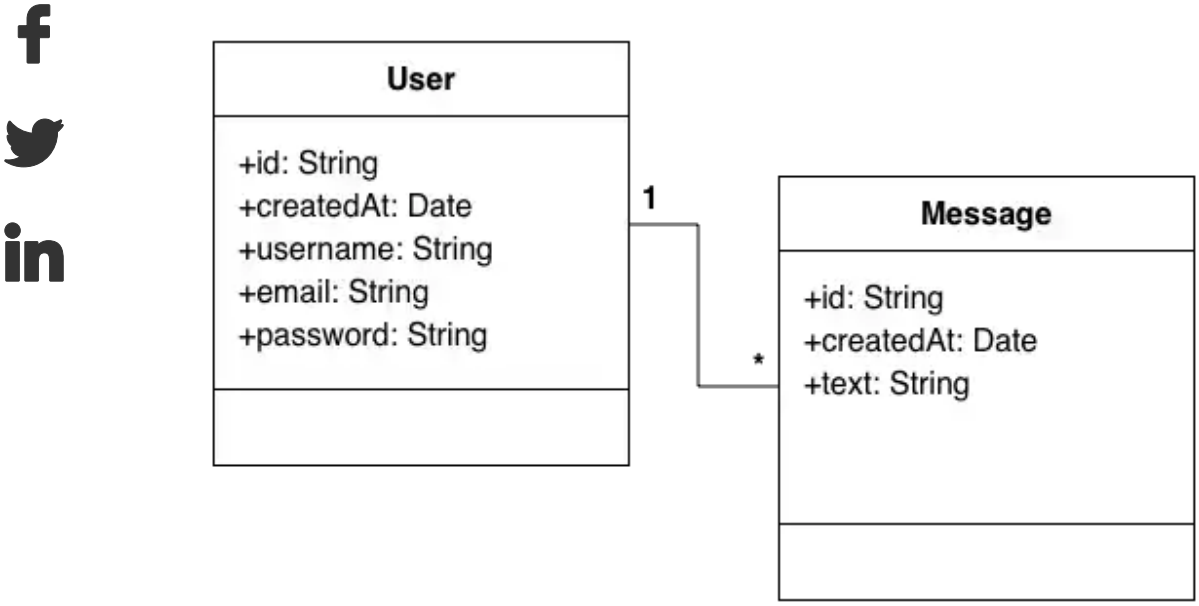
The following case implements a database for your application with two database entities: User and Message. Often a database entity is called database schema or database model as well. You can distinguish them the following way:

- Database Schema: A database schema is close to the implementation details and tells the database (and developer) how an entity (e.g. user entity) looks like in a database table whereas every instance of an entity is represented by a table row. For instance, the schema defines fields (e.g. username) and relationships (e.g. a user has messages) of an entity. Each field is represented as a column in the database. Basically a schema is the blueprint for an entity.

- Database Model: A database model is a more abstract perspective on the schema. It offers the developer a conceptual framework on what models are available and how to use models as interfaces to connect an application to a database to interact with the entities. Often models are implemented with ORMs.

- Database Entity: A database entity is actual instance of a stored item in the database that is created with a database schema. Each database entity uses a row in the database table whereas each field of the entity is defined by a column. A relationship to another entity is often described with an identifier of the other entity and ends up as field in the database as well.

  Before diving into the code for your application, it's always a good idea to map the relationships between entities and how to handle the data that must pass between them. A UML (Unified Modeling Language) diagram is a straightforward way to express relationships between entities in a way that can be referenced quickly as you type them out. This is useful for the person laying the groundwork for an application as well as anyone who wants to additional information in the database schema to it. An UML diagram could appear as such:



The User and Message entities have fields that define both their identity within the construct and their relationships to each other. Let's get back to our Express application. Usually, there is a folder in your Node.js application called *src/models/* that contains files for each model in your database (e.g. *src/models/user.js* and *src/models/message.js*). Each model is implemented as a schema that defines the fields and relationships. There is often also a file (e.g. *src/models/index.js*) that combines all models and exports all them as database interface to the Express application. We can start with the two models in the *src/models/[modelname].js* files, which could be expressed like the following without covering all the fields from the UML diagram for the sake of keeping it simple. First, the user model in the *src/models/user.js* file:

```
import mongoose from 'mongoose';

const userSchema = new mongoose.Schema(
  {
    username: {
      type: String,
      unique: true,
      required: true,
    },
  },
  { timestamps: true },
);

const User = mongoose.model('User', userSchema);

export default User;
```

As you can see, the user has a username field which is represented as string type. In addition, we added some more validation for our user entity. First, we don't want to have duplicated usernames in our database, hence we add the unique attribute to the field. And second, we want to make the username string required, so that there is no user without a username. Last but not least, we defined timestamps for this database entity, which will result in additional `createdAt` and `updatedAt` fields.

