Contents

[**1 - 3 Learn Node** 4](#_Toc36841081)

[**4 - Routing / Core Concept** 4](#_Toc36841082)

[**5 - Templating / Core Concept** 6](#_Toc36841083)

[**6 - Templating Helpers / Core Concept** 7](#_Toc36841084)

[**7 - Controllers and MVC Pattern / Core Concept** 8](#_Toc36841085)

[**8 - Middleware and Error Handling / Core Concept** 12](#_Toc36841086)

[9 - Creating our Store Model 14](#_Toc36841087)

[10 - Saving Stores and Using Mixins 18](#_Toc36841088)

[11 - Using Async Await 22](#_Toc36841089)

[12 - Flash Messages 23](#_Toc36841090)

[13 - Querying our Database for Stores 26](#_Toc36841091)

[14 - Creating an Editing Flow for Stores 27](#_Toc36841092)

[15 - Saving Lat & Lng for each store 29](#_Toc36841093)

[16 - Geocoding Data with Google Maps 32](#_Toc36841094)

[17 - Quick Data Visualization Tip 34](#_Toc36841095)

[18 - Uploading & Resizing Images with Middleware 34](#_Toc36841096)

This document will be kept up to date as we progress through the class.

What the instructor(s) use

Editor + Terminal

[Visual Studio Code](https://code.visualstudio.com/) is my current editor which I switched to in September 2017 after years of Sublime Text.

[Sublime Text](http://sublimetext.com/3dev) V3 beta is used in most of my video tutorials — I write more about my setup in [my book](http://sublimetextbook.com/).

I created a theme called Cobalt2 for [VS Code](https://marketplace.visualstudio.com/items?itemName=wesbos.theme-cobalt2), [Sublime Text](https://packagecontrol.io/packages/Theme%20-%20Cobalt2) and [iTerm2](https://github.com/wesbos/Cobalt2-iterm) v3 which I use.

I’m currently using [Operator Mono](http://www.typography.com/fonts/operator/overview/) for a font. Yes I paid the $200 for it. Before that I used [Inconsolata](http://www.levien.com/type/myfonts/inconsolata.html) for many years.

I use Operator Mono in VS Code and Hyper Term

I’m currently switching between Firefox Developer and Chrome Canary as my main browser

The cursive font you see is just the italic weight of Operator Mono. Your theme must have italics in it for it to work – my Cobalt2 uses italics heavily.

I currently use [Hyper](https://hyper.is/) for my terminal, but many of my videos show me using iTerm2.

For Sublime Text, The JavaScript suggestions you see in my videos is called [JavaScript Completions](https://packagecontrol.io/packages/JavaScript%20Completions) – you might have to play with the settings to get it to not popup some of the node completions. For VS Code, there are no plugins needed to get this.

You can see most of my config files (.eslint, VS Code settings) over at my [Dot Files repo](https://github.com/wesbos/dotfiles) but please don’t just take these settings carte blanche – they work for me but might not be the best

Desktop Apps

I use [Text Expander](https://smilesoftware.com/textexpander) for canned responses and for most of my code snippets

I’m a huge [Alfred](https://www.alfredapp.com/) fan and have many [workflows](http://wes.io/fY7q).

I use [Things App](https://culturedcode.com/things/) with a [GTD mindset](http://gettingthingsdone.com/) for managing my tasks.

I design almost entirely in [Sketch](https://www.sketchapp.com/)

I backup with 2 time machines local HDDs, [Dropbox](https://dropbox.com/) for most files, and [BackBlaze](https://secure.backblaze.com/r/008x60) for my entire hard drive.

The cool Emoji picker you see me use in my videos is [Rocket](http://matthewpalmer.net/rocket/)

Recording

I use [ScreenFlow](https://www.telestream.net/screenflow/) to record and edit my screencasts

I wrote about my [animated gif workflow here](https://wesbos.com/animated-gif-workflow/)

My mic is a [Heil PR40](https://www.amazon.com/gp/product/B000SOYOTQ/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B000SOYOTQ&linkCode=as2&tag=webo080-20&linkId=YH6ZR6GSGBI2JOJF) on a [PL2T boom arm](https://www.amazon.com/gp/product/B000SZVZ74/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B000SZVZ74&linkCode=as2&tag=webo080-20&linkId=HJ6K3KRLOL5XJJSC). Many of my videos were recorded on the cheaper but still amazing [AT2020](https://www.amazon.com/gp/product/B0006H92QK/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B0006H92QK&linkCode=as2&tag=webo080-20&linkId=PHY64E6Y43GHI55T)

I use a [DBX286s](https://www.amazon.com/gp/product/B004LWH79A/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B004LWH79A&linkCode=as2&tag=webo080-20&linkId=E3KFCLCQIXXLC7W4) to do almost all my audio compression and processing as a I record and pipe it into my computer with a [Focusrite 2i2](https://www.amazon.com/gp/product/B005OZE9SA/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B005OZE9SA&linkCode=as2&tag=webo080-20&linkId=ELZASVHMTR6ZTNYL)

My webcam is a [Logitech HD Pro c920](https://www.amazon.com/gp/product/B006JH8T3S/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=webo080-20&camp=1789&creative=9325&linkCode=as2&creativeASIN=B006JH8T3S&linkId=65a64ee5dd16f2fa18faf2f2d33d0538)

All this gear is pretty demanding, so I use an [Anker 7/3 port USB 3.0 hub](https://www.amazon.com/gp/product/B00VDVCQ84/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00VDVCQ84&linkCode=as2&tag=webo080-20&linkId=d8d1e7a3e461ab7921373dbc07dfbe38)~~DIED 😵~~ [Amazon Basics 10 Port USB Hub](https://amzn.to/2kVaDCz) to make it all work at once while only having to plug in a single cable.

Desk Setup

My main monitor is a 32″ 4k [LG 32ud59](https://amzn.to/2SYuX8G)

To my right I run a vertical 27″ 4k [Dell 4k P2715Q](https://www.amazon.com/gp/product/B00PC9HFO8/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00PC9HFO8&linkCode=as2&tag=webo080-20&linkId=5VCVP6ZUPF5GLCUJ) and an open loaded macbook pro nested on a [Rain mStand](https://www.amazon.com/gp/product/B000OOYECC/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B000OOYECC&linkCode=as2&tag=webo080-20&linkId=XWDACWNC2JWRF3DQ)

I sit on a [polished aluminum Herman Miller aeron](https://www.amazon.com/gp/product/B000LTAO8S/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B000LTAO8S&linkCode=as2&tag=webo080-20&linkId=6G4M6ZR4CYKT3TM5)

My desk is an [8 foot oak butcher block countertop](http://www.ikea.com/ca/en/catalog/products/20274961/) on legs

Here is a [photo of my setup](https://twitter.com/wesbos/status/1060643936140738560)

Everything — Monitors, Ethernet, Hard Drives, Webcam, and Power — is plugged in via a single USB cable and piped into my [Caldigit TS3 Plus](https://amzn.to/2D9Jnx0) – This is a pretty sweet setup!

I currently use a 2019 16″ MacBook Pro 2.3 GHz 8-Core Intel Core i9 with 32gb memory

My last laptop was a 2017 15″ MacBook Pro 3.1GHz i7 with 16gb memory

**1 - 3 Learn Node**

[Live Demo](https://now-thats-delicious-anja.herokuapp.com/)

NTS: directory: repo/../../assets/Learn-Node videos: /Movies/../../assets/Learn-Node

Pug is the template engine we are using to generate the HTML.

Error: exception in initAndListen: NonExistentPath: Data directory /data/db not found., terminating

Express - minimalist, unopinionated web framework for Node.js

Environmental variables (where you store your sensitive stuff)

npm assets is for web pack to compile sass and css not for node

I have created some things to get you started easily.

Startmongo.cmd

Startwebsite.cmd

Readme – instructor.txt

These can be found in most of the video homework completed stepped folders for this class

**4 - Routing / Core Concept**

Import express

Grab the router from express Const router = express.Router()

Then you define your routes (in index.js for this course).

