building a database backed application in Node

A lot of times when I try a new front end technology or decide to work with a new framework, I need a backend to work with.

For the longest time I've used a project databaase hosted locally but I've decided to use a different strategy and build a single REST API backed by a MongoDB database. With an API I can then concentrate on the front end and use an existing CRUD (Create, Read, Update, Delete) REST API.

The following table shows the different parts of the proposed API:

HTTP Verb	Endpoint	Description
GET	projects	Get all projects
POST	project	Create a new project
GET	project/:id	Get a single project by its ID
Patch	project/:id	Update a project
DELETE	project/:id	Delete a project indicated by its ID

We will use Express and Mongoose to build the API and follow Rahman Fadhil's How to Build a REST API with Express and Mongoose.

The post is broken up in three sections:

- The server
- The model
- The routes

We will also do a quick setup of the tools that we need to build the API.

Getting started

The project requires the following tools:

- Node.js

- MongoDB
- Postman

Install Node.js

My preferred way to install Node is to use NVM. It will let you install, run and update multiple versions of Node without manual interaction.

Using wget, run the following command to install NVM:

```
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh
```

Running the above command downloads a script and runs it. The script clones the nvm repository to `~/.nvm`, and attempts to add the startup code snippet to the correct profile file (`~/.bash_profile`, `~/.zshrc`, `~/.profile`, or `~/.bashrc`).

If it doesn't work, check the following troubleshooting tips on macOS and troubleshooting tips on Linux for more information.

If successful you should be able to run the following command to check if there's a version of Node up and running

```
node --version
```

You can then install the latest version of Node 16 and switch to using it:

```
nvm install 16
nvm use 16
```

Install MongoDB

We will use the latest version of MongoDB Community Edition Server running locally. We might revisit this later and move it to a managed MongoDB Atlas cluster.

First we will install MongoDB Community Edition Server (MongoDB Community) using Homebrew.

```
brew install mongodb-community
```

And the start the server using the following command:

```
mongod --config /usr/local/etc/mongod.conf --fork
```

Note that this command will not run MongoDB as a service. You will have to start it manually every time

Install Postman

We will use Postman to test our API.

> You can install Postman using the following command:

```
brew install --cask postman
```

You can also download it from the Postman wwebsite after you've created an account and logged in.

Initializing the project

Before we can start writing code, we need to prepare the Node.js environment. To do so run the following commands on your shell/terminal:

```
mkdir api-project
cd api-project
npm init --yes
```

Running `npm init --yes` will create a package.json file and set up the project with default information.

The server

The first portion of the project is the server, located in `index.js`.

We first require and configure Express.js.

We use the `body-parser` to parse the body of the request, and the `json` module to parse JSON incoming requests.

```
const express = require('express');
const app = express();
app.use(express.urlencoded({
  extended: true,
}));
app.use(express.json());
```

We define the routes for the API and we use associate them with the `/api` endpoint.

```
const routes = require('./routes/routes');
app.use('/api', routes);
```

We require `dotenv` to store secrets and we then use a secret in `.env` to configure the database URL that we want to use.

```
require('dotenv').config();
const mongoString = process.env.LOCAL_DB_URL;
```

We require `mongoose` to perform database-related operations.

We connect to the database using the `mongoString` string defined earlier and define a `mongoose.connection` string.

We then define two events, one for errors where we log the error to the console.

The second event is registered after we are connected to the database. We log the status of the connection to the console.

```
const mongoose = require('mongoose');
```

```
mongoose.Promise = global.Promise;

mongoose.connect(mongoString);
const database = mongoose.connection;

database.on('error', (error) => {
  console.log(error);
});

database.once('connected', () => {
  console.log('Database Connected');
});
```

The final part of the server is to start the server by listening on port 3000.

```
app.listen(3000, () => {
  console.log(`Server Started at ${3000}`);
});
```

In the future we need to change the way we listen to the port by using `process.env.PORT` in addition to 3000.

The model

The model in Mongoose talk, is the schema for the objects we use in the database.

This is the schema that we will use for the `projects` collection.

The only special item in the schema is the `timestamps` property. The property will automatically add the `createdAt` and `updatedAt` properties to the objects it creates.

If you notice, we don't add an ID property to the schema. This is because MongoDB will automatically create an ID for us.

```
const mongoose = require('mongoose');
```

```
const projectSchema = new mongoose.Schema({
  name: {
    required: true,
    type: String,
  },
  stage: {
    required: true,
    type: String,
  },
  description: {
    required: true,
    type: String,
  },
  notes: {
    required: false,
    type: String,
  },
  type: {
    required: false,
    type: String,
  },
  codeURL: {
    type: String,
  },
  otherURL: {
    type: String,
  },
  writeupURL: {
    type: String,
  },
},
{
  timestamps: true,
});
;

module.exports = mongoose.model('Data', projectSchema);
```

The HTTP verbs

The final section includes the routes for the API. As a reminder, these are the tasks we want to create along with the associated HTTP verbs.

HTTP Verb	Endpoint	Description
GET	projects	Get all projects
POST	project	Create a new project
GET	project/:id	Get a single project by its ID
Patch	project/:id	Update a project
DELETE	project/:id	Delete a project indicated by its ID

With that in hand we first setup our `routes.js`by completing the following tasks.

