

How to Think about Web Architecture

John Browne

Technical Product Manager

Mobilize.Net

Historically we at Mobilize have focused on tools to transform app source code from one desktop platform to another, newer, desktop platform—the best known of these is VBUC which moves VB6 to .NET.

More recently we developed and released WebMAP to move desktop (C#/Winforms/.NET) apps to a modern web architecture. Because this is such a major paradigm shift (desktop to web) it leaves some desktop developers scratching their heads a little when they look at the migrated code and wonder "where's my class/form/event handler/etc.?"

First of all there are a lot of differences between a Windows app and a web app, not the least of which is the physical separation between client and server. I wrote about some of these issues on the Mobilize.Net blog; if you're unfamiliar with web development this is a good place to start.

In this article I want to explore the architecture that WebMAP creates following a migration. Since we are using a hybrid pattern pieces of it may be familiar to you but overall some explanation is called for.

First of all, let's clarify exactly what we mean by the word "WebMAP:"

- A tool for migrating source code from one platform to another
- A set of helper classes in C# that are part of the new, migrated app
- An architecture of the new, migrated app.

The tool is proprietary; the helper classes are included in the created app and are easily read for understanding; the architecture is the subject of this paper. Note that you probably won't need to spend a lot of time with the helper classes as they handle the "under the hood" stuff to make your app run. At least some familiarity will be helpful when you are debugging your app, however.

One tier or two?

As I mentioned above, the key difference between a desktop app and a web app is that the web app (and by that I mean the kind of modern web app we create) is a multi-tier application that is divided into a front end (i.e. client running in a browser) and a back end (server running remotely in a datacenter or cloud). Another key difference is that on the desktop you normally have a single user with a dedicated execution thread and on a web app the server needs to handle multiple requests all sharing an execution thread. Those two differences make for lots of changes in the app architecture.

Remember we are talking about applications on the browser, not web pages. Web pages—even interactive ones—are pretty simple compared to re-creating the functionality and UX of a desktop app in a browser. For our discussion it's helpful to assume any app we are talking about is heavily forms-based with a database back end. In a Windows application a traditional way to handle these kinds of apps was a "code-behind" approach where events on controls triggered handler code. Even older ASP.NET ("classic") apps using webforms supported this paradigm.

Front ends and back ends

First consider that the app has to be instantiated as a browser session. This means either we use some plug-in approach like Flash or Silverlight (both of which are losing support in modern browsers due to security concerns) or we have to generate pure standardized code like HTML, CSS, and JavaScript. Those will all run on all modern browsers and you shouldn't need hacks asking the browser what it can support, unless deep backwards compatibility is important to you. This combination of HTML5, CSS, and JS—especially adding on some sort of JS framework—can create very powerful and completely portable user experiences in the browser. For our purposes we will use Angular 5 and Kendo—together they create a framework to handle a lot of the heavy lifting on the client side plus a nice set of Windows-like controls for our UI.

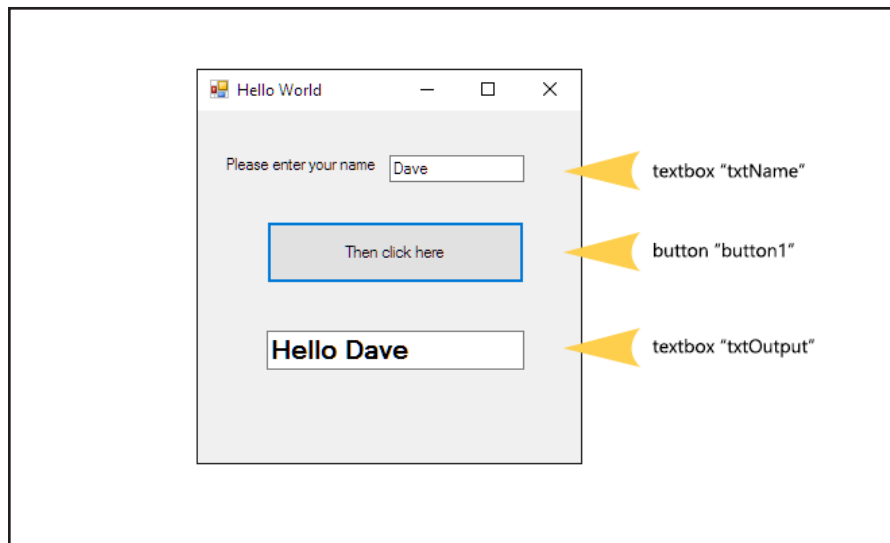
The back end of the app will still consist of C#, but we are going to use ASP.NET Core and construct a single page application (SPA) instead of a multi-page app. Single page apps differ from more traditional web apps in that they only have one web page (URL) that is refreshed dynamically. A "normal" web app switches from one page to another with each user action. SPAs rely on AJAX to allow the page to refresh without having to wait on a page load from the server (the "A" in AJAX stands

for "asynchronous" which means stuff happens without the user having to wait).

