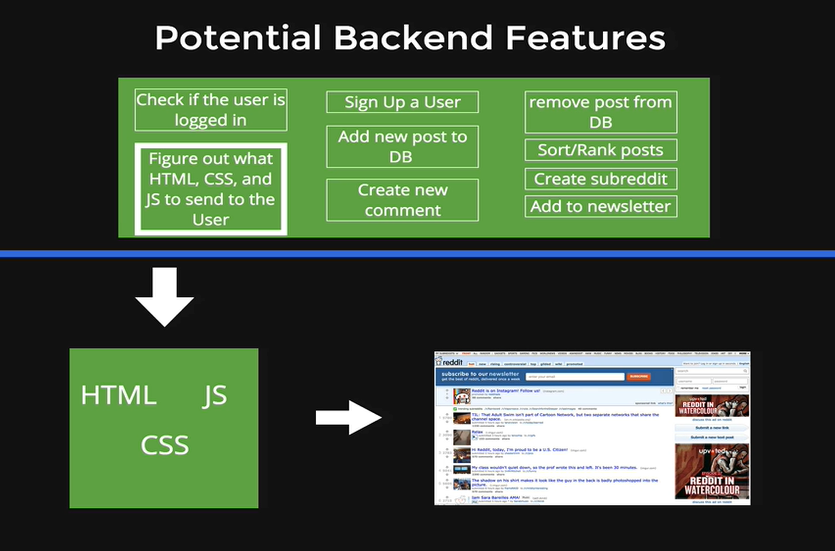
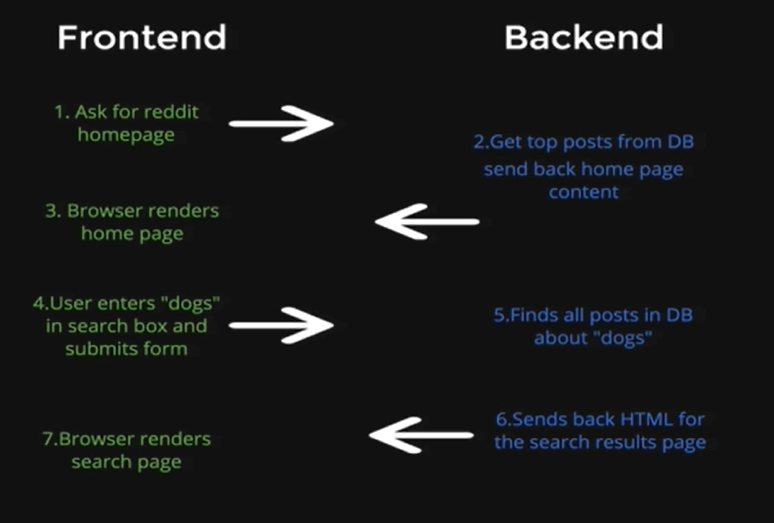
1. **Backend Intro**

* Udemy url -> IP address -> HTTP request sent to this IP address -> Udemy server figures out what we are asking for and builds the right content after pulling information from database -> Server responds with any combination of HTML, CSS, JS (HTTP Response) -> Our browser renders the page!
* Frontend and backend separated by blue line in the below image.

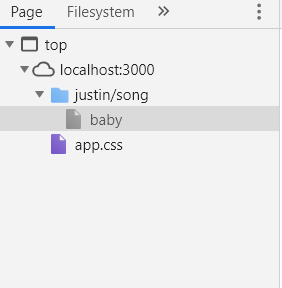


* Sequence of steps between front-end and backend. A request and response cycle!



