Contents

[**1 - 3 Learn Node** 5](#_Toc40711875)

[**4 - Routing / Core Concept** 5](#_Toc40711876)

[**5 - Templating / Core Concept** 7](#_Toc40711877)

[**6 - Templating Helpers / Core Concept** 8](#_Toc40711878)

[**7 - Controllers and MVC Pattern / Core Concept** 9](#_Toc40711879)

[**8 - Middleware and Error Handling / Core Concept** 13](#_Toc40711880)

[9 - Creating our Store Model 15](#_Toc40711881)

[10 - Saving Stores and Using Mixins 19](#_Toc40711882)

[11 - Using Async Await 23](#_Toc40711883)

[12 - Flash Messages 24](#_Toc40711884)

[13 - Querying our Database for Stores 27](#_Toc40711885)

[14 - Creating an Editing Flow for Stores 28](#_Toc40711886)

[15 - Saving Lat & Lng for each store 30](#_Toc40711887)

[16 - Geocoding Data with Google Maps 33](#_Toc40711888)

[17 - Quick Data Visualization Tip 35](#_Toc40711889)

[18 - Uploading & Resizing Images with Middleware 36](#_Toc40711890)

[19 - Routing and Templating Single Store 39](#_Toc40711891)

[20 - Using Pre-save hooks to make Unique Slugs 41](#_Toc40711892)

[21 - Custom MongoDB Aggregations 42](#_Toc40711893)

[Index of Expression Operators 44](#_Toc40711894)

[22 - Multiple Query Promise with Async Await 49](#_Toc40711895)

[23 - Creating User Accounts 51](#_Toc40711896)

[24 - Saving Registered Users to the Database 55](#_Toc40711897)

[25 - Virtual Fields, Login-Logout Middleware and Protecting Routes 57](#_Toc40711898)

[26 - Creating a User Account Edit Screen 60](#_Toc40711899)

[27 - Password Reset Flow 61](#_Toc40711900)

[28 - Sending email with Node.js 64](#_Toc40711901)

[29 - Locking down App with User Permissions 68](#_Toc40711902)

[30 - Loading Sample Data 69](#_Toc40711903)

[31 - JSON endpoints and creating mongodb indexes 69](#_Toc40711904)

[32 - Creating an Ajax Search Interface 71](#_Toc40711905)

[33 - Creating a Geospatial Ajax Endpoint 75](#_Toc40711906)

[34 - Plotting Stores on Custom Google Map 76](#_Toc40711907)

[35 - Pushing User Data to our API 79](#_Toc40711908)

[36 - Displaying our Hearted Stores 82](#_Toc40711909)

[37 - Adding Reviews Data Model 82](#_Toc40711910)

[38 - Advanced Relationship Population - Displaying our Reviews 84](#_Toc40711911)

[39 - Advanced Aggregation 88](#_Toc40711912)

[40 - Implementing Pagination 95](#_Toc40711913)

[41- Deployment Setup 97](#_Toc40711914)

[42 - Deploying to Now 97](#_Toc40711915)

[43 - Deploying to Heroku 98](#_Toc40711916)

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)

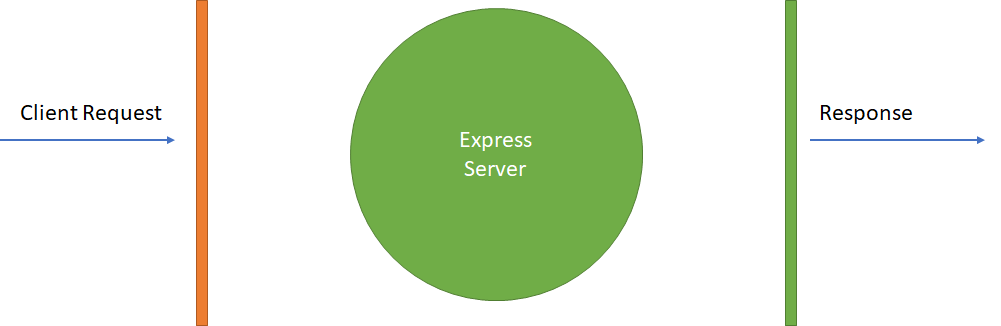
Everything the user will see. Basically, the pages that we’re going to send to the client.

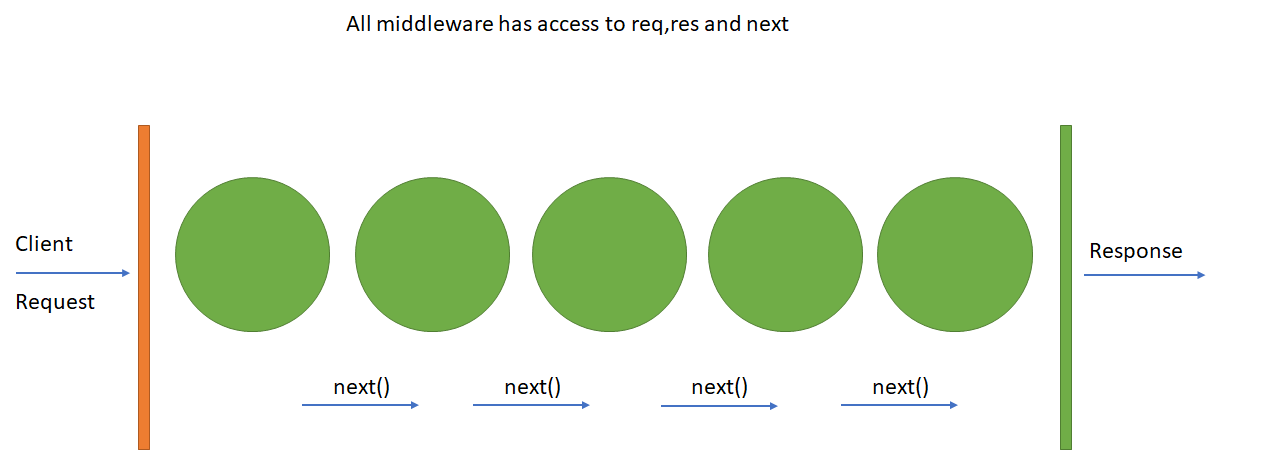
C = Controller (ex: controls flow of data )

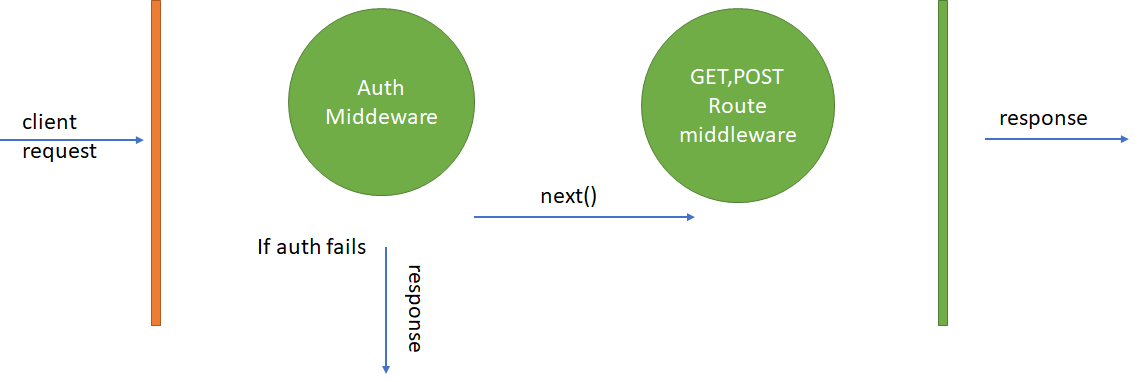
The logic of our site, and the glue between models and views. Call the models to get the data, then put that data on the views to be sent to the users.

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,'-')

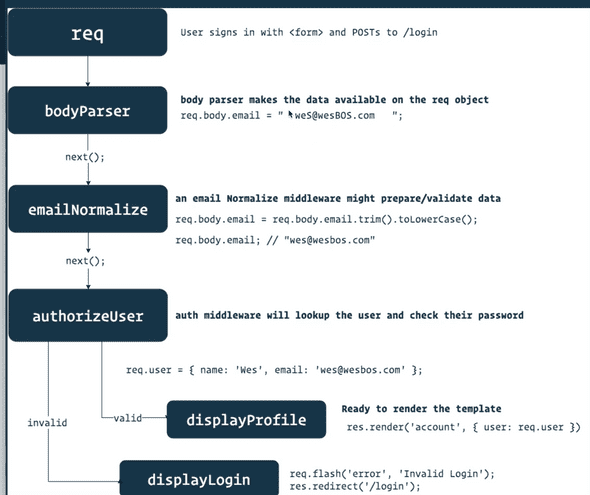
;

}

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 },

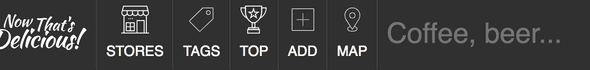
toObject: { virtuals: true },

}

)

## 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”)

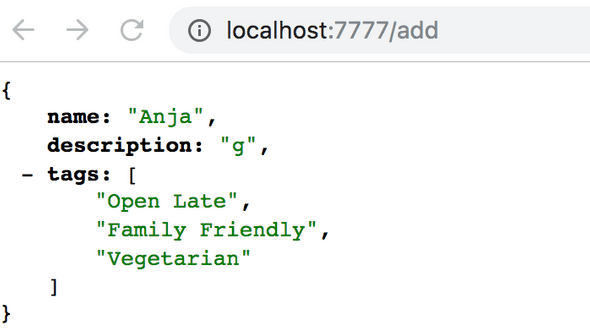
However when we submit it we get a 404 error because we need to add a POST route: router.post(“/add”, storeController.createStore);

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

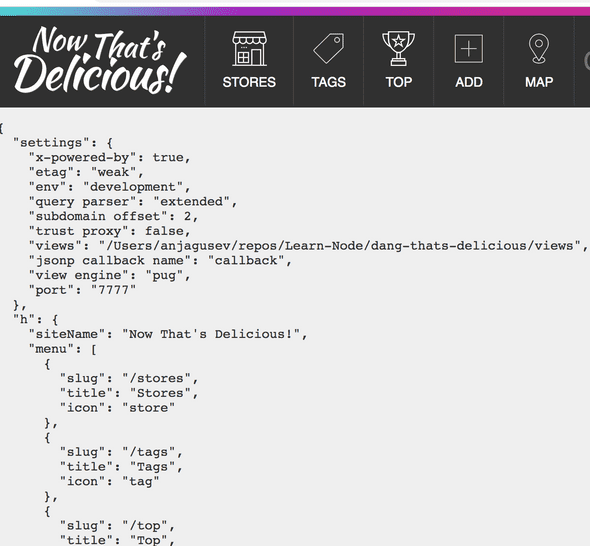
each message in flashes[category]

.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 })

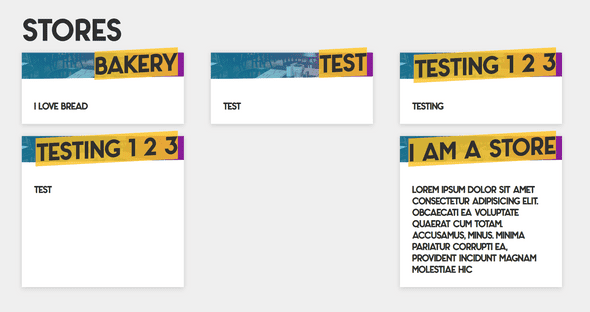
}

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.

};

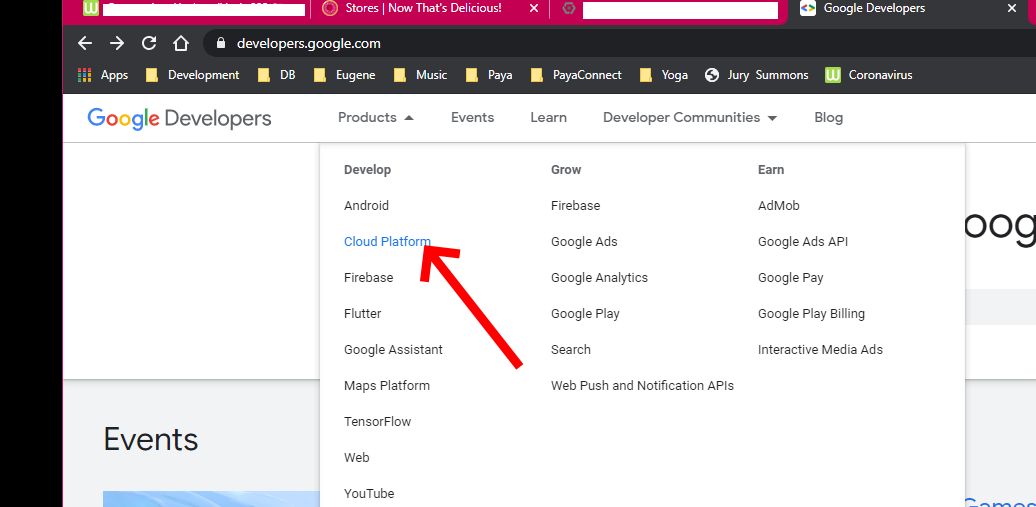
## 15 - Saving Lat & Lng for each store

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

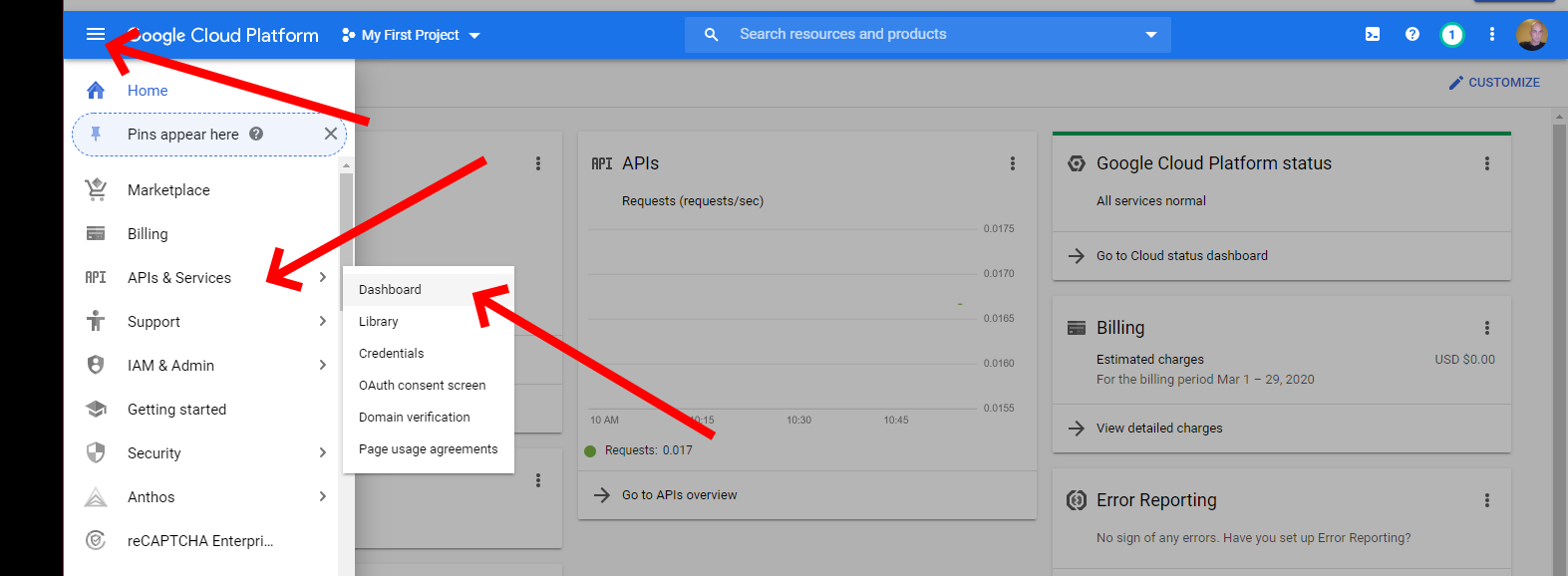
Head over to : <https://developers.google.com/>

