Client-based Web Application Development in F#

Adam Granicz, CEO
IntelliFactory

Hungarian Web Conference – 2009
Budapest, Hungary

Agenda

Issues with web development

Introducing F# and functional programming

WebSharper™

Conclusions

About IntelliFactory

Issues with web development

1. Applications are **server-centric** – the server performs too much:

    Rendering presentation

    Responding to most events from the client

    Business logic / back-end

→Server becomes the bottleneck → Scalability issues

→Applications are inherently slow to respond, with little interactivity

→<u>Need to move away from the server-centric view</u>

→Applications become client-based

→Applications **execute on client with asynchronous server communication**

Issues with web development

2. Programming models and patterns fall short

```
<asp:Button
        ID="AddContentButton"
        runat="server"

        OnClick="AddContentButton_Click"

        CssClass="button"
        Text="Add More Content" />
```

Issues with web development

3. Too many languages to understand and work with:

C#, VB, Java, etc. – business logic

HTML, CSS – presentation

JavaScript, XML – client-side interaction

SQL, MDX – database

→Need ways to generate one language from another (… to SQL, … to JavaScript)

4. Inability to check client-side code before execution (HTML, JavaScript)

→ Need to have a type-safe source language to translate from

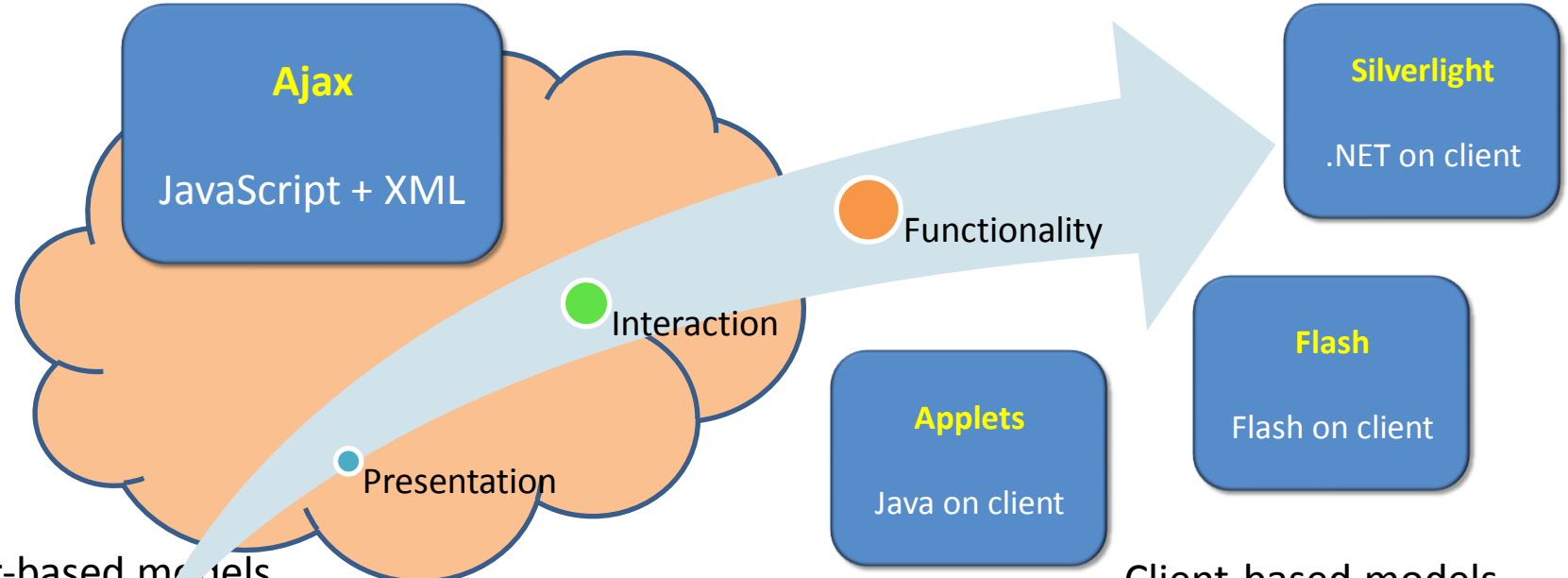
Moving functionality to the client

Ajax in browser

Requires JavaScript

Extending browser capabilities

Silverlight, Flash, applets, etc.