* Static vs Dynamic
  + Static- Server sends same HTML, CSS, JS files every time we reload.
  + Dynamic- Server constructs the webpage before sending us.
* <https://stackshare.io/stacks>
* HTTP!
  + We can make HTTP requests from browser, terminal, Postman apps, from other apps.
  + HTTP request verbs-
    - GET: retrieve
    - POST: signing up, adding a comment, uploading etc. (Some payload in the request, to add data to database.)
    - PUT/PATCH: change (of course developer would be the one to change)
    - DELETE
  + If we never send a response like res.render(), res.redirect(), res.send(); our request will time out!
  + Postman
    - Body
    - Headers: metadata
    - Status: like 404 status code means not found. 200 means ok.
      * 404 error code means that the server can be reached, but that specific page does not exist inside that server.
    - Query string: Starts after ? and key-value pairs are separated by &.
      * Check Params in Postman. You can modify the query string from here.
    - For POST request, add data (key-value) in the request’s body.

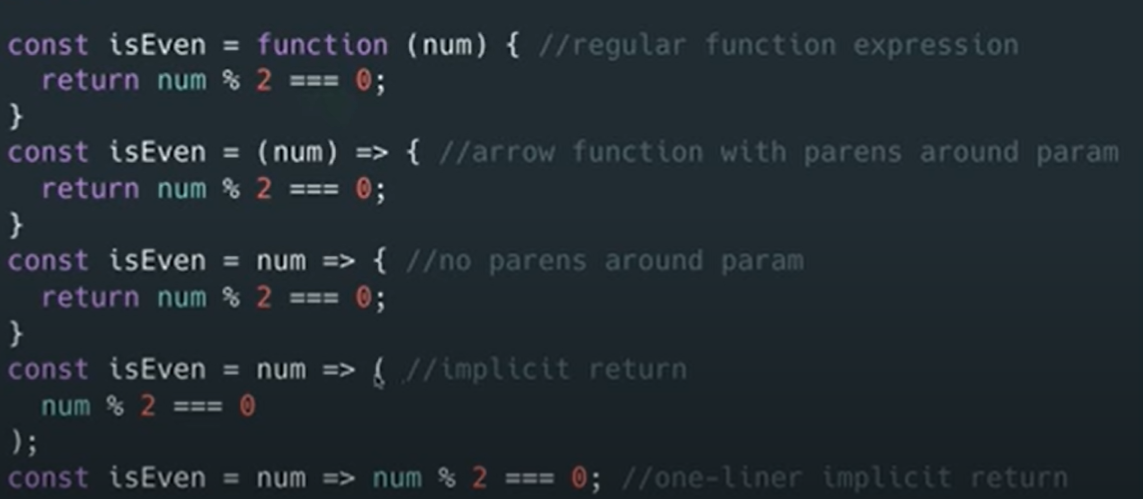
1. **Node JS**
   * Why learn Node? <https://stackshare.io/nodejs>
   * Use git bash in VS Code to be able to use commands like touch as in course.
   * Access node console (REPL) by typing node in the terminal. (Not going to use it much!)
   * alert() and some other methods are not available here because they come with the browser, not in the server side.
   * We can run JS files from terminal by- node filename.js
   * **NPM** <https://www.youtube.com/playlist?list=PLYxzS__5yYQmf-iF_9MTZmx7TxnmwnKIk>
     + Package manager for JS/node and even styling like Bootstrap (<https://www.npmjs.com/>).
     + All the packages are centralized here.
     + It has a command line tool to install the packages. So, no need of CDN now unlike in front-end. (It’s case-sensitive!)
     + node\_modules directory will be automatically created when we install a package for the first time. Rest all packages if installed will also reside inside it.
     + Use npm init to create package.json and install packages in the project directory (these packages won’t be accessible from other project directories). Otherwise it will be installed in parent directory like in my case, it is C:/Users/asus (packages installed here will be accessible from its child directories like C:/Users/asus/Desktop/Amazon/The Web Developer Bootcamp by Colt Steele/Codes/2-Backend/2-Express/2-PackageJsonDemo).
       1. Using –save can also we can download node\_modules in the same directory.
     + -g flag is used to install globally.
     + package.json file (contains dependencies of packages, licenses, authors and so on) is needed because npm’s work is not only to install packages, but also to maintain a record. It’s like a recipe for the ingredients needed for a particular package or application. So that one can install those dependencies when installing that package.
       1. Hence, when we uninstall a package, npm will also uninstall the dependencies of that package.
       2. When we try to install an installed package, it will not make copies.
       3. Inside node\_modules/packagename also we have package.json which contains the metadata for that particular package.
     + After package.json is created, typing npm install packagename will automatically add this package’s name and version under dependencies of the project inside package.json. (we no longer need –save flag for that)
     + We can install multiple packages at once by separating their names with spaces.
     + We can install all the dependencies mentioned in package.json file in a new system by running **npm install** (**Note:** Consider you have deleted the package-lock.json file).
       1. If a package’s version mentioned inside package.json is ^4.0.0, npm will install maximum version possible without breaking it (suppose 4.15.17). But developers don’t actually follow semantic versioning (so 4.15.17 may not be **backwards compatible** with 4.0.0) and that’s why package-lock.json comes in the picture.
     + NPM follows **semantic versioning (major.minor.patch)**.
       1. Major means breaking changes, and our code can be broken in a different version.
       2. Minor means an added functionality which we were not using.
          1. This is according to semantic versioning which npm considers, but npm can’t enforce developers to follow it. As developers don’t follow semantic versioning, there might be breaking changes in this case as well. This is the reason why we have package-lock.json file!
       3. Patch means bug fixes.
