“Hello World” Tutorial for WebMAP

Transform classic Windows apps to web and mobile. WebMAP, the industry’s number one tool for mobilizing Windows applications, is available on-demand at studio.mobilize.net. Try it now.

Contents

Section 1: Introduction

WebMAP converts your C# application to HTML5, creating a web application that can be run locally or in the cloud.This tutorial walks you through a simple “hello world” application in C# that gets migrated to a modern web app with WebMAP.

Prerequisites:

- Windows 7 or 8.1
- Visual Studio 2013 with TypeScript 1.4 installed

Setup:

Open the zip file and extract it to a local drive. Notice there are two directories:

- **WF**: this is the C#/.NET/Winforms version of the app—it’s where we will begin
- **Web**: this is the app after migrating it with WebMAP. It’s where we will end up.

Sample files:

Use the sample files to run through the tutorial.

Winforms version:
http://www.mobilize.net/hubfs/Downloads/WinformsHelloExample.zip

WebMAP version:
http://www.mobilize.net/hubfs/Downloads/WebMAPMigratedSolution.zip

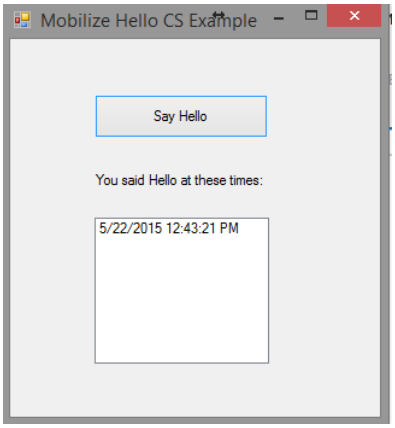
Section 2: Exploring the Windows Forms app

1. Open the **WF** directory and double-click the HelloExample.sln file to open it in Visual Studio.
2. Build and run the solution in Visual Studio.

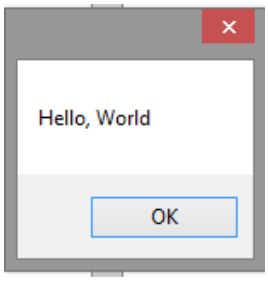
```
private void button1_Click(object sender, EventArgs e)
          {
                // Say Hello to user
                MessageBox.Show("Hello, World");
                // Fetch current time, add to listbox
                string now = DateTime.Now.ToString();
                listBox1.Items.Insert (0, now);

          }
```

This is a simple app that has four elements: a button, a list box, and a message box. The code associated with Form1.cs looks like this.



When you run the application, it looks like this.



When the user clicks “Say Hello” a message box is displayed with the words “**Hello World**”

And the date and time of the button click is added to the list box. Full stop. *Nada mas.* As simple as it gets.

Section 3: The Web is not Windows

Before we dive into our migration to the Web, let's talk through a few of the differences between a .NET application and a Web application. The fundamental differences between the two will account for most of the changes you see in the application source code after migration.

The single most jarring difference is that a Windows application runs on a single physical device; that is, the code, the objects, the processes, and the presentation are all on your PC[1]. The entire ecosystem of the application is tightly bound and highly deterministic. But on the Web applications run in dual environments: a Web server which is frequently hosted on a virtual machine perhaps in a public cloud like Azure or Amazon Web Services, and a browser which is on the physical device accessed by the application's user.

	Windows	**Web using WebMAP**
Memory	Local machine; private to process	Web server; shared by all simultaneous sessions
Storage	Local storage ("C:\"); built in API for easy access to read and write	No local storage; requires use of BLOB or DB for persistence
User configuration	Windows Registry	Nothing; must be created in DB
Simultaneous users/sessions	1	0 to many
Network required	None except when accessing network resources like DB	Always except in cases where app runs in localhost
OS	Windows Vista or newer	Server runs on IIS; app can run on any OS
OS access via API	Easy with PInvoke	Difficult to impossible
Hardware access	Easy with API	Not possible
User input	Known to be keyboard and mouse	Unknown; usually has to support keyboard, mouse, and touch
User experience	Rich set of controls from Microsoft and 3rd parties	Limited set of controls; usually requires 3rd party JavaScript framework for rich UI.
Development tools	C#, .NET, Winforms, DB	C#, ASP.NET, MVC, MVVM, JavaScript, JSON, AJAX, jQuery, KendoUI, HTML, CSS, Inversion of Control (Unity Framework).
Programmer mindset	And you thought this was hard!	This looks impossible (it's not actually)

Some conclusions to draw from this exercise:

- Creating a well-behaved, performant application to run on a browser is a more complex task than creating a Windows application
- The user piece is more constrained and less deterministic than in Windows
- There are more languages, patterns, and technologies you need to know about

[1] "Windows application" in this case means software written in C# using the Winforms designer and the .NET runtime. In reality a Windows app can be running on a variety of devices—PCs, Windows Phones, Windows Tablets, or embedded systems using development systems like WPF and Silverlight. However, since WebMAP currently only supports C#/Winforms we will limit ourselves in this discussion to apps written for the PC.

- If you were to start from scratch to build something like this it could be a daunting task
- *WebMAP makes a lot of this really simple.*

With that in mind, let's migrate our Hello World app to this Brave New World of the web.

Section 4: Migrating the app

Step 1: Create a New Solution

This is your portal to app migration to the web. From the home screen you can create a new solution to analyze and migrate; review existing solutions; or check out the demo apps.

1. If you haven't already, go to http://mobilize.net/register and create an account. There's no charge or obligation to do this.
2. Once you've activated your account, log into https://studio.mobilize.net.
3. We will start by clicking on New Solution.

Step 2: Download and Run the Assessment Wizard

Before we can perform a migration we need to run our assessment wizard on your code to see how ready it is to migrate to HTML5.

1. Click Download from the explanatory dialog box. (The Wizard uses ClickOnce technology from Microsoft.) In Chrome or IE, just run the installer. In Firefox, you have to save the installer and start it from Windows explorer.
2. You'll need to log in to the Assessment Wizard to connect your assessment to the right account.
3. Use the same credentials you used to log into studio.mobilize.net.
4. Now we need to point to the .NET solution we want to migrate to the web.
5. Click the browse button and then pick the HelloWorld.sln file in the directory where you unpacked it.
6. Change the suggested name if you wish; this is how the solution will show up in studio.mobilize.net.

The assessment wizard collects metadata about your .NET application and sends it to our secure Azure based analysis engine. We can tell you how big your application is in lines of code, which specific components and classes (APIs) you're using, and more. Inside the solution directory on your hard drive will be a folder called Assessment; inside this folder you can read the XML files showing what data is collected. Spoiler: it's not as exciting as *The Lord of the Rings*.

Once the wizard is finished, it will close automatically and re-open the studio session.

