Image by Pete Linforth from Pixabay

THE ULTIMATE GUIDE — BUILD YOUR OWN FULL-STACK REACT APP

Set up an Express.Js App With Passport.Js and Mongodb for Password Authentication

Building the Backend for a Milestone Referral Program (I/II)

Fabian Bosler Follow

Feb 18, 2020 · 12 min read ★

In this article, you will learn how to build a web app with Node.js, Express.js, Passport.js, and MongoDB to authenticate a user via REST-endpoint with password and email. You will also learn how to connect that endpoint with a frontend of your choice. The article is intended as a code-along guide, so boot up your VisualStudio and let's go.

Structure of the article

In this piece, we will build the first version of the backend for a referral app. The article is self-contained up to the point where we connect the backend with our frontend. However, If you want to use your frontend, that is not a problem whatsoever. The same logic applies, and the explanations hold. If you want to read up on or download the frontend we have built, head over to the corresponding article:

> Building the Frontend for a Milestone Referral Program (II/II)
>
> Use react-router, styled-components, and react-bootstrap to build a landing page, profile page, signup/login modal...
>
> medium.com

We are going to use Passport.js and implement a so-called `local Strategy`. Strategies in Passport.js are authentication flows. There are hundreds of strategies out there. Each strategy covers different authentication flows. Every strategy comes as it's own npm package, which allows to only install and configure only the strategies you are actually going to use.

As the database of choice, we will use the wildly popular MongoDB for its ease of setup and flexible document-based structure. We are going to use Mongoose as our database object modeling layer.

