Slide 1

Wednesday, November 5, 2014

9:56 PM

Slide 2

Wednesday, November 5, 2014

9:56 PM

Slide 3

Wednesday, November 5, 2014

9:56 PM

Why does my server code need to know html?

Well, you because you want to display data in a complex manner on a webpage, and doing that in javascript and css is a mine field of problems and shortcomings without these powerful wrapper tools and a server lifecycle.

Well, that's not really the case anymore. More unification in browser standards and the potential unlocked by the html5 enhancements, incredible community support and thousands of open source libraries, as well as a lot of fancy new tooling - make developing an application that truly LIVES in the browser a pretty enjoyable task, and remarkably powerful.

Why not separate what the server is good at from what the client is \*NOW\* pretty good at? Both sides get a lot simpler. Let's take a quick look

Slide 4

Wednesday, November 5, 2014

11:06 PM

Single page application architecture deliberately separates the server's roles from the client's.

This allows us to focus on the performance and effectiveness of each component directly, while also making each piece simpler and more straight forward to implement

Web API is that guy on the left. On the client side, what you're looking at is essentially a complete application, living in the client's browser. Now a page lifecycle doesn't really involve the server anymore (except for the initial bootstrap). Javascript is used to manage the html/css on the client, and fetch data from the server when needed.

That SOUNDS like it might be a pain. Angular is one answer to that pain, and it does a fantastic job in my opinion

Slide 5

Wednesday, November 5, 2014

11:12 PM

Angular is a comprehensive framework that enables the construction of rich applications with complex interactions in a manner that lends itself to SOLID principles.

It contains many pieces that can be used independently or together in a modular fashion.

An important bit of information - With angular, the front half of your application no longer cares all that much about who provides its data from the server. Node, ASP .Net Web API, Perl, they'll all work as long as they can return data in the requested format.

At Castle, we're replacing a large Silverlight LOB with Angular and Web API. Our SL application was built using WCF RIA Services. It's basically the opposite of what we're looking at here. Exceptionally tight coupling between the server and client data architecture. They are essentially inseparable. We've learned the hard way that building an application in this fashion can have enormous consequences in maintenance and new development cycles.

Slide 6

Wednesday, November 5, 2014

11:19 PM

Web API is a framework that makes it easy to build http services that work across platforms and devices.

That's pretty important right there. Web API doesn't care who is calling it, and it's built to accept calls from pretty much any platform/device.

-Start on the talking points

It's straightforward. The basic templates gets you a project scaffolded and running in about 2 minutes. Out of the box it comes with a host of rich features that are preconfigured to work for you.

It's flexible. It can automatically switch between returning JSON and xml and for advanced needs there are many ways to customize

-It's lightweight. It doesn't even need to be hosted in IIS. You can host it in a Windows Service or even a console application. Can be run on linux, even if it is a fringe use case at the moment. vNext promises to do more to bridge the gap.

Makes transferring data between many sources trivial as it is designed to be RESTful. Allows for integrated token based authentication that alleviates the need for handling of cookies on some platforms.

-It's scalable, and well suited to an enterprise level application implementation. Rich tooling and high performance.

Ok, ok, ok, enough talk. Let's take a look at a simple Single Page Application and then peek under the hood to get a sense of what's going on.

Demo

Wednesday, November 5, 2014

11:43 PM

First:

Start creating a simple Web API application through visual studio.

It's throwaway, we're just showing how easy it is.

(do stuff)

For the scaffolding of the Angular app, I'm using a visual studio template that gives you the most basic application possible that works right out of the box. Show this as well. Also throwaway.

(do stuff)

Now open our solution and run it, and show the fully built application

It's Random Raleigh Restaurant Chooser! My wife and I like to eat out and try new places, but neither of us can decide on our own what we want to eat!

This is a very simple SPA that exhibits how the client has taken control of rendering duties as well as the application lifecycle.

When we click the generate button, Angular makes a call to our web API service, retrieves the data, and then assigns the returned data to a variable which is then data bound to an html template.

Let's give it a spin. As you can see, as we retrieve elements, several fields are updated, as well as a map.