We can also implement additional methods on our model. Let's assume our user entity ends up with an email field in the future. Then we could add a method that finds a user by their an abstract "login" term, which is the username or email in the end, in the database. That's helpful when users are able to login to your application via username *or* email adress. You can implement it as method for your model. After, this method would be available next to all the other build-in methods that come from your chosen ORM:

```
import mongoose from 'mongoose';

const userSchema = new mongoose.Schema(
  {
    username: {
      type: String,
      unique: true,
      required: true,
    },
  },
  { timestamps: true },
);

userSchema.statics.findByLogin = async function (login) {
  let user = await this.findOne({
    username: login,
  });

  if (!user) {
    user = await this.findOne({ email: login });
  }
```

```
  return user;
};

const User = mongoose.model('User', userSchema);

export default User;
```

The message model looks quite similar, even though we don't add any custom methods to it and the fields are pretty straightforward with only a text field:

```
import mongoose from 'mongoose';

const messageSchema = new mongoose.Schema(
  {
    text: {
      type: String,
      required: true,
    },
  },
  { timestamps: true },
);

const Message = mongoose.model('Message', messageSchema);

export default Message;
```

However, we may want to associate the message with a user:

```
import mongoose from 'mongoose';

const messageSchema = new mongoose.Schema(
  {
    text: {
      type: String,
      required: true,
    },
    user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  },
  { timestamps: true },
);

const Message = mongoose.model('Message', messageSchema);

export default Message;
```

Now, in case a user is deleted, we may want to perform a so called cascade delete for all messages in relation to the user. That's why you can extend schemas with hooks. In this case, we add a pre hook to our user schema to remove all messages of this user on its deletion:

```
import mongoose from 'mongoose';
```

```
const userSchema = new mongoose.Schema(
  {
    username: {
      type: String,
      unique: true,
      required: true,
    },
  },
  { timestamps: true },
);

userSchema.statics.findByLogin = async function (login) {
  let user = await this.findOne({
    username: login,
  });

  if (!user) {
    user = await this.findOne({ email: login });
  }

  return user;
};

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

const User = mongoose.model('User', userSchema);

export default User;
```

Mongoose is used to define the model with its content (composed of types and optional configuration). Furthermore, additional methods can be added to shape the database interface and references can be used to create relations between models. An user can have multiple messages, but a Message belongs to only one user. You can dive deeper into these concepts in the Mongoose documentation. Next, in your *src/models/index.js* file, import and combine those models and export them as unified models interface:

```
import mongoose from 'mongoose';

import User from './user';
import Message from './message';

const connectDb = () => {
  return mongoose.connect(process.env.DATABASE_URL);
};

const models = { User, Message };

export { connectDb };

export default models;
```

At the top of the file, you create a connection function by passing the database URL as mandatory argument to it. In our case, we are using environment variables, but you can pass the argument as string in the source code too. For example, the environment variable could look like the following in an *.env* file:

```
DATABASE_URL=mongodb://localhost:27017/node-express-mongodb-server
```

Note: The database URL can seen when you start up your MongoDB on the command line. You only need to define a subpath for the URL to define a specific database. If the database doesn't exist yet, MongoDB will create one for you.

Lastly, use the function in your Express application. It connects to the database asynchronously and once this is done you can start your Express application.

```
import express from 'express';
...

import models, { connectDb } from './models';

const app = express();

...

connectDb().then(async () => {
  app.listen(process.env.PORT, () =>
    console.log(`Example app listening on port ${process.env.PORT}!`),
  );
});
```

If you want to re-initialize your database on every Express server start, you can add a condition to your function:

```
...

const eraseDatabaseOnSync = true;

connectDb().then(async () => {
  if (eraseDatabaseOnSync) {
    await Promise.all([
      models.User.deleteMany({}),
      models.Message.deleteMany({}),
    ]);
  }

  app.listen(process.env.PORT, () =>
    console.log(`Example app listening on port ${process.env.PORT}!`),
  );
});
```

That's it for defining your database models for your Express application and for connecting everything to the database once you start your application. Once you start your application again, the command line results will show how the tables in your database were created.