Table of contents

① Creating the Folder Structure

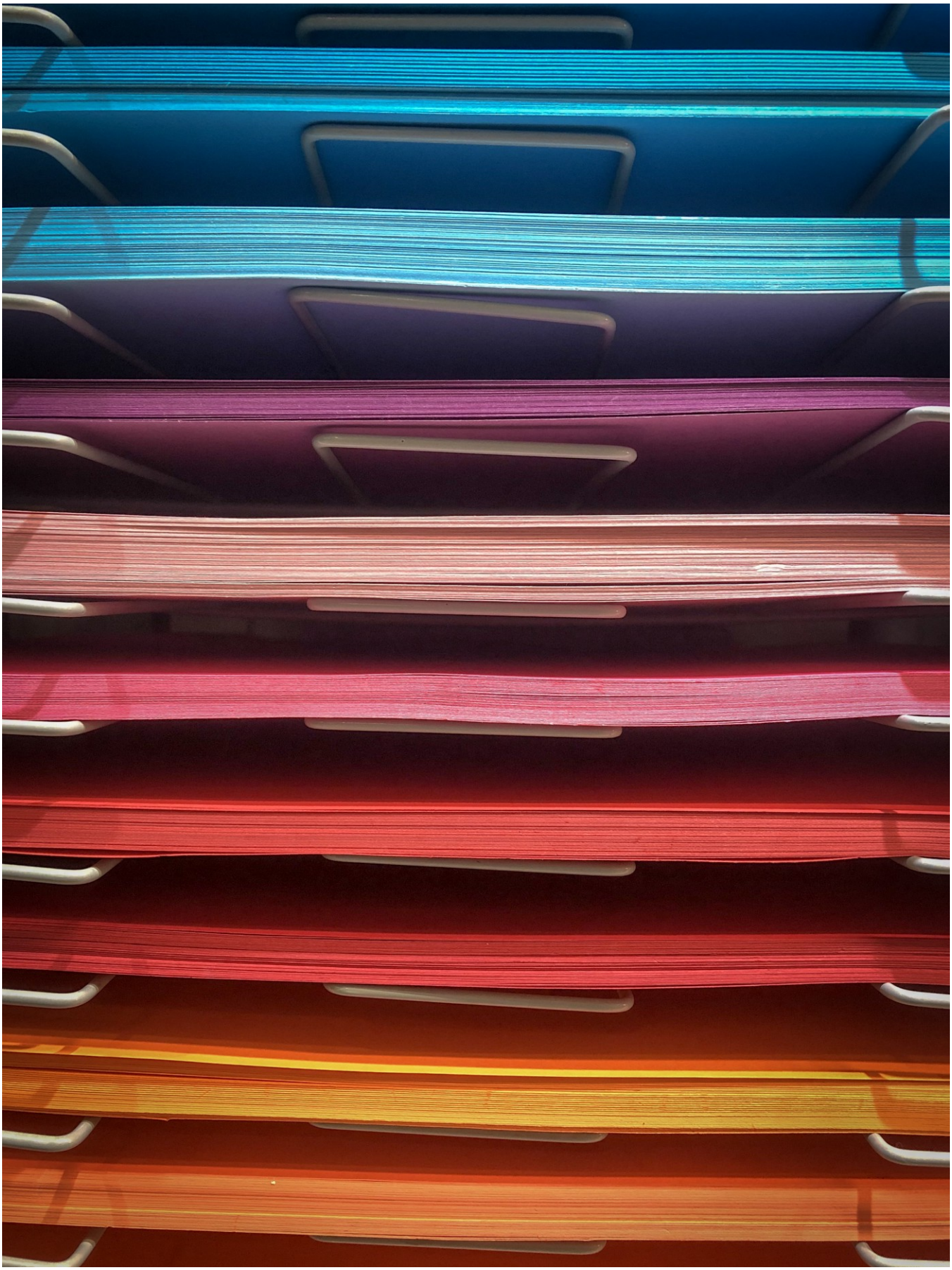


Photo by Omid Kashmari on Unsplash

Currently, our folder structure looks like this:

```
project-folder
# Existing
├── client
│   ├── public
│   │   └── assets
│   └── src
│       └── components
│           ├── data
│           │   └── icons
│           ├── layout
│           │   ├── About
│           │   ├── DelegatedAuthList
│           │   ├── Footer
│           │   ├── Landing
│           │   ├── Milestones
│           │   ├── Nav
│           │   ├── Profile
│           │   ├── ShareButtons
│           │   └── SignupLoginModal
│           └── styles
# To come in this article
└── server
    ├── models
    ├── passport
    └── routes
```

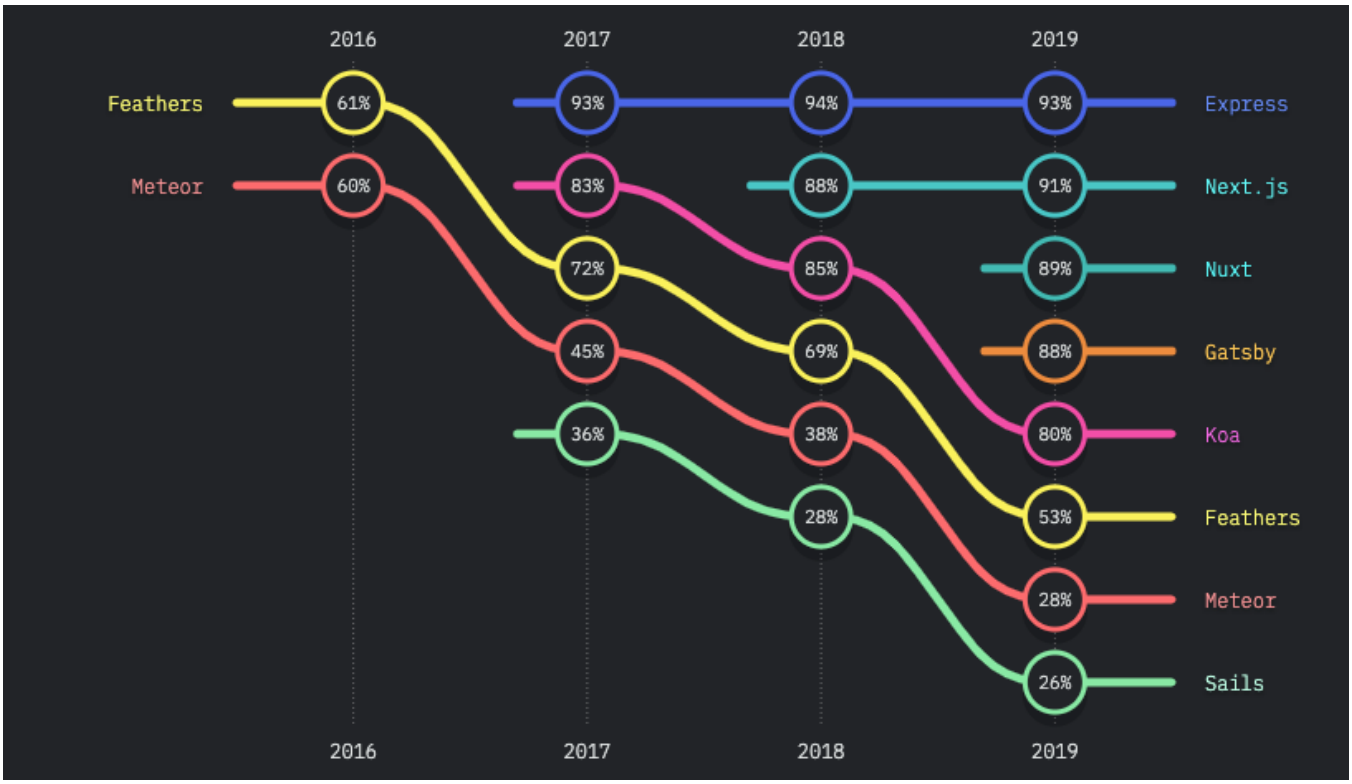
If you are curious about how to generate a tree view like this, run `tree -d -I node_modules` in the project folder, to print out this helpful overview.

Go ahead and create a new folder on the root level of the project called `server` by typing `mkdir server`. Navigate into the new folder (`cd server`) and initialize it as a `node.js` project by typing `yarn init`. After doing so a couple of questions pop up which you can answer or you can just hit enter until all the questions are answered. The answers will populate the `package.json`. I typically change the entry point to `server.js` instead of the default `index.js` during the init-phase but use the defaults for the other questions.

②Setting up an express.js app

Express.js calls itself a “Fast, unopinionated, minimalist web framework for Node.js” and is pretty much my default for web applications. Express.js has scored the highest

grades (in terms of awareness, satisfaction, and interest) over the past couple of years, according to StateOfJS, when it comes to JavaScript-based backends.



