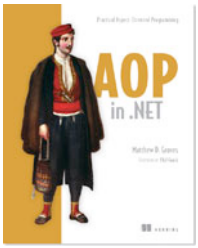

Using AOP to Detect Mobile Users

By Matthew D. Groves, author of *AOP in .NET: Practical Aspect-Oriented Programming*

Aspect-oriented programming produces clean code that's easier to read, less prone to bugs, and easier to maintain. More specifically, AOP can help you to solve problems in your .NET projects. For example, ASP.NET has an `IHttpModule` *that you can implement and set up in web.config. Each module runs for every page request to your web application. Inside an* `IHttpModule` *implementation, you define event handlers that run at the beginning or at the end of requests (*`BeginRequest` *and* `EndRequest`*, respectively). When you do this, you're creating a* boundary *aspect: code that's running at the boundaries of a page request. In this article, based on chapter 4 of* AOP in .NET*, the author shows you how to write boundary aspects for mobile browser detection.*

With the increase of smartphone market share came an increase of mobile users, which means that it's important to provide a good experience to both traditional desktop browsers and mobile browsers. Offering a native mobile application, alternative mobile site, or at least a mobile theme improves the mobile user's experience, and (hopefully) maintains the same level of functionality.

These days, using the standard desktop layout rendering can be adequate on many mobile devices due to larger screens, better resolutions, and faster connections. But using mobile devices, such as phones and tablets, is still different from using desktop or laptop computers in many ways.

- An operation that can be quite easy with a mouse or keyboard can be much more difficult with touch and touch keyboards.
- Operations that can be done with hotkey combinations or a series of clicks and drags become challenging on a mobile device.
- Unless a stylus is available (currently uncommon among smartphone users), the precision of screen touches isn't as accurate as a mouse cursor.
- Clicking a small link or button can be a source of frustration, and typing can also be more arduous.
- Although display resolution continues to improve, you can't show as much readable information on a single mobile screen as you can a desktop screen.
- Cellular networks continue to improve, but it's not yet a certainty that a mobile user will have a good, fast connection at all times. A user on a road trip with a spotty connection might be more concerned about getting information quickly than about getting the full rich experience of a high-bandwidth site.

For these reasons, it's important to be able to recognize mobile users and present them with alternatives or options to best enhance their experience. In this article, I'll show you how to use AOP to detect mobile users and give them the option of installing a native application.

Providing a link to a native application

Offering a link to your mobile application (or apps) on your home page can be a good way to let your users know that a native application is available for them to use on their device. But keep in mind that not every user always uses the front door. They could be going directly to a page of content via a link that a friend sent, or they could be clicking on a result from a search engine. You could put a link to your mobile application on every page. But now you're taking up screen space for something that only a portion of your users will be interested in (some may not have a smartphone, aren't using the smartphone, or aren't interested in installing an application).

I think a nice way to deal with this is to show the mobile user an interstitial or splash screen—a web page that is displayed in between the source of the link and the page being linked to. Interstitials are often used for ads, warning messages, and so on. Two examples of interstitials are shown in figure 4.9. For our purposes, we'd like to show incoming users an interstitial to let them know that there's a native application available that they can use instead.

Figure 1 Examples of a mobile interstitial when visiting LinkedIn or the Verge

You'll create your own interstitial using an ASP.NET `HttpModule`. It won't look nearly as nice as these examples, but it'll have the same functionality. You're going to create an `HttpModule` that detects if the user is a mobile user, as shown in figure 4.10. If so, you redirect them to the interstitial page to give them the option of either downloading the native application (via App Store or Google Play) or continuing on to browse the site in the mobile web browser.