Step 3: Review the assessment results

At this point the assessment should be complete and you should be in a new browser tab showing the results. Here's what you need to know:

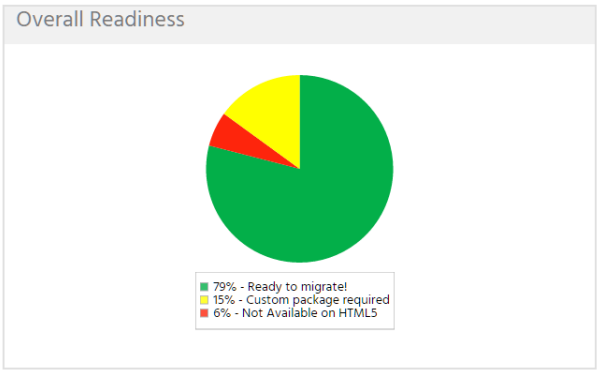


Figure 1: Assessment results for Hello World sample app

1. The first thing is how "ready" it is to migrate to HTML5. In this case we see that this app is about 93% green—this represents the amount of work you save through automation moving .NET classes, forms, controls, properties, and methods over to HTML5 UI.

2. The yellow on the pie chart represents properties, methods, and events that could be mapped to HTML but aren't automated yet. We track the occurrence of these across the universe of metadata so we can plan our product roadmap. If you have "yellow" PMEs that you would like to see get automated you have a couple of choices: you can wait until we notify you that we have now mapped some of your PMEs, or you can engage with us for a specific set of mappings to accelerate your migration.
3. The red area represents things that won't be mapped, things like accessing the local file system or Windows registry. These will require some level of redesign to get the equivalent functionality in a Web app.

SideBar: Why isn't Hello World all green?

It's true, something as simple as Hello World doesn't come up all greenish in our assessment.

What's going on here? Shouldn't all the code—all three lines of it—migrate to the web perfectly? (Remember that although we only wrote three lines of "logic" code, by creating a Windows Form Visual Studio created some design code to handle the instantiation and processing of the form itself.)

In a perfect world, yes, but in reality if we investigate our .NET code we will see some things that don't or can't migrate. This doesn't mean our app will be broken, however. In short, .NET does (by default) some things that either can't or currently don't migrate to HTML. Let's look at one example of these to understand the pattern:

- System.Windows.Forms.ContainerControl.AutoScaleMode is a property used by Windows to handle monitors with different resolutions; 96 dpi, 110 dpi, and so on.
- The purpose of the property is to allow Windows to try to make forms "look right" regardless of the native resolution of the monitor they are being rendered on.

- This functionality, which is no doubt cool, nevertheless doesn't exist inside a browser, where you have different approaches and solutions to scaling or supporting different form factors.[2]

In a large application, the few lines of code created by .NET that are not transferrable to the Web wouldn't represent a significant percentage of our code in this assessment report. Since our Hello World example is just 3 lines of code (plus all the generated design code) the pie chart looks worse than normal. For now, just ignore the yellow and red and let's go ahead with the migration.

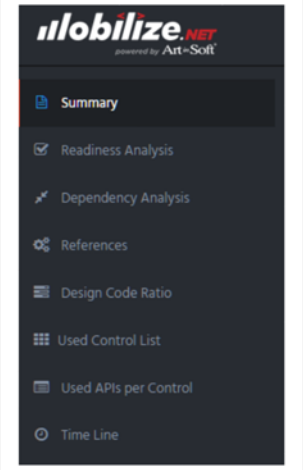


Figure 2: Assessment report options

We have other information here about your app:

- Dependency Analysis
- List of references used in all projects
- Breakdown of the ratio between lines of code used in forms designers vs logic files
- List of all controls used
- Which APIs are used for each control
- Chronology of this solution.

According to our analysis, this app should migrate easily to HTML5 with WebMAP. Let's do just that.

[2] For example, you can use Bootstrap to create a responsive design. But this tutorial will not cover that, as the version of WebMAP being described here doesn't use it.

Step 4: Configure the Migration Options

Before our migration can begin, we have to set a couple of configuration options.

1. From the Summary Page, click the "Migrate" button.

Generate fonts

- *Generate Fonts* will create a css file for each converted form's HTML file with font declarations taken from the font properties in your Winforms designer files. Set to "False" (default) the fonts are only declared in the site.css file. What does that mean exactly? Well if you use the default option WebMAP will create font declarations only in the site.css file; this means that your web app will use a basic set of default fonts, and whatever font properties the Windows forms had will be ignored.
- The font declaration from using the default on the default option (from site.css) looks like this:
- `font-family: Sans-Serif, "Segoe UI", Verdana, Helvetica, Sans-Serif;`
- This is a common way to handle fonts in a web page—picking a family and letting the browser pick from what's available.
- Setting "Generate Fonts" to True will cause font declarations to be created in each individual form's CSS file; those declarations will be identical to the ones used in the Windows forms designer from your app. Of course, when a browser tries to render the resulting HTML it may not find that font and will substitute a default font instead.

Code separation

- *Code Separation* lets you choose between having the least amount of JavaScript on the HTML client to keep the app small, or trying to minimize server roundtrips by creating JavaScript to handle form-side processing, such as simple calculations and field validations. If you select "Speed" WebMAP will attempt to determine which event handling code can be translated to JavaScript without incurring a server round trip, resulting in potentially a faster client-side experience but with more JavaScript to maintain.

For now, leave the defaults and click "Next".

Step 5: Upload Files

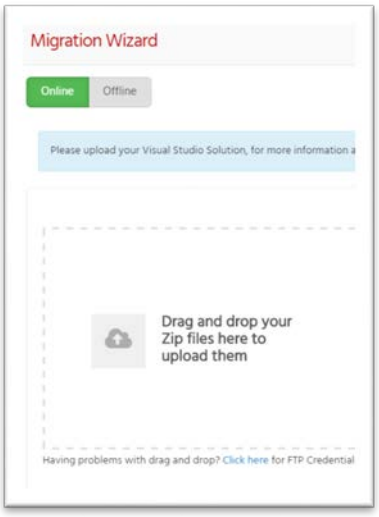


Figure 3: Drag and drop file upload

Up to this point we have only uploaded metadata about the app to the studio, not any source code.

1. To start the migration, first we have to create a zip file with all our app source code and then upload that to the migration system.
2. Switch to Windows Explorer and go to the directory where you unpacked the HelloWorld app. Select all the files and folders and put them into a ZIP archive. (Did you remember to build and run the app? WebMAP needs a \bin directory with a compiled version of your application in order to migrate it).
3. Grab the resulting .zip file and drag it onto the browser icon which has studio.mobilize.net open. Alternatively, click the icon in the "drag and drop" area inside studio and browse to the zip file. [3]
4. This starts the upload process to our secure Azure storage; once the files are uploaded we can start the migration to HTML5.
5. Click "Migrate" to begin the migration process.

