* Config/
  + Connection.js
  + Orm.js
* Controllers/
  + thingsController.js
* Db/
  + Schema.sql
  + Seeds.sql
* Models/
  + Thing.js
* Public/
  + Assets/
    - Css/
      * Style.css
    - Js/
      * Things.js
    - Img/
* Views/
  + Layouts/
    - Main.handlebars
  + Partials/
    - Things/
      * Thing-block.handlebars
  + Index.handlebars
* Server.js
* Package.json
* .gitignore

Relevant Folders and Files

* Config/ - Contains the basic MySQL config configuration and CRUD queries
  + Connection.js – Requires the MySQL package, creates a connection object off of it, makes the connection, then exports that connection object
  + Orm.js – Requires the custom connection.js file and creates an orm object. The orm object contains basic CRUD queries (or more complicated queries) and passes them to the connection object to run a query. Each method may take in any combination of: a table name to alter, columns to add, values to add, an object of key/value pairs to update, and/or a condition to check against. These CRUD methods typically have a “cb” parameter as a final parameter to shoot back the query’s data to the controller function. You may also want to include some helper functions up top to convert object/array data into MySQL syntax. This orm object is exported at the end of the file. These CRUD methods may take in parameters such as:
    - All: tableName
    - Create: tableName, columns, values
    - Update: tableName, object with key/value(s), condition
    - Delete: tableName, condition
* Controllers/ - Contains all the route logic your program intends to use.
  + thingsController.js – Requires the Express package, then makes a router from it. The router primarily uses the get and post methods (and perhaps put and delete as well) and contains the logic to react to those requests. Also requires the thing.js from the models folder, stored in a thing variable. The goal of this file is to either render the Handlebars page (usually the result of a GET request) OR extract the juicy tidbits from the req.params and shoot them over to the thing.js object to call the orm with the broken up data the orm is asking for. Thing’s callback function will have a result parameter in it that contains the latest insert id or how many rows were affected. If the result contains the desired results, the router.REQUESTMETHOD will use the res parameter to give back the latest insert id (useful for a POST), or an error code (200 if good, 40FUDGE if bad, useful for a PUT or DELETE). This file will export the router at the end.
* Db/ - The MySQL commands used to initialize your database and tables, and possibly seed your table with some rows
  + Schema.sql – The commands to create your database and table(s)
  + Seeds.sql – contains INSERT command(s) to populate your table with some rows for testing or production.
* Models/ - Contains the file(s) specific to a named route and generates an orm call based on that route and juicy request parameters. For example, an actor.js file contain the CRUD methods that are specific to the “/api/actors” endpoint and tell the orm to interact with the actors table.
  + Thing.js – requires the orm.js file from the config folder and creates an orm orbject. Creates a thing object with the same CRUD methods in the orm file. Each method in the thing object expects the pertinent information from request passed into it, then it sends the table name as a string into the orm function as the first parameter, then the parameters passed into it (the thing object), and ends with a callback as a parameter. Basically, each CRUD function in this object serves as an intermediary to tell the orm what table to mess with and what data to use. Exports the thing object at the end.
* Public/ - Contains various client-side files
  + Assets/js/things.js – A good place to store your client-side jQuery event handlers, specific to a particular page. Your main.handlebars file can include a script tag to jquery, followed by a script tag that points to this file (assets/js/things.js)
* Views/ - Contains the framework that Handlebars makes use of.
  + Layouts/
    - Main.handlebars – Contains the main blueprint for the html page you want to render. When you call res.render() in the controller file, Express will generate an html response with Handlebars in mind, and replace the {{{body}}} tag with the content of the handlebars file in the folder above that has the same name as the first render parameter. So calling res.render(“index”, {thing: data}); will render the entire html page and replace the body with the content of index.handlebars above.
  + Partials/ - An optional folder that you can write complicated tidbits in and call repeatedly to use in another handlebars file. Useful for lists
  + Index.handlebars – Contains a bunch of html tags and content that you intend to use in the body tag of your html file. You can also use the key/value pairs from the object that res.render() passed in.
* Server.js – The main file that Node looks at to startup the server. Normally contains ALL the backend logic used to run the server, but that would take forever to find the one piece of code that’s messing you up. This file should only be used to establish a port, tell Express what middleware and routes to use, and finally listen on the specified port. The following steps are necessary to kick on the server:
  + Require Express and create an app object from its constructor
  + Establish a port, whether letting the server set one or one on your own.
  + Tell the app to use static files
  + Tell the app we’re sending/receiving JSON data
  + Require the express-handlebars package and make an object from it
  + Tell the app to use the express-handlebars engine
  + Require the thingsController.js file and set it to a routes object
  + Tell the app to use the routes object
  + Finally, tell the app to listen at the specified port. When Node hits this method, your terminal will execute everything inside and then just hang there until it receives a SIGINT (Ctrl-C).