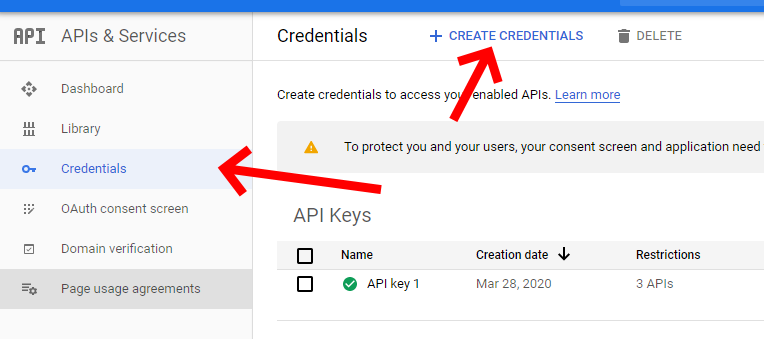
Register and provide billing

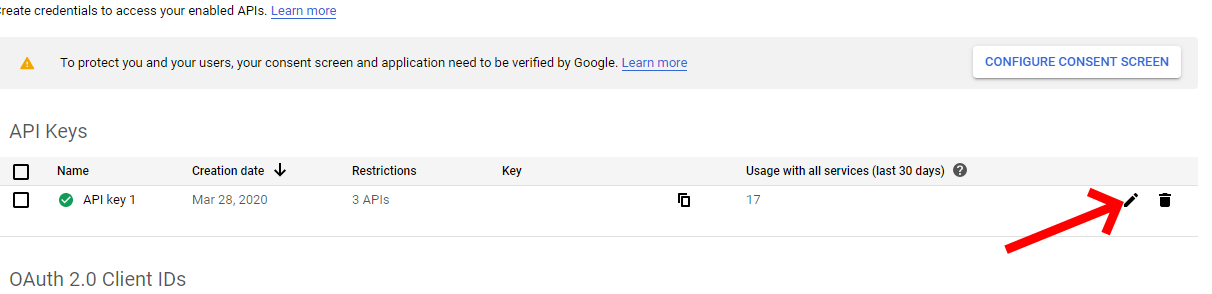
Then go to the Cloud Dashboard.

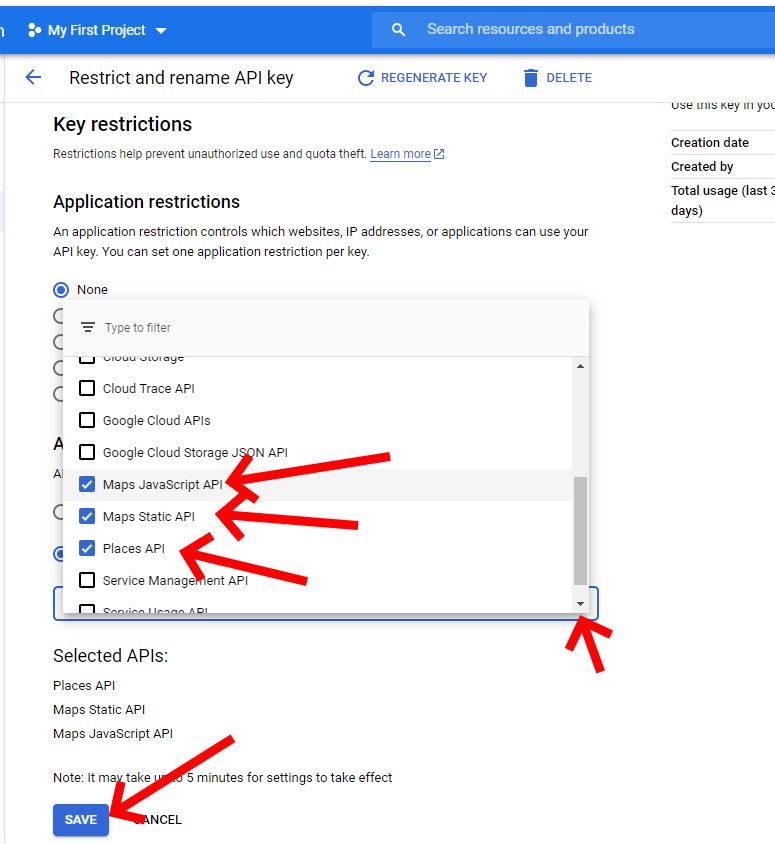


Next ensure you have the right api activated. You will need Static Maps and Javascript MAPs as well as places.









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!"

}

}

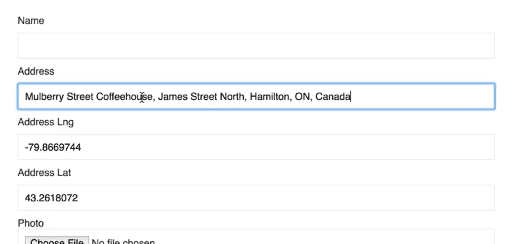
**IN PUG YOU ADD COMMENTS LIKE THIS** //- Address, lng and lat

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).

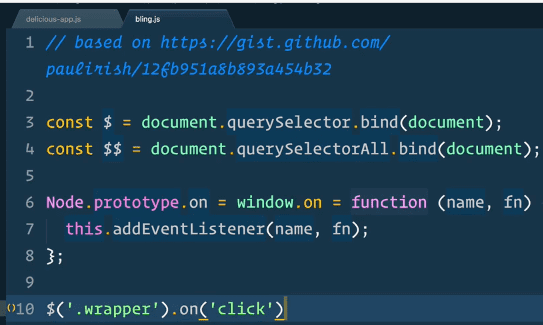
To fix that we do: (store.location && store.location.address ) which will only return the address is store.location exists, otherwise it will return false / nothing.

## 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

In autocomplete.js

  // log the input parms as a checker only

  console.log("Called Autocomplete", input, lngInput, latInput);

You will need to create a file

Autocomplete.js in the modules directory

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:

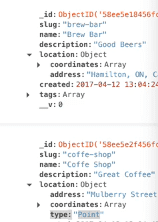
input.on(“keydown”, e => {

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”,

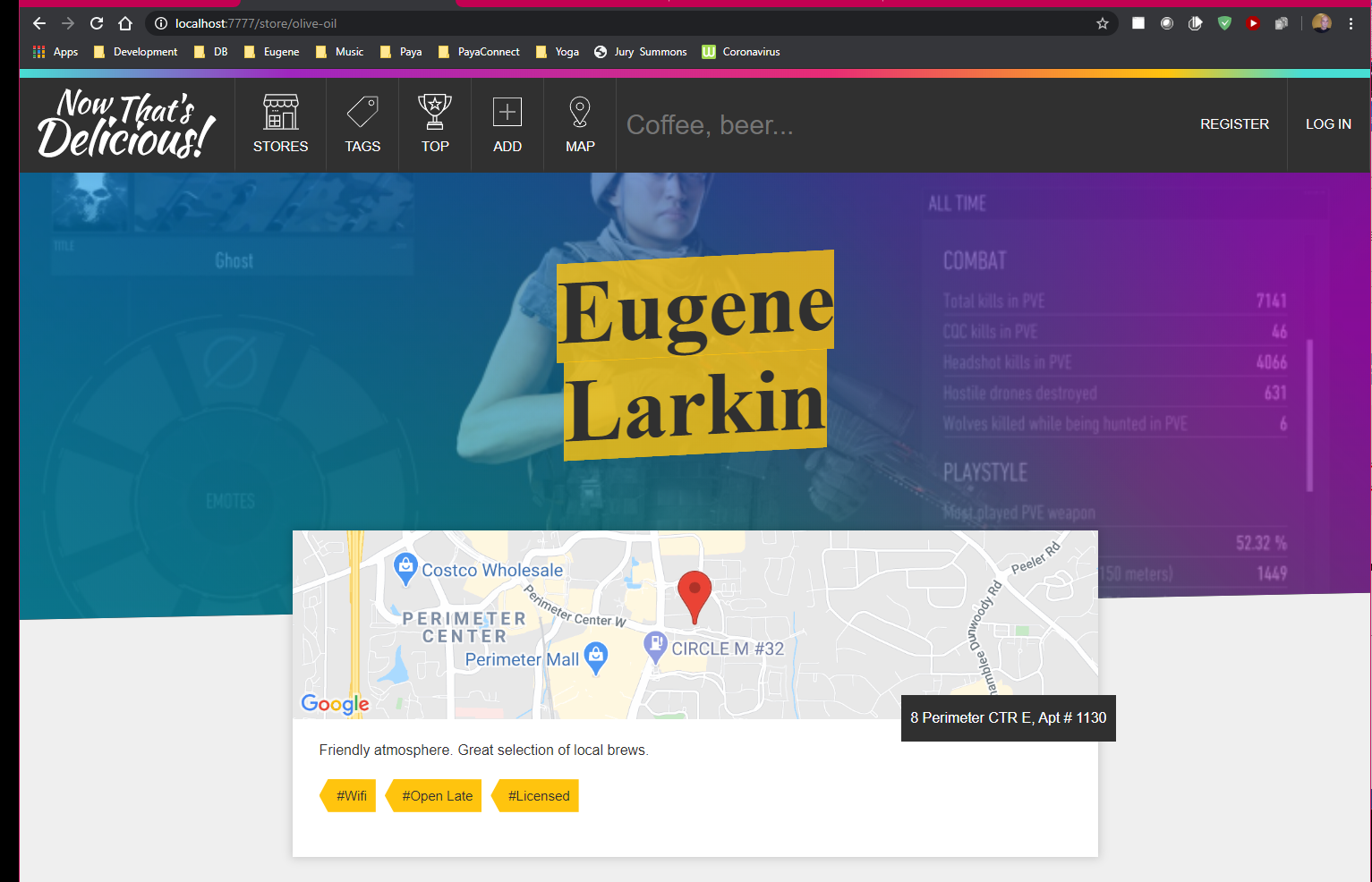
storeController.upload,

catchErrors(storeController.resize),

catchErrors(storeController.updateStore)

);

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



);

## 19 - Routing and Templating Single Store

My Visual Studio Code updated. I got an error when I try to start it. The error is about a missing DLL. Here is the solution steps.

**Method 1: Try System File Checker**

Please follow the steps mentioned in the link, How to use the System File Checker tool to troubleshoot missing or corrupted system files on Windows Vista or on Windows 7: <http://support.microsoft.com/kb/929833>

**Method 2: System restore**

You may try system restore and check if it helps if the issue happened after any changes.

System Restore can return your PC's system files and programs to a time when everything was working fine, potentially preventing hours of troubleshooting headaches. It will not affect your documents, pictures, or other data.

Please refer to the link:

<http://windows.microsoft.com/en-US/windows-vista/What-is-System-Restore>

**Method 3:** You may have to download and install the codec.

It might also be that the codec’s that are installed on the system are corrupted, I would suggest you to uninstall the codec’s that are present and then reinstall it and check whether the issue persists. Refer the article mentioned below.

**Codec’s: frequently asked questions**

<http://windows.microsoft.com/en-US/windows-vista/Codecs-frequently-asked-questions>

Creating a page to view an individual store.

First we need to find the store that has that slug, and then next we need to pull in the name of the photo, map, description, list of tags and showing login form eventually and the reviews.

To do this we need to: -create a new route: router.get(“/store/:slug”, catchErrors(storeController.getStoreBySlug));

Add a controller method that queries the database to find one store matching the slug.

We need to ensure that if there is no store, we do not proceed, because someone could type in a wrong url.

if(!store) return next();

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

//1. Query database for a store based on slug

const store = await Store.findOne({ slug: req.params.slug });

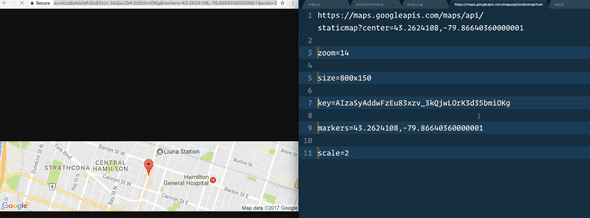
if (!store) return next();

// res.json(store);

res.render(“store”, { title: store.name, store });

};

Next we are working on the static map. The google static map takes parameters such as Lat & Lng.

[](https://notes.anjagusev.com/static/3c51f3b5b7b616ed263e5bcf390fcdf9/95223/7562090A-1885-456B-9401-3A94B27091F2.png)

We are going to use a helper method to return the map, which takes in an array as the parameters (lng & lat)

You will find this in the helpers.js file.

// Making a static map is really long - this is a handy helper function to make one

exports.staticMap = ([lng, lat]) =>

`https://maps.googleapis.com/maps/api/staticmap?center=${lat},${lng}&zoom=14&size=800x150&key=${

process.env.MAP\_KEY

}&markers=${lat},${lng}&scale=2`

Which is called like this img.single\_\_map(src=h.staticMap(store.location.coordinates))

Now we build out the rest of store.pug

## 20 - Using Pre-save hooks to make Unique Slugs

If you name two stores the same, the pre-save slug will override one of them.

We need to fix the pre-save slug to ensure no overwriting happens.

We need to modify the pre save function to check for that. First change it to async, and then we use a regex to look for store name with - # for example “anja-store”, “anja-store-1”, “anja-store-2”.