[3] As you can see in the screenshot above, if you have trouble with the drag and drop functionality, you can get some one-time-only FTPS credentials to upload your files.

Step 6: Review the migration results

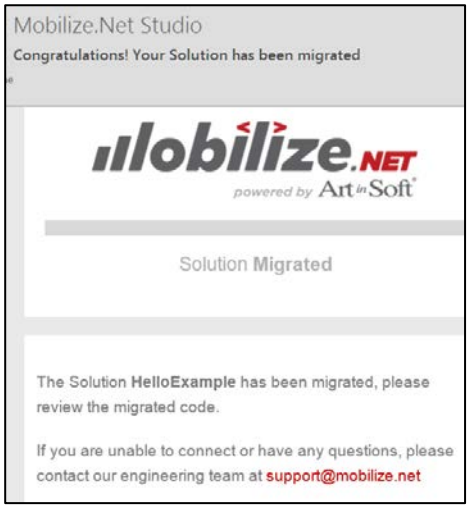


Figure 4: You'll get this email when your migration is finished

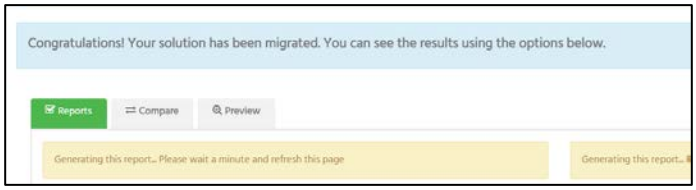


Figure 5: New tabs appear after the migration

The migration of your app runs in Azure as a web job. The time to complete depends on where your job is in the queue, how big it is, and other factors. Once it's queued, you can watch a progress indicator, or you can go do something else. When it's finished, you'll get an email.

1. Let's return to our solution at http://studio.mobilize.net.
2. You should now see two new tabs in the web page: "Compare" and "Preview:"

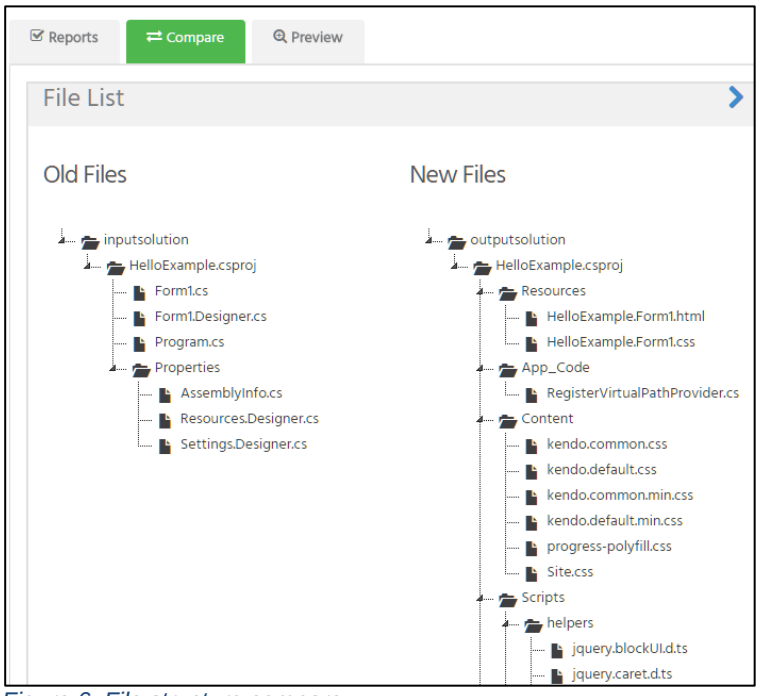


Figure 6: File structure compare

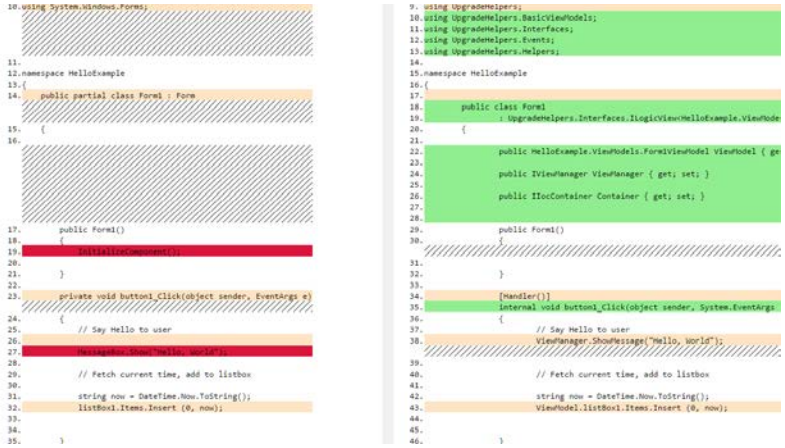


Figure 7: Code compare

3. Let's click on "Compare" to see how our app changed.
4. Here you see the original C#/.NET app on the left ("Old Files") and all the migrated files on the right ("New Files").
5. Click on the right arrow to expand the left-hand column.

6. Click on Form1.cs under "Old Files" and expand the right hand column to see the old and new source code for Form1.cs.
7. If you expand the left-hand column again and click on Form1.Designer.cs, you'll see how the Winforms designer file was converted to HTML5.[4]

[4] This is, as will be apparent in subsequent sections, a gross over-simplification.

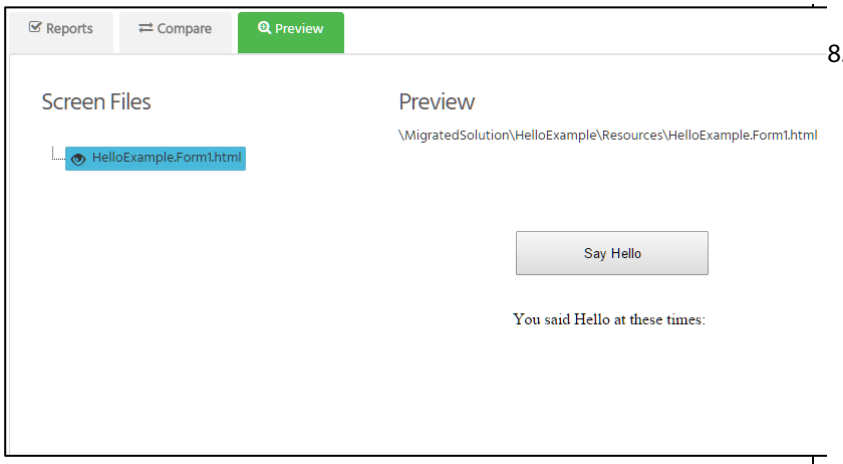


Figure 8: Preview screen

8. Click on the "Preview" tab and you'll see an approximate rendering of your form as HTML. It won't look perfect because many things are not created until the source code is compiled and executed.

