CHAPTER 15

Building Mobile Web Applications

Mobile applications are changing the way people interact with software. With the abundance of mobile devices such as smart phones and tablets, developing mobile applications and delivering content to mobile devices requires learning new skills and technologies. This chapter examines how you can build mobile web and native applications in F# using WebSharper. The topics covered are:

- Why web-based mobile applications are becoming increasingly important
- How you can use feature detection and polyfilling libraries in your applications to work around mobile browser limitations and missing features
- How you can serve mobile web content from your WebSharper applications
- How you can develop WebSharper applications for iOS that use platform-specific markup to access unique features such as multi-touch events
- How you can develop WebSharper applications that use the Facebook API, parts of which you bind manually using a WebSharper Extension project
- How you can use WebSharper Mobile to create native Android and Windows Phone packages for your WebSharper applications
- How you can integrate mobile formlets and Bing Maps in a WebSharper application

Web-based vs. Native Mobile Applications

Mobile applications are designed to run on mobile devices and to utilize their additional capabilities such as GPS positioning, camera, and touch screen; more technically, they rely on the executing mobile operating system to provide access to mobile capabilities found on the device. When mobile devices first appeared, various vendors produced operating systems for them, each providing a unique user experience and capabilities, and with that, a unique way to develop applications for these platforms.

Native applications are those designed to run specifically on a given platform (or even a specific hardware device), and depend on the exact capabilities that are provided on that platform by its execution environment. Typically, a native application needs heavy adaptation or even an entire rewrite in order to run on another platform. For instance, many iPhone or iPad applications are developed in Apple's own programming language, using vendor-specific libraries and development environments. The same holds for native Android, Windows Phone, Symbian, and other applications.

To add to the diverse mobile operating system landscape, each platform comes with its own *application store*, giving end-users a one-stop shop for new applications and their updates. While the

D. Syme et al., *Expert F# 3.0*

existence of application stores has resulted in a massive explosion in the number of mobile applications available and, with that, several mainstream mobile platforms solidifying, it has also played an important role in developers looking for *cross-platform solutions* to build their applications in order to reach a larger audience, and undoubtedly, to save on their efforts by avoiding to build several versions of the same application for different platforms.

One approach to cross-platform mobile development is utilizing the existing HTML4-based mobile browsers and providing mobile websites (you probably saw countless examples of these sitting at URLs like `http://m.something.com`) next to native mobile applications. These are web applications with reduced capabilities, compared to their mobile counterparts, due to their lack of web/HTML features that could provide similar user experience and functionality. Nonetheless, mobile websites played and continue to play an important role in providing content and applications on mobile devices, thus contributing to mobile device explosion. On one hand, mobile websites provide a clean and simple version of an application, therefore prompting the developers to consider the essential parts of their applications. On the other hand, mobile websites acted as a conduit to explore new and innovative uses of web applications coupled with the convenience of mobile devices.

Another parallel development is the appearance of various HTML5 standards. HTML5 is a large and constantly evolving set of standards, now covering important mobile features such as multi-touch, native-strength audio, video and camera capabilities, efficient client-server communication via web sockets, and so on. HTML5 along with CSS3 now provide a standards-driven alternative to native mobile applications, and have proven not only viable, but also in many respects, superior to native application development.

In this chapter, you will be taking a short introduction into developing mobile web applications in F# using WebSharper, the F# web framework you saw in Chapter 14. You will learn about progressive enhancement, responsive web design, feature detection and polyfilling, various HTML and JavaScript features such as multi-touch events and mobile meta tags supported on different mobile browsers, developing WebSharper mobile web applications using these features, and finally, using WebSharper Mobile to package these applications into native Android or Windows Phone applications.

PROGRESSIVE ENHANCEMENT AND RESPONSIVE WEB DESIGN

One of the challenges of mobile web development (or ordinary web development for that matter) is accommodating the vast diversity of end-user devices that come with different capabilities to render content. In the past, one way to work with this added complexity was to detect certain devices or browsers and implicitly causing or even explicitly advising users to update to a newer version that better supports the features of the given application. This is a failed attempt to provide *graceful degradation*, a technique that starts with a feature-rich version customized to a particular browser technology and provides various fallback mechanisms for certain features when they are detected missing.

In today's world, new techniques are emerging to tackle this problem in a fundamentally different, better way. For one, you can apply *progressive enhancement*, a strategy where you develop applications based on clean and rich semantic markup that is accessible to all devices, and add more sophisticated features as enhancements in layers that, where interpreted correctly, yield an improved user experience. These enhancements can be applied on the presentation layer using various CSS and styling elements, and on the behavioral layer using client-side scripting.

We have used the term *progressive enhancement* in a similar but slightly different context in Chapter 14, where you saw how you can apply incremental enhancements to formlets and their input controls. These enhancements concerned visual details such as labels, icons, and layout, and dynamic behaviors such as validation, and were applied incrementally to build progressively more enhanced formlets.

Responsive web design uses progressive enhancement to enable web applications to adapt to different screen resolution and plays an important role in developing modern mobile web applications. It relies on

three fundamental building blocks: fluid grids, flexible images, and media queries. Fluid grids structure their content depending on the available screen size, and may shift cells in predictable ways to better adapt to the real estate available on the device. Flexible images resize on demand to keep the content in which they are embedded consistently formatted. Media queries provide conditional styling based on a CSS media type (such as `handheld`, `print`, `projection`, or `tv`) and an optional set of expressions that involve particular media features such as width or orientation. A handful of media query examples are shown in Table 15-1.

Table 15-1. *Examples for Media Queries*

Media Query	Note
`@media screen and (max-width: 640px) {...}`	Apply if the width of the viewing area is no more than 640 pixels.
`@media screen and (min-width: 1024px) {...}`	Apply if the width of the viewing area is at least 1024 pixels.
`@media screen and (max-device-width: 480px) {...}`	Apply if the width of the device is no more than 480 pixels.
`@media screen and (min-device-width: 320px) {...}`	Apply if the width of the device is at least 320 pixels.
@media (orientation: portrait) {...}	Apply if the orientation of the device is portrait.

Feature Detection and Polyfilling in WebSharper

At first look, mobile web applications differ little from ordinary web applications. However, although HTML5 is spreading quickly to mobile browsers, there are still various differences that need to be dealt with on various platforms. Coming to the rescue, you should design your applications with running in different contexts in mind using *feature detection,* via a multitude of JavaScript libraries such as Modernizr (`http://modernizr.com`) or has.js. In conjunction with feature checking, you can also use what is often referred to as *polyfilling*, e.g., using JavaScript to fill in for missing browser features. Many of the polyfilling libraries exist to provide various HTML5 support such as `canvas`, `video` or `audio` on older browsers such as IE7 and IE8, that otherwise do not support HTML5.

It is impossible to do justice to the various available feature-checking and polyfilling libraries in one short chapter, nor are they set in stone, as there are new and better libraries appearing all the time. Instead, you can find a short list of libraries and the various features they relate to in Table 15-2.

Feature detection libraries provide an easy-to-use API to query for various browser features, often as a table lookup facility or a function call that identifies browser features with keys (usually as strings), and returning a Boolean value depending on whether the given feature is supported in the executing context or not. For instance, in has.js, you can check whether the end user's browser supports HTML5/video with the following JavaScript snippet:

```
if(has("video")) {
    // Your enviroment supports HTML5 video
} else {
    // It doesn't, so you must use some kind of video polyfill
}
```

Modernizr also provides a convenient shorthand notation not only to check for various features, but also to polyfill them if they are absent. Consider the following Modernizr JavaScript snippet:

```
Modernizr.load({
  test: Modernizr.geolocation,
  yep : 'geo.js',
  nope: 'geo-polyfill.js'
});
```

This snippet defines a conditional loading of a script based on the result of a particular test function `Modernizr.geolocation`, which checks for geolocation support in the executing environment. Modernizr includes a large number of tests for various CSS, JavaScript, HTML5, and other features you can use in your applications to detect context support and fall back accordingly.

Table 15-2. *Some Notable Polyfill Libraries Available*

Name	Library URL	Type
AmplifyJS	`http://amplifyjs.com/`	HTML5/LocalStorage
audio.js	`http://kolber.github.com/audiojs/`	HTML5/Audio
excanvas	`http://code.google.com/p/explorercanvas/`	HTML5/Canvas
Flashcanvas	`http://flashcanvas.net/`	HTML5/Canvas
HTML-History-API	`https://github.com/devote/HTML5-History-API`	HTML5/History
MediaElement.JS	`http://mediaelementjs.com/`	HTML5/Video+Audio
pdf.js	`https://github.com/mozilla/pdf.js`	PDF
Raphaël	`http://raphaeljs.com`	SVG/VML Graphics
Shumway	`https://github.com/mozilla/shumway`	Flash
Socket.IO	`http://socket.io`	HTML5/WebSockets
Video.js	`http://videojs.com`	HTML5/Video

To use JavaScript libraries like Modernizr or has.js in your WebSharper applications, you have several choices. First, you can define JavaScript inlines in your WebSharper code to "stub" certain functions and to instruct WebSharper to output your inlines for them at code generation time. For this to work correctly, you need to make sure that the module containing your inlined functions is properly annotated to require any JavaScript/CSS artifacts that belong to the library you are referencing. Here is an example to inline has.js's `has(…)` function you saw earlier and to make it available to your F# code as `HasJs.Has`:

```
open IntelliFactory.WebSharper

module MyResources =
    type HasJs() =
        inherit Resources.BaseResource("http://[put-URL-here]/has.js")

    module Features =
        type Video() =
            inherit Resources.BaseResource("http://[put-URL-here]/has.js")

        ...

[<Require(typeof<MyResources.HasJs>)>]
module HasJs =
    module Features =
```

```
        [<Require(typeOf<MyResources.Features.Video>)>]
        [<JavaScript>]
        let Video() = "video"
        ...

    [<Inline("has($s) ? $ifyes() : $ifno()")>]
    let Has (s: string) (ifyes: unit -> unit) (ifno: unit -> unit) =
        ()
```

With this JavaScript stub, your WebSharper code could use has.js to detect various browser features. For instance, the snippet you saw above could be phrased in WebSharper as:

```
do HasJs.Has (HasJs.Features.Video())
    <| fun () ->
        // Your enviroment supports HTML5 video
        ...
    <| fun () ->
       // It doesn't, so you must use some kind of video polyfill
       ...
```

In addition to manually stubbing out JavaScript functions with inlines, you can also create a new WebSharper extension for your JavaScript library at hand using the WebSharper Interface Generator tool you saw in Chapter 14. Extensions are an essential part of using WebSharper because they provide the bridge to the outside world and to any JavaScript library; and you may find the need to build WebSharper extensions from time to time, either to existing third-party JavaScript libraries or to libraries of your own. You will see an example of developing a WebSharper extension to a small part of the Facebook API later in this chapter.