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.
- Read more about Mongoose.

HOW TO SEED A MONGODB DATABASE?

Last but not least, you may want to seed your MongoDB database with initial data to start with. Otherwise, you will always start with a blank slate when purging your database (e.g. eraseDatabaseOnSync) with every application start.

In our case, we have user and message entities in our database. Each message is associated to a user. Now, every time you start your application, your database is connected to your physical database. That's where you decided to purge all your data with a boolean flag in your source code. Also this could be the place for seeding your database with initial data.

```
...

const eraseDatabaseOnSync = true;

connectDb().then(async () => {
  if (eraseDatabaseOnSync) {
    await Promise.all([
      models.User.deleteMany({}),
      models.Message.deleteMany({}),
    ]);

    createUsersWithMessages();
  }

  app.listen(process.env.PORT, () =>
    console.log(`Example app listening on port ${process.env.PORT}!`),
  );
});

const createUsersWithMessages = async () => {
  ...
};
```

The `createUsersWithMessages()` function will be used to seed our database. The seeding happens asynchronously, because creating data in the database is not a synchronous task.

Let's see how we can create our first user in MongoDB with Mongoose:

```
...

const createUsersWithMessages = async () => {
  const user1 = new models.User({
    username: 'rwieruch',
  });

  await user1.save();
};
```

Each of our user entities has only a username as property. But what about the message(s) for this user? We can create them in another function which associates the message to a user by reference (e.g. user identifier):

```
...

const createUsersWithMessages = async () => {
  const user1 = new models.User({
    username: 'rwieruch',
  });

  const message1 = new models.Message({
    text: 'Published the Road to learn React',
    user: user1.id,
  });

  await message1.save();

  await user1.save();
};
```

We can create each entity on its own but associate them with the neccassary information to each other. Then we can save all entities to the actual database. Let's create a second user, but this time with two messages:

```
...

const createUsersWithMessages = async () => {
  const user1 = new models.User({
    username: 'rwieruch',
  });

  const user2 = new models.User({
    username: 'ddavids',
  });

  const message1 = new models.Message({
    text: 'Published the Road to learn React',
    user: user1.id,
  });
```

```
  const message2 = new models.Message({
    text: 'Happy to release ...',
    user: user2.id,
  });

  const message3 = new models.Message({
    text: 'Published a complete ...',
    user: user2.id,
  });

  await message1.save();
  await message2.save();
  await message3.save();

  await user1.save();
  await user2.save();
};
```

That's it. In our case, we have used our models to create users with associated messages. It happens when the application starts and we want to start with a clean slate; it's called database seeding. However, the API of our models is used the same way later in our application to create users and messages. In the end, we have set up MongoDB in a Node.js with Express application. What's missing is connecting the database to Express for enabling users to operate on the database with the API rather than operating on sample data.

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.
- Explore:
  - What else could be used instead of Mongoose as ORM alternative?
  - What else could be used instead of MongoDB as database alternative?
  - Compare your source code with the source code from the PostgreSQL + Sequelize alternative.
- Ask yourself:
  - When would you seed an application in a production ready environment?
  - Are ORMs like Mongoose essential to connect your application to a database?

This tutorial is part 4 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 5: Creating a REST API with Express.js and MongoDB

Show Comments

KEEP READING ABOUT NODE

HOW TO SETUP EXPRESS.JS IN NODE.JS

Express.js is the most popular choice when it comes to building web applications with Node.js. However, when saying web applications with Node.js, it's often not for anything visible in the browser...

SETUP POSTGRESQL WITH SEQUELIZE IN EXPRESS

Eventually every Node.js project running with Express.js as web application will need a database. Since most server applications are stateless, in order to scale them horizontally with multiple server...

Open Source
Tutorials
What I use
How to work with me
How to support me

Contact Me Privacy & Terms