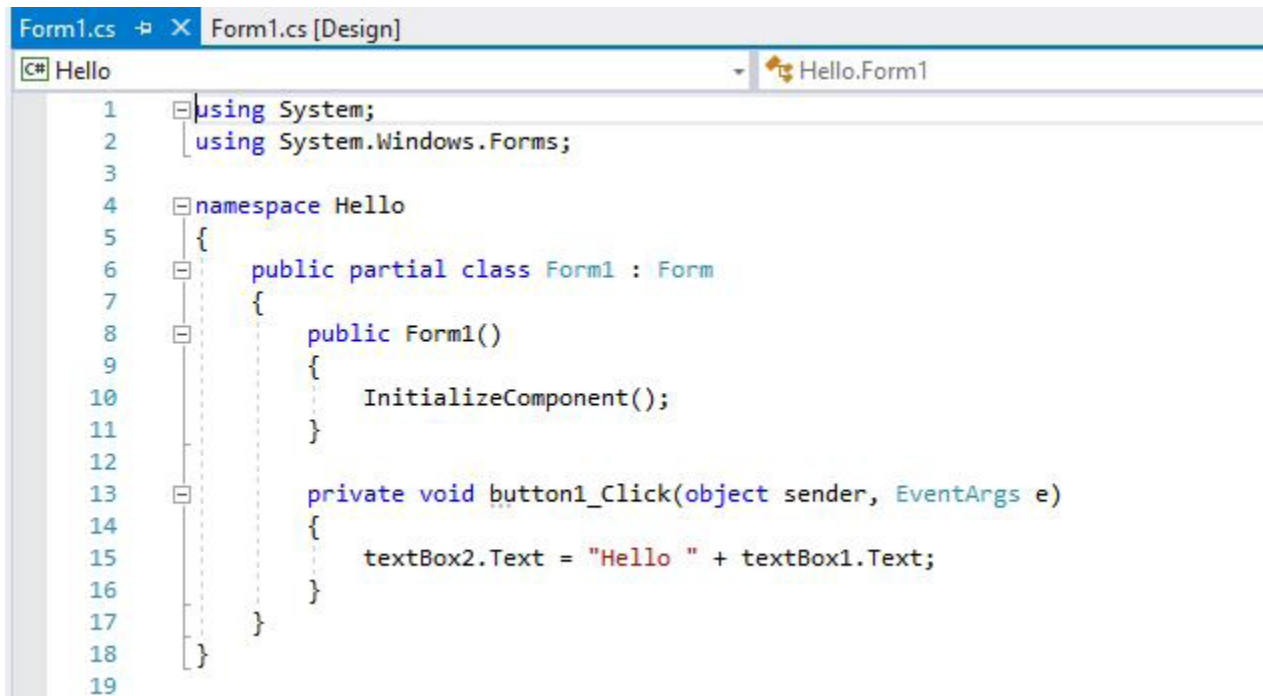
Server side code

Let's take the classic Hello World example program to see how WebMAP creates an web architecture. Using C#, let's create a simple form (form1) with a few controls.

See illustration:

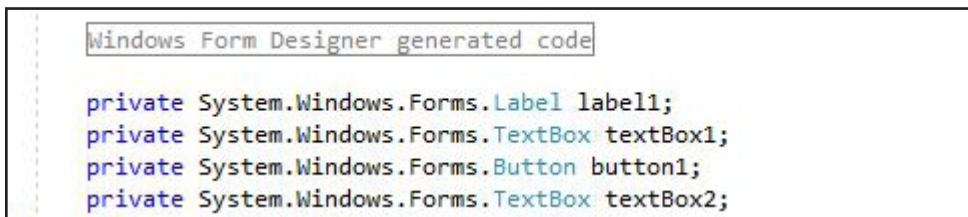


The code is super simple: after creating the form in Visual Studio and setting some basic properties on the lab and buttons. First we initialize the form, and then we handle the button click event.



```
1 using System;
2 using System.Windows.Forms;
3
4 namespace Hello
5 {
6     public partial class Form1 : Form
7     {
8         public Form1()
9         {
10             InitializeComponent();
11         }
12
13         private void button1_Click(object sender, EventArgs e)
14         {
15             textBox2.Text = "Hello " + textBox1.Text;
16         }
17     }
18 }
19
```

Notice in the classic C# code we are referencing user interface elements directly... look at the designer file:



```
Windows Form Designer generated code

private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.TextBox textBox2;
```

This is the very common "code behind" pattern. It's simple to write but can create some issues later. Patterns like MVP, MVC, and MVVM are all designed to eliminate the issues with code behind programming. And especially with a web application, coding practices since the days of Web Forms (classic ASP) have separated the code (ie logic) from the presentation (ie UI). Using the Microsoft stack, and Visual Studio, three common ways to write web applications are:

ASP.NET MVC

This pattern let's you interleave C# server-side code with HTML, so, for example, your HTML could call a C# variable. Simple example:

```
1      @{
2          ViewData["Title"] = "About";
3      }
4      @{
5          var aboutMessage = "This is the about page";
6      }
7
8      <h2>@ViewData["Title"]</h2>
9      <h3>@ViewData["Message"]</h3>
10
11      <p>Welcome. @aboutMessage</p>
12      |
```

When this code runs, the value of the variable `aboutMessage` will be inserted into the `<p>` element so the text the user sees will be “Welcome. This is the about page.”

ASP.NET MVC creates—assuming you use the Visual Studio app template—folders for your models (the data representation), Controllers (the traffic cop for routing URL requests), and Views (the presentation the user actually sees). By default the template does not create a Single Page Application, instead using URLs like `/index` and `/about` to return fully-built pages to the client (browser).

ASP.NET Single Page Application

The difference between a Single Page Application (SPA) and one that isn't, is that in an SPA we don't rebuild the entire view model with every request. This is great for actual applications (not web sites) where typically a request is to fetch some data used in part—but not all—of the page. Refreshing only that part of the page reduces the load on both the server and the client, as well as reducing the chattiness of the app. You can get a rich (but complex) template for these using ASP.NET Core with Angular, which has the added benefit of being server agnostic, so it can run equally well on Linux and Windows.

If there's any lingering doubt in your mind about how complex building well-behaved web apps can be, just look at what the empty shell of an ASP.NET Core app provides (note these are all things you would have to code anyway):

Hello, world!

Welcome to your new single-page application, built with:

- [ASP.NET Core](#) and [C#](#) for cross-platform server-side code
- [Angular](#) and [TypeScript](#) for client-side code
- [Webpack](#) for building and bundling client-side resources
- [Bootstrap](#) for layout and styling

To help you get started, we've also set up:

- **Client-side navigation.** For example, click *Counter* then *Back* to return here.
- **Server-side prerendering.** For faster initial loading and improved SEO, your Angular app is rendered by the server and then the browser where a client-side copy of the app takes over.
- **Webpack dev middleware.** In development mode, there's no need to run the `webpack` command. Updates are available as soon as you modify any file.
- **Hot module replacement.** In development mode, you don't even need to reload the page. Your Angular app will be rebuilt and a new instance injected into the page.
- **Efficient production builds.** In production mode, development-time features are disabled. The production build is a single JavaScript file.

Web API

The Web API template is really a way to create RESTful endpoints you can hit with HTTP requests using JSON. Web API is a simpler, faster, more lightweight method of moving data back and forth from a web server to a client compared to SOAP/XML web services. The Web API scaffolding doesn't really provide you much help in creating your client side code, event listeners, or look and feel. That can be done using classic Angular components, or React or Knockout or whatever you want. Plus HTML, CSS, and possibly some component library to create controls/widgets, such as Progress Kendo, or Syncfusion, or whomever.

Ok, so where are we exactly?

