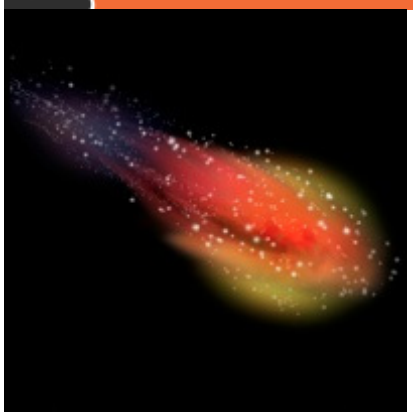

Creating a Multi-Page Site with Meteor

Gabriel Manricks on Jun 4th 2013 with 15 Comments

Tutorial Details

-
- **Framework**: Meteor
- **Difficulty**: Beginner
- **Estimated Completion Time**: 20 Minutes

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

As with any web application, creating multi-page sites requires a specialized set of tools. In this article, we'll take a look at developing a library that not only can differentiate between the different URIs, but one that takes advantage of Meteor's core features.

Intended Library Features

Whenever I have to develop a specific and focused library like this, I like to start with the outcome of, how do I want this to work?

So, we can begin by writing down some of the features that we'd like it to have:

- The ability to load different pages according to the URI
- Reading parameters from the URI (placeholders)
- Keeping the pages dynamic as per the Meteor standard

That looks pretty good. Now after looking over these features, you may think they are pretty similar to a 'router' type of library and I agree. So, let's take a look at how our 'router' library would work in action:

```
1  Router.addRoute('/home', 'homeTemplate');
2  Router.addRoute('/user/:username', 'profileTemplate');
3  Router.addRoute('/contact', 'contactTemplate');
4
5  Router.run();
```

In design, you have concepts like 'form–follows–function', which uses the idea of laying out everything first, and designing it later.

In code, I often find the opposite is more helpful. We as developers, can work in many directions and having a working example of what the design should look like, keeps us focused and efficient.

Now that I know what I want to do, it's just a matter of implementing it. So, let's take another look at our features that we wrote above; we want to be able to add routes, and have Meteor render the given template partial. Of course, Meteor's views work off of handlebar's templates by default, so I've made the decision to make this the denomination of our router.

The Breakdown

Next, let's break down the project into different sections so that we know what

features we need to work on.

- We'll start by getting the current page's URI, as we need to match the routes against something. This can be done, easily enough, using the `window.location.pathname` variable, provided by the browser.
- Next, we need the ability to add routes. This also, is pretty simple now that we have done some example code; we will have a function named `addRoute` which will accept a route pattern and a template name. This function will then have to store all of these route objects inside some kind of array.
- With the current URI and an array of routes stored, we will need some kind of method to see if they match.
- And last, but not least, we will need to take a matched route and display its attached template.

Hopefully, you can see that by laying out the requirements, it really helps to streamline the planning stage. We are now ready to jump in.

Setting Up

To get started, let's create a new Meteor project, I am going to name mine '`routerdemo`'. Now inside, we'll create a folder named '`lib`' inside another folder named '`client`':

```
meteor create routerdemo
cd routerdemo
mkdir -p client/lib
```

Next, create a file named '`router.js`' inside the newly created `lib` folder. The reason we are sticking it in '`client`', is because the server doesn't have access to the `window.location.pathname` variable and as such, won't work with our router. Putting stuff inside a folder named '`client`' assures they will only be run on the client–side.

Now inside the `router.js` file you just made, let's put in some scaffolding:

```
//////////////////////////
// Router
//////////////////////////
```

```
4
5  Router = {
6     uri: /*Current URL*/,
7     routes: [],
8
9     addRoute: /* function to add a route */,
10    getMatchingRoute: /* function to get matching route */,
11    run: /* function to display the matched route's template */
12 };
```

I think that's a pretty good scaffold, I even filled in the code for the routes array, and added some comments (progress!). Now to further develop our library, we need to discuss how we are going to match these routes together.

Matching Routes

This is not as simple as `currentRoute === route`, as we're dealing with dynamic placeholders. We want a route for '`/user/:id`' to match a URI of '`/user/42`' and so on.

To do this, we are going to have to split the URI up and do a little more in-depth analysis. Now some people might be thinking to use a regex, but, that's a little over the top if you ask me. A much simpler approach would be to split the segments up and make sure the two routes have the same number of segments, and also ensure that the parts of the route which aren't placeholders, match up.

This can easily be achieved by splitting the URI where ever there's a forward slash ('/'), using the `.split` method. So our first check would ensure that the two routes have the same number of segments.

