WebSharper:
Creating Functional, Reactive F# Web Applications

Adam Granicz
IntelliFactory
#granicz

#websharper
#trywebsharper

You are in good company

80k+ downloads
50+ mainstream JavaScript libraries ("extensions")

60+ talks in 30+ cities in 20+ countries

WebSharper W#

Agenda

Part I

- Warm-up – why are we doing this?
- Random bits on F#
- WebSharper
  - What resources are out there to learn from
  - Installing, Project templates
  - Getting Started examples
  - Fundamentals
    - HTML combinators and templates
    - Pagelets – markup and events
    - Sitelets – request routing, safe URLs, serving contexts

Agenda

Part II

- Reactive development
  - UI.Next, Dynamic dataflow, Reactive DOM and templating
- Functional user interfaces
  - Formlets/Flowlets, Piglets, WebSharper.Forms
- Working with JavaScript libraries
  - WIG, resources, proxies
- Hands-on examples – REST, data charting, FRP

Background

CEO of IntelliFactory,
The F# Company

Started FP in 1999 with OCaml, worked on generic frontends, multi-lingual compilers and theorem proving at Caltech

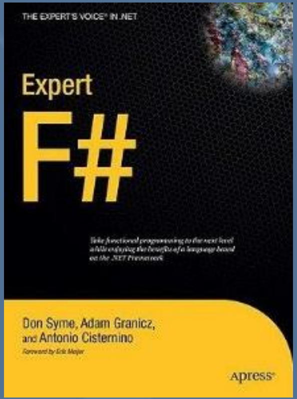
6x F# MVP – 2010 - present

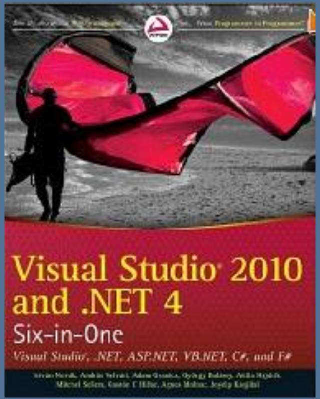
Coauthor of 5 F# books, 4 of them with Don Syme, the designer of F#

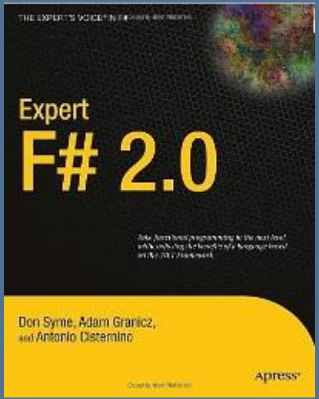
Regular speaker in numerous developer conferences, and committee member in academic conferences/workshops

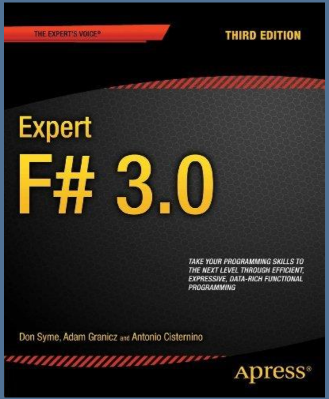
F# books

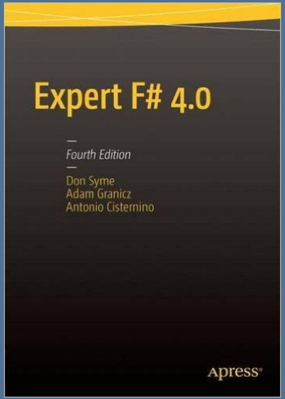
- Expert F# - 2007
- Expert F# 2.0 – 2010
- Visual Studio 2010 and .NET 4 Six-in-One – 2010
- Expert F# 3.0 – 2012
- Expert F# 4.0 - 2015


THE EXPERT'S VOICE® IN .NET
Expert
F#
Don Syme, Adam Granicz, and Antonio Cisternino
Apress®


Visual Studio® 2010 and .NET 4
Six-in-One
Visual Studio®, .NET, ASP.NET, VB.NET, C#, and F#




THE EXPERT'S VOICE®
THIRD EDITION
Expert
F# 3.0
TAKE YOUR PROGRAMMING SKILLS TO THE NEXT LEVEL THROUGH EFFICIENT, EXPRESSIVE, DATA-RICH FUNCTIONAL PROGRAMMING
Don Syme, Adam Granicz and Antonio Cisternino
Apress®

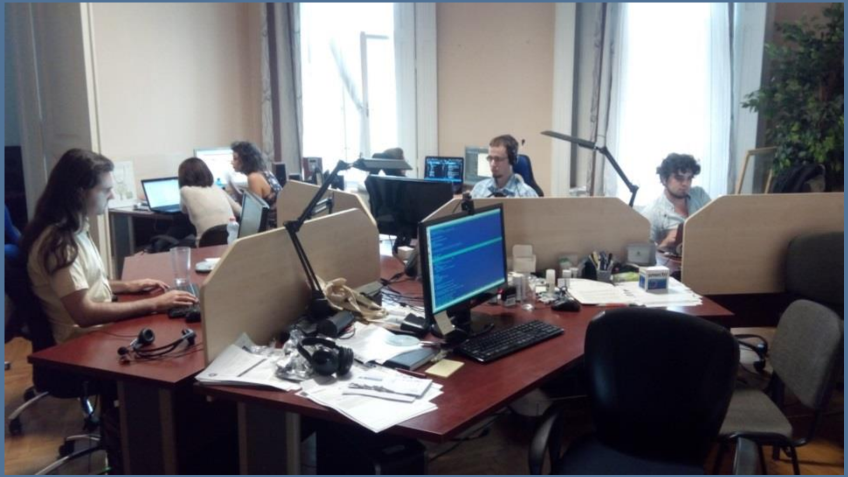


IntelliFactory

F# consulting, training, development

Headquartered in Budapest
Founded in 2004

- Doing functional, reactive web development in F#
- Making web and cloud technologies for developers
- Extensive experience with building full-stack F# enterprise apps



- 280+ F# projects, ~60 open source
- One of the largest F# codebases around
- 30+ commercial applications built

Motivation

Web applications are everywhere, but they still require a myriad of languages and libraries/technologies

We want to

- Yield more programmer productivity
- Distill FP abstractions for the web
- Make these widely and easily available
- Trigger industry adoption

F#

.NET, F#, and WebSharper

- .NET ecosystem
  Multiple languages (C#, F#, VB, IronPython/Ruby)
  Industry standard libraries (web, data, communication, ...)
- F#
  Functional-first
  Advanced features (active patterns, units of measure, TPs, ...)
  Highest ranked FP language on the TIOBE index
- WebSharper
  The largest F# web ecosystem – 80k+ downloads
  Entire client-server applications in F#
  F# to JavaScript compiler
  Web abstractions (formlets, flowlets, sitelets, piglets, ...)

F# ...

- Is a functional programming language developed by MS(R)
- Is an ideal language for rapid and robust software development
- Packs more functionality in less code – script-like syntax
- Yields code that is easier to extend and maintain
- Is a standard front-end in Visual Studio
- Has full access to the .NET APIs and components
- Runs within the .NET CLR, making it possible to use within existing .NET projects

F# - Key benefits

- Application code is considerably shorter than in C#, VB, or Java
- Dramatically reduces development time by providing powerful programming constructs
- Ideal for a wide range of domains including finance, science and technology, and those with numerical and symbolic computation
- Language support for developing distributed, parallel, asynchronous and reactive applications

Why F#?

- Functional core language
  - Functions as values
  - Anonymous and higher-order functions
  - Parametric polymorphism
  - Type inference
  - Functional data structures
  - Pattern matching
  - Lazy vs. eager evaluation
- Cool, more advanced features
  - Active patterns
  - Units of measure
  - Type Providers

Snippet

```
type Expr =
    | Integer of int
    | Binop   of (int -> int -> int) * Expr * Expr
with
    static member Sum  (e1, e2) = Binop (( + ), e1, e2)
    static member Diff (e1, e2) = Binop (( - ), e1, e2)
    static member Prod (e1, e2) = Binop (( * ), e1, e2)
    static member Div  (e1, e2) = Binop (( / ), e1, e2)

let rec Eval = function
    | Integer i         -> i
    | Binop (f, e1, e2) -> f (Eval e1) (Eval e2)

let _ =
    let i = Expr.Integer
    Expr.Prod (Expr.Sum (i 4, i 9), Expr.Diff (i 9, i 4))
    |> Eval
    |> printf "Result=%d"
```

Bindings

Bindings assign a name to a value, example:

```
let x = 1+1
```

In pure functional programming, bound values do not change. An attempt to “change” them:

```
let x = 1+1    -->  let x = 1+1
let x = 3           let x1 = 3
```

... is called shadowing, where the latter binding shadows the former one, effectively losing a reference.

Bindings

Consider the Single Static Assignment (SSA) form:

```
let x = 1                  let x = 1
let x = x+1       ->       let x2 = x+1
let x = x+2                let x3 = x2+3
x+3                        x3+3
```

SSA = each name is assigned a value once, thus bindings are immutable.

Type inference

Types of values do not need to be annotated, they can usually be inferred.

In some cases, they can and need to be.

```
let (x: int) = 1
```

Key data structures

Tuples $(T_1, T_2, ...)$

Records $\{ Field_1 = V_1; Field_2 = V_2; ... \}$

Discriminated

unions $| \; Shape_1 \text{ of } T_1$

$| \; Shape_2 \text{ of } T_2$

$| \; ...$

Sequences (lazy), lists, sets, maps, arrays, etc.

Functional data structures

Purely functional data structures are immutable.

Notable imperative data structures:

- arrays
- records with mutable fields
- reference cells

Lazy vs Eager data structures

Sequences are lazy, e.g. their elements are computed on demand.

Eager data structures compute their elements at the time the data structure is created.

Intervals and Comprehensions

Intervals and comprehensions are expressions that describe how to generate successive elements.

Examples:

```
0 .. 100
0 .. 2 .. 100
for i in 1 .. 100 -> i*i
```

Containers

Comprehensions and enumerating elements can be placed inside containers (data structures with elements of a given type):

<table>
<tr><th></th><th></th></tr>
<tr><td>{ ... }</td><td>Sequence</td></tr>
<tr><td>[...]</td><td>List</td></tr>
<tr><td>[| ... |]</td><td>Array</td></tr>
</table>

Example:

```
{ for i in 1 .. 100 -> i*i }  // The first 100 squares
```

Functions as values

Functions can be returned or taken as parameters

- Functions are first-class values just like strings and numbers
- Higher order functions (HOFs)

Anonymous functions (lambdas) are function values with a name:

```
fun x -> x+1
fun x y -> x+y
```

Functions as values

Named functions are simply lambdas assigned a name:

```
let foo = fun x -> x+1
let foo x = x+1
```

Functions as values

Functions with multiple parameters are written without parentheses. Those parameters within parentheses are single, tupled, parameters.

```
let f x y = x+y         // two arguments
let f (x, y) = x+y      // single argument
```

Pattern matching

Pattern matching is a mechanism to examine structured input data by decomposing it into smaller parts to match against.

