WebSharper™ Manual

Version 2.4

April 23, 2012

This manual describes the WebSharper™ web application development platform, including the F# to JavaScript compiler and the accompanying libraries and tools. It is intended as a reference for developers and assumes a basic familiarity with F#, JavaScript and web development.

This is the 2.4 version of the manual. It is so named to match the current version of WebSharper™.

1 Overview

WebSharper™ is an F#-based web programming platform. It lets you develop web applications from a single F# codebase, to be run in the .NET environment on the server, or the JavaScript environment on the browser, or both.

WebSharper™ enjoys the type safety of F# and the code completion of Microsoft Visual Studio, making it a very productive envionment for developing JavaScript. WebSharper™ server-side components make communicating with the server is as easy as calling a function.

WebSharper™ includes:

- A compiler from assemblies to JavaScript.
- Extensible support for a large part of the F# and .NET standard library (including sequences, events and asynchronous workflows) on the client.
- Support for type-safe programming with the standard JavaScript library and DOM.
- F# bindings to a number of third-party libraries including jQuery.
- Formlets - an innovative library for type-safe web form combinators.
- Support for seamless client/server communication.
- Tools for embedding raw JavaScript code and binding to external JavaScript codebases from F#.
- Integration with ASP.NET and Microsoft Visual Studio.
- Resource dependency management for CSS, image and other supporting files.

2 Developing JavaScript Libraries

Developing JavaScript libraries with WebSharper™ is as simple as writing F# code, annotating it with a few custom attributes, compiling it with F#, and running WebSharper™ to generate the JavaScript:

```
module MyModule =
  [<JavaScript>]
  let rec Factorial n =
    match n with
    | 0 -> 1
    | n -> n * Factorial (n - 1)
```

This section provides the overview of the required custom attributes, the F# language features and the F# standard library functions supported by WebSharper™ in the JavaScript environment.

2.1 Member Annotations

There are required and optional custom attribute annotations that influence how F# code gets compiled to JavaScript. To be useable from the client-side code, any member must be annotated with either the `JavaScript` attribute or one of the attributes inheriting from `AbstractInlineAttribute`. To customize the name in the compiled JavaScript output, it might also be annotated with an attribute inheriting from `AbstractNameAttribute`.

JavaScriptAttribute

`JavaScriptAttribute` marks members for compilation into JavaScript. It is the single most important attribute in WebSharper™. The annotated members are translated to JavaScript by the WebSharper™ compiler by inspecting and translating their F# bodies.

For example:

```
module MyModule =
  [<JavaScript>]
  let rec Factorial n =
    match n with
    | 0 -> 1
    | n -> n * Factorial (n - 1)
```

The attribute is implemented as an alias for the `ReflectedDefinitionAttribute` that comes with F#. The F# comiler recognizes members marked with this attribute and stores their reflected (abstract syntax tree or quotation) form within the resulting assembly, in addition to compiling them to IL as regular members. WebSharper™ compiler is then able to find the quoted form of the members and translate them to JavaScript.

Naming Attributes

These attributes influence the member names as in JavaScript. The base class, `Naming.AbstractNameAttribute`, allows to create custom attributes with arbitrary logic for determining the compiled name. This is useful to avoid naming clashes.

A simple implementation, the `NameAttribute`, explicity sets the JavaScript-compiled names of members and types. For example:

```
[<Stub>]
[<Name "my.package.Date">]
type Date =
    /// Returns the day of the month.
    [<Name "getDate">]
    member this.GetDate() = 0
```

Inlining Attributes

Inlining attributes mark functions for inline compilation to JavaScript. The base class, `Inlining.AbstractInlineAttribute`, allows to create custom attributes with arbtirary macro-expansion logic. Three common forms are provided: `InlineAttribute`, `ConstantAttribute` and `StubAttribute`.

InlineAttribute

`InlineAttribute` is a simple attribute that specifies that members are to be compiled inline. This attirbute either complements the `JavaScriptAttribute`, or serves standalone with a JavaScript template string. The following two forms are equivalent:

```
[<Inline>]
[<JavaScript>]
let Add (x: int) (y: int) = x + y
[<Inline "$x + $y">]
let Add (x: int) (y: int) = 0
```

The sytnax of the template string is regular JavaScript. Variables that start with `$` are treated as placeholders. There are named (`$x`), positional (`$0`), and special (`$this`, `$value`) placeholders. To use an actual variable that starts with a `$` sign, duplicate the sign, as in `$$x`.

ConstantAttribute

`ConstantAttribute` allows members to compile to constant literals. Its most common use is to annotate union cases. For example:

```
type Align =
  | [<Constant "left">]   Left
  | [<Constant "center">] Center
  | [<Constant "right">]  Right
```

With these annotations, `Align.Left` is compiled as literal `"left"`, and pattern-matching against any union case is compiled as an equality test against the corresponding literal.

This pattern is useful for providing type safety for JavaScript code.

StubAttribute

This attribute commonly marks types that expose JavaScript-implemented functionality to WebSharper™. `StubAttribute` is useful for enabling WebSharper™ code to consume and interoperate with legacy and third-party JavaScript code.

Methods and fields on types marked with `StubAttribute` that are not marked with special translation attributes such as `JavaScriptAttribute` are translated by-name. Methods do not have to have a meaningful body, but should be correctly typed.

Sample usage:

```
[<Name [| "Date" |]>]
[<Stub>]
type Date() =
  /// Returns the day of the month.
  member this.getDate() = 0
  /// Returns the day of the week.
```

```
    member this.getDay() = 0
    /// Returns the year.
    member this.getFullYear() = 0
```

The above example exposes to F# code some of the functionality of the `Date` object as present in most JavaScript environments (and specified in ECMA-262 3rd ed.).

2.2 F# Language Coverage

Most of the F# language features are directly supported by WebSharper™. The philosophy is to produce readable, straightforward JavaScript code, making it possible to analyze the output and use it from external JavaScript code or apply JavaScript-targeting tools.

2.2.1 Data Representation

In general, matching JavaScript data types are reused, where possible, to represent F# data types:

- F# numbers, booleans, strings, and arrays are represented directly as their JavaScript counterparts.
- F# lambda expressions are directly compiled to JavaScript lambda expressions.
- F# algebraic data types are represented as JavaScript objects. In particular, tuples are represented as arrays, records as objects with matching field names, and unions as objects with field names of the form `$n` where `n` is the field position.
- F# objects are represented as JavaScript objects, with fields and members as JavaScript fields.

JavaScript representations of F# data types:

F#	JavaScript
`null`	`null`
`"foo"`	`"foo"`
`true`	`true`
`1`	`1`
`1.25`	`1.25`
`(1, 2)`	`[1,2]`
`[| 1 |]`	`[1]`
`None`	`{$: 0}`
`Some 1`	`{$: 0, $0: 1}`
`[1,2,3]`	`{$: 1, $0: 1, $1: {$: 1, $0: 2, $1: {$: 1, $0: 3, $2: @;{$: 0}}}}`
`{A = 1}`	`{A: 1}`

Arrays

Arrays are represented directly as JavaScript arrays. Multi-dimensional arrays are not currently supported.

Tuples

Tuples are represented as JavaScript arrays.

F# Records

Records are represented as JavaScript objects.

For example, consider this code:

```
type R =
  {
    a: int
    b: string
  }
[<JavaScript>]
let F() =
  { a = 1; b = "foo" }
```

This will compile to:

```
function F() {
  return {a: 1, b: "foo"};
}
```

F# Unions

Unions are represented as JavaScript objects, with fields for the union case tag and every field. Consider:

```
type U =
  | A
  | B of int * string
[<JavaScript>]
let F() = [| U.A; U.B(1, "!") |]
```

This will compile to:

```
function F() {
  return [{$: 0}, {$: 1, $1: 1, $2: "!"}];
}
```

Enumerations

Enumerations with integer values are supported for both construction and pattern-matching.

For example, consider this code:

```
type E =
  | One = 1
  | Two = 2
[<JavaScript>]
let F () = E.One
```

It will compile to:

```
function F() {
  return 1;
}
```

Enumerations do not require any annotations. Enumeration support includes standard enumerations such as `System.DayOfWeek`.

2.2.2 Functional Features

JavaScript is a functional language and therefore allows most of the F# features to be represented directly. Two notable omissions are its lack of support for static typing and tail-call optimization.

- First-class functions and closures map directly to their JavaScript counterparts.
- Methods are uncurried during compilation. For example:

  ```
  let f x y z = ...
  ```

  This translates to the equivalent of:

  ```
  function f(x, y, z) ...
  ```

- Curried lambda functions a translated directly:

  ```
  (fun x y -> ...)
  ```

  Translates to:

  ```
  function (x) { return function (y) { ... }}
  ```

- Lambda functions that accept tuples are compiled to accept either a single or multiple arguments. When called with a single argument, the function expects the argument to be an array representing the tuple. When called with multiple arguments, it assumes the arguments to be the tuple components. For example:

  ```
  let f = (fun (x, y) -> ...)
  ```

  Given the above definition, JavaScript can call `f` in two ways:

  ```
  f(1, 2)
  f([1, 2])
  ```

- Types are erased during compilation.
- Tail call optimization is not currently supported. Future versions of WebSharper™ will support it with a combination of local optimizations that transform recursive functions to loops and a global optimization with trampolining.

2.2.3 Object-Oriented Features

- Inheritance is modelled with JavaScript prototype chains. For example:

  ```
  type A [<JavaScript>]() =
    class end
  type B [<JavaScript>] () =
    inherit A()
  ```

  Translates to the equivalent of:

  ```
  function A() {...}
  function B() {...}
  B.prototype = new A();
  ```

  Chaining the prototypes allows JavaScript objects to inherit instance members and interface implementations from the superclasses.