If the route is '`/user/:id`' and we get a URI of '`/profile/42/foo/bar`', we don't even need to do any further checking, one has two segments and the other has four, so that seems like a good primary check. The next thing we can do is filter through the lists and make sure each piece that isn't a placeholder matches up. If these two checks are true, we know the route matches.

Setting the URI Variable

So let's get started with setting the `uri` variable:

```
uri: _.compact(window.location.pathname.split("/")),
```

In the above code, we're splitting the array on forward slashes, and turning the string into an array of segments. Then we're using Underscore's `compact` function to remove any empty items from the list, these could be caused by a forward slash at the beginning or by someone using two forward slashes by mistake. By doing this, it makes our system much more forgiving.

Adding Routes

Next, we need to create the function to add a route, this is a fairly similar process, but because we are going to be matching the placeholders later, we are going to store not just the segments and the template name, but the indexes for the placeholders as well.

Here's the completed function:

```
addRoute: function(route, template) {
   var segments =  _.compact(route.split("/"));

   var placeholders = _.reduce(segments, function(currentArr, piec
      if (piece.substr(0, 1) === ":") {
         currentArr.push(index);
         segments[index] = piece.substr(1);
      }
      return currentArr;
   }, []);

   this.routes.push({
      route: segments,
      template: template,
      placeholderIndexes: placeholders
   });
},
```

We start by splitting up the route into segments, just like we did for the URI, but this time we also need to store the indexes of the placeholders for future reference, using Underscore's `reduce` method.

For the unaware, the `reduce` function is similar to the `each` method, it also cycles through all the elements of a list, the difference being, it passes whatever each iteration returns to the next item, ultimately returning the results to the given variable. We are starting with a blank array (the 3rd parameter) and we are adding each index as we find them and passing that array along until finally, it's returned back to the `placeholders` variable.

The next thing you'll see going on in here, is that we are renaming the segments that are placeholders and removing the colon. We do this purely for aesthetic reasons and later on, it will make it easier to reference in the templates.

Finally, we push the new data to our routes array, which we created earlier.

Matching a Route to a URI

The next step is to filter through the list and look for a route that matches the current URI.

Here is the complete function:

```
getMatchingRoute: function(){
     for (var i in this.routes) {
         var route = this.routes[i];
         var data = {};

         if (route.segments.length === this.uri.length) {
             var match = _.every(route.segments, function(seg, i){
                  if (_.contains(route.placeholderIndexes, i)) {
                       data[seg] = this.uri[i];
                       return true;
                  } else {
                       return seg === this.uri[i];
                  }
             }, this);

             if (match) {
                  return {
                      data: data,
                      template: route.template
                  }
             }
         }
     }
```

```
24        //no matches (add 404 or default template maybe?)
25        return false;
26    },
```

We are doing quite a few things here, so let's walk through it. We begin by cycling through the array of routes, and we assign the current route to a variable, along with an empty data object to store the placeholders.

Next, we do the initial check of making sure the two routes have the same number of segments, otherwise, we just cycle on to the next route. If they do have the same number of components, we have to check whether the segments match, this can be done using Underscore's '`_.every`' function. This function is again like the '`_.each`' method, except that it returns a boolean. The way it works is it will run the function for each item in the array, if they all return true, the function will return true, otherwise it will return false, so it's perfect for doing things like this where we need to verify each segment.

Now the check that we are performing is pretty easy, if it's a placeholder, then it automatically fits, as a placeholder can be equal to any value. If it isn't a placeholder, we just make sure the two segments match, pretty simple.

In order to check whether or not this is a placeholder, we pass the current segments index (stored in '`i`') to Underscore's `_.contains` function, which will check its value.

Now you may be wondering what the first line inside this ‘`if`‘ statement is doing, well, it’s storing the segment in the data array under the given placeholder name. So, say for example you had a route of ‘`/user/:name`‘ and the current URI is ‘`/user/bob`‘, then this line will add a property to the data object called ‘`name`‘ and pass it a value of `bob`.

The rest is fairly obvious, we pass true or false, depending on the circumstances, and the result gets stored in ‘`match`‘. If match is true, we return the data along with the templates name, and if there was no match, we return false. And that’s it for our `getMatchingRoute` method.

So far, we can get the current URI, we can add routes, and we can find a matching route, the only thing left is to display the correct route, and for this we need to write the ‘`run`‘ method.