Alternatively, you can check if a given WebSharper extension is already available on the WebSharper downloads page at `http://websharper.com/downloads`. WebSharper extensions are added regularly, so there is a good chance that someone has already done the work for you. At the time of writing, WebSharper has over two dozen extensions available, covering a good selection of mobile libraries, including jQuery Mobile (`http://jquerymobile.com/`), Sencha Touch (`http://www.sencha.com/products/touch/`), and Kendo UI (`http://kendoui.com/`), and many other JavaScript libraries you can use in your web or mobile web applications.

Mobile Capabilities, Touch Events, and Mobile Frameworks

One of key differentiators of mobile applications is the ability to use various touch events to control them. You will see later in this chapter how you can add handlers to these events using JavaScript libraries, such as jQuery Mobile, that provide access to vendor-specific JavaScript events in a unified way. There are a growing number of similar JavaScript libraries that aim to enable your applications to respond to touch events. For instance, hammer.js (`http://eightmedia.github.com/hammer.js`) and zepto.js (`http://zeptojs.com`) both provide support for touch events. Table 15-3 summarizes the different kinds of touch events that emerged as "standard" on touch-based devices.

Table 15-3. *Summary of Touch Events*

Event	Description
`Tap`	Touching on a single point for a short period.
`Double Tap`	Tapping on a single point twice within a time threshold.

Event	Description
`Swipe`	Touching on a single point and moving horizontally or vertically.
`Hold`	Touching on a single point for a longer period.
`Transform`	Touching on at least two points and moving them in any direction.
`Drag / Pan`	Touching on a single point and moving in any direction.

Beyond single and multi-touch events, mobile devices also provide a number of other capabilities to applications running on them. Some of these mobile capabilities are listed in Table 15-4. Mobile JavaScript libraries and frameworks, and technologies such as WebSharper Mobile (introduced later in this chapter) and PhoneGap (`http://phonegap.com`) provide access to some or all of these capabilities using JavaScript. More advanced capabilities, such as access to local storage, the file system on the device, its communication and networking capabilities, and any contacts databases stored on the device are also becoming available to JavaScript as these technologies mature.

Table 15-4. *Some Mobile Capabilities Available in Mobile Devices*

Capability	Description
Accelerometer	Providing information on the device's acceleration
Audio/Media	Playing sounds using the device's speakerphone(s)
Camera	Taking pictures using the device camera
Communication	Enabling access to the device's communication channels
Compass	Providing directional/compass information
Geolocation	Providing geolocation information using GPS or A-GPS
Vibration	Responding by vibration
Storage	Providing access to local storage
Touch and Multi Touch	Responding to touch events, summarized in Table 15-3

As you will shortly see in this chapter, serving mobile content and developing mobile web applications that use various mobile capabilities doesn't involve any magic and is actually quite fun to learn to do. To take this experience even further, a large number of JavaScript mobile web frameworks exist that not only enable you to use various mobile user interface widgets and capabilities, but also provide a sort of application framework (some basic, some nearly end-to-end) in which to develop your mobile web applications. Some of these frameworks are listed in Table 15-5.

Table 15-5. *A Handful of JavaScript Mobile Web Application Frameworks*

Framework	URL
Appcelerator Titanium	http://www.appcelerator.com/
DHTMLX Touch	http://www.dhtmlx.com/touch/
iUI	http://www.iui-js.org
jQuery Mobile	http://jquerymobile.com/
Sencha Touch	http://www.sencha.com/products/touch/
Sproutcore	http://sproutcore.com/
wink	http://www.winktoolkit.org/

Serving Mobile Content

So far in this chapter you saw how you can apply feature detection and polyfilling to accommodate different rendering environments, preparing your application to gracefully fall back on older devices and browsers. In this section, you will look at a couple fundamental details that you need to be aware of before you can start serving mobile web content, in particular, understanding how mobile browsers interpret your content differently than desktop browsers.

To start with, consider what happens when you view a basic web page with a header and some text in a desktop browser and compare it with the same page viewed in a mobile browser. You will find that the mobile browser will scale it like a web page. For instance, on Safari it will be scaled to a 980 pixels width, with the expectation that you want to surf pages like on a desktop. However, most mobile devices have fewer physical pixels and therefore the header and text in your test page will appear tiny, if at all visible.

You can fix this by embedding more information in your markup to let your mobile browser know that it is receiving content intended for mobile devices. For instance, to solve the scaling issue, you can emit an additional `meta` tag in the header of your page:

```
<meta name="viewport" content="width=device-width, initial-scale=1"/>
```

This tells mobile browsers to set the default width of the page to the actual width of the device (likely to something fewer than 980 pixels) and not to scale the content. You can also set `minimum-scale` and `maximum-scale` to 1 to avoid the mobile browser zooming in and out, but many users might find this less usable.

The available options for `viewport` and some further iOS meta tags are listed in Table 15-6.

Table 15-6. *Special Meta Tags Available in iOS/Safari*

Meta tag key (name="...")	Meta tag value (content="...")	Description
`apple-mobile-web-app-capable`	`yes/no`	If yes, run application in full screen mode.
`apply-mobile-web-app-status-bar-style`	`default/black/black-translucent`	In full screen mode, specifies how the status bar is displayed.
`format-detection`	`telephone=no`	Disable making strings that look like phone numbers into dial links.

`viewport`	`width=... [980]`
	`height=...`
	`initial-scale=...`
	`maximum-scale=... [0.25]`
	`minimum-scale=... [5.0]`
	`user-scalable=... [yes]`
		Using comma as a separator, configure various viewport options. The defaults of each option are shown in brackets. You can use `device-height` and `device-width` to refer to the dimensions of the device.

Another way to fix the viewport is to apply a different document type, one that specifically targets mobile devices (such as XHTML Mobile Profile 1.0/1.2). However, to use HTML5 features, your best bet remains using the plain HTML5 document type:

```
<!DOCTYPE html>
```

You can serve content with this document type and with the extra `meta` tags programmatically as you saw in Chapter 14, or you can embed them directly into your dynamic template file. You will use this latter approach for the examples in the remainder of this chapter.

Building a Mobile Web Application for iOS Devices

For the examples that follow in this chapter, you will be using various WebSharper extensions, providing stubs to different JavaScript libraries. Some of these extensions you will develop yourself, and some you will download from the WebSharper website (http://websharper.com).

In this section, you are going to build a web application that uses specific iOS/Safari mobile browser features and its JavaScript support for multi touch events. Your application will work on other mobile browsers as well, but some features such as multi-touch events may be missing. To simplify your coding, we assume that no feature detection needs to be done; you can insert these on demand in your real life coding. As you'll see, these mobile browser features provide for a user experience that is reasonably similar to native applications. Yet, the development effort and the ability to move a lot of the code to other platforms without changes makes mobile web applications much more attractive than their native counterparts.

The application you are going to be developing in this section is shown in Figure 15-1. It implements a basic image viewer application that preloads a large image and enables the user to rotate it, move it, and zoom it using multi-touch events.

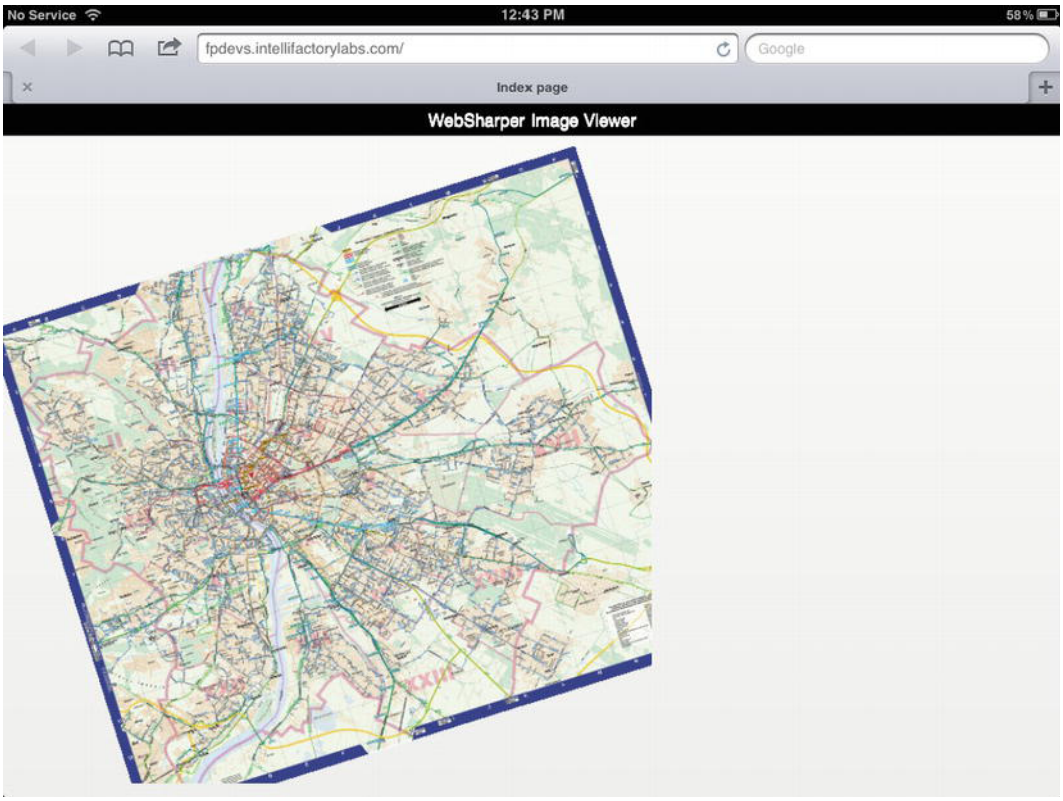

Figure 15-1. *The image viewer application running on iPad*

Fleshing Out the Application

The application is implemented using the HTML Site WebSharper template. You can use this template for a basic application that will implement an HTML site with pure HTML and JavaScript code. Go ahead and create a new solution based on this template. At this point, you should have the basic project references configured, and your project should contain three files: `extra.files`, `Main.fs`, and `Main.html`.

In this application, you will be using WebSharper Extensions for jQuery Mobile, available on the WebSharper download page (`http://websharper.com/downloads`). At the time of writing, WebSharper extensions install under `%ProgramFiles%\IntelliFactory\WebSharper\Extensions`. After installing this extension, find and add a library reference to `IntelliFactory.WebSharper.JQuery.Mobile.dll`. At this point, you are all set to use this extension later in your application code.