In app.js we have: // After allllll that above middleware, we finally handle our own routes! app.use(‘/‘, routes);

However you can have multiple route handlers (such as for admin. )  
app.use(‘/admin‘, adminRoutes);

How it works: You get the url and then you have a callback function that will run whenever someone visits that specific url. It gives you three things:

Router

router.get(‘/‘, (req, res) => {

res.send(‘Hey! It works!’);

});

**request**: which is an object full of information that is coming in.

**response**: object full of methods for sending data back to the user.

**next:** third value which we will review in our middleware file because it’s possible you don’t wanna send data back at the route but you want to pass it off to somebody else.

When you console.log in node, you see it in there terminal.

One of the methods we can use to send data: res.send(“hey”); res.json

We can also use it to send JSON back.

Never send data more than once or you will get an error that says can’t set headers after they are sent.

We can also pull in data from the url.

Lets say there are url parameters like… localhost:777/?name=wes&age=100

How to get the data that is in the url? That is part of the request! You can access the url params like so: res.send(req.query.name);

Or, you can just return the JSON from the query like so: res.json(req.query)

The reason we have the query property on the request, is because if we head back to app.js, you will see on LINE 27 & 28:

// Takes the raw requests and turns them into usable properties on req.body

app.use(bodyParser.json())

app.use(bodyParser.urlencoded({ extended: true }))

^ this is middleware which before we even hit the routes, express will check the url and check if anyone has posted data from a form element, and it will put all the data in the request so we can easily access it via req.query. Or request.params

router.get('/reverse/:variable’, (req, res)=>{

res.send(‘it works”);

})

To access the :variable, we do: res.send(req.params.name)

request.body for posted parameters

[Express 4.x - API Reference](https://expressjs.com/en/api.html#req)

response.render();

**5 - Templating / Core Concept**

When we are not dealing with API endpoints is res.render() which will render out a template for us.

This will be about how to use templating language called Pug.

Jade/Pug is our templating language.

App.js line 20 or 21 we have a views folder, and that is where all our pug files live.

// view engine setup

app.set("views", path.join(\_\_dirname, "views")) // this is the folder where we keep our pug files

app.set("view engine", "pug") // we use the engine pug, mustache or EJS work great too

(You can use alternatives to Pug)

res.render() takes two parameters. First, it needs name of the template to render. The second thing is some local variables (some info that it needs)

**PUG**

How pug works is you don’t write open and closing tags. Instead you just do the open tag that you want, and then the text beside it like so: p Hello!

you have: tag, space and then all of your elements inside of it If you wish to nest elements, you need to indent it one level:

div

p Hello!

You can add classes and ids like so:

div.wrapper

p.hello Hello!

span#yo Yo!!!

If it’s just a div, you don’t need to specify div. If you leave the element type out it will assume that it is always a div.

We also need to know about attributes… you put them in parenthesis like so: img(src=“dog.jpg” alt=“dog”) (you can separate the attributes with commas)

If you want to put something on it’s own line without it rendering as an element you use | like so:

h2

| Hello

em How are you?

(End pug)

To get info from the request into the “hello” template, you use the second parameter on the render() method.

PUG CONTINUED

In pug, you can reference the variables that were passed in the second param of the render() method.

You interpolate a variable inside of text in pug like #{dog}

If you want to use a variable into an attribute, you do it like in javascript:

img.dog(src=“dog.jpg” alt=`dog ${dog}`)

[6F024006 96ED 45EE A14D BF1B4E606524](https://notes.anjagusev.com/static/2fc064e25eca4177d633da5b03a742e9/01387/6F024006-96ED-45EE-A14D-BF1B4E606524.png)

If you need to make a variable in pug, you can do it using a dash like so:

-const upDog = dog.toUpperCase();[5F3ADA7E 71F0 4F5A 9D6B E865738FCA80](https://notes.anjagusev.com/static/b6d995952b987e0ecc88af25f45b4eb9/7adc3/5F3ADA7E-71F0-4F5A-9D6B-E865738FCA80.png)

You can run javascript in Pug.

You can reuse parts of website like footers and headers.

You do that by extending templates, and making “layouts”

We want to extend the layout.pug

extends layout

block content

p Hello

You can also overwrite the layout (default) portions by referencing them in the extends file.

(end pug)

**6 - Templating Helpers / Core Concept**

Sometimes you need data available in every single request.

Wes creates a helpers.js file and in that file puts any helper libraries or data that is needed in every single template.

exports.menu = [

{ slug: "/stores", title: "Stores", icon: "store" },

{ slug: "/tags", title: "Tags", icon: "tag" },

{ slug: "/top", title: "Top", icon: "top" },

{ slug: "/add", title: "Add", icon: "add" },

{ slug: "/map", title: "Map", icon: "map" },

]

each item in h.menu

li.nav\_\_item

a.nav\_\_link(href=item.slug, class=(currentPath.startsWith(item.slug) ? 'nav\_\_link--active' : ''))

!= h.icon(item.icon)

span #{item.title}

In middleware we can append variables to all our requests

app.js

// pass variables to our templates + all requests

app.use((req, res, next) => {

res.locals.h = helpers

res.locals.flashes = req.flash()

res.locals.user = req.user || null

res.locals.currentPath = req.path

next()

})

You can export arrays, strings ,or entire libraries

Example: exports.moment = require(‘moment’);

**7 - Controllers and MVC Pattern / Core Concept**

Instead of writing directly to the routes, we factor that code out to it’s own controller.

There is a folder called controllers.

Wes likes to make a controller for every specific area of the website.

On the controller, we have to export all of the different functions that happen.

On our exports variable, (global variable, anything you put on it is importable in another file). Example:

exports.homePage = (req, res) => {

res.render("index")

}

To use it, import:

const storeController = require(“../controllers/storeController”);

// Do work here

router.get(“/“, storeController.homePage);

Just like C# MVC patterns.

M = Model (ex: db access)

The part of our application that will deal with the database or any data-related functionality.

Think of the Model as a way to abstract the data into a “model” of something that can store the data. So we can create functions that do things like save, delete, update and other functions to the data without messy manipulation.

V = View (pug)

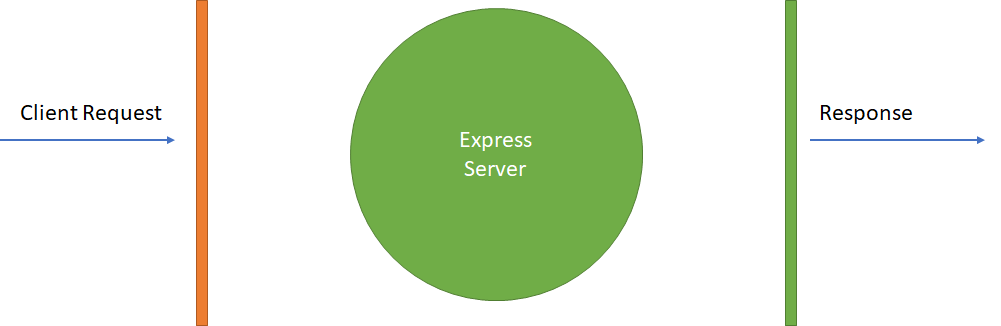
Everything the user will see. Basically, the pages that we’re going to send to the client.

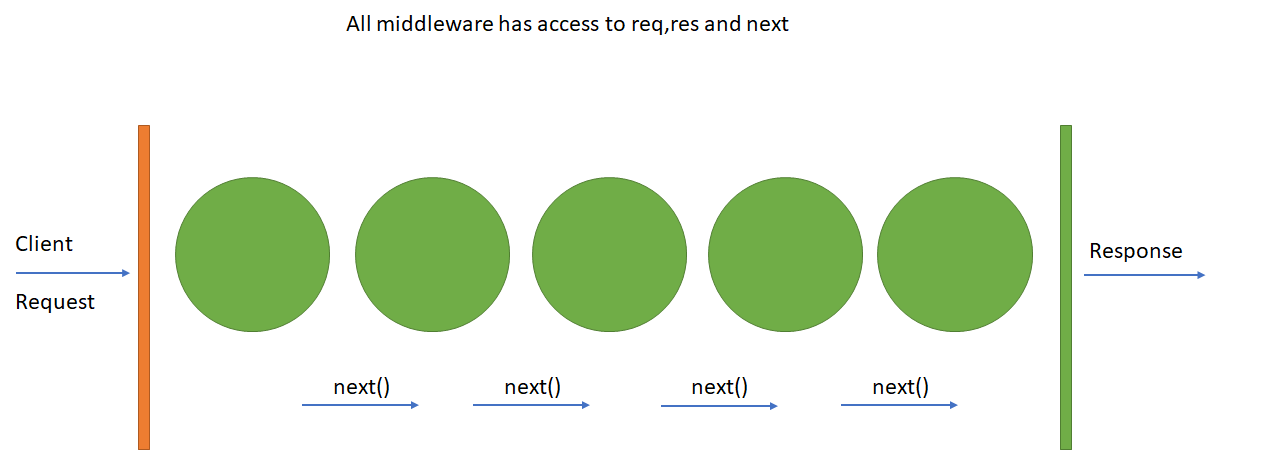
C = Controller (ex: controls flow of data )