Server-based models
(ASP.NET, PHP, JSF, etc.)

Client-based models
(Silverlight, Flash, applets)

RIA via Ajax

Developing JavaScript code is problematic because

many JavaScript errors can only be detected at run time

How do we program client-based Ajax web applications?

Ideally, we need to:

1. **separate client and server** concerns
2. provide **seamless communication** between client and server
3. move to a **more robust and effective programming model**

Introducing F#

- Is a functional programming language developed by Microsoft
- Is an ideal vehicle 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**

Why F#?

Key benefits:

- Application code is **considerably shorter** than in C#, Visual Basic 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 heavy numerical and symbolic computation
- Language support for developing **distributed, parallel, asynchronous and reactive applications**

Why F#?

Functional programming

- Higher-order functions
- Parametric polymorphism
- Type inference
- Functional data structures
- Pattern matching
- Lazy vs eager evaluation

Programming productivity

- Visual Studio integration
- IntelliSense™

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

Why F#? – Active patterns

Converting

```
let (|NiceString|) s =
    if s |> String.IsNullOrEmpty then
        ""
    else
        s.Trim()

let _ =
    let (NiceString str) = "  with whitespace "
    str
    |> printf "Result = [%s]\n"
```

Why F#? – Active patterns

Partitioning

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

Describing

```
let (|Prime|_|) (n: int) =
    { 2 .. n |> float |> Math.Sqrt |> Math.Floor |> int }
    |> Seq.exists (fun i -> n % i = 0)
    |> function
        | true -> None | false -> Some n

let _ =
    match 123327 with
    | Prime _ -> printf "Prime\n"
    | _       -> printf "Not prime\n"
```

Why F#? – Units of measure

```
module UnitsOfMeasure =
    [<Measure>]
    type m

    [<Measure>]
    type s

    let CalculateD (v: float<m/s>) (t: float<s>) = v*t
    let CalculateV (d: float<m>) (t: float<s>) = d/t

    let _ =
        let d = 100.<m>
        let t = 8.95<s>
        let v = CalculateV d t
        let d' = CalculateD v t
        ...
```

Why F#?

Quotations to model embedded DSLs

Allows to express alternative execution mechanisms

Computation expressions to manipulate stateful objects

Build asynchronous computation

WebSharper™

View applications as **primarily client-based**

Applications execute in the client and call the server asynchronously on demand.

Write code in F# - no more ASPX, HTML, JavaScript, etc.

Type-safe, statically checked code that is guaranteed to run
No more runtime errors
Much shorter implementation time and code
Pagelets → compose into larger pagelets/pages
Formlets → compose forms programmatically and take their typed data

Annotate server vs client functionality

Client functions are translated to JavaScript
Pagelet dependencies are managed

WebSharper™ Pagelets

```
module HelloWorld =
    open IntelliFactory.WebSharper
    open IntelliFactory.WebSharper.Html
    open IntelliFactory.WebSharper.Html.Elements
    open IntelliFactory.WebSharper.Html.Attributes
    open IntelliFactory.WebSharper.Jquery

    [<JavaScript>]
    let Main () =
        let id = UniqueId "__"
        Div [
            P      [ Id id; Text "Welcome!" ]
            Button [ Text "Click me!" ]
              |> OnClick
                  (fun e ->
                      e.PreventDefault()
                      JQuery.[ById(id)].Text("Hello, world!“) |> ignore)
        ]
```

WebSharper™ Pagelets

Integrating with ASP.NET

F# type for server control

```
type HelloWorld() =
    inherit IntelliFactory.WebSharper.Web.Control()
    override this.Pagelet = <@ HelloWorld.Main @>
```

```
<pagelet:HelloWorld runat="server" />
```