Displaying the Template

Meteor uses handlebars for templates and stores all the templates in a variable, appropriately named, ‘`Template`‘. Now, if you are familiar with handlebars then you know these templates are just functions, and by calling them `(`optionally passing in some data`)` we get back the template’s HTML.

Now, calling these functions to get the template’s HTML would work fine, but it isn’t very Meteor–like, as what we’d end up with is just a normal static website. Luckily, adding in the dynamic behavior is easier than you might think, all we need to do is wrap the function call in a ‘`Meteor.render`‘ call. Putting it inside this function will make it react to changes in the data and keep it ‘live’.

The Run Method

Because of this, running the router is very simple, let’s create the run method:

```
1  run: function(){
2      var route = this.getMatchingRoute();
```

```
    if (route) {
        var fragment = Meteor.render(function() {
            if (Template[route.template] !== undefined) {
                return Template[route.template](route.data);
            }
        });

        document.body.appendChild(fragment);
    } else {
        //404
    }
}
```

We start by getting the matched route, using the `getMatchingRoute` function which we just wrote, we then make sure there is a match, and finally we use an `else` statement to handle displaying a `404`.

Inside the if statement, we call `Meteor.render` and inside, we check and call the returned template, passing with it the data from the placeholders. This function will return an HTML fragment, which we can then just append to the document's body.

So with about `60` lines of code, we've completed our router.

Testing It Out

The next step is to test it. I'm going to use the same code that we wrote earlier when we planned out this project, as it will be a good measure of whether we accomplished, what we wanted to accomplish. Let's add a file named `main.js` inside the `client` folder and add in the following:

```
Meteor.startup(function(){
    Router.addRoute('/home', 'homeTemplate');
    Router.addRoute('/user/:username', 'profileTemplate');
    Router.addRoute('/contact', 'contactTemplate');

    Router.run();
});
```

In the above code, we first need to make sure our templates and body will be available before we try working with our Router. We do this by wrapping all of our code inside of the `Meteor.startup` method call. This will ensure everything is ready, and inside of the `startup` method, we can then add our routes and run the router.

Creating Our Templates

Now let's create a couple of templates, this can be done anywhere, you can create a subfolder inside the `client` folder named templates and create a separate HTML file for each, but since these will be short templates and just for example purposes, I am going to put them together inside a file named '`templates.html`' inside the '`client`' folder:

```
<template name="homeTemplate">
    <h1>This is the Home Page</h1>
</template>

<template name="profileTemplate">
    <h1>Profile Page</h1>
    <p>Welcome back {{username}}</p>
</template>

<template name="contactTemplate">
    <h1>Contact Page</h1>
    <p>Contact me on twitter at {{twitterName}}</p>
</template>
```

The first template is pretty basic, it just contains a little HTML code for the home page's heading. The second template is very similar to the first template, but this time we use the `username` route parameter. Now the last template also uses a placeholder, but its route doesn't have the `twitterName` segment. This is because standard Meteor placeholders will still work, and work reactively.

Back inside the `client` folder, let's now create a file named '`templates.js`', to declare the contact placeholder.

```
Template.contactTemplate.twitterName = function (){
    Session.setDefault('twitter_name', '@gabrielmanricks');
    return Session.get('twitter_name');
}
```

You could have just returned a string, but I wanted to demonstrate that everything is still reactive. The last step is to delete the default html and js files from the root directory (in my case they are named routerdemo.html and routerdemo.js) With that done, start the Meteor server and navigate to the given routes.

Try going to '`/home`' or '`user/gmanricks`' or '`/contact`' and they should all work for you as expected. Another thing is, since we stored the twitter name in Session, we can just open up the browser's console on the contact page and enter:

```
1 Session.set('twitter name', '@nettuts');
```

And you will see that the page will update in real time!

Summary

In this article, we built a basic router library, while still giving it a Meteor twist. We covered a lot of the core concepts and as it turns out, a lot of Underscore concepts as well.

In the end, I hope I got the message across that there is no real "magic" going on here. It's really all about implementing what you need, as opposed to what you can.

Thank you for reading, I hope you've enjoyed it. Like always, if you have any questions you can leave them below or ask me on the NetTuts IRC or on my Twitter.

Note: If you are interested in learning more about Meteor, I have just released my new book, which details the process of building an app from its conception & planning to securing & deploying. You can pick–up the book in both ebook format as well as softcover from Amazon.

Tags: meteormulti–page site

By Gabriel Manricks

This author has yet to write their bio.

Note: Want to add some source code? Type <pre><code> before it and </code> </pre> after it. Find out more