Next we need to pass the regex to a query to look up stores. Within the Store schema, we will do that using this.constructor.find({slugRegEx})

Query the db with the number of stores with that store name, and then generate the slug using storesWithSlug.length + 1

const storesWithSlug = await this.constructor.find({ slug: slugRegEx })

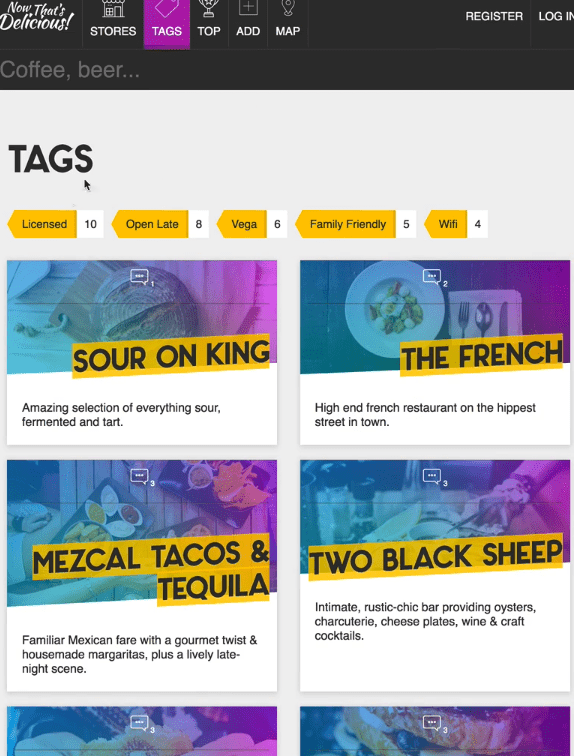
if (storesWithSlug.length) {

this.slug = `${this.slug}-${storesWithSlug.length + 1}`

}

## 21 - Custom MongoDB Aggregations

Here we will be building the tags page

[](https://notes.anjagusev.com/static/3e3ba7f0af7fb62225fa4c4da51515ee/ed75b/A5349E65-077C-46AB-996D-9FA3EEED323C.png)

We need a list of all possible tags, and we need to know how many of each store are in those tags. When you go to a tag page, it should filter for all of the stores that have that specific tag.

Create two new routes:

router.get(“/tags”, catchErrors(storeController.getStoresByTag));

router.get(“/tags/:tag”, catchErrors(storeController.getStoresByTag));

You can add a regex / wildcard to say the param after is optional like so: router.get(“/tags/:tag\*?”,

Next, in storeController, we get a list of all of the stores. Instead of Store.find() we can create our own static methods.

Store.getTagsList()

Now we are going to take out store schema and add a method onto it.

storeSchema.statics.getTagsList which is equal to a function, that is bound to our model (don’t use arrow function because we need this)

We are going to unwind our document. Inside of getTagList we will return this.aggregate with an array of options.

Each pipeline operator gets it’s own object.

{$unwind: $tags} the $ of tags lets mongo know this is a field on my document (which I want to unwind).

For unwinding, we get an instance of a store for family friend, and an instance of a store for WiFi. It’s duplicating the content but we don’t care because each store needs one tag per say. We need to group them in our next operator.

**Points to remember:**

* If the value of a field is not an array, db.collection.aggregate() generates an error.
* If the specified path for a field does not exist in an input document, the pipeline ignores the input document and displaying no output.
* If the array is empty in an input document, the pipeline ignores the input document and displaying no output.

Unwind can be the opposite of push

Say we have previously $group-ed our data, and $push-ed to generate an array. we can use $unwind to restore the values with the count attached.

Say we have a list of customer interactions. For each interaction we want to add an 'engagement' field which lists out the number of times the customer appears in the list.

We might:

$group by email and $count interactions

$push the $$ROOT into a 'customer' field

$unwind the customer field

$project to tidy up the data

We now have the original time series data, but with the addition of an engagement field.

We want to group by tags and count them. We also want to sort from most to least.

storeSchema.statics.getTagsList = function() {

return this.aggregate([

{ $unwind: "$tags" },

{ $group: { \_id: "$tags", count: { $sum: 1 } } },

{ $sort: { count: -1 } },

])

}

Next create the tag.pug view. Loop over the tags, and highlight the active tag.

We want to highlight the selected tag to be active. To do that, we need to pass the data about what page we are on to our template.

In the getStoresByTag method, we need to grab the tag from the parameter req.param.tag and pass it to the view in the render method.

Within the view, we want to assign the tag with active if we are on that tags page ->

a.tag\_\_link(href=`/tags/${t.\_id}` class=(t.\_id === tag ? tag\_\_link—active : ‘’))

Change title to show the tag name if on specific page or just tag if on generic tag page.

Search for mongodb aggregators to find a list of the aggregators available.

<https://docs.mongodb.com/manual/meta/aggregation-quick-reference/>

Index of Expression Operators

|  |  |  |  |
| --- | --- | --- | --- |
| * [$abs](https://docs.mongodb.com/manual/reference/operator/aggregation/abs/#exp._S_abs) * [$acos](https://docs.mongodb.com/manual/reference/operator/aggregation/acos/#exp._S_acos) * [$acosh](https://docs.mongodb.com/manual/reference/operator/aggregation/acosh/#exp._S_acosh) * [$add](https://docs.mongodb.com/manual/reference/operator/aggregation/add/#exp._S_add) * [$addToSet](https://docs.mongodb.com/manual/reference/operator/aggregation/addToSet/#grp._S_addToSet) * [$allElementsTrue](https://docs.mongodb.com/manual/reference/operator/aggregation/allElementsTrue/#exp._S_allElementsTrue) * [$and](https://docs.mongodb.com/manual/reference/operator/aggregation/and/#exp._S_and) * [$anyElementTrue](https://docs.mongodb.com/manual/reference/operator/aggregation/anyElementTrue/#exp._S_anyElementTrue) * [$arrayElemAt](https://docs.mongodb.com/manual/reference/operator/aggregation/arrayElemAt/#exp._S_arrayElemAt) * [$arrayToObject](https://docs.mongodb.com/manual/reference/operator/aggregation/arrayToObject/#exp._S_arrayToObject) * [$asin](https://docs.mongodb.com/manual/reference/operator/aggregation/asin/#exp._S_asin) * [$asinh](https://docs.mongodb.com/manual/reference/operator/aggregation/asinh/#exp._S_asinh) * [$atan](https://docs.mongodb.com/manual/reference/operator/aggregation/atan/#exp._S_atan) * [$atan2](https://docs.mongodb.com/manual/reference/operator/aggregation/atan2/#exp._S_atan2) * [$atanh](https://docs.mongodb.com/manual/reference/operator/aggregation/atanh/#exp._S_atanh) * [$avg](https://docs.mongodb.com/manual/reference/operator/aggregation/avg/#grp._S_avg) * [$ceil](https://docs.mongodb.com/manual/reference/operator/aggregation/ceil/#exp._S_ceil) * [$cmp](https://docs.mongodb.com/manual/reference/operator/aggregation/cmp/#exp._S_cmp) * [$concat](https://docs.mongodb.com/manual/reference/operator/aggregation/concat/#exp._S_concat) * [$concatArrays](https://docs.mongodb.com/manual/reference/operator/aggregation/concatArrays/#exp._S_concatArrays) * [$cond](https://docs.mongodb.com/manual/reference/operator/aggregation/cond/#exp._S_cond) * [$convert](https://docs.mongodb.com/manual/reference/operator/aggregation/convert/#exp._S_convert) * [$cos](https://docs.mongodb.com/manual/reference/operator/aggregation/cos/#exp._S_cos) * [$dateFromParts](https://docs.mongodb.com/manual/reference/operator/aggregation/dateFromParts/#exp._S_dateFromParts) * [$dateFromString](https://docs.mongodb.com/manual/reference/operator/aggregation/dateFromString/#exp._S_dateFromString) * [$dateToParts](https://docs.mongodb.com/manual/reference/operator/aggregation/dateToParts/#exp._S_dateToParts) * [$dateToString](https://docs.mongodb.com/manual/reference/operator/aggregation/dateToString/#exp._S_dateToString) * [$dayOfMonth](https://docs.mongodb.com/manual/reference/operator/aggregation/dayOfMonth/#exp._S_dayOfMonth) * [$dayOfWeek](https://docs.mongodb.com/manual/reference/operator/aggregation/dayOfWeek/#exp._S_dayOfWeek) * [$dayOfYear](https://docs.mongodb.com/manual/reference/operator/aggregation/dayOfYear/#exp._S_dayOfYear) * [$degreesToRadians](https://docs.mongodb.com/manual/reference/operator/aggregation/degreesToRadians/#exp._S_degreesToRadians) | * [$divide](https://docs.mongodb.com/manual/reference/operator/aggregation/divide/#exp._S_divide) * [$eq](https://docs.mongodb.com/manual/reference/operator/aggregation/eq/#exp._S_eq) * [$exp](https://docs.mongodb.com/manual/reference/operator/aggregation/exp/#exp._S_exp) * [$filter](https://docs.mongodb.com/manual/reference/operator/aggregation/filter/#exp._S_filter) * [$first](https://docs.mongodb.com/manual/reference/operator/aggregation/first/#grp._S_first) * [$floor](https://docs.mongodb.com/manual/reference/operator/aggregation/floor/#exp._S_floor) * [$gt](https://docs.mongodb.com/manual/reference/operator/aggregation/gt/#exp._S_gt) * [$gte](https://docs.mongodb.com/manual/reference/operator/aggregation/gte/#exp._S_gte) * [$hour](https://docs.mongodb.com/manual/reference/operator/aggregation/hour/#exp._S_hour) * [$ifNull](https://docs.mongodb.com/manual/reference/operator/aggregation/ifNull/#exp._S_ifNull) * [$in](https://docs.mongodb.com/manual/reference/operator/aggregation/in/#exp._S_in) * [$indexOfArray](https://docs.mongodb.com/manual/reference/operator/aggregation/indexOfArray/#exp._S_indexOfArray) * [$indexOfBytes](https://docs.mongodb.com/manual/reference/operator/aggregation/indexOfBytes/#exp._S_indexOfBytes) * [$indexOfCP](https://docs.mongodb.com/manual/reference/operator/aggregation/indexOfCP/#exp._S_indexOfCP) * [$isArray](https://docs.mongodb.com/manual/reference/operator/aggregation/isArray/#exp._S_isArray) * [$isoDayOfWeek](https://docs.mongodb.com/manual/reference/operator/aggregation/isoDayOfWeek/#exp._S_isoDayOfWeek) * [$isoWeek](https://docs.mongodb.com/manual/reference/operator/aggregation/isoWeek/#exp._S_isoWeek) * [$isoWeekYear](https://docs.mongodb.com/manual/reference/operator/aggregation/isoWeekYear/#exp._S_isoWeekYear) * [$last](https://docs.mongodb.com/manual/reference/operator/aggregation/last/#grp._S_last) * [$let](https://docs.mongodb.com/manual/reference/operator/aggregation/let/#exp._S_let) * [$literal](https://docs.mongodb.com/manual/reference/operator/aggregation/literal/#exp._S_literal) * [$ln](https://docs.mongodb.com/manual/reference/operator/aggregation/ln/#exp._S_ln) * [$log](https://docs.mongodb.com/manual/reference/operator/aggregation/log/#exp._S_log) * [$log10](https://docs.mongodb.com/manual/reference/operator/aggregation/log10/#exp._S_log10) * [$lt](https://docs.mongodb.com/manual/reference/operator/aggregation/lt/#exp._S_lt) * [$lte](https://docs.mongodb.com/manual/reference/operator/aggregation/lte/#exp._S_lte) * [$ltrim](https://docs.mongodb.com/manual/reference/operator/aggregation/ltrim/#exp._S_ltrim) * [$map](https://docs.mongodb.com/manual/reference/operator/aggregation/map/#exp._S_map) * [$max](https://docs.mongodb.com/manual/reference/operator/aggregation/max/#grp._S_max) * [$mergeObjects](https://docs.mongodb.com/manual/reference/operator/aggregation/mergeObjects/#exp._S_mergeObjects) * [$meta](https://docs.mongodb.com/manual/reference/operator/aggregation/meta/#exp._S_meta) | * [$millisecond](https://docs.mongodb.com/manual/reference/operator/aggregation/millisecond/#exp._S_millisecond) * [$min](https://docs.mongodb.com/manual/reference/operator/aggregation/min/#grp._S_min) * [$minute](https://docs.mongodb.com/manual/reference/operator/aggregation/minute/#exp._S_minute) * [$mod](https://docs.mongodb.com/manual/reference/operator/aggregation/mod/#exp._S_mod) * [$month](https://docs.mongodb.com/manual/reference/operator/aggregation/month/#exp._S_month) * [$multiply](https://docs.mongodb.com/manual/reference/operator/aggregation/multiply/#exp._S_multiply) * [$ne](https://docs.mongodb.com/manual/reference/operator/aggregation/ne/#exp._S_ne) * [$not](https://docs.mongodb.com/manual/reference/operator/aggregation/not/#exp._S_not) * [$objectToArray](https://docs.mongodb.com/manual/reference/operator/aggregation/objectToArray/#exp._S_objectToArray) * [$or](https://docs.mongodb.com/manual/reference/operator/aggregation/or/#exp._S_or) * [$pow](https://docs.mongodb.com/manual/reference/operator/aggregation/pow/#exp._S_pow) * [$push](https://docs.mongodb.com/manual/reference/operator/aggregation/push/#grp._S_push) * [$radiansToDegrees](https://docs.mongodb.com/manual/reference/operator/aggregation/radiansToDegrees/#exp._S_radiansToDegrees) * [$range](https://docs.mongodb.com/manual/reference/operator/aggregation/range/#exp._S_range) * [$reduce](https://docs.mongodb.com/manual/reference/operator/aggregation/reduce/#exp._S_reduce) * [$regexFind](https://docs.mongodb.com/manual/reference/operator/aggregation/regexFind/#exp._S_regexFind) * [$regexFindAll](https://docs.mongodb.com/manual/reference/operator/aggregation/regexFindAll/#exp._S_regexFindAll) * [$regexMatch](https://docs.mongodb.com/manual/reference/operator/aggregation/regexMatch/#exp._S_regexMatch) * [$reverseArray](https://docs.mongodb.com/manual/reference/operator/aggregation/reverseArray/#exp._S_reverseArray) * [$round](https://docs.mongodb.com/manual/reference/operator/aggregation/round/#exp._S_round) * [$rtrim](https://docs.mongodb.com/manual/reference/operator/aggregation/rtrim/#exp._S_rtrim) * [$second](https://docs.mongodb.com/manual/reference/operator/aggregation/second/#exp._S_second) * [$setDifference](https://docs.mongodb.com/manual/reference/operator/aggregation/setDifference/#exp._S_setDifference) * [$setEquals](https://docs.mongodb.com/manual/reference/operator/aggregation/setEquals/#exp._S_setEquals) * [$setIntersection](https://docs.mongodb.com/manual/reference/operator/aggregation/setIntersection/#exp._S_setIntersection) * [$setIsSubset](https://docs.mongodb.com/manual/reference/operator/aggregation/setIsSubset/#exp._S_setIsSubset) * [$setUnion](https://docs.mongodb.com/manual/reference/operator/aggregation/setUnion/#exp._S_setUnion) * [$sin](https://docs.mongodb.com/manual/reference/operator/aggregation/sin/#exp._S_sin) * [$size](https://docs.mongodb.com/manual/reference/operator/aggregation/size/#exp._S_size) * [$slice](https://docs.mongodb.com/manual/reference/operator/aggregation/slice/#exp._S_slice) | * [$split](https://docs.mongodb.com/manual/reference/operator/aggregation/split/#exp._S_split) * [$sqrt](https://docs.mongodb.com/manual/reference/operator/aggregation/sqrt/#exp._S_sqrt) * [$stdDevPop](https://docs.mongodb.com/manual/reference/operator/aggregation/stdDevPop/#grp._S_stdDevPop) * [$stdDevSamp](https://docs.mongodb.com/manual/reference/operator/aggregation/stdDevSamp/#grp._S_stdDevSamp) * [$strLenBytes](https://docs.mongodb.com/manual/reference/operator/aggregation/strLenBytes/#exp._S_strLenBytes) * [$strLenCP](https://docs.mongodb.com/manual/reference/operator/aggregation/strLenCP/#exp._S_strLenCP) * [$strcasecmp](https://docs.mongodb.com/manual/reference/operator/aggregation/strcasecmp/#exp._S_strcasecmp) * [$substr](https://docs.mongodb.com/manual/reference/operator/aggregation/substr/#exp._S_substr) * [$substrBytes](https://docs.mongodb.com/manual/reference/operator/aggregation/substrBytes/#exp._S_substrBytes) * [$substrCP](https://docs.mongodb.com/manual/reference/operator/aggregation/substrCP/#exp._S_substrCP) * [$subtract](https://docs.mongodb.com/manual/reference/operator/aggregation/subtract/#exp._S_subtract) * [$sum](https://docs.mongodb.com/manual/reference/operator/aggregation/sum/#grp._S_sum) * [$switch](https://docs.mongodb.com/manual/reference/operator/aggregation/switch/#exp._S_switch) * [$tan](https://docs.mongodb.com/manual/reference/operator/aggregation/tan/#exp._S_tan) * [$toBool](https://docs.mongodb.com/manual/reference/operator/aggregation/toBool/#exp._S_toBool) * [$toDate](https://docs.mongodb.com/manual/reference/operator/aggregation/toDate/#exp._S_toDate) * [$toDecimal](https://docs.mongodb.com/manual/reference/operator/aggregation/toDecimal/#exp._S_toDecimal) * [$toDouble](https://docs.mongodb.com/manual/reference/operator/aggregation/toDouble/#exp._S_toDouble) * [$toInt](https://docs.mongodb.com/manual/reference/operator/aggregation/toInt/#exp._S_toInt) * [$toLong](https://docs.mongodb.com/manual/reference/operator/aggregation/toLong/#exp._S_toLong) * [$toLower](https://docs.mongodb.com/manual/reference/operator/aggregation/toLower/#exp._S_toLower) * [$toObjectId](https://docs.mongodb.com/manual/reference/operator/aggregation/toObjectId/#exp._S_toObjectId) * [$toString](https://docs.mongodb.com/manual/reference/operator/aggregation/toString/#exp._S_toString) * [$toUpper](https://docs.mongodb.com/manual/reference/operator/aggregation/toUpper/#exp._S_toUpper) * [$trim](https://docs.mongodb.com/manual/reference/operator/aggregation/trim/#exp._S_trim) * [$trunc](https://docs.mongodb.com/manual/reference/operator/aggregation/trunc/#exp._S_trunc) * [$type](https://docs.mongodb.com/manual/reference/operator/aggregation/type/#exp._S_type) * [$week](https://docs.mongodb.com/manual/reference/operator/aggregation/week/#exp._S_week) * [$year](https://docs.mongodb.com/manual/reference/operator/aggregation/year/#exp._S_year) * [$zip](https://docs.mongodb.com/manual/reference/operator/aggregation/zip/#exp._S_zip) |

