creating a model object and saving it, there is another way

**const user = await User.create({email , password})**

creates a document inside the collection, and returns that newly created document as object.

you don't have to create a user object first and then save it in this case.

npm i validator

has many validator functions.

**const {isEmail} = require('validator')**

**isEmail** is a method which you can pass onto the validator in schema.





This is the error thrown when a user is not created in database due to some **invalid input** based on **schema.**

**Unique** is not a validator in mongoose schema. When a duplicate is added in a unique declared field, this error is thrown. You can identify it using **code.**

A screen shot of a computer

Description automatically generated

const handleErrors = (err) => {

  console.log(err.message, err.code);

  // console.log(err);

  let errors = { email: '', password: '' };

  // duplicate email error

  if (err.code === 11000) {

    errors.email = 'that email is already registered';

    return errors;

  }

  // validation errors

  if (err.message.includes('user validation failed')) {

    // console.log(err);

    Object.values(err.errors).forEach(({ properties }) => {

      // console.log(val);

      // console.log(properties);

      errors[properties.path] = properties.message;

    });

  }

  return errors;

}

This is how you can use the **err** object thrown and convert it to a smaller useful error and send it as a response. **errors** which is returned is send. **res.json({errors})**

  catch(err) {

    //This handle error function will return a small usable object containing

    //error information. error object is thrown when user cannot be created.

    const errors = handleErrors(err);

    res.status(400).json({ errors });

  }

**A screen shot of a computer

Description automatically generated**

This error is sended as a response. **errors** contain errors related to **input fields.**

This code belongs to the same file where schema is made and model is exported.

//next() is required to continue because these are middleware,

//also known as pre and post hooks

// read more on doc

// fire a function after doc saved to db.(post)

//doc is the document which is finally saved.

userSchema.post('save', function (doc, next) {

  console.log('new user was created & saved', doc);

  next();

});

// fire a function before doc saved to db(pre)

//We don't receive 'doc' because this is before saving to database.

//use normal function, not arrow function because we want to use 'this'.

//this in this case refers to the instance of 'User' which is about to be saved.

userSchema.pre('save', function (next) {

  console.log('user about to be created & saved', this);

  next();

});

## Cookies

npm i cookie-parser

Parse Cookie header and populate req.cookies with an object keyed by the cookie names.

// cookies

const cookieParser = require('cookie-parser');

app.use(cookieParser());

//middleware, parses cookie to object which we can use.

app.get('/set-cookies', (req, res) => {

  // res.setHeader('Set-Cookie', 'newUser=true');

  //This is the default way of creating a cookie.

  //This is with the help of cookie-parser library.

  res.cookie('newUser', false);//(name , value , {options})

  res.cookie('isEmployee', true, { maxAge: 1000 \* 60 \* 60 \* 24, httpOnly: true });

  //maxAge : time in milliseconds when cookie will expire, cookie remains only till session.

  //httpOnly : bool , if true, you can't acces the cookie using javascript. 'document.cookie'

  //secure : true, only secure connection are alllow, eg https , NOT http

  res.send('you got the cookies!');

});

If you set cookies, they are send along with response.

If cookie already exist, it is replaced. If it does not, a new one is created.