**MEAN Stack (Angular Js)**

So far in this session, you focused on building the server portion of your web application. Now that you completed the rest API chapter, you have a fully-fledged web server. However, to provide an interface that's friendly to humans, you're going to need a clean, user friendly UI. In this chapter, you will focus on building a browser based client for the server from the rest API chapter. AngularJS is a popular framework for building sophisticated browser based clients.

So let’s fallow below recipes to hands on angular js.

**Recipe1: Hello angular js**

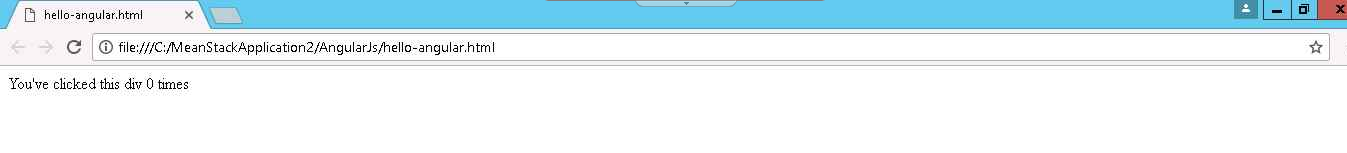
Create an html page as below.

//hello-angular.html



So here's a hello world level example of writing code with AngularJS. Every time I click this text, this counter increments.

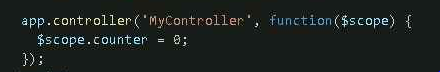
This is a stand-alone HTML file. So you can open this file in the browser by right clicking and opening it in Chrome. So for instance, this file://prefix means that the HTML is loaded from a local file. There is no web server for the web page in this HTML file.



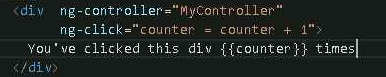
Now, the code in this example is not terribly complex. But it highlights the most fundamental feature of AngularJS, which is two way data binding. Two way data binding ties the state of a JavaScript variable, which is this counter variable to the state of an HTML element, which is these particular curly braces. So when you click on this div, this ng-click directive executes to increment the counter, and the text updates to reflect the new state of the variable.

Now, you may have noticed that this myApp module is also referenced in this ng-app attribute. The ng-app tells AngularJS which module to use for that page. Now this array, which contains the string ng, is a list of modules that your myApp will depend on. The ng module represents the AngularJS core. You can omit the ng, but the ng is useful for testing directives.

Now the most interesting part is this controller block. Controller is a function which initializes an AngularJS scope. This dollar scope variable is a local pass into your controller, which you control the initializes with some variables.



Then this ng controller directive tells AngularJS to create a new scope, and initialize it using the MyController controller. Then AngularJS will expose the scope's internal state to all elements within this div.



**Recipe2: Single page app**

In a single page app, or S-P-A, or SPA for short, your browser loads a single page and then switches between different views.

Create an html page as below.

//hello-spa.html



This is a stand-alone HTML file. So you can open this file in the browser by right clicking and opening it in Chrome.



For Instance, this page is a trivial single page app. Notice this About link. When you click this link, you visit a separate view. Note that the URL changed a bit, but only the part of the URL after this first hashtag. The part after the first hashtag is known as the fragment identifier. When the fragment identifier changes, the browser doesn't contact the server. So when you clicked on the About link, the page changed without reloading the page or without contacting the server.

This file pulls down two JavaScript files from the CDN, angular.js and angular-route.js. The angular-route.js file is AngularJS's client side routing module. Now the concept of routing in AngularJS behaves similarly to the express routers that you wrote in the rest API session. Like how express routers mapped URLs to route Angular functions, AngularJS's router maps fragment identifier values to AngularJS templates.

You need to add this ngRoute dependency to your module. The ngRoute module, which is defined in the angular-route.js file that you included here, includes all of the AngularJS functionality for single page app routing. You need to list it as a dependency of your myApp module.

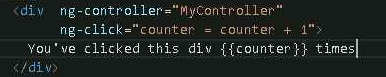


Body uses this div right here that's marked with the ng-view directive. This enables your single page app to have a common header and footer that don't change when the view changes.



**Recipe3: Directives and Controllers**

You may recall from recipe1 that attributes like this ngClick attribute are examples of AngularJS directives. Directives are a very powerful AngularJS feature.