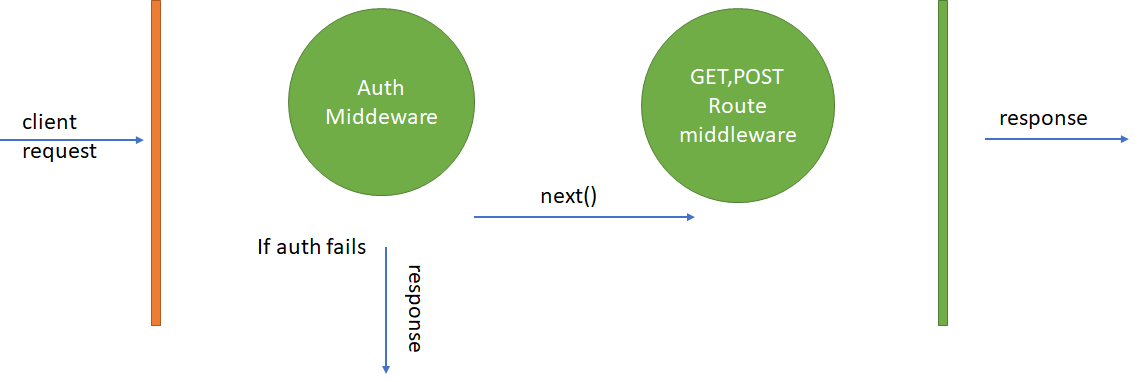
The logic of our site, and the glue between models and views. Call the models to get the data, then put that data on the views to be sent to the users.

Middleware - Middleware has access to request and response object

* Execute any code.
* Make changes to the request and the response objects.
* End the request-response cycle.
* Call the next middleware in the stack.







Then you can add layers to the middleware stack by calling .use as in app.use

// express

app.use(express.static(..));

// middleware

stack.use(function(data, next) {

next();

});

App.js mostly routes and app.use for middleware access

Examples of use

Cookies

Parsers

Authentication

Image serving

Routes

Session handling

Error handling

Local variable helpers

Passing any parameter to Next is assumed to be an error you are passing

Node has a callback type feature called a Promise. A callback is something you often see waiting for events to happen, task to complete, messages to be delivered and can be used for mundane task like updating the UI thread from the underlying thread.

Example :

var promise = doSomethingAsync()

promise.then(onFulfilled, onRejected)

A promise is just an enhanced callback. When a promise is returned, it can have 2 outputs. This is defined by the 'then clause'.

promise.then(onFulfilled, onRejected)

Either the operation can be a success which is denoted by the 'onFulfilled' parameter. Or it can have an error which is denoted by the 'onRejected' parameter.

promise.then(onFulfilled, onRejected)

\*\*\* Note \*\*\*

So the key aspect of a promise is the return value. There is no concept of a return value when working with normal callbacks in Node.js. Because of the return value, we have more control of how the callback function can be defined.

Get access to promise with a module called promise.

**npm install promise**

When defining promises, it needs to be noted that the "then" method itself returns a promise. So in a sense, promises can be nested or chained to each other.

Slugs are part of Markdown framework which is another class already posted on your share.

A slug is a URL markdown friendly way to show a title in the URL or a string. Here is a JAVAScript function to create a slug from a string.

function convertToSlug(Text)

{

return Text

.toLowerCase()

.replace(/[^\w ]+/g,'')

.replace(/ +/g,'-')

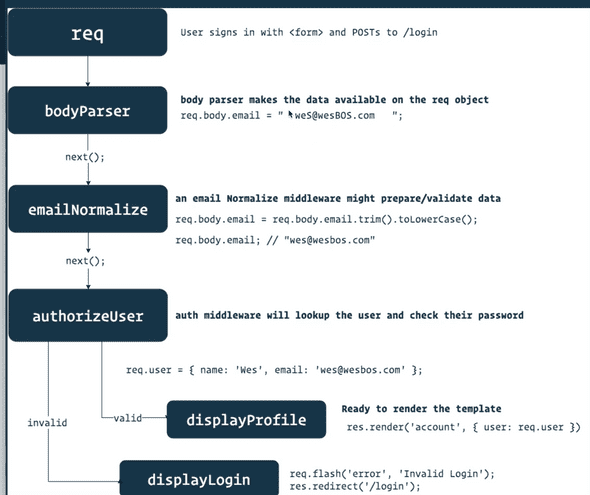
;

}

Wes Recommend doing data normalization close to the model. I agree with Wes on this point completely. It is where you go for data change and it should be where you go for normalization.

**8 - Middleware and Error Handling / Core Concept**

What is middleware? Stuff that happens between the request and response.

[](https://notes.anjagusev.com/static/22f470187c8e13698f59a8eca16f4a8c/08b59/7EB5FABE-6FDA-4646-9D75-AF4F4FF9600C.png)

The request goes through an assembly line of middleware and tacks on things

If you are writing middleware you need to pass the next

exports.myMiddleware = (req, res, next) => {

req.name = "wes"

next()

}

What next does is it says I am done my work in this middleware and pass it off to the next piece.

How do we use that then? We go to our routes. Instead of router.get(“/“, storeController.homePage);

You would do router.get(“/“, storeController.myMiddleware, storeController.homePage);

And since storeController.myMiddleware calls .next() it will pass it to storeController.homePage.

Inside the homePage index method, we can now console log req.name; because we are setting it in the middleware.

Middleware allows us to run some code after the request but before the response happens.

This is an example of route specific middleware. However express also has global middleware, where every single request that goes through the application will run through the middleware before it even gets to your router.

That is generally where we introduce any application wide functions or features or plugins.

Where does that happen? In app.js.

Anywhere you see app.use() it means we are using that global middleware. Even before we get to our routes, we are going to run all those middleware.

app.use(express.static(path.join(\_\_dirname, ‘public’))); -> means that if somebody requests anything from that public folder, it’s going to server that actual file before it even tries to think it’s a route or not.

app.use(bodyParser.json()); app.use(bodyParser.urlencode()); -> anytime anybody submits data via a form tag, you will get the data that was submitted on the request.body and it will all be url encoded.

bodyParser is a package in express.

app.use(expressValidator()) -> validates any email stuff

app.use(cookieParser()): -> by default express doesn’t do anything with cookies, but if you want to set cookies you can. you can use cookieParser middleware which will give you req.cookies. We will need that to work with login sessions.

Sessions will allow us to store data on a visitor, any info you need to go from request to request.

After all that middleware, we finally inject our own routes.

app.use(‘/‘, routes);

Our routes file exports our router which will after handle the individual forward slash routes that are needed.

**Error Handlers**

After routes, we have more middleware so we can run stuff If something happens when grabbing the route as a safety net… error handlers!

If no route found, we queue up the next middleware.

app.use(errorHandlers.notFound); -> runs when we try to access a route that doesn’t exist.

errorHandlers.js -> own file for errorHandlers.

We have a notFound middleware which will make a new error and set the status to 404, and then call next.

If you pass next() something it will assume what you are passing it is an error.

So we send some error info and pass it along, and the next middleware.

If we are in production, we handle the errors one way and if we are in development we handle it another way.

Wes added a function to error handling to highlight the file names where the errors occur for visibility.

## 9 - Creating our Store Model

\*\*\* NOTE \*\*\*

If you’re getting a URIError: URI malformed error when running npm start, break out your environment variables. Go into variables.env and split the URI like this MONGO\_URI=mongodb://host.com:port DB\_USER=username and DB\_PASS=password. Then inside your start.js replace mongoose.connect(process.env.DATABASE) with mongoose.connect(process.env.MONGO\_URI, {user: process.env.DB\_USER, pass: process.env.DB\_PASS});. I had issues connecting to my mongodb because my password contained symbols.