Ordinary pattern matching works on constants, discriminated unions, tuples, records, lists and arrays.

Units of Measure

```
[<Measure>] type C
[<Measure>] type F

let ConvertCtoF ( c: float<C> ) =
    9.0<F> / 5.0<C> * c + 32.0<F>
```

Similar implementations elsewhere:

Haskell: Dimensional

Ruby: Quantity.rb

C++, etc.

Active Patterns

Partitioning:

```
let (|Even|Odd|) input =
    if input % 2 = 0 then Even else Odd
```

Decomposition:

```
let (|HSB|) (col: System.Drawing.Color) =
    (col.GetHue(), col.GetSaturation(),
     col.GetBrightness())
```

Partial Active Patterns

“Shaving off” from the input value space:

```
let (|Integer|_|) (str: string) = ...
let (|Float|_|) (str: string) = ...

let ParseNumber = function
    | Integer i -> ...
    | Float f -> ...
    | _ -> ...
```

WebSharper

WebSharper

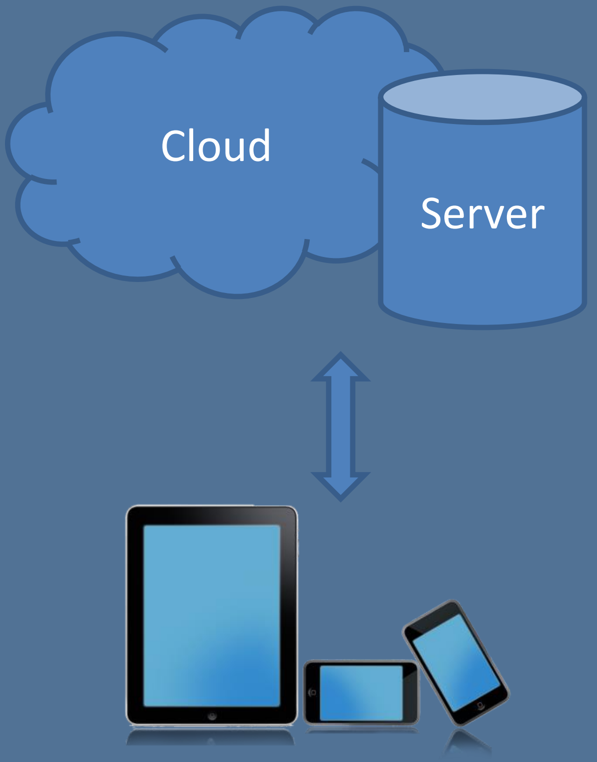
Open source project, available at:

```
https://github.com/IntelliFactory/websharper
```

Code contributions are welcome

Bridging the language mismatch

```
open WebSharper

module Server =
    [<Rpc>]
    let MyServerFunction(...) = ...

module Client =
    [<JavaScript>]
    let MyClientFunction(...) =
        ...
        let v = MyServerFunction(...)
        ...
```



Getting WebSharper

- Downloads for Visual Studio and Xamarin Studio
- In an online IDE – cloudsharper.com
- Using yeoman (generator-fsharp)

```
npm install -g yo
npm install -g generator-fsharp
yo fsharp
```

Project templates

- UI.Next vs WebSharper.Html
  - Single-Page Applications (SPAs) - client-only
  - Client-Server Applications - sitelet-based
  - HTML Applications - client only, sitelet-based

`http://websharper.com/docs/templates`

WebSharper

What do I get with WebSharper?

1. Automatic resource management
   - => no need to include dependencies by hand
   - => each page loads only the resources it needs
2. Type-safe access to JavaScript libraries via F#
   - => dozens of extensions available (visualization,charts,...)
   - => has its own eDSL for describing JavaScript APIs
   - => TypeScript type provider in upcoming version

WebSharper

What do I get with WebSharper?

3. Uniform programming model (everything is F#)

   => write all server and client code in F#

4. Client-Server applications

   => `[<JavaScript>]` vs `[<Rpc>]` annotations

   => Seamless communication via RPC

   => No need to worry about how to pass data

WebSharper

What do I get with WebSharper?

5. Composable functional programming abstractions

   a) Pagelets: to represent dynamic markup and behavior

   b) Sitelets: to represent web applications

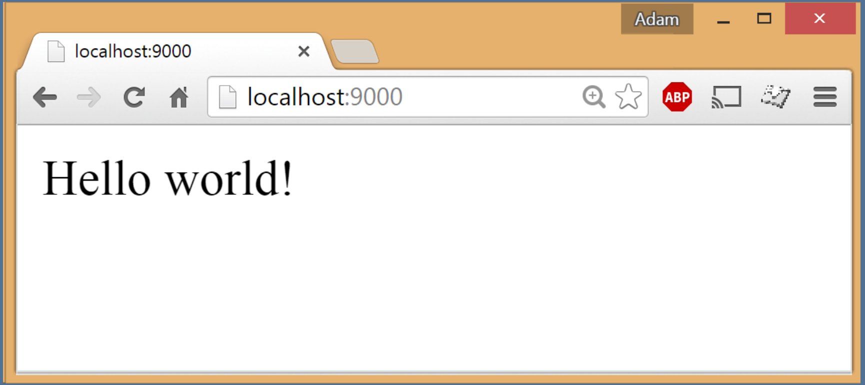
   c) Formlets: to represent complex and dependent web forms

   d) Flowlets: to represent sequences of user forms

   e) Piglets: formlets on steroids: UIs for any device

Getting Started

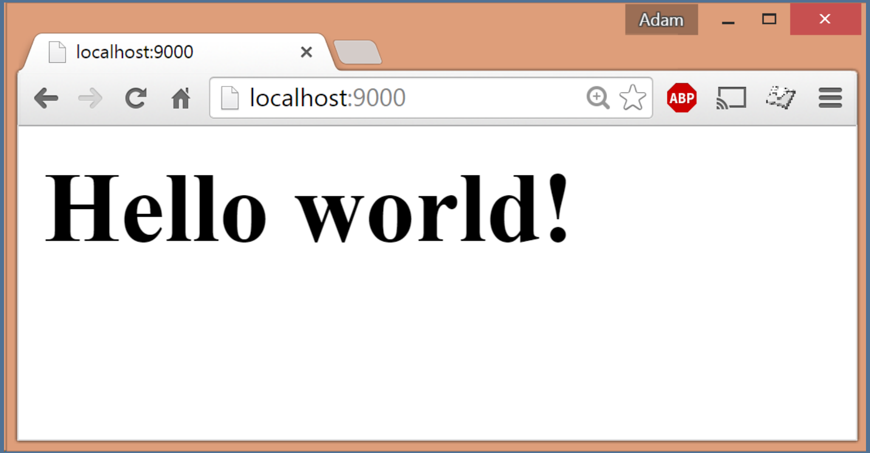
```
module MyApplication

open WebSharper
open WebSharper.Sitelets

[<Website>]
let Main =
    Application.Text (fun ctx ->
        "Hello World!")
```



Hello world!

Getting Started - SPAs

```
module MyApplication

open WebSharper
open WebSharper.Sitelets
open WebSharper.UI.Next.Html
open WebSharper.UI.Next.Server

[<Website>]
let Main =
    Application.SinglePage (fun ctx ->
        Content.Page(h1 [text "Hello World!"]))
```



Hello world!

Getting Started – Single endpoint apps

```
type EndPoint = int

[<Website>]
let Main =
    Sitelet.Infer (fun ctx (endpoint: EndPoint) ->
        match endpoint with
        | i -> Content.Text (string (i*i))
    )
```

Getting Started - Endpoints

<table>
<tr><th>Endpoint Type</th><th>Sample Request</th><th>Parsed Request</th></tr>
<tr><td>Int</td><td>/12</td><td>12</td></tr>
<tr><td>Float</td><td>/12.34</td><td>12.34</td></tr>
<tr><td>String</td><td>/abc1234</td><td>"abc1234"</td></tr>
<tr><td>System.Net.HttpStatusCode</td><td>/200</td><td>HttpStatusCode.OK</td></tr>
<tr><td>System.DateTime</td><td>/2015-08-24-12.55.14</td><td>System.DateTime(2015,8,24,12,55,14)</td></tr>
<tr><td>string * int</td><td>/abc/1234</td><td>("abc", 1234)</td></tr>
<tr><td>{ Name: string; Age: int }</td><td>/john/12</td><td>{ Name="John"; Age=12 }</td></tr>
<tr><td>string option</td><td>/None
/Some/abc</td><td>None
Some "abc"</td></tr>
<tr><td>int list
float list
string list</td><td>/2/1/2
/2/1.1/2.2
/2/abc/1234</td><td>[1; 2]
[1.1; 2.2]
["abc"; "1234"]</td></tr>
<tr><td>int array
float array
string array</td><td>/2/1/2
/2/1.1/2.2
/2/abc/1234</td><td>[|1; 2|]
[|1.1; 2.2|]
[|"abc"; "1234"|]</td></tr>
</table>

Getting Started – Endpoint modifiers

- `[<EndPoint ...>]`: Specifying URL/method pairs

```
type EndPoint =
  | [<EndPoint "GET /about">] About
```

Getting Started – Endpoint modifiers

- `[<Query("param1", ...)>]`: specifying query parameters

```
type EndPoint =
    | [<EndPoint "/doc"; Query "version">] Document of int * version: int option
```

Sample Request	Parsed Request
`/doc/1234?version=1`	`Document(1234, Some 1)`
`/doc/1234`	`Document(1234, None)`

Getting Started – Endpoint modifiers

- [<Json “param”>]: Specifying arguments to be passed as JSON (on POST)

```
type EndPoint =
    | [<EndPoint "POST /create"; Json "order">]
      CreateOrder of data: OrderData

and OrderData =
    { item: string; quantity: int }
```