ASP.NET markup

WebSharper™ Pagelets

```
module HelloWorld =
    open IntelliFactory.WebSharper
    ...

    [<RPC.Callable>]
    let YourServerFunction (...) = ...

    [<JavaScript>]
    let YourClientFunction () =

        let data = YourServerFunction (...)
        ...

    [<JavaScript>]
    let Main () =
        ...
```

Seamless interaction with the server

WebSharper™ Pagelets

Pagelets can define dependencies

```
[<Requires("http://yui.yahooapis.com/2.7.0/build/button/assets/skins/sam/button.css")>]
[<Requires("http://yui.yahooapis.com/2.7.0/build/yahoo-dom-event/yahoo-dom-event.js")>]
[<Requires("http://yui.yahooapis.com/2.7.0/build/element/element-min.js")>]
[<Requires("http://yui.yahooapis.com/2.7.0/build/button/button-min.js")>]
type Button =
  …
```

Dependencies are managed by a WebSharper™ ASP.NET script manager control.

```
<websharper:ScriptManager runat="server" />
```

Similar technologies

Source language -> JavaScript

GWT – Google Web Toolkit
Java to JavaScript

Script# (S#) –
C# to JavaScript, also comes with ExtJs bindings as a separate project

…

XML to JavaScript

xap – Apache Extensible Ajax Platform
XML-based declarative specification -> JavaScript/HTML/CSS
Project is dead

Assembly to JavaScript

jsc – .NET decompiler → JavaScript

WebSharper™

Generates HTML programmatically
Generates optimized JavaScript and manages pagelet dependencies
Provides a uniform event model
Has access to the **entire F# language** including
- Types - algebraic data types, tuples, records, classes/objects
- Pattern matching and active patterns
- Lazy sequences
- Asynchronous computation

Has access to a number of core .NET namespaces
- Providing efficient mappings into JavaScript

Provides unit and functional testing capabilities – test fixtures, exception handling