\*\*\* MONGOD complains that there is no /data/db folder

Note:

MongoDB also has an option where you can create the data directory in another location, but that's generally not a good idea, because it just slightly complicates things such as DB recovery, because you always have to specify the db-path manually. I wouldn't recommend doing that.

Edit:

the error message you're getting is **"Unable to create/open lock file: /data/db/mongod.lock errno:13 Permission denied"**. The directory you created doesn't seem to have the correct permissions and ownership -- it needs to be writable by the user who runs the MongoDB process.

To see the permissions and ownership of the '/data/db/' directory, do this: (this is what the permissions and ownership should look like)

$ ls -ld /data/db/

drwxr-xr-x 4 mongod mongod 4096 Oct 26 10:31 /data/db/

The left side 'drwxr-xr-x' shows the permissions for the User, Group, and Others. 'mongod mongod' shows who owns the directory, and which group that directory belongs to. Both are called 'mongod' in this case.

**If your '/data/db' directory doesn't have the permissions and ownership above, do this**:

First check what user and group your mongo user has:

# grep mongo /etc/passwd

mongod:x:498:496:mongod:/var/lib/mongo:/bin/false

You should have an entry for mongod in /etc/passwd , as it's a daemon.

sudo chmod 0755 /data/db

sudo chown -R 498:496 /data/db # using the user-id , group-id

You can also use the user-name and group-name, as follows: (they can be found in /etc/passwd and /etc/group )

sudo chown -R mongod:mongod /data/db

that should make it work..

In the comments below, some people used this:

sudo chown -R `id -u` /data/db

sudo chmod -R go+w /data/db

or

sudo chown -R $USER /data/db

sudo chmod -R go+w /data/db

The disadvantage is that $USER is an account which has a login shell. Daemons should ideally not have a shell for security reasons, that's why you see /bin/false in the grep of the password file above.

------------------------

Models are where our data is going to be stored, and before we can create a piece of data we need to describe what that data will look like.

A model can tell what type of data is expected (string ,array), cleanup before the data is saved, creating a slug.

MongoDB can be a loose database, meaning you don’t need to specify what your data will look like ahead of time.

We interface with mongoldb with mongoose package

const mongoose = require(‘mongoose’);

We also need to tell mongoose that the promise to use is the global promise. mongoose.Promise = global.Promise What does that mean?

When we get into querying our db, there are a few ways we can wait for the data to get back because it happens asynchronously. You can uses the built in callbacks, you can use external library like bluebird or since we are learning about async await, we are using the built in es6 promise.

We set the mongoose property to be global (sort of like the window in browser).

Don’t put things on the global, it’s generally not what you want.

we import slugs, which allow us to make url friendly names const slug = require(‘slugs’);

If the main thing you are exporting from a file is going to be importable, then you can put it on modules.exports. for example modules.export=router

However for storeController.js we are exporting exports.homePage .

When you import a package, is the main thing you import from it a function or are you just importing an object that has many properties on it?

Do all your data normalization as close to the model as possible.

For the model, you can just put the property name and then the type, but you can also pass in an object for the property.

How do we make mongo know about the model?

Go to start.js file. You only need to import it once. It’s using a singleton, once you import it you don’t have to import it in every single file.

The slug property in our Store model is auto generated whenever someone saves.

What we use for that is a pre-save hook in MongoDB. Before someone saves a store, we are going to auto-generate the slug field.

We do that with this code:

storeSchema.pre(‘save’, function(next){

this.slug = slug(this.name);

})

(Don’t use arrow function because we need to reference this, which is the store)

We call .next() because this is pre-save and we want to pass it on to the save.

Right now, the pre-save will run overtime we save but we only want it to run when the name is updated.

Modify method like so:

if(!this.isModified(‘name’)){

next();//skip it

return; //stop the function from running ( you can also do return next(); in one line

}

this.slug=slug(this.name);

next();

});

**Finalized Store Model**

const storeSchema = new mongoose.Schema(

{

name: {

type: String,

trim: true,

required: "Please enter a store name!",

},

slug: String,

description: {

type: String,

trim: true,

},

tags: [String],

created: {

type: Date,

default: Date.now,

},

location: {

type: {

type: String,

default: "Point",

},

coordinates: [

{

type: Number,

required: "You must supply coordinates",

},

],

address: {

type: String,

required: "You must supply an address!",

},

},

photo: String,

author: {

type: mongoose.Schema.ObjectId,

ref: "User",

required: "You must supply an author",

},

},

{

toJSON: { virtuals: true },

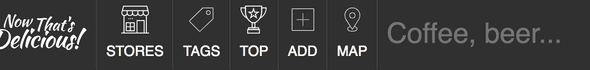
toObject: { virtuals: true },

}

)

## 10 - Saving Stores and Using Mixins

We are going to make the ADD + button on the nav menu functional by adding a route.

[](https://notes.anjagusev.com/static/905f14a041fa4982b653014636d4127f/6095e/682B4A77-F4AE-4AF3-936D-1BF5D6F62A40.png)

Add route: router.get(“/add”, storeController.addStore);

Add method in storeController to render a template.

exports.addStore = (req, res) => {

res.render(‘editStore’, { title:'Add Store'});

};

That returns an error because we have not created the associated view. We add a view called editStore.pug and access the title in the template like so #{title} ->

extends layout

block content

.inner

h2 #{title}

You could also do h2= title if there was no additional text needed on that element, just the variable.

We need a form, but instead of putting it in editStore.pug we will include the store form in a separate file which we can then pull in, so it’s like a reusable component.

Wes likes to do that with a mixin. A mixin is like a function in javascript, you can pass it some data and it will return to you some HTML that needs to be displayed.

You import the mixing in the view folder include mixins/\_storeForm That will make all the mixins defined in that file available as functions inside of the view pug file.

You can have multiple mixins per file, but good practice to keep it one.

To use a mixin, you access it like so: +storeForm()

You can pass info to the mixin like so: +storeForm({name:’Tim Hortons’}) and access it in the mixin.

In pug you can do javascript like so:

- const choices : [‘Wifi’, ‘Open Late’]

We created this form:

mixin storeForm(store = {})

form(action="/add" method="POST" enctype="multipart/form-data" class="card")

label(for="name") Name

input(type="text" name="name")

label(for="description") Description

textarea(name="description")

- const choices = ['Wifi', 'Open Late', 'Family Friendly', 'Vegetarian', 'Licensed' ]

ul.tags

each choice in choices

.tag.tag\_\_choice

input(type="checkbox" id=choice value=choice name="tags")

label(for=choice) #{choice}

input(type=“submit” value=“Save →” class=“button”)

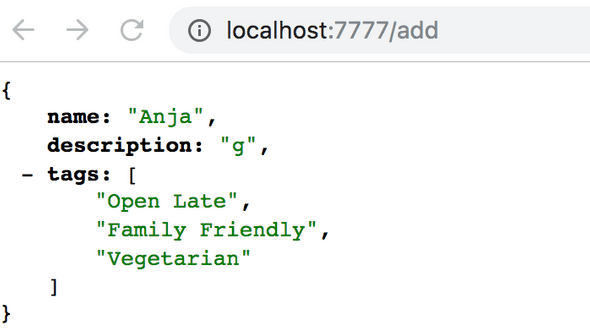
However when we submit it we get a 404 error because we need to add a POST route: router.post(“/add”, storeController.createStore);

In storeController we need to add a createStore function.

exports.createStore = (req, res) => {

res.json(req.body)

}

[](https://notes.anjagusev.com/static/15a84377b09cc7dca37afb2c7f11dcb1/28569/DC818237-4B0B-407D-9E9A-FDB9CD66774C.png)

The best way to store forms is as a MIXIN. This can be added to a page without retyping the code over and over.

Wes Bos uses \_ as the name of a Mixin. Which is not required but as he says, it is a good standard.

Think of Mixin as a way that you can bypass the fact that in JavaScript and hence NODE you can only inherit from a single object. The mix of two objects, objecta and object is what you may consider a mixin. In simple terms anyway. “A [mixin](https://en.wikipedia.org/wiki/Mixin) is a class containing methods that can be used by other classes without a need to inherit from it.”

I will post an event MIXIN.js as part of the homework for Video 16 after I return. This is a way to create an event handler you can subscribe to.

