Functional Web and Mobile Development

Adam Granicz, IntelliFactory

Downloading and installing WebSharper

Download the Visual Studio extension from http://websharper.apphb.com

In the ZIP file, there is a VSIX file – this is the Visual Studio extension installer. Once you run it, it will install the WebSharper basic project templates into your Visual Studio.

These templates installed are:

- Extension – starts a new WebSharper extension project. Extensions are "bindings" for various JavaScript libraries.
- Library – starts a WebSharper library project. Libraries are packages that can contain server and client-side WebSharper functionality for reuse.
- Sitelet Host Website – starts an ASP.NET host web project. Host projects are used to contain WebSharper applications (sitelets).
- Sitelet Html Generator – starts a client-only sitelet project that produces HTML and JavaScript.
- Sitelet Website Definition – starts a sitelet project that can contain client and/or server side functionality.

Your first WebSharper application

You can create a simple WebSharper application by embedding a simple sitelet into a host project. Just follow these simple steps:

1) Create a new Sitelet Host Website project to host your WebSharper application. This will create a solution as well.
2) Add to your solution a new Sitelet Website Definition project to contain the logic of your WebSharper application.
3) Add the sitelet project as a project reference to your host project.
4) Set the host project as your start-up project and run your solution.
5) You should see a simple page with Home/About links, each pointing to their respective pages.

What is a sitelet?

A sitelet is a WebSharper abstraction to represent web applications. Sitelets are F# values, and they can be programmatically created (`Sitelet.Content`, `Sitelet.Infer`, `Sitelet.Protect`) and composed into larger sitelets (`Sitelet.Sum`).

Sitelets go one step further in bridging the gap between the server and client by allowing server-side HTML to be constructed by combinators similar to those used in its WebSharper client-side counterpart. These combinators allow you to embed WebSharper controls, making sitelets a perfect tool set to create dynamic, markup-less web applications.

You will benefit from using sitelets by:

- Removing the need for managing static files.
- Being able to dynamically construct pages and serve arbitrary content.
- Having full controls of your URLs by specifying your custom routers for linking to content.
- Compose contents into sitelets, which may themselves be composed into bigger sitelets.
- Having safe links for referencing other content contained within your site.
- Being able to use the type-safe HTML templating facilities that come with sitelets.

Constructing a simple sitelet-based application

Below is a minimal example of a complete site serving one HTML page:

```
namespace SampleWebsite

open IntelliFactory.WebSharper.Sitelets

module SampleSite =
    open IntelliFactory.WebSharper
    open IntelliFactory.Html

    type Action = | Index

    let Index : Content<Action> =
        PageContent <| fun context ->
            { Page.Default with
                Title = Some "Index"
                Body =
                    let time = System.DateTime.Now.ToString()
                    [H3 [Text <| "Current time: " + time]]}

    type MySampleWebsite() =
        interface IWebsite<Action> with
            member this.Sitelet =
                Sitelet.Content "/" Action.Index Index
            member this.Actions = []

[<assembly: Website(typeof<SampleSite.MySampleWebsite>)>]
do ()
```

In the example above, first a custom action type is defined. It is used for linking URLs to content within your sitelet. Here, you only need one action corresponding to your single page.

The content of the index page is defined as a `PageContent`, a `Content` constructor, where the body consists of a simple server side HTML element. Here the current time is computed and displayed within an `H3` tag.

The `MySampleWebsite` type specifies the sitelet to be served by implementing the `IWebsite` interface. In this case, the sitelet is defined using the `Sitelet.Content` combinator which constructs a sitelet for the index page content. In the resulting sitelet, the `Action.Index` value is associated with the root path (/) and the given content.

Serving content of other types

Contents are not restricted to produce HTML. To change the content type and encoding, you can customize the meta information that drives the HTTP headers of the response. Below is an example of defining JSON data.

```
let JsonData : Content<Action> =
    CustomContent <| fun context ->
        {
            Status = Http.Status.Ok
            Headers = [Http.Header.Custom "Content-Type" "application/json"]
            WriteBody = fun stream ->
                use tw = new System.IO.StreamWriter(stream)
                tw.WriteLine "{X: 10, Y: 20}"
        }
```

The building blocks of sitelets

Sitelets are parameterized by a type representing *actions*. The action type is typically user-defined, and encodes all the possible ways to link to content on the site. Instead of linking to content using string URLs, the URLs are inferred by linking to values of your action type.