WebSharper™

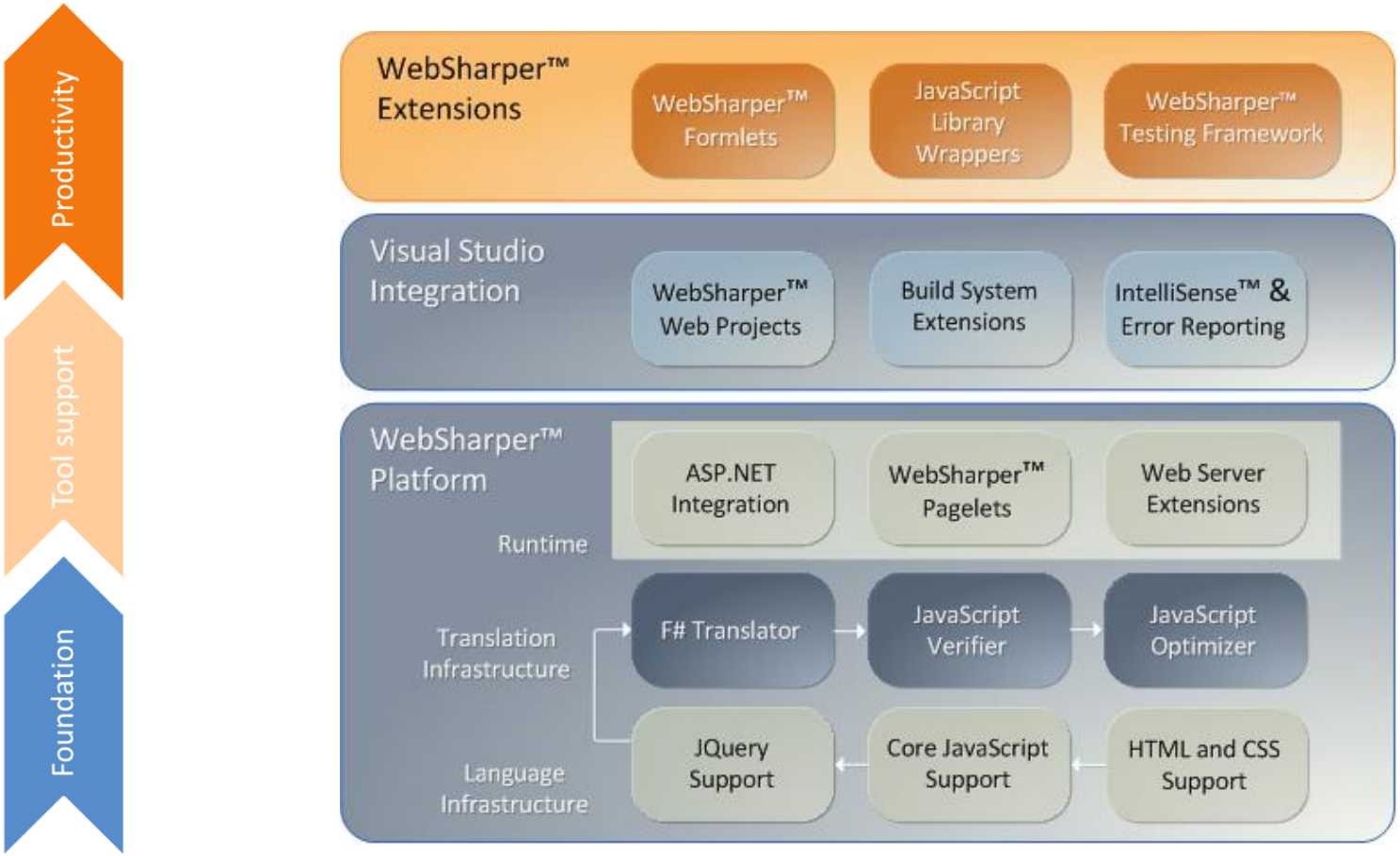
ASP.NET integration – include pagelets into ASPX markup using server tags

Main concepts

Pagelets – represent client-side functionality and presentation
|-- **Formlets** – functional construction of UIs
|-- **Flowlets** – functional construction of UI transitions / flows

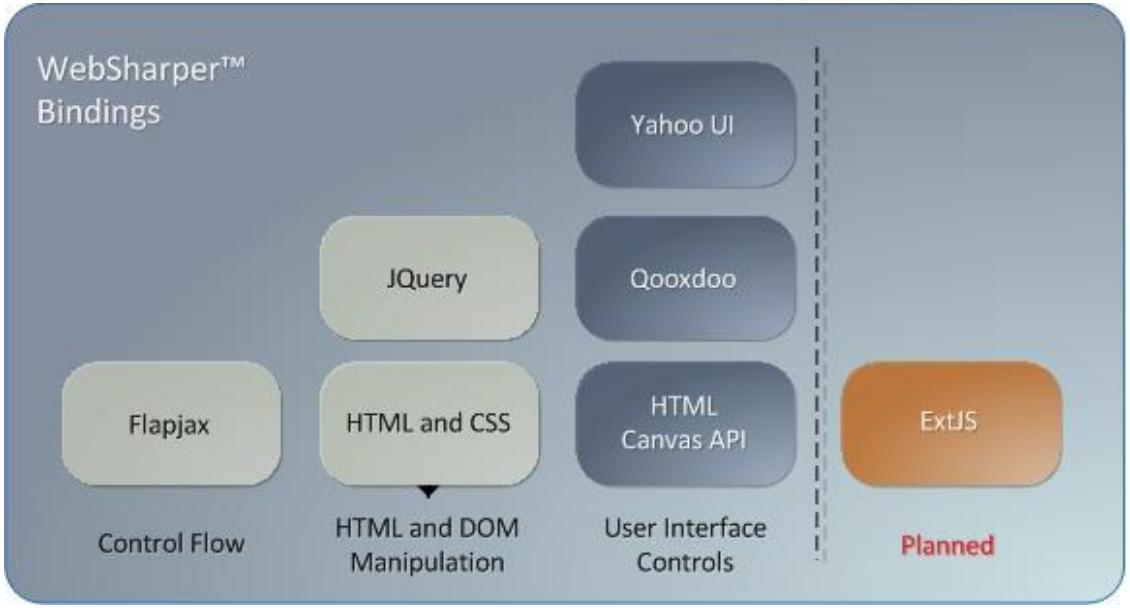
Core foundation –
DOM manipulation (JQuery) and uniform event handling (Flapjax)
Extensions –
UI libraries (Yahoo UI, Qooxdoo, etc.)

