**MVC Patterns In Express -** [Ryan Schleck](https://ryan-m-schleck.medium.com/?source=post_page-----c7efa25a7c86--------------------------------) Jul 10, 2022

**Introduction**

Express is a web framework built on top of the Javascript run time environment known as Node. As discussed in a previous [blog post](https://ryan-m-schleck.medium.com/using-express-for-static-html-bc4235351473) Express can be used to serve html. Once we begin to use Express to serve multiple files of html however we run into a problem. How do we organize the data? One such solution is by using a pattern known as MVC.

Diagram

Description automatically generated

**What Is MVC?**

The acronym [MVC](https://developer.mozilla.org/en-US/docs/Glossary/MVC) stands for Model, View, Controller, and is a way of separating concerns so that each part of our app has a specific responsibility. Lets break down what each of these topics do individually:

* **Model**— Is what data the app contains. It defines the shape of what our objects look like and the form this data should take.
* **View** — This sets up how our data should be displayed and presented to the user. An HTML page would be a view hence in most MVC frameworks these are contained within a ‘views’ folder.
* **Controller**— Is the logic that will update either the model or view in response to user input.

The MVC pattern is not unique to Express at all and can be found across multiple frameworks such as [AngularJS](https://www.javatpoint.com/angularjs-mvc-architecture#:~:text=MVC%20stands%20for%20Model%20View,and%20supports%20separation%20of%20concerns.) and [Ruby On Rails](https://www.javatpoint.com/ruby-on-rails-mvc) albeit each with different implementations.

A common analogy used to explain MVC is that of a restaurant. In this restaurant there is a Table, a Waiter, and a Chef. You are sitting at the Table and are privy to how the food is presented. Thus the Table is the View. The Waiter is in charge of taking your requests and interacting with both you and the kitchen (the Chef). Thus the Waiter is the Controller. Finally the Chef is the one who knows what the food is supposed to look like, what ingredients the food is supposed to have etc. The Chef is the Model. Now that we understand what MVC is lets try to implement it within the Express framework!

**MVC In Express**

Lets examine how we would implement MVC in Express. For this we will make some changes to the simple pug creator app we used in my previous [blog post](https://ryan-m-schleck.medium.com/html-templating-within-express-js-7b668e34482a) (repo for that [here](https://github.com/FluxOfPingIntegers/express-pug-demo)). Lets focus on the M in MVC first, thus lets make a **models** folder and use it to create a new **pug** class within a file called **pug.js**. We will use this to define what a pug object looks like. We know our pug creation form will take 3 properties: name, age, and owner. Lets make sure our new pug class takes these arguments upon instantiation. We will also create a static property called **all** which is an array we will use to contain all instances of a pug we create. Finally we will need an instance method called **save** that will push our created pug into this array. Putting all of this together our pug class will look something like this:

A screenshot of a computer

Description automatically generated with medium confidence

For a better look at Object Oriented Programming feel free to check out my previous embarrassingly nerdy [blog post](https://ryan-m-schleck.medium.com/a-brief-look-at-object-oriented-programming-ecccdb08256c) about it!

Next is the V in MVC which is our Views. For this example there is no need to change our views. We will still have a setup resembling:

Graphical user interface, application

Description automatically generated

As before I am using the [**pug**](https://pugjs.org/api/getting-started.html) templating engine hence all of these files that would normally end in .html now end in .pug

Because the V hasn’t changed in MVC for this example we can move straight on to C…where a fair amount will need to be changed. Lets create a **controllers** folder at the root of our project and a **pugs.js** file within it to contain the logic which interacts with our models and views.

It’s generally convention that files within this folder are pluralized. We will also create an **error.js** file within this folder but because all this is doing is serving an error page…I feel no need to go into the contents of it. Feel free to checkout [the repo](https://github.com/FluxOfPingIntegers/express-mvc-demo) for this demo if you are curious of the contents for that file.

Anyways within **pugs.js** we are going to import our Pug class and export 3 middleware functions via our [exports object](https://nodejs.org/api/modules.html#exports-shortcut) (short for module.exports). The first middleware function will be to render our pug creation form and we will call it **getAddPug**. Just like before we will use **[res.render](https://expressjs.com/en/api.html" \l "res.render" \t "_blank)** within our middleware function and pass it a string representing our view as well as an object containing local variables consumed by our view. Next we will create a middleware function that is responsible for handling our POST request logic thus we will call it **postAddPug**. In this we will create an object called pugParams which contains the information sent via the request from our pug creation form. We will then use this object to initialize an instance of our pug class. After that we will save this instance and redirect to our pugs index page. Lastly we will have a middleware function that renders our pugs-index view where we display all of our pugs. We will call this **getPugs**. Within this middleware we are going to want to make our Pug.all array available to our pugs-index view.

The whole **pugs.js** file within the controllers folder should look like this:

Text

Description automatically generated

With this completed we will then need to head over to our **routes** folder and make some changes to **pug.js**. Within this file we are going to want to pass all of the routes (in this case the sections of the url path that are after /pugs) to their corresponding controller actions that we just defined. We will do this via a variable we will call **pugController.**We will then create a new router object via **[express.Router()](https://expressjs.com/en/4x/api.html" \l "express.router" \t "_blank)** and pass this object our defined routes via our pugController variable. lastly we will export our router object. Our **pug.js** file within our routes folder should look like this:

Text

Description automatically generated

Finally lets go to the top of our application which is **app.js.** The only change we will make here is that we will import our error middleware logic directly from our controllers folder.

Our **app.js** file should resemble the following:

Text

Description automatically generated

**Conclusion**

By using MVC we organize our app so that each aspect of it serves a concise purpose. As applications become larger and more complex this becomes increasingly important. I hope this has been a good example of how MVC can be used within Express. Using this pattern is so popular there are actually frameworks built on top of Express that implement MVC natively. Two of the most popular ones are [Locomotive](http://www.locomotivejs.org/) and [Sails](https://sailsjs.com/). I encourage you to check them out when you get a chance. Thanks for reading!