Step 7: Explore the solution structure

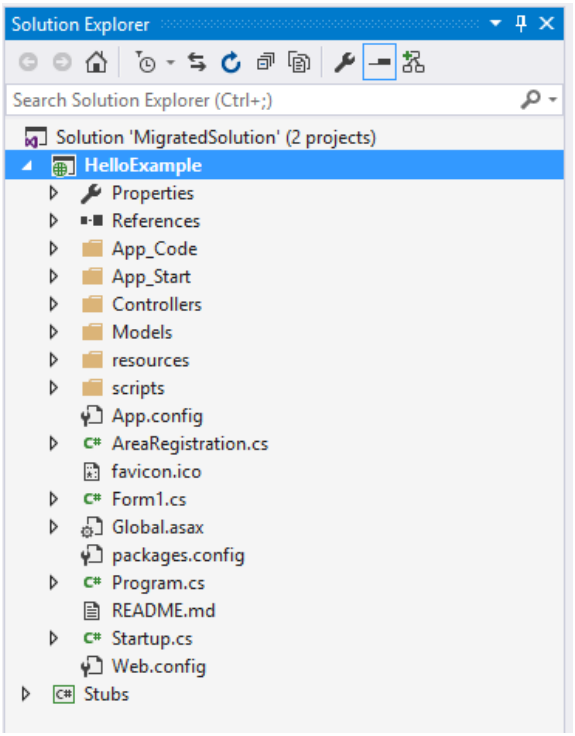


Figure 9: Project structure of migrated app

After the migration we get a newly architected application. Before it was Windows client-server. Now it's much more: ASP.NET MVC 4 Single Page Application with KendoUI.

1. Let's look at the source code and see what the changes are.[5]
2. Open the Web directory in the demo folder, then open the Migrated Solution folder.
3. Click the migrated solution (HelloExample.sln) file to open it in Visual Studio 2013.
4. WebMAP has created an ASP.NET/MVC4 single page application with some additional bits and pieces.
5. Let's look at the application's structure before we dive into the code:

Note: The application uses Nuget for package management. Your project depends on several Microsoft assembly libraries that are inconvenient for you to gather manually – items such as WebMAP helper classes, JQuery, Ajax, and the special build of the Unity dependency injection container that comes with ASP.NET MVC. This package manager ensures that all of these are downloaded to your development machine. If you care, you can open the packages.config file to see which ones it's fetching for you. Otherwise, forget about it; you shouldn't have to touch it during your project.

[5] Once the code has been migrated in http://studio.mobilize.net you can use the "Compare" view to read the source code, but you will only see a limited set of files. This is similar to the "Look Inside" view that Amazon.com uses for books. To get the source code, click the Buy button and follow the directions. Once the migration license has been purchased you will be able to download your migrated application.

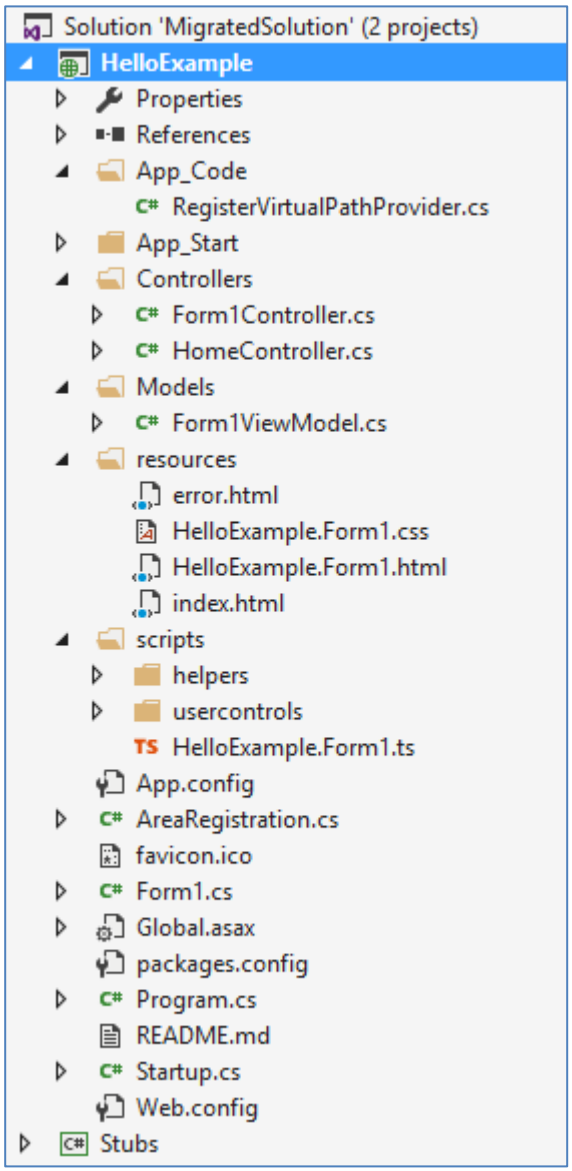


Figure 10: Expanded project files

Migrated Solution: If you expand the HelloExample project you will see a LOT of files and folders:
If you are familiar with the ASP.NET MVC architecture, some of these items are familiar, such as Controllers, Models, Views, Scripts, and so forth. Indeed, WebMAP generates the ASP.NET MVC template so you will even find some files from that pattern that aren't actually used by our program. You are probably wondering where the visual elements of your Windows Forms program have gone, in this case, the button that we used to invoke our "say hello" logic, and the listbox used to show the times.

Step 8: Review the HTML

```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    this.label1 = new System.Windows.Forms.Label();
    this.listBox1 = new System.Windows.Forms.ListBox();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(94, 56);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(192, 44);
    this.button1.TabIndex = 0;
    this.button1.Text = "Say Hello";
    this.button1.UseVisualStyleBackColor = true;
    this.button1.Click += new System.EventHandler(this.button1_Click);
```

Figure 11: Winforms designer code--not here anymore

In your Windows Forms program, these controls were created by C# statements in your InitializeComponent method in the Form1.Designer.cs file.

Where has this gone in our ASP.NET MVC program?