The WebSharper™ Platform



The WebSharper™ Platform

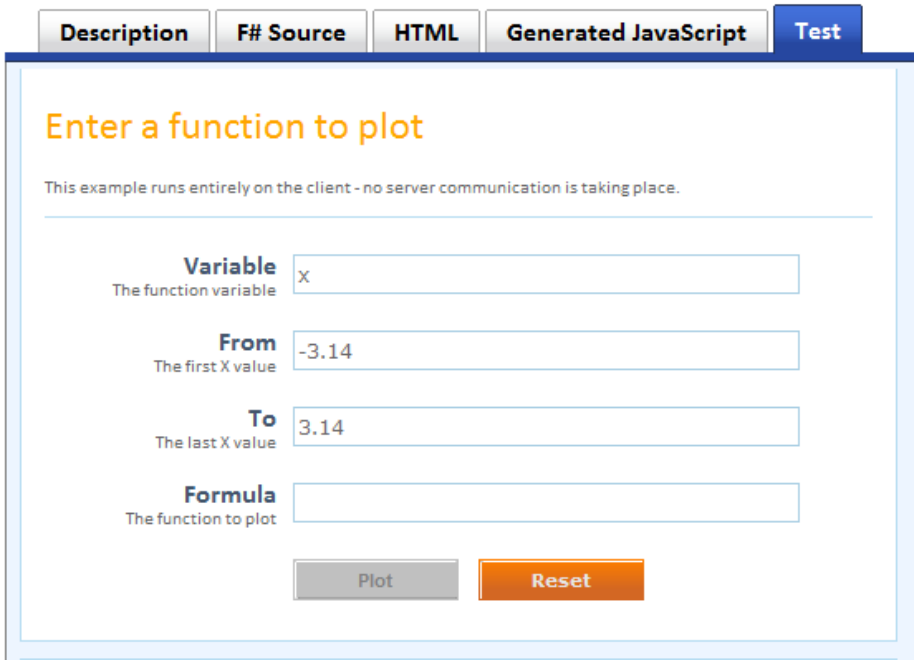
Bindings to third-party JavaScript-based technologies:



The WebSharper™ Platform

Build impressive UIs in mere minutes!

A Yahoo UI instantiation →



WebSharper™ Formlets

```
type PlotInfo =
    { Variable : string
      From     : float
      To       : float
      Formula  : string }

[<JavaScript>]
let PlotFunctionForm () : Formlet<PlotInfo> =

     let compose vVar vFrom vTo vFormula =
         { Variable = vVar
           From     = vFrom |> float
           To       = vTo   |> float
           Formula  = vFormula }

    Pure compose
    <$> (LabelWithDescription "Variable" "The function variable" (Input "x") |> SatNotEmpty "")
    <$> (LabelWithDescription "From"     "The first X value" (Input "-3.14") |> SatNotEmpty "")
    <$> (LabelWithDescription "To"       "The last X value"   (Input "3.14") |> SatNotEmpty "")
    <$> (LabelWithDescription "Formula"  "The function to plot"   (Input "") |> SatNotEmpty "")
```

WebSharper™ Flowlets

Flowlets allow to compose formlets in a **type-safe** way into a **sequence of user interface interactions**, including