     + **package-lock.json** (for npm version >= 5)
       1. When changing system or environment, we want a consistent module behaviour.
       2. It was introduced because developers don’t follow semantic versioning.
       3. It is created when we install a package using npm.
       4. We can run **npm install** to install all the dependencies mentioned inside it on a new system.
       5. It not only contains details (exact version, url from where that package was fetched, integrity of the file) of the package installed by us like express, but also the details of dependencies of express.
       6. This way we will have the same dependency tree when shifting to a different system and we will never run into semantic versioning issue.
2. **Server-Side Frameworks**
   * Express is a web development framework.
   * Express is a lightweight framework.
     + Heavyweight vs Lightweight is about how much the framework does for you or how much you have to do yourself.
     + Heavyweight means it has a lot of text and only a few blanks for us to fill.
     + Unopinionated – flexible.
     + So, it’s better to start with Express as we will be able to understand the underlying concepts because we will have to understand how the things work in order to get things done.
   * Difference between library and a framework is Inversion of Control.
     + When you call a library, you are in control. Coz it’s a collection of functionality we can use in our code.
     + In case of framework, all the control flow is already in the framework and there’s just a bunch of predefined white spots that you can fill out with your code.
   * We can only use one res.send() inside a route’s code.
   * Only one of the routes satisfying the url pattern and appearing first will be executed.
   * Inside res.send(), we can also write HTML! (But a bad idea! Use EJS instead!)
   * Routes: Code that is responsible for listening/receiving requests and deciding what code to run based on requests, and what to send back.
   * First Express App!
     + console.log() will appear in the terminal.
     + res.send() will appear in the browser/postman.
     + Ctrl + c to stop the server.
     + Every time we make changes in the server side files, we will have to restart the server for the changes to reflect. This process can be automated using nodemon.
       1. We can directly write nodemon in the terminal if inside package.json, main is correctly defined (in my case, main is app.js).
       2. Otherwise, if package.json doesn’t exist or main is wrongly defined (but the main should be correct), we can write nodemon app.js to automate the process.
   * Routes
     + \* catches all the routes. So, use it at the end as order of routes in \*.js matters.
     + Route params: For pattern matching. When we use :, it won’t care whether it’s a number of alphabet or whatever except /. It will only match pattern where chunks of characters are divided by /.
   * While creating a new node project with npm init, be sure not to name the project (or the folder that the project lives inside of) the same as any of the node packages being used in the project itself.
   * EJS (Embedded JavaScript) template (There are many other templates available with similar core functionality)
     + Whenever we render a file, it automatically looks inside the views directory. So, put the \*.ejs file inside the views directory.
     + Use special tags inside EJS file to treat it as JavaScript and pass values in an object. Outside will be considered as HTML.
     + Those special tags are
       1. <%= %> (Return value will be added in HTML)
       2. <% %>
       3. <%- %>
   * Put your front-end assets like .css and .js files inside a folder named public (can name something else as well, but public is conventional). **Express doesn’t serve anything other than views directory automatically. So, in app.js, specify public directory so that Express serves our public assets.**
     + We can also use style tag inside \*.ejs file, but it’s a bad idea.
   * Better use partials for including HTML boilerplate and even for linking css files (because DRY!). Inside partials folder (can be named something else but it is common), we will put the header.ejs and footer.ejs files (can be named something else but header and footer are the standard names).
     + We are not limited to header and footer (they are more common). Can also make partials for navbar or something else!
   * **IMPORTANT**! Regarding path of stylesheet. (Lecture: Serving Custom Assets)
     + See picture below for <http://localhost:3000/justin/song/baby>