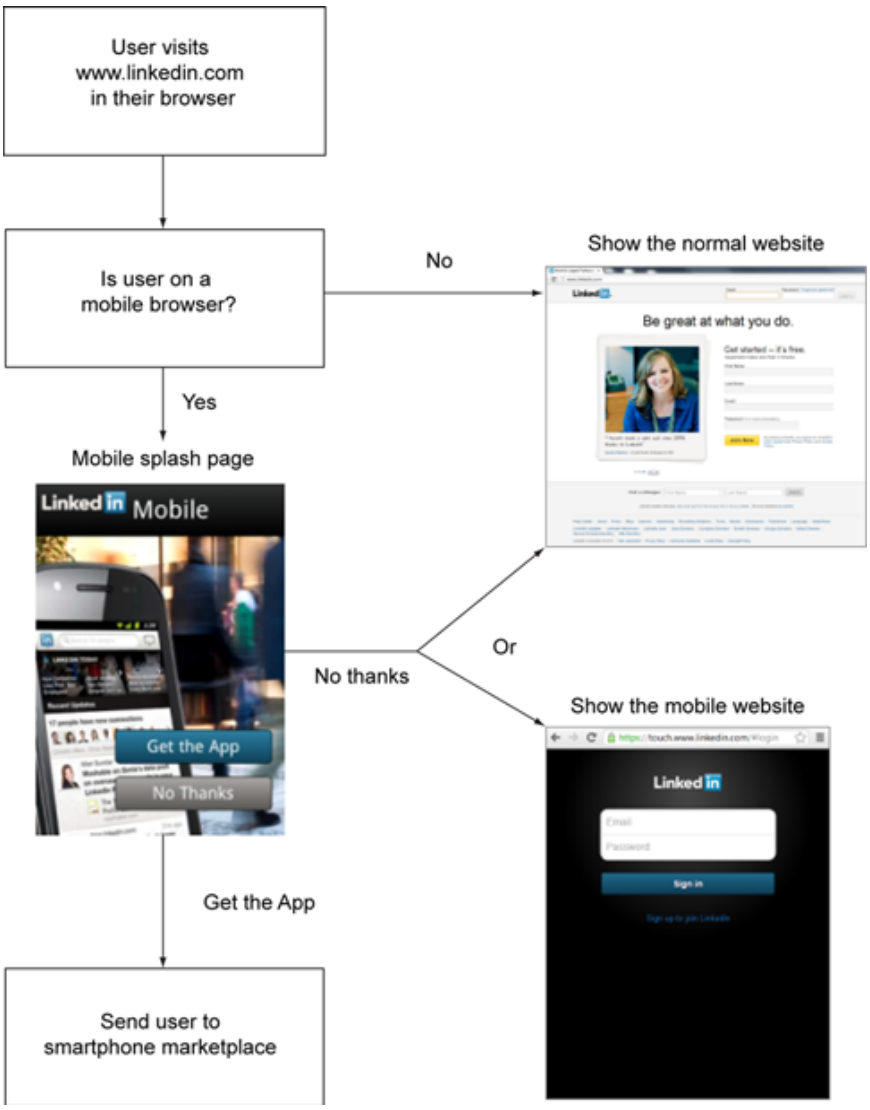

Figure 2 Routing mobile users to their preferred experience

And finally, you don't want to annoy mobile users by showing this interstitial for every request, so you'll use a cookie to make sure that it's being shown only every so often.

Create an HttpModule

Start by creating an ASP.NET project in Visual Studio. I'm using a plain ASP.NET application, not an MVC project. Name it what you want: I'm calling mine MobileBrowserAspNet. The default application template creates a Default.aspx and an About.aspx page, which are sufficient for this example. But because you're using an `HttpModule`, any additional pages that you add automatically pick up the same functionality (one of the key benefits of this approach: it's modular and reusable).

A bare-bones example of a class that implements `IHttpModule` looks like this:

```
public class MobileInterstitialModule : IHttpModule {
    public void Init(HttpApplication context)
    {
    }
    public void Dispose()
    {
    }
```

```
    }
```

For this example, you don't need to put anything in the `Dispose` method because you aren't using any resources that require disposal (for example, a `FileStream` or `SqlConnection`—some resource that's not already handled by .NET's garbage collector).

ASP.NET `HttpModules` run on every HTTP request to the site. The `HttpApplication context` parameter passed to `Init` provides events that are invoked for certain boundaries. For the interstitial, you're interested in subscribing to the `Begin-Request` boundary event, as in this listing.

Listing 1 Subscribing to the `BeginRequest` boundary event

```
public void Init(HttpApplication context) {
    context.BeginRequest += context_BeginRequest;
}

void context_BeginRequest(object sender, EventArgs e) {

}
```

The code in `context_BeginRequest` runs before the page executes, so this is where you check whether the user is a mobile user.