A special configuration file used by offline WebSharper projects, `extra.files` is used to copy files that belong to sitelets from the containing F# project. This enables you to encapsulate external artifacts such as images and style sheet files inside a single F# project instead of having to manually manage them in a consuming project. Playing an important role in Android and Windows Phone WebSharper applications as well, `extra.files` instructs the packaging application to include any additional and necessary artifacts in target mobile packages.

In this example, you will only need a large image (say, containing more than 1000x1000 pixels) that will be preloaded into the application for manipulating its size, position, and dimensions. Assuming that you put it under an `images` folder as `map.jpg` in the HTML Site project, add it to `extra.files` as shown in Listing 15-1:

Listing 15-1. `extra.files`

```
images/map.jpg
```

`Main.html` is used as the dynamic site template for this application, and you can make a few changes to it to fit your needs. First, you can remove the title-related markup to keep the application to a minimum. Then you can add a `meta` tag to set the viewport as you saw in the previous section, and also introduce a few lines of styling. The resulting template is shown in Listing 15-2.

Listing 15-2. `Main.html`

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="generator" content="websharper" data-replace="scripts" />
    <meta name="viewport" content="width=device-width, initial-scale=1"/>
    <style type="text/css"><![CDATA[
        *
        {
            margin: 0;
            padding: 0;
        }

        .main
        {
            position: absolute;
            top: 0;
            left: 0;
            width: 100%;
            height: 100%;
```

```
            overflow: hidden;
        }

        .head
        {
            position: absolute;
            height: 30px;
            text-align: center;
            line-height: 30px;
            background-color: Black;
            color: White;
            z-index: 1;
            width: 100%;
        }

        .head > *
        {
            float: left;
            margin-right: 10px;
        }

        .pan-image
        {
            display: none;
        }

        .pan-canvas
        {
            position: absolute;
            top: 30px;
            left: 0;
        }
    ]]></style>
  </head>
  <body>
    <div data-hole="body"></div>
  </body>
</html>
```

Note in Listing 15-2 that you use the HTML5 document type, and that the style markup is contained in a `CDATA` block to avoid any XML parsing errors. WebSharper dynamic templates must be valid XML documents, and using `CDATA` blocks to embed text with special characters is a convenient way to get around any limitations you may encounter otherwise. And the last bit, your main content placeholder `body` is wrapped in a `div` node.

Now that you have your basic template and the `extra.files` configuration set up, the only piece left is the main application code itself. This is shown in Listing 15-3.

Listing 15-3. `Main.fs`

```
namespace MyNamespace
```

```
open System
open System.IO
open System.Web
open IntelliFactory.WebSharper
open IntelliFactory.WebSharper.Sitelets

module MySite =
   open IntelliFactory.Html

   type Action = | Index

   module Skin =
      type Page =
         {
             Body: Content.HtmlElement list
         }

      let MainTemplate =
         let path = Path.Combine(__SOURCE_DIRECTORY__, "Main.html")
         Content.Template<Page>(path)
            .With("body", fun x -> x.Body)

      let WithTemplate body : Content<Action> =
         Content.WithTemplate MainTemplate <| fun context ->
            {
                Body = body context
            }

   module Client =
      open IntelliFactory.WebSharper.Html
      open IntelliFactory.WebSharper.JQuery

      type State =
         {
            mutable x: float
            mutable y: float
            mutable scale: float
            mutable angle: float
         }

      type MyControl() =
         inherit IntelliFactory.WebSharper.Web.Control()

         [<JavaScript>]
         override this.Body =
            let main = Div [Attr.Class "main"]
            let head =
               Div [Attr.Class "head"] -< [
                  Text "WebSharper Image Viewer"
               ]
            main -< [
```

```
head
HTML5.Tags.Canvas [
    Attr.Class "pan-canvas";Attr.Width "600";Attr.Height "600"
]
|>! OnAfterRender (fun e ->
   let canvas = As<Html5.CanvasElement> e.Body
   let ctx = canvas.GetContext "2d"
   Img [Attr.Src "images/map.jpg"; Attr.Class "pan-image"]
   |> Events.OnLoad (fun img ->
      let state = { x = 0.; y = 0.; scale = 1.; angle = 0. }
      let delta = { x = 0.; y = 0.; scale = 1.; angle = 0. }
      let redraw() =
         // Reset the transformation matrix
         // and clear the canvas.
         ctx.SetTransform(1., 0., 0., 1., 0., 0.)
         ctx.ClearRect(0., 0.,
                       float canvas.Width, float canvas.Height)

         // In order to have centered rotation and zoom, we
         // must first put the center of the image at the
         // origin of the coordinate system.
         ctx.Translate(float canvas.Width / 2.,
                       float canvas.Height / 2.)
         ctx.Scale(state.scale * delta.scale,
                   state.scale * delta.scale)
         ctx.Rotate(state.angle + delta.angle)

         // Then, when the rotation and zoom are done, we put
         // the image back at the center of the screen, plus
         // the (x, y) translation.
         ctx.Translate(state.x+delta.x - float canvas.Width/2.,
            state.y + delta.y - float canvas.Height / 2.)
         ctx.DrawImage(img.Body, 0., 0.)
      let settleDelta() =
         state.x <- state.x + delta.x
         state.y <- state.y + delta.y
         state.angle <- state.angle + delta.angle
         state.scale <- state.scale * delta.scale
         delta.x <- 0.
         delta.y <- 0.
         delta.scale <- 1.
         delta.angle <- 0.
         redraw()
      let panStartPosition = ref None
         // Pan events
      Mobile.Events.VMouseDown.On(JQuery.Of(e.Body), fun ev ->
         panStartPosition:=Some(ev.Event.PageX, ev.Event.PageY)
         ev.Event.PreventDefault())
      Mobile.Events.VMouseMove.On(JQuery.Of(e.Body), fun ev ->
         match !panStartPosition with
         | None -> ()
```

```
                        | Some (sx, sy) ->
                            let dx = float(ev.Event.PageX - sx)
                            let dy = float(ev.Event.PageY - sy)
                            let angle = state.angle + delta.angle
                            let scale = state.scale * delta.scale
                            delta.x <- (dx * Math.Cos angle + dy * Math.Sin angle) / scale
                            delta.y <- (dy * Math.Cos angle - dx * Math.Sin angle) / scale
                            redraw()
                            ev.Event.PreventDefault())
                    Mobile.Events.VMouseUp.On(JQuery.Of("body"), fun ev ->
                        if (!panStartPosition).IsSome then
                            settleDelta()
                            panStartPosition := None
                            ev.Event.PreventDefault())
                    // iOS-only rotozoom events
                    e.Body.AddEventListener("gesturechange", (fun (ev: Dom.Event) ->
                        delta.scale <- ev?scale
                        delta.angle <- ev?rotation * System.Math.PI / 180.
                        redraw()
                        ev.PreventDefault()
                        ), false)
                    e.Body.AddEventListener("gestureend", (fun (ev: Dom.Event) ->
                        settleDelta()
                        ev.PreventDefault()
                        ), false)
                    redraw()
                )
            )
        ] :> IPagelet

    let Index =
        Skin.WithTemplate <| fun ctx ->
            [
                Div [new Client.MyControl()]
            ]

    let MySitelet =
        Sitelet.Content "/index" Action.Index Index

type MyWebsite() =
    interface IWebsite<MySite.Action> with
        member this.Sitelet = MySite.MySitelet
        member this.Actions = [MySite.Action.Index]

[<assembly: Website(typeof<MyWebsite>)>]
do ()
```

Digging Deeper

The main application code above sets up basic dynamic templating based on `Main.html` in your project, and defines the main pagelet `MyControl` in the `Client` module, along with a single-page sitelet, `MySitelet`. These are then used to declare the sitelet as a website on the assembly. Note that you specified a singleton list of actions for the `Actions` member in the website declaration: this is used to instruct the offline sitelet generator tool to output the sitelet page that handles this action; in other words, to output the single HTML page in your application. This will yield an HTML file named after the single `Action` case, `index.html`. Building the project in Visual Studio successfully will pop up a Windows Explorer window containing the generated files, and you can open `index.html` in a browser to see it in action.

To use it from an iOS device, you need to publish the application to a public URL and open that URL in your iPad's or iPhone's Safari browser. These devices support multi-touch events, so you should be able to rotate and scale the image you added to your application.

Now, let's look at how the application works. The main client-side control `Client.MyControl` creates the following dynamic markup:

```
<div class="main">
    <div class="head">WebSharper Image Viewer</div>
    <canvas class="pan-canvas" width="600" height="600">
    </canvas>
</div>
```

When the `<canvas>` element is first created and rendered, the `OnAfterRender` event is fired. This event handler is the heart of your application and is attached using the `|!>` operator, which registers a callback function to the given event without returning the parent DOM node. First, it creates an `<img>` node and loads the image you added earlier to the project. Although this image is never added as an actual DOM node, creating it dynamically still causes the browser to construct it and to call its `OnLoad` event. This event then performs a number of steps:

- Creates some bookkeeping for representing the visual state and the delta in between various touch events
- Defines a `redraw()` function that draws the loaded image onto the `<canvas>` element respecting the current state and delta registers
- Defines a `settleDelta()` function that propagates the current delta register into the state and initiates a redraw
- Registers various event handlers

The general format for registering an event handler involves finding the event handler by name and registering a callback function to it taking the event argument and performing the actions you need.

In the above example, you take two different approaches for binding to various events. Recall that earlier you added a reference to WebSharper Extensions for jQuery Mobile to your project. You did this because jQuery Mobile provides a useful abstraction over single-touch events under what it calls *virtual mouse* events, making touch and mouse events uniform regardless of running on mobile or desktop browsers. These events can be addressed with the jQuery Mobile extensions in a type-safe manner:

```
Mobile.Events.VMouseDown.On(JQuery.Of(e.Body), fun ev ->
    panStartPosition := Some (ev.Event.PageX, ev.Event.PageY)
    ev.PreventDefault())
```

The above call registers an event handler for jQuery Mobile's `VMouseDown` event: the event that occurs when the user clicks down using the mouse or when a touch device is touched. The event handler is added to the `canvas` element (represented by `e`), and the callback function takes an event arguments parameter `ev`

of type `Mobile.VMouseEventArgs`. This type contains various important virtual mouse-related members, such `ClientX`, `ClientY`, `ScreenX`, `ScreenY`, and an `Event` member to refer back to the underlying jQuery event object. This object carries, among others, the click/touch coordinates in `PageX` and `PageY`.

The various other events available in the `IntelliFactory.WebSharper.Mobile.Events` class are summarized in Table 15-7. Many of these events are key to mobile applications, providing interactions such as swiping, scrolling, and tapping on a mobile device, and responding to important events such as changes in the orientation of the device or updating the presentation layout.

Next to binding to events using jQuery Mobile, the second approach you saw in the example above involves binding to non-standard events, such as the multi touch events not available in the current version of WebSharper Extensions for jQuery Mobile, using the plain JavaScript event binding primitive `AddEventListener`. Consider the following snippet from the example:

```
e.Body.AddEventListener("gesturechange", (fun (ev: Dom.Event) ->
    delta.scale <- ev?scale
    delta.angle <- ev?rotation * System.Math.PI / 180.
    redraw()
    ev.PreventDefault()
    ), false)
```

This registers an event handler for the `gesturechange` event, a non-standard event available in iOS/Safari browsers. Here, the callback function takes a plain DOM event object that contains standard JavaScript functions to manipulate how the event is propagated (such as `PreventDefault`), where and when it was initiated, etc. It also contains in various browser implementations additional pieces of data that are tagged onto it for specific events. In the above snippet, `scale` and `rotation` are extracted using the dynamic access operator (`?`). This operator attempts to fetch from its first argument a field with a name given by an F# identifier following it: roughly the dynamic equivalent of the dot used for ordinary .NET member access. In our example, `scale` and `rotation` are known to be available in browsers that support the `gesturechange` event (most notably, iOS/Safari), but in general, you should provide for a fallback when the dynamic lookup fails.

Table 15-7. *Mobile Events Provided by WebSharper Extensions for jQuery Mobile*

Event	Type	Description
`OrientationChange`	`Mobile.Event<Mobile.OrientationChangeEvent>`	Triggered when the orientation of the devices changes. [`portrait` | `landscape`]
`ScrollStart`	`Mobile.Event`	Triggered when scrolling starts.
`ScrollStop`	`Mobile.Event`	Triggered when scrolling stops.
`Swipe`	`Mobile.Event`	Triggered when a horizontal drag occurs within a specific duration.
`SwipeLeft`	`Mobile.Event`	Triggered when a swipe event occurs moving in the left direction.
`SwipeRight`	`Mobile.Event`	Triggered when a swipe event occurs moving in the right direction.
`Tap`	`Mobile.Event`	Triggered after a quick, complete touch event.
`TapHold`	`Mobile.Event`	Triggered after a held complete touch event.
`UpdateLayout`	`Mobile.Event`	Triggered when the application's layout changes.
`VClick`	`Mobile.Event`	Triggered on a mouse click or a touch event.
`VMouseCancel`	`Mobile.Event`	Triggered when a virtual mouse event is cancelled.

Event	Type	Description
VMouseDown	Mobile.Event	Triggered on a mouse or touch click down event.
VMouseMove	Mobile.Event	Triggered on a mouse or touch move event.
VMouseOut	Mobile.Event	Triggered on a mouse or touch out event.
VMouseOver	Mobile.Event	Triggered on a mouse or touch over event.
VMouseUp	Mobile.Event	Triggered on a mouse or touch up event.

Developing Social Networking Applications

In the previous section you saw how you can build an HTML5 mobile application that uses specific mobile browser features such as multi-touch events to enhance user experience. In this section, you are going to build another HTML5 mobile application that interfaces with the Facebook API (`http://developers.facebook.com`) to retrieve a Facebook user's status updates and to display them in a mobile application built with WebSharper Extensions for jQuery Mobile. The example demonstrates how you can build a WebSharper extension to the subset of the Facebook API you need, and how to use this extension to build an HTML5 application that is enhanced to run on mobile devices using the jQuery Mobile library and its corresponding WebSharper extension.

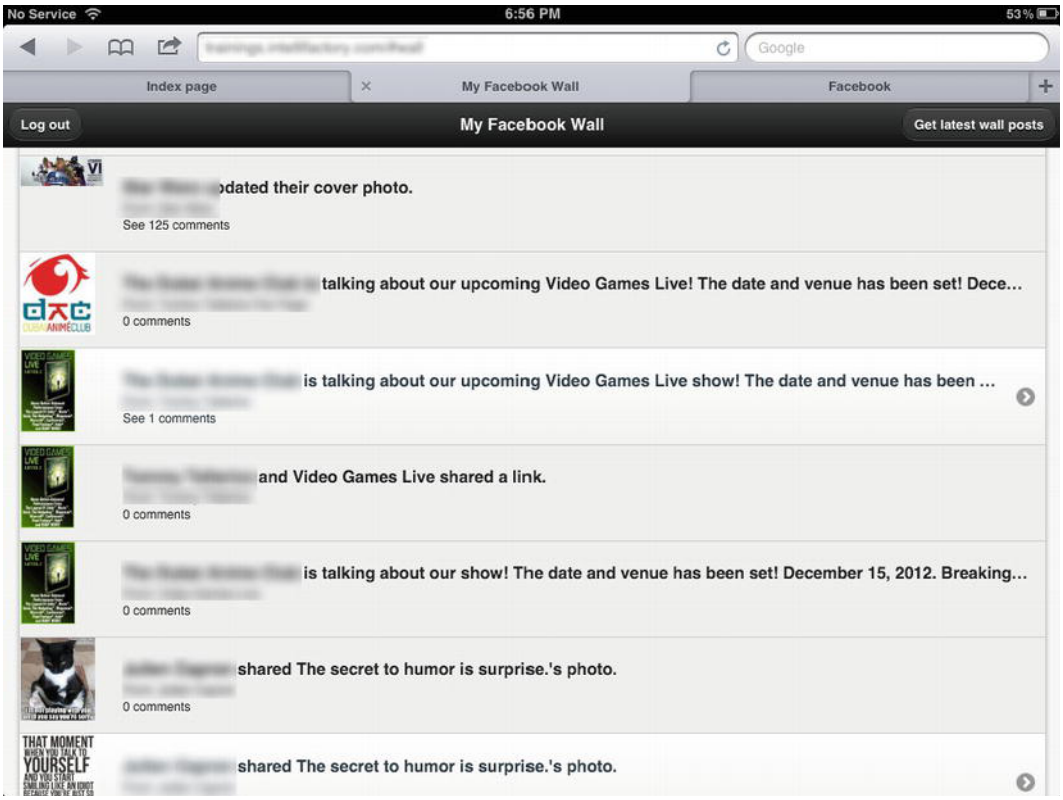

Figure 15-2. *Displaying Facebook status messages*

The application in this section is built as a multi-project solution in Visual Studio. You should start by creating a WebSharper Extension project, and later add a WebSharper HTML Site project to complete the application. As a first step, after you created your new Extension project, replace the contents of `Main.fs` with the code in Listing 15-4.

Listing 15-4. `Main.fs` *– Defining a WebSharper Extension to a Part of Facebook API*

```
namespace IntelliFactory.WebSharper.Facebook

module Definition =
    open IntelliFactory.WebSharper.InterfaceGenerator
    open IntelliFactory.WebSharper.Dom

    module Res =
        let FacebookAPI =
            Resource "FacebookAPI" "https://connect.facebook.net/en_US/all.js"

    let FlashHidingArgs =
        Class "FB.FlashHidingArgs"
        |+> Protocol [
                "state" =? T<string>
                "elem" =? T<Element>
            ]

    let InitOptions =
        Pattern.Config "FB.InitOptions" {
            Required = []
            Optional =
                [
                    "appId", T<string>
                    "cookie", T<bool>
                    "logging", T<bool>
                    "status", T<bool>
                    "xfbml", T<bool>
                    "channelUrl", T<string>
                    "authResponse", T<obj>
                    "frictionlessRequests", T<bool>
                    "hideFlashCallback", FlashHidingArgs ^-> T<unit>
                ]
        }

    let AuthResponse =
        Class "FB.AuthResponse"
        |+> Protocol [
                "accessToken" =? T<string>
                "expiresIn" =? T<string>
                "signedRequest" =? T<string>
                "userId" =? T<string>
            ]
```

```
        let UserStatus =
            Pattern.EnumStrings "FB.UserStatus"
                ["connected"; "not_authorized"; "unknown"]

        let LoginResponse =
            Class "FB.LoginResponse"
            |+> Protocol [
                    "authResponse" =? AuthResponse
                    "status" =? UserStatus
                ]

        let LoginOptions =
            Pattern.Config "FB.LoginOptions" {
                Optional =
                    [
                        "scope", T<string>
                        "display", T<string>
                    ]
                Required = []
            }

        let FB =
            Class "FB"
            |+> [
                    "init" => !?InitOptions ^-> T<unit>
                    "login" => (LoginResponse ^-> T<unit>) * !?LoginOptions ^-> T<unit>
                    "logout" => (LoginResponse ^-> T<unit>) ^-> T<unit>
                    "getLoginStatus" => (LoginResponse ^-> T<unit>) ^-> T<unit>
                    "getAuthResponse" => T<unit> ^-> AuthResponse
                    "api" => T<string>?url * !?T<string>?''method'' * !?T<obj>?options * (T<obj> ^->
    T<unit>)?callback ^-> T<unit>
                ]
            |> Requires [Res.FacebookAPI]

        let Assembly =
            Assembly [
                Namespace "IntelliFactory.WebSharper.Facebook" [
                    FlashHidingArgs
                    InitOptions
                    AuthResponse
                    UserStatus
                    LoginResponse
                    LoginOptions
                    FB
                ]
                Namespace "IntelliFactory.WebSharper.Facebook.Resources" [
                    Res.FacebookAPI
                ]
            ]
```