What's important here is that the only data being sent to and from the server to make that happen is data relevant to the call at hand (no viewstates, etc), and it's being returned and then data bound to a view with minimal code. This all happens without page reloads.

How about a detail view? Angular makes routing to different pages within your application possible. Let's look at a page that lets you simply peruse the list.

Here I'm pulling data down about all of the restaurants and displaying them in a master-detail view.

Again, the only thing being pulled down is data relevant to the operation, and we're still running the same instance of the application that we used on the homepage. Notice that our URL has changed as we moved to this new page, and that I can click back and forth to be returned to a given state, all while remaining in the application.

Let's take a look under the hood

What's going on here?

Well, we've got our front-end, let's start there, and walk through building it

(open half built project).

Build Index Page

Thursday, November 6, 2014

3:14 PM

We've got 4 views

- Home, detail, about, and "other"

Let's build the homepage

Using angular's routing services we define these views as states. States allow for powerful deep linking with parameters, promises, and resolves. We're not doing a whole lot in our application other than defining a few vanilla states. States can also act as the glue between your view and your controller.

So we'll start by tossing in some html that outlines the way our template should look

<h2 class="text-center">Welcome To The Random Triangle Restaurant Selector!</h2>  
   
   
   
<button class="text-center btn btn-primary btn-lg"   
        ng-click="getRandomRestaurant()"   
   
        style="display:block; margin-left:auto; margin-right: auto;">Feed Me!</button>

There's our header and our button, pretty standard stuff so far.

Let's toss in the main div

<div class="col-xs-12 ng-cloak">  
    <div class="row">  
        <div class="row">  
            <div class="col-xs-12">  
                <h2>**{{**randomEntry.Name**}}**</h2>  
                <p>  
                    <ul class="list-inline list-unstyled">  
                        <li>  
                            **{{**randomEntry.Address**}}**  
                        </li>  
                        <li>|</li>  
                        <li>  
                            **{{**randomEntry.Phone**}}**  
                        </li>  
                        <li>|</li>  
                        <li>  
                            **{{**randomEntry.Email**}}**  
                        </li>  
                    </ul>  
                </p>  
   
            </div>  
        </div>  
        <div class="row">  
            <div class="col-xs-6">  
                <p>  
                    <strong>Cuisine</strong><br />  
                    <span>**{{**randomEntry.Cuisine**}}**</span>  
                </p>  
                <p>  
                    <strong>Pricing</strong><br />  
                    <span>**{{**randomEntry.Pricing**}}**</span>  
                </p>  
                <p>  
                    <strong>Rating</strong><br />  
                    <span>**{{**randomEntry.AggregateRating**}}**</span>  
                </p>  
                <p>  
                    <h4>Quotes</h4>  
                    <blockquote>**{{**randomEntry.Quotes**}}**</blockquote>  
                </p>  
            </div>  
            <div class="col-xs-6">  
                <ui-gmap-google-map center="map.center" zoom="map.zoom">  
                    <ui-gmap-marker coords="map.center" idkey="1"></ui-gmap-marker>  
                </ui-gmap-google-map>  
            </div>  
        </div>  
          
    </div>  
      
</div>

Say whaaaat? What's with the {{}}, and what is the <ui-gmap-google-map….stuff all about?

In short, the curly braces say, replace this text with the value of the model property inside the brackets. The ui-gmap thing is a directive, you can think of it as a nice wrapper around google maps that allows us to interact with it in a more declarative fashion. Let's not spend too much time here, but…

As we'll see in a few minutes, the data that is requested from the server is returned and then assigned to a javascript object, that with the help of Angular, we can BIND the data to the view.

No long lists of jQuery calls to update each field. That gets unwieldy very quickly when you get into dozens of properties that get conditionally updated. It's a lot to keep track of, and this syntax REALLY makes things easier - data gets updated automatically when new data is assigned. Additionally, it makes your HTML more expressive. Knowing what we know, it's easy to look at this view and see where our model properties will be mapped, and how they will be used.