Check for mobile user

Let's start by creating a `MobileDetect` class. Let's assume that you have native applications available for the big three smartphone platforms: Android, iOS (Apple), and Windows Phone. You can approach browser detection in several ways, but let's keep it simple for now and look to see if the `UserAgent` contains certain keywords. The following listing isn't a particularly sophisticated method but is fine for our purposes.

Listing 2 A class to detect mobile users

```
public class MobileDetect {
    readonly HttpRequest _httpRequest;                                  #A

    public MobileDetect(HttpContext httpContext) {
        _httpRequest = httpContext.Request;
    }

    public bool IsMobile() {                                            #B
        return _httpRequest.Browser.IsMobileDevice &&
           (IsAndroid() || IsApple() || IsWindowsPhone());
    }

    public bool IsWindowsPhone() {                                      #C
        return _httpRequest.UserAgent.Contains("Windows Phone OS");
    }

    public bool IsApple() {                                             #D
        return _httpRequest.UserAgent.Contains("iPhone")
           || _httpRequest.UserAgent.Contains("iPad");
    }

    public bool IsAndroid() {                                           #E
        return _httpRequest.UserAgent.Contains("Android");
    }
}
```

#A This class needs an HttpRequest object to examine the user's browser information.
#B This method should tell us if the user is using one of the big 3 smartphones.
#C This method examines the UserAgent to see if a Windows Phone is being used.
#D This method checks to see if an Apple device is being used.
#E This method checks if a device running the Android OS is being used.

Now that this class available, let's use it to check the incoming request.

Redirect to a mobile splash screen

In the next listing, you use `MobileDetect` within the `context_BeginRequest` event handler. If `MobileDetect` says that the incoming request is from a smartphone, it redirects the user to the splash screen page (MobileInterstitial.aspx). You'll create this page next.

Listing 3 Detecting a mobile browser

```
void context_BeginRequest(object sender, EventArgs e)
{
    var httpContext = HttpContext.Current;
    var mobileDetect = new MobileDetect(httpContext);              #A
    if(mobileDetect.IsMobile())
    {
        var url = httpContext.Request.RawUrl;                      #B
        var encodedUrl = HttpUtility.UrlEncode(url);               #C
        httpContext.Response.Redirect(

            "MobileInterstitial.aspx?returnUrl=" + encodedUrl);    #D
    }
}
```

#A Get a MobileDetect object using the current HttpContext.
#B If users decline to download the application, they need to be directed back to the page they were trying to get to in the first place.
#C The return URL will be sent in the querystring, so it must be encoded as such.
#D Redirect to the splash screen interstitial and pass along the return URL in case it's needed.

At this point, users are redirected to the interstitial page. You also need to communicate the page that the user is trying to get to, in case the user taps the No thanks option, so that's why the original URL is being sent to the MobileInterstitial page as the `returnUrl` querystring value.

Create the splash screen

Create a MobileInterstitial.aspx page. The design of my example in figure 4.11 isn't beautiful, but it's functional: some text and two buttons.

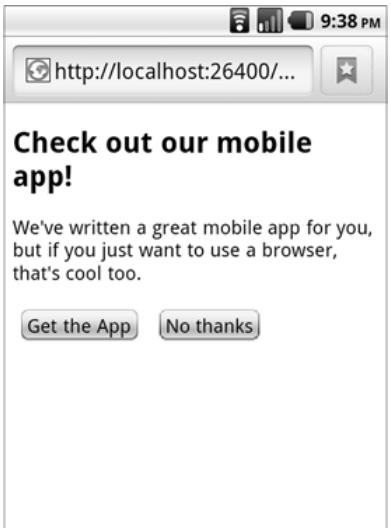

Figure 3 Display an interstitial to the mobile user

MobileInterstitial.aspx has HTML markup for two buttons, as shown in this listing. Make sure that the buttons are in a `form`.