- Interfaces are supported structurally. A JavaScript object is assumed to implement an interface if it has methods with matching names. Just as types, interfaces are therefore an F#-level concept that gets erased during compilation. Type tests against interfaces are not compiled. While F# allows a class to implement two interfaces with clashing method names, using distinct method implementations for each, it is an error to do so in WebSharper™.

Equality and Hashing

JavaScript notion of pointer equality does not match structural equality required by F#. Moreover, JavaScript does not provide a generic hash primitive, `obj -> int`. To model these F# features, WebSharper™:

- Implements object hashing by destructively modifying the hashed object and assigning a freshly generated unique hash to one of the object's fields. Subsequent calls to the `hash` function will read the field.
- Implements custom (structural) hashing by overriding `GetHashCode` for datatypes that require it. The `hash` function always checks for the presence of `GetHashCode` before falling back to the generic implementation.
- Implements a generic equality algorithm that recursively traverses and compares all fields of the two objects being compared.
- Allows to override `Equals` and provide a custom equality logic. The equality primitive always checks for the presence `Equals` before falling back to the generic implementation.

Comparisons

Structural comparisons are modelled in a manner similar to equality and hashing. A generic implementation works for all objects by recursively comparing their fields, respecting `IComparable` implementations when those are provided by the user.

Limitations

This section documents the limitations of F# language support in WebSharper™ and possible workarounds to these limitations.

Inner Generic Functions

Due to F# quotations limitations, the following code does not compile under F#:

```
[<JavaScript>]
let F() =
  let id x = x
  id 5
```

The workaround is to specialize the generic function to a concrete type, or lift it to the module level:

```
[<JavaScript>]
let id x = x
[<JavaScript>]
let F() = id 5
```

Anonymous Interface Implementations

Another limitation of F# quotations prohibits the following code:

```
[<JavaScript>]
let F() =
  {
    new System.IDisposable with
      member this.Dispose() = ()
  }
```

The workaround is to provide an explicit name to the class:

```
type MyDisposable = | D with
  interface System.IDisposable with
    [<JavaScript>]
    member this.Dispose() = ()
[<JavaScript>]
let F() = D :> System.IDisposable
```

Record Expressions in Constructors

F# reflected definitions provide insufficient information about the record expressions in object constructors, preventing WebSharper™ from correctly compiling them.

A simple example:

```
type T =
  [<JavaScript>]
  new () = {}
```

Workaround: avoid record expressions, use simple constructors with overloads if necessary.

```
type T [<JavaScript>] () =
  class end
```

Operator Overloading

This feature is not currently fully supported. For example, `a + b` expression translation ignores static `op_Addition` methods on the type of `a`.

Units of Measure

This feature is not currenty supported.

Recursive Values

The following does not translate:

```
type R = { R : R }
let rec r : R = { R = r }
```

Generics Limitations

Certain uses of generic arguments are invalid because of type erasure, for instance:

```
let F<'T>() =
  try () with :? 'T -> ()
```

For the same reason, JSON serialization fails with generic functions, for example:

```
[<Rpc>]
let F<'T> x = x
[<JavaScript>]
let G<'T> x = F x
```

This will fail to compile because the concrete type `'T` is not statically known.

2.3 F# and .NET Library Coverage

WebSharper™ includes a reasonably comprehensive F# and .NET standard library coverage, allowing to use the familiar APIs in JavaScript, including such modules and classes as `List`, `Array`, `Map`, `Set`, `Async`, `Event`, `DateTime`, `TimeSpan`, `Dictionary`, `Stack`, `Queue`. The support for these classes is sometimes incomplete, with a focus on functionality that is reasonable to implement and useful to have available on the client. WebSharper™ will warn you if you attempt to use a feature that is not supported. You can consult the comprehensive coverage report for details.

2.4 JavaScript Library Coverage

WebSharper™ makes it easy to access JavaScript APIs in a typed way from F# by shipping bindings to JavaScript libraries. While a lot of these are available as WebSharper™, the standard distribution ships:

- The `IntelliFactory.WebSharper.JavaScript` module with common utilities, such as getting or setting fields on JavaScript objects, doing `alert` or `setTimeout` calls, and the like.
- JavaScript standard library bindings based on the ECMA 262 3rd edition. Consult the `IntelliFactory.WebSharper.EcmaSc` namespace for details.
- DOM level 3 bindings. Consult the `IntelliFactory.WebSharper.Dom` namespace for details.
- jQuery bindings, based on the 1.6.1 version of jQuery. Consult the `IntelliFactory.WebSharper.JQuery` namespace for details.

3 Developing Pagelets

In most JavaScript user interface frameworks widget construction goes through the same three-stage workflow:

- Construct DOM nodes.
- Embed DOM nodes on the page, allowing the browser to calculate their position and size.
- Enhance the nodes with custom appearance and behavior.

WebSharper™ defines the `IPagelet` interface to capture this protocol. In essense, a pagelet is DOM element combined with a rendering function:

```
type IPagelet =
  abstract member Body : Dom.Node
  abstract member Render : unit -> unit
```

To use a pagelet, its `Body` element is inserted into the document, and then its `Render` method is called. This is an important part of the `IPagelet` protocol not captured by the type system: the `Render` method is always called exactly once, after the `Body` has been inserted into the document.

`Render` should rarely, if ever, be called explicitly, as WebSharper™ does it automatically for pagelets embedded in web pages.

HTML Combinators

The most commonly used pagelets in WebSharper™ are constructed using the HTML combinators from the `IntelliFactory.WebS`

```
Div [Width "200px"] -< [
  H1 [Text "HELLO, WORLD!"]
  P [Text "123.."]
]
```

The above code roughly corresponds to:

```
<div width="200px">
  <h1>HELLO, WORLD!</div>
  <p>123...</p>
</div>
```

The combinators both consume and produce `IPagelet` values, allowing to compose simple pagelets into more complex pagelets.

Some attributes such as `Attr.Type` and `Attr.Value` have to be qualified so as not to clash with common identifiers.

HTML Events

The combinators support a subset of DOM/HTML events. Event handlers are attached destructively with functions `OnClick`, `OnMouseDown` and so on that take a pagelet and mutate its body, returning `unit`. To assist with compositionality, WebSharper™ exposes a destructive piping combinator `|>!` defined as:

```
let ( |>! ) x f = f x; x
```

With this combinator, event handlers can be attached locally:

```
Div [
  Input [Attr.Type "button"; Attr.Value "Click me!"]
  |>! OnClick (fun element eventArguments ->
    element.Value <- "Clicked")
]
```

For convenience, the event combinators pass the typed reference to the event target (the `element` argument in the example above).

Pagelet Rendering Events

In addition to DOM events, WebSharper™ exposes combinators to hook into the rendering event, `OnBeforeRender` and `OnAfterRender`.

Example:

```
Div [
  Div [Text "Foo"]
  |>! OnAfterRender (fun div -> ...)
]
```

The initialization logic in the handler is only called once before or after the pagelet is attached to the page.

Custom Pagelets

Custom pagelets can be created by implementing IPagelet directly, or by using the HTML combinators.

Developing ASP.NET Controls and Pages

WebSharper™ integrates with ASP.NET by exposing pagelets as ASP.NET controls, for example:

```
let MyPagelet : IPagelet =
  Div [Text "Hello World!"] :> _
type MyControl() =
  inherit Web.Control()
  [<JavaScript>]
  override this.Body = MyPagelet
```

With these definitions, in ASP.NET code you now can use:

```
<MyControl runat="server" />
```

The mechanics are as follows:

- An instance of the control is constructed on the server.
- The instance is serialized to JSON that is emitted with the page.
- The instance is deserialized on the client, the pagelet is reconstructed and rendered into the placeholder corresponding to the location of the control.

In this way the controls can have fields that pass information from the server-side to the client-side context.

Every page that uses WebSharper™ controls is required to include the `IntelliFactory.WebSharper.Web.ScriptManager` control in the `<head>` section:

```
<head runat="server">
<ws:ScriptManager runat="server" />
```

During rendering, `ScriptManager` emits a correctly ordered transitive closure of all JavaScript and CSS resources required by the controls on the page.

4 Communicating With the Server

WebSharper™ supports remote procedure calls from the client (JavaScript environment) to the server (ASP.NET or other hosting environment). Remoting is designed to be safe and efficient while requiring as little boilerplate code as possible.

Here is a simple example of a client-side and a server-side function pair that communicate with RPC:

```
module Server =
  [<Rpc>]
  let GetBlogsByAuthor author =
    use db = new DbContext()
    db.GetBlogsByAuthor author
module Client =
  [<JavaScript>]
  let GetBlogsByAuthor (author: Author) =
    Server.GetBlogsByAuthor author
```

In the above example, the `Server.GetBlogsByAuthor` call is blocking. To make the communication asynchronous, WebSharper™ uses the F# asynchronous workflows:

```
module Server =
  [<Rpc>]
  let GetBlogsByAuthor author =
    use db = new DbContext()
    let blogs = db.GetBlogsByAuthor author
    async { return blogs }
module Client =
  [<JavaScript>]
  let GetBlogsByAuthor (author: Author)
                       (callback: Blog [] -> unit) =
    async {
      let! blogs = Server.GetBlogsByAuthor author
      return callback blogs
    }
    |> Async.Start
```

The conceptual model assumed by WebSharper™ is this: the client always has the control, and calls the server when necessary.