Express scores the highest satisfaction levels according to SateOfJS2019

To me, one of the most important questions when making a technology choice is:

> “Will the technology still be en vogue in five years?”

I do love to tinker around with new, fancy frameworks and libraries. However, over the years, the maintainability, ecosystem, and community around popular technologies have convinced me to rather go with a safe bet than a fancy one. After all, I did build apps with Meteor.js in 2014–2016, and I loved it, but the decline in popularity and the somewhat negative outlook make me not use Meteor.js for recent projects anymore.

I don’t think that Express.js will suffer the same fate as it is the de-facto lingua franca when it comes to JavaScript backends for web applications.

Prerequisites

Let’s install nodemon as a dev dependency:

```
yarn add nodemon --dev
```

`nodemon` restarts the server whenever we change files and is thus perfect for development. The `--dev` installs the dependency in the devDependencies, which are only used for development and can be excluded when we build our project for production (e.g., `yarn install --prod`) and thus reduce the final size of the project.

Of course, we also have to install express.js:

```
yarn add express
```

Setting up the express app

Now let's create our app by typing `touch server.js` and add the following code:

```
const express = require("express");
const app = express();
const port = 3000;

app.get("/", (req, res) => res.send("Good monring sunshine!"));

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

After saving, we can start our app via `nodemon` by simply typing `yarn nodemon server.js` in our server folder. You will then see the following output in the terminal:

starting nodemon via yarn

A quick check of `localhost:3000` reveals that everything went as expected:

The output of our “Hello World!” express.js app

Now, and that’s the good thing about `nodemon`, if we make changes to our app, the server will be restarted, and all we have to do is to refresh the page. Try it out by changing "Hello World!" to "Good morning sunshine!" and save the file. Hit refresh in your browser, and you should see the new text.

③ Setting up our user model with Mongoose

Photo by Startaê Team on Unsplash

We are going to use `mongoose.js` for our MongoDB object modeling. Mongoose provides a **schema-based** solution to model our user data. In simple words, `mongoose` is a translation layer between our database (documents) and our code (objects), as we can, for example, create a new `user` object and then persist it to the database, without ever concerning ourselves with the JSON representation of that object. First, Install Mongoose:

```
yarn add mongoose
```

Then create a new folder in our `server` folder called `models` by typing `mkdir models`, within that folder let's create a `Users.js` file.

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

const ThirdPartyProviderSchema = new mongoose.Schema({
    provider_name: {
        type: String,
        default: null
    },
    provider_id: {
        type: String,
        default: null
    },
    provider_data: {
        type: {},
        default: null
    }
})

// Create Schema
const UserSchema = new mongoose.Schema(
    {
        name: {
            type: String
        },
        email: {
            type: String,
            required: true,
            unique: true
        },
        email_is_verified: {
            type: Boolean,
            default: false
        },
        password: {
            type: String
        },
        referral_code: {
            type: String,
            default: function() {
                let hash = 0;
                for (let i = 0; i < this.email.length; i++) {
                    hash = this.email.charCodeAt(i) + ((hash << 5) - hash);
                }
```

```
                let res = (hash & 0x00ffffff).toString(16).toUpperCase();
                return "00000".substring(0, 6 - res.length) + res;
            }
        },
        referred_by: {
            type: String,
            default: null
        },
        third_party_auth: [ThirdPartyProviderSchema],
        date: {
            type: Date,
            default: Date.now
        }
    },
    { strict: false }
);

module.exports = User = mongoose.model("users", UserSchema);
```

For code see Gist

Our user model defines a couple of fields:

- **`name`** : Username
- **`email`** : Email address of our user. `email` has to be unique. No two users can have the same email.
- **`email_is_verified`** : `boolean` value that indicates whether or not the email address is verified
- **`password`** **:** This field will hold the decrypted password.
- **`referral_code`** **:** This is a custom function and creates a six-character hash of the `email` . I.e., a new referral code is created whenever somebody signs up.
- **`referred_by`** **:** Indicates by whom the user was referred by.
- **`third_party_auth`:** Since we will include some third-party-authentication (in part II), we will have to store the information somewhere. **third_party_auth** is an array of the third-party provider's data (only fundamental data like profile link) a user uses to log in.
- **`date`** **:** Date when the user has been created

`{strict: false}` means that the object will accept additional not in the schema specified data. We need `{strict: false}` because there is no unifying schema for our third-party provider.