```
module Main =
    open IntelliFactory.WebSharper.InterfaceGenerator

    do Compiler.Compile stdout Definition.Assembly
```

The various operators you see in this code listing are explained in more detail in the WebSharper documentation in the chapter that describes the WebSharper Interface Generator tool. For instance, given a class type defined via the `Class` function, `|+>` enhances it with new members. You can create a new member giving its name as a string, the `=>` operator, and its type. In addition, you can also use the `=?/=!/=@` operators to create a read-only/write-only/mutable property, or `=%` to create a field. Types are constructed using the types you created in code, using the `T<…>` operator to refer to existing .NET types, using the star (*) operator for tuple types, using the `^->` operator for function types, using the `!?` operator for optional parameters, and `!+` operator for a variable number of arguments, among others.

The extension definition above covers a subset of the Facebook API. In particular, it deals with authenticating users, retrieving their login status, and provides a generic `api` function to retrieve various bits of information from the Facebook service. These functions are defined in a class called `FB`, inside the main `IntelliFactory.WebSharper.Facebook` namespace, along with the various configuration and helper types necessary. To make these functions work correctly at runtime, the `FB` class is enhanced to require the main Facebook API JavaScript file, defined as the resource `Res.FacebookAPI`, pointing to the Facebook domain. This resource will be automatically referenced and included in any sitelet page or ASPX markup that uses the functionality provided by the `FB` class.

Configuring Your New Facebook Application

Before you can implement the main application code, you need to register your application with Facebook to receive an ID that you can use to query the Facebook API. To register your application, go to `http://developers.facebook.com/apps`, sign in if you haven't already, make sure your account is verified, and click the Create New App button on the top. In the popup in Figure 15-3, add the name of your application, select if you need optional hosting, and click Continue.

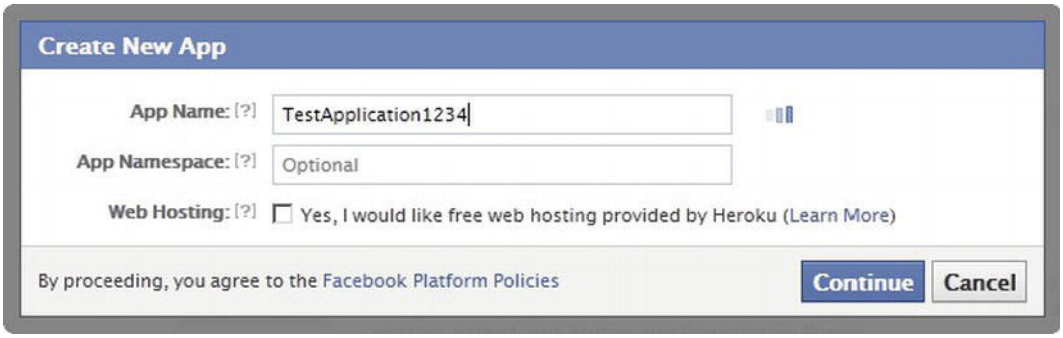

Figure 15-3. *Creating a new Facebook application*

In the next step, you need to enter some CAPTCHA validation, and then proceed onto setting up the application properties. By this point, your new Facebook application has been created and it has a unique ID and secret code pair as shown in Figure 15-4, and shortly you will need the former in your application code.

Figure 15-4. *The unique identifiers of your Facebook application*

But first, in the Basic Info panel, enter the URL of your new application: either the URL where you will be manually hosting your application files, or you can create and configure a hosting domain on Heroku if you have chosen to tick the Web Hosting option in the previous step. You can see an example setup in Figure 15-5, where you used Heroku to host your application files.

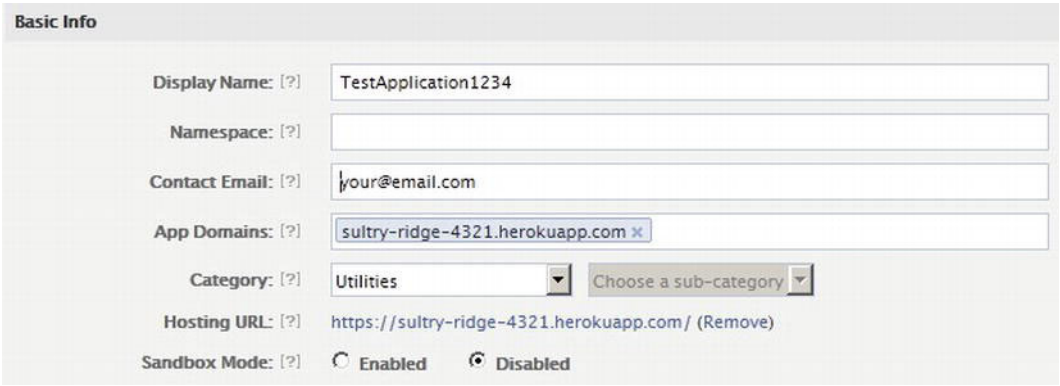

Figure 15-5. *Configuring the domains and the URL of your application*

As a last configuration step, enter your application URL in the Website with Facebook Login option of the application integration section as shown in Figure 15-6.

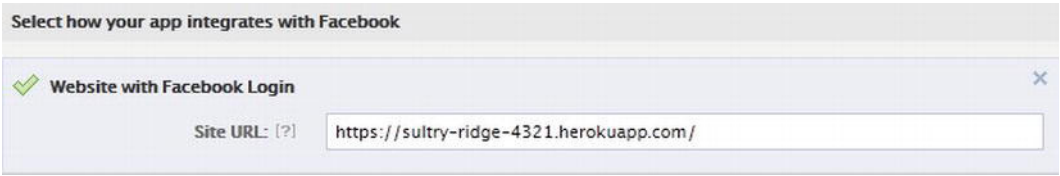

Figure 15-6. *Configuring your application*

Defining the Main HTML Application

The main HTML application uses the same kind of dynamic templating that you saw in the previous section. The template markup in Main.html is shown in Listing 15-5.

Listing 15-5. Main.html – *Defining the Dynamic Template for the Facebook Application*

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Facebook Wall</title>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="generator" content="websharper" data-replace="scripts" />
  </head>
  <body>
    <div data-replace="body" />
    <div data-role="page" id="dummy" />
  </body>
</html>
```

The templating functionality should look familiar to you already. The main application is placed in the Client module and is shown in Listing 15-6.

Listing 15-6. Main.fs– *The Application Code*

```
namespace IntelliFactory.Facebook.Application

open System
open System.IO
open IntelliFactory.WebSharper.Sitelets

module MySite =
   open IntelliFactory.WebSharper
   open IntelliFactory.Html

   type Action = | Index

   module Skin =

      type Page =
         {
             Body : Content.HtmlElement list
         }

      let MainTemplate =
         let path = Path.Combine(__SOURCE_DIRECTORY__, "Main.html")
         Content.Template<Page>(path)
            .With("body", fun x -> x.Body)

      let WithTemplate body : Content<Action> =
         Content.WithTemplate MainTemplate <| fun context ->
            {
                Body = body context
            }

   module Client =
      open IntelliFactory.WebSharper.Html
```

```
open IntelliFactory.WebSharper.Facebook
open IntelliFactory.WebSharper.JQuery
open IntelliFactory.WebSharper.JQuery.Mobile

type Control() =
   inherit IntelliFactory.WebSharper.Web.Control()

   [<JavaScript>]
   let (||?) (x: 'a) (y: 'a) = if As<bool> x then x else y

   [<JavaScript>]
   override this.Body =
      Mobile.Use()
      Mobile.Instance.DefaultPageTransition <- "slide"
      FB.Init(InitOptions(AppId="<<PUT-YOUR-APPID-HERE>>"))
      let btncStatus = Span [Text "Log in"]
      let btnGetPosts =
         A [
            Text "Get latest wall posts"
            Attr.HRef "#"; Attr.Style "display: none;"
         ]
      let updateStatus (resp: LoginResponse) =
         if resp.Status = UserStatus.Connected then
            btncStatus.Text <- "Log out"
            JQuery.Of(btnGetPosts.Body).Show().Ignore
         else
            btncStatus.Text <- "Log in"
            JQuery.Of(btnGetPosts.Body).Hide().Ignore
      let wallPage =
          Div [
             Attr.Id "wall"; HTML5.Attr.Data "role" "page"
             HTML5.Attr.Data "url" "#wall"
          ]
      let wall =
          UL [
             HTML5.Attr.Data "role" "listview"
             HTML5.Attr.Data "inset" "true"
          ]
      wallPage -< [
         Div [
            HTML5.Attr.Data "role" "header"
            HTML5.Attr.Data "position" "fixed"
         ] -< [
            H1 [Text "My Facebook Wall"]
            A [Attr.HRef "#"] -< [btncStatus]
            |>! OnClick (fun el ev ->
               FB.GetLoginStatus <| fun resp ->
                  if resp.Status = UserStatus.Connected then
                     FB.Logout updateStatus
                  else
                     FB.Login(
```

```
                updateStatus,
                LoginOptions(Scope = "read_stream"))
        )
        |>! OnAfterRender (fun _ -> FB.GetLoginStatus updateStatus)
        btnGetPosts
        |>! OnClick (fun el ev ->
           Mobile.Instance.ShowPageLoadingMsg("a","Receiving posts")
           FB.Api("/me/home", fun o ->
              wall.Clear()
              o?data |> Array.iter (fun x ->
                 let message = x?message ||? x?story ||? x?caption
                 LI [
                    yield H6 [Text message]
                    yield P [Text ("From: " + x?from?name)]
                    let numComments =
                       if x?comments && x?comments?count > 0 then
                          "See " + x?comments?count
                       else
                          "0"
                    yield P [Text (numComments + " comments")]
                    yield Img [Attr.Src (if x?picture then x?picture else "")]
                    if x?comments && x?comments?data then
                       yield UL [
                          yield! x?comments?data |> Array.map (fun comment ->
                             LI [
                                H6 [Text comment?message]
                                P [Text comment?from?name]
                             ]
                          )
                          yield LI [
                             HTML5.Attr.Data "iconpos" "left";
                             HTML5.Attr.Data "icon" "arrow-l"] -< [
                                A [Attr.HRef "#"] -< [Text "Back"]
                                |>! OnClick (fun _ _ ->
                                   Mobile.Instance.ChangePage(
                                      JQuery.Of(wallPage.Body),
                                      ChangePageConfig(
                                         Reverse = true))
                                )
                             ]
                          ]
                 ]
                 |> wall.Append
                 Mobile.Instance.HidePageLoadingMsg()
              )
              JQuery.Of(wall.Body) |> ListView.Refresh
           )
        )
     ]
     Div [HTML5.Attr.Data "role" "content"] -< [wall]
]
```

```
            |>! OnAfterRender (fun el ->
               JQuery.Of(el.Body)
               |> JQuery.Page
               |> Mobile.Instance.ChangePage
            ) :> IPagelet

      let Index =
         Skin.WithTemplate <| fun ctx ->
            [
                Div [new Client.Control()]
            ]

      let MySitelet =
         Sitelet.Content "/" Action.Index Index

   type MyWebsite() =
      interface IWebsite<MySite.Action> with
         member this.Sitelet = MySite.MySitelet
         member this.Actions = [MySite.Action.Index]

   [<assembly: Website(typeof<MyWebsite>)>]
   do ()
```

The main client control creates the following skeleton markup:

```
<div id="wall" data-role="page" data-url="#wall">
   <div data-role="header" data-position="fixed">
      ...
   <div>
   <div data-role="content">
      <ul data-role="listview" data-inset="true">
         ...
      </ul>
   </div>
</div>
```

This markup represents a jQuery Mobile "page" with a header and a content panel, the latter with a list of items displayed. Pages are an essential user interface abstraction in jQuery Mobile, as they provide the basic building blocks of multi-page applications where control transfers from one page to another usually enhanced with some sort of visual effect such as sliding in and out, creating a native-like user experience.