Right. It's complicated, isn't it? From our perspective, here you are with a desktop app (could be VB6, could be PowerBuilder, could be C# with WinForms), and you want a web version. Desktop apps are so 2000s. But to get that desktop app to a web version is hard. Sure, you could rewrite it into one of these patterns (or some other—we haven't touched Java, for example), but they are really complicated.

Well, cheer up Bucky. We've got exactly what you want:

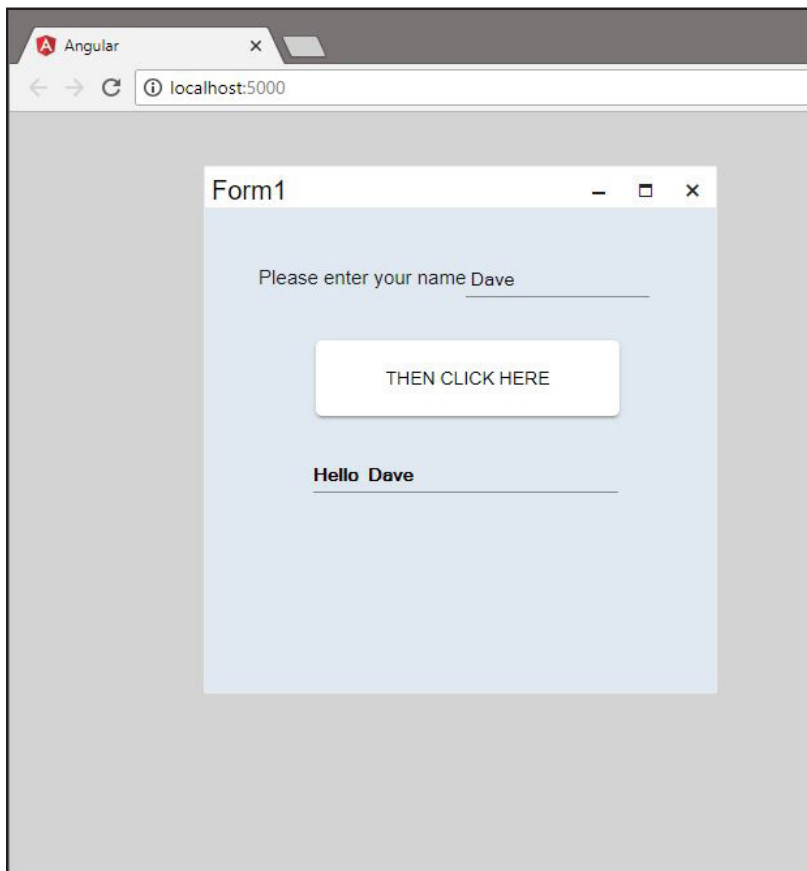
WebMAP to the rescue

The good news is that WebMAP solves most of your problems, by an enormous amount of cumulative cleverness from our development team.

- You can use coding skills you already have
- You can read the code
- You can add to the code without going crazy
- You can get a high-performance web app that's actually simple
- And we do it all with source code, not voodoo binaries.

Let's look at some code

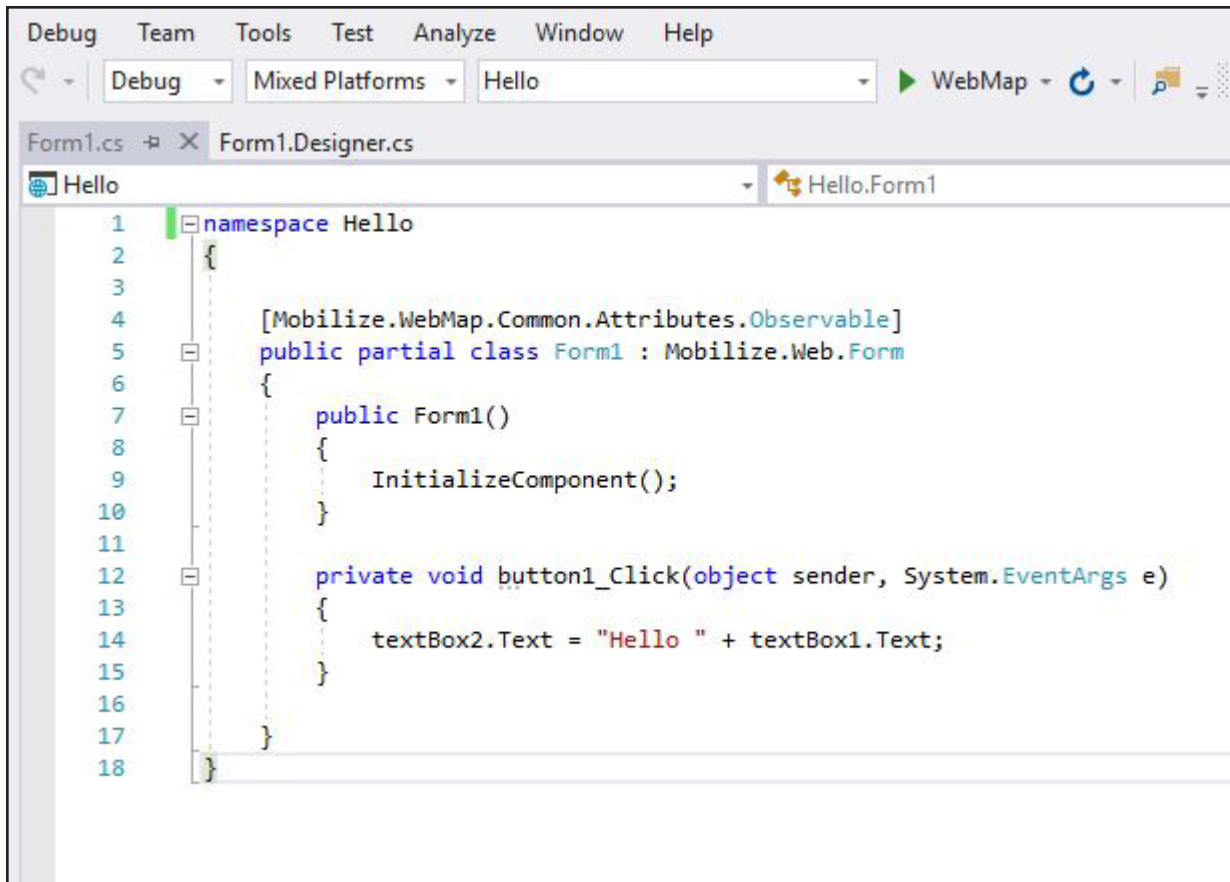
What if we migrated our HelloWorld app from part 1 to the web, using WebMAP?
What would that look like?