Some notes on autocomplete as I test the code.

Working with Paul Barrick

Stop website

Change extract-text version

Delete nodemodules

Copy and paste packages.json from below over yours.

{

  "name": "dang-thats-delicious",

  "version": "1.0.0",

  "private": true,

  "engines": {

    "node": ">= 7.6.0"

  },

  "scripts": {

    "prod": "node ./start.js",

    "watch": "nodemon ./start.js --ignore public/",

    "start": "concurrently \"npm run watch\" \"npm run assets\" --names \"💻,📦\" --prefix name",

    "assets": "webpack -w --display-max-modules 0",

    "sample": "node ./data/load-sample-data.js",

    "blowitallaway": "node ./data/load-sample-data.js --delete",

    "now": "now -e DB\_USER=@db\_user -e DB\_PASS=@db\_pass -e NODE\_ENV=\"production\" -e PORT=80"

  },

  "browserslist": "last 3 versions",

  "dependencies": {

    "@google/maps": "^1.1.0",

    "autocomplete": "0.0.1",

    "axios": "0.15.3",

    "bling": "^0.12.0",

    "body-parser": "1.17.1",

    "connect-flash": "0.1.1",

    "connect-mongo": "2.0.3",

    "cookie-parser": "1.4.3",

    "dompurify": "0.8.5",

    "dotenv": "4.0.0",

    "es6-promisify": "5.0.0",

    "express": "4.15.2",

    "express-session": "^1.17.0",

    "express-validator": "3.1.2",

    "faker": "4.1.0",

    "forever": "0.15.3",

    "glob": "7.1.1",

    "html-to-text": "3.2.0",

    "jimp": "^0.2.27",

    "juice": "4.0.2",

    "md5": "2.2.1",

    "moment": "2.17.1",

    "mongoose": "5.1.0",

    "mongoose-mongodb-errors": "0.0.2",

    "multer": "1.3.0",

    "nodemailer": "3.1.5",

    "passport": "0.3.2",

    "passport-local": "1.0.0",

    "passport-local-mongoose": "4.0.0",

    "pug": "2.0.0-beta6",

    "slugs": "^0.1.3",

    "uuid": "3.0.1",

    "validator": "7.0.0"

  },

  "devDependencies": {

    "autoprefixer": "6.7.7",

    "babel-core": "6.24.0",

    "babel-loader": "6.4.0",

    "babel-preset-env": "^1.6.0",

    "concurrently": "3.4.0",

    "css-loader": "0.28.11",

    "extract-text-webpack-plugin": "2.1.0",

    "node-sass": "^4.7.2",

    "nodemon": "^1.18.3",

    "now": "^6.4.1",

    "postcss-loader": "1.3.3",

    "sass-loader": "6.0.3",

    "webpack": "2.2.1"

  },

  "description": "Just you wait folks!",

  "main": "app.js",

  "author": "",

  "license": "ISC"

}

In your root website folder

Npm install

Start website NPM start

You should now see the webpack running.

[💻]

[💻] > dang-thats-delicious@1.0.0 watch A:\Development\Node

[💻] > nodemon ./start.js --ignore public/

[💻]

[📦]

[📦] > dang-thats-delicious@1.0.0 assets A:\Development\Node

[📦] > webpack -w --display-max-modules 0

[📦]

[💻] [nodemon] 1.19.4

[💻] [nodemon] to restart at any time, enter `rs`

[💻] [nodemon] watching dir(s): \*.\*

[💻] [nodemon] watching extensions: js,mjs,json

[💻] [nodemon] starting `node ./start.js`

[📦]

[📦] Webpack is watching the files…

[📦]

[💻] Express running ? PORT 7777

[📦] Hash: cb138f299389230fcc1c

[📦] Version: webpack 2.2.1

[📦] Time: 1281ms

[📦] Asset Size Chunks Chunk Names

[📦] App.bundle.js 4.86 kB 0 [emitted] App

[📦] style.css 25.7 kB 0 [emitted] App

[📦] App.bundle.js.map 6.68 kB 0 [emitted] App

[📦] style.css.map 62.7 kB 0 [emitted] App

[📦] + 5 hidden modules

[📦] Child extract-text-webpack-plugin:

[📦] + 2 hidden modules

If no errors you should be calling the autocomplete.js when you load addstore. The binding is not yet right to the field change event but mush closer.

Then I figured out it is the startup.

Needs to be npm start instead of nodemon start

😊

## 22 - Multiple Query Promise with Async Await

In this lesson we are going to show the stores associated with a tag when you click on a tag.

This is the first time we are doing two queries inside of one controller method.

If your queries don’t depend on each other, don’t make them synchronous.

In getStoresByTag, switch from const tags = await Store.getTagsList(); to const tagsPromise = Store.getTagsList();

Next we do another query for stores : const storesPromise = Store.find({ tags: tag})

Now we have two promises but none of the data. Next we are going await from both of the promises.

const result = Promise.all([tagsPromise, storesPromise])

The result returns an array containing the tags and the stores. We can descructure that and pass it along to the view:

const [tags, stores] = await Promise.all([tagsPromise, storesPromise]);

res.render(“tag”, { tags, title: “Tags”, tag, stores });

In tag.pug we loop through the stores and display them using the storecard mixin.

.stores

each store in stores

+storeCard(store)

Promise.All() waits for all promises to pay off.

If you want to use values from one query in the next or you want them to be sequential you can do something like the following example.

async function getResults() {

  let results = [];

  let table\_1\_data = await client.query("SELECT table\_1\_id AS id FROM table\_1");

  let table\_2\_data = await client.query("SELECT table\_2\_id AS id FROM table\_2 WHERE table\_1\_id = " + table\_1\_data.rows[0].id);

  for (let table\_2\_row of table\_2\_data.rows) {

      let table\_3\_data = await client.query("SELECT table\_3\_id AS id FROM table\_3\_id WHERE table\_2\_id = " + table\_2\_row.id);

      for (let table\_3\_row of table\_3\_data.rows) {

          let table\_4\_data = client.query("SELECT table\_4\_id AS id FROM table\_4\_id WHERE table\_3\_id = " + table\_3\_row.id);

          results.push({

              "table\_1\_id": table\_1\_data.rows[0].id,

              "table\_2\_id": table\_2\_row.id,

              "table\_3\_id": table\_3\_row.id,

              "table\_4\_id": table\_4\_data.id

          });

      }

  }

  return results;

}

getResults().then(results => {

  // process results here

  console.log(results);

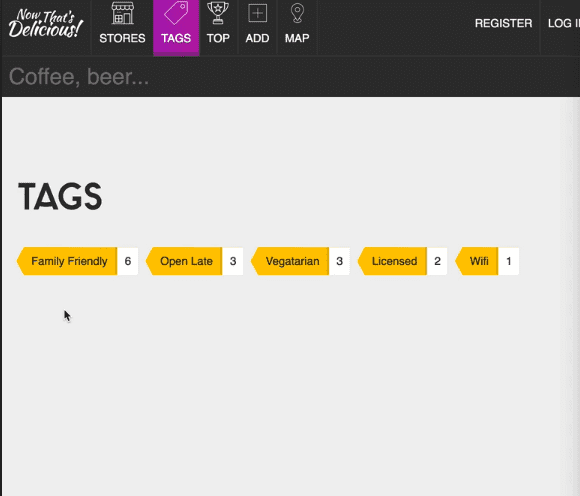
}).catch(err => {

  // process error here

  console.log(err);

});

However, now when you go to the tags page, it displays no stores

[](https://notes.anjagusev.com/static/7e5f426777ca87e4733f0eeb89e5b3e0/54e54/9340B542-1565-4EED-B2BB-C61069AE1EBF.png)

However we want to show all the stores and filter them when you select a tag.

Take out the tag from the storePromise query and create a new variable for the tag

const tagQuery = tag || {$exists:true}

This variable will be equal to the tag or the property called exists true. If there is no tag, it will fallback to the second query (just give me any store that has the tag property on it. It will show every single store that has at least one tag in it).

## 23 - Creating User Accounts

We are going to create user accounts. A user can create a store belonging to them, comment on other stores, edit their own stores.

Create the login form first. We create a route router.get(“/login”, userController.loginForm); And we will create another controller (userController). And then a View.

We are going to create a mixin for the login form so we can reuse it throughout our application.

//mixins/\_loginForm.pug

mixin loginForm()

form.form(action="/login" method="POST")

h2 Login

label(for="email") Email Address

input(type="email" name="email")

label(for="password") Password

input(type="password" name="password")

input.button(type="submit" value="Log In →")

//views/login.pug

block content

.inner

+loginForm()

Now we need to make a User model. Create new file min models called User.js & import mongoose, schema and mongoose promise (mongoose promise is just to suppress a console error).

We import a few more dependencies

const md5 = require(‘md5’);

const validator = require('validator');

const mongodbErrorHandler = require('mongoose-mongodb-errors');

const passportLocalMongoose = require('passport-local-mongoose');

We are exporting the schema as a module

const userSchema = new Schema({});

module.exports = mongoose.model(“User”, userSchema);

In the schema, adding this option lowercase:true, will ensure that a string is always saved as lowercase. We also want to do custom validation.

email: {

type: String,

unique: true,

lowercase: true,

trim: true,

validate: [validator.isEmail, “Invalid Email Address”],

required: “Please supply an email address”

},

You do that in the options by passing validate: in the options and an array of things to check, and then the error message.

We will also implement client side validation however we still need server side validation.

We also want to add password to our schema which we will save a hashed version of. We will be using a package called passport.js. It is a middleware and can be used with any node.js framework that has built in middleware. It takes away the heavy lifting for managing sessions, creating tokens, logging in, interfacing with login providers. [Passport.js](http://www.passportjs.org/)

Passport has strategies that you can implement to achieve different things.

We don’t need to add any more fields to the schema, we just need to use the passportLocalMongoose library that we imported and it will take care of adding the additional fields to our schema and additional methods to create our new logins.

We add the plugin in the schema and pass it one option, where we say the username field is going to be the email field.

userSchema.plugin(passportLocalMongoose, {usernameField:’email’})

This is saying “here is our schema, can you please add all of the methods and fields that are needed for us to add authentication to our schema and by the way I want to use email as the login field”.

Next we need to use a second plugin called mongodbErrorHandler to handle errors if someone for example tries to use an email that isn’t unique. This plugin will change those errors from ugly to nice versions.

userSchema.plugin(passportLocalMongoose, {usernameField:’email’})

Next we want to move onto posting our data to our login.

We also need a register form. Create a file views/register.pug .

extends layout

block content

.inner

h2

form.form(action=“/register” method=“POST”)

h2 Register

label(form=“name”) Name

input(type=“text” name=“name”)

label(for=“email”) Email

input(type=“email” name=“email”)

label(for=“password”) Password

input(type=“password” name=“password”)

label(for=“password-confirm”) Confirm Password

input(type=“password-confirm” name=“password-confirm”)

input.button(type=“submit” value=“Register →”)

Now we need to create the route to post our registration form. When someone posts to the registration form we need to do a few things:

1. Validate the registration data
2. Validate the user
3. Log them in

First we need to validate the data the person is registering with. We try to do that its much as possible on the moddle, but sometimes there is additional checks that need to happen before you start to send the data to the db. In that case, you can do the actual validation in the controller.

In userController we are going to create a bunch of middleware to check that we have done everything correctly.

First we are going to sanitize their name to make sure they haven’t put script tags or anything. req.sanitizeBody(‘name’)

(We can use .sanitizeBody() because in app.js we imported expressValidator from package express-validator) and then in app.js we did app.use(expressValidator()). That adds a bunch of validation methods to every request in the application and when you have a request you can just call the validation methods that live on top of it without importing the library.

Express-validator does a lot more. <https://github.com/express-validator/express-validator>

Next we need to check for a name property, check that it is not empty and provide an error.

req.sanitizeBody(“name”);

req.checkBody(“name”, “You must supply a name!”).notEmpty();

req.checkBody(“email”, “That email is not valid!”).isEmail();

Next we need to normalize email. Sometimes when people signup, they use a variation of emails like: Wesbos@gmail.com wesbos@gmail.com wes.bos@gmail.com wesbos@googlemail.com wesbos+test@gmail.com

Which are all the same thing. .normalize() will normalize them to wesbos@gmail.com . You can turn the various settings on and off.

