# Assignment 3.2: Generic Social Media Backend

This assignment closes the loop on **assignment 3.1** by having you implement the API backend for the Generic Social Media App.

## Learning goals

Through this assignment, you will

* Implement a REST API using Node and Express,
* Store and retrieve data in a MongoDB database,
* Work with an API specification from an implementer's perspective, and
* Integrate and test an API with an existing frontend.

## Database structure

You will store the data for your app in a MongoDB database. In your code, you must use the DATABASE\_NAME variable to select the database to store and retrieve data.

The database contains two collections: users and posts. The documents in each collection have the following fields:

### users

* id: the user's ID. This should always be unique. This won't be enforced by MongoDB, so you should be careful not to store two user documents with the same id.
* name: the user's display name.
* avatarURL: the user's avatar URL.
* following: an array of the users this user is following. Each element of the array is a user ID. The IDs in the array should be unique (again, not enforced by MongoDB).

### posts

* userId: the user ID of the poster.
* time: the date/time the user made the post. This is a JavaScript Date object.
* text: the text of the post.

### Initialization

To initialize (or reset) the database, run mongosh init\_db.mongodb from your terminal. (On Windows, replace mongosh with the full path to mongosh.exe, e.g. "PATH\_WHERE\_YOU\_EXTRACTED\_THE\_ZIP\_FILE\bin\mongosh.exe"; use tab completion to enter the path more easily.)

After running this command, your database will look like this:

> use cs193x\_assign3

switched to db cs193x\_assign3

> db.users.find()

{ "\_id" : ObjectId(<...>), "id" : "mchang", "name" : "Michael", "avatarURL" : "images/stanford.png", "following" : [ ] }

> db.posts.find()

{ "\_id" : ObjectId(<...>), "userId" : "mchang", "time" : ISODate(<...>), "text" : "Welcome to the Generic Social Media App!" }

## API description

You are implementing the [same API as you used in assignment 3.1](https://web.stanford.edu/class/cs193x/assign3.1/api.html). A few assignment 3.2 specific details:

[start deliverable]

* All API endpoints are under the /api prefix.
* You can assume the request body, if present, will be a well-formed JSON object. That is, you don't have to worry about body-parser failing to parse the request body.
* You are expected to generate all of the errors listed in the specification. For each, you should set the HTTP status according to the spec, and your response should be an object that includes an error key with a message describing the error. Your messages should be human-readable and descriptive. They don't need to match ours exactly.
* You do not need to edit the / route to match the example in the spec.
* You do not need to handle any concurrency issues, e.g. getting two requests at the same time to create the same user.

Here are a few tips for implementing these endpoints:

[end deliverable]

* If you don't get the "Server started" message when you run npm start, this likely means your MongoDB server isn't running, so your backend can't connect to it.
* The order you define endpoints can matter; Express checks for a match from top to bottom. For example, you need to put all your endpoints above the api.all("/\*") "catch-all" endpoint, or your handler will never be called.
* Pay close attention to the structure of the response in the spec (and the remote API in the tester). E.g. returning an array is not the same as returning an object with one key/value pair, where the value is an array. If your assign3.1 app works fine with our API but gets "Cannot read properties of undefined" or similar errors with yours, this is a likely cause.
* When returning data to the client, don't forget to remove (or not include) key/value pairs that aren't in the spec, e.g. the internal \_id for each document.
* Use new Date() to get a Date object that represents the current date and time. You can directly return (via res.json) a Date object, and it will be formatted correctly.
* For the feed endpoint, you'll need to do some work to build the return value, because the posts collection only contains the poster's user ID. This is done so that older posts won't end up with stale values if the poster changes their display name or avatar.
* The logic for the feed endpoint can be a little complex. A clean approach would be to look up only the users and posts that you need, using MongoDB's [$in query operatorlaunch](https://docs.mongodb.com/manual/reference/operator/query/in/" \l "op._S_in). Alternatively, you can just get all the users and posts and filter them in JavaScript; that will work fine for our needs.

[external]

* You will need to sort posts using a comparison function. See the description of how this works in [the documentation for sortlaunch](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort). You can compare (or subtract) Date objects as if they are numbers.

[external]

* We hope many of the endpoints are relatively direct applications of the material (and code) covered in the Node/backend lectures. The feed endpoint is a bit more of a step up from that, incorporating a bit more JavaScript logic. You don't need to use any fancy tricks or make the code as efficient as possible.

Aside: CORS: You might notice the line api.use(cors()); in api/index.js. We didn't get a chance to talk about what this means, though there is a slide about it in lecture 13. In short, this line allows your API to be accessed from other origins, i.e. a server running on another port or machine. This won't come up for the assignment, but it's a common line to have in your API and can avoid some confusing errors in future projects.