Sample Request	Parsed Request

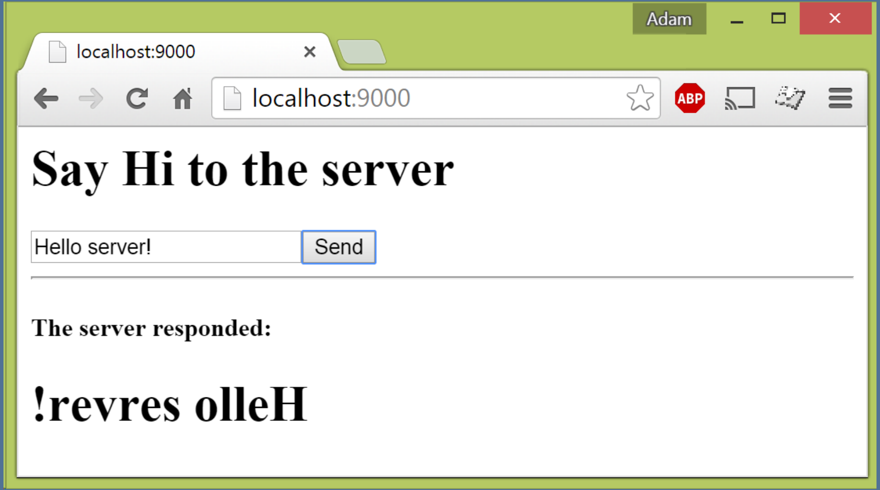
/create
{ item:"Book", quantity:1 }
	CreateOrder({
	item="Book";quantity=1 })

Getting Started – Endpoint modifiers

- `[<FormData("param1", …)>]`: Specify arguments to be passed as form data

Getting Started – Multiple endpoint apps

```
type EndPoint =
    | [<EndPoint "/">] Home
    | [<EndPoint "/about">] About
```



Say Hi to the server

Hello server! Send

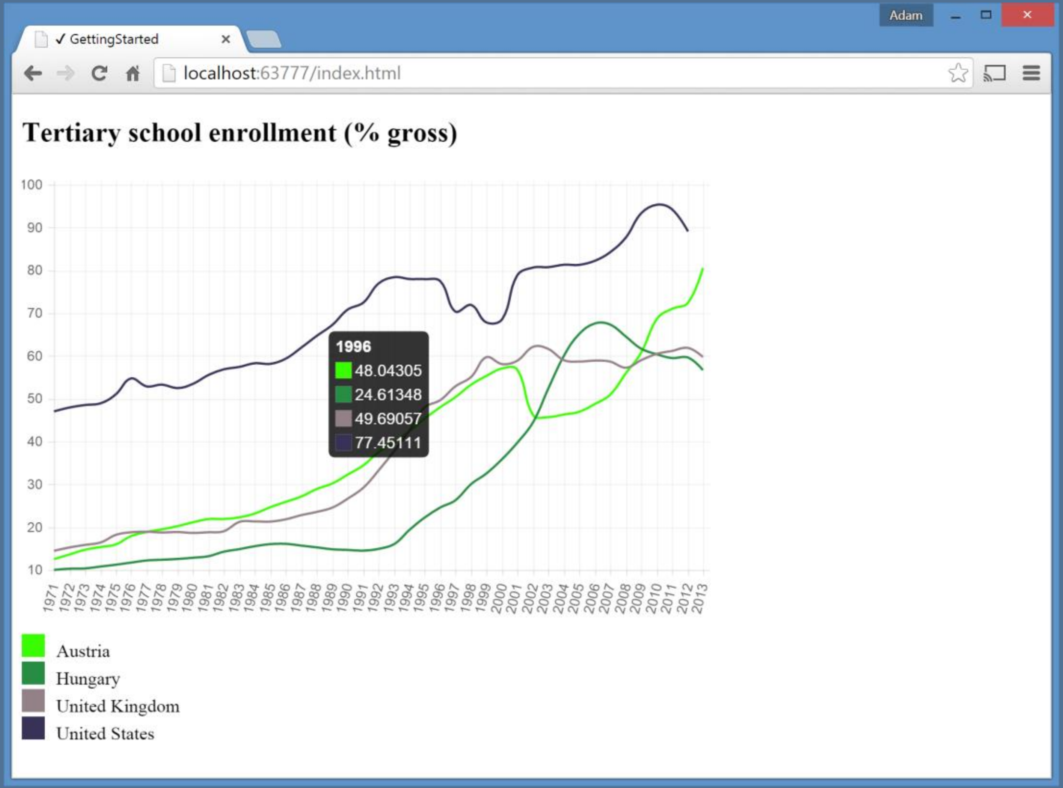
The server responded:

!revres olleH

Getting Started – MPAs (client-server)

```
let HomePage ctx = ...
let AboutPage ctx = ...

[<Website>]
let Main =
    Application.MultiPage (fun ctx endpoint ->
        match endpoint with
        | EndPoint.Home -> HomePage ctx
        | EndPoint.About -> AboutPage ctx
    )
```

Getting Started – Working with data



http://try.websharper.com/snippet/adam.granicz/00003p

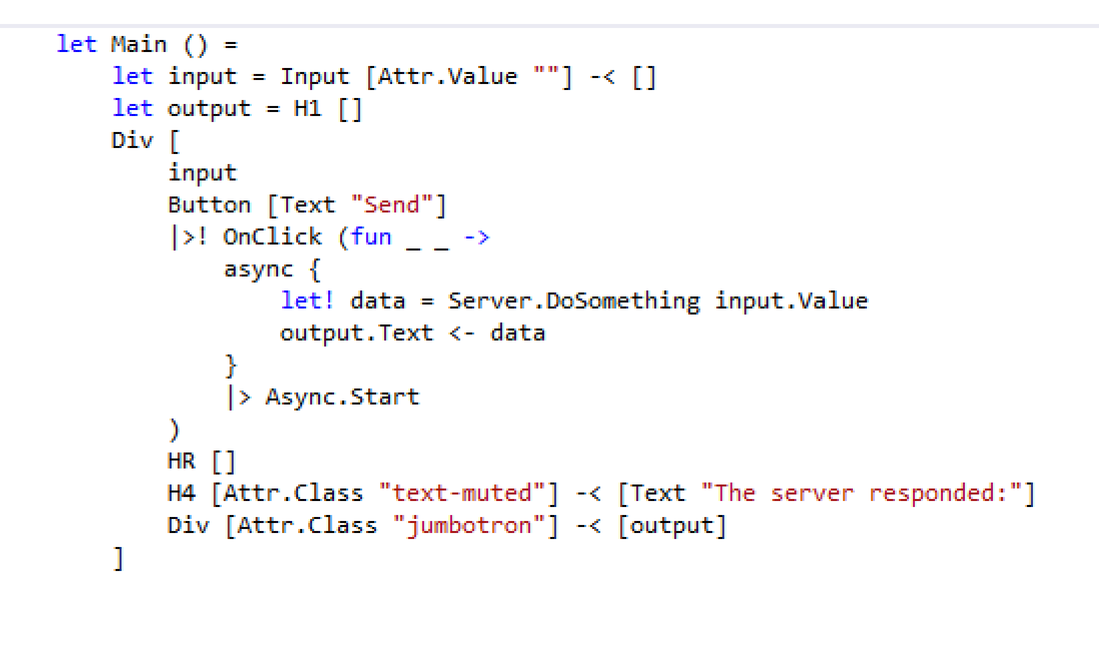
Pagelets

Pagelets

Constructing markup with dynamic behavior

Pagelets – WebSharper.Html

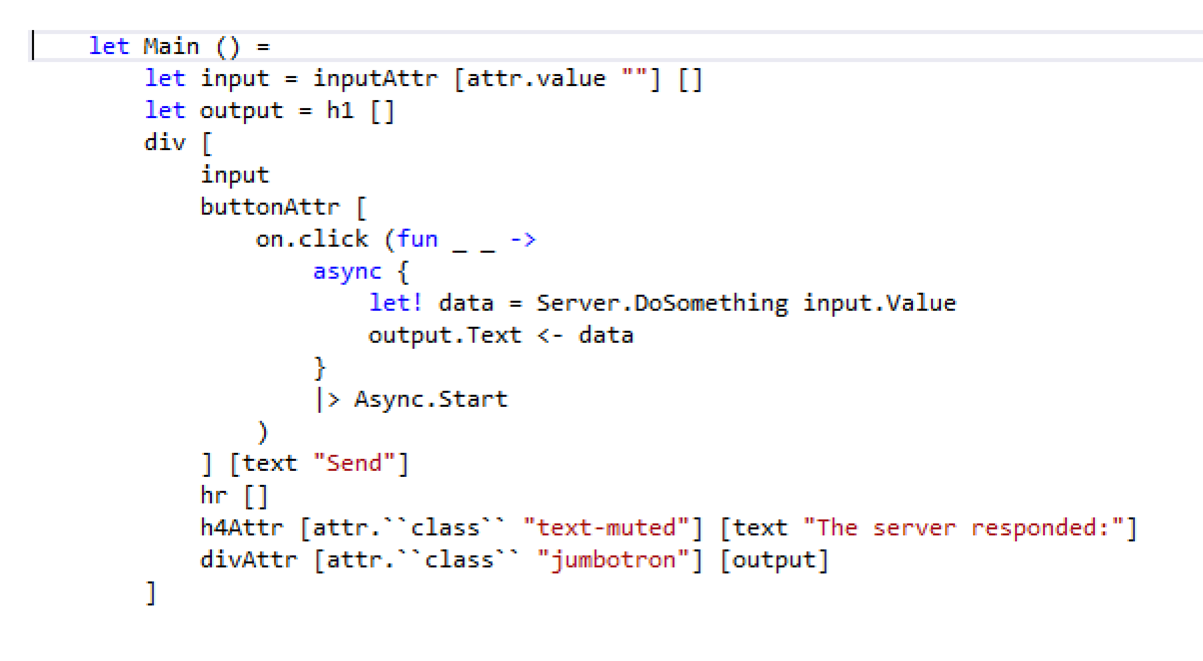
In `WebSharper.Html.Client/Server`



```
let Main () =
    let input = Input [Attr.Value ""] -< []
    let output = H1 []
    Div [
        input
        Button [Text "Send"]
        |>! OnClick (fun _ _ ->
            async {
                let! data = Server.DoSomething input.Value
                output.Text <- data
            }
            |> Async.Start
        )
        HR []
        H4 [Attr.Class "text-muted"] -< [Text "The server responded:"]
        Div [Attr.Class "jumbotron"] -< [output]
    ]
```

Pagelets – UI.Next

In WebSharper.UI.Next.Html.Client/Server



```
    let Main () =
        let input = inputAttr [attr.value ""] []
        let output = h1 []
        div [
            input
            buttonAttr [
                on.click (fun _ _ ->
                    async {
                        let! data = Server.DoSomething input.Value
                        output.Text <- data
                    }
                    |> Async.Start
                )
            ] [text "Send"]
            hr []
            h4Attr [attr.``class`` "text-muted"] [text "The server responded:"]
            divAttr [attr.``class`` "jumbotron"] [output]
        ]
```

Pagelets – DOM combinators

HTML	**UI.Next**	**WebSharper.Html**
Plain text	text "Plain text"	Text "Plain text"
class "abc" src "abc"	attr.``class`` "abc" attr.src "abc"	Attr.Class "abc" Attr.Src "abc"
<h1>ABC</h1>	h1 [text "ABC"]	H1 [Text "ABC"]
<div> <div>..</div> </div>	div [div [...]]	Div [Div [...]]
<div class="abc"> <div>..</div> </div>	divAttr [attr.``class`` "abc"] [div [...]]	Div [Attr.Class "abc"] -< [Div [...]]

