



MERN Stack

[Intro](#)

[Node.js + Express.js](#)

[Environment variables](#)

[app.js](#)

[server.js](#)

[Postman](#)

[Express Router & Api Routes](#)

[MongoDB](#)

[Setup](#)

[Model + Schema](#)

[Controller](#)

[Post \(example of how to think a CRUD op in the backend \)](#)

[Fetching APIs in React](#)

[To Keep In Mind + Others](#)

[Login](#)

[Password hashing & dehashing](#)

[How to check if a hashed password == input password \(for Login\)](#)


[Input validation](#)

[JSON Web Tokens](#)

[Backend](#)

Intro

- React - frontend
- Node.js + Express.js - backend
 - Postman
- MongoDB - NoSQL database

 **React Notes**

Node.js + Express.js

- install nodemon globally
- to start: nodemon server.js

Environment variables

- create a .env file in the backend folder
- add it to .gitignore
- create variables
 - VARNAME = VALUE
- npm install dotenv
- require(dotenv).config()
- process.env.varname to access a varname

app.js

```
const express = require('express')

//to create the app:
const app = express()

app.get('/', (request, response) => {
}
```

server.js

```
//to listen for requests
app.listen(port-number, () => {
  //do smth
})
```

Postman

- download postman, log in
- create a new type of request and give it localhost:yourport/ and then check if it works

Express Router & Api Routes

```
const express = require('express')
const router = express.Router()
router.get/delete/patch/post('/yourRouter', (req, res) => {
  do-smth
})
```

- `router.get/.../...('/...', functionToCall)` is another way (the actual way, go to controller)
 - you then need to require it in the server.js:
`const yourRoutes = require('./routes/players')`
 - then use it:
`app.use(yourRoutes)`
 - you have to create different routes files for each type of routes you do (`playerRoutes.js`, `characterRoutes.js` and so on)
-

MongoDB

Setup

- go to mongodb/atlas
- create a database, an admin and add an ip address
- install mongodb and mongoose
- add the URI to `.env` so you can use it later (add your admin password)
- request mongoose and
`mongoose.connect(process.env.MONGO_URI)`
`.then(() => {`

```
app.listen(process.env.PORT, () => {  
  console.log('connect to db and listening on the wanted port');  
}  
})  
.catch( (dbError ) => { console.log( dbError ) } )
```

Model + Schema

```
const mongoose = require('mongoose')  
const Schema = mongoose.Schema;  
const playerSchema = new Schema({  
  id: {  
    type: Number,  
    required: true,  
  },  
  username: {  
    type: String,  
    required: true  
  }  
}, {timestamps: true}) → to show when it was created and last updated
```

```
module.exports = mongoose.model('Player', playerSchema)
```

you create a Player model with that schema

Controller

- in your controller, you will basically define CRUD functions and whatever other utility you need
- your functions will have a request and response parameters
- if everything is fine, your response will have a status like 200 and you can output a json or an object or id etc.
 - else, you return 400 or another error code

- in these functions, you basically do your functionalities
- your routes will call one of these function:
`router.get('/addPlayer', addPlayer)`
 where add player is a function defined and exported in your controller.js and imported in your routes
- your request has:
 - params (if you do for example /players/:id) then the id, that is passed in the URL, is the content of params
 - body - this is the actual data you transmit

Post (example of how to think a CRUD op in the backend)

- for posting a player, make sure you get the first free id or smth and generate inside the backend everything else that needs to be generated automatically
 - in your frontend, when you create a character or player or whatever, have every field in the constructor and get that player from the post request:

```
const addPlayer = async (req, res) =>{
  const {username, nickname, pictureURL} = req.body
  // create the player instance and add it to your database or whatever
  if(not good){ return res.status(400)...}
  res.status(200).json(player) => so you can get all the data in the frontend
  // also make sure to console log it, it helps w debugging a lot
}
```

Fetching APIs in React

```
fieldsToEdit = { username: 'marcel', ...}
```

```
axios.edit(`http://localhost:5000/api/players/${id}`, fieldsToEdit
```

```
.then( (response) => {
```

```
  // do something if needed
```

for example you could refresh the frontend context so it appears without refreshing

```
}  
.catch(error){..}
```

To Keep In Mind + Others

- it is important that your frontend and backend are on different port
- make sure you use cors on your backend so you can access it from your frontend
- always check the console for errors
- use as many try's and catches as possible
- when passing data to your backend, you need to make sure the names are 100% the same:
changing smth from playerId to playerId and going from 5 errors to 0 will make you break your monitor
- make sure your paths are correct
 - also, use postman, it's actually really useful and fast
 - also, if you want to add new data or delete data, do it from postman, it's way faster
- use `const myShit = require('myShit')`, don't use imports
- make sure you install packages in your backend folder, not in your main folder, it can't import from there for some reason
(I have a myApp repository that has both the backend and the frontend there and i installed cors both in myApp/node_modules and in myApp/backend/node_modules and it tried also didnt work; installing it in myApp/node also doesnt work)
- `console.log` everytime an api is ran successfully and actually output the entity you worked with, it's a life saver when debugging

Login

Password hashing & dehashing

npm install bcrypt

- use BCRYPT lib
 - `const bcrypt = require('bcrypt')`
- create a static function for your schema:
`userSchema.statics.signuup = async (username, email, password) =>`

```
const salt = await bcrypt.genSalt(10)
const hash = await bcrypt.hash(password, salt)
```

`}` → hash is your hashed password, the one you'll store in your db

- then use `this.create(user-parameters)` and then just call this function inside your controller

How to check if a hashed password == input password (for Login)

```
const match = await bcrypt.compare(password, user.password)
if (!match) throw Error('Wrong password')
```

use the login function that checks this in the controller to get the user and then create a JWT token for it

Input validation

npm install validator

```
if (!validator.isEmail(email)) {
  throw Error('Invalid email')
}
```

```
if (!validator.isStrongPassword(password)) {
  throw Error('Password not strong enough')
}
```

```
if (!validator.isAlphanumeric(username)) {  
    throw Error('Username must be alphanumeric')  
}
```

JSON Web Tokens

npm install jsonwebtoken

require('jsonwebtoken')

- if the signup is correct, you create a json web token and send it to the frontend
 - this indicates that you are logged in properly
 - also, when frontend tries to access a backend API, it can't do so without a json web token
- JWT → Header: algorithm for JWT | Payload: non-sensitive user data | Signature: Token
 - JWT → The server hashes together the header and payload and generates the signature
 - headerHash.payloadHash.signatureHash = the JWT

Backend

- you send a JWT Token after signing up (auto-login) and one when you login
- CREATE A TOKEN

```
const generateToken = (id) => {  
    return jwt.sign({ id: id }, process.env.JWT_SECRET, { expire
```

- const token = generateToken(user.id)
- res.status(200).json({user, token})