④ Setting up Passport.js with a local strategy

Photo by Levi Ventura on Unsplash

Install passport:

```
yarn add passport.js
```

Then let's create a folder called `server/routes`. We will use this folder to house our authentication routing. Additionally, we will need to define our `passport` strategies somewhere, for this, let's create a folder called `server/passport`. In `passport` let's create a file called `setup.js`.

passport/setup.js

In the `passport/setup.js` , we are going to define a `LocalStrategy` for passport.js to use for now. Passport.js can handle many different authentication flows and provides a strategy for most of them. Each strategy is packaged in its npm package. The local strategy uses a basic authentication flow based on a username/password. We are going to use bcrypt.js to encrypt our users' passwords. Let's add the required packages

```
yarn add passport-local
yarn add bcryptjs
```

Our local strategy is a little bit adjusted as we use `email` instead of `username` . We then check for the existence of the provided email and proceed as follows:

- If we can't find a user, a new user is created with the encrypted password.
- If a user is found, the encrypted provided password is compared against the encrypted password in the database

```
const bcrypt = require("bcryptjs");
const User = require("../models/Users");
const passport = require("passport");
const LocalStrategy = require("passport-local").Strategy;

passport.serializeUser((user, done) => {
    done(null, user.id);
});

passport.deserializeUser((id, done) => {
    User.findById(id, (err, user) => {
        done(err, user);
    });
});

// Local Strategy
passport.use(
    new LocalStrategy({ usernameField: "email" }, (email, password, done) => {
        // Match User
        User.findOne({ email: email })
            .then(user => {
                // Create new User
                if (!user) {
                    const newUser = new User({ email, password });
                    // Hash password before saving in database
                    bcrypt.genSalt(10, (err, salt) => {
                        bcrypt.hash(newUser.password, salt, (err, hash) => {
                            if (err) throw err;
                            newUser.password = hash;
                            newUser
```

```
                            .save()
                            .then(user => {
                                return done(null, user);
                            })
                            .catch(err => {
                                return done(null, false, { message: err });
                            });
                    });
                });
                // Return other user
            } else {
                // Match password
                bcrypt.compare(password, user.password, (err, isMatch) => {
                    if (err) throw err;

                    if (isMatch) {
                        return done(null, user);
                    } else {
                        return done(null, false, { message: "Wrong password" });
                    }
                });
            }
        })
        .catch(err => {
            return done(null, false, { message: err });
        });
    })
);

module.exports = passport;
```

To copy code see Gist

We also need to add serialization and deserialization functionality for Passport.js to use. This one took me some time to understand, so let's summarize what `passport.serializeUser` and `passport.deserializeUser` does (heavily inspired by stack overflow):

- **`serializeUser`** : The user id (we provide as the second argument of the `done` function) is saved in the session and is later used to retrieve the whole object via the `deserializeUser` function. `serializeUser` determines which data of the user object should be stored in the session. The result of the `serializeUser` method is attached to the session as `req.session.passport.user = {}` . In our case, it would be (as we provide the user id as the key) `req.session.passport.user = {id: 'xyz'}` . `serializeUser` is triggered after a successful invocation of our local strategy and **when we are going to log the user in** as part of `passport.logIn` , to which we pass

the `user` object generated by the local strategy. If you want to know more, check this article out.

- **`deserializeUser`** : The first argument of `deserializeUser` corresponds to the key of the user object that was given to the `done` function (see `serializeUser`). So, the whole object is retrieved with the help of that key. That key here is the user id (key can be any key of the user object i.e., name, email, etc.). In `deserializeUser` that key is matched with the in-memory array/database or any data resource. The fetched object is attached to the request object as `req.user`. `deserializeUser` is called when we send a request with an attached session Cookie containing a serialized user id.

routes/auth.js

In our use case, I don't see a massive benefit from separating registration from login, which is why we bundle it into one endpoint. After all, our adjusted local strategy can handle both cases.

```
const express = require("express");
const router = express.Router();
const passport = require("passport");

router.post("/register_login", (req, res, next) => {
    passport.authenticate("local", function(err, user, info) {
        if (err) {
            return res.status(400).json({ errors: err });
        }
        if (!user) {
            return res.status(400).json({ errors: "No user found" });
        }
        req.logIn(user, function(err) {
            if (err) {
                return res.status(400).json({ errors: err });
            }
            return res.status(200).json({ success: `logged in ${user.id}` });
        });
    })(req, res, next);
});

module.exports = router;
```

To copy code see Gist

The callback we have defined in our `passport.authenticate` allows invoking the local strategy, and then, if there is no error and a user is found, it logs the user in.

⑤ Putting it together

Great, we now have defined a user model, a LocalStrategy, and an endpoint. Let's integrate that into our `server.js`. Note: We are going to change the port to 5000, as our frontend will use 3000. First, we need to add a few more packages:

```
yarn add express-session
yarn add connect-mongo
```