We also make sure the password and password-confirm fields are not empty.

Last we check that the confirm password and password match.

Next we need to call the validation and get the errors -> const errors = req.validationErrors().

If something goes wrong, we want to show the errors: req.flash(“error”, errors.map(err => err.msg)); The errors will contain an array of all the errors and the fields that they are related to.

If something goes wrong we want to re-render the register form but we don’t want to clear everything .

So we re-render the register form, and we have to pass the body, and the flashes along. Normally the flashes get passed along because it’s on the next request, but since all of the validation and rerendering is happening on a single request, we need to explicitly pass the flashes.

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

req.sanitizeBody("name")

req.checkBody("name", "You must supply a name!").notEmpty()

req.checkBody("email", "That email is not valid!").isEmail()

req.sanitizeBody("email").normalizeEmail({

remove\_dots: false,

remove\_extension: false,

gmail\_remove\_subaddress: false,

})

req.checkBody("password", "Password cannot be blank!").notEmpty()

req

.checkBody("password-confirm", "Confirmed Password cannot be blank!")

.notEmpty()

req

.checkBody("password-confirm", "Oops! your passwords do not match")

.equals(req.body.password)

const errors = req.validationErrors()

if (errors) {

req.flash("error", errors.map(err => err.msg))

res.render("register", {

title: "Register",

body: req.body,

flashes: req.flash(),

})

return // stop the fn from running

}

next() //there were no errors!

}

Now we have to hook up the validation method to our router.

router.post(“/register”, userController.validateRegister);

## 24 - Saving Registered Users to the Database

We need to import the User model into our start.js

require(`./models/User`)

Now go to userController and grab model that we imported. const User = mongoose.model(“User”);

Add another middleware to the route:

router.post(“/register”,

userController.validateRegister,

userController.register);

In userController add another method which will be async which will be responsible for saving the user to the db.

create the user: const user = new User({ email: req.body.email, name: req.body.name });

However we don’t save yet. We aren’t going to call .save() we are going to call .register(). .register() is the method that will take the password that we pass it and hash it and save it to our actual database.

Where did .register() come from? In our user model, we imported the passportLocalMongoose plugin. That exposed to us the method called .register() which is going to take care of the registration for us.

To the register method we pass: the user, the password, and then a call back ->

User.register(user, req.body.password, function(err, user) {})

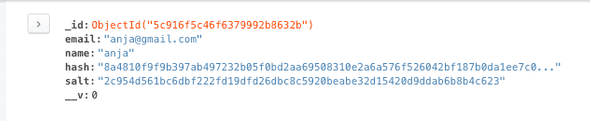
We use a callback because the library doesn’t use promises OR you can use the promisify library to take the callback based function and turn it into a promise based function .

How to turn it into promisifyed function. First, make the method that is promisified:

You take promisify and you pass it two things:

1. The method that you want to promisify (User.register)
2. Because it’s a method and not just a top level function, we need to pass it which object to bind to (User) … (if you ever call promisify on a method that is on an object, you also have to pass the entire object so it knows where to bind itself to).

Now we have this register method which we can await: await register(user, req.body.password) -> it is going to take the password the email and the name and it’s not going to store the actual password, it will store the hash in the database.

Now you can see the user in the db:[](https://notes.anjagusev.com/static/d64fe6a3524fc84e582d3be881ed91c2/f84ea/9EB82ECF-1FCA-4565-A00C-C6F9AF779ADA.png)

Now we need to log them in automatically.

Next we are going to create another controller called the authcontroller which will handle all of our logging in, passport.js, password reset and email. Anything about being logged in and authenticating.

Next we need to send passport data and it should tell us whether the user should be logged or not, and that is called a strategy in passport (strategy is something that will interface with checking if you are allowed to login)

We are using a local strategy to check if username and password have been sent in correctly.

We are going to create a login middleware using methods from passport. First we pass it the strategy (local)

Passport handles the failure redirect and success redirects.

Next add the login middleware to the register route, and import the authController.

However, before we use any of the passport strategies, we need to configure them in our application. Before local, we need to tell it what to do with the actual users once they have signed in (we are going to put the User object on each request).

Create one more file in the /handlers/ directory called passport.js which will configure our actual passport.

Import passport, mongoose, and User model.

Next we say: passport.use(User.createStrategy()); (we can do that because we used that plugin in our users.js)

Now we have to tell passport what to do with the actual user. We are going to login to passport and then it’s going to say, okay now what information do you want on each request. We want to pass along the actual user object so we can do things like import their avatar and show the stores they have created.

const passport = require(“passport”);

const mongoose = require(“mongoose”);

const User = mongoose.model(“User”);

passport.use(User.createStrategy());

passport.serializeUser(User.serializeUser());

passport.deserializeUser(User.deserializeUser());

We need to take this and import it somewhere in our actual application.

Next we go to app.js and we require the passports handler we just created: require('./handlers/passport');

Which will invoke the code that lives inside the passport handler.

## 25 - Virtual Fields, Login-Logout Middleware and Protecting Routes

In this video we are dealing with logging in, logging out, making sure that when a user adds a store that store is related to them.

We are going to start with the logout button.

Add route router.get("/logout", authController.logout);

Add method in authController

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

req.logout();

req.flash(“success”, “you are now logged out! 👋”);

res.redirect(“/“);

};

Next we need to make post login route: router.post(“/login”, authController.login);

Next we have an avatar when a user is logged in.

A gravatar is a globally recognized avatar. Every single time you have a user’s email address, you can drum up the avatar from their email.

We don’t need to add a field for gravatar because we can use a virtual field, which is a field that is generated.

Rather than storing all that data, sometimes your fields can be generated on the fly.

For example if you are storing someones weight in kilograms, you don’t also want to store it in pounds even if you are displaying it like that.

To make a virtual field we add the following to our schema:

userSchema.virtual(‘gravatar’).get(function(){

return [WHatever we want to return here]

});

A gravatar uses a hashing of the email to display an image.

Within the function above, this line of code hashes the email const hash = md5(this.email); . We can use this to access the email because we are within the schema and will have access to that instances’ email.

userSchema.virtual(“gravatar”).get(function() {

const hash = md5(this.email);

return `https://gravatar.com/avatar/${hash}?s=200`;

});

Next, no one should be able to create a store without being logged in. In our authControllelr we will make middleware that checks if the user is logged in.

Check if user is authenticated req.isAuthenticated() -> will check with passport .

If authenticated, do next() and return; If not, flash and error and redirect to the login

Next we add the middleware to our /add get route like so: router.get("/add", authController.isLoggedIn, storeController.addStore);

Which will prevent anyone from accessing add a store

Some errors to be aware of.

Getting the Error: Failed to serialize user into session

* You are missing the two methods passport.deserializeUser and passport.serializeUser. These are required for session use and authentication.

Problems with passport-local strategy (user and password form)

* Outside the scope of this tutorial but make sure you are using the body-parser package to POST the login form.
* Make sure your form field names match the expected names in passport. If they do not match you will need to configure passport to know what field names to expect on a POST.

Passport JS has over 500 authentication “Strategies” that can be used within a Node/Express app. Many of these strategies are highly specific (i.e. passport-amazon allows you to authenticate into your app via Amazon credentials), but they all work similar within Express.

Not only does Passport consist of two modules (Passport base + Specific Strategy), but it is also a middleware

(passport-local) is a middleware that modifies an object created by another middleware (express-session).

Install a local strategy:

npm install --save passport passport-local  
Unless you customize it Passport expects the following parameters by name.

{  
"username": "sample@email.com",  
"password": "sample password"  
}

These two middleware functions are necessary for integrating PassportJS with express-session middleware. That is why these two functions **must come AFTER** the app.use(session({})) middleware! Just like passport.serializeUser() and passport.deserializeUser()

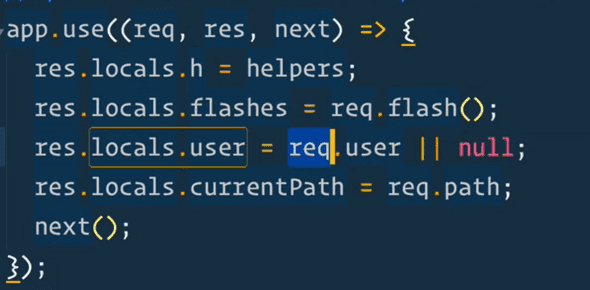
## 26 - Creating a User Account Edit Screen

We need to make /account url work. We need to add it to the routes

router.get(“/account”, userController.account);

Next we make a view, with a form to edit the account.

Within the view, we have access to the user object because it is available on every request and we pass it to our locals. In app.js we have :

[](https://notes.anjagusev.com/static/6bb9b87ccbe41049252d55f72a4d8f56/576cc/EEF1A9AC-6B3B-4E65-9A66-4A4D15608839.png)

extends layout

block content

.inner

h2 Edit Your Account

form(action=“/account” method=“POST”)

label(for=“name”) Name

input(type=“text” name=“name” value=user.name)

label(for=“email”) Email Address

input(type=“email” name=“email” value=user.email)

input.button(type=“submit” value=“Update My Account”)

We have to make sure they are logged in before they can access the accounts. Add isLoggedIn middleware to the route.

Now we make a route to handle the post of the account form. router.post(“/account”, catchErrors(userController.updateAccount));

updateAccount is async. First grab the updated values and assign them to an object. Next, update them using mongo method findOneAndUpdate() which takes three parameters, the query, the info to update and then the options.

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

const updates = {

name: req.body.name,

email: req.body.email

};

const user = await User.findOneAndUpdate(

{ \_id: req.user.\_id },

{ $set: updates },

{

new: true,

runValidators: true,

context: “query”

}

);

req.flash("success", "Updated the profile!");

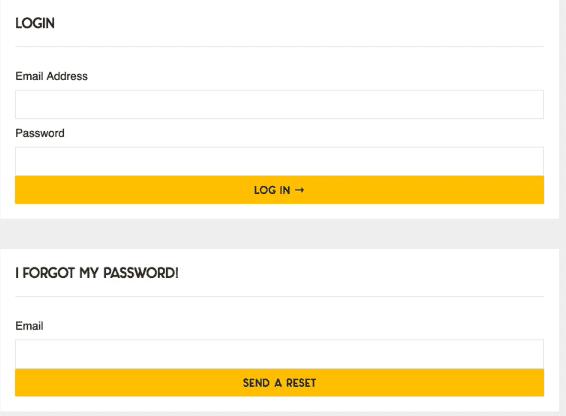
res.redirect(‘back’);

};

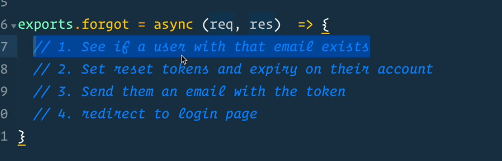
res.redirect(‘back’); <- will take the user back to wherever they were .

## 27 - Password Reset Flow

Create reset password mixin named \_forgot.pug and import it into login form.

[](https://notes.anjagusev.com/static/0f281e5692164ae8bd6c6982ea3dab04/7b273/1A9CDDD7-4C2E-4EAD-9E4C-4516A1BC7064.png)

1. Check if there is a user with that email account
2. If so, give them a token with an expiry date to reset their password
3. Those two pieces of email are emailed to them
4. If proper token and unexpired date then they need to reset the password
5. see if user exists



const user = await User.findOne({ email: req.body.email });

if (!user) {

req.flash(“error”, “No account with that email exists”);

return res.redirect(“/login”);

}

1. Import built in module crypto const crypto = require(‘crypto’);

Set reset token and expiry on their account

`user.resetPasswordToken = crypto.randomBytes(20).toString(“hex”);

`user.resetPasswordExpires = Date.now() + 3600000; //1 hour from now

1. Fds
2. Fads

We need to add the extra fields to our schema (resetPasswordToken, resetPasswordExpires)

To get the reset url, we need to get the current url using {req.headers.host} like so: http://${reg.headers.host}.account/reset/${user.resetPasswordToken};

Next we have to actually make a route for the password reset link.

router.get(‘/account/reset/:token', catchErrors(authController.reset));

Now we go make the controller method:

//authController.js

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

const user = await User.findOne({

resetPasswordToken: req.param.token,

resetPasswordExpires: { $gt: Date.now() }

});

We grab the token from the param and look for a user who has a matching token, and whose token has not expired.

Next, check if there is a user. If not, flash error and redirect to login.

If there is a user, show the reset password form.

Create a view called reset.pug. In the reset form, leave out the action. If you leave out an action on an HTML form, it posts the form to the same url it’s on.

Next we want to create the post route for the reset form. router.post(“/account/reset/:token”, authController.confirmedPasswords, catchErrors(authController.update))

Next we create the update and confirmedPasswords methods in the authController.

When you need to access a request property that has dashed in it, you need to use square brackets like req.body[‘password-confirm’]

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

if (req.body.password == req.body[password - confirm]) {

next(); //keep it going

return;

}

req.flash(“error”, “Passwords do not match!”);

res.redirect(“back”);

};

In the update method, we first check again to make sure the password hasn’t expired.

If it has or the password is invalid, redirect them to the login.

If its valid, we want to set their password. We can use the user.setPassword() method but that isn’t promisified.

Promised version:

const setPassword = promisify(user.setPassword, user)

await setPassword(req.body.password)

\*\*\* TO KNOW \*\*\*

req.headers.host;

or

req.header('host');

give you the same thing. See these links for more.

[Express.js guide - request.hostname](http://expressjs.com/api.html#req.hostname)

[Express.js guide - request.redirect](http://expressjs.com/api.html#res.redirect)

One more thing to know. Req.hostname returns the same thing as req.headers.host but without a port. In other words the req.headers.host preserves the port number. But starting with express v5 either will return the port. So you have essentially 3 ways to get the same data.

Next we have to set the resetPasswordToken and resetPasswordExpires fields to undefined because we no longer need them and that’s how you get rid of fields in mongo.

const setPassword = promisify(user.setPassword, user)

await setPassword(req.body.password)

user.resetPasswordToken = undefined

user.resetPasswordExpires = undefined

const updateUser = await user.save()

await req.login(updatedUser)

req.flash(

"success",

"Nice! Your password has been reset. You are now logged in!"

)

res.redirect("/")

## 28 - Sending email with Node.js

POSTMARK APP, MailGun, SendGrid are for production use

We are going to use mailtrap.io a service which traps the mail you send so you can see it instead of sending in a dev environment.

Go to variables.env and update the mail username and password.

Next create a new handler called mail.js and we need to import node mailer.

Nodemailer will interface with SMTP or any other transports and it will do the sending of the email for you.

Const node mailer = require(‘nodemailer’)

Now we need to create a transport. A transport is the way you interface with different ways of sending emails (SMTP being the most common)

const transport = nodemailer.createTransport({

host:process.env.MAIL\_HOST,

port: process.env.MAIL\_PORT,

auth:{

user: process.env.MAIL\_USER,

pass: process.env.MAIL\_PASS

}})

transport.sendMail({

from: ‘Wes Bos <wesbos@gmail.com>`,

to: `randy@example.com`,

subject: ‘Just trying things out!’,

html: ‘Hey I <strong>love</strong> you’,

text: ‘Hey I \*\*love you\*\*”

And then you send the email..