In the example above, the header part of the main page contains a title and two buttons: one that reflects the user's login status and displays Log in or Log out, accordingly, and another to fetch the user's wall posts if the user is logged in. This piece of logic is implemented in the `updateStatus` function, which is called when the user logs in or signs out. Logging in and signing out is implemented using the `FB.Login` and `FB.Logout` functions, respectively. Logging in prompts the familiar Facebook login as a new page, and upon successfully authenticating with it, the popup is closed and control is returned to the requesting page. It is therefore important that your application sits on a public URL and that this URL is set up correctly with Facebook, so this redirection can successfully happen.

The main part of the application is in the `Click` event handler for `btnGetPosts`: the "Get latest wall posts" button. Since the call to the Facebook API to retrieve the user's wall posts can take a few moments, a loading animation is shown using `Mobile.Instance.ShowPageLoadingMsg`. This animation is then turned off by the time control is transferred to handling the data returned from the Facebook service. This is initialized with an `FB.Api` call to `/me/home`, and registering a callback to display the data received. The data

you get back from this call is highly dynamic in structure and may contain different pieces for the different types of wall posts. The following line is used to extract either the message, the story, or the caption (for picture posts) of the post:

```
let message = x?message ||? x?story ||? x?caption
```

This bit uses the dynamic "OR" operator `||?` you defined earlier in the `Client` module. This operator uses the fact that a dynamic lookup (using the standard `?` operator) in a record for a non-existing field returns null, which can be checked against as a Boolean value. So the fragment above returns either the `message`, the `story`, or the `caption` fields – whichever is non-null first in the given order. You should refer to the Facebook API documentation for the various post types and the structure they adhere to in case you need to cater to further post types.

The code then returns an `LI` node for each wall post retrieved and these are then wrapped inside the main `UL` list view placeholder node. These `LI` nodes contain the extracted message text, the author of the post, the number of comments, and an optional image that might accompany the post. Any embedded comment data is shown as a nested list view with a Back button to return to the original post.

You should take a look at the Facebook API and incorporate new bits of it into your WebSharper extension, and in turn use these in your application to provide more features such as posting comments, enabling users to like posts, etc. These pieces of functionality are straightforward to add, and you are urged to try out more possibilities the Facebook API and a bit of WebSharper coding can enable.

WebSharper Mobile

In the previous sections you saw how you can build web applications that use specific mobile browser features and WebSharper mobile extensions to implement a native-like user experience on various mobile devices. These included the ability to use the entire device screen for your applications, adding them as icons to your iOS device, using multi-touch events and other gestures, etc. In this section, you are going to build native application packages using WebSharper Mobile, a set of extensions to WebSharper that provide mobile capabilities and native application containers for mobile web applications. These mobile features are shipped with the standard WebSharper installer.

The native mobile capabilities supported in WebSharper Mobile across different platforms are encapsulated in `IntelliFactory.WebSharper.Mobile` as a set of interfaces and helper types, and are given concrete implementations in the corresponding mobile namespaces such as `IntelliFactory.WebSharper.Android`. At the time of writing, WebSharper Mobile supports creating native applications for Android and Windows Phone, and exposes WebSharper access to accelerometer and geo location data, camera functionality, and logging. These members are summarized in Tables 15-8, 15-9, 15-10, and 15-11.

Table 15-8. *Members Related to Acceleration in* `IntelliFactory.WebSharper.Mobile.IAcceleration`

Member	Type	Description
`AccelerationChange`	`IEvent<Mobile.Acceleration>`	Allows subscription to acceleration updates
`IsMeasuringAcceleration`	`Bool`	Gets or sets the state of acceleration subscription

Table 15-9. *Members Related to Locations in* `IntelliFactory.WebSharper.Mobile.IGeolocator`

Member	Type	Description
`GetLocation`	`unit -> Async<Mobile.Location>`	Returns the current location of the device

Table 15-10. Members Related to Logging in `IntelliFactory.WebSharper.Mobile.ILog`

Member	Type	Description
`Trace`	`Mobile.Priority -> string -> string -> unit`	Traces a message to the system log

Table 15-11. Members Related to Camera Support in `IntelliFactory.WebSharper.Mobile.ICamera`

Member	Type	Description
`Location.Latitude`	`Double`	Gets the latitude of the current location
`TakePicture`	`unit -> Async<Mobile.Jpeg>`	Uses the device camera, if available, to take a picture

The concrete platform implementations may also contain specific features beyond the core set of mobile functions. For instance, WebSharper, at the time of writing, provides experimental support to Bluetooth communication on Android devices. This API is listed in Table 15-12.

Table 15-12. Members in `IntelliFactory.WebSharper.Android.Bluetooth`

Member	Type	Description
`CancelDiscovery`	`unit -> unit`	Cancels the Bluetooth discovery process
`ConnectToDevice`	`Bluetooth.Device * string -> Async<Bluetooth.Socket>`	Connects to the given Bluetooth device asynchronously
`Enable`	`unit -> async<unit>`	Enables Bluetooth on the device
`GetBondedDevices`	`unit -> seq<Bluetooth.Device>`	Gets all paired Bluetooth devices
`MakeDiscoverable`	`int -> Async<unit>`	Makes the current device discoverable for a given number of seconds.
`Serve`	`string -> string -> (Bluetooth.Connection -> Async<unit>) -> Bluetooth.Server`	Starts a Bluetooth server with the given name and UUID
`StartDiscovery`	`unit -> unit`	Starts the Bluetooth device discovery process
`Discovery`	`IEvent<Bluetooth.Device>`	The Bluetooth device discovery event
`IsDiscoverable`	`bool`	Tests if the current Android device is discoverable via Bluetooth
`IsDiscovering`	`bool`	Tests if the Bluetooth device discovery process is active
`IsEnabled`	`bool`	Tests if Bluetooth is enabled on the current Android device
`this.Get`	`unit -> Bluetooth.Context option`	Gets a Bluetooth context, if present on the current platform

The easiest way to get started developing native mobile applications with WebSharper is by creating a WebSharper Android or Windows Phone Application project. These templates contain, beyond the basic offline sitelet library hooks and the basic WebSharper Mobile foundation, the Android and Windows Phone implementations of the WebSharper Mobile core capabilities and any platform-specific extensions, respectively.

Developing Android Applications with WebSharper

In this section, you will be developing a small native Android application that combines some of the WebSharper Mobile capabilities with specific extensions such as Formlets for jQuery Mobile and Bing Maps. In particular, Formlets for jQuery Mobile provides a formlet abstraction using the jQuery Mobile look and feel, giving you the ability to define type-safe, composable mobile user interfaces: an important building block to declaratively build native mobile applications with attractive user interfaces.

The application you will develop is shown in Figure 15-7, running on the Android emulator from the Android SDK. It constructs a simple login dialog using jQuery Mobile formlets, and proceeds onto a page with a Bing map showing your current location, updating regularly to reflect the user's location as he or she may be moving around. Naturally, to see this application in action you should deploy it on an actual Android device; nonetheless, testing with the Android emulator is a good way to get going. Alternatively, you can also test device-neutral functionality in a plain browser such as Firefox or Chrome.

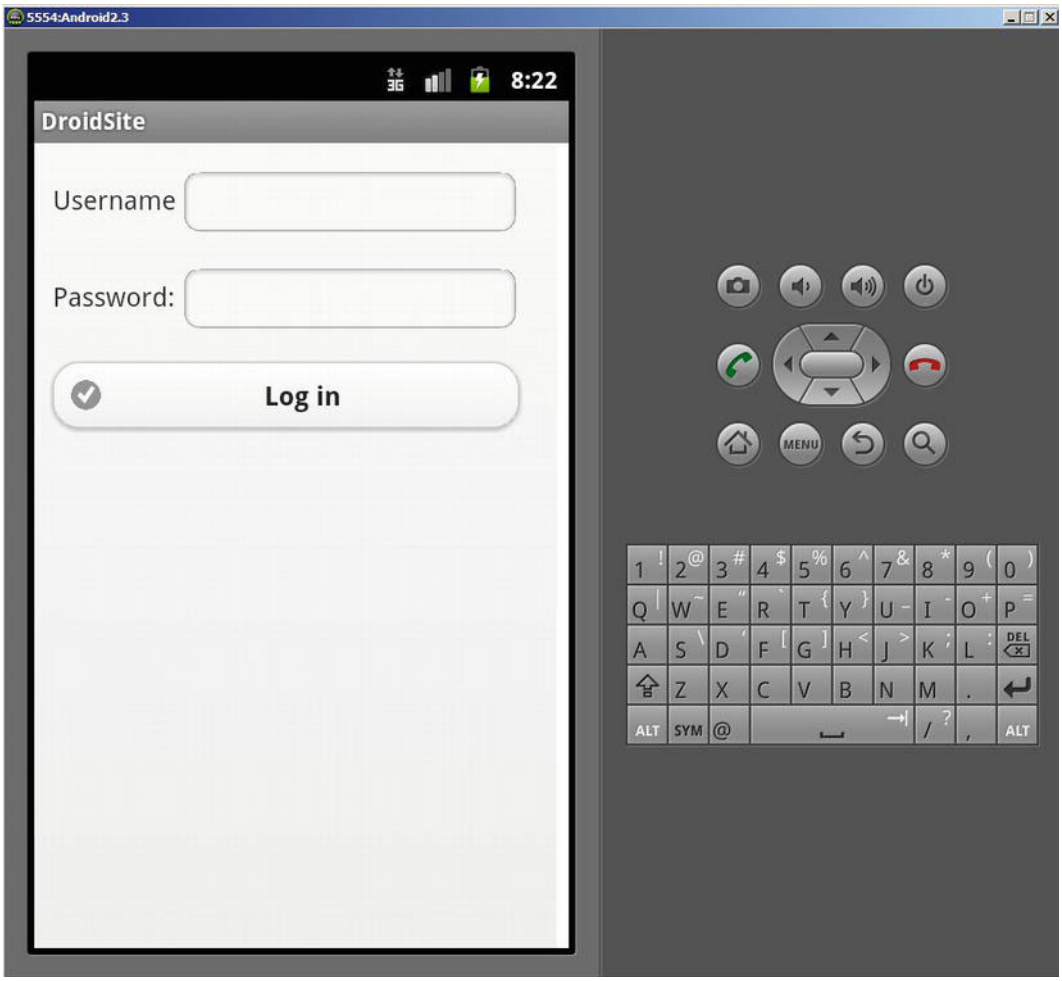

Figure 15-7. *Your native Android application running in the Android emulator*

Setting Up and Testing with Your Android Environment

To develop for Android, and before you get started with your application, you should make sure to have the following prerequisites installed and set up on your machine:

