

Project: Blog API

NodeJS Course

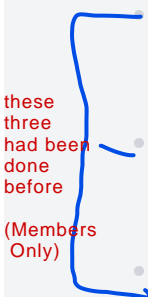
Introduction

Do you know what you need? You need a blog. Or maybe you don't, or maybe you already have one, in any case, this project will be a great way to practice and see the benefits of creating an API only backend. We're actually going to create the backend and **two different front-ends** for accessing and editing your blog posts. **One of the front-end sites will be for people that want to read and comment on your posts**, while the **other one will be just for you to write, edit and publish your posts.**

Why are we setting it up like this? Because we can! If you already have a portfolio site and you want to add your blog posts to that site feel free to do that instead of creating a new site just for that. The important exercise here is setting up the API and then accessing it from the outside. There are some security benefits to setting up separate websites for blog consumption and blog editing, but really we're just doing it like this to **demonstrate the power and flexibility of separating your backend code from your frontend code.**

Assignment

1. How you structure this project is up to you. Some people prefer separate GitHub repos for each of the three apps you will make, to keep them and their commit histories separate. Some people prefer a monorepo, with each app in their own directory within the same single repo.
2. Begin by designing your back end models and schemas. How you design it is up to you, but you might want to think through a few things:

- 
- Your blog should have posts and comments, so think about the fields you are going to want to include for each of those.
 - Are you going to require users to leave a username or email with their comments?
 - Are you going to display a date or a timestamp for posts and comments?
 - Posts should probably have a title, but should comments?
 - A useful feature for a blog is the ability to have posts that are in the database but not published for the public to read. How might you designate published vs unpublished posts in your DB?
 - You will want a user model that will contain any blog authors and any normal user accounts. Even if you decide to only have a single author and no normal user accounts, a minimal user model will still be helpful to allow for easier route protection via authentication.

3. Set up your Express app, and define the models in Mongoose.
4. Set up your routes and controllers! Think about RESTful organization for this one. Most of the examples in the previous lesson were centered around posts and comments so this shouldn't be too tricky.
 - You can test your routes however you want. Using `curl` in a terminal is one handy way, but it can be just as effective to use a web browser. There are some platforms that allow you to send `PUT` and `POST` requests without needing to set up and fill out HTML forms. [Postman](#) is probably the most popular.

5. Certain routes will need to be protected via authentication. You wouldn't want any random stranger online to edit your articles! If you also implement normal user accounts then you may also want to protect some routes behind being logged in.
 - Though there are many ways you can handle authentication, in this project, use JWTs.
 - You can use [jsonwebtoken](#) to create and verify JWTs. You may wish to use [Passport's JWT strategy](#) for verifying JWTs, especially if you already have Passport set up with a local strategy to handle logging in.
 - A successful login will grant the user a JWT. That user can then attach their JWT to any future requests, where your API can verify the JWT in order to allow or deny access to the rest of the protected route. When the user logs out, you can have the client remove the JWT from storage.
 - There are many ways to send and store JWTs, such as via cookies, storing in localStorage, using access/refresh tokens etc. Some of these methods are more complicated (though with the right implementation, potentially more secure), especially once you deploy both ends. For example, cross-site cookies can be a real headache if you aren't aware of certain extra details. You may wish to explore some of these alternatives in the future. For now, keep it simple and send your JWTs via an "Authorization" header with "Bearer" schema, and have the client store a JWT in localStorage.
6. Once your API is working you can focus on your front-end code. Really, how you go about this is up to you. If you are comfortable with a front-end framework then go for it! If you're happier using plain HTML and CSS that's fine too. All you should have to do to get your posts into a website is to `fetch` the correct API endpoint and then display the results. Working with fetch and APIs from a front-end perspective is covered in the [Working with APIs lesson](#).

7. Create a second website for authoring and editing your posts. You can set this up however you like but the following features might be useful:
 - A list of all posts that shows whether or not they have been published.
 - A button to publish unpublished posts, or to unpublish published ones!
 - A 'NEW POST' form. If you want to get fancy, you could use a rich text editor such as [TinyMCE](#).
 - The ability to manage comments (i.e. delete or edit them).
8. How much work you want to put into the front-end code on this one is up to you. Technically this is a backend focused course so if you would prefer, feel free to focus on the REST API.
9. Deploying your separate apps isn't anything fancy. Deploy your API like with your previous projects using a PaaS from the [Deployment lesson](#), and deploy your front-ends like you would have deployed your front-ends before. If you used React, recall several hosting options from the [CV Application project](#).

Additional resources

This section contains helpful links to related content. It isn't required, so consider it supplemental.

- As mentioned earlier, the cli-tool `curl` and [Postman](#) are popular choices for testing your routes.

 [Improve on GitHub](#)  [Report an issue](#)

Your solution

[View community solutions](#)