A sitelet is constituted of two parts; a *router* and a *controller*. The job of the router is to map actions to URLs and to map HTTP requests to actions. The controller is responsible for handling actions, by converting them into *content* that in turn produces the HTTP response. The overall architecture is analogous to ASP.NET (MVC), and other *Model-View-Controller* based web frameworks.

Routers

The router component of a sitelet can be constructed in a variety of ways. The following example shows how you can create a complete customized router of type `Action`.

```
type Action = | Page1 | Page2

let MyRouter : Router<Action> =
    let route (req: Http.Request) =
        if req.Uri.LocalPath = "/page1" then
            Some Page1
        elif req.Uri.LocalPath = "/page2" then
            Some Page2
        else
            None
    let link action =
        match action with
        | Action.Page1 ->
            System.Uri("/page1", System.UriKind.Relative)
            |> Some
        | Action.Page2 ->
            System.Uri("/page1", System.UriKind.Relative)
            |> Some
    Router.New route link
```

Specifying routers manually gives you full control of how to parse incoming requests and to map actions to corresponding URLs. It is your responsibility to make sure that the router forms a bijection of URLs and actions, so that linking to an action produces a URL that is in turn routed back to the same action.

Luckily, constructing routers manually is only required for very special cases. The above router can for example be generated using `Router.Table`:

```
let MyRouter : Router<Action> =
    [
        Action.Page1, "/page1"
        Action.Page2, "/page2"
    ]
    |> Router.Table
```

Even simpler, the routing table can be inferred automatically for basic F# types, including *tuples*, *records* and *unions*.

```
let MyRouter : Router<Action> =
    Router.Infer ()
```

Controllers

If an incoming request can be mapped to an action by the router, it is passed on to the controller. The job of the controller is to map actions to content. Here is an example of a controller handling actions of the `Action` type defined above.

```
let MyController : Controller<Action> =
    {
        Handle = fun action ->
            match action with
            | Action.Page1  -> Page1Content
            | Action.Page2  -> Page2Content
    }
```

Finally, the router and the controller components are combined into a sitelet:

```
let MySitelet : Sitelet<Action> =
    {
        Router = MyRouter
        Controller = MyController
    }
```

Content

Content is conceptually a function from a context to an HTTP response. For convenience it differentiates between normal content and ones producing HTML pages:

```
type Content<'Action> =
    | CustomContent of (Context<'Action> -> Http.Response)
    | PageContent   of (Context<'Action> -> Page)
```

Values of type *Context* contain run time information of how to resolve links to actions and resources.

The example below defines a page content with a link to another page:

```
let Page1 : Content<Action> =
    PageContent <| fun context ->
        { Page.Default with
            Title = Some "Title of Page 1"
```

```
            Body =
                [
                    h3 [Text "Page 1"]
                    A [HRef (context.Link Action.Page2)] -< [Text "Page 2"]
                ]
        }
```

Note how `context.Link` is used in order to resolve the URL to the *Page2* action.

Sitelet combinators

Combinators found in the `Sitelet` module provide means of constructing and composing sitelets.

The `Sitelet.Content` function generates a sitelet with a router that simply links a path with an action, and a controller that will always respond with the given content. Here is an example of constructing a complete sitelet serving one page:

```
let IndexSitelet = Sitelet.Content "/index" Action.Index Index
```

The `<|>` operator combines two sitelets into one. The resulting sitelet will try to map an incoming request using the router of the first sitelet. If this router fails to map the request, it is forwarded to the second sitelet. Here is an example of composing three sitelets:

```
let Site =
    Sitelet.Content "/index" Action.Index Index
    <|>
    Sitelet.Content "/page1" Action.Page1 Page1
    <|>
    Sitelet.Content "/page2" Action.Page2 Page2
```

Alternatively, you can use the `Sitelet.Sum` function to compose a sequence of sitelets:

```
let Site =
    Sitelet.Sum [
        Sitelet.Content "/index" Action.Index Index
        Sitelet.Content "/page1" Action.Page1 Page1
        Sitelet.Content "/page2" Action.Page2 Page2
    ]
```

The `Sitelet.Shift` operator is used to shift the URL of a sitelet by adding a prefix:

```
let Pages =
    Sitelet.Sum [
        Sitelet.Content "/page1" Action.Page1 Page1
        Sitelet.Content "/page2" Action.Page2 Page2
    ]
    |> Sitelet.Shift "/pages"
```

In this way, the URL of the `Page1` action will be inferred to `/pages/page1`.