Build Angular Service Method

Thursday, November 6, 2014

3:22 PM

Angular encourages you to use built in services to perform common tasks, like data calls. Using the built in services makes system level interceptions pretty easy. Imagine a large application where all calls require an authentication header to be set. Using angular's services allows you to globally inject that header into all requests. Another fun use case that Web API plays well with: Imagine an application using Oauth tokens for authentication. Angular appends the token to all outgoing requests. Well, you can also intercept the responses and do things before handing it back over to the caller. We wrote an interceptor that detects when a call has returned with a 401 response. It holds off returning the value to the caller, and instead, reissues a new token request. If it works, it then REISSUES the original request with the new token, and THEN returns it to the caller. The original callers doesn't even know the endpoint is secured, let alone that it's request failed and was reissued before returning. That's pretty powerful.

return $http({  
            method: 'GET',  
            url: '<http://localhost:57390/api/Random/GetRandom>',  
            data: { },  
   
        }).then(function (response) {  
            if (typeof response.data === 'object') {  
                return angular.fromJson(response.data);  
            } else {  
                return $q.reject('Unknown error. Please try again.');  
            }  
   
        }, function (response) {  
            var msg = 'Unknown error. Please try again.';  
            return $q.reject(msg);  
        });

Ok, this ought to look pretty familiar to those with jQuery ajax experience. It's similar in some ways. Most importantly though, we can see that all it does is make a call to an endpoint, and then deserialize a JSON object and then return it to the caller.

Moving to the server side

Thursday, November 6, 2014

3:46 PM

Ok, so we've seen at a high level how Angular glues together the pieces of a Single Page Application.

Let's take a look at how Web API fits in.

Let's start by creating our controller. The controller is where our service methods will live.

Creating a controller is pretty easy.

First, of note: The data we're accessing is provided by a SQL database. I've used Entity Frame work 6 to get some data access working in pretty short order. I won't go into the how-to's of that today, but how that works isn't really that relevant. It could just as easily be from a csv file.

Ok, let's create that controller. Web API uses routing to handle incoming requests. By default, requests will be under /api/{ControllerName}/{Action}. So, we created our controller, called RandomController. When calling it, the controller name will be Random, because the convention of Web API is that your controllers will have that at the end of the name, and can therefore be omitted when calling.

Now, let's go ahead and add our first method.

[EnableCors("\*", "\*", "\*")]

private static Random rnd = new Random();  
         
       [HttpGet]  
       public RestaurantEntry GetRandom()  
       {  
           using (var context = new CodeCamp2014Entities())  
           {  
               var entries = context.RestaurantEntries.ToList();  
               int r = rnd.Next(entries.Count);  
               return entries[r];  
           }  
       }  
       [HttpGet]  
       public List<RestaurantEntry> GetAll()  
       {  
           using (var context = new CodeCamp2014Entities())  
           {  
               var entries = context.RestaurantEntries.ToList();  
               return entries;  
           }  
       }

Ok, so the method is called GetRandom, so our routing path will look like /api/Random/GetRandom

We've given the method an attribute called HttpGet that tells Web API that we'll be accessing this endpoint via GET calls as opposed to a POST or a PUT, or a DELETE.

Note that the return type here is simply our entity type. We're not doing any special packaging or anything like that. We don't convert it to JSON, we don't really do much of anything other than return it.

Come to think of it, this looks a lot like a standard C# function. The difference is that the request pipeline will handle all of the serialization and lower level details. We can get into that pipeline for our own purposes if needed, but for right now we're good to go.

As I've mentioned, Web API doesn't really care who calls it, or what platform they're from. In fact, it's made to please everyone. If you call the service and request and XML content type, you'll get XML. If you request JSON, you'll get JSON.

Let's run our solution and hit a few breakpoints and inspect what's going on.

Running the application

Thursday, November 6, 2014

4:49 PM

Ok, first, let's set a breakpoint in our Web API method. We'll load the application, and we should immediately see a call to it because I've set the application to go ahead and fetch the first response as soon as the application loads.

Hit breakpoint.

Step through to far side of the DAL call.

Inspect the element in Visual Studio. We'll examine the properties.