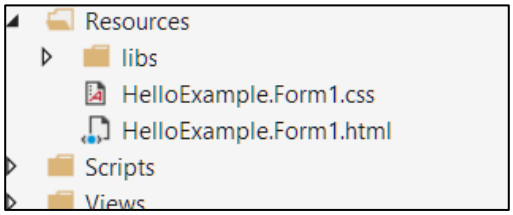


Figure 12: HTML and CSS files in migrated application code

1. We will find the descendants of this file in the folder \Resources. In this case, we see the files HelloExample.Form1.html and HelloExample.Form1.css

Figure 13:Form1.html listing

2. Excerpts are shown in Figure 13. These together are what make up the view.

```
<button id="button1" tabindex="1" class=" button1" type="button"
data-bind="events : { click : logic.button1_Click }, ">Say
Hello</button>
```

3. In the former case, we see the HTML that, when rendered into a modern browser, will produce controls similar to the ones that we had in our Windows Forms program.
4. We will see in later examples that this HTML, along with other information, gets sent from the server to the browser, and is rendered to resemble the Winforms original application.[6]

```
<link rel="stylesheet" href="HelloExample.Form1.css" />
```

5. You can see, for example, an HTML <form> element with a name attribute of "Form1", and a data-title attribute of the text that our original Windows form carried. Inside the <form> element, you see <button> and <label> elements that should look familiar. This HTML does not use a specific tag for a list box, but the data-role attribute tells the server to render the described data in that format.

[6] Remember this tutorial uses the KendoUI JavaScript framework, which, in addition to providing an MVVM architecture on the client side, creates a user experience (UX) similar to the original Windows application. While this might seem odd for a Web application, some developers would like to move their application to the Web with the minimum set of UX differences to avoid the need to re-train users.

Step 9: Review the CSS

HelloExample.Form1.css | Form1Controller.cs

```
1  .HelloExample_Form1 {
2  }
3  .HelloExample_Form1 .Form1 {
4      min-width: 409px;
5      min-height: 385px;
6      position: absolute;
7      overflow: hidden;
8  }
9  .HelloExample_Form1 .button1 {
10     left: 94px;
11     top: 56px;
12     position: absolute;
13     width: 192px;
14     height: 44px;
15     padding: 0px 0px 0px 0px;
16     display: table-cell;
17     vertical-align: middle;
18     display: table-cell;
19 }
20 .HelloExample_Form1 .label1 {
21     white-space: nowrap;
22     overflow: hidden;
23     left: 91px;
24     top: 135px;
25     position: absolute;
26     width: 195px;
27     height: 17px;
28 }
29 .HelloExample_Form1 .listBox1 {
```

Figure 14:The form-level CSS file

In the preceding HTML, you saw some, but not all, of the properties that you had set on your Windows Form C# control objects.

1. If you look carefully, you can find the height and width of the form itself, but not of the button or label or listbox in it.
2. These items are kept in the CSS style sheet, in this case named HelloExample.Form1.css[7], shown in Figure 15. We saw in the previous example how the HTML incorporates this code by reference.
3. The purpose of a cascading style sheet is to separate the styling of the HTML elements form the actual page markup, making it easier to standardize presentation across similar elements and to update them if necessary. In your Windows Forms code, when you wanted to change, say, the width of a button, you had to find a C# programmer, get him to do that work, and then recompile. With a .CSS stylesheet, a designer can simply edit the formatting instructions with a text editor. This approach was so successful that its concept (XAML vs. C#) was adopted for WPF.
4. In this example, you can see the placement (left and top attributes) and size (width and height attributes) of the various controls.
5. During the migration, WebMAP attempts to map the position and styling of the forms and controls from the .NET version to the HTML version of the application.

[7] When you migrate a Winforms application, WebMAP creates a separate CSS file for each form in your .NET code. These CSS files will be in the \Resources directory along with their associated HTML files—one of each for each form.

Step 10: The business logic

```
using System; //More namespaces used in actual code,
removed for clarity

namespace HelloExample
{
   public class Form1
       :
UpgradeHelpers.Interfaces.ILogicView<HelloExample.ViewMode
ls.Form1ViewModel>,
UpgradeHelpers.Interfaces.IDefaultInstances<Defaults>
   {
       public HelloExample.ViewModels.Form1ViewModel
ViewModel { get; set; }
       public IViewManager ViewManager { get; set; }
       public Defaults Defaults { get; set; }
       public IIocContainer Container { get; set; }

       public virtual string Text { get; set; }
       public Form1()
       {
       }

public void Init()
{
}

       [Handler()]
       internal void button1_Click(object sender,
System.EventArgs e)
       {
         // Say Hello to user
         ViewManager.ShowMessage("Hello, World");

         // Fetch current time
         DateTime  now = DateTime.Now ;

         // Add to listbox in view
         ViewModel.listBox1.Items.Insert (0, now);
       }
   }
}
```

Figure 15 Migrated Form1 Code

OK, there are my controls. Their instantiation and initial property settings are covered. Now where is the code-behind in which I wrote their behavior?

Your code has been modified by WebMAP. At the root of our solution structure, you'll find a file called Form1.cs, which is where the application logic for this form lives. (In the MVC world, this is called the Controller[8].)

Perhaps it is best to start with the part that is most familiar. If you look about halfway down, you will find the method button1_click. As before, this is the handler that gets called when the user clicks the "Say Hello" button. (Note: the event handler uses a routing mechanism based on the control's name, conceptually similar to the automatic wire up used in Visual Basic, so be careful to keep them the same.)

WebMAP has changed some of the code. First, we see that the previous call to MessageBox.Show() is gone. It has been replaced by the new ViewManager object, which is defined in the \Models folder. This is how your logic code now interacts with windows that appear on the browser. In this case, the method ShowMessage() does more or less what MessageBox.Show() did, generating the HTML to display a pop-up box on the browser.

Wait a minute, where the heck did this ViewManager thingie come from? The answer is that it is injected into the logic class as part of its construction process. You can see that ViewManager is a get/set property of class IViewManager. The helper classes will set this property as the logic object is created. The injection process is discussed in more detail in later sections.

After saying hello to the user, we now get the current date and time. This class has not changed. We now want to display it in the Listbox, as we had previously.

[8] Not to be confused with the Form1Controller.cs file in the \Controllers which has a different function.

The Listbox has moved. It is no longer a member of the logic class. It has moved to be a property of the ViewModel class, which handles its persistence over multiple ASP.NET web page calls. We will be discussing the ViewModel class in more detail in later sections as well. The Listbox object still works the way it used to, it just has moved to a different location.

```
        public HelloExample.ViewModels.Form1ViewModel
ViewModel { get; set; }
```

How did we get this ViewModel? Look at the first line of code in the class; it is another get/set property accessor.

The helper classes inject the ViewModel as well when the logic class is constructed. The logic class has a method called Init() method. This is conceptually similar to the Form Load event you would see in Winforms. It means that the object is not just instantiated, but that all of its dependencies are in place, and the view is now open for business.

Section 5: Overview of the ASP.NET MVC Environment

Concept 1: Code Running in a Context

The most important thing to understand about the application architecture generated by WebMAP is that your application's code is running inside a context. By this, I mean that your object is accompanied by helper classes that provide services to accomplish infrastructural tasks. There are a number of new pieces in your WebMAP program that you probably haven't seen before. Let's see how they all get stitched together.