Embedding client-side controls

WebSharper Sitelets offers a solution for defining the server-side content, but how does it interplay with the client-side components? The integration of WebSharper controls (i.e. code that translates to JavaScript and runs on the client) is straight forward. They can be directly embedded within server-side HTML:

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

    type MyControl() =
        inherit IntelliFactory.WebSharper.Web.Control ()
        [<JavaScript>]
        override this.Body =
            I [Text "Client control"] :> IPagelet

let Page : Content<Action> =
    PageContent <| fun context ->
        { Page.Default with
            Title = Some "Index"
            Body =
                [
                    Div [new Client.MyControl ()]
                ]
        }
```

Here, `MyControl` inherits from `IntelliFactory.WebSharper.Web.Control` and overrides the `Body` property with some client-side HTML. This control is then placed within a server-side DIV tag.

Using HTML templates

WebSharper supports two kinds of templating: static and dynamic. Static templates are processed at compile time and translated into F# functions. In practice, static templates trade type safety with flexibility, and are thus less practical, so most applications use dynamic templating. Dynamic templates are bound at runtime but still provide a limited form of type safety. Consider the following example for a new dynamic template based on a single placeholder Body:

```
module Pages =

    type Index =
        {
            Body : Content.HtmlElement list
        }

    let IndexTemplate =
        Content.Template(__SOURCE_DIRECTORY__ + "/Main.html")
            .With("body", fun x -> x.Body)
```

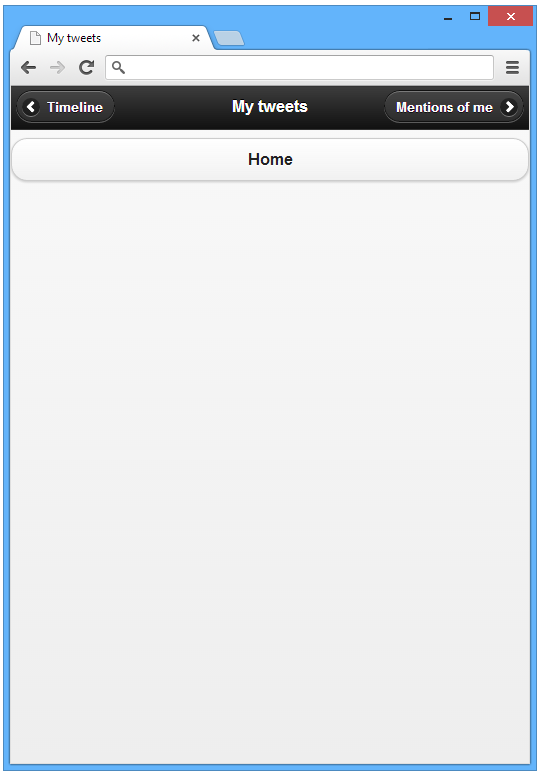
This can now be instantiated with content as follows:

```
    open IntelliFactory.Html
```

```
    let IndexPage =
        Content.WithTemplate IndexTemplate <| fun ctx ->
            { Body = [Div [...]] }
```

Example – A simple mobile application with jQuery Mobile

You can create an HTML WebSharper application project and paste in the following two files to create a simple jQuery Mobile application with sliders to span over a handful of pages. You need to plant in the WebSharper extension for jQuery Mobile (or even WebSharper), you can do so with NuGet using the following package.config file:




```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="WebSharper" version="2.5.10-alpha"
targetFramework="net40" />
  <package id="WebSharper.JQuery.Mobile" version="2.5.0-
alpha" targetFramework="net40" />
</packages>
```

Alternatively, you can execute the appropriate NuGet InstallPackage command on demand. The resulting application is shown here:

SlideApp.fs

```
[<IntelliFactory.WebSharper.Pervasives.JavaScript>]
module SlideApp

open System.Collections.Generic
open IntelliFactory.WebSharper
open IntelliFactory.WebSharper.JQuery
open IntelliFactory.WebSharper.JQuery.Mobile
open IntelliFactory.WebSharper.Html

[<AutoOpen>]
module private Internal =

    let JQM = Mobile.Instance

    type Transition =
        | NoTransition
        | SlideLeft
        | SlideRight
        | SlideDown
        | SlideUp

        member this.Reverse =
            match this with
            | SlideUp
            | SlideLeft -> true
            | NoTransition
            | SlideDown
            | SlideRight -> false
```