The remoting component also assumes that:

- RPC-callable methods are marked with `RpcAttribute`.
- RPC-callable methods are safe to call from the web by an unauthenticated client.
- RPC-callable methods have argument types that are serializable to JSON.
- RPC-callable methods have a return type that is serializable to JSON, or are of type `Async<'T>` where `'T` is such a type.

JSON serializers are automatically derived for the following types, where `'T1, 'T2, ...` are arbitrary JSON-serializable types:

- `unit`
- `bool`
- `int`
- `int64`
- `double`
- `string`
- `System.DateTime`
- `'T []`
- `'T1 * 'T2 * ... * 'Tn`
- F# unions (including `option<'T>` and list<'T>)
- F# records
- Classes with a default constructor

For records, unions and classes to be JSON-serializable, all their fields must also be JSON-serializable.

The remoting mechanism supports three different ways of doing a remote call: message-passing, synchronous and asynchronous.

Synchronous Calls

Synchronous RPC calls block the browser until the server's reply is available. For the user they look just like ordinary client-side function calls.

Example:

```
[<Rpc>]
let Increase(x: int) = x + 1
```

With these definitions, a client call to `Increase 0` proceeds as follows:

- The client serializes `0` to JSON.
- The client sends a RESTful request to the server. The request contains information on which method to call, and its JSON-serialized arguments (`0`)
- The client blocks the browser.
- The server (in ASP.NET context, the WebSharper™ handler) parses the request and looks up the requested method.
- The server makes sure the method is marked with `RpcAttribute`
- The server binds to the method, deserializes the arguments from JSON, and calls it.
- The server serializes the method's response to JSON and responds to the request.
- The client deserializes the response (`1`) from JSON and returns it.
- The client unblocks the browser.

Message-Passing Calls

Message-passing calls are similar to RPC calls but they do not lock the browser, returning immediately on the client. If an RPC function has the return type of `unit`, calls to this function are message-passing calls.

Example:

```
[<Rpc>]
let Log(msg: string) =
  System.Diagnostics.Debug.Write("MSG: {0}", msg)
```

With these definitions, a call to `Log "foo"` proceeds as follows:

- The client serializes `"foo"` to JSON.
- The client sends a RESTful request to the server.
- The client returns `unit` immediately.
- The server parses the request.

- The server binds to and calls the requested method with the arguments deserialized from JSON.

Asynchronous Calls

These calls allow for asynchronous, callback-based processing of the server response. They utilize the `Async<'T>` abstraction from F# to express multi-step asynchronous workflows. The implementation uses nested JavaScript callbacks.

For example:

```
[<Rpc>]
let Increment(x: int) =
  async {
    return! x + 1
  }
[<JavaScript>]
let Foo (callback: int -> unit) =
  async {
    let! x = Increment 0
    let! y = Increment x
    let! z = Increment y
    return callback z
  }
  |> Async.Start
```

With these definitions, a call to `Foo f` proceeds as follows:

- The client sends `0` to the server and registers a callback, proceeding immediately.
- The server replies with `1` and the browser invokes the callback from step 1, binding `x` to `1`.
- The client sends `1` to the server and registers another callback. These asynchronous steps repeat according to the workflow, until the line `return callback z` is reached, with `z` being bound to `3`.
- `f 3` is called.

The mechanics of individual calls are similar to the message-passing calls.

WebSharper™ currently does not support parallel execution of Async values, as JavaScript is a single-threaded environment, and WebSharper™ does not emulate threads on top of it. This may change in the future releases. Nevertheless, the `Async<'T>` abstraction is a useful way to express callback workflows.

Handler Objects

WebSharper™ 2.0 introduces the ability to use instance methods for client-server communication. The syntax for this is:

```
Remote<MyClass>.MyMethod(...)
```

The method invoked should be annotated with the `RpcAttribute` and follows the same convention as static methods:

```
type MyType(..) =
  [<Rpc>]
  member this.MyMethod(..) = ..
```

When the server receives such a request, it obtains an instance of `MyClass` via an `IRpcHandlerFactory`, and invokes the instance method on the obtained object:

```
type IRpcHandlerFactory =
  abstract member Create : Type -> option<obj>
```

The default `IRpcHandlerFactory` always returns `None`, refusing to create instances. This can be customized by using the following method during the web application startup (such as in `Global.asax`):

```
SetRpcHandlerFactory : IRpcHandlerFactory -> unit
```

The support for handler objects makes it more natural to use WebSharper™ remote procedure calls in object-oriented server-side frameworks, such as ASP.NET MVC. This approach also lends itself to the use of Inversion of Control containers to implement `IRpcHandlerFactory`.

Communication Protocol

The communication protocol used by WebSharper™ is a custom protocol built on top of HTTP and JSON. The client sends HTTP POST requests marked with a special HTTP headers to the current URL of the page (`?`), with the bodies of the requests containing the JSON-serialized method arguments. The server responds with a JSON reply.

The URL to which the requets are sent can be customized by subclassing from the `RpcAttribute`.

Hosting in IIS

WebSharper™ applications hosted in IIS should use the `RpcModule` to intercept and handle the RPC requests:

```
<configuration>
 <system.web>
  <httpModules>
   <add name="WebSharperModule"
        type="IntelliFactory.WebSharper.Web.RpcModule,
              IntelliFactory.WebSharper.Web"/>
```

For IIS 7, the configuration is:

```
<configuration>
 <system.webServer>
  <modules>
   <add name="WebSharperModule"
        type="IntelliFactory.WebSharper.Web.RpcModule,
              IntelliFactory.WebSharper.Web"/>
```

Hosting in Other Containers

While WebSharper™ does not yet ship any ready-made components for hosting RPC in non-IIS containers, it provides all the required tools to build one. An example for an alternative container would be a `FastCGI` .NET program that can be deployed to Apache. Please refer to the `IntelliFactory.WebSharper.Remoting` assembly and its API documentation.

5 Managing Resource Dependencies

WebSharper™ automates the management of resource dependencies. For the purposes of WebSharper™, a *resource* is any HTML code that can be rendered to the `<head>` section of a page, for example a CSS or a JavaScript reference. Pages written with WebSharper™ infer their minimal necessary resource set and the correct resource ordering.

Declaring Resources

The code necessary to declare resources can be found in the `Resources` module under the `IntelliFactory.WebSharper.Core` namespace:

```
module R = IntelliFactory.WebSharper.Core.Resources
```

To declare a resource, you create a new type with a default constructor and implement the `IResource` interface:

```
type MyResource() =
    interface R.IResource with
        member this.Render ctx writer = ..
```

You can emit arbitrary HTML in the `Render` method. It will appear in the `<head/>` section of pages that depend on this resource.

A convenient way to do declare resources is to derive from the `BaseResource` class:

```
type MyResource() =
  inherit Resources.BaseResource("http://my.cdn.net",
    "file1.js", "file2.js", "file3.css")
```

The above code declares a resource that renders to this HTML:

```
<script type="text/javascript"
        src="http://my.cdn.net/file1.js"></script>
<script type="text/javascript"
        src="http://my.cdn.net/file2.js"></script>
<link rel="stylesheet" type="text/css"
      href="http://my.cdn.net/file1.css" />
```

The ID of the resource is set to `MyNamespace.MyResource`, the `FullName` of the class. The base URL (`http://my.cdn.net`) can be overriden by providing an application setting with the key `MyNamespace.MyResource`.

`BaseResource` can also declare resources embedded into the current assembly:

```
[<assembly: System.Web.UI.WebResource("My.js", "text/javascript")>]
do ()
type MyEmbeddedResource() =
  inherit Resources.BaseResource("My.js")
```

Declaring Resource Dependencies

A resource dependency can be declared on a type or a member by annotating it with `RequireAttribute`. It is parameterized by the type of the resource to require:

```
[<Require(typeof<MyResource>)>]
type MyWidget() = ...
[<Require(typeof<MyResource>)>]
let F (..) = ..
```

Types, modules and static methods can be annotated with dependencies. All code that calls the annotated methods is assumed to depend on the resource.

Dependency Graph

When constructing a page, WebSharper™ infers the set of resources to include in the `<head/>` section. It starts by looking at the set of controls present on the page. Every control has a type, and this type corresponds to a node in the dependency graph. WebSharper™ computes the set of all nodes in the graph reachable from the control nodes. It then renders all resources found in this set.

The dependency graph is a directed graph with .NET classes and static methods as nodes. An edge from A to B singifies that A depends on B. The graph is partially inferred and partially specified by the user:

- Graph edges are inferred from the call graph. For an example, if a function `f` calls a function `g`, then there is an edge from `f` to `g`. There are also structural rules, such as classes depending on their base classes, or methods and modules depending on the modules that declared them.
- Graph edges are also declared by using the `RequireAttribute`.

Resource Implementation

The resource graphs are constructed for every WebSharper™-processed assembly and are serialized to binary. They are stored within the assembly itself. At runtime all the graphs of all the referenced assemblies are deserialized and merged into a single graph.

6 Developing with Formlets

The WebSharper™ Formlets library provides a high-level abstraction for working with web forms and constructing interactive user interfaces.

One of the most common tasks in web development is creating web forms for collecting user data. Among other things this involves:

- Creating forms to collect data.
- Looking up the submitted data on the server.
- Validating data.
- Providing validation feedback to the user.

WebSharper™ Formlets address all of the above tasks. The advantages of using formlets for web-form construction are:

Type Safety

Every `Formlet<'T>` collects data of type `'T`, preventing typing errors. The data type is not limited to primitive values, and often includes convenient to consume records and unions.