<div onclick="...">
    <div>...</div>
</div>
	divAttr [
	    on.click <@ fun e arg ->
	        ... @>
	] [
	    div [...]
	]
		Div [
		    Div [...]
		]
		|>! OnClick (fun e arg -> ...)

Sitelets

Sitelet combinators

```
Application.SinglePage
Application.MultiPage
Application.Text
+
Sitelet.Empty
Sitelet.Content
Sitelet.Sum
Sitelet.Map
Sitelet.Protect
Sitelet.Infer
<|>, Sitelet.Shift, Sitelet.Folder
```

Authenticated sitelets

- `Sitelet.Protect` - see GH repo for example

HTML and other responses

- Plain text using `Content.Text`
  - `Content.Text “Hello World!”`
- JSON using `Content.Json`

  -
    ```
    type Person = { First: string; Last: string; Age: int }

    Content.Json { First="John"; Last="Smith"; Age=30 }
    ```
- Files using `Content.File`

  -
    ```
    Content.File("../../Main.fs",
                 AllowOutsideRootFolder=true,
                 ContentType="text/plain")
    ```

HTML and other responses

- Error codes
  - `Content.Unauthorized` – Error code 401
  - `Content.Forbidden` – Error code 403
  - `Content.NotFound` – Error code 404
  - `Content.MethodNotAllowed` – Error code 405
  - `Content.ServerError` – Error code 500
  - `Content.Custom(Status=Http.Status.Custom 402 (Some "Payment Required"))`

Reactive development with UI.Next

UI.Next

WebSharper's reactive dynamic dataflow and DOM construction library.

Reactive data model + Reactive DOM/presentation layer

S. Fowler, L. Denuziere, A. Granicz. *Reactive Single-Page Applications with Dynamic Dataflow*. PADL 2014.

Vars: observable, mutable reference cells

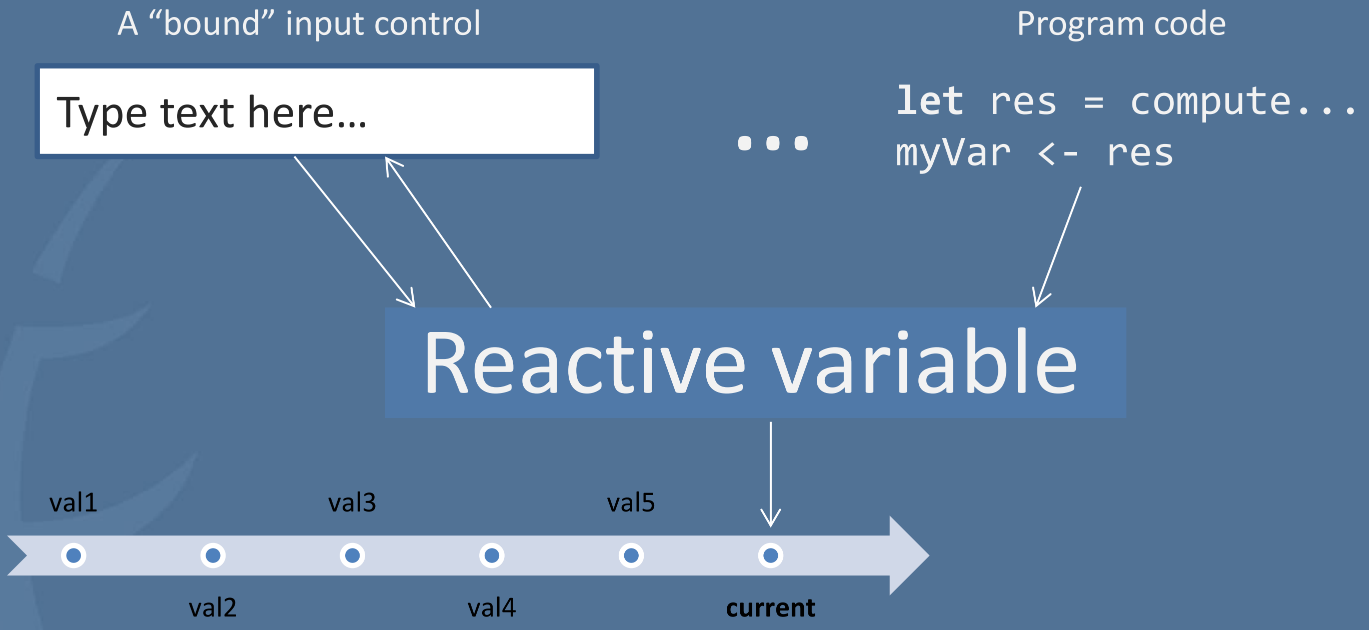
Views: projections of `Var`'s in the dataflow graph

UI.Next reactive DOM

Represented by the Doc type

- Monoid – can be empty, can be concatenated
- Can contain reactive DOM nodes

Basics – Two-way data binding



A “bound” input control

Type text here…

Program code

```
let res = compute...
myVar <- res
```

…

Reactive variable

Ex 1: Reactive vars, bound controls, and views

```
let var = Var.Create ""
let view = View.FromVar var
let input = Doc.Input [] var
let capitalized = view |> View.Map (fun txt -> txt.ToUpper())
let label = textView capitalized
div [
    input
    label
]
```

http://try.websharper.com/snippet/adam.granicz/00003N

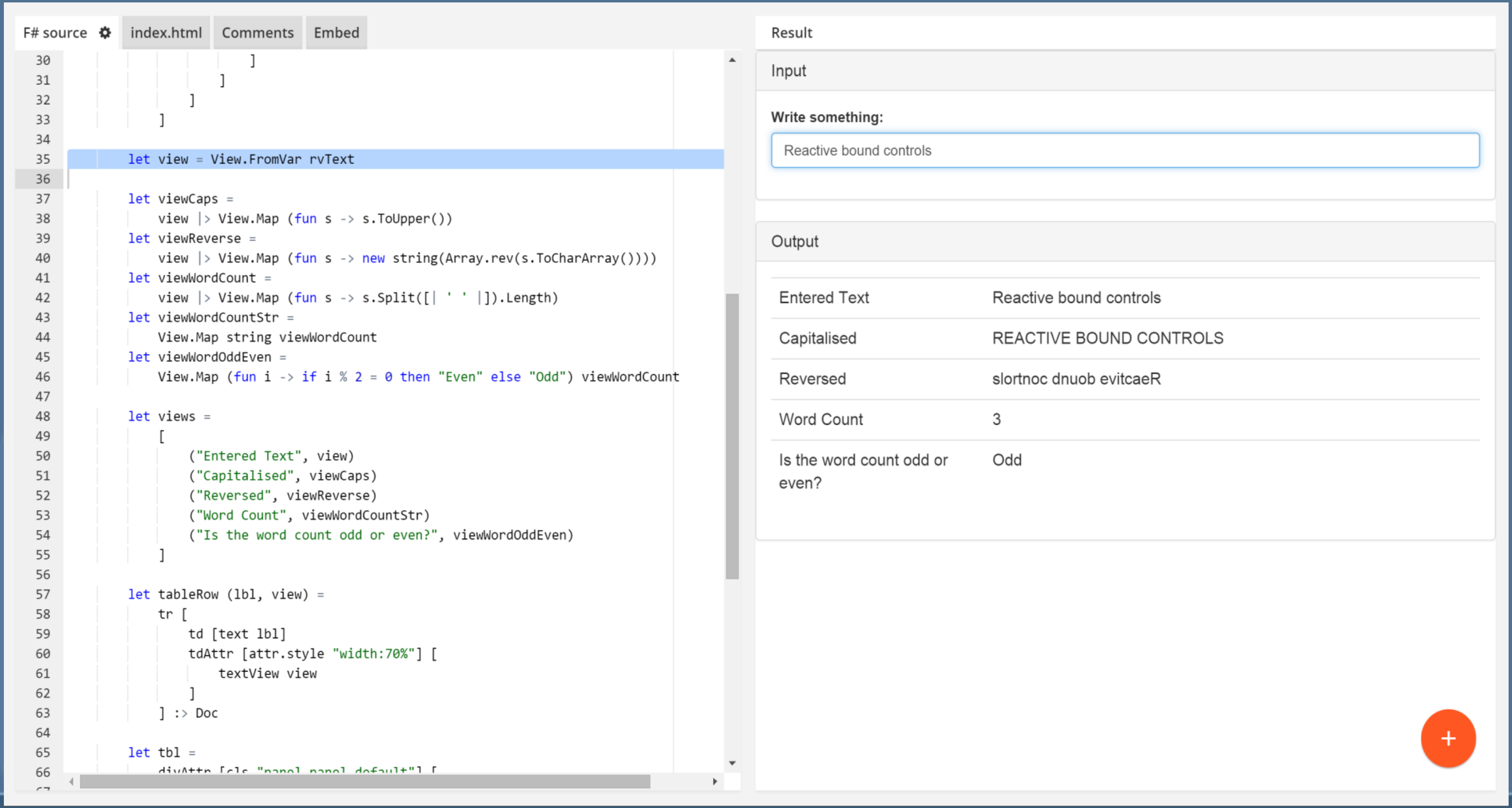
Ex 2: Reactive vars, bound controls, and views

```
open WebSharper.UI.Next
open WebSharper.UI.Next.Client

let v = Var.Create “first value”

let textbox = Doc.Input [] myVar

let view = View.FromVar v
```