- **`express-session`** stores a session identifier on the client within a cookie and stores the session data on the server, typically in a database (we will use our MongoDB to store the session data).
- **`connect-mongo`** allows to set up a `MongoStore` for our session middleware to use, i.e., this allows storing the session information in the MongoDB.

```
const express = require("express");
const session = require("express-session");
const MongoStore = require("connect-mongo")(session);
const mongoose = require("mongoose");

const passport = require("./passport/setup");
const auth = require("./routes/auth");

const app = express();
const PORT = 5000;
const MONGO_URI = "mongodb://127.0.0.1:27017/tutorial_social_login";

mongoose
    .connect(MONGO_URI, { useNewUrlParser: true })
    .then(console.log(`MongoDB connected ${MONGO_URI}`))
    .catch(err => console.log(err));

// Bodyparser middleware, extended false does not allow nested payloads
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
```

```
// Express Session
app.use(
    session({
        secret: "very secret this is",
        resave: false,
        saveUninitialized: true,
        store: new MongoStore({ mongooseConnection: mongoose.connection })
    })
);

// Passport middleware
app.use(passport.initialize());
app.use(passport.session());

// Routes
app.use("/api/auth", auth);
app.get("/", (req, res) => res.send("Good monring sunshine!"));

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

To copy code see Gist