- The Java Development Kit (JDK) 7, available from `http://www.oracle.com/technetwork/java/javase/downloads/jdk-7u4-downloads-1591156.html`
- The Android SDK, available from `http://developer.android.com/sdk/index.html`
- Apache Ant, available from `http://ant.apache.org/`
- Set `JAVA_HOME` to the path of your JDK installation, such as `c:\Program Files\Java\jdk1.7.0`
- Set `ANDROID_SDK` to the path of your Android SDK, such as `c:\android-sdk`
- Set `ANT_HOME` to the path of your Ant installation, such as `c:\tools\apache-ant-1.8.4`
- Add the values of `%JAVA_HOME%`, `%ANDROID_SDK%`, `%ANDROID_SDK%\platform-tools`, and `%ANT_HOME%` to your `PATH`

Figure 15-8 shows the Android SDK Manager, the tool you can use to manage your installed Android APIs and their associated libraries. This tool will also notify you if you have updates to any library, or when a new API version or revision is published. Android has evolved quickly, and up to date, still enjoys a strong momentum of updates, with consecutive API versions introducing new and enriched capabilities and a wider reach of mobile devices to run on. Android 3.X or above is an appropriate choice for tablets, so if you

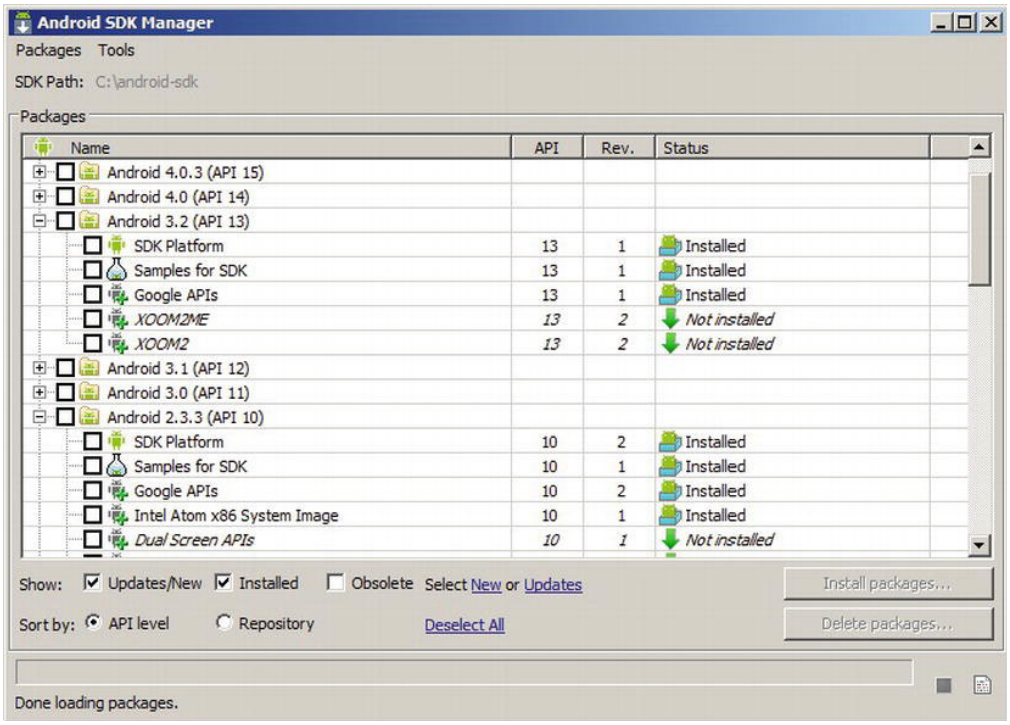

Figure 15-8. *The Android SDK Manager*

want to develop and test applications for Android tablets, choose and install an API 3.x or above. Older Android versions such as API 2.2 and API 2.3 are an appropriate choice for targeting smaller devices such as smart phones.

Once you have the appropriate Android APIs installed, you need to configure the Android virtual machines you want to target and test with.

This you can do by creating an Android Virtual Device using the Manage AVDs link in the Tools menu in the SDK Manager, or the separate AVD Manager tool installed with Android SDK. We recommend that you create an Android 2.3 and an Android 3.2 virtual device for the sample in this section, so you can test it on various formats such as smart phones and tablets.

Go ahead and click New… in the AVD Manager and create these new AVDs: an example configuration for Android 2.3 is shown in Figure 15-9. The Target dropdown is populated based on the APIs you have installed on your system. For each API, you can choose between various built-in skins that are available, or specify a custom resolution for your device. The Hardware section lists various properties for your device. Figure 15-9 contains a few properties that enable, among others, GPS support on the emulated device. You should consult the Android SDK documentation for a more in-depth treatment of communicating "hardcoded" settings such as your current location into your virtual devices, or interacting with them on the fly in various other ways.

A couple useful tools that you should be aware of while developing Android applications:

- `adb.exe` (in `%ANDROID_HOME%\platform-tools`) – the Android Debug Bridge. This is the tool/server that enables you to interact with a physical Android device connected to your machine in USB debug mode, or with a running virtual device. Running it without parameters will display a long list of options and parameters that control different aspects of communicating with the connected Android device. A few important commands are:
    - `adb.exe install <apk-package>` – installs the given `.apk` package. In general, this is the easiest and quickest way to install your WebSharper-generated `.apk` to your emulator or to your connected physical device.
    - `abd.exe uninstall <apk-package>` – uninstalls the given `.apk` package. Alternatively, you can also uninstall a package in the emulator itself via Settings \ Applications.
    - `adb.exe kill-server` – attempts to kill the Android Debug Bridge server and might be necessary if your device is not detected.
    - `adb.exe start-server` – attempts to start the Android Debug Bridge server on a given port and might be necessary if your device is not detected.
- `telnet.exe` (in `%SystemRoot%\System32`) – your old friend from the past, a client tool that enables you to communicate with the Android Debug Bridge using the Telnet communication protocol. Telnet can be installed via the system Control Panel, under Programs and Features, by turning on the Telnet feature. In general, you invoke it as `telnet.exe localhost <port>`, where `<port>` is the port number your Android Debug Bridge server is listening on.

  Once you have successfully connected to the Android bridge server, you can issue various commands. For instance, you can hardcode the GPS coordinates to the connected Android device with the following command:

  ```
  geo fix <long> <lat>
  ```

  Here, `<lat>` and `<long>` are the latitude and longitude values, respectively.

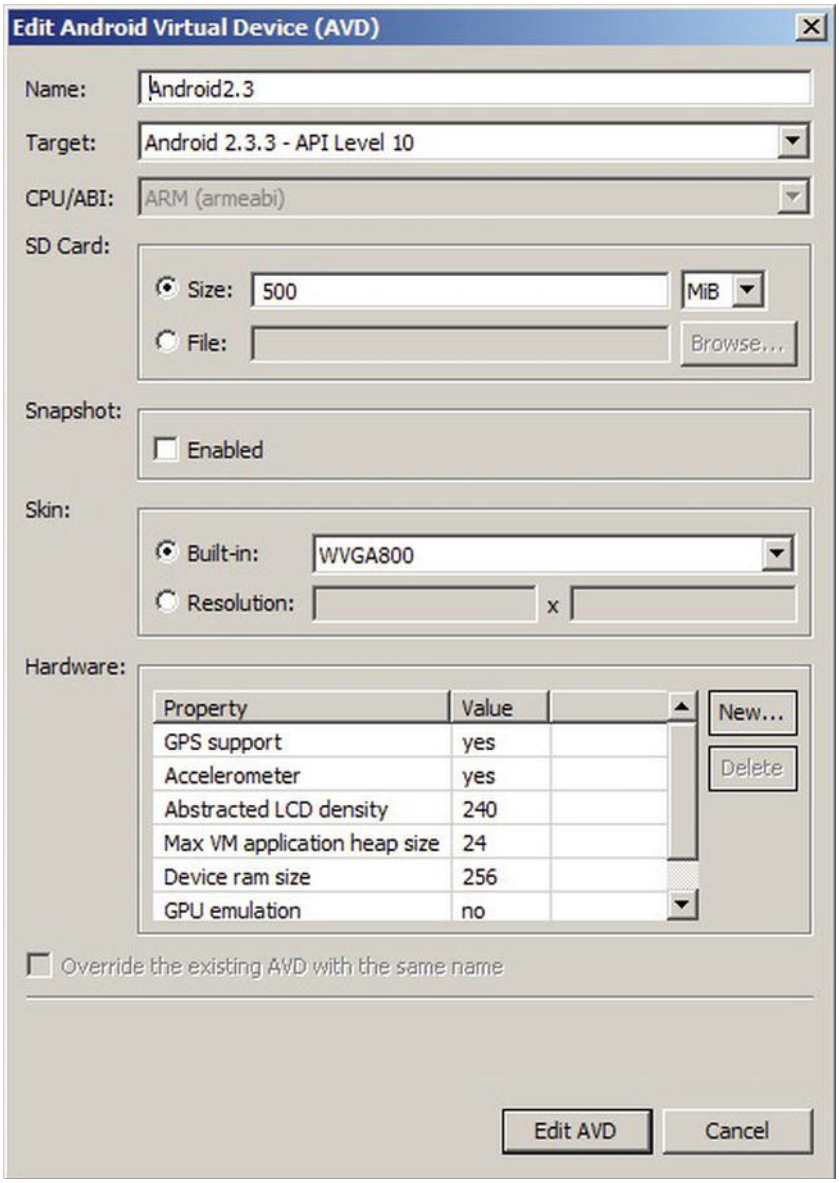

Figure 15-9. *Configuring an Android virtual device*

Using the Android Application Visual Studio Template

Now that you are familiar with installing different Android APIs and setting up virtual devices running those APIs, you can get started on your native Android application. First, choose the Android Application template in Visual Studio and create your new project. This template provides the necessary tooling to distill a set of JavaScript files and other artifacts from your application, and to package them into a native Android application, a technique employed by other popular tools like PhoneGap (`http://phonegap.com`).

PhoneGap provides packaging for several mobile platforms, including Android, Windows Phone, iOS, Blackberry, Bada, and others, and can be used in conjunction with WebSharper offline sitelets to package your WebSharper applications for any of these mobile platforms. You may want to use PhoneGap for packaging WebSharper applications for iOS, such as the ones you saw in earlier sections in this chapter, as iOS is not yet supported as a target native application format in WebSharper Mobile. Nonetheless, WebSharper Mobile provides a hassle-free packaging experience for Android and Windows Phone applications and is tightly coupled with WebSharper.

The Android Application template you used to create your application from contains an `android` folder with files that are related to packaging your final Android application. In particular, the `.project` file in the root of this folder contains the name of your Android package, set by default to `DroidSite`. You may want to rename this according to your needs later on.