Listing 4 Splash screen with two buttons

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Mobile Interstitial</title>
    <meta name="viewport"
        content="user-scalable=no, width=device-width" />
</head>
<body>
    <h2>Check out our mobile app!</h2>
```

```
    <p>We've written a great mobile app for you,
       but if you just want to use a browser, that's cool too.</p>
    <form id="form1" runat="server">
        <asp:Button ID="btnDownload" Text="Get the App"              #A
                    runat="server" />
        <asp:Button ID="btnNoThanks" Text="No thanks"                #B
                    runat="server"/>
    </form>
</body>
</html>
```

#A This button directs users to their native application store/marketplace.
#B This button sends users to the page they </html> originally requested.

To make the buttons do something, you need to write code in the code-behind page (MobileInterstitial.aspx). The `btnDownload_Click` and `btnNoThanks_Click` methods are wired to the click events in `Page_Load` (see listing 5).

The "No thanks" click event redirects the browser to the originally requested page. Recall that this information was passed to this page via querystring (`return-Url`), so we can use `Request.QueryString` to get that information and send users on their way.

The "Get the App" click event must redirect the user to the appropriate mobile download portal (the iPhone App Store, for instance). You can get more use out of the `MobileDetect` class to determine where to send the user. Each phone can interpret the URL and send the user to the correct location. For instance, if users are on an Android, they're directed to your application in Google Play.

Listing 5 Handling the button clicks in code-behind

```
public partial class MobileInterstitial : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        btnDownload.Click += btnDownload_Click;                                  #A
         btnNoThanks.Click += btnNoThanks_Click;                                 #B
    }

void btnNoThanks_Click(object sender, EventArgs e) {
    var returnUrl = Request.QueryString["returnUrl"];                            #C
    Response.Redirect(HttpUtility.UrlDecode(returnUrl));                         #D
    }
void btnDownload_Click(object sender, EventArgs e) {
     var mobileDetect = new MobileDetect(Context);                                #E
     if (mobileDetect.IsAndroid())
        Response.Redirect(
           "market://search?q=pname:com.myappname.android");                      #F
     if (mobileDetect.IsApple())
        Response.Redirect(
            "http://itunes.com/apps/appname");                                   #G
     if (mobileDetect.IsWindowsPhone())
        Response.Redirect(
            "http://windowsphone.com/s?appid={my-app-id-guid}");                 #H
     }
}
```

#A An event handler for the download button click
#B An event handler for the original page
#C Retrieve the returnUrl from the querystring
#D Decode it so that it's a valid URL that the user can be returned to
#E Using the same MobileDetect class that's used in the HttpModule
#F The URL format to go to the Google Play store for a specific application
#G The URL to use to go to the Apple App Store for a specific application
#H The URL format to use with the Windows Phone Marketplace

And there you have it. Now every page on your site requested by smartphone users first presents them with a splash screen asking if they want to download the mobile application instead. But we're not done.

Note that I said every page gets a splash screen. That's a problem because the splash screen itself is also a page. Sounds like an infinite loop. Also, when the user clicks No thanks and is redirected back, then the `HttpModule` runs again, which we don't want. Let's add some checks to the `HttpModule` to avoid these situations.

Adding Checks

Let's write two methods in `context_BeginRequest` that check for each of the following conditions:

- Is the interstitial itself being requested?

  If so, stop execution and return; otherwise, you'll be in an infinite redirect loop.

- Did the user choose not to download the native application?

  Without checking for this condition, you'll send the user right back to the interstitial again and again.

The following listing shows the `OnMobileInterstitial` and `ComingFromMobileInterstitial` methods.

Listing 6 Adding checks to avoid redirect loops

```
void context_BeginRequest(object sender, EventArgs e) {
    if (OnMobileInterstitial())                                    #A
        return;
    if (ComingFromMobileInterstitial())                            #B
        return;