Let's look at those properties side by side with our Angular view. As we can see, the property names align.

Good. Now let's allow it to return, and then let's take a look at our request in Fiddler.

Quickly show the request, and then quickly show how changing the content type to xml changes the output with no further configuration necessary.

This time, let's set a breakpoint on the Angular service that made this call, and see how the response looks to it.

(hit breakpoint)

Ok, bingo. Here we can see that a JSON string is returned to the called. Angular provides a helper method to convert this to a javascript object. From there, we return it, and in the body of that return, we can see that the value is assigned to a variable in our controller.

As I mentioned earlier, the angular routing we're using binds this controller to the view we've been looking at. From here, you can see how that binding follows through. Note there's nowhere in javascript that we're assigning values to DOM elements.

So, to review, we've essentially been able to take data in its natural state on the server (C# entities), and return it to an arbitrary client without any manipulation of the data on our part. From there that same object is used by Angular to render the view, and all is well. We've gone through the entire stack with very little action specific code.

Slide 7

Thursday, November 6, 2014

4:59 PM

Ok, let's return to our slides for a moment and review what we've seen with Web API.

It's not a kitchen sink. WCF is a kitchen sink. It can do anything, but even something simple is kind of difficult.

Web API works right out of the box, and is highly extensible so when the time comes, more advanced scenarios are well supported

Haven't mentioned this yet because it's beyond the scope of this discussion, but it supports Odata right out of the box. V1-4. One advanced use case that's likely to come up quickly in a large LOB application is the need for showing grid data with server side paging/filtering/sorting. This feature comes in pretty handy when you get there.

It's open source! Web API is actually open software! As a practical matter, this doesn't have huge implications for most uses, but it does speak to the spirit behind Web API. It wants to be the best at providing RESTful http services, and open sourcing the code base has undoubtedly resulted in improvements that the entire community benefits from.

It's so flexible! We looked at returning JSON and a bit of XML, but we can return anything! Images, documents, you name it.

Let's review some of the benefits to the entire Angular + Web API approach.

Slide 8

Thursday, November 6, 2014

5:27 PM

So, end of the day, what does ALL of this buy us? Why do I care?

First, separation of concerns. There's not one framework to rule them all, they are target frameworks that stick to what they're good at. Web API isn't in the business of generating dynamic html/css, it's in the business of acquiring and return datasets to be used by any caller. Angular is in the business of effectively managing client side application behaviors like routing, data display, and statefulness.

Code simplicity. There IS a mismatch between something like an ASP.NET MVC Web Control or a WebForms control. It's hiding what's really happening from you, and there ARE cases where that can bite you, or at minimum, provoke you to concede to a sub-par user experience because that particular control couldn't QUITE do what you want. Relevant code lives where it should. Client side code lives together, server side code lives together. They exchange contracts, but that's it!

Testability. We've glossed over this in this presentation, but separation of concerns, built in DI support, and modular design allow, and promote, you using design patterns that are highly testable. Testable code is maintainable code :)

Uncompromised User Experience. Many users are used to static pages that refresh (yuck!), but it doesn't mean that we can't do better. SPAs allow for richness that is PRETTY dang difficult to mimic using other means. Even if you can, you're swimming upstream.

Slide 9

Thursday, November 6, 2014

5:44 PM

SPAs are a very natural way to develop an application. It allows you to actually build things the way that other frameworks only attempt to (with varying success). It's a big change, especially for you Webforms guys, but I promise, it's a lot of fun if you embrace it!

Angular is a very rich framework that provides the building blocks and tools to build the rich interactions with the browser that are required for an SPA to flourish.

Web API is a focus driven framework for delivering http services to ANY client, including Angular.

Web development is fun! It's not 2002, when admittedly, some aspects of web development were not a lot of fun. There were wildly different implementations or gaps in implementation between browsers, a smaller community of hardcore client developers, and a real shortage of utilities like Angular to pick up the slack. We've come a long way in the intervening time, and now a vibrant, rich ecosystem of tools and techniques exist to make it ACTUALLY a fun experience.

Thank you :)