* In the sources of page rendered-
  + baby template contains the html code. (Locally, I had song.ejs file, not baby template. So .ejs file names don’t have to determine the name of actual generated HTML code that is served back to the browser. The name of the served page is the last segment of rout path. That is how Express.js framework serves the rendered EJS view template.).
  + You will find \*.css file is in the root directory as it was inside public directory in our backend Node.js/Express.js application structure (if it were inside public/stylesheets, then here it would be inside stylesheets directory).
    - We can have both CSS and JS files inside public directory, hence we can make stylesheets and scripts directory inside it and then write paths this way:
      * href=”/stylesheets/app.css” and src=”/scripts/main.js”.
    - If inside root, then when using the link tag inside style tag of.ejs or using partials, use href=“/app.css” and not “app.css”.
    - As “app.css” will look for the css file inside the same directory where baby template is present but “/app.css” will look for “app.css” inside the root directory (because of the “/” in the beginning!) which is actually where app.css is present.
* In the below mentioned case “app.css” will work perfectly (unlike the above mentioned case).
  + If the url had been <http://localhost:3000/songs>**,** we will have songs template containing HTML and app.css both in the root directory.
  + If the url had been <http://localhost:3000>, we will have (index) template and app.css both in the root directory.
* The public folder and the views folder are part of your backend Node.js/Express.js application structure, which isn't directly reflected on the files that get sent/served to the browser in the frontend (the frontend browser doesn't get direct access to your backend app files, only what the node app sends as a response to it, like the generated HTML pages or static files like .css files). Hence source files are not same as our application folder structure.
* **IMPORTANT**
  + The **EJS include statements are evaluated in the backend**, when the view is being rendered - that's why we use relative paths inside the views folder for the EJS include statements, based on your backend project file/folder structure there.
  + However, **public assets like CSS files are accessed in the frontend**, **when the EJS view is generated and served to the visitor browser.** Basically, **we serve the public folder in the root of the response to the user browser, and therefore we need to link static assets with a starting forward slash.**
* POST request
  + We can send a POST request using Postman or a form.
  + Action attribute’s value should match with the route where we want to send the POST request.
  + Use the name attribute in the input tag of the form, to retrieve the information inserted by the user. This information will be present in the body of request made to that route.
  + Install body-parser so that request’s body is converted to JS object called req.body for us to use. (Every time we have to extract data from forms using POST request, we need it)
    - If req.body is returning an empty object, then in your browser , use **Ctrl+Shift+R** for a full page reload **without cache**.
  + Since we are not using a database yet, as we close the server, the inserted data will be lost!