First, we set up our `express.js` app, we then define the port the app is supposed to run on and our MongoDB connection string. We then connect via `mongoose` , we then add a body-parser middleware to read the payloads of `post` and `put` requests. We then configure `express-session` to use our existing Mongo connection to store the session data. Finally, we initialize `passport.js` which we import from our setup file, we then tell `passport` to use sessions to store the serialized user and try to read the unserialized user from.

⑥ Testing the endpoint

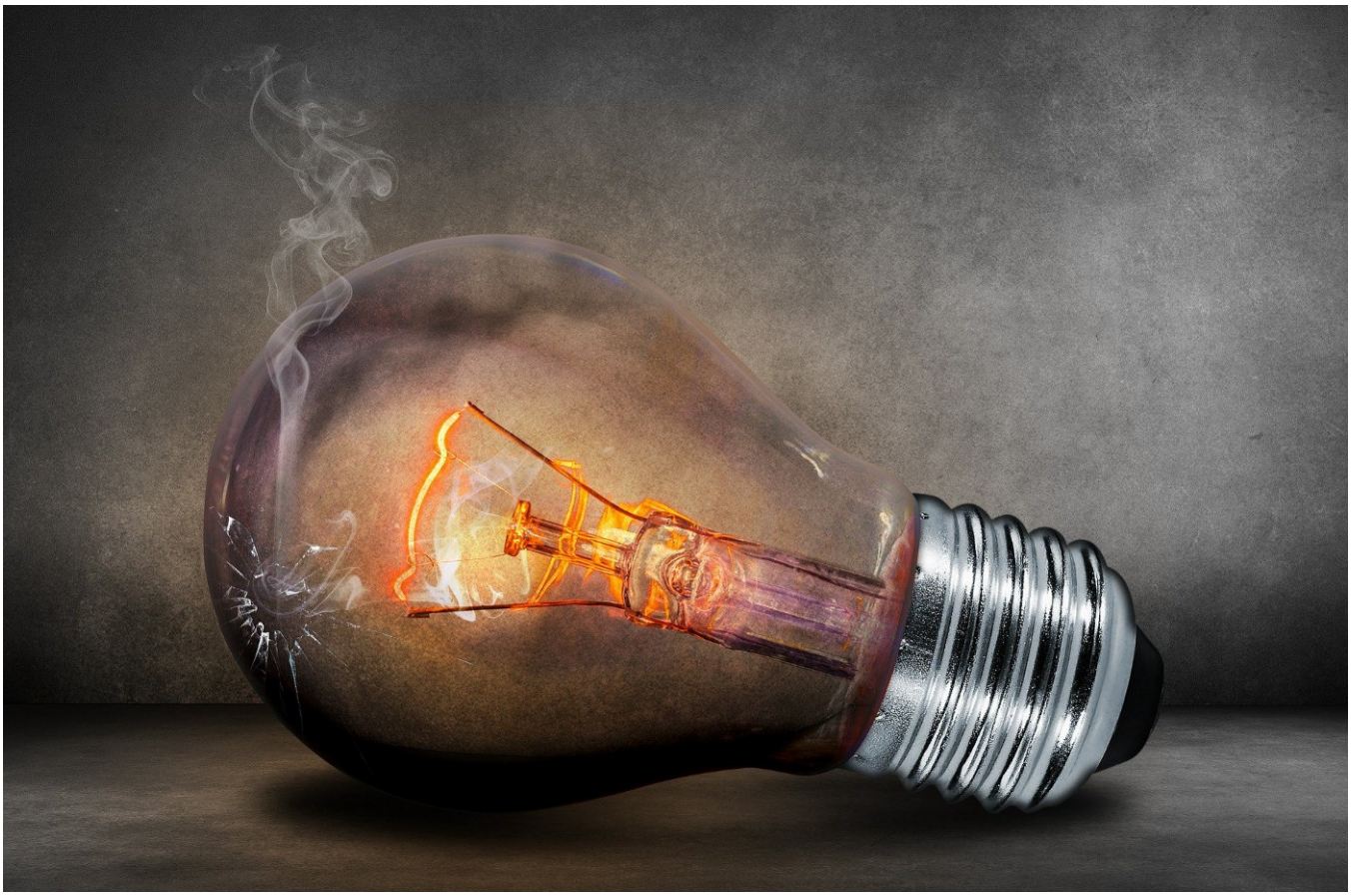
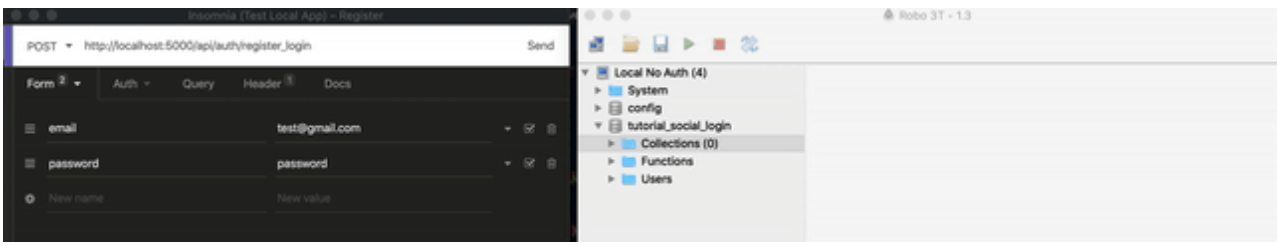


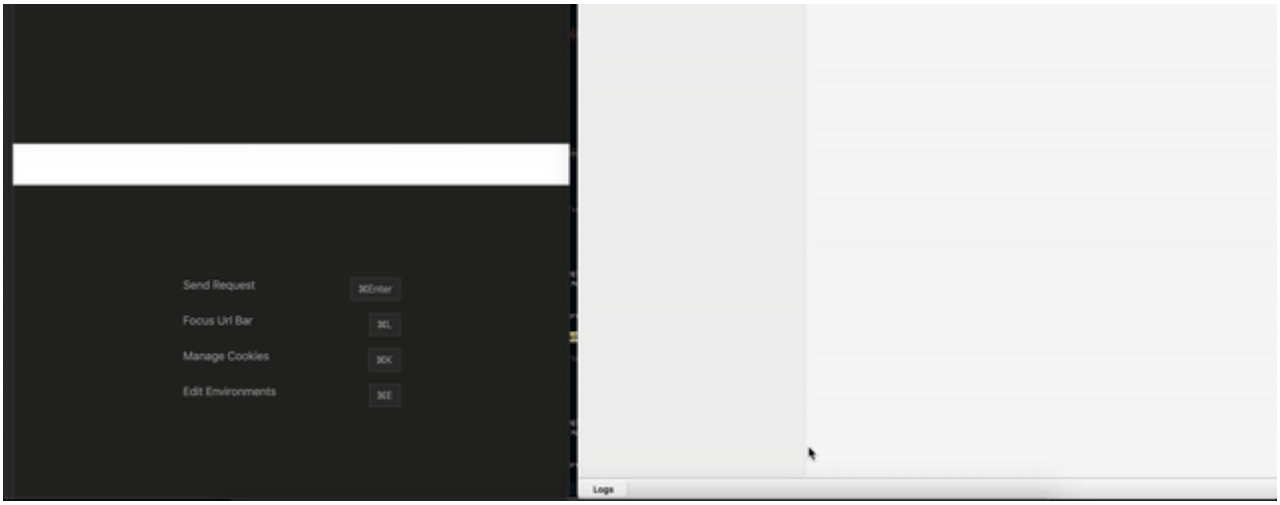
Image by 024–657–834 from Pixabay

Since we are trying to connect to a local MongoDB, we need to have a running Database. Not a problem, if you have not set up a MongoDB yet, follow this article!

After having started the MongoDB, we can start our development server with `yarn nodemon server.js` from the server folder.

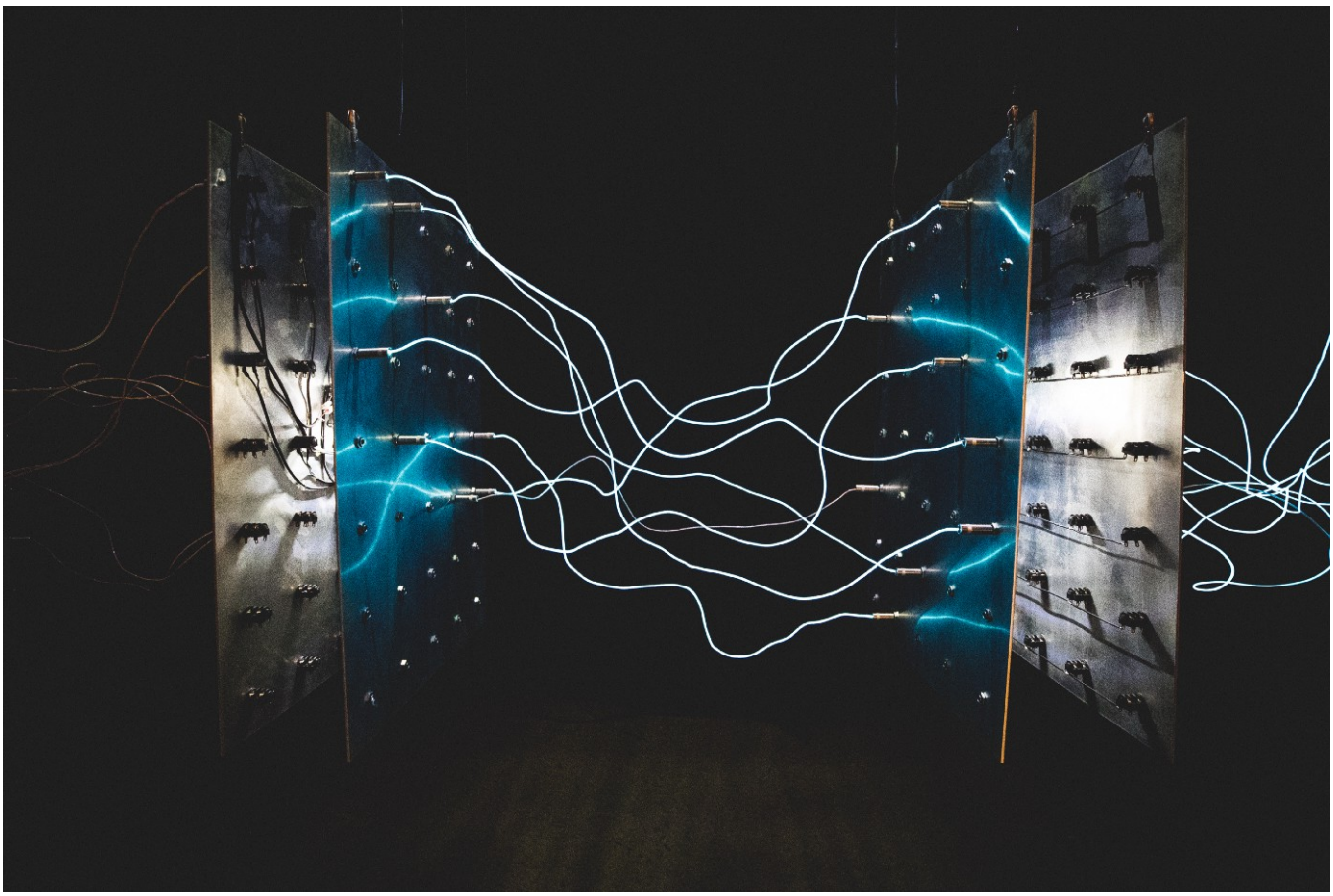
To test the endpoint, we have to make a post request. I like Insomnia to do so, but feel free to use Postman or whatever you prefer to make REST calls.





Insomnia and Robo3T to quickly test that our endpoint is working

⑦ Connecting the frontend



Starting frontend and backend simultaneously

We have validated that our backend works and that it allows us to register and log users in. Let's hook that up with our frontend. Right now we have a `client` and a `server` folder and we would have to start our frontend and backend separately via:

- **Frontend:** Navigate to the `client` folder and run `yarn start`
- **Backend:** Navigate to the `server` folder and run `yarn nodemon server.js`

Let's simplify this, such that we can start both in unison. First, we have to initialize our project folder as an npm project. Then we to add concurrently as a dev dependency.