A mixin for PUG is a way to create reusable blocks of PUG code. This makes editing reused logic centralized.

#{ … } is interpolation. What is interpolation? Interpolation is simply a trick that lets you directly inject variables into a string. In JAVAScript and NODE it looks like #{ }

We use MONGOOSE for accessing Mongo DB

We need to import our SCHEMA to use it in the DB. We can get that from the Mongoose variable. This is a result of the singleton reference of Mongoose allowing it to be imported only once.

Using a strict schema means the only fields we will see is the ones we define in our schema.

You have to call the controller.save to save data to the DB. Until you do this all the DB work you are doing is temporary and not actually in the DB but is instead properties of the schema object.

By setting mongoose promise to a global promise you can use promise throughout the use of mongoose without redeclaring Promise.

Async Await : An await function can only be called inside a async function. A regular function cannot call await.

Because of the previous statement. There is no way to call await inside top-level code.

Await respects “then” logic like a promise

A flash is a message attached to the request. The flash is a middleware that allows us to flash messages to the user. This is obtained like this. app.use(flash());

These can be available after installing flash module

npm install express-session --save

npm install cookie-parser --save

npm install connect-flash –save

You can put this directly in your ES6 code and import to UI like this.

<% if(success\_msg) { %>

<div class="alert alert-success">

<%= success\_msg %>

</div>

<% } %>

<% if(error\_msg) { %>

<div class="alert alert-danger">

<%= error\_msg %>

</div>

<% } %>

## 11 - Using Async Await

In storeController, we need access to the database via mongoose so we need to import it, and we also need access to the schema.

For the schema, since we have already imported it once in our start.js file, we can simply reference it off the mongoose variable because it’s a singleton, which means we can import our models once and reference them anywhere in our application.

const Store = mongoose.model("Store");

We use “Store” because we named it that in Store.js where we exported it.

We create the store in Mongo like so: const store = new Store(req.body). It will know only to insert the fields that belong to the model because we are using a strict schema. You save to the db using store.save()

With Mongoose, before you had to have callbacks because JS is asynchronous and you needed to know whether saving was successful or not before redirecting the user. Now we can use Promises, which you can chain together.

store

.save()

.then(store => {

return Store.find();

})

.then(stores => {

res.render(“storeList”, { stores: stores });

})

.catch(err => {

throw Error(err);

});

Let’s say we only want to console.log() it worked only once the save has finished. The way that we can do that is we can tell javascript to wait until something has completed using async await.

You do that by marking the function as async which lets js know it will possibly have awaits in it. Then go in front of the thing that returns a promise and await it. This means we will not move onto the next line until the save has successfully happened.

exports.createStore = async (req, res) => {

const store = new Store(req.body);

await store.save()

console.log(“It worked!”);

};

To catch errors, wrap entire method in a try catch which is ugly, or we can use composition. Wrap createStore in another function that will catch any errors.

Wes already coded this in errorHandlers.js

exports.catchErrors = fn => {

return function(req, res, next) {

return fn(req, res, next).catch(next)

}

}

This is basically a middleware that we will use which will take in the function, and it will return a function that calls it, and if there is an error it will catch it and call .next().

If any of our routes throw errors, the errorHandler will catch the error and call Next and say something happened so we cannot use the route and we will pass it along the middleware chain (not found, validation error, or show the dev errors or production errors).

If you do not wrap async await in try catch you need to wrap it in catchErrors function.

We need to go to our routes and wrap createStore in the errorHandler method because it is using async await, using composition (wrapping one function in another function).

Import {catchErrors} into routes and wrap createStore like so: router.post(“/add”, catchErrors(storeController.createStore));

## 12 - Flash Messages

I don’t want to redirect them to a page, I just want to tell them something. But next time they load a page, I want to send them a bit of information.

The user can refresh the page to remove it or close it to dismiss it.

To create a flash you do: req.flash();

We have flash available because we imported the flash middleware.

Flash takes two things:

1. type of slash (success, error, warning, you can make up your own).
2. Message

req.flash(

“success”,

`Successfully Created ${store.name}. Care to leave a review?`

);

If you load them like:

exports.homePage = (req, res) => {

console.log(req.name)

req.flash("error", "Something happened")

req.flash("info", "Something happened")

req.flash("warning", "Something happened")

req.flash("success", "Something happened")

res.render("index")

}

They won’t appear until you redirect to a new page (or refresh same page) because flashes only show up on the next view.

Flashes is available because in app.js we have the following line:

// pass variables to our templates + all requests

app.use((req, res, next) => {

res.locals.h = helpers

res.locals.flashes = req.flash()

res.locals.user = req.user || null

res.locals.currentPath = req.path

next()

})

Which makes it available in all our templates. In our pug template we have conditional logic to show it:

block messages

if locals.flashes

.inner

.flash-messages

- const categories = Object.keys(locals.flashes)

each category in categories

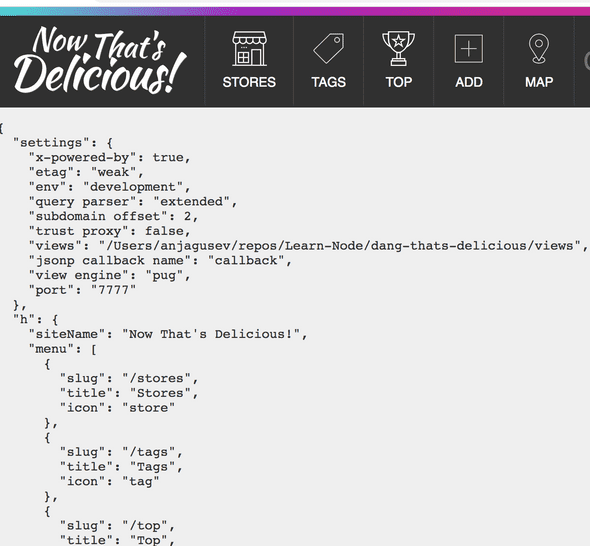
each message in flashes[category]

.flash(class=`flash--${category}`)

p.flash\_\_text!= message

button.flash\_\_remove(onClick=“this.parentElement.remove()”) &times;

To dump data in our layout to view locals do the following: pre= h.dump(locals) (from helpers file).

[](https://notes.anjagusev.com/static/f0c435268f1050f03334144be42c5734/c20f6/EB5A3D71-4968-494D-849D-0048D7F6293F.png)

p.flash\_\_text!= message this parses the HTML in the string, if you just did p.flash\_\_text= message it would display the html like <p>Test</p>

They allow us to show information on the page, we use middleware to pass those values to our locals and then in our layout we check if there are any flashes and we show those on the page.

Now instead of redirect to the homepage after a new store is created, we are going to redirect to the newly created store by doing:

res.redirect(`/store/${store.slug}`)

But in order to access the slug property, we need to refactor the save like so: const store = await (new Store(req.body)).save();

This will make the generated property (slug) available to us for the redirect.

exports.createStore = async (req, res) => {

const store = await new Store(req.body).save()

req.flash(

"success",

`Successfully Created ${store.name}. Care to leave a review?`

)

res.redirect(`/store/${store.slug}`)

}

## 13 - Querying our Database for Stores

We are going to display the stores in the database on the homepage and on the stores tab. We need a controller method that is going to run on both of those routes.

Modify homepage route and add another route for /stores and wrap them in error handlers since they will be async await.

router.get(“/“, catchErrors(storeController.getStores));

router.get(“/stores”, catchErrors(storeController.getStores));

Now we create another method in storeController.js and make it async await. This method is responsible for returning all the stores:

exports.getStores = async (req, res) => {

//1. Query database for a list of all stores

const stores = await Store.find()

console.log(stores)

res.render("stores", { title: "Stores", storres })

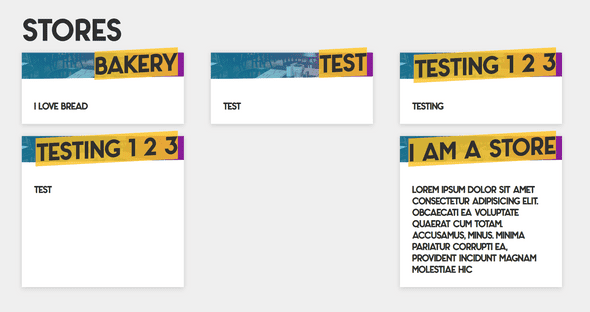
}

We want to make the stores variable available to our actual template like so: res.render(“stores”, { title: “Stores”, stores: stores });

You can access it now in stores.pug like so: each store in stores

We want to render individual files in a separate mixin however called a “store card” to display each store.