1. **APIs**

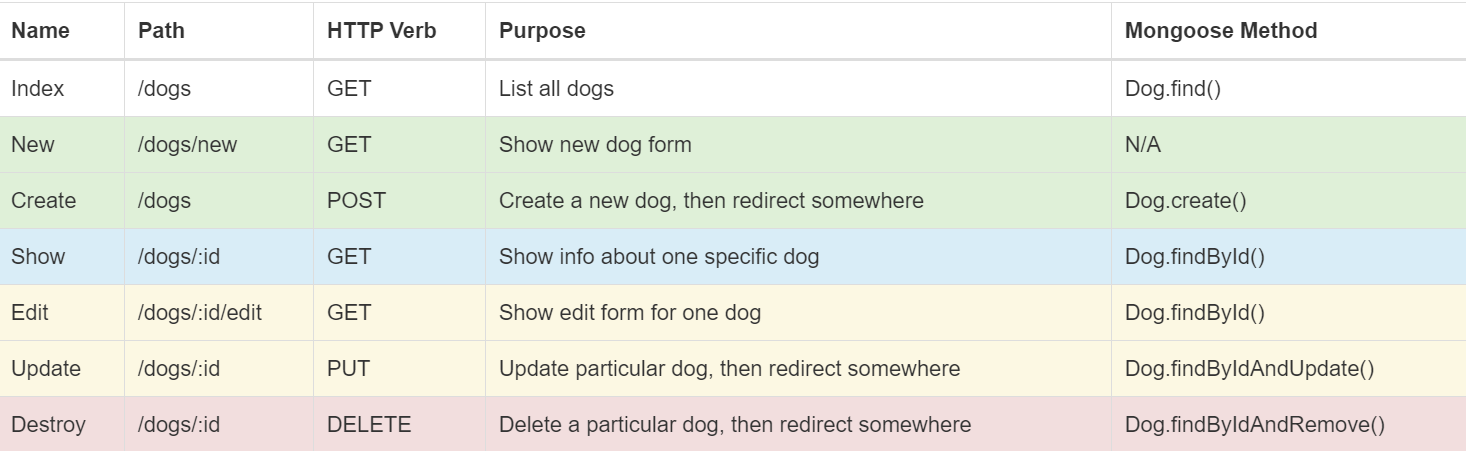
* Application Programming Interface.
* Interfaces for code/computers/servers to talk to one another.
* For connecting with other applications.
* <https://ifttt.com/explore>
* <https://www.programmableweb.com/>
* Internet of Things: When we control a physical object using API, like Tesla car using its API.
* Web APIs are more common (generally communicate via HTTP) like Twitter, Facebook, Weather API.
  + 2 main formats they respond with
    - JSON (JS Object Notation)
      * Which is returned as a string not a JS object and we need to parse it using JSON.parse(body) to convert it to JSON.
    - XML
  + JSONView Chrome Extension to automatically format the JSON data.
* **Deprecation** just means that it will no longer receive updates and could potentially break in the future.
* Installing locus: *npm i -D locus*
  + Here -D stands for development dependency. This means this package locus is only required for development or testing. Hence locus will be added to the devDependencies list. These development dependencies are unneeded in production for example testing packages like locus.
  + When we go in production, and type *npm install* and the dependencies mentioned in package.json file will be installed. As npm assumes they are development deploy. We can instead write *npm install –production* to avoid installing those development dependencies.
* We will learn how to make request from code using a package called request (like we have been doing from browsers and postman)! Our code will get information from an API through a request.
* We can also make requests from command line using a tool called curl (not needed because we will be using APIs inside an application).
* Some APIs require an API Key (they want us to identify ourselves for access to their data). This way they can track how often we are accessing the data and what are we using it for. And if we build an application using that API with many users, we may have to pay.
* Latest syntaxes introduced in ES6: template literals, arrow functions, promises.
  + Arrow functions are syntactically compact alternative to a regular function expression. They are like anonymous functions and we can store them in a variable.