Doc: a representation for a reactive DOM fragment (empty, or single, or multiple node)

http://try.websharper.com/snippet/adam.granicz/00001u

UI.Next example



```
                    ]
                ]
            ]
        ]

    let view = View.FromVar rvText

    let viewCaps =
        view |> View.Map (fun s -> s.ToUpper())
    let viewReverse =
        view |> View.Map (fun s -> new string(Array.rev(s.ToCharArray())))
    let viewWordCount =
        view |> View.Map (fun s -> s.Split([| ' ' |]).Length)
    let viewWordCountStr =
        View.Map string viewWordCount
    let viewWordOddEven =
        View.Map (fun i -> if i % 2 = 0 then "Even" else "Odd") viewWordCount

    let views =
        [
            ("Entered Text", view)
            ("Capitalised", viewCaps)
            ("Reversed", viewReverse)
            ("Word Count", viewWordCountStr)
            ("Is the word count odd or even?", viewWordOddEven)
        ]

    let tableRow (lbl, view) =
        tr [
            td [text lbl]
            tdAttr [attr.style "width:70%"] [
                textView view
            ]
        ] :> Doc

    let tbl =
        divAttr [cls "panel panel-default"] [
```

Input

Write something:

Reactive bound controls

Output

Entered Text	Reactive bound controls
Capitalised	REACTIVE BOUND CONTROLS
Reversed	slortnoc dnuob evitcaeR
Word Count	3
Is the word count odd or even?	Odd

HTML Templating

- UI.Next Type Provider

Uses a TP to read markup content and generate Docs with placeholders for reactive content and event handlers.

```
open WebSharper.UI.Next

type MyTemplate = Templating.Template<“main.html”>
```

Reactive template placeholders

`data-var`: bind the value of an input control to a reactive variable

`data-attr`: assign an attribute

`data-event-xxx`: bind an event handler for xxx

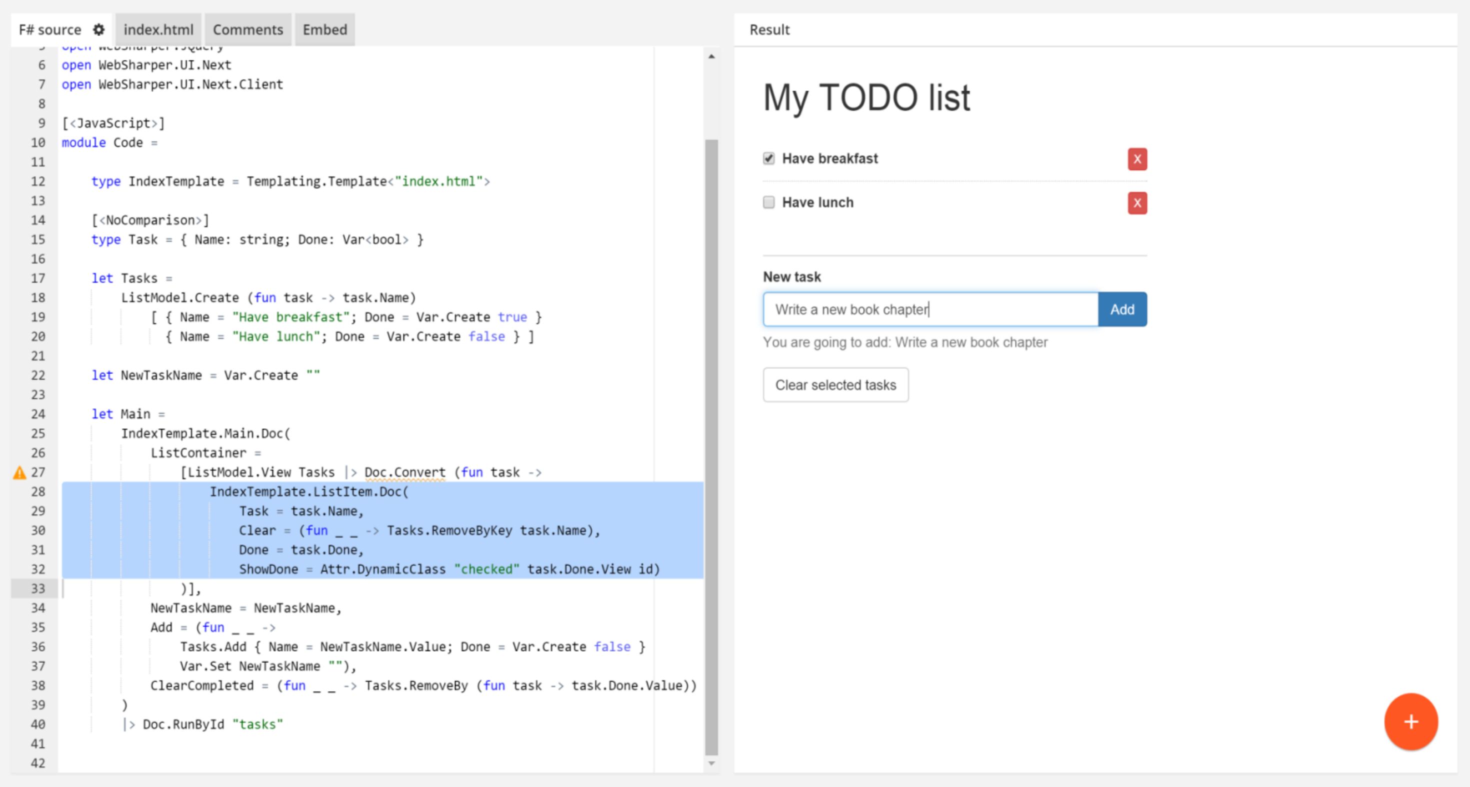
`data-template`: use the given node as a template

`data-children-template`: use the contents of the given node as a template

`$!{var}`: the view of a reactive variable

`http://try.websharper.com/example/todo-list`

Reactive templates



```
open WebSharper.JQuery
open WebSharper.UI.Next
open WebSharper.UI.Next.Client

[<JavaScript>]
module Code =

    type IndexTemplate = Templating.Template<"index.html">

    [<NoComparison>]
    type Task = { Name: string; Done: Var<bool> }

    let Tasks =
        ListModel.Create (fun task -> task.Name)
            [ { Name = "Have breakfast"; Done = Var.Create true }
              { Name = "Have lunch"; Done = Var.Create false } ]

    let NewTaskName = Var.Create ""

    let Main =
        IndexTemplate.Main.Doc(
            ListContainer =
                [ListModel.View Tasks |> Doc.Convert (fun task ->
                    IndexTemplate.ListItem.Doc(
                        Task = task.Name,
                        Clear = (fun _ _ -> Tasks.RemoveByKey task.Name),
                        Done = task.Done,
                        ShowDone = Attr.DynamicClass "checked" task.Done.View id)
                )],
            NewTaskName = NewTaskName,
            Add = (fun _ _ ->
                Tasks.Add { Name = NewTaskName.Value; Done = Var.Create false }
                Var.Set NewTaskName ""),
            ClearCompleted = (fun _ _ -> Tasks.RemoveBy (fun task -> task.Done.Value))
        )
        |> Doc.RunById "tasks"
```

My TODO list

☑ Have breakfast X

□ Have lunch X

New task

Write a new book chapter | Add

You are going to add: Write a new book chapter

Clear selected tasks

+

Reactive “sitelets”

Client-side routing

`http://try.websharper.com/snippet/adam.granicz/000033`

Formlets

Formlets

A compositional abstraction for constructing web forms based on applicative functors:

```
Formlet.Return (fun fn age -> { FirstName=fn; Age=age })
<*> Controls.Input "First name"
<*> (Controls.Input "20"
    |> Validation.IsMatch "^[1-9][0-9]*$" "Need an integer"
    |> Formlet.Map (int))
```

```
http://try.websharper.com/snippet/adam.granicz/00003G
```

Formlets

`Formlet.Return` – embedding pure expressions in a formlet

```
: ‘T -> Formlet<‘T>
```

`<*>` - sequencing formlets and combining their results

```
: Formlet<‘A -> ‘B> -> Formlet<‘A> -> Formlet<‘B>
```

Implemented via IF’s own reactive library, based on a partial implementation of Rx’s hot observables and explicit subscription to future value streams.

Reactive formlets

Enable formlet controls to be bound to a reactive variable.

No need to manually manage subscriptions, these are inferred from the dataflow graph.

Makes data binding natural and easy.

Two sets of controls available: with and without explicit Vars

Reactive formlets with explicit Vars

```
let FN = Var.Create "First name"
let AGE = Var.Create "20"

Formlet.Return (fun fn age -> { FirstName=fn; Age=age })
<*> Controls.InputVar FN
<*> (Controls.InputVar AGE
    |> Validation.IsMatch "^[1-9][0-9]*$" "Need an integer"
    |> Formlet.Map (int))

http://try.websharper.com/snippet/adam.granicz/00003P
```

Dependent formlets and flowlets

Enhance flowlets with dynamic composition

Use the bind operator (`let!` in an F# computation expr)

J. Bjornson, A. Tayanovskyy, A. Granicz. *Composing Reactive GUIs in F# using WebSharper*. IFL 2010.

```
Formlet.Do {
    let! fn = Control.Input “First name”
    let! age = (Control.Input “20” |> …)
    return { Firstname=fn; Age=age }
}
```

Customizing presentation via piglets

L. Denuziere, E. Rodriguez, A. Granicz. *Piglets to the Rescue*. IFL 2013.

```
Piglet.Return (fun user pass -> (user, pass))
<*> Piglet.Yield ""
<*> Piglet.Yield ""
|> Piglet.WithSubmit
|> Piglet.Run (fun (user, pass) ->
    JS.Alert ("Welcome, " + user + "!"))
|> Piglet.Render (fun rvUsername rvPassword submit ->
    form [
        ...
    ]
)
```

http://try.websharper.com/snippet/00004B

WebSharper.Forms = Reactive piglets

`Form.YieldVar`

`Var`'s can be bound to form controls

Form controls can be nested in reactive markup

WebSharper.Forms

```
let fname, age = Var.Create “...”, ...

Form.Return (fun fn age -> { FirstName=fn; Age=age })
<*> Form.YieldVar fname
<*> Form.YieldVar age
|> Form.WithSubmit
|> Form.Render (fun fn age submitter ->
    div [
        Doc.Input [] fn
        Doc.IntInputUnchecked [] age
        Doc.ButtonValidate "Submit" [] submitter
    ]
)
```

http://try.websharper.com/snippet/adam.granicz/00004Q

WebSharper.Forms.Bootstrap

http://try.websharper.com/snippet/adam.granicz/00004x

List models

```
type Task = { Name: string; Done: Var<bool> }

let Tasks =
    ListModel.Create (fun task -> task.Name)
       [ { Name = "Have breakfast"; Done = Var.Create true }
         { Name = "Have lunch"; Done = Var.Create false } ]
```

http://try.websharper.com/example/todo-list

Working with JavaScript libraries

50+ extensions to various JavaScript libraries

- Core: JQuery, EcmaScript, WebGL
- Visualization: Google Visualization, D3, Raphael, Protovis, etc.
- Charting: Highcharts, Chart.js, etc.
- GIS: Google Maps, Bing Maps, Leaflet.js
- Mobile: jQuery Mobile, Sencha Touch, Kendo Mobile
- ...