If you want to display the description but limit it to 25 words for example you can do javascript right in pug like so: p= store.description.split(‘ ‘).slice(0, 25).join(‘ ‘)

[](https://notes.anjagusev.com/static/eb1b368914053390229a0d66d0641875/e09fe/06055421-ED5E-4C07-B9DD-8B3C5C6DAD96.png)

## 14 - Creating an Editing Flow for Stores

We will be creating an edit button with the functionality to edit a store.

The edit button should like to something like a(href=“/store/123/edit”) where 123 is the store id. To achieve this we use:

a(href=`/store/${store.\_id}/edit`)

We want to add an SVG of a pencil for the edit icon. We have a helped class linking to an SVG file that we can use.

**TO TELL PUG WE ARE WRITING HTML USE !=**

Now we need to hook up a route for this to work. We are going to create a route for that specific url combination.

How do we get the store.\_id into the route, like a wildcard parameter? We can use this syntax: a colon and then the name of the parameter that is going to be passed in router.get(‘/stores/:id/edit’). When we visit this page we are going to have a variable available to us on our request object that tells us what the actual id of the url is.

Edit store needs to do 3 things:

1. Find the store given the ID
2. confirm they are the owner of the store
3. Render out the edit form so the user can update their store.

On the request object we will have an object called params which will give you any parameters that have come through the URL. Params are anytime your url contains something like :id

We can use the param.id to find the associated store in the db like so: const store = await Store.findOne({ \_id: req.params.id });

We can reuse the edit form view that we created earlier.

res.render(‘editStore’, {title:`Edit ${store.name}`, store})

Now we need to populate the edit form with the store information. We pass the storeForm mixin the existing store that we are editing in the editStore.pug view like so: +storeForm(store).

Now within \_storeForm.pug add the value of this store in the appropriate inputs like so:

If you try to do on array.includes on something that doesn’t exist it will throw an error. For tags, we need to check the checkbox if that store includes that tag. To do so:

const tags = store.tags || []

ul.tags

each choice in choices

.tag.tag\_\_choice

input(type=“checkbox” id=choice value=choice name=“tags” checked=(tags.includes(choice)))

-> we are checking whether the “tags” include the current choice in the loop. If so, it will return true. (To use javascript wrap in parenthesis like it is for checked=

Finally we have to handle the updating of the URL, so we don’t create a second store but actually update the existing store. On a branch new store, we want to post to /add but on an editing one we want to post to /add/id

form(action=`/add/${store.\_id || ''}` method=“POST” class=“card”)

Now we have to create a route. router.post(“/add/:id”, catchErrors(storeController.updateStore));

In storeController.js we add a method called updateStore that uses mongo findOneAndUpdate(query, data, options) which takes three parameters, one is the query, the second is the data and third it takes some options.

For this we first pass the \_id, then we tell mongo what to update it with, which is req.body which contains all the form fields that got sent over.

const store = Store.findOneAndUpdate({\_id:req.params.id}, req.body, { new=true, runValidators:true })

new:true returns the new store instead of the old one. runValidators:true makes sure the model required parameters are checked because it only runs them on initial creation otherwise.

To run it, you have put .exec() to run the query that we have. And then finally we have to await the data.

Next we flash a message:

req.flash( “success”, `Successfully updated <strong>${store.name}</strong>. <a href=“stores/${store.slug}”>View Store → </a>`);

And redirect to edit store page/.

Find and update the store, run the validators, tell them it successfully updated and then redirect them to the edit screen they were just on.

exports.updateStore = async (req, res) => {

//find and update the store

const store = await Store.findOneAndUpdate({ \_id: req.params.id }, req.body, {

new: true, //return the new store instead of the old one

runValidators: true

}).exec();

req.flash(

“success”,

`Successfully updated <strong>${store.name}</strong>. <a href=“stores/${

store.slug

}”>View Store → </a>`

);

res.redirect(`/stores/${store.\_id}/edit`);

//redirect them to the store and tell them it worked.

};

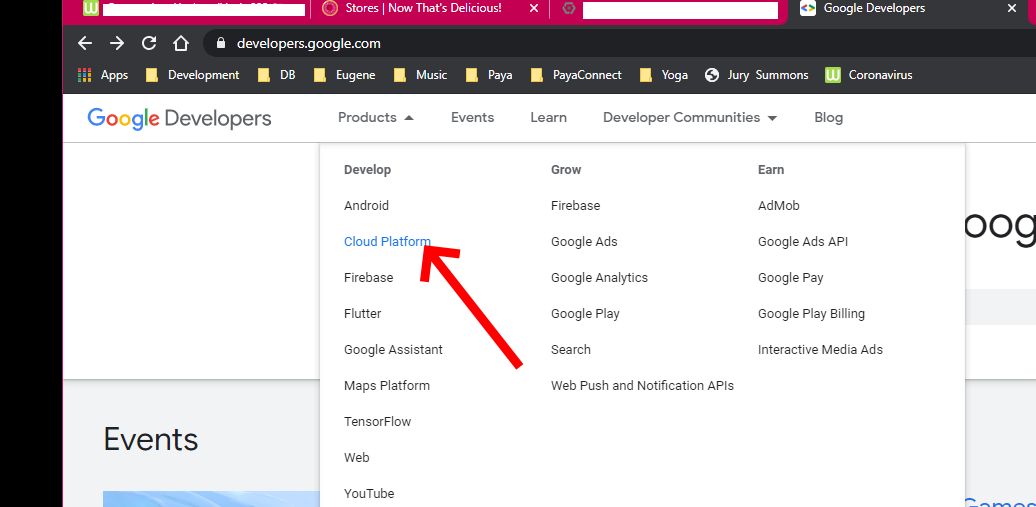
## 15 - Saving Lat & Lng for each store

Before you get started you need to have your Google profile completed.

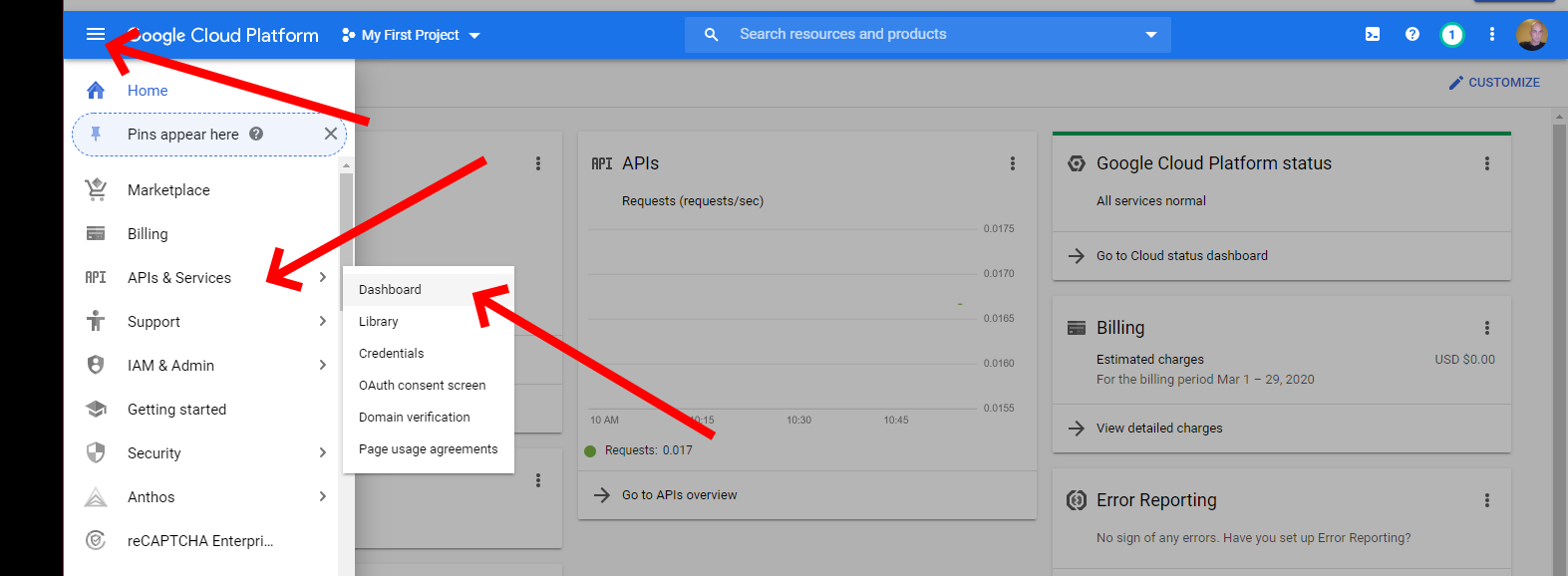
Head over to : <https://developers.google.com/>

