Node JS Cheat Sheet

|  |  |  |
| --- | --- | --- |
| Serial -Video Number | Topic/Code | Explanation |
|  | Different command line codes | *Touch*  *Mkdir*  *Rm*  *Rm -rf (*rf is a flag = recursive force*) =* Removing entire directory  *Node -v*  *Npm -v*  *Cd*  *Cd ..* |
|  | Running a node file | Node fileName.ext |
|  | NPM | Node package manager  Helps us install a lot of packages. |
|  | Npm install | Npm install package name    This is used to install a package using NPM.  Whenever we install a package there would be a folder called node\_modules. Its contains the files of that package.  When we want to use that package we just create a variable and store it inside it. EX:  Var something = require(“package name”); |
|  | Npm init | Creates package.json |
|  | Express | Npm install -express  Require(“express”)  Now express has a lot of methods so just calling express(); wouldn’t do we call this method and store it inside a variable.  Var app = express(); |
| 1. 268 | Creating routes | app.get("/", function(req,res){  })  Req is for request. What information was it requested with.  Res is for respond. With what information re you going to respond with.  These are objects  This code is for creating routes.  app.listen(port,"ip address”, function(){  })  we have to tell it to listen for particular request |
| 1. 269 | Package.json | Contains all the meta data of that corresponding package. Includes authors, tags and dependencies that are related to the project. |
| 1. 269 | --save | Adds dependencies.EX:  Npm install express -save OR npm I -S express |
| 1. 270 | Keeping your servers on all the time | Installing: npm I -g nodemon ()  Then nodemon app name |
| 1. 271 | For other undefined routes | We use \*.  app.get("\*",function(req,res){  res.send("doesn’t exist");  })  REMEMBER ORDER MATTERS HERE. SHOULD ALWAYS BE IN THE END. |
|  | /:name | Match with anything that comes after /  Get access to the value that was replaced with a user given value (r/:name => r/puppy) via req.params this would return an object with attributes and values so for the given example we would get  {  Name: “puppy”  }  Directly getting the value req.params.name |
| 1. 275 | Res.render(“filename”, {object Type so key value pair}) | Renders HTML file/EJS (embedded JS) file  Has to be inside views directory. For ejs files have to have that pacakage.  EX:  Res.render(“friends”, {  Friends(this is the variable that would be refered to inside the EJS file): friends(The value of that variable in this case friends is an array so were passing the value of friends)  })  In our EJS file we need to put value here to make the website dynamic. And we can have access to passed values through <%= TREATED AS JS %>  We have to wrap every line of JS with the tags.  <%= %> vs <% %>  1st one: The value would be calculated and return the result to the html  2nd one: for logics and loops etc |
| 1. 277 | Partials | Partials are templates we can use inside other templates. Has to be inside views/partials/fileName.  Adding partials  <% include(“partials/file”)%>  Remember to put / Infront of the CSS file so that it understands that the file is not inside the views directory rather look somewhere else. |
| 1. 278 | Use other directories | For using other directories such as for CSS files we use public directory (EX) we have to tell express to use that directory and serve the contents. |
|  | View engine | app.set(“view engine”, “ejs”); is basically telling express that the files that we would be rendering would always be ejs files so that we don’t have to write EJS as an extension every time we put a new file in render. |
| 1. 279 | App.post(“path”) | For posting data (ex; forms) |
| 1. 280 | Getting values from a form | To get values from a form we need to use req.body however this wouldn’t work as express doesn’t support this so we need to install another package. Called body-parser which returns an object  Npm install body-parser –save  var bodyParser = require("body-parser");  app.use(bodyParser.urlencoded({extended: true})); |
| 1. 280 | redirect | res.redirect("/pathname");  redirects you to another defined path  the default is a get request |
| 1. 281 | API | Application programming interface  It basically helps you to connect with other applications or hardwares.  Resources: IFFTT, programmable Web |