WIG

eDSL to describe JavaScript APIs in F#/WebSharper

Creating new extensions

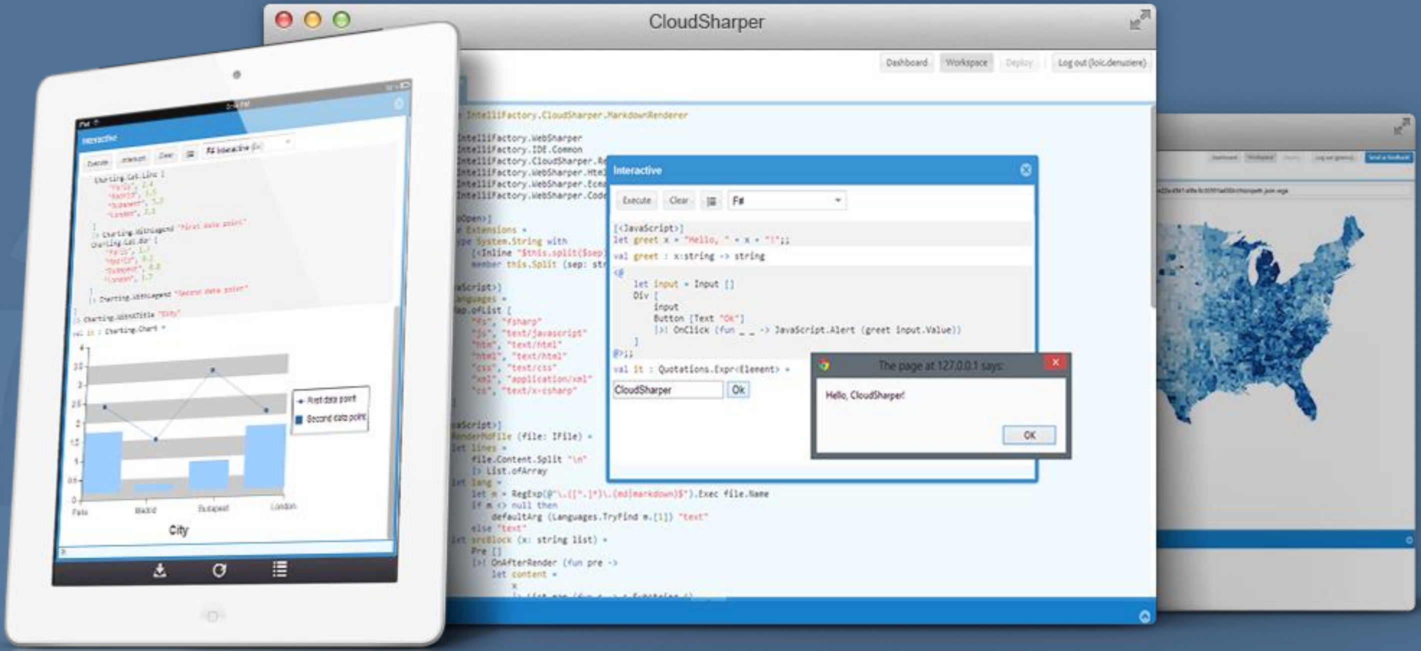
You can implement your own extension:

- manually (via JavaScript inlines)
- using WIG
- importing TypeScript declarations

Where do you go next?

CloudSharper – an online IDE that supports F#

… and much more



```
IntelliFactory.CloudSharper.MarkdownRenderer

IntelliFactory.WebSharper
IntelliFactory.IDE.Common
IntelliFactory.CloudSharper.Re
IntelliFactory.WebSharper.Htel
IntelliFactory.WebSharper.Ecma
IntelliFactory.WebSharper.Code

oOpen>]
 Extensions =
pe System.String with
   [<Inline "$this.split($sep
   member this.Split (sep: str

aScript>]
anguages =
ap.ofList [
   "fs", "fsharp"
   "js", "text/javascript"
   "htm", "text/html"
   "html", "text/html"
   "css", "text/css"
   "xml", "application/xml"
   "cs", "text/x-csharp"
]

aScript>]
RenderMdFile (file: IFile) =
let lines =
    file.Content.Split "\n"
    |> List.ofArray
let lang =
    let m = RegExp(@"\.([^.]*)\.(md|markdown)$").Exec file.Name
    if m <> null then
        defaultArg (Languages.TryFind m.[1]) "text"
    else "text"
let srcBlock (x: string list) =
    Pre []
    |>! OnAfterRender (fun pre ->
        let content =
            x
```

```
[<JavaScript>]
let greet x = "Hello, " + x + "!";;
val greet : x:string -> string
<@
    let input = Input []
    Div [
        input
        Button [Text "Ok"]
        |>! OnClick (fun _ _ -> JavaScript.Alert (greet input.Value))
    ]
@>;;
val it : Quotations.Expr<Element> =
```

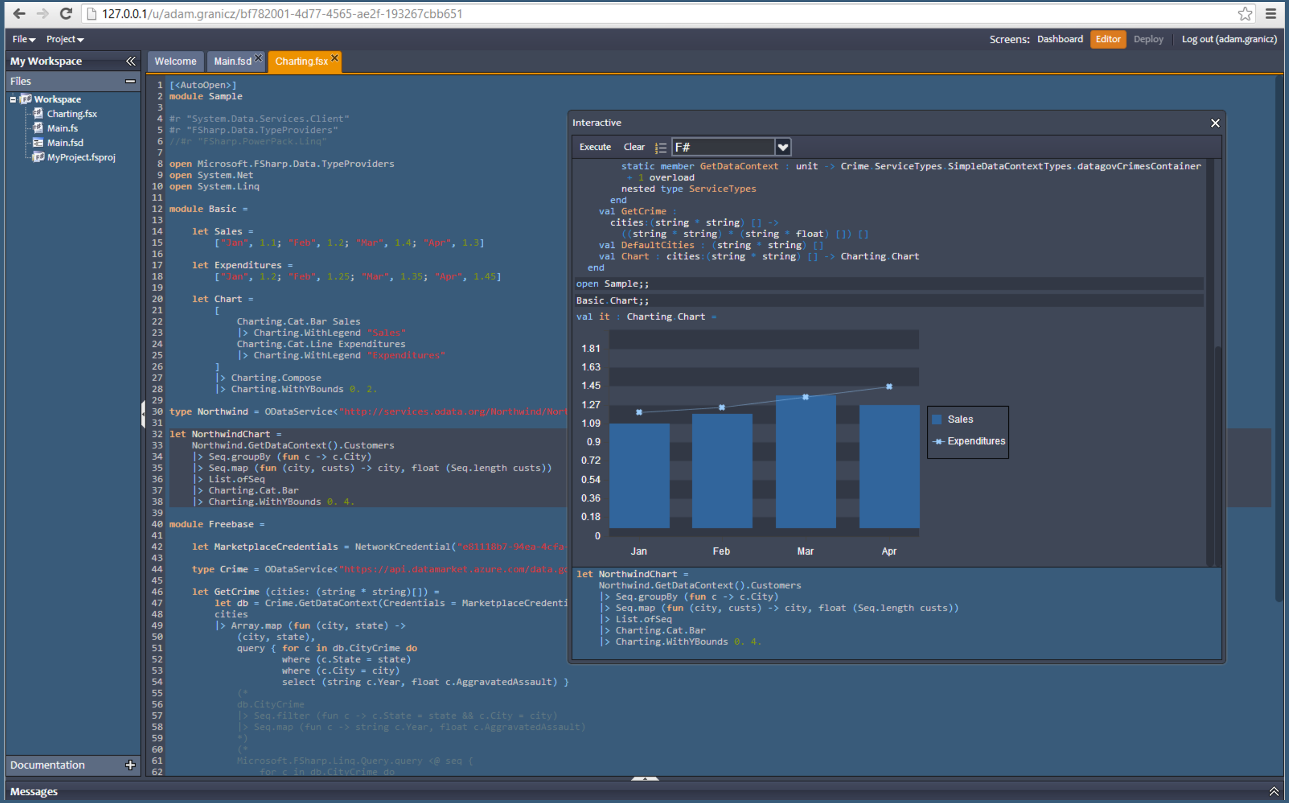
CloudSharper

The page at 127.0.0.1 says:

Hello, CloudSharper!

OK

- Full F# language support
- Multi-project solutions
- **Web and mobile Apps**
- Syntax highlighting
- On the fly type checking
- Interactive exploration
- Integration with data
- Support for type providers