Which super-complex method, described in Part 1, did we choose to create this amazing web app?

None of the above.

WebMAP—like we said earlier—hides a lot of the complexity. Ok, basically all of the complexity. Let's look at the server-side code (remember a web app has to have a server component and a client that runs on the local browser):



If this code looks suspiciously like Winforms code, it's by design. In fact, there are only a few subtle differences between this code (which is now running on ASP.NET Core) and the original C# code (which ran on Windows desktop only):

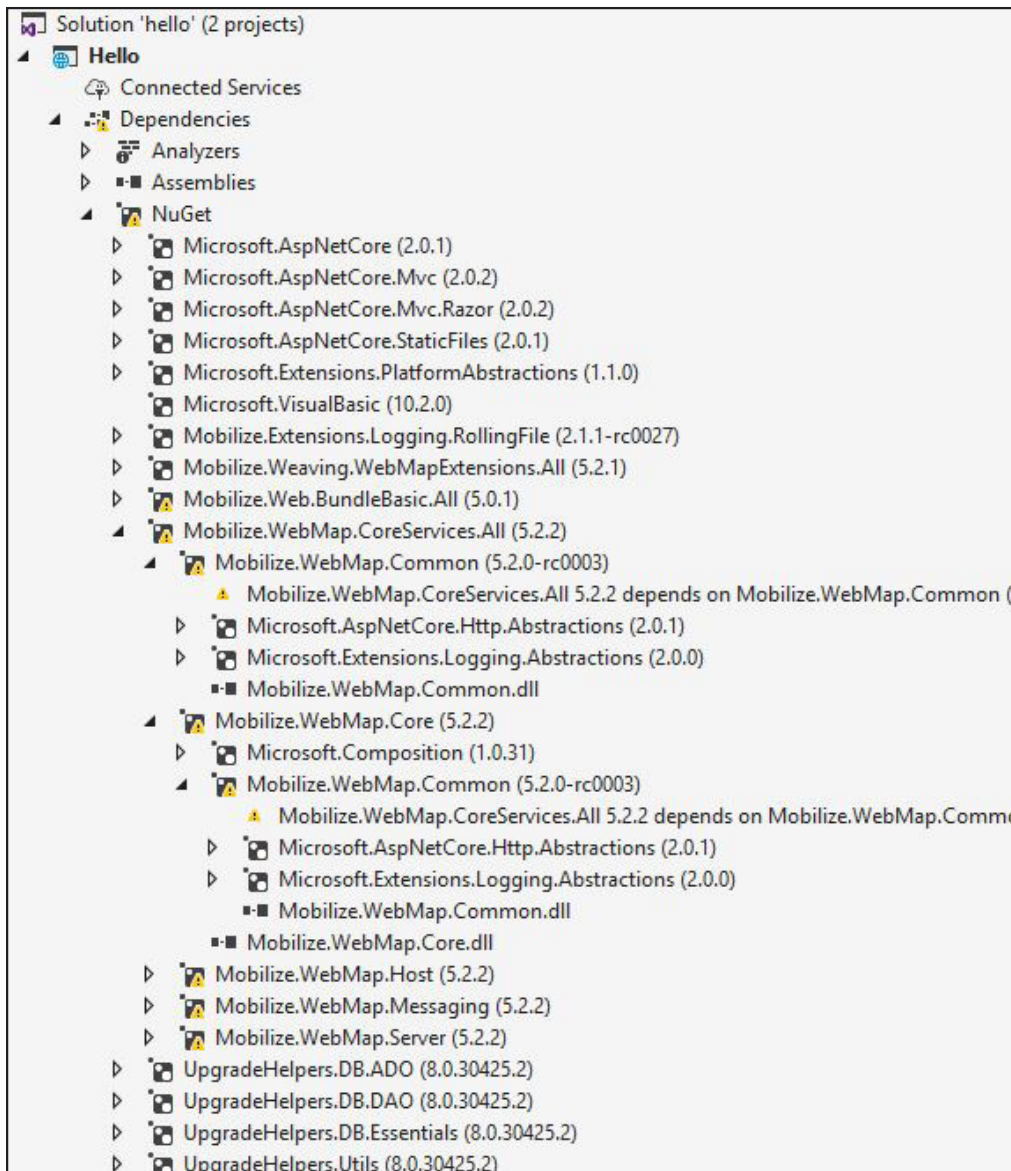
1. The Form1 class inherits from `Mobilize.Web.Form` (we'll get back to that in a minute)
2. The form declaration has an attribute: `[Mobilize.WebMap,Common.Attributes.Observable]`

3. There are no “using” statements.

That’s it. No controllers, no views or viewmodels, no models, just what appears to be almost exactly the same code that worked for a desktop app. How is this possible?

My favorite things...

I love packages: both the kind I get from Amazon and the ones that handle dependencies in my apps. Let’s look at the packages that our migrated HelloWorld app has:

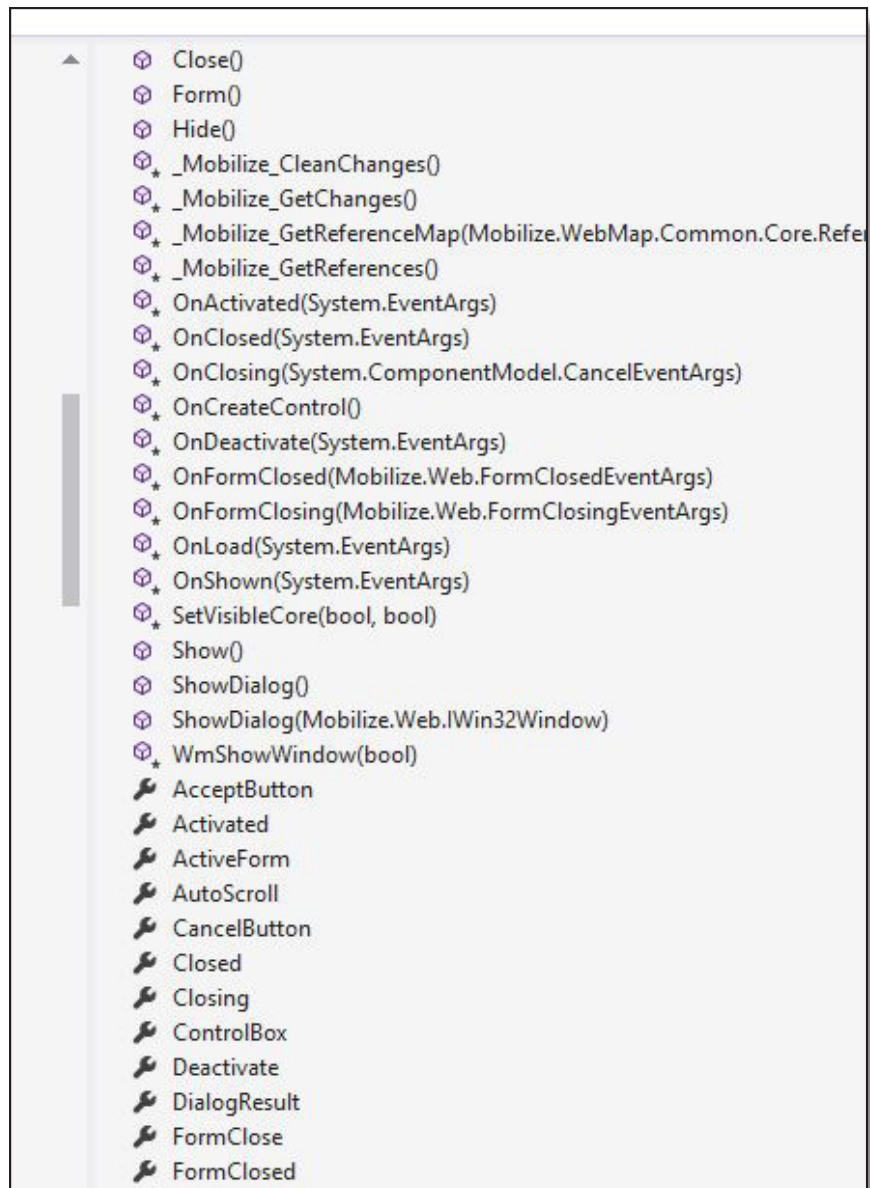


Ok, so this should make me pretty happy—lots of packages. And this is where the magic happens in WebMAP—these DLLs let the server side code be highly

decoupled from the web server housekeeping and the client side.

The Form class

Again—staying with the server side—the `Mobilize.Web.Form` class is used to instantiate all our forms, build the UI, and handle events. If we look at the Object Browser here's what we'll see:



Some of these methods and properties should look pretty familiar (`Show()`, `CancelButton`, etc) and some not (`_Mobilize_GetChanges()`). Remember, the goal of WebMAP is to migrate your Windows Forms apps to web, so we aren't going to have a bunch of properties or methods for classes that aren't representative of

what you find in Windows. (NB: WebMAP can support a variety of input formats, including non-Windows stuff like PowerBuilder. For the purpose of this blog, however, let's just focus on Windows Forms.)

Observable, Weaving, and AOP

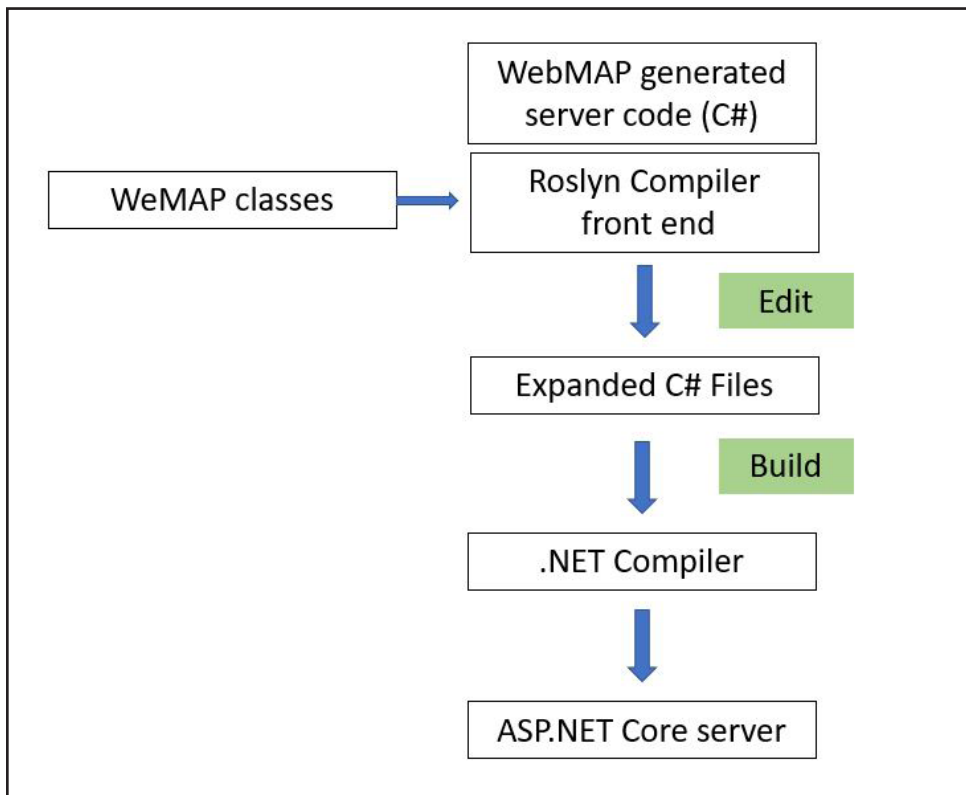
The other big change referenced above is the `Observable` attribute we see for the class `Form1`.

Now you've got to get your geek on.

The `Observable` attribute flags this class as one that will be monitored for changes. In short, when our app is running on the browser, we need a way to know that the user has done something—basically the same idea as “listening” for an event in the DOM in JavaScript. Our SPA we're building wants to send changes (ie state deltas) back to the server as JSON messages. An `Observable` object is something we monitor for changes.

Likewise, in our designer file (`Form1.Designer.cs`), we'll find a different attribute on the class: `[Mobilize.WebMap.Common.Attributes.Intercepted]`. In this class we create all the visual controls that will be on our form. The `Intercepted` attribute tells the Visual Studio compiler that this is a place to inject code.