| 1. 283 | Making requests | var request = require("request");  request(‘link’,function(error,response,body){  })  Ex:  var request = require("request");  request('http://www.google.com',function(error,response,body){      //error handles the error      if(error){          console.log("Something went wring: " + error);      }      else if(response.statusCode == 200){          //recieved successfully          console.log(body);      }  })  //Also we have another way of doing it (ES6 version)  //request as of now is depreicated instead we have request promise  //to install it we need to do the following  //npm i -S request-promise  const rp = require("request-promise");  rp('https://jsonplaceholder.typicode.com/users/1')  .then((body)=> { //these are called promises this part is basically the response      const parsedData = JSON.parse(body);      console.log(parsedData.name + " Lives In " + parsedData.address.city);  })  .catch((err)=> {      console.log("Error ", err );  }); |
| 1. 285 | Working with API | The response that comes in the body is always a string. So when we want to get information given by an api we need to turn it into an object and to do that we need to do the following.  Var parsedData = JSON.parse(body);  To get access to an object value.  parsedData[“ ”][“ ”] or  parsedData.ObjectName.ObjectName or  ‘$(parsedData.ObjectName.ObjectName) lives in parsedData.ObjectName.ObjectName’ //here $() this is basically making it a string and returning the dynamic value. |
| 1. 287 | Locus | Locus is a trouble shooting tool package.  Npm i -D locus (here d stands for development tool)  Using locus: eval(require(‘locus’))  Where ever this code is written it would have access to all the variables uptill that part.  For example in the console we can use response.statusCode to get the value. |
| MONGO DB (ODM) | | |
| 1. 303 | Mongod | Runs the mongo demon/process |
| 1. 303 | Shows dbs | Shows all databases |
| 1. 303 | Use dbName | Creates and switches to that db (if created just switches) |
| 1. 303 | Db(refers to the current db).collectionName.insert( {object} ) | For inserting into a collection |
| 1. 303 | Show collection | Shows all the collections in that database |
| 1. 303 | Db.collectionName.find() | Shows all the content inside.  Db.collectionName.find({name: “value”}) => finds corresponding item. |
| 1. 304 (notes) | Db.collection.updateOne({Attribute},{newAttribuute})  OR  Db.collection.updateMany({Attribute},{newAttribuute}) | What this does is it finds the object and replaces the whole thing with the new value. IF we want to preserve the value.  Db.collection.update({Attribute},{$set: {attributes}})  Ex:  Db.dogs.update({name: “puffy”},{$set: {breed: “puddle”, cute: false } } ) |
| 1. 304 (notes) | Db.collection.deleteOne({attribute})  OR  Db.collection.deleteMany({attribute}) | Specify with attribute. |
| 1. 305 | Mongoose | Package to use mogoDB.  Npm install mongoose  Var mongoose = require(“mongoose”)  Mongoose.connect(“mongodb://localhost/dbName”) //this creates DB if not created |
| 1. 305 | Creating schema | NOT DEFINING A STRUCTURE but creating a pattern. This is easy for us to code.  Var schemaName = new mongoose.Schema( {  attributes  } );  //compiling it into a model/pattern  Var varName = mongoose.model(“varName(singular version of the collection so cat to cats)”, catSchema);  varName. methods |
| 1. 305 | Adding values | EX:  Var puffy = new cat({  Name:”puffy”,  Age:”5”  });  Puffy.save(function(err, cat){  If(err){  Console.log();  }else{  Console.log(“saved”);  }  }); |
| 1. 306 | Create and save | Creates and saves at the same time.  EX: cat.create({  Name:”puffy”,  Age:”5”  }) |
| 1. 309 | Droping collection | Db.collection.drop() |
| 1. 310 | Find using an id | EX: campground.findById(id. Function(){  }) |
| 1. 312 | Adding default values to a attribute | EX: created :{type: Date, default: Date.now} |
| RESTFUL Routing | | |
| 1. 309 & 311\*\* | Restful routing | There are 7 of them.   |  |  |  |  | | --- | --- | --- | --- | | Name | URL | HTTP Verb | Description | | Index | /campground | GET | Shows the list of campgrounds | | New | /campground/new | GET | Displays the form to new camp | | Create | /campground | POST | The form submits the data to /campground as a POST request. | | Show | /campground/:id | GET | Shows info about one campground | | Edit | /campgrounds/:id/edit | GET | Editing one campground (FORM) | | Update | /campgrouds/:id | PUT | The form submits here.  Update a particular camp. | | Destroy | /campground/:id | DELETE | Delete a particular campground. | |
| 1. 312 | CSS public directory | app.use(express.static("public")); |