To test this we will run it on app start up.

In the bottom of start.js we will temporarily add this code: require(‘./handlers/mail’) Clear the console and then save start.js and check mailtrap.io

When someone resets their password we are going to call a method called send.

We want to create the mail options object with from, to, html and text values and then promisify the send method.

Exports.send = async options => {

const mailOptions = {

from: `Anja Gusev <noreply@anjagusev.com>`,

to: option.user.email,

subject: options.subject,

html: “This will be filled in later”,

text: “This will also be filled in later”

};

const sendMail = promisify(transport.sendMail, transport);

return sendMail(mailOptions);

};

Next, we want to send the email when someone wants to reset their password. In authController, we have a password reset method (.forgot) where we are returning the reset link in a success message. Remove the reset url from the success message. Import the mail handler we created earlier, and before we show the success message, send the mail like so:

await mail.send({

user,

subject: “Password Reset”,

resetUrl,

filename: “password-reset”,

})

Next we have to generate the actual HTML to send. We are going to create a method called generateHTML() which takes in two parameters, a filename and options which defaults to an empty object. We are going to make that a const instead of an export because it is only needed within this file.

Const generateHTML = (filename, options = {}) => {

const html = pug.renderFile()

}

You pass the renderFile() method the name of the file you are looking for.

There is a problem. Whenever you pass a function a reference to something on your disk, you don’t actually know where you are in the file system because the renderFile method lives in a different folder. What we do is use to reference a variable that is available in every file \_\_dirname.

Const generateHTML = (filename, options = {}) => {

const html = pug.renderFile(

`${\_\_dirname}/../views/email/{filename}.pug`,

options

);

We pass in the pug email template as the first param and the options as the second.

Next, we modify the send function to get the html and pass it to the send function ->

exports.send = async options => {

const html = generateHTML(options.filename, options)

const mailOptions = {

from: “Anja Gusev <noreply@anjagusev.com>”,

to: options.user.email,

subject: options.subject,

html,

text: “This will also be filled in later”,

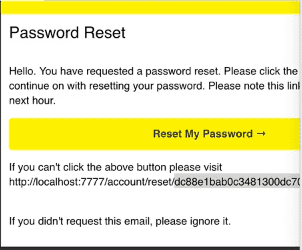
}

const sendMail = promisify(transport.sendMail, transport)

return sendMail(mailOptions)

}

Now when you trigger a password reset email you get the following ->

[](https://notes.anjagusev.com/static/b2dcf398e2c37de6bdf63a1e47367732/37f7a/6C4F2AAD-039F-4AE4-8FC3-F8FAA57E9FDA.png)

However the plain text version is not correct and the css is not inlined.

For plain-text, in the send method, you add the following code: const text = htmlToText.fromString(html); and replace it in the mailOptions object.

Juice is a library which will look at your style tags and inline them all appropriately by matching selectors etc.

Modify the generateHTML() method like so ->

const generateHTML = (filename, options = {}) => {

console.log(\_\_dirname)

const html = pug.renderFile(

`${\_\_dirname}/../views/email/${filename}.pug`,

options

)

const inline = juice(html)

return inline

\*\*\* TO KNOW \*\*\*

In Node.js, \_\_dirname is always the directory in which the currently executing script resides ([see this](https://nodejs.org/api/modules.html#modules_dirname)). So if you typed \_\_dirname into /d1/d2/myscript.js, the value would be /d1/d2.

By contrast, . gives you the directory from which you ran the node command in your terminal window (i.e. your working directory) when you use libraries like path and fs. Technically, it starts out as your working directory but can be changed using process.chdir().

The exception is when you use . with require(). The path inside require is always relative to the file containing the call to require.

## 29 - Locking down App with User Permissions

Right now we have stores and users collections in mongo but no relationship between them. We need to add an author store to the stores.

Add author field to the Store.js model of type ObjectId and referencing the “User” model.

author: {

type: mongoose.Schema.ObjectId,

ref: “User”,

required: "You must supply an author"

}

Now we have to make sure when you create a store there is a user associated.

In the storeController modify the createStore method like so:

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

req.body.author = req.user.\_id;

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

If you want to populate the author field with the actual user object you can call .populate(‘author’) on a database find request like so in storeController.js ->

const store = await Store.findOne({ slug: req.params.slug }).populate(

“author”

);

Now we need to prevent people from editing stores which they do not own

In the editstore method, before they actually edit the store, we have to make sure they are the owner.

const confirmOwner = (store, user) => {

if (!store.author.equals(user.\_id)) {

throw Error("you must own a store in order to edit it")

}

}

The .equals method comes along because the store.author is an objectid and in order to compare an objectid with an actual string, we use the .equals method. If it’s not a match, throw an error.

We aren’t using middleware here because we need to look up the store before we perform the check.

So we are making a function called confirm owner and call it from edit store, which should throw an error if they are not which is hopefully caught by our error handling middleware.

We currently have incomplete data because some of our stores do not have authors associated. We need to cleanup mongo at this step.

**ANJA DONT FORGET**

Now we need to edit the stores interface to only show the edit pencil if it is a store the user owns.

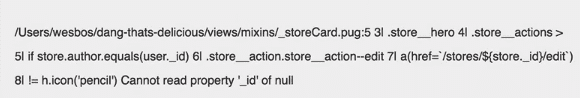
To do that, we need to edit \_storeCard.pug to add a conditional

if store.author.equals(user.\_id)

.store\_\_action.store\_\_action—edit

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

!= h.icon(‘pencil’)

However, now if you open up the stores view in incognito you get this error[](https://notes.anjagusev.com/static/bbea3494a7716e41ead96a1681b88943/54e54/FF248EC8-9F8F-41B2-AB93-14BB1F8611A8.png)Because no one is logged in. We need to add check for user in the if statement

if user && store.author.equals(user.\_id)

You could create different security roles by adding a “level” property to the user object for example.

## 30 - Loading Sample Data

If you open package.json file, there are a few scripts. The sample and blowitallaway will run a script that loads a bunch of sample data.

load-sample-data.js

npm run sample

Note: Make sure database exists first.

## 31 - JSON endpoints and creating mongodb indexes

When you search for coffee, it should show a list of stores with coffee in the name and description.

We are going to add indexes.

In the store schema Store.js we will define the indexes

//define our indexes

storeSchema.index({

name: "text",

description: "text",

})

We can tell mongodb what to index the field as. We are going to tell it to index the field as text so we can search strings and also search by case sensitive.

If you refresh compass you will see the indexes under stores. The indexes we created is a compound index.

Now we are going to make an api endpoint.

router.get(“/api/search”, catchErrors(storeController.searchStores));

This endpoint is going to take one param ?q=searchterm which is actually a query object not a param.

You can access it like so within the storeController.js

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

res.json(req.query)

}

Now we need to search the db for the stores that include name or description

Since we indexed the two fields as compound indexes and as text, we can use the mongodb text operator which will perform a text search on any fields that are indexes with text index.

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

const stores = await Store.find({ $text: { $search: req.query.q }, score: {$meta:'textScore });

res.json(stores);

};

However it doesn’t put them in any specific order, just orders them by the order they were added to the database. We need a sort based on a “score” where the store where the store where the search term shows up the most has the highest score.

We can use $meta operator, it allows us to project a field using some of the hidden data that mongo has about our search results. We can add it to the actual end result.

We take our find, and pass a second object to tell it to project field called score which grabs the textStore from the meta object.

Now we want to sort the query.

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

const stores = await Store.find(

{ $text: { $search: req.query.q } },

{ score: { $meta: “textScore” } }

).sort({ score: { $meta: “textScore” } });

res.json(stores);

};

Now we only want to return 5 results. Add .limit(5);`

## 32 - Creating an Ajax Search Interface

Create a file in this directory named typeAhead.js /public/javascripts/modules/

Next open file delicious-app.js and import typeAhead. Modify it like so ->

import “../sass/style.scss”;

import { $, $$ } from “./modules/bling”;

import autocomplete from “./modules/autocomplete”;

import typeAhead from “./modules/typeAhead”;

autocomplete($(“#address"), $("#lat"), $("#lng"));

typeAhead($(“.search”));

In the typeAhead function, add the following so if search doesn’t exist this method doesn’t return anything if (!search) return;

Now we want to grab the search inputs from the page to grab the search query and display the results.

const searchInput = search.querySelector('input[name="search"]')

const searchResults = search.querySelector(".search\_\_results")

We will next use blingjs syntax for add a listener to the input.

When the input function runs ,we run a callback which will log the value.

searchInput.on(“input”, function() {

//if there is no value,quit it

if(!this.value){

searchResults.style.display = "none";

return // quit it

}

});

Next we want to show the search results searchResults.style.display = ‘block’;

And then we use axios to hit our endpoints and we get the results.

searchResults.style.display = "block"

axios.get(`/api/search?q=${this.value}`).then(res => {

console.log(res.data)

})

This is the effect we want to have when someone searches -> the user can tab up and down and then when they select one, it will take them to that store.

Instead of writing this directly in the function, we will create another method called searchResultsHTML which takes in an array of stores and returns some HTML.

function searchResultsHTML(stores) {

return stores

.map(store => {

return `

<a href="/stores/${store.slug}" class="search\_result">

<strong>${store.name}</strong>

</a>

`

})

.join("")

}

call this function from within the typeAhead function.

function typeAhead(search) {

if (!search) return

const searchInput = search.querySelector('input[name="search"]')

const searchResults = search.querySelector(".search\_\_results")

searchInput.on("input", function() {

if (!this.value) {

searchResults.style.display = "none"

return //stop

}

searchResults.style.display = "block"

axios.get(`/api/search?q=${this.value}`).then(res => {

if (res.data.length) {

searchResults.innerHTML = searchResultsHTML(res.data)

}

})

})

}

Now when you type coffee, 3 stores show up. One issue is when you start backspacing, the stores are still showing.

To solve that, before the axis request and after setting the style to display block, add searchResults.innerHTML=‘’; which will remove the html for that component if there are no matches.

At this stage, the search dropdown works and you can click one of the results and go to the store, however you cannot navigate the search using up and down arrows on your keyboard. We need to handle the keyboard inputs.

Add the following code to figure out which inputs we need to listen for:

//handle keyboardINputs

searchInput.on(“keyup”, e => {

console.log(e.keyCode);

});

UP = 38 Down = 40 ENTER = 13

[Screen Recording 2019-03-29 at 21.58.gif](https://cl.ly/573927962939) One issue we have here is that when you use the up and down arrows, when you get to the bottom, if you press down it should go to the top. Similarly if you are on the first result and go up, it should go to the bottom.

The active search result will have a class marking it as currently active.

What we need to do is find the current active result, and all the other results

const current = search.querySelector(`.${activeClass}`);

const items = search.querySelectorAll(“.search\_\_result”);

Next we need a let variable called next because we will be updating it.

We are going to figure out what are we currently on, and if someone presses down or up, which is going to be the next one.

If they press down, and there is one selected, we want to set next to be the one after it or if on the last one, fall back to the first item.

if (event.keyCode === 40 && current) {

next = current.nextElementSibling || items[0]

}

Then, if the key code is equal to 40, and there is no active one, set the next item to the first item.

else if (e.keyCode === 40) {

next = items[0];

}

If the key code is 38 (up) and there is a current one, set the next item to be the previous sibling element, or if there is no previous sibling element, the last item is next

else if (e.keyCode === 38 && current) {

next = current.previousElementSibling || items[items.length - 1];

}

If the up key is pressed and there is no currently selected item, set the next item to be the last item.

else if (e.keyCode === 38) {

next = items[items.length - 1];

}

If someone submits key code 13 and there is an href value, set the window location to that value, and return to prevent this method from continuing to run

window.location = current.href;

return;

}

Now we want to add the active class to the next item, and remove the current active class from the current item.

if (current) {

current.classList.remove(activeClass)

}

next.classList.add(activeClass)

On thing from earlier is we are requiring axios rather than importing it like we are in the rest of our files. The reason it worked is because we are using web pack and it gets compiled to common javascript after.

Modify from const axios = require(“axios”); To import axios from ‘axios’ To keep it consistent where we are using es6 modules on the front end and common runtime modules on the server side because es6 modules aren’t available yet.

When someone is searching for something and there are no results, we want to show them that there are no results.

Remove searchResults.innerHTML = “”;

And in the axios modify like so

searchResults.innerHTML = `<div class=“search\_\_reslts”>No results for ${

this.value

} found!</div>`

We are open to a cross site scripting attack here however.

Lets say you put an image tag in the name and put an onLoad event that alerts the user “you’ve been hacked”. Pug will take care of this on the front end by not actually loading the image. However in the search results we are adding the html.

Before you add HTML you should sanitize it.

We should also do it on the search input because someone could inject something by searching for it.

if (res.data.length) {

searchResults.innerHTML = dompurify.sanitize(searchResultsHTML(res.data))

}

searchResults.innerHTML = dompurify.sanitize(

`<div class=“search\_\_results”>No results for ${this.value} found!</div>`

)

That will now cause the onLoad alert to not trigger when you search for that store, although the image will still show up.

If you wanted to prevent an image from being added ,you could go to the store model and add a pre-save event where you strip out all unnecessary tags.

## 33 - Creating a Geospatial Ajax Endpoint

We are going to be making a store locator with google maps built in We are going to make an api endpoint that we can ping given a lat and lng and it will return the top 10 closest stores within x distance.

It will centre the map, take everything you got from the api and plop the markers.

We need to index our data, create a route etc like we did for the search functionality.

We are going to tell it that the location is geospatial, it will store the metadata about that location as geospatial data and then it will be able to quickly and easily search for things near the lat and lng we pass it.

Add this code ->

//Store.js

storeSchema.index({ location: “2dsphere” });

If you refresh, you will see these indexes have been added on the store object.

Next we create a route and then a method to handle that route.

router.get(‘/api/stores/near’, catchErrors(storeController.mapStores))

In storeController add a mapStores method for the endpoint which accepts query parameters like so:

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

const coordinates = [req.query.lng, req.query.lat]

res.json(coordinates)

}

However this will give us the lat & lng in string format. To unstring it we do:

const coordinates = [req.query.lng, req.query.lat].map(parseFloat);`

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

const coordinates = [req.query.lng, req.query.lat].map(parseFloat);

const q = {

location: {

$near: {

$geometry: {

type: “Point”,

coordinates: coordinates

},

$maxDistance: 10000//10 km

}

}

};

const stores = await Store.find(q);

res.json(stores);

};

One thing we want to do is keep our Ajax requests as slim as possible.

We need the name, photo, description and address.

You can chain .select() method on the mongo query.

To only select some you do -> const stores = await Store.find(q).select(‘name photo’);

To exclude fields you do -> const stores = await Store.find(q).select(‘-author -tags’);

We will select the following: const stores = await Store.find(q).select(“slug name description location”).limit(10)

This will keep our Ajax request quick and light.