For instance, here is a directive-based implementation of the hello-angular.html example from the recipe1. It works exactly the same way, but the div you're clicking on is actually a directive.



So now, as you can see, the MyController controller stayed the same, but there's this directive function that creates something named counter-Directive. The function passed to counter-Directive returns an object that contains a template and a controller.

So now at this point, you may be wondering why you're using a directive here rather than just running the HTML and controller, as you saw in the recipe1. So there are two reasons.

Breaking up your code into directives leads to cleaner and more modular code. A classic AngularJS rookie mistake is to write one controller per page and put all your code in that one controller. This works OK for small apps, but as your app grows and you need to reuse the same HTML in different places, the one-page, and one-controller paradigm becomes untenable.

The second reason is related to a common AngularJS pitfall, the famous flicker effect. In other words, when the angular.js file is slow to load, a user may briefly see your AngularJS templates in their uncompiled form, curly braces and all.

However, directives provide a cleaner way to avoid the flicker. To see this in action, let's go to the recipe1 example but disconnect from the internet so the page can't load AngularJS. So as you see here, couldn't load AngularJS. As you can see, the page also shows the uncompiled template.



However, if you do the same thing with the directive version, you get a blank screen, which is a good thing, because in this case loading nothing is better than displaying this uncompiled template. This means no curly braces on your site when the user is on a slow Wi-Fi connection.

**Recipe4: Services and HTTP**

In this recipe we have to create client in angularjs for rest api. To make http request from angularjs , follow below steps.

//hello-http.html



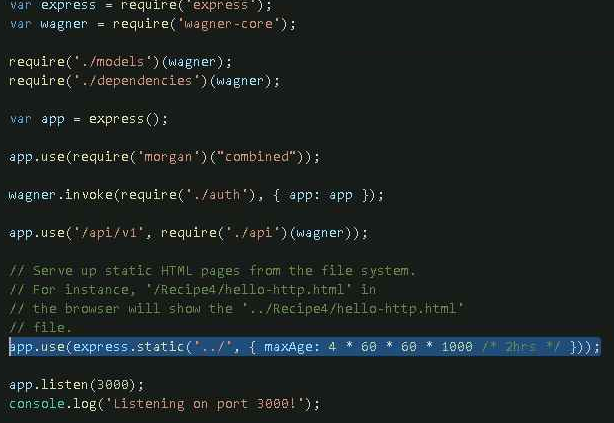
This is the hello-http.html file. This code will be the basis of your retail applications user menu, which will show the information about the currently logged in user. The user menu is contained in a separate JavaScript file, which is included with this app.js call. All this HTML file does is use this user menu directive.

**Let's take a look at app.js.**



So here's the user menu directive, and here's the MyHttpController controller. MyHttpContoller relies on AngularJS's built-in $http service, which enables you to make http requests to a server.

There's one addition to the server code that you should be aware of-- this one express.static call. This call makes HTML files and this chapter sample code available via http.



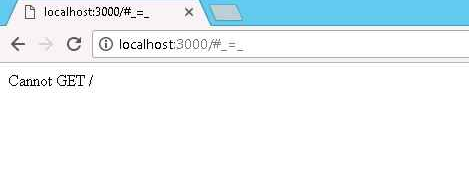
Now in order to talk to your rest API, you need to actually start the http server that you wrote in the rest API session by running nodemon index.js in command prompt. Just copy facebook Oauth credentials in the config.json file into the server directory in this session sample code.



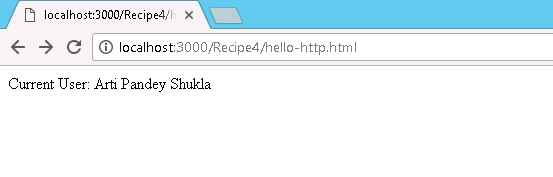
For instance, if I navigate to this hello-http.html file in the browser, I get a prompt to log in. This URL links to the /auth/facebook route that you wrote in the rest API session which will log you in with your Facebook account.



So when you click Login, you get redirected to Facebook to authenticate. And now once you log in, you get redirected back to the home page.



Now, if you go back to hello-http.html, you'll see that I'm now logged in with my Facebook account.



So now you'll notice that the browser is no longer just visited a static HTML file. It's actually talking to a node.js server.