| 1. 315 | New way of submitting data | <input type="text" name="blog[title]">  What this will do is, instead of making the value for title available directly from req.body.title it will put it inside of an object, like so: req.body.blog.title  Now all of the values from the inputs in the form get added into one object (req.body.blog) and you can easily input that data into the database simply by passing in req.body.blog to Blog.create()  IAN’s comment:  In this instance you cannot use dot notation over bracket notation. If you used blog.title, then in your back-end code you'd have to access that value separately using req.body['blog.title'], which wouldn't allow you to access all of the values of the items inside of the object simultaneously with something like req.body.blog  In the form, each input has a value (once the user has typed something in), then on submit of the form, all that data gets sent to the back-end (to be put into the database in this case). We could just set each input to it's own variable, so if you used name then the value for that input would be accessible in the back-end via req.body.name, but this is cumbersome so we short circuit the whole process by assigning each input as a key (pointing to the input value) within one object (e.g., blog). Body-parser is the middleware that allows us to do this and unless we use bracket notation in the form, then it will not recognize the input's value as being a part of an object, able to be referenced by a key. |
| 1. 317 | Making a field mandatory in semantic | Add required to the input |
| 1. 317 | File traversal | / 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. |
| 1. 318 | JS date manipulation | .toDateString()  Makes the date human readable. |
| 1. 318 | How to write HTML inside a text body(form) | Use <%- blog.body %> instead of <%=blog.body%>    What this will do is it would rather than substituting the value there it would evaluate the content inside. So run HTML if there are any inside. |
| 1. 318 | Show small amount of content | JS code  .substring(0,100) |
| 1. 319 | Why PUT doesn’t work | PUT request is basically a get request because HTML forms don’t support put request they only support GET and POST. So when we say method=”POST” then the it actually goes to the GET route with all the values inside the form in the URL. So how can we use PUT as it is still an HTTP request that updates values. So for that we need to do the following.ADD the following code at the end of the url in the action attribute of the form.  **?\_method=”PUT”**  What this is doing is that it sends as a get request however because of that additional line it treats as PUT. (METHOD OVERRIDE)  EX:   <form action="/blogs/<%=blog.\_id%>?\_method=PUT" method="POST" class="ui form">  This itself doesn’t do anything so we need to install the method-override package.  Npm i -S method-override  var methodOverride  = require("method-override");  app.use(methodOverride("\_method"));  So what this is saying that whenever in the url you see method you make it a put request.  ALL OF THIS IS BECAUSE OF RESTful ROUTING |
| 1. 319 | Updating items in one go | EX:   blog.findByIdAndUpdate(id to find, new data, function(err, updatedBlog){        }) |
| 1. 320 | Delete Request | As we are following the RESTful convention we have to use this,  app.delete("/blogs/:id", function(req,res){  <form action="/blogs/<%=blog.\_id%>?\_method=DELETE" method="POST" class="ui form">  otherwise we could have easily done  app.get("/deleteBlogs/:id", function(req,res){ |
| 1. 322 | Sanitizer | Sanitizes inputs of all sorts of script tags.  Npm i -S express-sanitizer  var expressSanitizer= require("express-sanitizer");  //this should always be after body parser  app.use(expressSanitizer());  req.sanitize(content);  ex:  req.body.blog.body =  req.sanitize(req.body.blog.body); |
| 1. 324 | Embedding Data | To have a one to many or many to many relationship we need to update our schema to handle it. For example, a person can have multiple post so his schema should have an array that can hold references to all his posts. To do that we need to have a post attribute in the schema that would hold post content. So,  posts:[postSchema] //Has to be a schema of the content |
| 1. 326 | Object Reference | Rather than putting the corresponding post in one array. So an array would have something like this,  Post =[  {  Title:”asdasdas”,  Author:”asasdasd”  },  {},{} …  ]  We can put references to these post in the array.So it would look like this,  Post =[  {  Id:5656535  },  {},{} …  ]  In side a post we have  { id: 5656535  Title:”asdasdas”,  Author:”asasdasd”  }  To do this we have to change the post attribute in the schema.  Posts:[  {  Type:mongoose.Schema.Types.ObjectId,  ref:”model variable name (Post)”  }  ]  So what this is saying is that the post array holds references to id of a post object. |
| 1. 327 | Modularizing the code | Put diffferent parts in different sections, import things that are required then at the end put  module.exports = send the file you want to be exported.  Var varName = require(“./models/post”)  This is basically the return statement. |