Composabilty

Complex formlets are built from simple components. In basic composition, two formlets `Formlet<'T1>` and a `Formlet<'T2>` can be combined into a `Formlet<'T3>` using a function `'T1 -> 'T2 -> 'T3`. Additional formlet composition methods allow to express wizard-like workflows.

Reusability

Once defined, a formlet may be reused as a component of many other formlets and pages.

Validation

Formlets are aware of the need to validate the collected data and provide interactive feedback to the user.

Declarative Style

Formlets are defined in a declarative way, meaning that you only need to specify the descriptive parts of your form applications, i.e. which controls to include, what validation logic to apply and how to aggregate the result of the sub-forms. All the tedious work of creating HTML elements, extracting the form data and displaying error messages is automated.

6.1 Overview

The WebSharper™ Formlets library contains a set of components for creating primitive form controls, composing formlets, adding validation, and enhancing formlets in various ways.

6.1.1 Primitive Formlet Controls

In the `Control` module you find a set of predefined formlets corresponding to some basic web form components such as input text fields, text areas and check boxes.

As an example, the `TextArea` formlet is exposed as a function from the default value to a `Formlet<string>`.

```
let textForm = Controls.TextArea ""
let selForm =
    [ "Option A", "A"; "Option B", "B" ]
    |> Controls.Select 0
let buttoForm = Controls.Button "Click"
```

6.1.2 Formlet Composition

The most important property of formlets is the ability to build complex formlets by composing simpler ones.

Static Composition

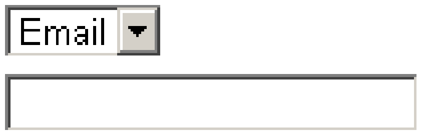
The `Formlet.Yield` and `Formlet.Apply` (or `<*>`) functions provide the default way to combine formlets. `Formlet.Yield` accepts a function specifying how to combine the values of several formlets, and `Formlet.Apply` incrementally provides the formlets to compose.

A composed formlet `Formlet.Yield f <*> g1 <*> g2 .. <*> gn` at any point in time carries the value of `f g1.value g2.value... gn.value`. Visually it renders as a concatenation of the component formlets `g1 .. gn`.

In the following example two string formlets are composed:

```
[<JavaScript>]
let ComposedFormlet: Formlet<string> =
    Formlet.Yield (fun x y -> x + " " + y)
    <*> Controls.Select ["Email", "E"; "Mail", "M"] 0
    <*> Controls.Input ""
```

The value of the `ComposedFormlet` is the concatenation of the component formlet values. The result looks like this:



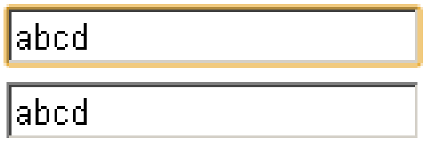
Formlets are closed under composition. This means that `ComposedFormlet` may itself be composed with other formlets.

Dynamic Composition

Dynamic/Dependent formlets are formlets that depend on the values produced by some other formlet. You can specify such dependencies using F# computation expressions:

```
[<JavaScript>]
let DependentFormlet () =
    Formlet.Do {
        let! x = Controls.Input ""
        return! Controls.Input x
    }
```

The formlet above uses the values produced by the first text-box as input when constructing an additional (dependent) text-box.



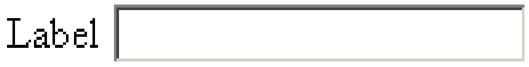
6.1.3 Enhancing Formlets

The `Enhance` module provides a set of predefined functions for enhancing formlets with additional properties.

Common for these functions is that they accept a formlet of type `T` along with additional arguments as an input and return a formlet of the same type extended with some properties.

As an example, the `Enhance.WitTextLabel` accepts a string value corresponding to the label, and a formlet, returning a new formlet with a label.

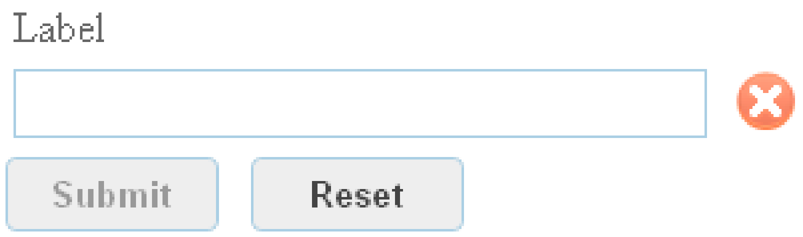
```
[<JavaScript>]
let LabeledFormlet =
    let label = Text "Label"
    Controls.Enhance.WithLabel label (Control.Input "")
```



The function `Control.Enhance.WithLegend` has a similar signature but instead of returning a formlet with a label, the body of the formlet is wrapped within an HTML `legend` element, visually creating a small box around the formlet.

You may also apply several transformation functions for enhancing formlets in different ways. Below is an example combining the `Enhance.WithTextLabel` and `WithSubmitAndResetButtons` functions for creating a text field formlet with a legend enhanced with submit and reset buttons:

```
let InputForm =
    Controls.Input ""
    |> Enhance.WithTextLabel "Label"
    |> Enhance.WithSubmitAndResetButtons
    |> Enhance.WithFormContainer
```

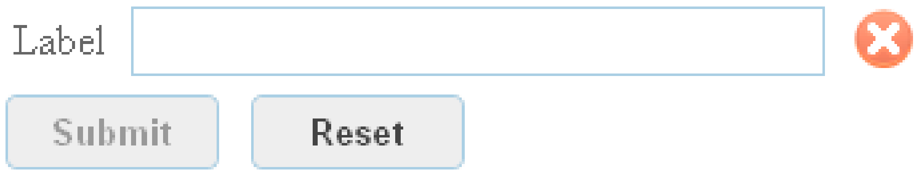


Note that the order in which the functions are applied is important.

6.1.4 Adding Validation

By adding validation you restrict the admissible formlet values by putting the formlet into a failing state whenever the current value is invalid. Among other things, this means that the formlet cannot be submitted since no value is available. Some predefined validators are found in the `Validator` module. Here is an example enhancing a text-box formlet with validation requiring that the input string is not empty:

```
let ValidationForm =
     Controls.Input ""
     |> Enhance.WithTextLabel "Label"
     |> Validator.IsNotEmpty "Enter non-empty value"
     |> Enhance.WithValidationIcon
     |> Enhance.WithSubmitAndResetButtons
     |> Enhance.WithFormContainer
```



6.1.5 Layout

Each formlet carries a layout-manager responsible for rendering the visual components produced when running the formlet. You can use the function `Formlet.WithLayout` to specify the layout manager to be used. Three default layout managers are provided:

- Layout.Vertical - Lays out components vertically.
- Layout.Horizontal - Lays out components horizontally
- Layout.Flowlet - Creates a wizard-like interface where subsequent components replace the previous ones.

Instead of using the `Formlet.WithLayout` the above layouts may be applied directly via the functions `Formlet.Vertical`, `Formlet.Horizontal` and `Formlet.Flowlet`.

6.1.6 Displaying Formlets

Since formlets implement the `IPagelet` interface, they may be directly embedded inside HTML combinators:

```
[<JavaScript>]
let Main () =
    Div [
        Controls.Input "Text-field formlet in div tag"
    ]
```

If you want to handle the values produced by a formlet, you may use the `Run` method. It accepts a handler that is invoked every time a new value is produced by the formlet, and returns an `IPagelet` instance corresponding to the form body:

```
[<JavaScript>]
let Main () =
    let input =
        Controls.Input ""
        |> Enhance.WithSubmitButton
    Div [
        input.Run (fun s ->
            processResult s
        )
    ]
```

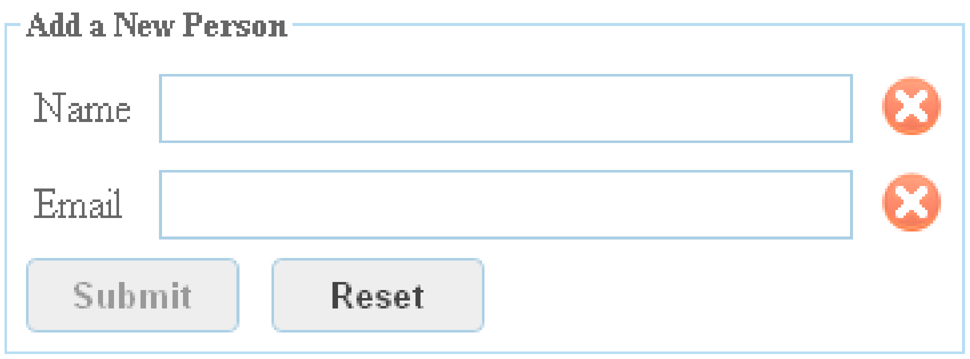
6.2 Examples

Statically Composed Formlet

Below is an example of a formlet for entering name and email information:

```
type Person = {
    Name: string
    Email: string
}
[<JavaScript>]
let PersonFormlet : Formlet<Person> =
    let nameF =
        Controls.Input ""
        |> Validator.IsNotEmpty "Empty name not allowed"
        |> Enhance.WithValidationIcon
        |> Enhance.WithTextLabel "Name"
    let emailF =
        Controls.Input ""
        |> Validator.IsEmail "Please enter valid email address"
        |> Enhance.WithValidationIcon
        |> Enhance.WithTextLabel "Email"
    Formlet.Yield (fun name email -> {Name = name; Email = email})
    <*> nameF
    <*> emailF
    |> Enhance.WithSubmitAndResetButtons
    |> Enhance.WithLegend "Add a New Person"
    |> Enhance.WithFormContainer
[<JavaScript>]
let Main () =
    Div [PersonFormlet]
```

`PersonFormlet` defines a formlet parameterized with the user-defined type `Person`. It contains validation logic to guarantee that the `Name` field is not empty and the email address is of the correct format.