Another important artifact in the `android` folder is `ant.properties`, a configuration file used in conjunction with building with `ant`. This file contains your signature settings for signing the resulting packages. By default, these settings are not configured and your application packages are unsigned. Therefore, before distributing your packages, you should generate a key store using the `keytool.exe` tool in your JDK, and configure `ant.properties` accordingly.

Building via `ant` is the preferred way of building Android packages, as the `msbuild` script shipped with the Android Application WebSharper template will trigger `ant` as part of the build process to bundle the final Android package automatically. Your resulting Android package will be under `android\bin`, whereas the distilled HTML+JavaScript code along with any additional artifacts that were packaged will be copied under `android\assets` to make it easier to investigate them offline.

Implementing Your Native Android Application

Other than the `android` folder you saw in the previous section, your empty Android application project has the same two files you saw in previous sections: `Main.fs` for your application code, and `Main.html` for your dynamic template markup. The latter is shown in Listing 15-7, and it may look quite familiar to you since it contains a barebones mobile-aware markup you used in other examples in this chapter.

Listing 15-7. `Main.html` *– Defining the Dynamic Template for the Android Application*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Your Android Application</title>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="generator" content="websharper" data-replace="scripts" />
  </head>
  <body>
    <div data-hole="body" />
  </body>
</html>
```

The main application code is shown in Listing 15-8. The code uses the jQuery Mobile formlets extension (which depends on the jQuery Mobile extension) and the Bing Maps extension, so you need to download and install WebSharper Extensions for jQuery Mobile, Formlets for jQuery Mobile, and Bing Maps, respectively, from the WebSharper download site (`http://websharper.com/downloads`), and add references to `IntelliFactory.WebSharper.JQuery.Mobile`, `IntelliFactory.WebSharper.Formlets.JQueryMobile`, `IntelliFactory.WebSharper.Bing`, and `IntelliFactory.WebSharper.Bing.Rest` from the folders you installed them into.

Listing 15-8. Main.fs– *Implementing your Android Application*

```
namespace MyApplication

open IntelliFactory.WebSharper
open IntelliFactory.WebSharper.Sitelets

module MySite =
    type Action = | Home

    module Skin =
        open System.IO

        type Page =
            {
                Body : list<Content.HtmlElement>
            }

        let MainTemplate =
            let path = Path.Combine(__SOURCE_DIRECTORY__, "Main.html")
            Content.Template<Page>(path)
                .With("body", fun x -> x.Body)

        let WithTemplate body : Content<Action> =
            Content.WithTemplate MainTemplate <| fun context ->
                {
                    Body = body context
                }

    module Client =
        open IntelliFactory.WebSharper.Html
        open IntelliFactory.WebSharper.Bing
        open IntelliFactory.WebSharper.JQuery
        open IntelliFactory.WebSharper.JQuery.Mobile
        open IntelliFactory.WebSharper.Formlet
        open IntelliFactory.WebSharper.Formlets.JQueryMobile

        [<JavaScript>]
        let BingMapsKey = "<put-your-bing-maps-key-here>"

        [<JavaScript>]
        let ShowMap () =
            let screenWidth = JQuery.Of("body").Width()
            let MapOptions = Bing.MapViewOptions(
                                Credentials = BingMapsKey,
                                Width = screenWidth - 20,
                                Height = screenWidth - 40,
                                Zoom = 16)
            let label = Span []
            let setMap (map : Bing.Map) =
                let updateLocation() =
```

```
            // Gets the current location
            match Android.Context.Get() with
            | Some ctx ->
                if ctx.Geolocator.IsSome then
                    async {
                        let! loc = ctx.Geolocator.Value.GetLocation()
                        // Sets the label to be the address of
                        // the current location.
                        Rest.RequestLocationByPoint(
                            BingMapsKey,
                            loc.Latitude, loc.Longitude, ["Address"],
                            fun result ->
                                let locInfo = result.ResourceSets.[0].Resources.[0]
                                label.Text <-
                                    "You are currently at " +
                                    JavaScript.Get "name" locInfo)
                        // Sets the map at the current location.
                        let loc =
                            Bing.Location(loc.Latitude, loc.Longitude)
                        let pin = Bing.Pushpin loc
                        map.Entities.Clear()
                        map.Entities.Push pin
                        map.SetView(Bing.ViewOptions(Center = loc))
                    }
                    |> Async.Start
                else
                    ()
            | None ->
                ()
        JavaScript.SetInterval updateLocation 1000 |> ignore
    let map =
        Div []
        |>! OnAfterRender (fun this ->
            let map = Bing.Map(this.Body, MapOptions)
            map.SetMapType(Bing.MapTypeId.Road)
            setMap map)
    Div [
        label
        Br []
        map
    ]

[<JavaScript>]
let LoginSequence () =
    Formlet.Do {
        let! username, password =
            Formlet.Yield (fun user pass -> user, pass)
            <*> (Controls.TextField "" Enums.Theme.C
                |> Enhance.WithTextLabel "Username"
                |> Validator.IsNotEmpty "Username cannot be empty!")
            <*> (Controls.Password "" Enums.Theme.C
```

```
                        |> Enhance.WithTextLabel "Password: "
                        |> Validator.IsRegexMatch "^[1-4]{4,}[5-9]$"
                                                 "The password is wrong!")
                    |> Enhance.WithSubmitButton "Log in" Enums.Theme.C
                do! Formlet.OfElement (fun _ ->
                    Div [
                        H3 [Text ("Welcome " + username + "!")]
                        ShowMap()
                    ])
            }
            |> Formlet.Flowlet

        type ApplicationControl() =
            inherit Web.Control()

            [<JavaScript>]
            override this.Body =
                Div [LoginSequence ()] :> _

    module Pages =
        open IntelliFactory.Html

        let Home =
            Skin.WithTemplate <| fun ctx ->
                [
                    Div [
                        HTML5.Data "role" "page"
                        Id "main"
                        HTML5.Data "url" "main"
                    ] -< [
                        new Client.ApplicationControl()
                    ]
                ]

    type MyWebsite() =
        interface IWebsite<Action> with
            member this.Sitelet =
                Sitelet.Content "/index" Action.Home Pages.Home
            member this.Actions = [ Action.Home ]

[<assembly: Website(typeof<MySite.MyWebsite>)>]
do ()
```

In order to be able to run your application, you need to configure your own Bing Maps key in `Client.BingMapsKey`. To obtain such a key, you should consult the Bing Maps home page. This key enables you to make programmatic queries to the Bing Maps service, such calling the various REST APIs provided by WebSharper Extensions for Bing Maps.

The main part of the application is in the `Client` module. There are two distinct components: one for showing a map marked with the user's current GPS location, and another for displaying a login form that "authenticates" the current user to see this map. The main "application" control, defined with the

ApplicationControl server control type, simply exposes this login formlet, which then in turn invokes the map if the user passed the authentication step.

The ShowMap() function creates a node, with a label and the map component. The map itself is embedded in a nested node, with an event handler firing once the corresponding DOM node has been inserted into and rendered in the document the user sees:

```
let map =
    Div []
    |>! OnAfterRender (fun this ->
        let map = Bing.Map(this.Body, MapOptions)
        map.SetMapType(Bing.MapTypeId.Road)
        setMap map)
...
```

This code creates a Bing Map control with the options defined in MapOptions, sets its map type to road map, and calls setMap on it. In turn, setMap defines an update function updateLocation and registers it to fire every second using the JavaScript.SetInterval function. The heart of the map functionality, updateLocation retrieves the GPS geo-locator object from the Android context (if either is missing, it does nothing – here, you may want to add some fallback logic yourself to make the application testable on plain non-mobile browsers, etc.), and uses this object to retrieve the device's current GPS coordinates, then queries the Bing Maps service to map those to an actual street name, and displays that in the label component along with a pushpin on the map itself.

The formlet code is even more intuitive. Take a look at the partial snippet below:

```
Formlet.Do {
    let! username, password =
        Formlet.Yield (fun user pass -> user, pass)
        <*> <... formlet-1 ...>
        <*> <... formlet-2 ...>
        |> Enhance.WithSubmitButton "Log in" Enums.Theme.C
    do! Formlet.OfElement (fun _ ->
        Div [
            H3 [Text ("Welcome " + username + "!")]
            ShowMap()
        ])
}
|> Formlet.Flowlet
```

Here, "formlet-1" and "formlet-2" declare two input boxes, enhanced with a label and a validator to input the user name and the password strings from the user. Note that these validators require that the user enters a non-empty user name, and a password that contains at least four digits between 1 and 4 followed by a digit between 5 and 9, for instance, "12345". These input boxes are then composed into a single formlet, returning a (username, password) tuple, and enhanced with a submit button. Here, all three functions Controls.TextField, Controls.Password, and Enhance.WithSubmitButton are from the IntelliFactory.WebSharper.Formlets.JQueryMobile namespace, and implement their functionality using jQuery Mobile look and feel.

The username/password formlet is enhanced as a flowlet; e.g., its individual formlet steps (marked with let!) are executed in a sequential wizard-style presentation, one following the other. In your formlet example above, once the user enters a username/password pair and successfully passes the validations after pressing the Submit button, control transfers to the do! block, which responds by welcoming the newly signed-in user and showing the Bing Map control defined in ShowMap.

Summary

This chapter gave you a short introduction to developing web and native mobile applications with WebSharper. You saw how a new breed of mobile applications is emerging, utilizing HTML5 and CSS3; how you can output this sort of mobile HTML5 markup from WebSharper sitelets; apply various feature detection and polyfilling libraries to get over missing browser features; and develop iOS mobile web applications that use easy-to-embed, Safari-specific markup instructions to mimic some of the features of native mobile applications, such as the ability to use the entire screen or adding web applications as icons to the desktop. In the latter half of the chapter, you also saw how you can develop WebSharper extensions to third-party JavaScript APIs and use these extensions in a mobile web application to display a user's Facebook status updates with jQuery Mobile-based list views, giving an elegant mobile look to your application. And last, you saw how you can combine the expressiveness of WebSharper formlets with mobile UI controls to build declarative, composable, type-safe mobile user interfaces and apply them in an application that uses Bing Maps to show your location.

All in all, you learned that the functional concepts such as formlets and sitelets that WebSharper enables yield a powerful device that you can employ to develop mobile web applications, and together with WebSharper Mobile, package them into native application packages for Android and Windows Phone, or use third-party packaging technologies such as PhoneGap (`http://phonegap.com`) to cover alternate platforms. At this point, you have keen eyes on upcoming HTML5 support of various mobile features, and recognize that it is only a short time away until mobile browser features are unified in further HTML5 standards to bring uniformity into developing for touch-based mobile devices.