**Recipe5: browserify and gulp-browserify**

One of the key advantages of writing your server in Node.js is that you can reuse code that you wrote for Node.js in the browser. In theory, NPM modules that you use can also be used in the browser, so long as they don't write to the file system.

Node.js JavaScript, also known as CommonJS, with its synchronous require calls and module to exports property, won't work in the browser as is. For instance, try pulling up the Chrome developer console and typing module.exports or require of moments. Errors all around.

So are we doomed to have to write JavaScript to two separate paradigms and not be able to share code between the two? Well, that's where the browser refine module comes in.

Suppose you have this simple Node.js JavaScript file sample.js that exports a single string, Hello World.

//sample.js



You have another file, where you require in the sample file. So in this file, you want to alert the Hello World string, that sample.js exports.

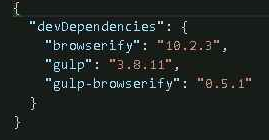
//index.js



So how are you going to run this code in the browser? Enter the Browserify package.

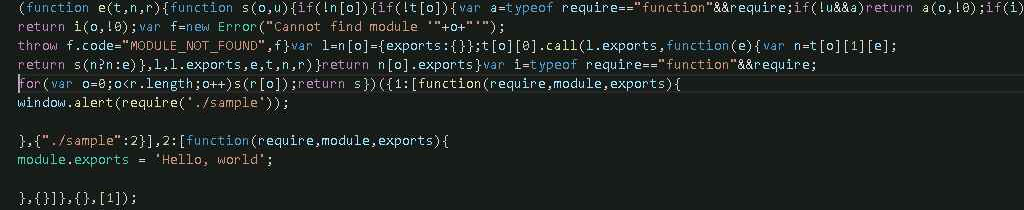
As the name implies, this package compiles Node.js style JavaScript into a file that you can include in the browser.

//Package.json



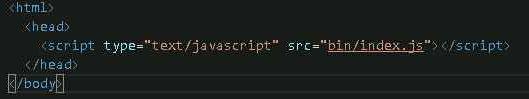
To use Browserify to compile the index.js file into browser side JavaScript, you need to run this command. browserify –o bin/index.js index.js

This dash o flag specifies the output file. Once this command is done, you now have a file in your bin directory called index.js. Let's take a look at what index.js looks like. So this is what the compiled index.js file looks like.

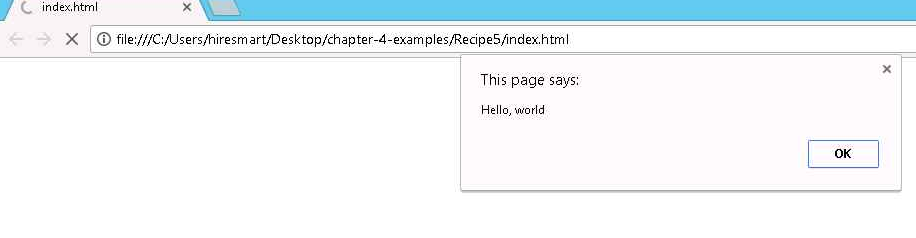


While it isn't very human readable, the general idea is that Browserify reads all of your Node.js files and wraps them in a function that exposes the browser port of the require function and the module and exports properties that Node.js uses. As you see here, the function wrapper takes parameters require, module, and exports, and exports Hello World.

So now, once you have this bin/index.js, you can include it in an HTML page with the script tag.



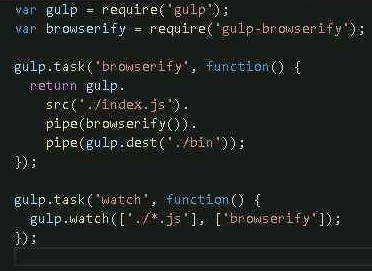
Once you open up index.html in a browser, you see this Hello World pop-up.



Now, recompiling using Browserify from the command line every time is a pain. So Node.js developers typically automate this process using Gulp.

The gulp-browserify package is a plug-in for Gulp that runs Browserify for you. So here's the Gulp file.js, containing the Gulp rules that you'll be using to build Browserify code in this session.

//gulpfile.js



The Browserify task takes this index.js file and compiles it into the bin directory. The watch task watches your JavaScript and executes the Browserify task every time something changes.