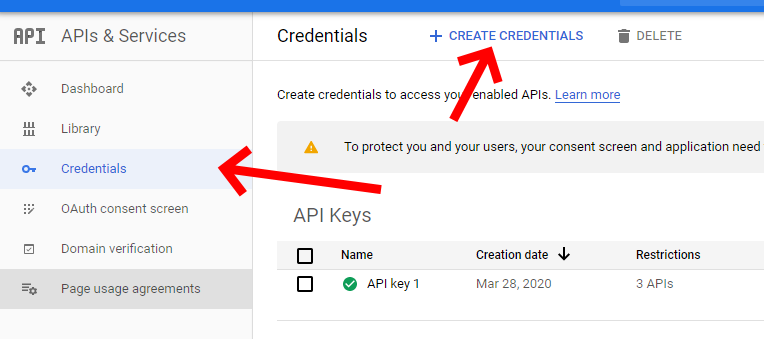
Register and provide billing

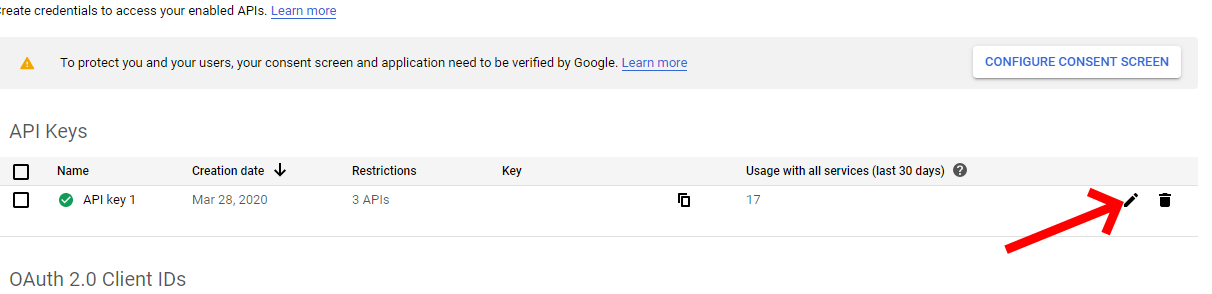
Then go to the Cloud Dashboard.

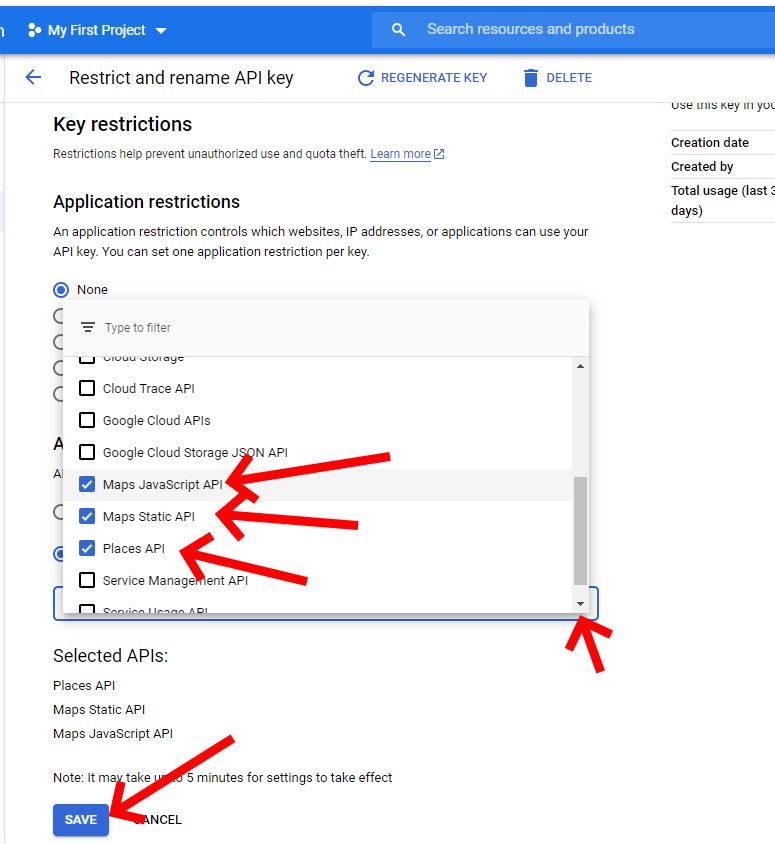


Next ensure you have the right api activated. You will need Static Maps and Javascript MAPs as well as places.









Now you are ready to query maps for information from google as a developer using the API. Developers are required to keep a payment method on file to access the API

Adding a date property to storeSchema:

created: {

type: Date,

default: Date.now

}

-> using a date object allows you to take advantage of custom queries in mongoldb.

Mongodb has stuff built in for location based things (co-ordinates).

There are many ways to store it, but we are going to do one point (a pin).

We added a few more properties for the store schema:

created: {

type: Date,

default: Date.now

},

location: {

type: {

type: String,

default: "Point"

},

coordinates: [

{

type: Number,

required: "You must supply coordinates"

}

],

address: {

type: String,

required: "You must supply an address!"

}

}

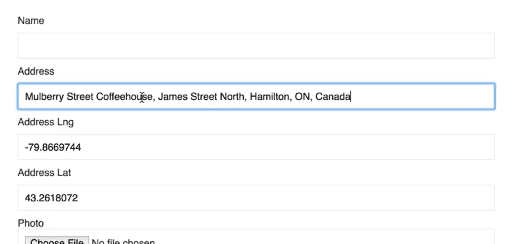
**IN PUG YOU ADD COMMENTS LIKE THIS** //- Address, lng and lat

input(type=“text” id=“address” name=“location[address]”) This will send the information as location.address (nested data) without doing heavy lifting

input(type=“text” id=“address” name=“location[address]” value=store.location.address) -> value will throw an error if there is no location (you can’t read a property of something that doesn’t exist).

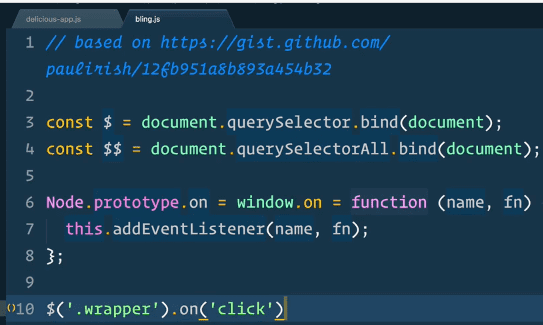
To fix that we do: (store.location && store.location.address ) which will only return the address is store.location exists, otherwise it will return false / nothing.

## 16 - Geocoding Data with Google Maps

[](https://notes.anjagusev.com/static/44963cbcc131c6fae827840ae1e877c1/42fef/D92AE35F-D997-4B0E-9B78-4AC3D8FD2487.png)address dropdown does reverse lookup for lat & lng

We are going to add the above functionality via client side javascript. In the /public/javascript/modules/ directory we will add it, and it is bundled into /public/javascript/modules/delicious-app.js

bling.js A wrapper which helps you grab things instead of writing out document.queryselector all the time.

[](https://notes.anjagusev.com/static/f3103a28cab868889dc6de05dfe9994e/1e3a7/CB89C424-982C-450C-900D-75729A5626FE.png)

We create new file in /modules directory named autocomplete.js. We can use ES6 here because although node doesn’t have ES6 yet, but we do have es6 and web pack here.