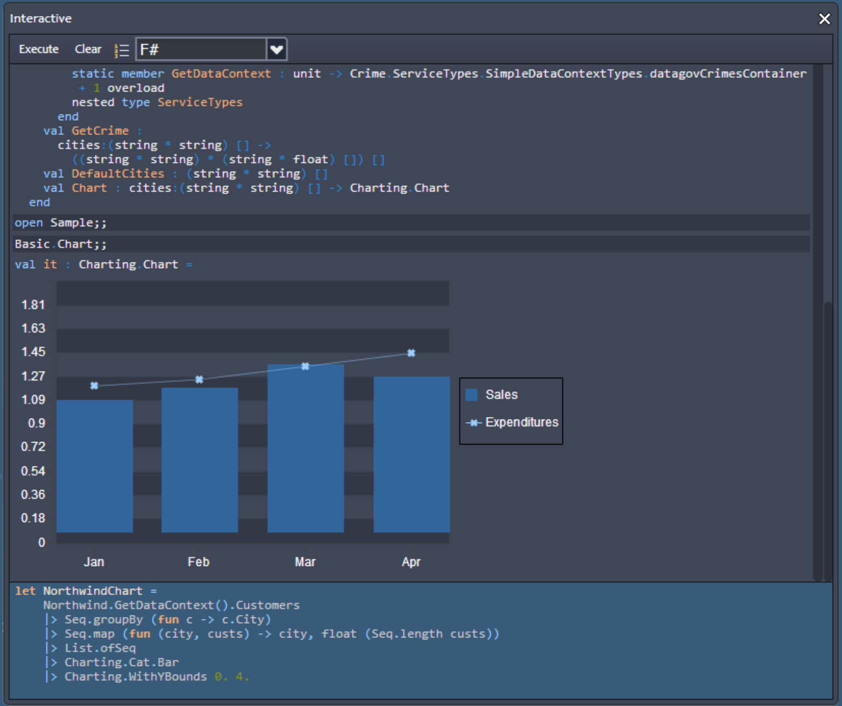
```
[<AutoOpen>]
module Sample

#r "System.Data.Services.Client"
#r "FSharp.Data.TypeProviders"
//#r "FSharp.PowerPack.Linq"

open Microsoft.FSharp.Data.TypeProviders
open System.Net
open System.Linq

module Basic =

    let Sales =
        ["Jan", 1.1; "Feb", 1.2; "Mar", 1.4; "Apr", 1.3]

    let Expenditures =
        ["Jan", 1.2; "Feb", 1.25; "Mar", 1.35; "Apr", 1.45]

    let Chart =
        [
            Charting.Cat.Bar Sales
            |> Charting.WithLegend "Sales"
            Charting.Cat.Line Expenditures
            |> Charting.WithLegend "Expenditures"
        ]
        |> Charting.Compose
        |> Charting.WithYBounds 0. 2.

type Northwind = ODataService<"http://services.odata.org/Northwind/Nort

let NorthwindChart =
    Northwind.GetDataContext().Customers
    |> Seq.groupBy (fun c -> c.City)
    |> Seq.map (fun (city, custs) -> city, float (Seq.length custs))
    |> List.ofSeq
    |> Charting.Cat.Bar
    |> Charting.WithYBounds 0. 4.

module Freebase =

    let MarketplaceCredentials = NetworkCredential("e81118b7-94ea-4cfa-

    type Crime = ODataService<"https://api.datamarket.azure.com/data.go

    let GetCrime (cities: (string * string)[]) =
        let db = Crime.GetDataContext(Credentials = MarketplaceCredenti
        cities
        |> Array.map (fun (city, state) ->
            (city, state),
            query { for c in db.CityCrime do
                    where (c.State = state)
                    where (c.City = city)
                    select (string c.Year, float c.AggravatedAssault) }
            (*
            db.CityCrime
            |> Seq.filter (fun c -> c.State = state && c.City = city)
            |> Seq.map (fun c -> string c.Year, float c.AggravatedAssault)
            *)
            (*
            Microsoft.FSharp.Linq.Query.query <@ seq {
                for c in db.CityCrime do
```

```
        static member GetDataContext : unit -> Crime.ServiceTypes.SimpleDataContextTypes.datagovCrimesContainer
         + 1 overload
        nested type ServiceTypes
      end
    val GetCrime :
      cities:(string * string) [] ->
        ((string * string) * (string * float) []) []
    val DefaultCities : (string * string) []
    val Chart : cities:(string * string) [] -> Charting.Chart
  end
open Sample;;
Basic.Chart;;
val it : Charting.Chart =
```

```
let NorthwindChart =
    Northwind.GetDataContext().Customers
    |> Seq.groupBy (fun c -> c.City)
    |> Seq.map (fun (city, custs) -> city, float (Seq.length custs))
    |> List.ofSeq
    |> Charting.Cat.Bar
    |> Charting.WithYBounds 0. 4.
```

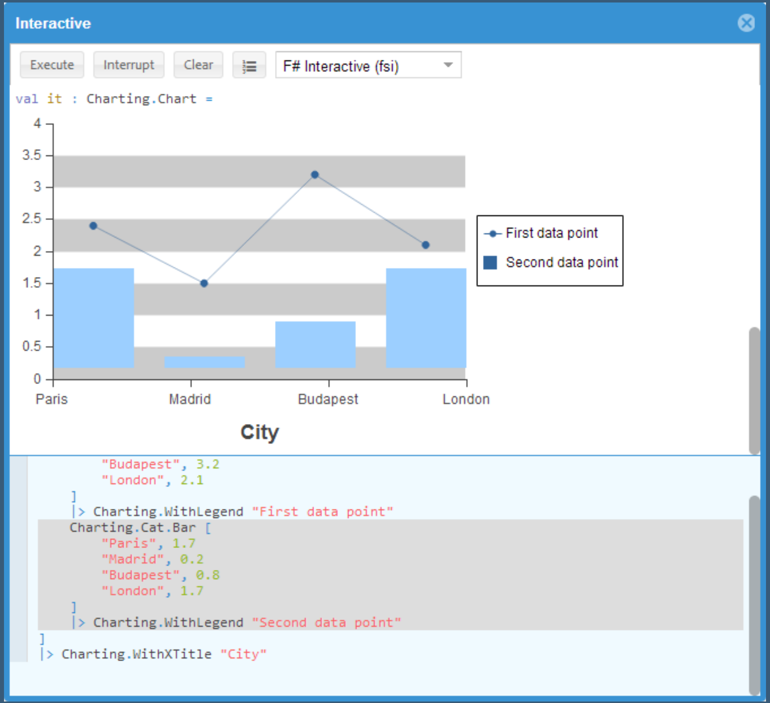
CloudSharper - The online development environment

Develop, deploy, and run full web and mobile solutions, with interactive programming.



```
        static member GetDataContext : unit -> Crime.ServiceTypes.SimpleDataContextTypes.datagovCrimesContainer
         + 1 overload
        nested type ServiceTypes
      end
    val GetCrime :
      cities:(string * string) [] ->
        ((string * string) * (string * float) []) []
    val DefaultCities : (string * string) []
    val Chart : cities:(string * string) [] -> Charting.Chart
  end
open Sample;;
Basic.Chart;;
val it : Charting.Chart =
```

```
let NorthwindChart =
    Northwind.GetDataContext().Customers
    |> Seq.groupBy (fun c -> c.City)
    |> Seq.map (fun (city, custs) -> city, float (Seq.length custs))
    |> List.ofSeq
    |> Charting.Cat.Bar
    |> Charting.WithYBounds 0. 4.
```



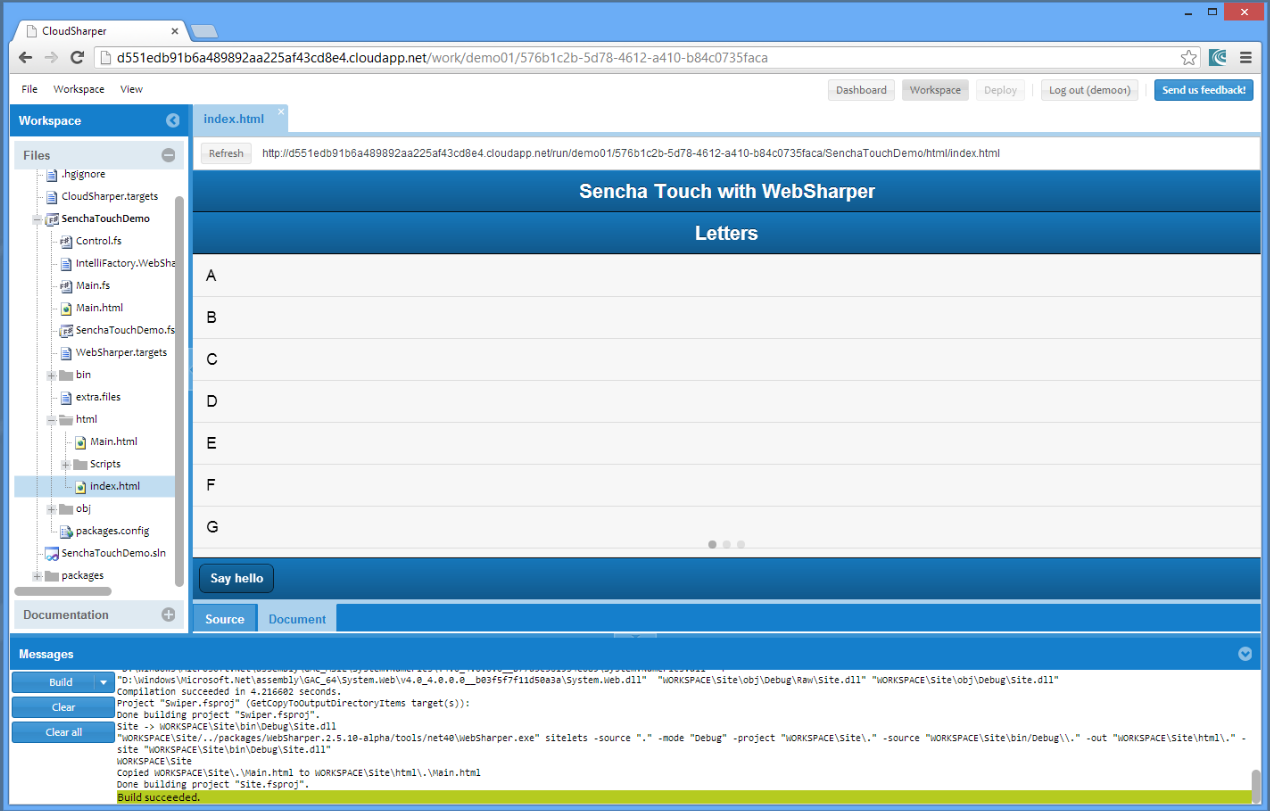
```
val it : Charting.Chart =
```

```
            "Budapest", 3.2
            "London", 2.1
        ]
        |> Charting.WithLegend "First data point"
        Charting.Cat.Bar [
            "Paris", 1.7
            "Madrid", 0.2
            "Budapest", 0.8
            "London", 1.7
        ]
        |> Charting.WithLegend "Second data point"
    ]
    |> Charting.WithXTitle "City"
```

Interactive running a simple data visualization

Using built-in charting, visualizing various data sets is simple and convenient.