You probably first saw this context-based approach back in the COM+ days, starting around 2000. Your small standalone COM object got plugged into the new COM+ context, and lo and behold, it could participate in complex database transactions. In today's world, you can think of ASP.NET as a context within which your objects run, providing the infrastructure needed for running in a web server. You can access that context directly via the `HttpRequest.Current` and `HttpResponse.Current` objects. You can think of WebMAP's helper classes as another context running inside ASP.NET.

Figure 16 below shows the WebMAP helpers and their relation to the other parts of the ASP.NET stack. Don't get too hung up on the individual pieces just yet. They're shown here now so that you have some notion of the sorts of things they are doing for you, and have a place to refer back to as your new knowledge unfolds. The client-side items that live on the browser are on the left-hand side of the diagram, while the server-side items are on the right. Your code will run almost entirely on the server side.

The items shown in light blue come primarily from ASP.NET and its MVC framework. The WebMAP helper classes were built to take advantage of their services, mediating them on behalf of your migrated code. You encounter and work with them in two different kinds of ways.

First, you may interact with them directly. The services you are most likely to use in this manner are shown in orange on the diagram. For example, you will use its dependency injection capability, the `IIocContainer` interface, to locate and instantiate other objects in the framework, and the `IViewManager` service interface to control the display of pop-up dialog boxes. The sample application will illustrate this way of interacting with WebMAP's helper classes.

Second, and more interesting, are the things that they do on your behalf behind the scenes, where you don't usually notice. These services are shown in green in Figure 17. For example, the `IStorageSerializer` service automatically serializes many of your server-side objects into and out of ASP.NET's session state. Your ASP.NET objects thus appear to your code as if they had long-running stateful lifetimes as your Winforms objects did, while behaving nicely in ASP.NET's environment by not consuming resources between calls. Later in this document, we will discuss how to use this background functionality from an application programmer's standpoint.

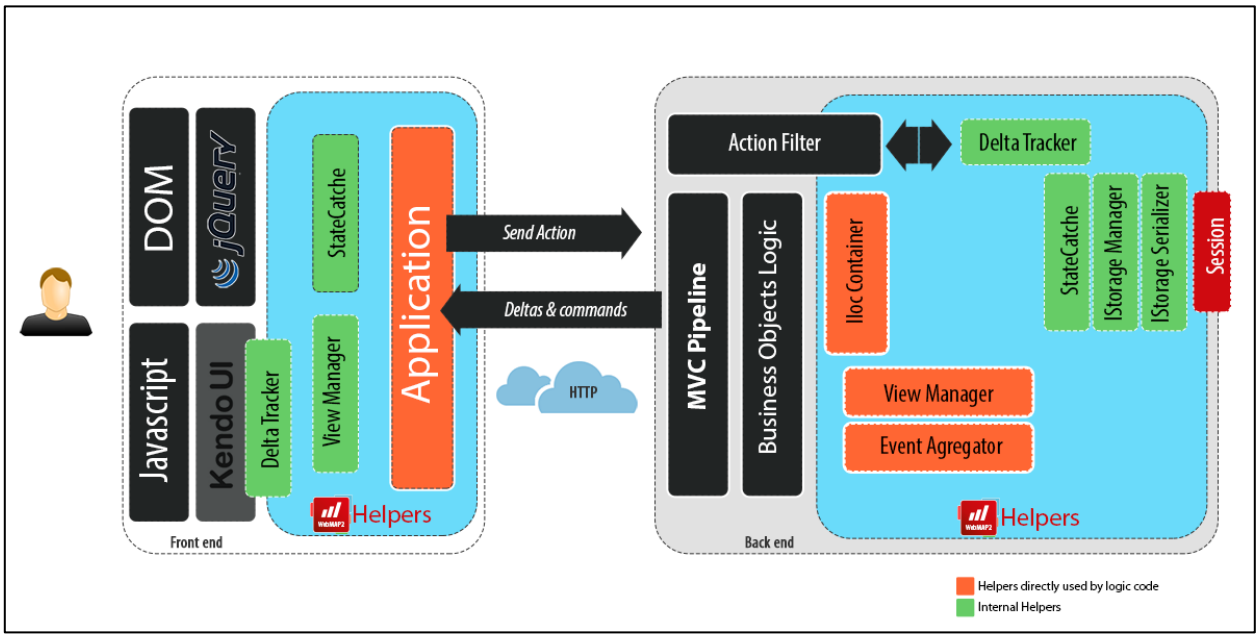


Figure 16: WebMAP application architecture

Concept 2: Overall Sequence of Events

Before discussing each component, let's review the overall sequence of events. You can then dig deeper into the pieces you most care about. Figure 17 shows the events that happen from start to finish when the user clicks the "Say Hello" button.