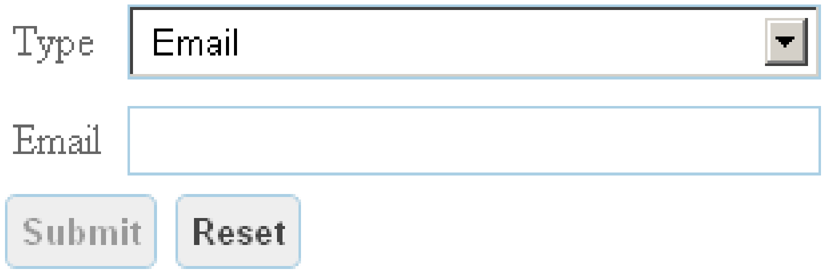


Dependent Formlet

Below is an exmaple of using computation expressions to define a dependent formlet:

```
    type ContactData =
        | Phone of string
        | Email of string
    [<JavaScript>]
    let ContactFormlet : Formlet<ContactData> =
        let phone =
            Controls.Input ""
            |> Enhance.WithTextLabel "Phone"
            |> Formlet.Map Phone
        let email =
            Controls.Input ""
            |> Validator.IsEmail "Please enter a valid email address."
            |> Formlet.Map Email
            |> Enhance.WithTextLabel "Email"
        Formlet.Do {
            let! contactTypeFormlet =
                Controls.Select 0 [("Phone", phone); ("Email", email)]
                |> Enhance.WithTextLabel "Type"
            return! contactTypeFormlet
        }
        |> Enhance.WithSubmitAndResetButtons
        |> Enhance.WithFormContainer
```

This formlet will interactively present either the *phone* or *email* form depending on the user's choice in the select box. Each time the user changes the selected option, the dependent form is updated.



7 Developing With Sitelets

WebSharper™ Sitelets provide a way to alleviate the need of serving content via ASPX pages. Instead, you can define complete websites programmatically using F#.

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 also allow you to embed WebSharper™ client-side controls.

You will benefit from using sitelets by:

- Being able to dynamically construct pages and serve arbitrary content.
- Having full control of your URLs by specifying custom routers for linking them to content.
- Composing contents into sitelets, which may themselves be composed into larger 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.

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()
                    [H1 [Text <| "Current time: " + time]]}
    type MySampleWebsite() =
        interface IWebsite<Action> with
            member this.Sitelet =
                Sitelet.Content "/index" Action.Index Index
            member this.Actions = []
[<assembly: WebsiteAttribute(typeof<SampleSite.MySampleWebsite>)>]
do ()
```

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 only page.

The content of the index page is defined as a `PageContent`, where the body consists of a server side HTML element. Here the current time is computed and displayed within an H1 tag.

The `MySampleWebsite` type specifies the sitelet to be served by implementing the `IWebsite` interface. In this case, the sitlet is defined using the `Sitelet.Content` operator for constructing a sitelet of the Index content. In the resulting sitelet, the `Action.Index` value is associated with the path `/index` and the given content.

7.1 The Building Blocks of Sitelets

A sitelet consists 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.

Actions

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.

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("/page2", 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.

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  -> MyContent.Page1
            | Action.Page2  -> MyContent.Page2
    }
```

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 custom 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 runtime 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 =
        [
          H1 [Text "Page 1"]
          A [HRef (context.Link Action.Page2)] -< [Text "Page 2"]
          A [HRef (context.ResolveUrl "~/Page3.html")] -< [Text "Page 3"]
        ]
    }
```

Note how `context.Link` is used in order to resolve the URL to the Page2 action. Action URLs are always constructed relative to the application root, whether the application is deployed as a standalone website or in a virtual folder. `context.ResolveUrl` helps to manually construct application-relative URLs to resources that do not map to actions.

Contents are not restricted to produce HTML responses. 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}"
        }
```

7.2 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, using this operator and using the `Sitelet.Sum` combinator:

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


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*.

7.3 Embedding Client-Side Controls

The integration of WebSharper controls(i.e. code that translates to JavaScript and runs on the client) in sitelet content is straightforward. 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.

7.4 User Sessions and Protecting Content

The `UserSession` module provides three primitive functions for handling user sessions:

- `LoginUser : string -> unit` : Logs in a user with the given identifier.
- `GetLoggedInUser` - Returns the currently logged in user identifier as an option value. If no user is logged in, the value `None` is returned.
- `Logout : unit -> unit` : Removes the logged in identifier.

The implementation of these functions relies on cookies and thus requires that the browser has enabled cookies.

In addition to the functions above a combinator `Protect` is provided in order create protected content, i.e. content only available for authenticated users:

```
Protect : Filter<'Action>) -> Sitelet<'Action>) -> Sitelet<'Action>
```

where the type `Filter` is defined as:

```
type Filter<'Action> =
    {
        VerifyUser : string -> bool;
        LoginRedirect : 'Action -> 'Action
    }
```

Given a filter value and a sitelet, `Protect` returns a new sitelet that requires a logged in user that passes the `VerifyUser` predicate, specified by the filter. If the user is not logged in, or the predicate returns false, the request is redirected to the action specified by the `LoginRedirect` function specified by the filter.

Below is an example of creating a protected sitelet containing two pages:

```
let protected =
    let filter : Sitelet.Filter<Action> =
        {
            VerifyUser = fun _ -> true
            LoginRedirect = fun _ -> Action.Login
        }
    Sitelet.Protect filter (
        [
            Sitelet.Content "/p1" Action.P1 P1
            Sitelet.Content "/p2" Action.P2 P2
        ]
        |> Sitelet.Folder "protected"
    )
```

7.5 Handling HTTP Parameters

The `Http.Request` represents the incomming request containing the GET and POST parameters. By defining your sitelet manually you can decide how to route requests depending on these parameters.

The `Router` module contains some funcions for simplifying the construction of routers that map requests containing *POST* parameters.

Here is an example of creating a sitelet with a router for handling POST requests. The function `Router.FromPostParameter` and `Router.At` are utilized for constructing a router mapping POST requests to the url */post*:

```
let MyRouter =
    [
        Router.Table [
            Action.Index , "/index"
        ]
        Router.FromPostParameter ParamName
        |> Router.At "/post"
        |> Router.TryMap
            (Action.Post >> Some)
            (function | Post x -> Some x | _ -> None)
    ]
    |> Router.Sum
let Handle action =
    match action with
    | Action.Index -> Pages.Index
    | Action.Post value -> Pages.Post value
let MySitelet =
    {
        Router = MyRouter
        Controller = { Handle = Handle}
    }
```

The function `Router.TryMap` is used for mapping the string router to one of type `Action`:

Alternatively, the parameteres can be obtained at the content generation phase, via the `Context` object:

```
let Index =
    Content.PageContent <| fun context ->
        let param =
            match context.Request.Post.["post"] with
            | Some n    -> n
            | None      -> "No Param"
        {Page.Default with Body = [Text param}
```

7.6 Using HTML Templates

The Visual Studio solution templates for sitelets that are installed with WebSharper 2.X contain a tool set for HTML templating.

When creating a new sitelet-based project, you will see that a file `Skin.template.xml` is included. This is a template file, specifying a simple HTML structure. Such templates may contain holes allowing dynamic elements to be inserted. Holes are defined using a special syntax: `${NameOfHole}`.

Below is an example of a simple template:

```
<html xmlns="http://www.w3.org/1999/xHtml">
    <head>
        <title>Your site title</title>
        <link href="/css/reset.css" rel="stylesheet" type="text/css" />
        <link href="/css/site.css" rel="stylesheet" type="text/css" />
    </head>
    <body>
        ${Content}
    </body>
</html>
```

Building the solution will convert the templates into F# functions, accepting a parameter for specifying the content of each hole. Assuming that the HTML template above is defined in a file `Skin.template.xml`, the corresponding template function may be invoked with a parameter specifying its dynamic content:

```
let Index =
    Templates.Skin.Skin (Some "Main Page") {
        Content = fun context ->
            [
                Div [Text "Hello, world!"]
            ]
    }
```

8 Developing Bindings to JavaScript Libraries

WebSharper™ is designed to enable the programmer to easily interoperate with existing JavaScript code. This section documents the techniques and guidelines for creating F# bindings to existing JavaScript libraries.

The development of WebSharper™ bindings to JavaScript involves:

- Writing class and function stubs.
- Providing static types for the stubs, possibly writing wrapper types or using inlining where appropriate.
- Defining CSS, JavaScript and other resources and packaging the bindings library.
- Providing configuration sections for the library users.

Stubs

A stub is an F# class or a function that has no F# implementation, but instead represents existing JavaScript code. The stubs provide F# identity, type safety and code completion to JavaScript-implemented functionality.

For example, assume a JavaScript file with the following declarations:

```
function counter() {
  var value = 0;
  return function () { return value++; };
}
function Counter() {
  this.value = 0;
  this.next = function () {
    this.value ++;
    return this.value;
  };
}
```

The corresponding F# module with stubs might look like this:

```
module Counter =
  [<Name "counter">]
  [<Stub>]
  let makeCounter () : (unit -> int) = X<_>
  [<Stub>]
  type Counter() =
    [<DefaultValue>]
    val mutable value: int
    [<Name "next">]
    [<Stub>]
    member this.Next() : int = 0
```

Given the stub definitions, the following F# code:

```
open Counter
let c = new Counter()
c.Next()
```

Will translate to the equivalent of:

```
var c = new Counter()
c.next()
```

Things to note:

- Marking a class declaration with `StubAttribute` enables all its members for use from F#.
- `StubAttribute`-annotated members do not have to have meaningful bodies if they are not intended to be called on the server side. `Unchecked.defaultof<_>` or its shorthand `X<_>` is acceptable as a body of any such member.
- `NameAttribute` can be provided when the inferred names (F# names) do not match the desired compiled names.