Some Notes:

* A PUT request in action (using week 14, exercise 17 as an example)
  + User clicks a button that wakes up a particular cat
  + Public/assets/js/cats.js - The jQuery handler assigned to listen to that particular button extracts the cat’s id from the data-id attribute and the newSleep value from the data-newSleep attribute. Since the cat is currently asleep, data-newSleep is equal to “awake” to tell the DB what to set the cat’s sleepy status to. The script creates an object called newSleepState set to {“sleepy”: “awake”} and will send that off. Finally, $.ajax is called to send a PUT request to the database to change the sleepiness status of a particular cat. The call sends this new object to the “/api/cats/:id” endpoint. So the AJAX call will look like   
    $.ajax(“/api/cats/” + id, {  
    type: “PUT”,  
    data: newSleepState  
    }.then(…);  
    and that will send the new data to the endpoint specified in the catsController.js. After the endpoint services the request, a response comes back here and the .then executes as specified in the final bullet.
  + Controllers/catsController.js – The Express router determines that it is receiving a PUT request, so it executes the router.put method. Since the id portion of the endpoint is preceded with a colon, that means its value could be anything, and we want that value because it represents the database id of the cat we want to alter. Now we obtain that id and the req data (that the AJAX call just passed in) and set them to variables. Then we send an object containing the name of the data we want to alter as a key and the value we want to update it to as a value (in this case, the object will be {“sleepy”: req.params.newSleepState}) as the first param, and the condition (“id = “ + req.params.id) as the second param. This is passed into the model cats.js.
  + Models/cats.js – A PUT request is a MySQL update statement in disguise. Javascript files in the models folder are, by convention(?), meant to query a table with the same name. Since this is a cats file, we’re going to query a cats table. Methods in this file have the table name they intend to query as a first argument, followed by some combo of table name, table column, and/or value. The only purpose of this file is to call the orm object with the table name and values. The update method in this cats object takes the object and condition from the function parameters and makes a call to the orm. If the application you’re making also had an actors endpoint, then you would make an actors.js as well that’s identical to this cats.js, but the first argument in each orm call will specify the actors table instead of the cats table.
  + Config/orm.js – The parameters this object’s methods take in is pretty much identical to the model above, but with a key difference. The queries in this object are table agnostic. They don’t care what table you intend to use when you call them, they just need to know what table to target. Essentially, the purpose of the model files are just to call this object, but tell it (this orm) which table to hit. The update method in this takes in the name of the table to target as the first argument (which is supplied by the cats.js file) and assembles the MySQL query. Finally, the method queries the MySQL database, then hops all the way back up this chain of calls to deliver the news. The function returns the results of the query…
  + Models/cats.js - …to the callback function of cats.js, which just treats it as a hot potato and returns that immediately to…
  + Controllers/catsController.js - …this callback function which determined if the query was successful by the number of rows affected. The controller will decide if rows affected led to a good thing or bad thing, then give back a 200 or 404 to…
  + Public/assets/js/cats.js - …the .then method of the ajax handler. If a 200 came back, then the cat was successfully woken up (and you’ll probably get your face mauled later) in the database. The ajax function will make the page reload, which will then query the database for all the cat data again, and this time, the cat you just picked will load up in the awake list.