You've probably used inversion of control (IoC) AKA dependency injection in code somewhere, right? Well, this is very similar. You may already be familiar with it: Aspect-oriented programming or AOP. AOP allows us to “de-clutter” the code you have to deal with from all the complexities needed to handle issues common to web applications. Like, for example, modality.



WebMAP takes advantage of the open-source “Roslyn” compiler platform in Visual Studio. Among other things, this platform offers real-time code parsing (which is how Intellisense works). WebMAP uses that capability to inject code from helper classes into “Intercepted” code and create new, expanded C# files. Since the compiler front end is running all the time, as you edit those user-facing C# files, these expanded C# files are always being changed as well. Press F5 to kick off a build and the compiler knows to use the expanded files—not the files you were editing—for the build process. Injecting these classes where necessary is what we mean when we talk about “weaving.”

Ok, that sounds really complicated. But it’s not really. So let’s get to it.

All the rest

Before we jump into the last installment, let's briefly review what we learned so far:

- Web apps are complex by nature
- Moving from desktop to web is also complex
- Web app development can require you to learn a bunch of new stuff, compared to Winforms apps
- WebMAP uses super cool rocket-powered tech to make it easy to move to the web
- WebMAP uses weaving so you can work on familiar code while the web complexity is hidden.

A few more words on weaving

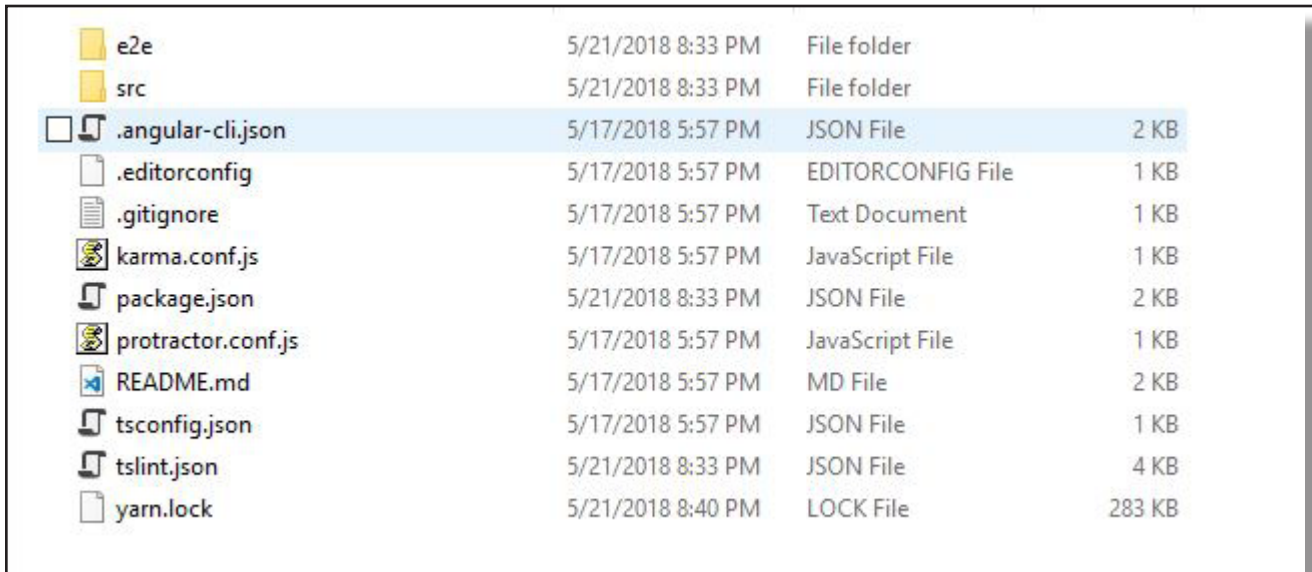
Ok, so about all that weaving. First of all, aspect-oriented programming (AOP) isn't a new concept—it's been around for years, although possibly more familiar to Java developers than C# devs. Tools like Postsharp have made code injection for C# easier, but the Roslyn compiler was really the game changer. Roslyn made AOP easier because you could rewrite the code before compilation as part of the build—rather than having to massage the IL after the compiler was through with it. The other thing Roslyn had to do to make AOP simpler was to make sure debugging wasn't hosed—you want to debug the original "undecorated" code, not the rewritten code.

It's beyond the scope of this blog post to dissect the rewritten code—you can see it for yourself in the `\obj` folder. I can tell you our simple `form1.cs` file grows substantially when all the attribute code is injected. Sure that code could be in the original source files (before injection), but it would require the same code to be duplicated over and over again. For example, every object in the UI needs code to track whether or not it has changed (ie is it "dirty"?). All that duplicated code in our sources would violate the Don't Repeat Yourself (DRY) rule. Instead, injecting it via weaving into our pre-compile C# files keeps the user-facing source code clean and uncluttered, while still allowing for really functional code that gets built.

Looking at the front end

That basically covers the back-end, or server side, of the code. What about the client, or front end?

Our app gets a bunch of files that will be familiar to front-end web devs who use Angular but maybe unfamiliar to winforms developers. What's in the folder?