Static Typing

Stubs must provide types for all represented methods. The bindings author is expected to pick or construct types that reflect the expectations of the JavaScript code. To do this effectively requires an awareness of how WebSharper™ represents F# data in JavaScript.

For cases when static typing is difficult or impossible to determine, it is always legal to give all argument and return parameters the `obj` type. This approach gives the least information to the user, but is viable if the user is to properly cast the parameters. Note that `box`, `unbox`, **`upcast`** and **`downcast`** are all implemented as the identity function in WebSharper™, and are safe when the source and target types of the cast have the same data representation, as all type information is erased during compilation.

Common scenarios for providing static typing to stubs include:

- *JavaScript code expects a scalar* (such as a string, a number, or a boolean). Use the matching F# type (`string`, `int`, `double`, `bool`).
- *JavaScript code expects an array.* If the array has a fixed known length, use an F# tuple. If the array varies in length and is monomorphic, use an F# array. If the array varies in length and is polymorphic, use `obj []`.
- *JavaScript code expects a first-class function.* If at all possible, determine the parameter and return types and use `p1 * p2 * ... * pn -> r`.
- *JavaScript code expects a string from a fixed set of possible values.* Use `ConstantAttribute` to create a new union type to represent this fixed set of values.

- *JavaScript code expects an object with a certain field structure.* Express the structure as a new record type and use it as the type of the parameter.
- *JavaScript code expects an object with optional fields.* This is a common idiom for configuration objects in graphical user interface frameworks. The recommended way to expose the assumptions about these objects to F# is to create a class with `DefaultValue` fields, which will be left undefined if not explicitly set:

  ```
  type ButtonConfiguration [<Inline "{}">] () =
    [<DefaultValue>]
    val mutable label: string
    [<DefaultValue>]
    val mutable onclick: unit -> unit
  ...
  let cfg = new ButtonConfiguration(label = "Click me!")
  ```

- *JavaScript code expects either a value of type `A`, or a value of type `B`.* Use member overloads where appropriate.

Packaging and Configuration

The F# code written against stub classes can only work if the implementing JavaScript is available in the runtime environment. The recommended way to do it is to annotate all stub modules and classes with the `RequireAttribute` and define resource classes to include the necessary files. If that is properly done, WebSharper™ automatically includes the relevant JavaScript and CSS links on the page when any of the stubbed functionality is used, freeing the library user from manually keeping track of these assumptions.

9 WebSharper™ Interface Generator

The WebSharper™ Interface Generator is a tool for generating WebSharper™ bindings to JavaScript libraries. Bindings allow developers to access these libraries from typed F# code that gets compiled to JavaScript by WebSharper™. While it is possible to create bindings manually, WebSharper™ Interface Generator allows to write the binding definitions in F#, making full use of the language to streamline repetitive tasks.

WebSharper™ Interface Generator includes an assembly with binding-generating code and Visual Studio project templates. Simply put, WebSharper™ Interface Generator takes an F# value representing a set of classes, interfaces, and members together with their documentation and mappings to JavaScript, and generates a binding assembly from that definition. The binding is a .NET assembly containing the generated types and method stubs annotated with raw JavaScript code using the `InlineAttribute` custom attribute.

9.1 Getting Started

WebSharper™ Interface Generator installs together with WebSharper™ Professional. To create a new project open Visual Studio and select *File > New > Project > Visual F# > IntelliFactory > WebSharper 2.1 Binding.*

Below you will find a sample binding definition. For more examples, please check out the open-source code by IntelliFactory.

```
module WebSharperExtension.Definition
open IntelliFactory.WebSharper.InterfaceGenerator
let I1 =
  Interface "I1"
  |+> [
    "test1" => T<string> ^-> T<string>
```

```
      "radius1" =@ T<float>
    ]
  let I2 =
    Generic / fun t1 t2 ->
      Interface "I2"
      |+> [
      Generic - fun m1 -> "foo" => m1 * t1 ^-> t2
    ]
  let C1 =
    let C1T = Type.New ()
    Class "C1"
    |=> C1T
    |+> Protocol [
          "foo" =% T<int>
        ]
    |+> [
      Constructor (T<unit> + T<int>)
      "mem" =>
        (T<unit> + T<int> ^-> T<unit>)
      "test2" =>
        (C1T -* T<int> ^-> T<unit>) * T<string> ^-> T<string>
      "radius2" =@ T<float>
      |> WithSourceName "R2"
      "length" =% T<int>
      |> WithSourceName "L2"
    ]
  let Assembly =
    Assembly [
      Namespace "MyNamespace" [
        I1
        Generic - I2
        C1
      ]
    ]
```

9.2 Constructing Types

Defining classes, interfaces and member signatures requires an abstraction for types. Types are represented as `IntelliFactory.WebSharper.InterfaceGenerator.Type.IType` values. These values describe system, user-defined, existing, and generated types.

Existing Types

Existing system and user-defined types can be defined by using the `T` combinator with a generic parameter:

```
  let types : list<Type.IType> =
    [
      T<int>
      T<list<string>>
      T<MyType>
    ]
```

Type Combinators

Simpler types can be comined to form more complex types, including arrays, tuples, function types, and generic instantiations.

```
[
  T.ArrayOf T<int>
  T<int> * T<float> * T<string>
  Type.Tuple [T<int> * T<float>; T<string>]
  T<int> ^-> T<unit>
  T<System.Collections.Generic.Dictionary<_,_>>.[T<int>,T<string>]
]
```

In addition, delegate types can be formed. WebSharper™ treats delegate types specially: their are compiled to JavaScript functions accepting the first argument through the implicit `this` parameter. For example when this can be helpful, consider following JavaScript function:

```
function iterate(callback, array) {
  for (var i = 0; i < array.length; i++) {
    callback.call(array[i], i);
  }
}
```

To bind this function to WebSharper™ one needs to provide a type for the `callback` parameter, which is a function called with an element of the array passed through the `this` implicit parameter and the array index passed through the first parameter. This can be achieved thus:

```
let callbackType = T<obj> -* T<int> ^-> T<unit>
let iterateType  = callbackType * Type.ArrayOf T<obj> ^-> T<unit>
```

The type of the callback is then compiled to a delegate type in F#, `Func<obj,int,unit>`.

New Types

In addition to existing user-defined and system types, there are new types that correspond to the classes and interfaces being defined. Class and interface definitions implement the `Type.IType` interface and can be used where types are expected. When this is inconvenient, as it often is, for example, with mutually recursive classes, new type values can be constructed and used before being associated with a particular class or interface definition:

```
let Widget   = Type.New()
let Callback = Widget ^-> T<unit>
let WidgetClass =
  Class "Widget"
  |=> Widget
```

The last line associates the `Widget` type with the `WidgetClass` definition.

9.3 Defining Members

The primary use of type values is the definition of member signatures, methods, constructors, properties and fields, are defined.

9.3.1 Methods

Method representations are constructed using the `Method` (short form: `=>`) combinator that takes the name of the method and the corresponding functional type. Some examples:

```
let methods =
  [
    Method "dispose" T<unit -> unit>
    Method "increment" (T<int> ^-> T<int>)
    "add" => T<int> * T<int> ^-> T<int>
  ]
```

Void return types and empty parameter lists are indicated by the `unit` type, multiple parameters are indicated by tuple types. It is an error to define a method with a non-functional type.

Due to their prevalence in F#, by default all methods are generated as static and public. See Instance Member Definitions, Access Modifiers, Static and Instance Modifiers.

Parameter Names

By default, method parameters get autogenerated names. You can customize parameter names as follows:

```
let methods =
  [
    Method "dispose" (T<unit>?object ^-> T<unit>)
    Method "increment" (T<int>?value ^-> T<int>)
    "add" => T<int>?x * T<int>?y ^-> T<int>
  ]
```

Variable-Argument Signatures

F# supports variable-argument methods via the `System.ParamArrayAttribute` annotation. WebSharper™ understands this annotation and compiles such methods and delegates to variable-argument accepting functions in JavaScript. Here is the syntax to define a variable-argument signature:

```
let methods =
  [
    "t1" => !+ T<obj> ^-> T<unit>
    "t2" => T<string> *+ T<obj> ^-> T<unit>
  ]
```

When compiled to F#, these methods will have the following signatures:

```
val t1 : ([<System.ParamArray>] args: obj []) -> unit
val t2 : string * ([<System.ParamArray>] args: obj []) -> unit
```

Optional Parameters

Parameters can be made optional:

```
Method "exit" (!? T<string>?reason ^-> T<unit>)
```

Signatures such as the one above generate multiple members by implicit overloading (see below).

Implicit Overloads

Type unions facilitate describing JavaScript methods that accept arguments of either-or types. Type unions are implemented by implicit overloading of generated members. For example:

```
"add" => (T<int> + T<string>) * (T<obj> + T<string>) ^-> T<unit>
```

This method can accept either a `string` or an `int` as the first argument, and either an `obj` value or a `string` as the second. Four overloads are generated for this signature.