Then we import it in delicious-app.js import autocomplete from “./modules/autocomplete”;

Then pass the values from the inputs to the autocomplete method like so: autocomplete($(“#address"), $("#lat"), $("#lng")); in delicious-app.js

Needed to get an API key from google maps and replace it in variables.env

To get lng and lat to populate we need the following code:

const dropdown = new google.maps.places.Autocomplete(input);

dropdown.addListener(“place\_changed”, () => {

const place = dropdown.getPlace();

latInput.value = place.geometry.location.lat();

lngInput.value = place.geometry.location.lng();

console.log(place);

});

We can do input.on instead of input.addEventListener because of the bling library:

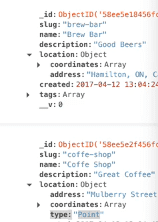
input.on(“keydown”, e => {

if (e.keyCode == 13) e.preventDefault();

});

## 17 - Quick Data Visualization Tip

When you update an address, in mongo it doesn’t save the address as a point.

[](https://notes.anjagusev.com/static/9b85b0249e1e5132d9f35c992c4060a5/0c596/B22B6C1C-229D-434C-A99D-98627A3641C3.png)

We need to add that, in updateStore add this line: req.body.location.type = ‘Point’;

## 18 - Uploading & Resizing Images with Middleware

Add uploading a file to our form. 3 stage process:

1. Modify form to accept image uploads
2. Add middleware to upload the file and resize it before we create the store

Before we create the store we need to add logic to upload and resize the file.

In \_storeForm.pug add enctype to form tag: enctype=“multipart/form-data”

Next we need to use Multer to handle all the fields that are being passed.

In storeController require muster. Next we need to tell Multer where the files will be stored when it’s uploaded and what types of files are allowed.

There are a many options for storage with multer. You can save it to disk, we are going to save it to memory (we don’t want to keep the original uploaded file. We want to resize it and save the resized version to file.

file.mimetype describes which type of file it is (jpg, png) .

In node, for callbacks, when calling next(null, true) the first value that gets passed is an error, and the second value is what needs to get passed along.

const multerOptions = {

storage: multer.memoryStorage(),

fileFilter(req, file, next) {

const isPhoto = file.mimetype.startsWith(“image/“);

if (isPhoto) {

next(null, true);

} else {

next({ message: “That file type is not allowed!” }, false);

}

}

};

Next create another method called upload. multer can handle multiple fields but we are looking for a single field -> exports.upload = multer(multerOptions).single(“photo”);

Next go to \_storeForm.pug and add the photo field.

label(for=“photo”) Photo

input(type=“file” name=“photo” id=“photo” accept=“image/gif, image/png, image/jpeg”)

if store.photo

img(src=`/uploads/${store.photo}` alt=store.name width=200)

The first method (upload) reads it into memory.

Next we import package called jimp to help resize the images. I

We need something to help us ensure the file names are unique so people cannot overwrite. We will use package uuid

Create another method under upload called resize and pass it req , res and next. The reason we pass next is because it’s middleware, we aren’t going to be doing any rendering or sending back to the client we are going to be saving the image , recording what the file name is and passing it to createStore.

Everytime we edit the form, we won’t be uploading an new file, and if that is the case multer won’t pass anything so we need to check for that: if(!req.file){ (multer puts the file on the file property) and call next(); and return();

Next we go to the router, and chain the middleware like so:

router.post(

“/add”,

storeController.upload,

catchErrors(storeController.resize),

catchErrors(storeController.createStore)

);

We added catch errors to resize because it’s async.

Next we go back to storeController.resize and grab the file time from req.file.mimetype.split(‘/‘)[1]; and create a unique name for the file like so:

req.body.photo = `${uuid.v4()}.${extension}`;

Next we resize. For jimp.read(), you either pass it the file path or a buffer (which is in memory).

exports.resize = async (req, res, next) => {

//check if there is no new file to resize

if (!req.file) {

next(); //skip to the next middleware

return; // stop the function from running any further

}

const extension = req.file.mimetype.split(“/“)[1];

req.body.photo = `${uuid.v4()}.${extension}`;

//now we resize

const photo = await jimp.read(req.file.buffer);

await photo.resize(800, jimp.AUTO);

await photo.write(`./public/uploads/${req.body.photo}`);

//once we have written the photo to our file system, keep going

next();

};

Next, add the photo to the schema at Store.js with photo: String

Now we need to make the photo editable. We do that by going to our routes.js and doing the same thing we did to the add route like so:

router.post(

“/add/:id”,

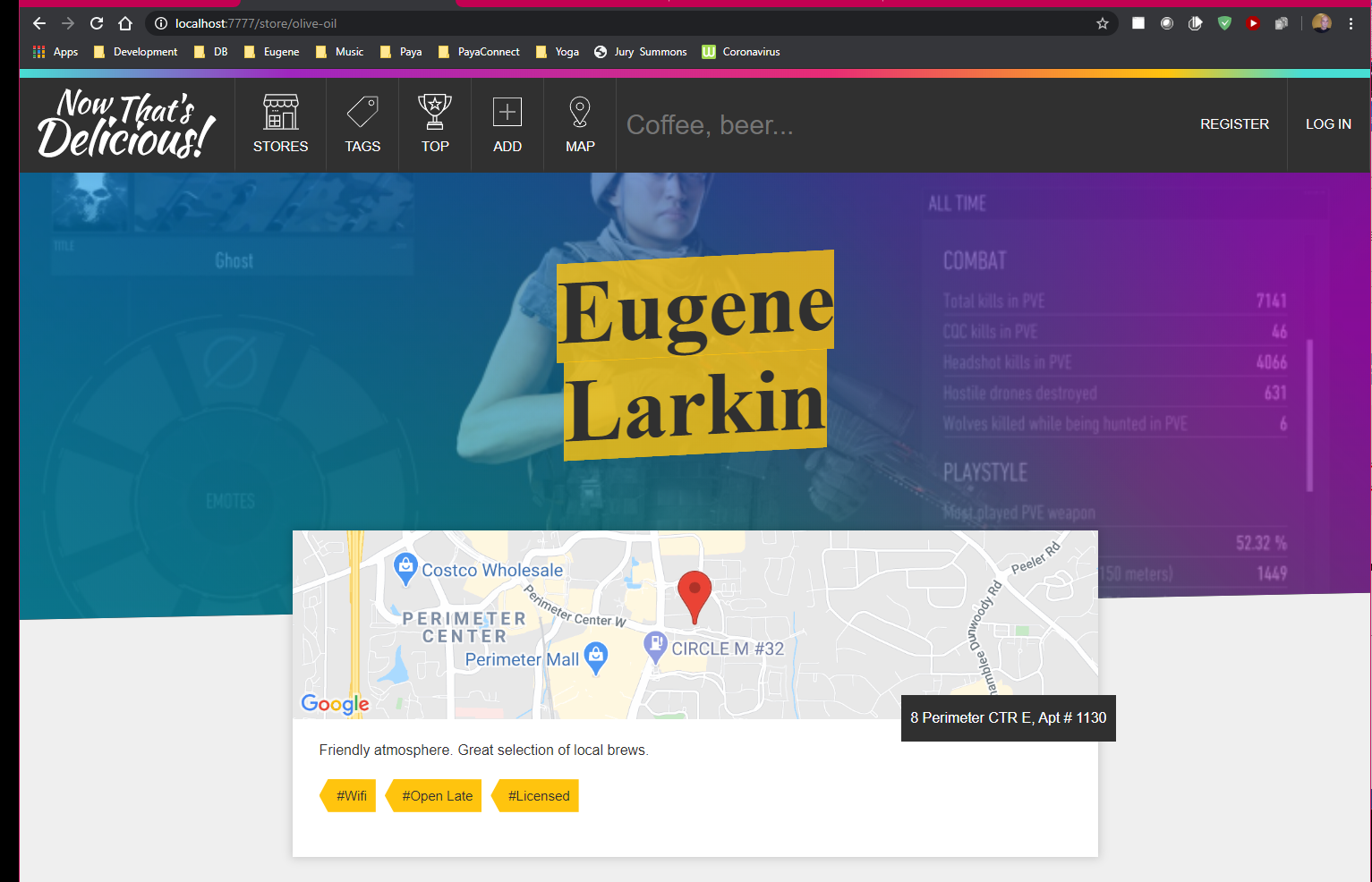
storeController.upload,

catchErrors(storeController.resize),

catchErrors(storeController.updateStore)

);

You can use any image. I used one from one of my game captures.



);