```
yarn add concurrently --dev
```

Additionally, we have to add a script to our `package.json` (in the project root folder) to start our backend and frontend simultaneously. Your `package.json` should look something like this now:

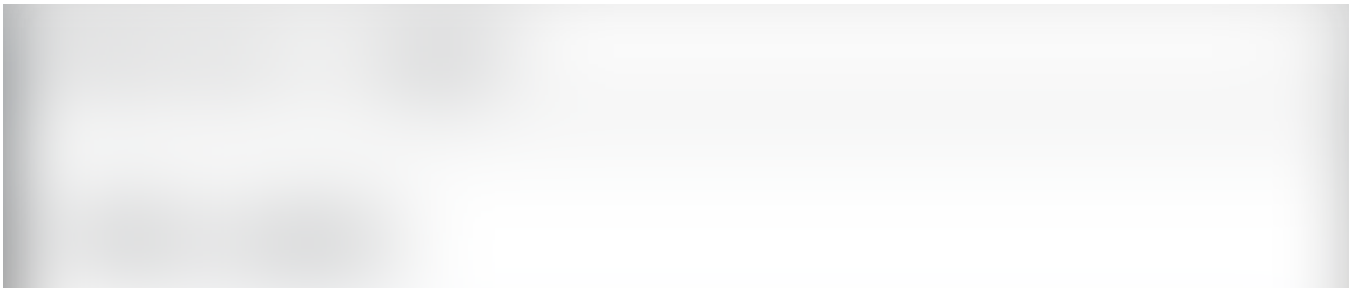
```
{
    "name": "social-login-root-folder",
    "version": "1.0.0",
    "description": "This is the root project for our frontend and backend",
    "main": "index.js",
    "repository": "https://github.com/FBosler/fb-tutorial-social-login",
    "author": "Fabian Bosler",
    "license": "MIT",
    "private": true,
    "scripts": {
        "dev-start": "concurrently \"cd server && yarn nodemon server.js\" \"cd client && yarn start\""
    },
    "devDependencies": {
        "concurrently": "^5.1.0"
    }
}
```

To download check out this Gist

Now, if we run `yarn dev-start` in our project root-folder, both the backend and the frontend will get booted up.

Sign up / Login modal

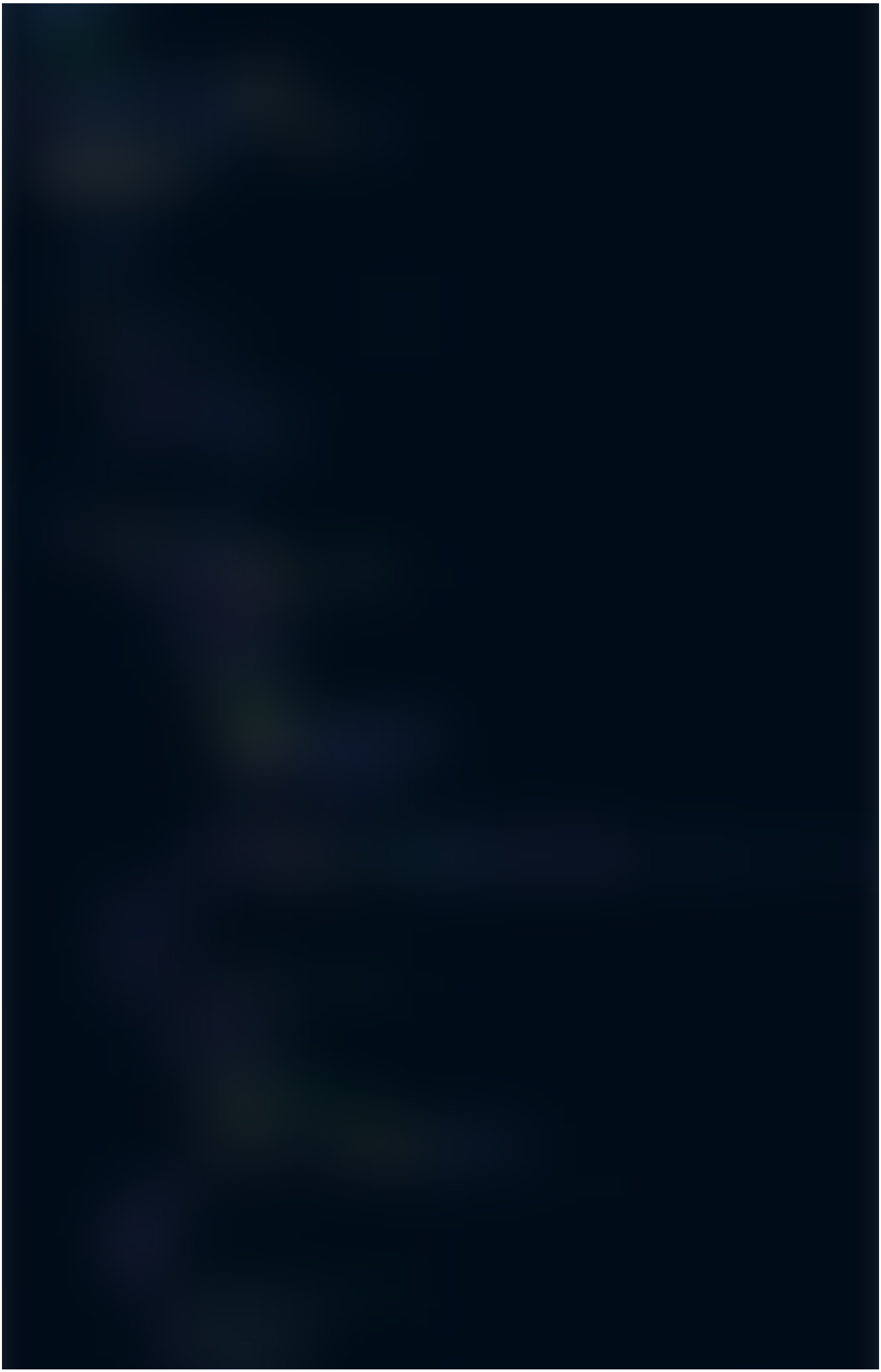
Remember, we had this Signup/Login modal in our frontend, which we want to connect now!



Signup/Login Modal

We are going to use `axios.js` to do so. Axios.js has some advantages over the native `fetch`.

To implement `axios` means we will have to navigate to our `client` folder `yarn add axios`, and then change our SignupLoginModal to include `axios` and accurately track our email changes and then submit them to our endpoint.



For code check Gist

However, right now, our frontend is running on port 3000, and our backend is running on port 5000. When making a post request to `/api/auth/register_login` we are trying to do that unsuccessfully on port 3000. To overcome that problem, we have to proxy our API requests in development.

To do so, we have to adjust our frontend `package.json` to include `"proxy": "http://localhost:5000"` . Adding the proxy will forward any unknown requests to our API server in development. Perfect!

Let's quickly validate that this is working

Successful signup with frontend

Summary

As always, you can find the final result in the corresponding repo!

In this article, you have learned how to build a web app with Node.js, Express.js, Passport.js, and MongoDB to authenticate users through a REST-endpoint via password and email. You have learned how to validate that the endpoint is working, and you managed to create users in your local database.

You also learned how to connect that endpoint with our previously created frontend. You learned how to proxy requests in development to the port your backend is running on. And you learned how to use concurrently to run the backend and frontend simultaneously.

Thanks for reading, stay tuned for next week when we will be talking about how to connect our third-party authentication and send out verification emails.

JavaScript Nodejs Programming Web Development Software Engineering

About Write Help Legal

Get the Medium app