9.3.2 Properties

Properties can be generated with a getter, a setter or both. Below are the full and abbreviated syntax forms:

```
let properties =
  [
    Getter "ReadOnly" T<int>
    Setter "WriteOnly" T<int>
    Property "Mutable" T<int>
  ]
let shorthand =
  [
    "ReadOnly"  =? T<int>
    "WriteOnly" =! T<int>
    "Mutable"   =@ T<int>
  ]
```

9.3.3 Constructors

Constructors definitions are similar to methods but do not carry a return type. Examples:

```
let constructors =
  [
    Constructor T<unit>
    Constructor T<int>?width * T<int>?height
  ]
```

9.3.4 Fields

Fields are generated using a similar syntax to properties:

```
let fields =
  [
    Field "width" T<int>
    Field "height" T<int>
  ]
let shorthand =
  [
    "width"  =% T<int>
    "height" =% T<int>
  ]
```

9.3.5 Interfaces

Interfaces are defined using the `Interface` keyword and then extended with members.

```
Interface "IAccessible"
```

Member Definitions

Member definitions are appended using the `|+>` combinator, for example:

```
Interface "IAccessible"
|+> [
  "Access" => T<unit->unit>
  "LastAccessTime" =? T<System.DateTime>
]
```

Inheritance

Interfaces can inherit or extend multiple other interfaces. The syntax is as follows:

```
Interface "IAccessible"
|=> Extends [T<System.IComparable>; T<System.IEnumerable<int>>]
|+> [
  "Access"         => T<unit->unit>
  "LastAccessTime" =? T<System.DateTime>
]
```

9.3.6 Clases

Class definition is very similar to interface definition. It starts with the `Class` keyword:

```
Class "Pear"
```

Static Member Definitions

Static members are added using the `|+>` combinator:

```
let Pear =
  let Pear = Type.New()
  Class "Pear"
  |=> Pear
  |=> [
    "Create" => T<unit> ^-> Pear
  ]
```

Instance Member Definitions

Instance members are usually added using the `Protocol` combinator:

```
Class "Pear"
|+> Protocol [
  "Eat"     => T<unit->unit>
  "IsEaten" =? T<bool>
]
```

The use of the `Protocol` function is equivalent to transforming the methods with the `Instance` function prior to inclusion.

Class Inheritance

The syntax for class inheritance is as follows:

```
Class "ChildClass"
|=> Inherits BaseClass
```

Interface Implementation

The syntax for class inheritance is as follows:

```
Class "MyClass"
|=> Implements [T<System.IComparable>]
```

Nested Classes

Class nesting is allowed:

```
Class "MyClass"
|=> Nested [
      Class "SubClass"
     ]
```

9.3.7 Generics

Generic Types

Generic types and interfaces are defined by prefixing the definition with the code of the form `Generic / fun t1 t2 .. tN ->`. The `t1..tN` parameters can be used as types in the definition and represent the generic parameters. For example:

```
Generic / fun t1 t2 ->
  Interface "IDictionary"
  |+> [
        "Lookup"      => t1 ^-> t2
        "ContainsKey" => t1 -> T<bool>
        "Add"         => t1 * t2 -> T<unit>
        "Remove"      => t1 -> T<unit>
      ]
```

This compiles to the following signature:

```
type IDictionary<'T1,'T2> =
  abstract member Lookup : 'T1 -> 'T2
  abstract member ContainsKey : 'T1 -> bool
  abstract member Add : 'T1 * 'T2 -> unit
  abstract member Remove : 'T1 -> unit
```

Generic Methods

Similarly, generic methods are generated using lambda expressions, but the syntax is now `Generic - fun t1 .. tN ->`, for example:

```
Generic - fun t ->
  "length" => T<list<_>>.[t] ^-> T<int>
```

This code would generate the following F# signature:

```
val Length<'T> : list<'T> -> int
```

9.3.8 Modifiers

Access Modifiers

By default, all members and types are generated with the public access modifier. This can be changed by applying one of the four access modifier setters: `Public`, `Internal`, `Protected` and `Private`.

Static and Instance Modifiers

By default, all members are generated static. Members can be made static or instance by using the `Static` or `Instance` functions. Interface definitions automatically apply the `Instance` function.

Documentation Comments

Documentation comments can be added using the `WithComment` function.

9.4 Customizing JavaScript

By default, inline JavaScript definitions are inferred for all methods and properties from their names. This is intuitive and convenient but not fully general. Is is therefore possible to bypass the inferred inlines and customize the generated bindings.

Default Inline Generation

Default bindings are name-based. For example, a static function called `foo` with three arguments on a class called `Bar`, produces the JavaScript inline `Bar.foo($0,$1,$2)`.

Generated .NET names are automatically capitalized, so that this function is accessible as `Bar.Foo` from F#.

Qualified names can be used on classes that are accessible in JavaScript with a qualified name, for example:

```
Class "geometry.Point"
```

This generates a .NET class `Point` which binds all static members as `geometry.Point.foo()` in JavaScript.

Custom Names

The simplest form of customization allows to decouple the .NET name of a member from the name used by the inline generation process. This is done by the `WithSourceName` function. For example:

```
"ClonePoint" => Point ^-> Point
|> WithSourceName "clone"
```

This generates a method that is available as `ClonePoint` from .NET but calls `clone` in JavaScript.

Custom Inline Methods

The method and constructor inlines can be set explicitly by using `WithInline`.

Custom Inline Properties

Properties have separete inlines for the getter and the setter methods. These can be set explicitly by using `WithGetterInline` and `WithSetterInline` respectively.

9.5 Best Practices

The benefit of using F# for generating the bindings is that repetitive tasks or patterns can be distributed as functions. Several such patterns are pre-defined in the standard library.

Configuration Class Pattern

This pattern is useful for describing JavaScript configuarion objects. Configuration objects typically are simple collections of fields, most of them optional, that are used to describe how another object it to be constructed or operate. Let us take a simple example:

```
let MyConfig : Class =
  Pattern.Config {
    Required =
      [
        "name", T<string>
      ]
    Optional =
      [
        "width", T<int>
        "height", T<int>
      ]
  }
```

This definition would produce a class useable from F# that would compile to simple JavaScript object literals:

```
MyConfig("Alpha", Width=140)
```

Enumeration Pattern

JavaScript functions often accept only a fixed set of constants, such as strings. Typing such parameters with `string` would be misleading to the user. The enumeration pattern allows to generate a type that is limited to a specific set of constants, specified as either inlines or strings literals. See `Pattern.EnumInlines` and `Pattern.EnumStrings`, both of which generate `Class` values.

10 Developing Proxies to .NET Libraries

Proxying in WebSharper™ is the process of providing JavaScript-compilable F# implementations for classes and modules that were compiled without WebSharper™, for example the Base Class Library classes such as `Dictionary`. The proxying graph relates proxied types to proxy types. The proxy graph is constructed by consulting all `Proxying.AbstractProxy` in an assembly and its references.

The simple implementation, `ProxyAttribute`, is applied to the proxied type and should reference the proxy type. Sample usage:

```
[<Proxy "Microsoft.FSharp.Collections.ArrayModule, \
      FSharp.Core, \
      Version=2.0.0.0, \
      Culture=neutral, \
      PublicKeyToken=b03f5f7f11d50a3a">]
module MyArrayModule =
  [<Inline "$0.length">]
  let length<'T> (arr : 'T []) = 0
```

WebSharper™ projects that reference the DLL containing the above code can use `Array.length` in client-side code.

There are several things to remember when developing .NET proxy code:

- The string parameter to the `ProxyAttribute` must exactly match the `FullName` property of the generic definition of the target .NET type.
- The name, type, number of arguments and the calling convention of a proxy member must match exactly those of the member being proxied.

11 Built-in WebSharper™ Extensions

WebSharper™ Extensions allow to access JavaScript libraries from F#, enjoying type safety and code completion, just as if these libraries were written in F#. For a complete list please check out the available extensions page.

Several extensions are built-in and available with your default WebSharper™ installation. These include DOM, jQuery, and the Standard ECMA-262 JavaScript library.

11.1 WebSharper™ Extensions for DOM

DOM types can be found under the `IntelliFactory.WebSharper.Dom` namespace. The extension is based on the DOM specification, and therefore does not provide any browser-specific methods, such as `innerHtml`. Be warned that DOM compliance varies from browser to browser, and therefore *relying on DOM may cause your code to be browser-specific.*

WebSharper Elements and DOM Nodes

WebSharper™ HTML elements create and instantiate DOM nodes lazily as they are attached to the document and rendered. A typical way to access these DOM nodes after they are rendered is using the `OnAfterRender` function:

```
Div []
|>! OnAfterRender (fun element ->
    element.Text <- "Hello World")
```

The Document Object

The DOM `Document` object instance accessed as `document` from JavaScript is available in F# as:

```
let doc = Document.Current
let doctype = doc.Doctype.Name
let uri = doc.DocumentURI
```

Events

You can easily add event listeners to any DOM element as follows:

```
let listener = fun () -> div2.TextContent <- "Clicked!"
div1.AddEventListener("click",listener,false)
```

11.2 WebSharper™ Extensions for EcmaScript

This extension enables you to use standard EcmaScript features in a direct, type-safe way without inlining actual EcmaScript code. You can use this extension to compile Standard ECMA-262 compliant code directly from F#.

This extension fully implements the 5th edition of the Ecma-262.

The ECMA code is available in the `IntelliFactory.WebSharper.EcmaScript` namespace. The implementation closely follows the standard, covering the following ECMA objects:

- `Global`
- `Object`
- `Function`
- `Array`
- `String`
- `Boolean`
- `Number`
- `Math`
- `Date`
- `RegExp`
- `Error`
- `JSON`

Each object has the methods defined in the Ecma-262 standard. This extension does not contain any browser-specific objects or methods.