| 1. 326 | Save vs update | The main difference is that with .save() you already have an object in your client side code or had to retrieve the data from the server before you are writing it back, and you are writing back the whole thing.  On the other hand .update() does not require the data to be loaded to the client from the server. All of the interaction happens server side without retrieving to the client.So .update() can be very efficient in this way when you are adding content to existing documents.  In addition, there is the multi parameter to .update() that allows the actions to be performed on more than one document that matches the query condition.  There are some things in convenience methods that you lose when using .update() as a call, but the benefits for certain operations is the "trade-off" you have to bear. For more information on this, and the options available, see the documentation.  <https://stackoverflow.com/questions/22278761/mongoose-difference-between-save-and-using-update/22278847>  save doesn’t take parameters. EX:  found. |
| 1. 332 | Important thing to remember while using object references in data associations. | When we use data association we put object ids rather than the whole content. So whenever we need to get access to the values of that object thus meaning getting the values of the object references we need to do any of the following.  (To clarify with an example, lets say we have camps and each camp has comments. Now if we use embedded data we have comment objects inside the camp schema containing lets say the author and hisComment. So its easy to access those values. [foundCamp.comment] however when we use object references we get ids rather than the whole name and hisComment.)  \*\*THE COMPLEX WAY\*\*  //usually findById takes 2 parameters the id and a function but in this case that’s different.  //With .exec we are executing the query and the foundCampground would come with a slightly different result. Where all the comments are filled with data.  campground.findById(req.params.id).populate(“comments [the attribute name that contains the object ID] ”).exec(function(err, foundCampground){  }  //This populates the comments array with values from the comment collection.  \*\*THE LONG WAY\*\*  /Get the comments id then using the comment id find the corresponding comments.  comment.findById(foundCamp.comments, function(err,comment){  console.log("The author is: " + comment.author);  console.log("The text is: " + comment.text); |
| 1. 338 | \_\_dirname | Gives you the directory through which the files are running from.  app.use(express.static(\_\_dirname + "/public")) |
| Authentication | | |
|  | Using passport | To use passport local we have to install the following,  Npm install -S passport passport-local passport-local-mongoose  Finally these are all the things needed for authentication to work (REMEMBER ORDER IS VERY IMPORTANT HERE!!).  var express                 = require("express"),      mongoose                = require("mongoose"),      passport                = require("passport"),      bodyParser              = require("body-parser"),      user                    = require("./models/user"),      localStrategy           = require("passport-local"),      passportLocalMongoose   = require("passport-local-mongoose");  mongoose.set('useUnifiedTopology', true); *//removing deprication errors*  *//mongoose connecton to the DB*  mongoose.connect("mongodb://localhost/auth\_demo\_app" ,{ useNewUrlParser: true });  var app = express();  app.set("view engine", "ejs");  app.use(bodyParser.urlencoded({extended:true}));  *//complex shortcut method to use a package in the app*  app.use(require("express-session")({  *//secret code to encode and decode the username and password*  *//can be anything you want*      secret: "Maruf Monem",      resave: false,      saveUninitialized:false  }));  *//These 2 lines are used to make sure our app uses passport*  app.use(passport.initialize());  app.use(passport.session());  *//this line is telling passport to use the user authenticate method*  *//which is given to the userSchema*  passport.use(new localStrategy(user.authenticate()))  *//These 2 lines are responsible for reading the session, taking the*  *//date from the session encode and decoding it*  *//this is able to do this becuase we included passport methods in*  *//the userSchema.*  passport.serializeUser(user.serializeUser()); *//encode*  passport.deserializeUser(user.deserializeUser()); *//decode*  *//Root or Index path*  app.get("/", function(req, res){      res.render("home");  });  app.get("/secret", function(req, res){      res.render("secret");  });  *//connection*  app.listen(5500,"127.0.0.1", function(){      console.log("Authentication has started");  });  In our user Schema we have to introduce passport.  var mongoose = require("mongoose");  var passportLocalMongoose   = require("passport-local-mongoose");  var userSchema = new mongoose.Schema({      username: String,      password: String  });  userSchema.plugin(passportLocalMongoose);  module.exports= mongoose.model("user",userSchema);  What the plugin does is it bring passport methods to the userSchema in order to bring authentication. |