A simple Sencha Touch application



```
"D:\Windows\Microsoft.Net\assembly\GAC_64\System.Web\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Web.dll"  "WORKSPACE\Site\obj\Debug\Raw\Site.dll" "WORKSPACE\Site\obj\Debug\Site.dll"
Compilation succeeded in 4.216602 seconds.
Project "Swiper.fsproj" (GetCopyToOutputDirectoryItems target(s)):
Done building project "Swiper.fsproj".
Site -> WORKSPACE\Site\bin\Debug\Site.dll
"WORKSPACE\Site/../packages/WebSharper.2.5.10-alpha/tools/net40\WebSharper.exe" sitelets -source "." -mode "Debug" -project "WORKSPACE\Site\." -source "WORKSPACE\Site\bin/Debug\\." -out "WORKSPACE\Site\html\." -
site "WORKSPACE\Site\bin\Debug\Site.dll"
WORKSPACE\Site
Copied WORKSPACE\Site\.\Main.html to WORKSPACE\Site\html\.\Main.html
Done building project "Site.fsproj".
Build succeeded.
```

FPish – a community for functional programmers

http://fpish.net

FPish – http://fpish.net

Aggregates and catalogs FP content about:

Q&A

Events/Conferences

Courses

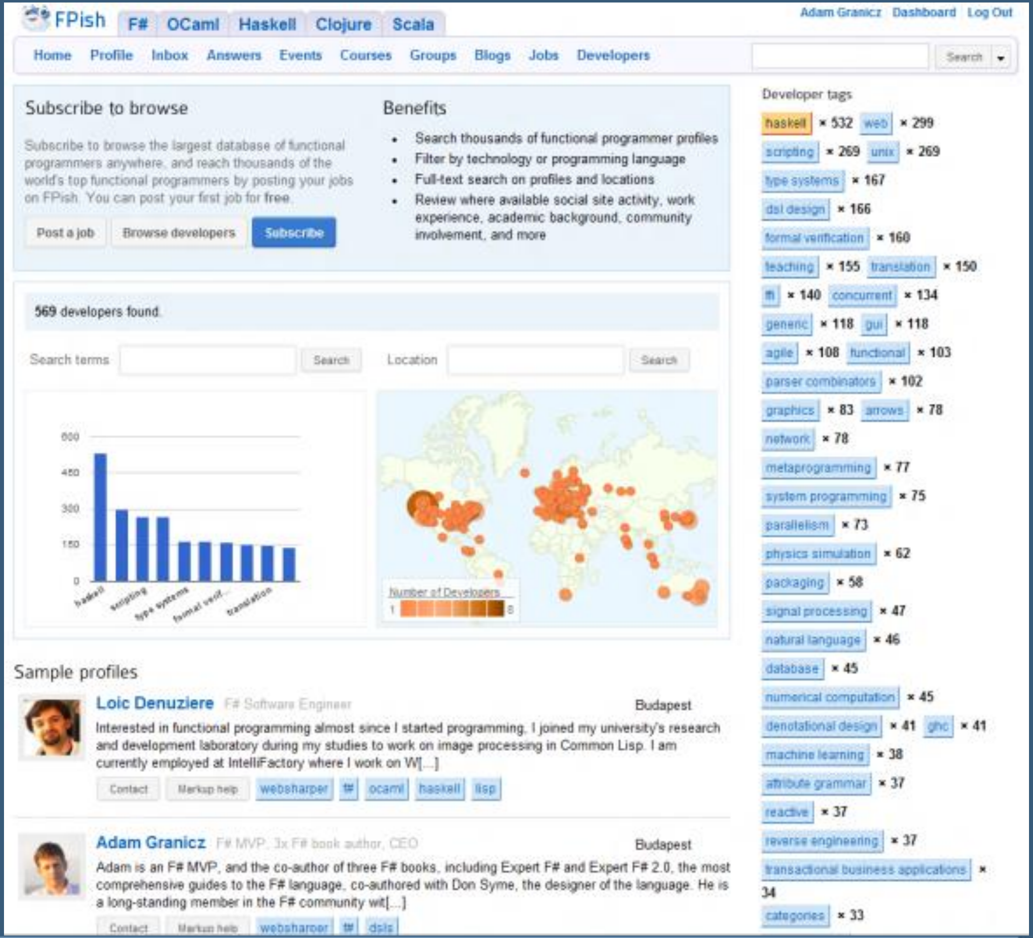
User Groups

Blogs

Jobs

Developers

etc...



Subscribe to browse

Subscribe to browse the largest database of functional programmers anywhere, and reach thousands of the world's top functional programmers by posting your jobs on FPish. You can post your first job for free.

Benefits

- Search thousands of functional programmer profiles
- Filter by technology or programming language
- Full-text search on profiles and locations
- Review where available social site activity, work experience, academic background, community involvement, and more

569 developers found.

Sample profiles

Loic Denuziere F# Software Engineer — Budapest

Interested in functional programming almost since I started programming. I joined my university's research and development laboratory during my studies to work on image processing in Common Lisp. I am currently employed at IntelliFactory where I work on W[...]

Adam Granicz F# MVP, 3x F# book author, CEO — Budapest

Adam is an F# MVP, and the co-author of three F# books, including Expert F# and Expert F# 2.0, the most comprehensive guides to the F# language, co-authored with Don Syme, the designer of the language. He is a long-standing member in the F# community wit[...]

Developer tags

Tag	Count
haskell	× 532
web	× 299
scripting	× 269
unix	× 269
type systems	× 167
dsl design	× 166
formal verification	× 160
teaching	× 155
translation	× 150
ffi	× 140
concurrent	× 134
generic	× 118
gui	× 118
agile	× 108
functional	× 103
parser combinators	× 102
graphics	× 83
arrows	× 78
network	× 78
metaprogramming	× 77
system programming	× 75
parallelism	× 73
physics simulation	× 62
packaging	× 58
signal processing	× 47
natural language	× 46
database	× 45
numerical computation	× 45
denotational design	× 41
ghc	× 41
machine learning	× 38
attribute grammar	× 37
reactive	× 37
reverse engineering	× 37
transactional business applications	× 34
categories	× 33

FsBlogger – a markdown-driven blog engine

http://fsblogger.com

F# Blogger

Posts

Previous Next

June 2015

Fear and Loathing with APL

WebSharper 3.2 with support for scriptable applications, better resource management, and additional streamlined syntax

CraftConf 15–Takeaways from “Jepsen IV: Hope Springs Eternal”

F# Weekly #23, 2015

CodeMotion 15–Takeaways from “Measuring micro-services”

Backwards compatibility is (still) hard

For a Few Dollars More

F# Weekly #22, 2015

Joy of Coding experience report

A Fistful of Dollars

May 2015

A consistent approach to track correlation IDs through microservices

CraftConf 15–Takeaways from “Architecture Without an End State”

F# Weekly #21, 2015

WebSharper 3.1 published

Save the day with a Visual Basic, C#, or F# T-Shirt!

The Detail-Oriented Nature Of Software Development

CloudSharper 0.9.31 published

Understanding homoiconicity through Clojure macros

Simplifying (?) State with C# Async

Null-checking considerations in F# - it's harder than you think

WebSharper 3.2 with support for scriptable applications, better resource management, and additional streamlined syntax

by adam.granicz 6/9/2015, 9:45:00 PM

We are thrilled to announce the availability of WebSharper 3.2, paving the road to further upcoming enhancements to streamline developing and deploying WebSharper apps, and also shipping several key changes summarized here.

No need to annotate sitelet assemblies with `Website`

This is what pre-3.2 code looked like:

```
module Site =
    ...
    let Main =
        Sitelet.Sum [
            Sitelet.Content "/" Home HomePage
            Sitelet.Content "/About" About AboutPage
        ]

[<Sealed>]
type Website() =
    interface IWebsite<Action> with
        member this.Sitelet = Site.Main
        member this.Actions = []

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

Now you can simply do:

```
module Site =
    ...
    [<Website>]
    let Main =
        Sitelet.Sum [
            Sitelet.Content "/" Home HomePage
            Sitelet.Content "/About" About AboutPage
        ]
```

Old code works as before, but we now look for the `Website` attribute on values as well if no assembly-level instance is found, yielding the shorter syntax above.

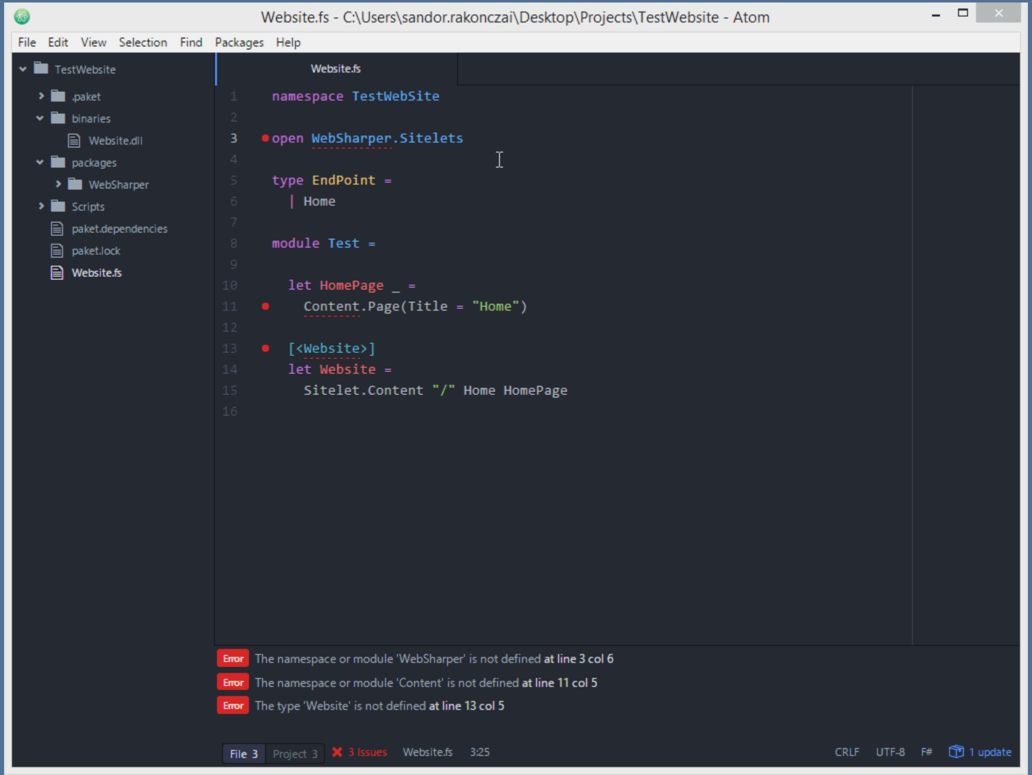
Dot-syntax for chained event handlers

The following code:

```
Button [Text "some text"]
|>! OnClick (fun e args ->
    JS.Alert "Clicked"
)
```

can now be written as:

Atom integration for WebSharper



```
namespace TestWebSite

open WebSharper.Sitelets

type EndPoint =
  | Home

module Test =

  let HomePage _ =
    Content.Page(Title = "Home")

  [<Website>]
  let Website =
    Sitelet.Content "/" Home HomePage

```

Error The namespace or module 'WebSharper' is not defined at line 3 col 6

Error The namespace or module 'Content' is not defined at line 11 col 5

Error The type 'Website' is not defined at line 13 col 5

Thanks for your attention

QUESTIONS?

Get in touch

@websharper
@trywebsharper
@cloudsharper
@granicz

http://websharper.com,
http://try.websharper.com
http://intellifactory.com
http://cloudsharper.com