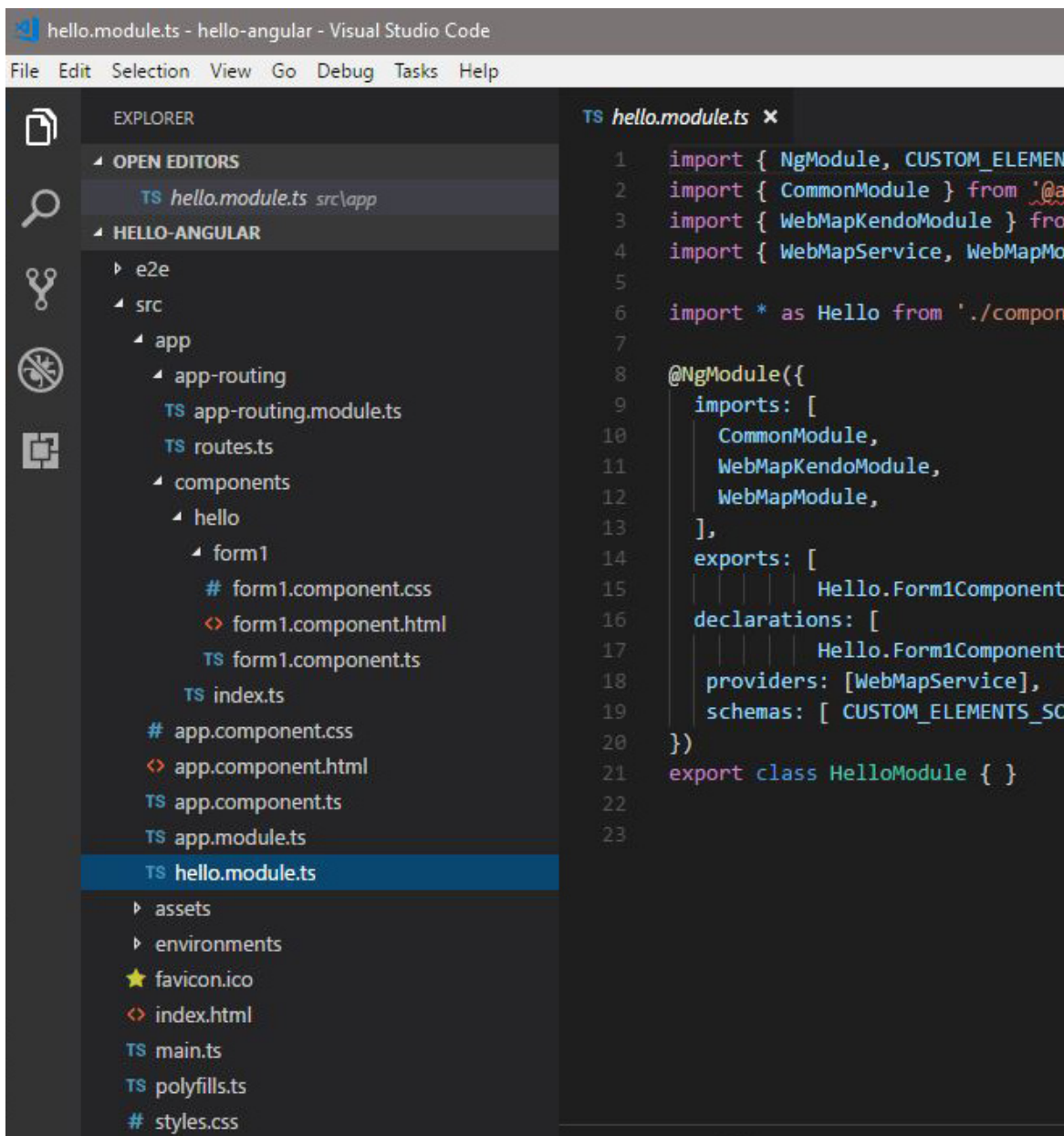


e2e	5/21/2018 8:33 PM	File folder	
src	5/21/2018 8:33 PM	File folder	
.angular-cli.json	5/17/2018 5:57 PM	JSON File	2 KB
.editorconfig	5/17/2018 5:57 PM	EDITORCONFIG File	1 KB
.gitignore	5/17/2018 5:57 PM	Text Document	1 KB
karma.conf.js	5/17/2018 5:57 PM	JavaScript File	1 KB
package.json	5/21/2018 8:33 PM	JSON File	2 KB
protractor.conf.js	5/17/2018 5:57 PM	JavaScript File	1 KB
README.md	5/17/2018 5:57 PM	MD File	2 KB
tsconfig.json	5/17/2018 5:57 PM	JSON File	1 KB
tslint.json	5/21/2018 8:33 PM	JSON File	4 KB
yarn.lock	5/21/2018 8:40 PM	LOCK File	283 KB

Most of these files you don't need to worry about—they're common to the Angular template. Karma and Protractor are testing tools—the e2e folder is for end-to-end testing files. tsconfig and tslint are for Typescript (which we use instead of JavaScript; Typescript compiles to JavaScript when you build). Yarn manages packages faster than npm. And so on.

The droids you're looking for are in the `\src` folder. Let's take a look.

The time has come to switch from Visual Studio 2017 to Visual Studio Code. VS Code is the free, open-source IDE from Microsoft and right now it just works a little better on Angular apps. For one thing, it comes with a built-in Powershell terminal so you can run Angular CLI commands like `ng build`.



Important folders

The folder we really care about is the app folder. The rest are:

- e2e is for the end-to-end integration tests (as opposed to unit tests).
- environments is for setting up your development and production environments (plus any others you want)
- assets for images, wav files, etc

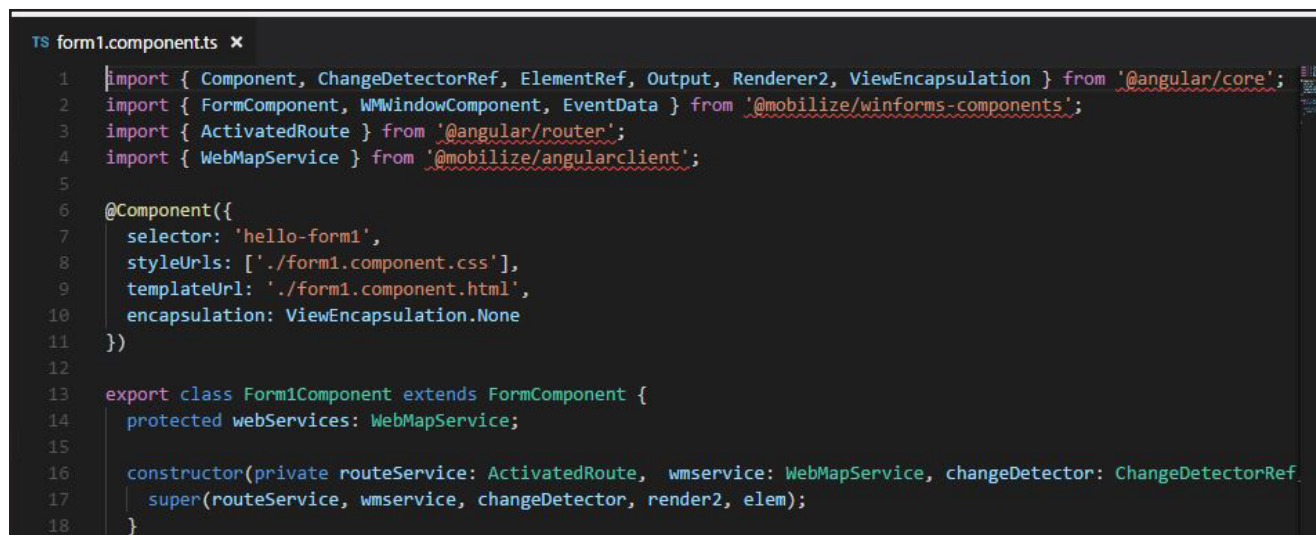
- Misc stuff like CSS, index.HTML, favicon, and more. Mostly this stuff we don't need to worry about. Index.html, for example, just loads the root component—it's not a rich web page like you might find with other templates.