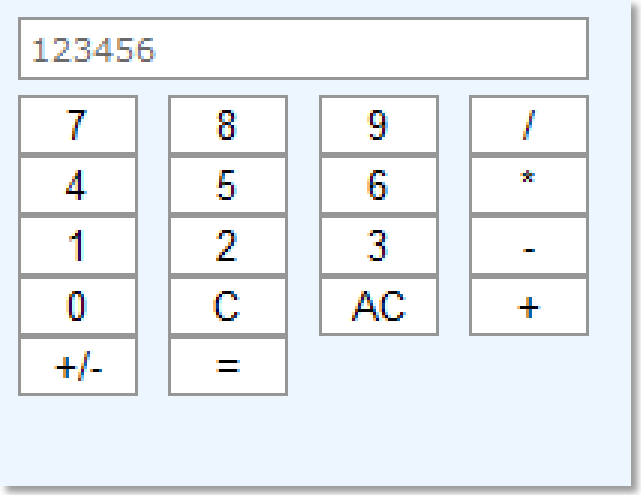
sub-form transformation (morphing a form to a new shape)
form transitions (replacing a form with another)

Flowlets are soon available as an extension to the WebSharper™ platform.

WebSharper™ Example

Building a simple calculator application

… in less than **100 lines of F#**
… including comments



http://www.websharper.com/products/wsp/Tutorial.aspx

WebSharper™ Example - Calculator

```
namespace IntelliFactory.WebSharper.Samples

module Calculator =
    open IntelliFactory.WebSharper
    open IntelliFactory.WebSharper.Html
    open IntelliFactory.WebSharper.Html.Elements
    open IntelliFactory.WebSharper.Html.Attributes
    open IntelliFactory.WebSharper.JQuery

    [<JavaScript>]
    let Main ()  =
```

WebSharper™ Example - Calculator

The state of the calculator

```
let onum, num, op =
    ref 0, ref 0, ref None
```

The calculator display

```
let displayId = UniqueId "display"
let display   = Input [Id displayId; Value "0"; Style "width: 182px"]

let updateDisplay () =
    JQuery.[ById displayId].Val(string !num) |> ignore
```

WebSharper™ Example - Calculator

```
let D n =
    num := 10 * !num + n
    updateDisplay ()

let C () =
    num := 0
    updateDisplay()

let AC () =
    num  := 0
    onum := 0
    op   := None
    updateDisplay ()

let N () =
    num := - !num
    updateDisplay ()
```

```
let E () =
    match !op with
    | None ->
        ()
    | Some f ->
        num := f !onum !num
        op  := None
        updateDisplay ()

let O o () =
    match !op with
    | None ->
        ()
    | Some f ->
        num := f !onum !num
        updateDisplay ()
    onum := !num
    num  := 0
    op   := Some o
```

WebSharper™ Example - Calculator

What remains is providing the user interface. To make it easier, define some helpers for creating buttons (as DOM nodes) and attaching click actions.

```
let btn caption action =
    Button [Text caption]
    |> InitWith (fun j -> j.Width(40))
    |> OnClick
        (
            fun e ->
                e.PreventDefault()
                action ()
        )

let digit n =
    btn (string n) (fun () -> D n)
```

WebSharper™ Example - Calculator

You can now easily compose the calculator from the components defined above:

```
Div [
    display
    Br []
    Div [
        digit 7; digit 8; digit 9; btn "/" (O ( / ))
        Br []
        digit 4; digit 5; digit 6; btn "*" (O ( * ))
        Br []
        digit 1; digit 2; digit 3; btn "-" (O ( - ))
        Br []
        digit 0; btn "C" C; btn "AC" AC; btn "+" (O ( + ));
        Br []
        btn "+/-" N; btn "=" E
    ]
]
```

More WebSharper™ Examples

Type a name in the input field below:

First Name: A

Suggestions: *Adam Anton*

Auto-completion: 70 lines

A draft is saved every 5 seconds or when the save button is clicked.

Last Saved: 10/3/2009 1:45:26 AM

Hi, I am writing this very long letter....