```
    member this.Name =
        match this with
        | NoTransition -> "none"
        | SlideLeft
        | SlideRight -> "slide"
        | SlideUp
        | SlideDown -> "slidedown"

type PageManager<'Page when 'Page : equality>() =

    let rendered = Dictionary<'Page, Element>()
    let mutable setupPage : 'Page -> Element = fun _ -> Div []

    member this.SwitchTo (p: 'Page, ?trans: Transition) =
        if not (rendered.ContainsKey p) then
            rendered.[p] <- setupPage p
            JQuery.Of("body").Append(rendered.[p].Body).Ignore
            (rendered.[p] :> IPagelet).Render()
        let trans = defaultArg trans NoTransition
        JQM.ChangePage(JQuery.Of(rendered.[p].Body),
            ChangePageConfig(
                Transition = trans.Name,
                Reverse = trans.Reverse))

    member this.Setup(setup: PageManager<'Page> -> 'Page -> Element) =
        setupPage <- setup this

let PageDiv content =
    Div [HTML5.Attr.Data "role" "page"] -< content
    |>! OnAfterRender (fun el ->
        Mobile.Page.Init (JQuery.Of el.Body))

let OnSwipeLeft f (e: #IPagelet) =
    JQuery.Of(e.Body).On("swipeleft", fun _ -> f e; true)

let OnSwipeRight f (e: #IPagelet) =
    JQuery.Of(e.Body).On("swiperight", fun _ -> f e; true)

let Header x = Div [HTML5.Attr.Data "role" "header"] -< x

let PageCarousel (pages: seq<string * #seq<Element>>) =
    let pages = Array.ofSeq pages
    let n = pages.Length
    fun (pm: PageManager<int>) (i: int) ->
        let i = i % n
        let ``i-1`` = (i+n-1) % n
        let ``i+1`` = (i+1) % n
        let title, content = pages.[i]
        let prevTitle, _ = pages.[``i-1``]
        let nextTitle, _ = pages.[``i+1``]
        let goPrev() = pm.SwitchTo(``i-1``, SlideLeft)
        let goNext() = pm.SwitchTo(``i+1``, SlideRight)
        PageDiv [
            yield Header [
```

```
                    Button [
                        HTML5.Attr.Data "icon" "arrow-l"
                        HTML5.Attr.Data "iconpos" "left"
                        Text prevTitle
                    ]
                    |>! OnClick (fun _ _ -> goPrev())
                    H1 [Text title]
                    Button [
                        HTML5.Attr.Data "icon" "arrow-r"
                        HTML5.Attr.Data "iconpos" "right"
                        Text nextTitle
                    ]
                    |>! OnClick (fun _ _ -> goNext())
                ]
                yield! content
            ]
            |>! OnSwipeLeft (fun _ -> goNext())
            |>! OnSwipeRight (fun _ -> goPrev())

let Init() =
    let carousel = PageManager<int>()
    let home = PageManager<unit>()
    let homeButton() =
        Button [Text "Home"]
        |>! OnClick (fun _ _ -> home.SwitchTo((), SlideDown))
    let carouselPages =
        [
            "Timeline", [
                homeButton()
            ]
            "My tweets", [
                homeButton()
            ]
            "Mentions of me", [
                homeButton()
            ]
        ]
    carousel.Setup(PageCarousel carouselPages)
    home.Setup(fun _ () ->
        PageDiv [
            yield Header [H1 [Text "Home"]]
            yield! carouselPages |> List.mapi (fun i (title, _) ->
                Button [Text title]
                |>! OnClick (fun _ _ -> carousel.SwitchTo(i, SlideUp)))
        ])
    home.SwitchTo(())
```

Main.fs

This module creates a sitelet and adds the jQuery Mobile page manager functionality in a single sitelet page.

```
namespace SlideAppForJQueryMobile

open IntelliFactory.WebSharper
open IntelliFactory.WebSharper.Sitelets
```

```
type Action = | Index

module Client =

    open IntelliFactory.WebSharper.Html

    [<Sealed>]
    type Control() =
        inherit Web.Control()
        [<JavaScript>]
        override this.Body =
            Div []
            |>! OnAfterRender (fun _ ->
                SlideApp.Init())
            :> _

module Pages =

    type Index =
        {
            Body : Content.HtmlElement list
        }

    let IndexTemplate =
        Content.Template(__SOURCE_DIRECTORY__ + "/Main.html")
            .With("body", fun x -> x.Body)

    open IntelliFactory.Html

    let Index =
        Content.WithTemplate IndexTemplate <| fun ctx ->
            { Body = [Div [new Client.Control()]] }

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

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