## 34 - Plotting Stores on Custom Google Map

Make a route for /map url, add a mapPage method to the store controller and create a map.pug view.

After that, we have to write the client side javascript. Add new file in /public/javascripts/modules/map.js

In that file, Import axios, create two functions called loadPlaces and makeMap and export it.

In delicious-app.js import it and use it. makeMap($(#map)

Now on the map page, the console should log the div. However, on the store page where there is no map it will be null so we have to handle that if (!mapDiv) return;

On the layout page, the javascript library for maps is already loaded so we can just make our map.

First we will worry about getting the map on the page, and then about putting the markers.

To get the map on the page, we need to pass some options.

const mapOptions = {

center: { lat: 43.2, lng: -79.8 },

zoom: 10,

}

To create the map, you pass it the div and then the options like so ->

const map = new google.maps.Map(mapDiv, mapOptions); Next line call other method loadPlaces(map);

We also enable autocompleting: const autocomplete = new google.maps.places.Autocomplete(input);

When the page loads, we run makeMap which will make the map, once it’s done being created loadPlaces will take that map and take a specific lat and lng to hit the endpoint with.

Inside of loadPlaces, we need to return if there are no places found.

Next we need to add markers.

const markers = places.map(place => {

const [placeLng, placeLat] = place.location.coordinates

})

-> we are destructuring. The order of lng and lat matters here.

next we need to create a position object and pass it to create our markers like so:

const position = { lat: placeLat, lng: placeLng }

const marker = new google.maps.Marker({ map, position })

We also want to pass along the location information and attach it to the marker. Add the following code ->

marker.place = place

return marker

One problem we have is zoom level. We can use “bounds” which will detect where the markers are but will zoom in as far as it can with all the markers still in place.

const bounds = new google.maps.LatLngBounds();

And before we pass the marker to the google maps method, we want to do: bounds.extend(position);

Once we are doing looping over all the markers, we want to zoom the map to fit them perfectly

map.setCenter(bounds.getCenter())

map.fitBounds(bounds)

However when you click on the markers right now, nothing happens. Now we need to make an info window.

const infoWindow = new google.maps.InfoWindow();

Next, once we have all the markers, we want to loop over then and add an event listener for onclick.

//when someone clicks on a marker, show the details of that place

markers.forEach(marker =>

marker.addListener(“click”, function() {

console.log(this);

})

);

You want to use a regular function because the “this” variable contains info about the marker and has the place info in it.

We need to create some html to pass to the info window using the places info.

//when someone clicks on a marker, show the details of that place

markers.forEach(marker =>

marker.addListener("click", function() {

infoWindow.setContent(this.place.name)

infoWindow.open(map, this) //we pass this so it opens ontop to the marker.

})

)

When you click on a marker, the name appears. There is only one info window , so when you click another marker that marker info window disappears and the newly clicked one gets a info window.

Now we want to add information into the info window like so:

//when someone clicks on a marker, show the details of that place

markers.forEach(marker =>

marker.addListener(“click”, function() {

const html = `

<div class=“popup”>

<a href=“/store/${this.place.slug}”><img src=“/uploads/${this.place

.photo || “store.png”}” alt=“${this.place.name}”/></a>

<p>${this.place.name} - ${this.place.location.address}</p>

</div>`;

infoWindow.setContent(html);

infoWindow.open(map, this); //we pass this so it opens ontop to the marker.

})

);

Next is when you type in an address on the map, the map should take the lat and lng of the place you typed in and find markers within that area.

To do this, we go back to our makeMap function and listen to when the autocomplete changes itself. We also want to load the places.

const autocomplete = new google.maps.places.Autocomplete(input)

autocomplete.addListener("place\_changed", () => {

const place = autocomplete.getPlace()

loadPlaces(map, place.geometry.location.lat(), place.geometry.location.lng())

})

## 35 - Pushing User Data to our API

We will add the ability to like stores, and then the ability to filter stores to only see the ones you’ve liked.

First we need to modify the user schema because every time you like a store, that stores id will get added to your user account.

Open up Users.js and add a new property called hearts which will take an array of objects with type of mongoose schema objectId, which references the store.

hearts: [{ type: mongoose.Schema.ObjectId, ref: “Store” }]

Next modify \_storeCard.pug so that if the user is logged in, they can heart it.

.store\_\_action.store\_\_action—heart

form.heart(method=“POST” action=`/api/stores/${store.\_id}/heart`)

button(type=“submit” name=“heart”)

!= h.icon(‘heart’)

Now we need to make a route to handle that.

Next we make our controller method. We need to grab all the stores the user has currently liked, so we know to unlike if they press the same store again or if it’s not on the list, when the user presses like we should add it to the list.

const hearts = req.user.hearts.map(obj => obj.toString()); <- we can use toString here because mongo have overwritten the toString method .

const operator = hearts.includes(req.params.id) ? ‘$pull’: ‘$addToSet’; -> if there is already the heart id, pull it from the list. If there isn’t add it to the set. We use addtoset rather than push so we don’t duplicate the value.

const user = await User.findByIdAndUpdate(

req.user.\_id,

{ [operator]: { hearts: req.params.id } },

{ new: true }

)

The new operator specifies return the updated user, not the user as they were before the update.

In the layout, you can view how many hearts you currently have due to this code:

if user

li.nav\_\_item: a.nav\_\_link(href=“/hearts”, class=(currentPath.startsWith(‘/hearts’) ? ‘nav\_\_link—active’ : ‘’))

!= h.icon(‘heart’)

span.heart-count #{user.hearts && user.hearts.length}

[38C54725 FF83 4189 AECE A36DA7B7AFDB](https://notes.anjagusev.com/static/935ebac8601613ec862b59bfcbf1d36c/e551d/38C54725-FF83-4189-AECE-A36DA7B7AFDB.png)

We will update the heart so it’s red if we have hearted it.

First we need a list of all of the heart strings, and then look if that store is include in it.

.store\_\_action.store\_\_action—heart

form.heart(method=“POST” action=`/api/stores/${store.\_id}/heart`)

-const heartStrings = user.hearts.map(obj=>obj.toString())

-const heartClass = heartStrings.includes(store.\_id.toString()) ? ‘heart\_\_button—hearted’ : ‘’

button(type=“submit” name=“heart” class=heartClass)

!= h.icon(‘heart’)

-> This will show a red heart is on the stores that are hearted

Now we will use api endpoint to do that automatically without leaving the page, turn the colour, add to the user and update the heart count.

Create new js file in public/javascript/modules called heart.js

In that we export axios and make a method that will take an event (we are going to hook it up to the submit form we have)

In delicious-app.js import ajaxHeart method we created in the hearts file and grab all the heart forms, and bind the on submit to the ajaxHeart method like so ->

const heartForms = $$(“form.heart”);

heartForms.on(“submit”, ajaxHeart);

The $$ is query selector all. And we can call “on submit” directly to the node list using blingjs.

Next in heart.js we are going to post via javascript rather than the form.

We are going to post to this.action.. this is the form and the action is the url we want to hit:

function ajaxHeart(e) {

e.preventDefault()

console.log("HEART ITTTT")

axios

.post(this.action)

.then(res => {})

.catch(console.error)

}

Inside the .then add the following ->

.then(res => {

const isHearted = this.heart.classList.toggle(“heart\_\_button—hearted”);

console.log(isHearted);

That will toggle the heart red or white depending on whether it has been liked.

Now we need the number to be updated in the header. In the header there is a class .heart-count which is responsible for the number of hearts.

Inside the then, we grab it using bling and write the following code -> $(“.heart-count”).textContent = res.data.hearts.length;

Res.data is the user object and it contains the number of hearts.

Now we are adding a css animation to the heart. On the button we have an after element with content heart. When there is a class of float, we add an animation of fly that will transition it from where it is to the top.

After 2.4 seconds we want to remove the heart button float class.

.then(res => {

const isHearted = this.heart.classList.toggle(“heart\_\_button—hearted”);

$(“.heart-count”).textContent = res.data.hearts.length;

if (isHearted) {

this.heart.classList.add(“heart\_\_button—float”);

setTimeout(

() => this.heart.classList.remove(“heart\_\_button—float”),

2500

);

}

We use the timeout with an arrow function so we can keep the this reference from the parent function.

## 36 - Displaying our Hearted Stores

We want to show the stores that we have hearted when a user navigates to to /hearts

We can do that two ways.. we could query the current user and call .populate() on their hearts, or we can query a bunch of stores and find the stores who ids are in the current heart array.

One issue is if a user is not logged in and they navigate to this tore, it will give them an error so we need to add middleware that will double check if the person is logged in and tell them they must login to view if they haven’t.

router.get(

"/hearts",

authController.isLoggedIn,

catchErrors(storeController.getHearts)

)

## 37 - Adding Reviews Data Model

Add a review data model called Review.js and add it to start.js.

const mongoose = require(“mongoose”);

mongoose.Promise = global.Promise;

const reviewSchema = new mongoose.Schema({

created: {

type: Date,

default: Date.now

},

author: {

type: mongoose.Schema.ObjectId,

ref: “User”,

required: “You must supply an author!”

},

store: {

type: mongoose.Schema.ObjectId,

ref: “Store”,

required: “You must supply a store!”

},

text: {

type: String,

required: “Your review must have text!”

},

rating: {

type:Number,

min: 1,

max:5

}

});

module.exports = mongoose.model(“Review”, reviewSchema);

Next create a reviewForm as a mixin and import it to the bottom of store.pug file. Display the form if the user is logged in.

If you want to be able to click a label and select the input that is associated with it, you need to make sure your for and id are identical like so:

input(type=“radio” required id=`stars${num}` name=“rating” value=num)

label(for=`stars${num}`)

Entire review form so far:

mixin reviewForm(store)

form.reviewer(action=`/reviews/${store.\_id}` method=“POST”)

textarea(name=“text” placeholder=“Did you try this place? Have something to say? Leave a review…”)

.reviewer\_\_meta

.reviewer\_\_stars

each num in [5,4,3,2,1]

input(type=“radio” required id=`star${num}` name=“rating” value=num)

label(for=`star${num}`) #{num} Stars

input.button(type=“submit” value=“Submit Review →”)

Create reviewController.js and addReview method inside of it.

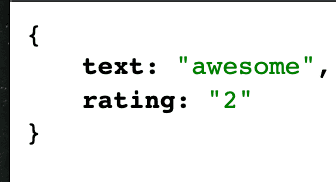
const mongoose = require(“mongoose”);

const Review = mongoose.model(“Review”);

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

res.json(req.body);

};

[](https://notes.anjagusev.com/static/5a354362f9f400e16e9e7410e4ea9861/d51f7/208B585C-BD11-4F94-874F-073299160A22.png)

To add author and store to the request, and save the review we can use this code ->

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

req.body.author = req.user.\_id;

req.body.store = req.params.id;

const newReview = new Review(req.body);

await newReview.save();

req.flash(“success”, “Review saved!”);

res.redirect(“back”);

};

## 38 - Advanced Relationship Population - Displaying our Reviews

In this tutorial we are going to get reviews to show up on a stores page. We need a second query where we search for any reviews with that store’s id.

We can also do a virtual populate which is new in mongoose.

In the store schema we add.. We are going to tell it to go off to another model and do a query.

//find reviews where the stores \_id property ===reviews store property

storeSchema.virtual(“reviews”, {

ref: “Review”, //what model to link?

localField: “\_id”,//which field on the store?

foreignField: “store”//which field on the review

});

In storeController.getStorebySlug add .populate(‘author reviews’);

//storeController.js

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

//1. Query database for a store based on slug

const store = await Store.findOne({ slug: req.params.slug }).populate(

"author reviews"

)

if (!store) return next()

// res.json(store);

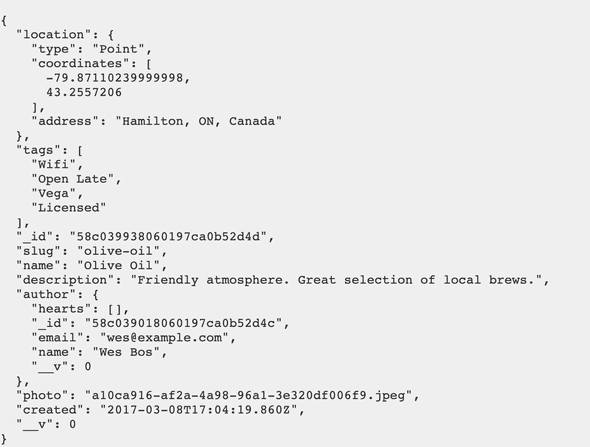
res.render("store", { title: store.name, store })

}

Virtual fields do not actually go to an object or JSON unless you specifically ask it to. So if in store.pug you did ->

block content

pre=h.dump(store)

You would see the following:[](https://notes.anjagusev.com/static/2cff337ddd1f18e90b5c648e3b95b645/a83f2/70CE00E2-64DB-490B-9EAB-7C4412FB28A6.png)

But if you explicitly asked for pre=h.dump(store.reviews) then it would show the reviews.

If you want to make the virtual fields show up in the .toJSON or the .toObject you can modify the schema like so:

//Store.js

author: {

type: mongoose.Schema.ObjectId,

ref: “User”,

required: “You must supply an author”

}

},

{

toJSON: { virtuals: true },

toObject: { virtuals: true }

}

);

In store.pug add the following:

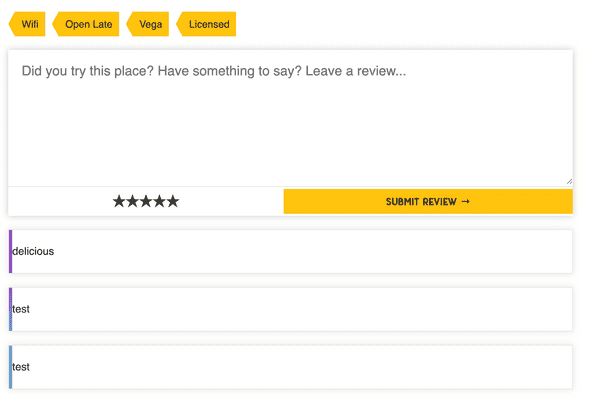
if store.reviews

.reviews

each review in store.reviews

.review

p= review.text

Should look like ->[](https://notes.anjagusev.com/static/62ee04b2e65d255dce453c50f2e1e3fc/1ebe0/3D002EAA-8532-4C36-9473-6AB8E05CF36B.png)

Add a mixin titled \_review.pug

mixin review(review)

.review\_\_header

.review\_\_author

.review\_-starts

time.review\_\_time

.review\_\_body

p= review.text

One problem we have is in the review json only contains the id of the author (it’s not populated). What we need to do is to ensure when the review is queried, it’s going to auto-populate the author field.

Go back to Review.js model and we will create a function called autopopulate()

function autopopulate(next) {

this.populate(“author”);

next();

}

Now we can add hooks so whenever review is queried, we are going to automatically populate the author field.

Add this code ->

reviewSchema.pre(‘find’, autopopulate);

reviewSchema.pre(‘findOne’, autopopulate)

Which will add hooks so that anytime someone finds or finds one, it will populate the author field.