| 1. 343 | How to add new users | user.register(new user({username: req.body.username}), req.body.password, function(err, newUser){      });  EXPLANATION: Saving passwords to the database is not a good idea so what the above line is doing is that we create a new user with just his/her name nothing else but we don’t save it to the database yet. The 2nd argument the password retrieved from the form is then hashed and then the username and the hashed password is saved in the database.  *//once the user is created this would run.*  *//This logs the user in handles everything related to the session*  *//then run the serialize user method*  *//then specifying that we would use local strategy*   passport.authenticate("local")(req,res, function(){          res.redirect("/secret");  });  So in easy terms it basically logs the user in. This part would be inside the first line of code.  app.post("/register", function(req, res){      user.register(new user({username: req.body.username}), req.body.password, function(err, newUser){  *if*(err){              console.log("There is an error: " + err );              res.redirect("/register");          }*else*{               passport.authenticate("local")(req,res, function(){                  res.redirect("/secret");              });          }      });  });  So what is happening in the database?  I have given username: maruf , password: maruf  As I mentioned before the password isn’t stored in the database rather a hash is stored there. Which we can see below:  {    "\_id": {      "$oid": "5ee85e99699d8450a076b27f"    },    "username": "maruf",    "salt": "0cfecf8c080aefef0d5966292e9fed57e86171a339cec69ebfd2e90472c086d6",    "hash": "",    "\_\_v": 0  }  Our password is hashed and that hash value is stored there and the salt is to decode this password.  By default passport handles the problem of same username. Its going to give an error. |
| 1. 344 | Login | *//Create*  app.post("/login", passport.authenticate("local", {      successRedirect: "/secret", *//if the login was successful*      failureRedirect: "/login" *// if it wasnt*  }) ,function(req, res){    });  This is the part that handles the login of users. The 2nd argument  [  passport.authenticate("local", {      successRedirect: "/secret", *//if the login was successful*      failureRedirect: "/login" *// if it wasn’t*  }  ]  Is known as a middleware. It’s the code that runs immediately when the code is called. This basically checks the username and password and logs you in. |
| 1. 345 | Logout | req.logOut();  This is easy and handled by passport. It destroys all the user data in the session. EX:  app.get("/logout", function(req, res){      req.logOut();      res.redirect("/");  }); |
| 1. 345 | Loggedin middleware | app.get("/secret", isloggedIn, function(req, res){      res.render("secret");  });  Here we are using a middle ware that we created called isLoggedIn. This is given in the get route of the path that shouldn’t be accessible if the user is not logged in. so whenever is route is called the first thing that runs is the middleware (that’s the nature of middle wares). The code for the middle ware is below:  function isloggedIn(req,res,next){  *if*(req.isAuthenticated()){  *return* next();      }      res.redirect("/");  }  What is happening here that when the method is called the middleware checks if the request is authenticated or not (passport method) then based on that run the next method [next ()] and passport knows which method to run next in this case it’s the function () that renders the secret page.  Middleware always take 3 parameters. |
| 1. 350 | User information in every page | Req.user gives us the current logged in user. With the help of this information we can show login logout sign up buttons. But we need to make this value available in all the pages that have a navbar either through pass it as a object in every render or we can create a middleware for it. Which is :  app.use(function(req,res,next){      res.locals.currentUser = req.user;      next(); *// for running the next part of the code*  });  In the navbar we would have this:   <% if(!currentUser){ %> //current user variable empty or not    <li *class*="nav-item">     <a *class*="nav-link" *href*="/login">login</a>     </li>    <li *class*="nav-item">         <a *class*="nav-link" *href*="/register">Sign Up</a>    </li>    <% }else{ %>     <li *class*="nav-item">           <a *class*="nav-link" *href*="/logout">Log out</a>  </li>    <% } %> |
|  | | |