    var httpContext = HttpContext.Current;
    var mobileDetect = new MobileDetect(httpContext);
    if (mobileDetect.IsMobile())
    {
       var url = httpContext.Request.RawUrl;
       var encodedUrl = HttpUtility.UrlEncode(url);
       httpContext.Response.Redirect(
        "MobileInterstitial.aspx?returnUrl=" + encodedUrl);
    }
}
bool ComingFromMobileInterstitial() {
    var httpRequest = HttpContext.Current.Request;
    if (httpRequest.UrlReferrer == null)                           #C
          return false;
    return httpRequest.UrlReferrer.AbsoluteUri
        .Contains("MobileInterstitial.aspx");                      #D
}
bool OnMobileInterstitial() {
    var httpRequest = HttpContext.Current.Request;
    return httpRequest.RawUrl
        .Contains("MobileInterstitial.aspx");                      #E
}
```

#A If the page being bounded by this HttpModule is the splash page, then don't proceed.
#B If we're coming from the splash page, then don't proceed.
#C The referrer could be null, so put in some defensive programming here to check.
#D If the referrer is the splash page (MobileInterstitial.aspx), then return true.
#E If the page being bounded is itself the splash page, then return true.

Now the user won't be stuck in loops. But I think we can do better.

Don't be a pest

Suppose a user doesn't want to use the native application and wants to view your site in a standard mobile browser. As it stands now, every time such users want to view a page, they're shown the splash screen first, which could get annoying. Instead of showing the splash screen for every request, it's much better to show the splash screen only once. If users click No thanks, you won't bother them again.

One way to do this is to set a cookie when a user clicks No thanks. Open MobileInterstitial.aspx.cs, and set a cookie in the `"NoThanks"` click event:

```
void btnNoThanks_Click(object sender, EventArgs e) {
    var cookie = new HttpCookie("NoThanks", "set");                #A
    cookie.Expires = DateTime.Now.AddMinutes(2);                   #B
    Response.Cookies.Add(cookie);                                  #C

    var returnUrl = Request.QueryString["returnUrl"];
    Response.Redirect(HttpUtility.UrlDecode(returnUrl));
}
```

#A Create a cookie called "NoThanks", the value of which ("set") is arbitrary
#B Set the cookie to expire after two minutes
#C Add the cookie to the response

For this example, I set the cookie to expire after two minutes so that users see the splash screen up to once every two minutes (if you're following along, you have to wait only two minutes to see this in action). In reality, you may want to set a much longer expiration period (one month, six months, one year) or set it to never expire. To give users more flexibility, you could add a checkbox (labeled Don't Ask Me Again) to the splash screen, which determines whether to set an expiration on the cookie.

After you set the cookie, you need to check for that cookie back in the `HttpModule`. In this listing, I added a cookie check to `context_BeginRequest`.

Listing 7 Checking for a cookie

```
void context_BeginRequest(object sender, EventArgs e) {
    if (IsNoThanksCookieSet())
          return;
    if (OnMobileInterstitial())
          return;
    if (ComingFromMobileInterstitial())
          return;

    var mobileDetect = new MobileDetect(httpContext);
    if (mobileDetect.IsMobile())
    {
        var url = httpContext.Request.RawUrl;
        var encodedUrl = HttpUtility.UrlEncode(url);
        httpContext.Response.Redirect(
            "MobileInterstitial.aspx?returnUrl=" + encodedUrl);
    }
}
bool IsNoThanksCookieSet() {
    return HttpContext.Current.Request.Cookies["NoThanks"] != null;
}
```

#A If the cookie is set, then stop execution of the HttpModule

HttpModule used on every page?

Depending on your web server, `HttpModules` could literally affect every page that's requested, including CSS files, image files, text files, and PDF files. You don't want an interstitial that runs on CSS files.

With IIS7, you can configure the module in the Web.config file with `preCondition="managedHandler"`, which means that the `HttpModule` executes only on files that run managed code (for example, it provides boundaries for ASPX files but not for CSS files).

If you're using IIS6, it might or might not be configured to allow static files such as CSS files to be processed by ASP.NET. If you can't change the configuration as you wish, then you can add some code to check on this. For example, see if `httpContext.Request.Url.AbsolutePath` contains a .aspx extension.

`HttpModules` can be an excellent way to use AOP to address cross-cutting concerns in your web application. They have powerful and flexible capabilities that are baked right into the ASP.NET framework but are too often overlooked and forgotten.

Summary

This article covered a common type of aspect: boundary aspects. Boundaries are like the borders between states. A boundary aspect gives you the opportunity to put behavior at the boundaries in your code. ASP.NET gives you the ability to write web page boundary aspects via `HttpModules`. The API provides contextual information (for

example, information about the HTTP request) as well as the ability to control the flow of the program (for example, redirects to another page).

Here are some other Manning titles you might be interested in:

Windows 8 Apps with HTML5 and JavaScript
Dan Shultz, Joel Cochran, and James Bender

Fast ASP.NET Websites
Dean Alan Hume

Windows Store App Development: C# and XAML
Pete Brown

Last updated: August 2, 2013