* + Implicit return works only for single expressions.
  + No separate this.
* Movie Search App Project
  + We didn’t need body-parser as we are extracting from request’s query and not request’s body coz it’s a GET request. (Note: It’s different from route params)
  + We are using a GET request as the user is searching for a particular movie, not appending it in the database. Though using POST request also, we could get the same functionalities in the App.
  + *POST request is more secure as in case of sensitive data, it won’t get appended to the URL unlike in case of GET request. (HTML forms only support GET and POST. So it defaults PUT as GET and hence appends the data entered in form to the URL.)*
* YelpCamp version-1
  + Don’t put a preceding slash in the path here as we are not checking in the root.
    - <%- include("partials/header") %>
  + In EJS, the retrieved the URL of campground image which was a string, but the string doesn’t have quotes around, hence we are writing it inside quotes. Think of it as when we print a string in console, the quotes are not displayed.
    - <img src="<%= campgrounds.image %>">
  + app.get(“/campgrounds”, ); and app.post(“/campgrounds”, ); are totally different routes because of different request methods, though they have the same URL. (A convention called REST for how we name our routes)
  + When we redirect, it’s a GET request by default.
  + We used header and nav tags instead of div in campgrounds.ejs because it has a semantic meaning, though both are functionally same.
  + Due to different image sizes, our grid of images is messed up. Colt fixed it by-

style="display:flex; flex-wrap: wrap;"

1. **Databases**
   * A collection of information/data and has an interface to interact with it.
   * SQL (relational) vs NoSQL (non-relational) databases.
     + NoSQL are much more flexible.
   * Setup guide: <https://zarkom.net/blogs/windows-local-coding-environment-1204>
     + To be able to run the mongod.exe and mongo.exe from any location in PowerShell, we will have to add the path in environment variables. We won’t have to navigate to that MongoDB folder every time to run these files.
     + mongod.exe will start the MongoDB server. Ctrl+C to close it.
     + Keep the server running and, in another PowerShell, and run the Mongo shell which will connect to the MongoDB server. So now we can try MongoDB shell commands and access your databases directly.
     + 27017 is MongoDB's default port on which it runs.
     + Our Nodejs application will also be able to connect to the local MongoDB server if it’s running.
   * Mongo Shell Basics
     + Shells are generally used for learning purpose, testing and debugging.
     + For the applications to be self-contained, we will make a different database for each of our applications (which is not necessary).
     + ObjectId is a hexadecimal number which is automatically assigned by Mongo which is unique.
     + CRUD are the four basic functions it provides.
     + Mongo Commands
       1. mongod - server
       2. mongo - shell
       3. help - basic commands list
       4. show dbs
       5. use
       6. insert
       7. find
       8. update
       9. remove
   * Mongoose
     + It’s like a JS layer on top of MongoDB.
     + Makes it easier and cleaner for us to interact with the MongoDB inside our JavaScript files. This can also be done without mongoose hence it’s not necessary.
     + Can be installed using npm.
     + We pass a callback function in every method to make sure it is completed successfully or not. Errors can be caused due to no internet if we are using an online database.
   * <https://www.udemy.com/course/the-web-developer-bootcamp/learn/lecture/9037146#questions/12692644>
     + Node.js is asynchronous. That means it starts the execution of a method that takes time to fully execute but won't wait for it to finish, so it fires off next lines of code, and when the asynchronous method finishes it fires the return result also. In the meantime, other, faster code could have executed too, thus the order gets changed.
     + Mongoose methods like .create(), .find() or .save() are asynchronous queries, since we first execute the request to get something from the database which takes some time to return to our node application, so it doesn’t make the application wait for it to finish.
     + The reason we use callback functions or promises is that their execution will actually start only after the asynchronous function finishes and returns the result. (**Note:** Callback functions are common in several languages, not only JS. However, not all callbacks are guaranteed to be executed after the outer function as it in in Node.js).
     + Database calls are fired in the order of how the code is written, but with asynchronous queries (like database calls) one can finish before the other (regardless of the line order), so the one that first finishes (the faster one) will be shown in the terminal first. They can have different speeds and produce different results every time you run it, on different computers (different hardware and software versions), etc. - so because it's not consistent you may get different results compared to Colt.
     + In simple words:
       1. Think of it this way… We ask the program to do something, but that something may take an arbitrary amount of time before it finishes. So, in the meantime, we can have the program do something else.
       2. But, some of your other operations may depend on the original something to have finished running in order for them to run/work properly. Say, for instance, we need a user to be created, before we can create a post and assign it to the user as its author.
       3. So, that's where callbacks, promises, and promises with async+await come into play.
   * YelpCamp version-2
     + Making data persistent by using Mongoose.
     + RESTful Routes: <https://medium.com/adventures-in-code/snow-white-the-7-restful-routes-afcf87bbe5bd>
       1. There are 7 of them.
       2. Architecture standards that so many applications follow and it makes predictable and reliable for us to interact with those applications (like a RESTful API).
2. **RESTful Routing**
   * A convention or architecture for mapping between HTTP routes and CRUD.
   * Use it when making a CRUD app.
   * 7 RESTful routes-
     + Same colours means they are linked because they correspond to same task.