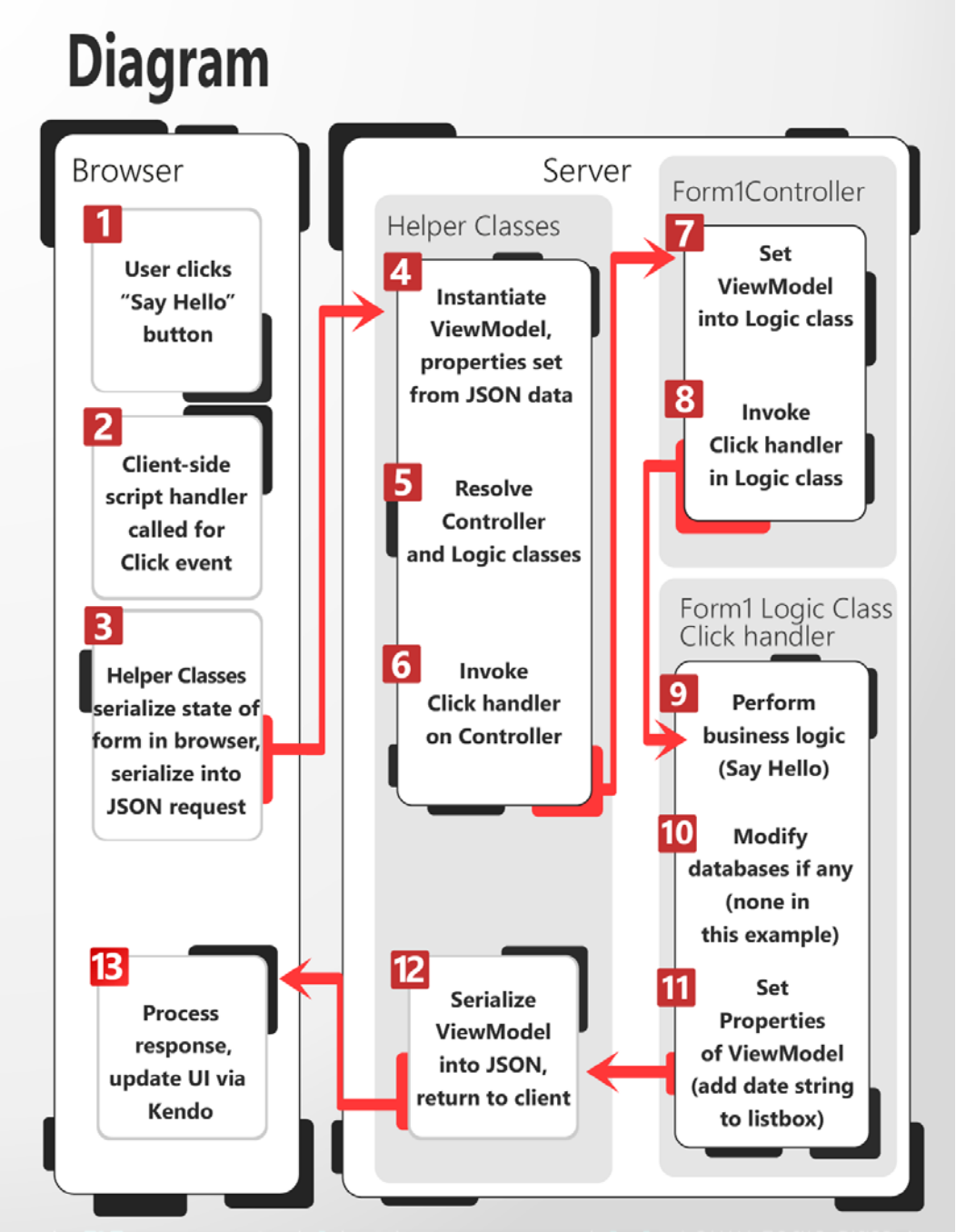


Figure 17: Event flow

The button click (1) triggers the handler function in the client-side scripts (2) that the helper classes provide. The helper classes then serialize the state of the browser's form into a packet and send it to the server as a JSON object over HTTP (3). Our focus now shifts to the server side.

On the server, the WebMAP helper classes are invoked by ASP.NET's MVC mechanism. These classes de-serialize the JSON packet containing the state of the browser's form, instantiate the ViewModel object (in

this case, of class `Form1ViewModel`, discussed later in this section), and set its properties so that they represent the current state of the browser's form (4).

The helper classes now instantiate the logic object (in this case, of class Form1), and the logic object's Controller object (in this case, of class Form1Controller, discussed later in this section) (5).

The controller has a handler for each event that occurs on the form. In this case, the only one we have is for the button1_click. The helper classes now invoke this handler on the controller (6). The controller places the ViewModel into the logic object (7), and invokes the click handler on that object (8):

```
// Form1Controller.cs

public System.Web.Mvc.ActionResult
button1_Click(HelloExample.ViewModels.Form1ViewModel viewFromClient, object
eventSender)
     {
       logic.ViewModel = viewFromClient;
       logic.button1_Click(null, null);
       return new UpgradeHelpers.WebMap.Server.AppChanges();
     }
```

You are now in your logic object's click handler, the migrated code that you saw previously in Figure 6. This is the code that the migration tool produced from your Windows Forms code. It should be familiar to you. This is where you perform your object's business logic (9), doing whatever it is that you want your app to do:

```
// Form1.cs

[Handler()]
    internal void button1_Click(object sender, System.EventArgs e)
    {
       // Say Hello to user
       ViewManager.ShowMessage("Hello, World");

       // Fetch current time, add to listbox

       string now = DateTime.Now.ToString();
       ViewModel.listBox1.Items.Insert (0, now);

    }
```

In this case, the business logic simply pops up a message box saying "Hello, World." In the world of .NET we would simply call `MessageBox.Show("Hello World")` and all would be good. But we're not in Windows anymore, Toto. Instead we have a JavaScript framework on our browser (KendoUI) and it has its own methods for creating a MessageBox-like object. So Form1.cs invokes `ViewManager.ShowMessage("Hello, World")`.

Remember that we have an MVC architecture on the server side? We also have an MVVM architecture on the client side. So in this instance neither the server nor the client need to know **how** the `ShowMessage()` method will be resolved. It's only when the view is created from the ViewModel in the browser that some actual widget is invoked that can display a message. This means that the server code is completely

uncoupled from the presentation, making it both easy to test and also to modify. For more about this, check out articles and tutorials on both Model View Controller (MVC) and Model View ViewModel (MVVM) on MSDN, StackOverflow, or Code Project.

The business logic for this sample app is trivial; a larger, more useful app might have some sort of database manipulation to perform, perhaps adding or modifying a record based on the user's input. You would probably do that right here (10). Finally, your logic code makes whatever modification it needs to the form's state based on its business logic. As you've seen earlier, this gets done through the ViewModel. In this case, we add another entry to the list box of the time at which the user said "Hello" (11).

Your logic handler now returns. Your code is finished; you are back in the arms of the helper classes. They serialize the changes to the state of the ViewModel[9] back into a JSON object (12), and return it to the originating browser, which takes the new state and renders it accordingly as HTML with some jQuery goodies (13): since the KendoUI JS framework binds the UI to the model, the UI (presentation) is automatically updated. That is one round-trip cycle.

Concept 3: The ViewModel

You saw your logic code interact with something called a ViewModel. What the heck is that?

The ViewModel is an object that manages the state of your view from one web POST operation to the next. In Winforms, your code would create an object such as the Say Hello button. That specific object instance would live on your local PC, maintaining its state internally, staying alive until you were finished with it.

This direct long-running connection to specific object instances isn't possible on the Web. Internet apps work via page posts and requests, or JSON calls to the web server. The object instances don't survive on the web server from one call to another; that would kill any sort of scalability.[10]

However, a client usually makes several related web calls while performing any sort of business operation. In this case, we'll probably say hello three or four times, and want to see those times in our listbox. To provide the client with logically seamless service over these related calls, a web server maintains a session, a per-user cache of state that retains its values for a limited time. Objects needed to service a web call are re-instantiated on demand, and restore their internal state from the session state.

ASP.NET provides a mechanism for handling this session state. Historically you had to write code to explicitly move your web objects' state into and out of this session state, which was time-consuming and brittle. The helper classes provide automatic storage and retrieval of your ViewModel's public properties in this ASP.NET session state, meaning less work for you.

---

[9] A certain amount of magic and cleverness are involved here in order to minimize the amount of "chattiness" between the client and server. In brief, the server side attempts to only send the actual changes back to the client viewmodel. Discussing how that works is beyond the scope of this tutorial.

[10] Basically consider that every time a web page sends a request to a server, the server has to think "Do I know you?" WebMAP uses Inversion of Control (IoC) via Microsoft's Unity Framework to manage some persistence of state on the server side. Among other things, it attaches a unique session ID to each web client running the application.