In the app folder we will find our components, which is where the work is done. Each form from our Windows forms app gets its own folder under `\app\components`: in this case there's only `form1` so that's the only folder created: `src\app\components\hello\form1`.

Each form gets three files: a CSS file, a typescript file, and an HTML file. They're pretty simple:

Typescript files

The `.ts` file lists all the required imports to make the app work, defines this component, exports the class, and constructs all the standard pieces a WebMAP app needs to run:



```
1 import { Component, ChangeDetectorRef, ElementRef, Output, Renderer2, ViewEncapsulation } from '@angular/core';
2 import { FormComponent, WMWindowComponent,EventData } from '@mobilize/winforms-components';
3 import { ActivatedRoute } from '@angular/router';
4 import { WebMapService } from '@mobilize/angularclient';
5
6 @Component({
7   selector: 'hello-form1',
8   styleUrls: ['./form1.component.css'],
9   templateUrl: './form1.component.html',
10  encapsulation: ViewEncapsulation.None
11 })
12
13 export class Form1Component extends FormComponent {
14   protected webServices: WebMapService;
15
16   constructor(private routeService: ActivatedRoute, wmservice: WebMapService, changeDetector: ChangeDetectorRef,
17     | super(routeService, wmservice, changeDetector, render2, elem);
18   }
```

Style sheets and HTML

The remaining two files are super simple. The css file has a class for every HTML element, which is basically every control on the form:

```
TS form1.component.ts    # form1.component.css x
1  |.Hello_Form1 {
2    left: -1px;
3    top: -1px;
4  }
5  |.Hello_Form1 .Form1 {
6    width: 392px;
7    height: 372px;
8    overflow: hidden;
9  }
10 |.Hello_Form1 .textBox1 {
11   font-family: "Microsoft Sans Serif";
12   font-size: 14.4px;
13   left: 177px;
14   top: 43px;
15   position: absolute;
16   width: 141px;
17   height: 26px;
18 }
19 |.Hello_Form1 .label1 {
20   white-space: nowrap;
21   overflow: hidden;
22   left: 41px;
23   top: 45px;
24   position: absolute;
25   width: auto;
26   height: auto;
27 }
28 |.Hello_Form1 .button1 {
29   left: 85px;
30   top: 102px;
31   position: absolute;
32   width: 233px;
33   height: 58px;
34   padding: 0px 0px 0px 0px;
35   display: table-cell;
36   vertical-align: middle;
37   display: table-cell;
38 }
```

Note the absolute positioning of all the elements: it's necessary to make the form render in the browser like it looks on Windows.

It's not going to be a super-slick responsive web look and feel: it's supposed to be a close to the Windows user interface as possible so the users/customers will not need re-training when the app is deployed. Of course, if it makes sense for you, you can always jack the css later to make the app look however you want.

Finally, let's look at the generated HTML:



```

1.component.ts  <> form1.component.html x
<div *ngIf="model">
  <wm-window [model]="model" id="Form1" class="Hello_Form1">
    <ng-template let-model>
      <div class="Form1">
        <wm-textbox id="textBox1" tabindex="2" class="textBox1" [model]="model.textBox1"></wm-textbox>
        <wm-label id="label1" tabindex="0" class="label1" [model]="model.label1">Please enter your name</wm-label>
        <wm-button id="button1" tabindex="3" class="button1" [model]="model.button1"></wm-button>
        <wm-textbox id="textBox2" tabindex="4" class="textBox2" [model]="model.textBox2"></wm-textbox>
      </div>
    </ng-template>
  </wm-window>
</div>

```

Each control on the form gets its own HTML element tag—and they are all named "wm-[some control type]". The ids are set to the actual object name in our source code.

Where's Waldo?

Two things to notice in the HTML: there is no references to the Kendo framework, and there are no references to Angular-specific attributes.

Why not?

We like Kendo and Angular very much, And I'm sure lots of other folks do as well. But not everyone has made that particular lash-up their standard for web dev. So one of our design goals for WebMAP5 was to de-couple the generated code from the framework and the UI control set. We already know some additional control libraries will be implemented (stay tuned for announcements later this year) and at some point someone will build a case for using PrimeNG or React instead of Angular. In the past replacing either Kendo or Angular would have meant ripping out a whole lot of code and building multiple, parallel versions of WebMAP. With WM5 it only means changing some config files to use different packages (once the packages are built, of course).

Those packages hold the code that binds both Kendo and Angular to our components. They are source code, so there is no long-term dependency on Mobilize for your app. Over time we will be enhancing them and making those

enhancements available to our customers post-migration, so as they extend and maintain their new apps they can continue to use the architecture and code patterns we've created. Or, they can do something entirely different, like write more "normal" Angular components.

That has huge benefits for our customers—not only can they switch a previously-migrated app from one MVVM framework to another post-migration, but they will at some point have the ability to mix and match control libraries—possibly to implement a control from one library that isn't available in their default set.

Building the front end

Once the app has been migrated, we simply use the Angular CLI to install all the packages (using yarn or NPM) and then do `ng build`. That creates the `wwwroot` folder in the source tree. Then we can go back to the ASP.NET Core `sln` file and build and run it from Visual Studio 2017.

And it will run perfectly so we're done, right?

Not on your life.

Unless you are migrating something as trivial as this app, there will be work. Code that accessed the file system, or the registry, or attached hardware devices, or invoked the Win32 API directly (unmanaged code) will have to be re-worked. You can't just print from your app anymore; you'll have to invoke some kind of web service that can, for example, create a PDF, store it, generate a URL and hand that URL back to the user. The user can then download the PDF and save it or print it from a local resource like Acrobat.

Wrap up

Moving applications from desktop to web requires solving some difficult problems: state management, “impedence mismatch” between platforms, object lifetime, scalability, authentication, and more. Some of it can be simplified by using patterns like AOP. Migration of desktop apps to web using the WebMAP tooling and resulting architecture can dramatically cut the time, cost, and risk of modernizing legacy apps.