In \_review.pug we add ->

//\_review.pug

mixin review(review)

.review\_\_header

.review\_\_author

img.avatar(src=review.author.gravatar)

p= review.author.name

.review\_\_stars(title=`Rated ${review.rating} out of 5 stars`)

= `★`.repeat(review.rating)

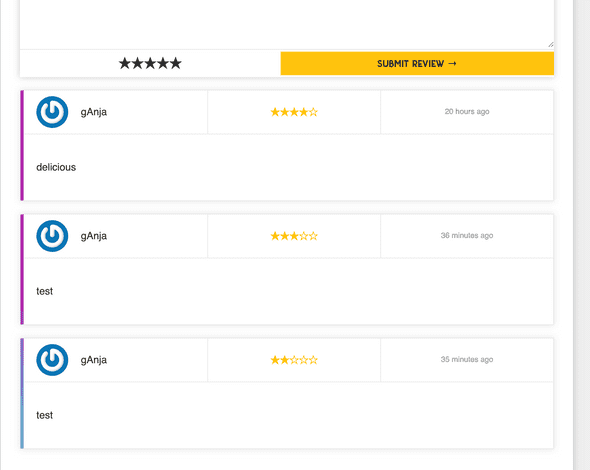
=`☆`.repeat(5-review.rating)

time.review\_\_time(datetime=review.created)= h.moment(review.created).fromNow()

.review\_\_body

p= review.text

Which in turn gives us ->

[](https://notes.anjagusev.com/static/7e22cc50bcc14a29ecdb348f88d69f7c/a81ca/CEF772D8-2FA9-4EA2-B291-5A2EB6F3AFC1.png)

We import the library moment from our helpers here to display relative time and also give a datetime value to the time tag for screenreader.

We also added a title for ur screenreader in our review\_\_stars div.

time.review\_\_time(datetime=review.created)= h.moment(review.created).fromNow()

## 39 - Advanced Aggregation

We are going to get a list of top 10 stores based on their ratings.

The issue we have reviews and stores in separate collections, so we need to aggregate. We want to grab a list of our stores, populate their rating and then find out what the average rating of that store is. We also don’t want to show stores that have one rating because that would bring them right to the top.

Create a route for the Top page in index.js

router.get(“/top”, catchErrors(storeController.getTopStores));

Add a method in storeController.js

//storeController.js

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

const store = await Store.getTopStores()

res.render("topStores", { stores, title: "⭐ Top Stores!" })

}

Instead of putting Store.getTopStores(); in the controller, we are going to put it in the Store.js schema so it is closer to the model because it’s a custom query.

storeSchema.statics.getTopStores = function() {

return this.aggregate([])

}

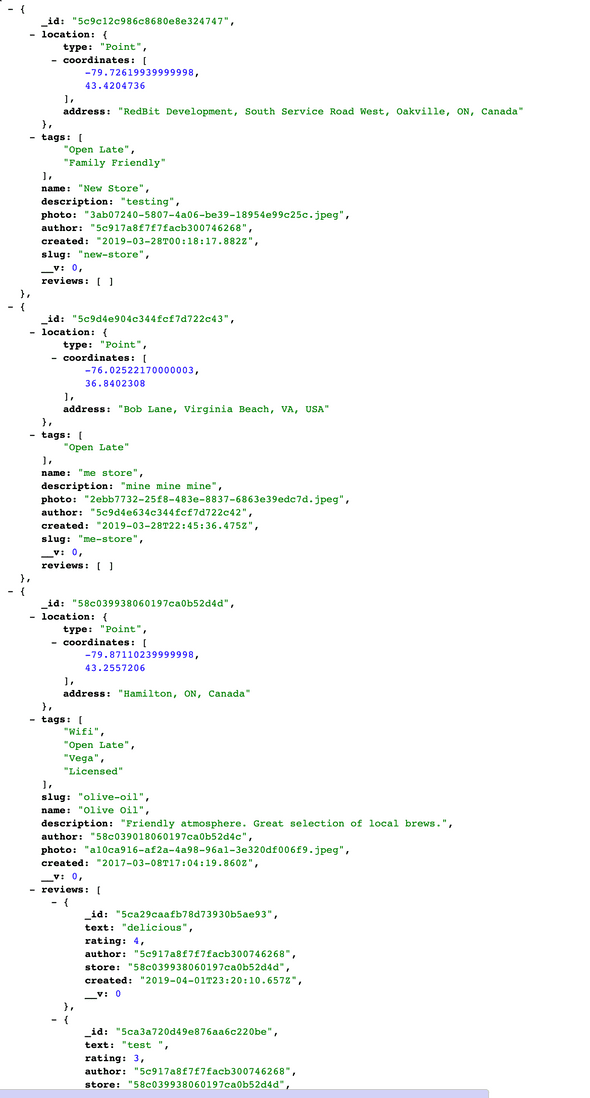
We use a proper function so we can use “this” which references the instance of the store model.

In the aggregate, we cannot use the virtual field because that is a mongoose thing but the aggregate goes right to mongodb.

{$lookup: { from: ‘reviews’, localField: ‘\_id’, foreignFIeld: ‘store’, as: ‘reviews’}}

The from: ‘reviews’ portion, mongo actually takes the model name and lowercases and adds an s. That is where we get that field name from.

The “as: reviews” portion is the name under which the reviews will show up. We could name this field whatever we want.

If that is all we have in the aggregate method and we return res.json(stores) from storeController.getTopStores() we will see the following:[](https://notes.anjagusev.com/static/b4d22be63efbac6b73194644691d59b0/08be5/1A5CCD5E-F534-48E5-9709-1957D4D3CB3D.png)

Next method in the aggregate is -> { $match: { “reviews.1”: { $exists: true } } } -> this looks at whether a second review item exists.

\*\* We take a break here to load sample data which we uncommented earlier . We uncomment the reviews that we commented earlier (4 lines in total) and then stop the node process and run the blowitallaway script instead $ npm run blowitallaway And then $ npm run sample \*\*

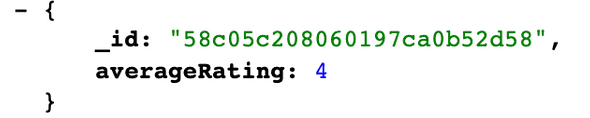
Next we want to add a field for the average rating using a project.

{ $project: {

averageRating: { $avg: ‘$reviews.rating’}

}}

This is saying create the new field using the average of each review’s rating.

However now all our returned data looks like this ->[](https://notes.anjagusev.com/static/5f13e9cca3ed1fc8e4806ab712782a52/b2a12/EF08E499-2F9A-45E0-8D1A-812380D5B45B.png)

But if instead of $project you used $addFields it would work, however that is only available in mongo 3.4 and above.

Otherwise we have to do ->

{

$project: {

photo: “$$ROOT.photo”,

name: “$$ROOT.name”,

reviews: “$$ROOT.reviews”,

averageRating: { $avg: “$reviews.rating” }

}

}

```e

$$ROOT is equal to the original document.

We want to sort by average rating from highest to lowest like so ->

` {$sort : { averageRating: -1}}`

Now we limit to 10 ` { $limit: 10 }`

Entire aggregation ->

```javascript

storeSchema.statics.getTopStores = function() {

return this.aggregate([

//Lookup Stores and populate their reviews

{

$lookup: {

from: “reviews”,

localField: “\_id”,

foreignField: “store”,

as: “reviews”

}

},

//filter for only items that have 2 or more reviews

{ $match: { “reviews.1”: { $exists: true } } },

//add the average reviews field

{

$project: {

photo: “$$ROOT.photo”,

name: “$$ROOT.name”,

reviews: “$$ROOT.reviews”,

slug: “$$ROOT.slug”,

averageRating: { $avg: “$reviews.rating” }

}

},

//sort it by our new field, highest reviews first

{ $sort: { averageRating: -1 } },

//limit to at most 10

{ $limit: 10 }

]);

};

Next create a new view called topStores.pug

//topStores.pug

extends layout

block content

.inner

h2 Top #{stores.length} Stores

table.table

thead

td photo

td ranking

td name

td reviews

td Average Rating

each store, I in stores

tr

td

a(href=`/store/${store.slug}`)

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

td #{i + 1}

td: a(href=`/store/${store.slug}`)= store.name

td= store.reviews.length

td #{Math.round(store.averageRating \* 10) / 10} / 5

The last thing we want to do is show the number of reviews on the store card.

If we go into the storecard and dump the store, the reviews will be null because we have not called populate on every single time that we want it. We need to look for getStores and find the find and populate the reviews.

const stores = await Store.find().populate(“reviews”);

However we do not want to have to do that every time so we can use our auto populate.

Make a function called autopopulate in Store.js

function autopopulate(next) {

this.populate(“reviews”);

next();

}

storeSchema.pre(“find”, autopopulate);

storeSchema.pre(“findOne”, autopopulate);

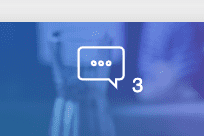
Next we add the reviews to the store card like so ->

if store.reviews

.store\_\_action.store\_\_action—count

!= h.icon(‘review’)

span= store.reviews.length

[](https://notes.anjagusev.com/static/28a7ee40581e23534288ba1f101f1693/9973b/FF133A76-C31D-4831-86F2-626482654E75.png)

## 40 - Implementing Pagination

Stores page is getting really long.

We need to go into the stores controller and modify the .getStores method to return a specific number of stores and then skip it.

We first need to create a new route like stores/page/2

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

//add the route below

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

In storeController.getStores, we need to know the page number which we can get from the url unless they are on the first page in which case we show them 1, and then we need a limit which will be 4 for us.

Now we need a visual pagination component.

We will make a mixin called pagination.

If there is more than one page, show the previous button

if page > 1

a(href=`/stores/page/${page -1}`) Prev

If we are not on the last page, show the next button

if page < pages

a(href=`stores/pages/${parseFloat(page) + 1}`)

Next

We have to do parseFloat because it things that the page is string not a number and “1” + 1 will give you 11.

//\_pagination.pug

mixin pagination(page, pages, count)

.pagination

.pagination\_\_prev

if page > 1

a(href=`/stores/page/${page - 1}`) Prev

.pagination\_\_text

p Page #{page} of ${pages} - ${count} total results

.pagination

if page < pages

a(href=`stores/pages/${parseFloat(page) + 1}`)

Next

We know which page we are on, but we don’t know the total number of pages nor the total number of records.

We need to go back to our store controller. Now that we are limiting, how can we actual know the number of items in our db?

We can use count! Instead of awaiting the store, we will be doing two queries and using promise.all to wait on both of them.

//storeController.getStores

const storesPromise = Store.find()

.skip(skip)

.limit(limit)

const countPromise = Store.count()

const [stores, count] = await Promise.all([storesPromise, countPromise])

Now we can figure out how many pages there will be.

const pages = Math.ceil(count / limit);

We use ceiling because if we had 17 store and the limit is 4, it would give us 4 and a decimal. We need to round that up using ceiling which is the upper limit.

One thing we need to guard against is someone going to /pages/20 and that page not existing. Right before we render out, we will put an if statement.

if (!stores.length && skip) {

req.flash(

"info",

`Hey! You asked for page ${page}. But that doesn't exist. So I put you on page ${pages}`

)

res.redirect(`/stores/page/${pages}`)

return

}

## 41- Deployment Setup

We want to make sure we have the following in our .gitignore file:

node\_modules/

.DS\_Store

\*.log

.idea

haters/

variables.env

variables.env.now

We need a production version of variables.env

We need to modify our package.json

We don’t need to watch or recompile on the server. We need to rename start to “dev”.

“scripts”: {

“prod”: “node ./start.js”,

“watch”: “nodemon ./start.js —ignore public/“,

“start”: “concurrently \”npm run watch\” \”npm run assets\” —names \”💻,📦\” —prefix name”,

“assets”: “webpack -w —display-max-modules 0”,

“sample”: “node ./data/load-sample-data.js”,

“blowitallaway”: “node ./data/load-sample-data.js —delete”,

“now”: “now -e DB\_USER=@db\_user -e DB\_PASS=@db\_pass -e NODE\_ENV=\”production\” -e PORT=80”

},

Change prod to start Change start to dev

## 42 - Deploying to Now

We are going to use Now at [Now - ZEIT](https://zeit.co/now)

We need to add values to our package.json for deployment to now.

Duplicate and rename variables.env to variables.env.now  
When we deploy, it won’t actually upload this file but in the package.json we tell it where to find those values and it copies those over to the server.

Signup for an account at Now/Zeit.

Install now globally on the terminal using command npm install now -g

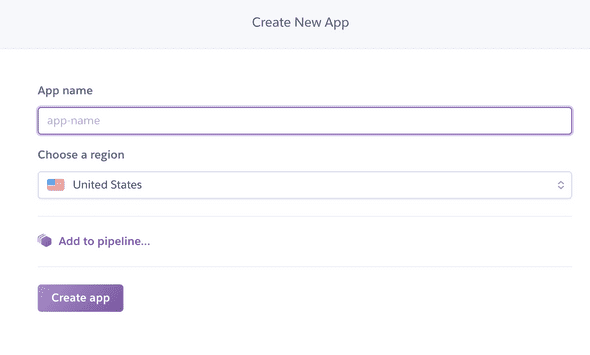
We have to add our email the first time and verify it. Once that is done, in the dang-thats-delicious directory we can simply type the command now and it will begin deploying it.

ISSUE: showing directory instead of loading project..

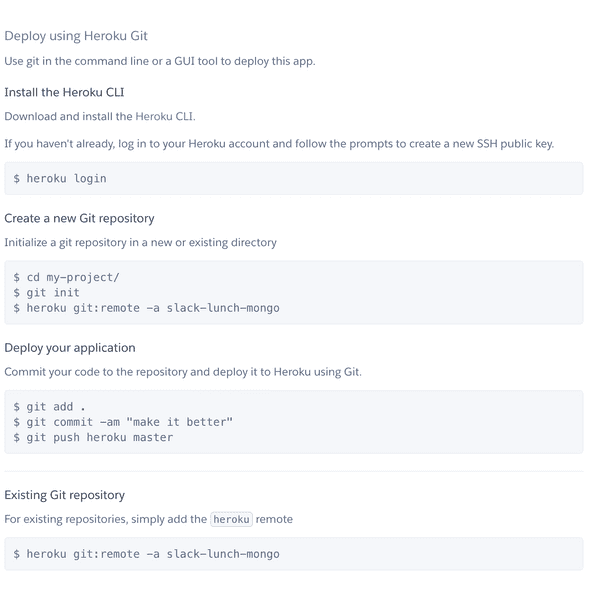
## 43 - Deploying to Heroku

Create new app on heroku.

Login, select Create New App.

[](https://notes.anjagusev.com/static/cdc14c1dd40c01466115ff4ffbd63abc/355b0/heroku-login.png)

This will give you these instructions:

[](https://notes.anjagusev.com/static/1053bac4f5e2104ed5d1fb1a12d326e0/87985/heroku-commands.png)

Download heroku cli Then run heroku login on the terminal.

Then in the directory on the terminal run: heroku git:remote -a now-thats-delicious-anja

Run : git push heroku master To push the code to heroku,

For the environmental variables in heroku we have to set them in our dashboard.

Go to settings -> Set Config Values