1. PUT request means updating something. We can achieve the same functionality using POST request, but it doesn’t have that semantic meaning as PUT.
2. RESTful Blog App
   * Inserting a default date in DB which is when the blog was created.
     + We converted it to a more readable format for users using JS methods like toDateString(). There are JS libraries like Moment.js which are more flexible with formatting options.
   * We will be using Semantic UI for styling (Its classes are more meaningful than bootstrap).
     + In this, we have the option to include styling for selective components like forms unlike bootstrap where we had to include the entire library.
     + Here, navbar is called a menu.
   * Trick! (Note: It’s specific to body-parser)
     + <input type="text" name="blog[title]">
     + <input type="text" name="blog[image]">
     + In name attribute, if we use the above values, then title and body can be accessed by req.body.blog.title and req.body.blog.image. And req.body.blog is the object. Then we can also directly pass this object to suppose Blog.create().
   * Traversing folders
     + / means go back to the root folder, then traverse forward/downward.
     + ./ means begin in the folder we are currently in (current working directory) and traverse forward/downward in the tree.
     + ../ means go up one directory, then begin the traverse.
   * It’s not safe to let users write HTML as input as they can enter a script tag and run JS!
     + In this app, users cannot style the content of their blog. Like if there is a paragraph change. Or if they want to bold some part. Hence in EJS file, we can change <%= %> to <%- %>. In Node version of EJS, <%= escapes the HTML going into the buffer, while <%- does not.
     + We can sanitize the blog body which removes the script tags.
       - npm install express-sanitizer
       - We can write a middleware which would run this sanitizing part at in CREATE and UPDATE routes. So that we won’t have to repeat our code.
   * We limited blog content to 100 characters in the index page by using a JS method substring(0, 100).
   * HTML forms only support GET and POST requests!! So, it defaults PUT as GET request. Hence the data entered in the form is appended in the URL. This is why Method-Override comes in the picture!
     + npm install method-override.
     + Use \_method=PUT query string.
     + *For method-override to work, make sure method=”POST” in the HTML form.*
3. **Data Associations**
   * Relationships
     + one:one (one entity related to one entity)
     + one:many (one user uploads many photos on Facebook)
     + many:many (A student can sign-up for multiple courses and each course has multiple students enrolled.)
   * Ways of associating data
     + Embedding data
     + Referencing data
   * So many nested callbacks is termed as a callback hell! We can deal with it using promises.
   * In some cases, we need to chain some methods first, like populate (when using referencing data, we used it to stick the actual post inside the user instead of displaying the object id of posts), and only then perform the normal flow of the method, like calling the callback function, and exec helps to glue all that together.
   * module.exports to clean-up our code and make it more modular and reusable by making a separate file containing schema and model. Then we can require the files at top. Also, this way, multiple files can use this code hence no need to duplicate the code.
   * YelpCamp version-3
     + Refactored the app.js by making it modular.
     + Tackling comments.
     + seeds.js to save time for fake data removal and creation in one go.
   * YelpCamp version-4
     + Nested routes for adding comments in a particular Campground. As */comments/new* and */comments* are not associated with a particular Campground. Nested routes are-
       1. NEW ROUTE: /campgrounds/:id/comments/new
       2. CREATE ROUTE: /campgrounds/:id/comments
     + Created separate directories inside views for campgrounds and comments. This way the same file names won’t clash and will also look cleaner and understandable.
     + New comment form.
   * YelpCamp version-5
     + Added sidebar in show page.
     + Displayed comments nicely.
     + img-responsive class in Bootstrap so that image scales according to its parent object (observe by shrinking the browser horizontally).
     + A slightly better way using \_\_dirname. \_\_dirname refers to the directory where this script is run in.