Examples

You have access to the `Math` object with all of its constants and methods:

```
let pi   = Math.PI
let sq25 = Math.Sqrt 25.
```

Strings can be manipulated with the ECMA `String` object and its methods:

```
let str       = new String("a lowercase string")
let upperstr  = str.ToUpperCase() // "A LOWERCASE STRING"
let tenthchar = str.CharAt(10) // "e"
let substring = str.Substring(2,11) // "lowercase"
```

`RegExp` objects can be used to manipulate text:

```
let str    = new String("Bob likes pineapples.")
let regex  = new RegExp("^\w+") // matches the first word
let newstr = str.Replace(regex,"Alice")
```

11.3 WebSharper™ Extensions for jQuery

The jQuery extension exposes jQuery 1.6. The API is as far as possible a one-to-one mapping of the JavaScript API, making it straightforward to convert existing jQuery code to F#.

Selecting DOM Nodes

jQuery enables you to construct wrappers for DOM nodes or to create new elements by supplying a string argument to the `jQuery` function:

```
var itemElems = jQuery(".Item")
var myNewElem = jQuery("<p>Foo</p>")
```

In the jQuery WebSharper™ extension, the same functionality is provided by the static member `Of` on the `JQuery` class:

```
let itemElems = JQuery.Of(".Item")
let myNewElem = JQuery.Of("<p>Foo</p>")
```

Methods

The return value of the `Of` method is an object of type `JQuery`, containing all the familiar instance members.

The following JavaScript example shows how you can invoke the `ready` function on the jQuery object:

```
jQuery(document).ready(function(){
   // Your code here
});
```

The equivallent code in WebSharper™ is:

```
JQuery.Of(Dom.Document.Current).Ready(fun _ ->
    // Your code here
)
```

In jQuery, functions are often flexible with regards to their input parameters. In F# this is represented by overloaded functions corresponding to different ways to invoke a method.

For example the `fadeOut` function that hides an element after applying a *fade-out* effect accepts various types of arguments, e.g:

```
jQuery("#MyElem").fadeOut()
jQuery("#MyElem").fadeOut("slow")
jQuery("#MyElem").fadeOut(100, function () {alert("Faded out");})
```

In F#, you write:

```
JQuery.Of("#MyElem").FadeOut()
JQuery.Of("#MyElem").FadeOut("slow")
JQuery.Of("#MyElem").FadeOut(100., fun () -> JavaScript.Alert "Faded out")
```

Chaining

Just like in pure jQuery, `chaining` of method invocations is supported since the result type of most jQuery operations is another jQuery object. Here is an example of chaining in JavaScript:

```
jQuery('#MyDiv').removeClass('Off').addClass('On')
```

And the corresponding code written in F#:

```
JQuery.Of("#MyDiv").RemoveClass("Off").AddClass("On")
```

Ignoring Return Values

For situations when the result of a method invocation can be ignored, the extension provides an extra property `Ignore`, which simply changes the return type to unit in F#:

```
JQ.Of("#MyElem").FadeOut().Ignore
```

Implicit Arguments

Callback functions in JavaScript are sometimes passed an implicit argument - `this`. jQuery makes heavy use of this idiom. Here is an example of the `each` function:

```
jQuery("div").each(function () {
    jQuery(this).hide();
});
```

The `this` object refers to the current element when traversing the jQuery collection.

In the WebSharper™ extension the `this` parameter is explicit. The code is written as:

```
JQuery.Of("div").Each(fun el ->
    JQuery.of(el).Hide().Ignore
))
```

Dom Manipulation

The following example changes the background of every second list item in all the list with the ID `MyList`:

```
[<JavaScript>]
let ChangeBackground () =
    JQuery.Of("#MyList li").Each(fun (el: Dom.Element) ix ->
        if ix % 2 = 0 then
            JQuery.Of(el).Css("background-color", "red").Ignore
    )
    |> ignore
```

Ajax

Here is an example using the `GetJSON` function for fetching JSON data from the server.

```
[<JavaScriptType>]
type Data =
    {
       Name : string
       Email : string
    }
[<JavaScript>]
let AjaxCall () =
    JQuery.GetJSON("data.json", fun (data, _) ->
        let data = As<Data> data
        let nameLabel =
            JQuery.Of("<div/>").Text("Name: " + data.Name)
        let descrLabel =
            JQuery.Of("<div/>").Text("Email: " + data.Email)
        JQuery.Of("<p/>").
            Append(nameLabel).
            Append(descrLabel).
            AppendTo("body").
            Ignore
    )
```

Attaching Event Handlers

Below is an example of constructing a button and adding an event handler for the click event:

```
[<JavaScript>]
let ButtonWithEvent () =
    JQuery.Of("<button/>").
        Text("Click").
        Click(fun _ _ ->
            Window.Alert("Button clicked")
        ).
        AppendTo("body").
        Ignore
```

12 Mobile Applications

WebSharper™ includes support for building mobile applications for the Android and Windows Phone 7/7.1 platforms. The strategy is to pre-generate HTML, JavaScript and CSS files, and then package them as a native application for the target platform, exposing some of the native API to the JavaScript runtime. In addition, the server-side code can be deployed to a public server, and the mobile devices are then able to communicate with it using the WebSharper Remoting/RPC mechanism.

12.1 Android Applications

To build Android applications with WebSharper you will need:

- JDK (Such as Sun Java SE Development Kit 1.7 update 3)
- Android SDK (Recommended versions: 2.2, 3.2)

You will also need either of the following:

- Eclipse and Android Development Tools (ADT) plugin
- Apache Ant

To create a new application, open Visual Studio and create a new project using the "Android Application" template. This creates a variant of the standard WebSharper HTML site. It also generates a new Android application under the `android` folder.

12.1.1 Building with Eclipse/ADT

The recommended way to build and debug the Android application is by using Eclipse with the Android Development Tools (ADT) plugin. After installing the plugin, open Eclipse, import the project from the file system, pointing it to the `android` sub-folder in your WebSharper solution.

12.1.2 Building with Ant

You can also build the application from the command-line or on a continuous build server. Install Apache Ant and set `ANT_HOME`, `JAVA_HOME` and `ANDROID_SDK` environment variables to the installation paths of Ant, JDK, and Android SDK respectively. Your MSBuild-based builds will now invoke Ant to produce debug and release Android packages automatically in the `android\bin` folder.

Note that release packages should be signed before distribution. To enable signing with command-line builds, generate a keystore with the `keytool.exe` tool that comes with the JDK and configure `android\ant.properties`.

12.2 Windows Phone Applications

Windows Phone application development requires:

- Windows Vista or Windows 7
- Windows Phone 7.1 SDK
- Microsoft Visual Studio 2010 Service Pack 1

Create a new project using the "Windows Phone" template and build it. The default build of the Mobile project starts a Windows Phone Emulator and connects the debugger to it, enabling you to see log messages from WebSharper in the Visual Studio Output console (Debug).

Consider tweaking the mobile project template to customize your application metadata and enable signing.

13 News

The 2.x releases of WebSharper™ introduced a number of improvements and new features:

- Better code generation. Instead of a direct code generation strategy, WebSharper™ now translates F# to JavaScript via a simple intermediate functional language, Core JavaScript. Core expressions are then optimized, improving the size and speed of the generated code. The compressed JavaScript files generated by WebSharper™ 2.0 are up to 50% smaller compared to WebSharper™ 1.0.
- Server-side website composition with *sitelets*. This new abstraction allows to construct complete multi-page websites in a F#-friendly manner.
- Improved remote procedure calls. WebSharper™ 2.0 introduces instance-level RPC calls and handler factories, improving testability and allowing the use of IOC containers.

  Templates for ASP.NET MVC projects are now included.
- A simpler, JavaScript-friendly interface model. Unlike WebSharper™ 1.0, WebSharper™ 2.0 does not prefix interface method names with the name of the interface. While this does not allow an object to distinctly implement two interfaces with clashing member names, the new interface model is more in the spirit of JavaScript and is easier to use with JavaScript libraries.
- Refined resource dependency management. The resource graph is now refined to the level of methods, allowing for safe exclusion of unnecessary resources and faster page loads in certain scenarios.
- Refined library packaging. The WebSharper™ standard library has been partitioned into a few components, allowing to improve page loads for pages that use only a subset of the available features.

- JSON serialization extended to classes with a default constructor and serializable fields.
- Improved command-line tools and build integration. `WebSharper.exe` is now more easily scriptable. The MSBuild integration shipped with the platform now invokes `WebSharper.exe` on individual F# library projects to produce the JavaScript and resource dependency files, and these are copied to the web projects during build, avoiding needless recompilation.
- Extensible custom attributes. Custom attribute have been refactored to provide base classes that allow the user to override their logic.

14 Installation

For installation, configuration and upgrade instructions, please visit the WebSharper™ website.

15 Acknowledgements

JavaScript has been used a number of times as a compilation target:

- fswebtools - a very similar project targeted at an earlier F# release
- ocamljs - translates dlambda output (core) of the OCAML compiler to JS
- smltojs - compiles SML to JS
- yhc/javascript - compiles Haskell to JS
- scheme/js - compiles Scheme to JS
- gwt - compiles Java to JS
- jsc - translates .NET IL to JS
- Script# - translates C# to JS
- ST2JS - compiles SmallTalk to JS
- links - a Haskell-inspired JS-compiling web language.