1. Require the `express` module
2. Create a `Router` object
3. Require the model we created earlier and assign it to the `Projects` variable

We can now look at individual routes.

```
const express = require('express');
const router = express.Router();
const Projects = require('../models/model');
```

Get all projects

The first, and simplest route is to get all projects.

While this is not strictly a part of the Projects CRUD structure it's always a good idea to have a way to list all entries on the database.

```
// Get all projects
router.get('/projects', async '/projects'=> {
  // res.send('Get all projects');
```

```
  const posts = await Projects.find();
  res.send(posts);
});
```

Create a project

The first CRUD routen will create a new project (the C in CRUD). Note that we're using async functions and await operators to make the promise-based code more readable.

We first capture the structor of the post in a variable to use later.

We then run a try/catch block.

The try statement attempts to save the data to the database and returns a 200 (OK) status

The catch statement returns a status of 400 (Bad Request) and the error message.

```
router.post('/project', async (req, res) => {
  const data = new Projects({
    name: req.body.name,
    stage: req.body.stage,
    description: req.body.description,
    notes: req.body.notes,
    type: req.body.type,
    codeURL: req.body.codeURL,
    otherURL: req.body.otherURL,
    writeupURL: req.body.writeupURL,
  });

  try {
    const dataToSave = await data.save();
    res.status(200).json(dataToSave);
  } catch (err) {
    res.status(400).send(err);
  }
```

```
});
```

Get a project based on its ID

The next endpoint reads a single project using the ID as they key to retrieve it.

We add the value of the `_id` attribute as part of the URL, something like:

```
http://localhost:3000/api/project/622ead24ab960455b613dee8
```

The route will take the ID for t he project and look for it in the database using MongoDB's findOne method.

If the project is not found we set the status to 404 (not found) and return an error message to let the user know.

```
router.get('/project/:id', async (req, res) => {
  try {
    const project = await Projects.findOne({
      _id: req.params.id,
    });
    res.send(project);
  } catch {
    res.status(404);
    res.send({
      error: 'Post doesn\'t exist!',
    });
  }
});
```

This presents the first problem. The IDs generated by MongoDB are long enough to prevent collisions and duplicate IDs but they are hard to remember and hard to type.

Is thre a better way to handle this? For this proof of concept I choose to leave it as is. For more polished projects, I may want to generate separate IDs using UUID or similar tools.

Update a project

There are two ways to update a project using HTTP. The one I chose uses the HTTP PATCH verb to update only the items of the project that have changed. This is the `Update` part of CRUD.

The code grabs a reference to the project by it's ID and find the corresponding entry in the database.

It then checks if individual fields exist in the request object, indicating that they were changed. If so then it adds the new value to the project.

Once it has checked all the values and added the to the project, where appropriate, it saves the project to the database.

If it can't find the project it returns a 404 (not found) status and prints an error message.

```
router.patch('/project/:id', async (req, res) => {
  try {
    const project = await Projects.findOne({
      _id: req.params.id,
    });

    if (req.body.name) {
      project.name = req.body.name;
    }

    if (req.body.stage) {
      project.stage = req.body.stage;
    }

    if (req.body.description) {
      project.description = req.body.description;
    }

    if (req.body.notes) {
      project.notes = req.body.notes;
    }
```

```
    if (req.body.type) {
      project.type = req.body.type;
    }

    if (req.body.codeURL) {
      project.codeURL = req.body.codeURL;
    }

    if (req.body.otherURL) {
      project.otherURL = req.body.otherURL;
    }

    if (req.body.writeupURL) {
      project.writeupURL = req.body.writeupURL;
    }

    await project.save();
    res.send(project);
  } catch {
    res.status(404);
    res.send({error: 'Post doesn\'t exist!'});
  }
});
```

The second way to update a project uses the HTTP PUT verb to update the entire project on the database. As such it requires you to send the entire project as the updated body of the request.

If the resource doesn't exist in the database, a PUT request will create a new entry in the database for it.

See this table, adapted from Difference Between PUT and PATCH Request, for a comparison of the two methods

PUT	PATCH
PUT is a method of modifying resource where the client sends data that updates	PATCH is a method of modifying resources where the client sends only the data that needs to be

PUT	PATCH
the entire resource.	updated without modifying the entire data.
In a PUT request, the client request that the stored version be replaced with the atached payload	A PATCH request however, is a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version.
HTTP PUT is idempotent, So if you send retry a request multiple times, that should be equivalent to a single request modification	A PATCH is not necessarily idempotent, although it can be. If you send the multiple request to patch a the same resource the data will not change.
If the resource doesn't exist, PUT will create it. The body of a PUT request contains the entire resource.	A PATCH request can be used to modify a resource that already exists. It will not create a new resource if it doesn't exist

Delete a project

The final method is Delete. It uses the DELETE method to permanently remove a resource from the database.

The route will take the ID for the project and look for it in the database. If it is found, then it is deleted.

```
// Delete a project by its ID
router.delete('/project/:id', async (req, res) => {
  try {
    const project = await Projects.findOneAndDelete({
      _id: req.params.id,
    });
    res.send(project);
  } catch {
    res.status(404);
    res.send({error: 'Post doesn\'t exist!'});
  }
});
```

Be extremely careful when you use the DELETE method. It is a permanent action and can't be undone.

Exporting the router

After you're done defining the routes, export the router so that other scripts can use via `require`.

```
module.exports = router;
```