app.use(express.static(\_\_dirname + "/public"));

1. **Authentication**
   * We will use Passport.js which is an authentication middleware that can help us easily add authentication without writing so much code. <http://www.passportjs.org/>
   * There can be different strategies of authentication like using email and password or though a service like Facebook or Twitter or so on.
     + We will be using the strategy: Passport Local which is for username and password.
     + We will also use Passport Local Mongoose which will help us make it even faster and simpler. We could have done without it as well.
   * HTTP is a stateless protocol. It doesn’t store information about past requests. So how to implement if a user is still logged in and allow him access to certain pages? By using sessions!
   * AuthDemo
     + Putting the button at the end of the form will ensure it submits. Else use input with type submit.
     + Password is hashed and then stored.
     + Salt is also stored in the DB which will help to unhash the password.
     + A glimpse of middleware!
       1. This is standard for middleware.
          1. req: req object.
          2. res: response object.
          3. next: next thing written after this middleware that will be called. Might be the next middleware or a callback function.

(req, res, next)

* + - 1. We can stack multiple middlewares between the start and end handler of the route. These middleware will be executed in the order they are defined.
      2. If someone tries registering using the same username which is already registered, we will get the error in console because it was handled by Passport Local Mongoose.
  + YelpCamp version-6
    - req.user returns the username and id of currently logged in user.
    - Colt created a middleware using app.use() (it is a middleware method) so that we won’t have to pass the object containing details of logged in users in every template. Hence it dried up our code.
      1. He used res.locals inside which we can put everything we want to be available in the templates.
    - When using if else in EJS, you can start else on a separate line. You have to start it in the same line where else ended. Otherwise it would throw error.
  + YelpCamp version-7
    - Refactoring routes.
    - index filename is used for all purpose routes. Like for landing page and authentication in this case.
    - When breaking down our code into multiple files, we might need to require packages more often.
      1. Not requiring models is still working fine. See here: [https://www.udemy.com/course/the-web-developer-bootcamp/learn/lecture/3861702#questions/7883004](https://www.udemy.com/course/the-web-developer-bootcamp/learn/lecture/3861702%23questions/7883004)

Hence, we can skip requiring the Comment, User, Campground models in other js files if they are already required in app.js. But as mentioned in the thread, it’s a good practice to require in the file which actually needs.

* Colt refactored the routes declaration using express router by finding out the common prefixes in the routes of a particular file, passing it inside corresponding route declaration, and removing that common prefix from the routes in that file.
  + So, our “/campgrounds” became “/”.
  + “/campgrounds/new” became “/new”.
  + And so on.
* You must pass {mergeParams: true} to the child router if you want to access the params from the parent router. Can find it in comments.js.
  + YelpCamp version-8
    - Each newly added comment will automatically store author name and id using referencing. So, we changed the comments.js accordingly and removed all the campgrounds, and commented seedDB() in app.js.
    - Prevent an unauthenticated user from creating the campground.
  + YelpCamp version-9
    - Store username and id to newly created campground.
  + YelpCamp version-10
    - Update, Edit and Destroy campground functionality for the user who uploaded the campground (requires authorization). This would complete our CRUD app.