| 1. 351 | Refactor | To refactor and make our app js more simple we need to separate the routes based on their types. So comments, campgrounds and index (which contains the root routes and auth). However we have to keep in mind what models and packages we need to require.  So for example in the campground page we need to have these  var express = require("express");  var router = express.Router();  var campground = require("../models/campground");  Because we have codes that are dependent on them. Here we have to also used a new variable called router so router is basically connecting these routes to the app.js. Replace all app.get with router.get(). EX:  *//creating new campgrounds form page (NEW)*  router.get("/campgrounds/new",isloggedIn, function(req,res){      res.render("campgrounds/new");  });  Finally at the end of each js page we need to do  module.exports = router;  Inside the app page we need to require these new js files.  var commentRoutes       = require("./routes/comments"),      campgroundRoutes    = require("./routes/campgrounds"),      indexRoutes         = require("./routes/index");  and before the listen route add app.use  app.use(indexRoutes);  app.use(commentRoutes);  app.use(campgroundRoutes);  However we can simplify these routes even more because lets say if we look into the campgrounds route we see that everytime we write /campgrounds/……. We can remove this duplicate campground writing by doing the following.  app.use("/campgrounds",campgrounds);  router.get("/", function(req,res){ //from /campgrounds to just /  router.post("/", isloggedIn, function(req,res){  router.get("/new",isloggedIn, function(req,res){  router.get("/:id", function(req, res){  Now we can change the comments route to something like this.  App.use(“/campgrounds/:id/comments”) however this would create a problem when getting the id (req.params.id) as id is not in the route [router.post("/",function(req,res){ ] to fix this we need the following:  var router = express.Router({mergeParams:true}); |
| 1. 352 | Adding username to the comment | We get the user information that we got from the req.user (isloggedin) we can add the user name to the comment (the comment scehma has to be changed.)  The new comment schema would look like this:  var commentSchema = new mongoose.Schema({      text: String,      author: {          id: {              type: mongoose.Schema.Types.ObjectId,              ref: "user"          },          username: String      }  });  Now the userschema has an attribute author that contains links to the user schema. (THAT’S HOW TABLES ARE CONNECTED IN NOSQL). As a result of this schema change we have to also change the views file. Which can now show the author name with relying on information passed on through the comment form.  <div *class*="card-body text-dark">         <p *class*="float-right">10 Days ago</p>          <h5 *class*="card-title"><%= comment.author.username %></h5>          <p *class*="card-text"><%= comment.text %></p>   </div>  Finally we have to make changes in the logic that handles the comments being passed through the form. Which is the following.   newComment.author.id = req.user.\_id;   newComment.author.username = req.user.username;   newComment.save();   foundCamp.comments.push(newComment); |
|  | Associate user with camp | We need to associate users with camps so that we know know uploaded the new camp. There 2 approaches to this we can either have the user table refer to the camp or camp refer to the user. The scenario is better because a camp has to be associated with a user where as a user doesn’t have to associated with a camp. So we have to make these changes ,  var campgroundSchema = new mongoose.Schema({      name: String,      image: String,      description: String,      author:{          id:{              type:mongoose.Schema.Types.ObjectId,              ref:"user"          },          username: String      },      comments: [          {              type: mongoose.Schema.Types.ObjectId,              ref: "comment"          }      ]  });  We also have to handle the logic behind it.  var name = req.body.name;      var url = req.body.imgUrl;      var description = req.body.description;      var author = {          id: req.user.\_id,          username: req.user.username      };      var newCampground = {          name:name,          image:url,          description: description,          author:author      } |
|  | Deleting comments along with camp | router.delete("/:id", function(req,res){      campground.findById(req.params.id, function(err, foundCamp){          foundCamp.comments.forEach(function(commentID){           comment.findByIdAndDelete(commentID,function(err){                  console.log("DELETED");                  });              });          foundCamp.remove();          res.redirect("/campgrounds");      })  }); |