The ViewModel's code is shown below. It was generated during the migration process. There's not much for you to see or do, which is the main point here. Most of this work is done for you behind the scenes.

```
// Form1ViewModel.cs

public class Form1ViewModel
       : UpgradeHelpers.Interfaces.IViewModel
{
       // ViewModel's required infrastructural members

       public string UniqueID { get; set; }
       public virtual string Name { get; set; }
       public virtual bool Visible { get; set; }

       public virtual void Build(UpgradeHelpers.Interfaces.IIocContainer ctx)
       {
              this.listBox1 =
              ctx.Resolve<UpgradeHelpers.BasicViewModels.ListBoxViewModel>();

              this.button1 =
              ctx.Resolve<UpgradeHelpers.BasicViewModels.ButtonViewModel>();

              Name = "HelloExample.Form1";

              < business logic code omitted, see next section >
       }

       // ViewModel's other standard properties

       public virtual string Text { get; set; }
       public virtual bool Enabled { get; set; }


       public virtual UpgradeHelpers.BasicViewModels.ListBoxViewModel listBox1 {
get; set; }

       public virtual UpgradeHelpers.BasicViewModels.ButtonViewModel button1 { get;
set; }
}
```

Example Code ViewModel Class

Our sample ViewModel implements the helper class interface IViewModel. This interface requires the object to implement the properties Name and Visible that you see. Its base interfaces add the UniqueID property and the Build method. These are required for automatic state management. [11]

When ASP.NET starts a new session for a client, it calls the Build method in your ViewModel. This is where you do whatever initialization you need to do at the start of the session. In this case, the ViewModel instantiates the helper class objects that represent the data stored in the ListBox and the Button[12]. It does

[11] Note that the ViewModel is a si*mple POCO object, with no business logic. If you start writing IF or CASE statements in this file, you're probably not following the MVC pattern.

[12] If you look at Program.cs—which is the main entry point for the program—you will see `Container.Resolve` instead of `New`. This is necessary to inject the state management into the application.

this via the injection container, hence injecting the dependencies of these objects. It places these helper class objects into properties of the ViewModel. The value of the property Name is also set. The Build method is only called at the beginning of each client session. In that sense, you can think of it as your session constructor.

That is all you have to do. When you make changes to these properties (adding an item to the listbox, changing the name string), the helper classes automatically serialize them into your ASP.NET session state. When the next request comes in from the client, causing the ViewModel object to be re-instantiated, their state is automatically fetched from ASP.NET and deserialized back into the objects. If you've ever worked with ASP.NET session state, you will appreciate what a huge advance this is.

An ASP.NET session always has a timeout value (default, 20 minutes). After that amount of time, the session state is dumped. In this case, the Build method does get called when the client makes its next call. It is up to you to determine which data should live in the volatile session state, versus which belongs in persistent storage in a database.[13]

Concept 4: Controller Classes

WebMAP uses ASP.NET's Model View Controller (MVC) architecture. This requires the provision of a controller class. The purpose of this class is to be a piece of glue, if you will, to accept the incoming requests from the browser and route them to the correct method calls on the correct objects on the server. You do not need to write a controller class. It is generated by WebMAP when you do the migration. Nevertheless, it is instructive to examine this code, to see what it is doing behind the scenes on your behalf. You will find it in the Controllers folder of your migrated project. Code for Form1Controller.cs shows the code for this class.

Look at the bottom of it first. You will notice that the controller is tied to a particular logic class. This class is instantiated and injected into this controller by the helper classes when a request comes from the browser. That's part of step 5 of Figure 18 (Event Flow). The controller contains a method for handling the click on button1. In a more substantial program, there might be more controls capable of causing a post. In this case, the controller would have a handler function for each of these.

After instantiating and initializing the controller object, the helper classes call the click handler method. Let's look at this a moment. The controller is passed the ViewModel object as a parameter. It sets that ViewModel into the logic class, then calls the logic class's click handler. Finally, the controller's handler returns a value called AppChanges. If you drill into this, you will see that this is a serialization of all the state that has changed in the ViewModel in response to the logic class's operation. This is what the browser needs in order to show the correct state of its form.

```
// Form1Controller.cs

public class Form1Controller
  : System.Web.Mvc.Controller
```

[13] This tutorial is based on a trivial sample app; in a "real world" application typically you begin with user authentication—one assumes you have an existing relationship with whomever is using the application. Also typically you would hold some specifics in persistent storage (ie a database) rather than just in session state. The latter is useful for streamlining the actual session when the application is in use, but useless for maintaining anything interesting between sessions. These things are not demonstrated here, however most of that code should just migrated straight across from C#/.NET to ASP.NET using WebMAP.

```
{

public System.Web.Mvc.ActionResult
button1_Click(HelloExample.ViewModels.Form1ViewModel viewFromClient,object
eventSender)
  {
     logic.ViewModel = viewFromClient;
     logic.button1_Click(null, null);

     return new UpgradeHelpers.WebMAP.Server.AppChanges();
  }

  private HelloExample.Form1 mLogic ;

  public HelloExample.Form1 logic
  {
   get { return mLogic; }
   set { mLogic = value; }
  }
}
```

Controller Class

Concept 5: The View

Let's now discuss the view on the browser side. Remember, we are moving to the web from a desktop client application. Users who are accustomed to the desktop client are expecting some sort of form in front of their eyes that more or less stays where it is. They aren't expecting a lot of jumping around, as most basic web applications do, by posting pages back and forth and generating new ones.

To meet these expectations, WebMAP uses the strategy of generating a single page application. As we saw previously, rather than post the pages back and draw new ones, the single page app uses JSON calls to make changes in place, as the server-side components alter its contents in response to user input.

The helper classes generate a number of client scripts that run on the browser side. When you migrate the code with WebMAP, you are offered a choice of more client-side scripting or less. It is not unusual for developers to experiment with both approaches and choose which works best for them.

Section 6: Conclusion

When migrating a Winforms app to the mobile device platform, time is of the essence. WebMAP from Mobilize.Net provides a solution that migrates your application as quickly and easily as possible.

Now that you've gone through the tutorial, you have a pretty good idea of how WebMAP works and what it can do for you. If you want to take the next step, you can check out the sample applications to see more robust code running. Or, you can upload your own projects and take WebMAP for a free test run.

If you have any questions or require assistance, we have a posse of migration architects who are happy to help. Email us at info@mobilize.net or call us at 425-609-8458.

Section 7: Resources

WebMAP is built on common development standards such as MVC and MVVM. Understanding these concepts will help you to understand the code that WebMAP generates. Here are some resources to help:

- Model-View-ViewModel (MVVM) Explained by Jeremy Likness
- Understanding the basics of MVVM design pattern, MSDN
- A Really Simple Explanation of MVC by Craig Strong
- Introduction to ASP.NET MVC (Slideshare) by LearnNowOnline