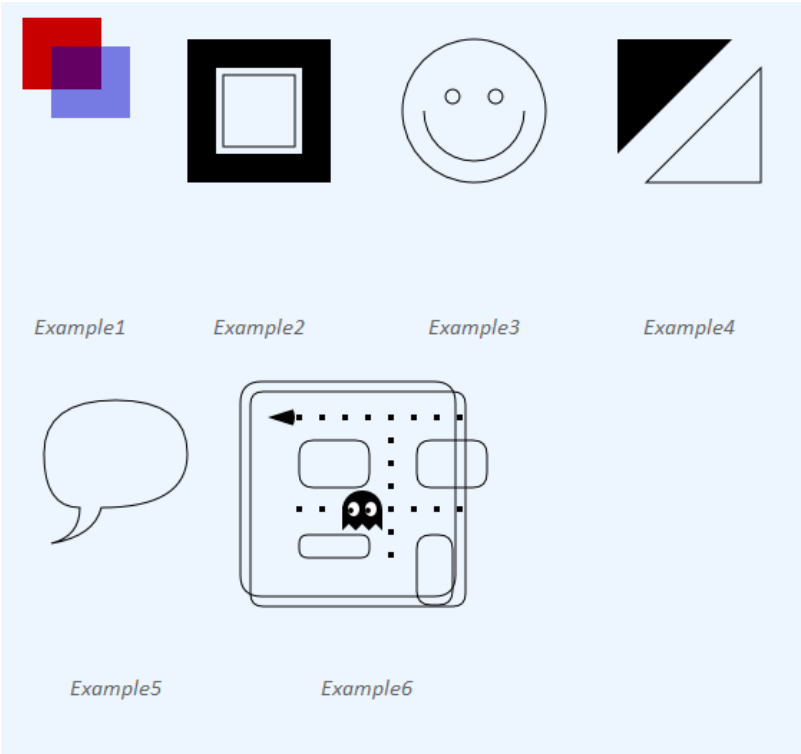
Save Now

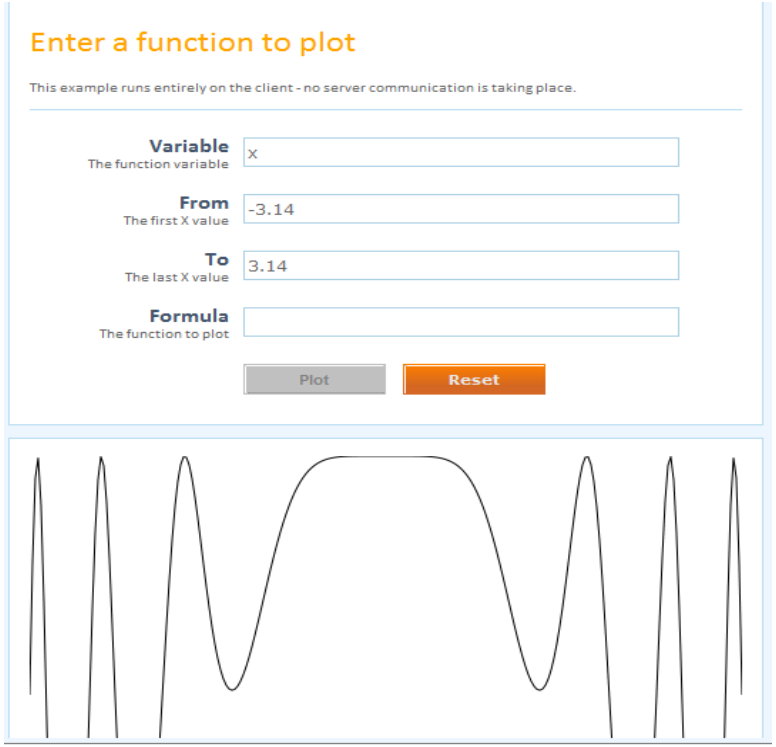
The currently saved draft is:

Hi, I am writing this very

Saving drafts: 50 lines

More WebSharper™ Examples

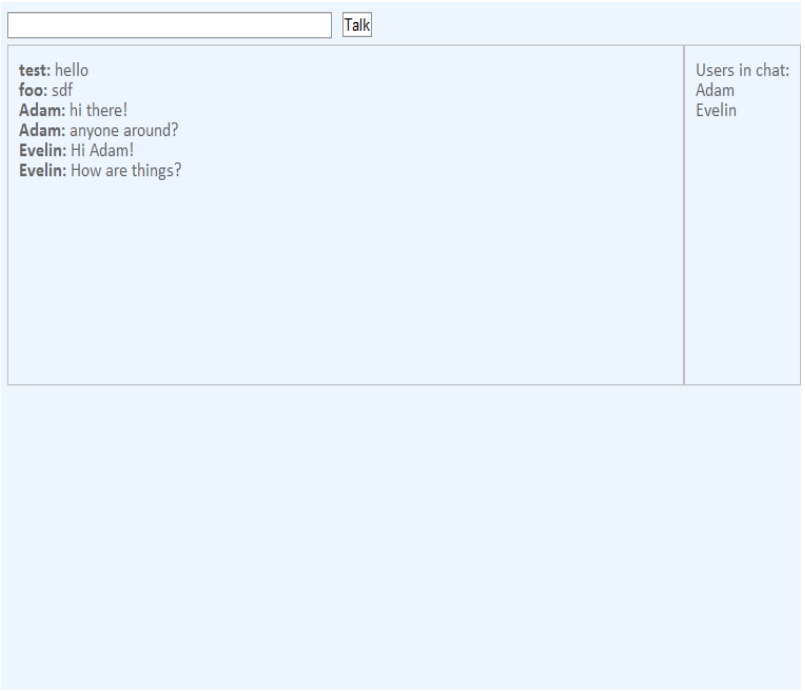


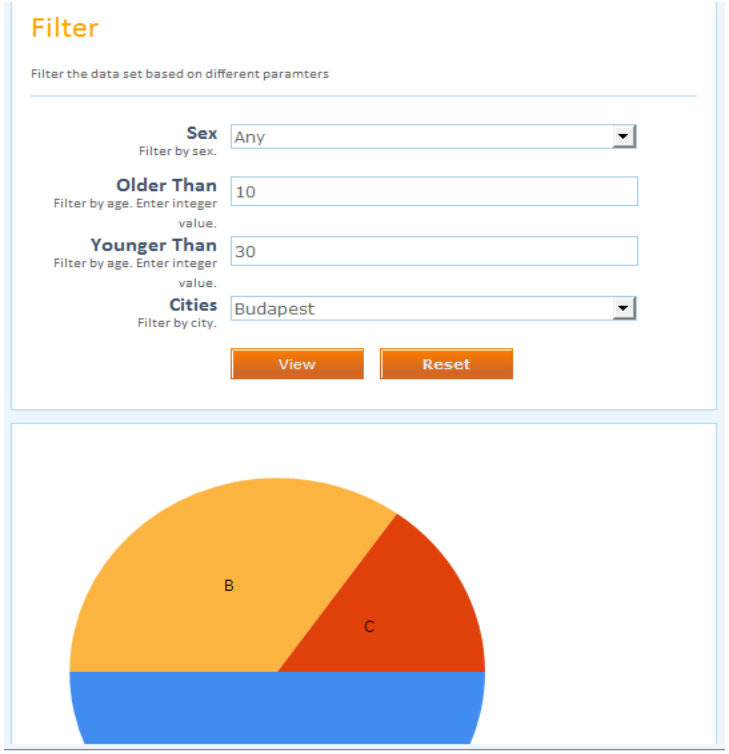


Drawing on a Canvas: 150 lines

Plotting functions: 350 lines

More WebSharper™ Examples





Live Chat: 280 lines

Charting: 160 lines

Conclusions

ASP.NET as a technology **benefits little** from F# and functional programming
→ a better, different, and functional approach is needed

WebSharper™ solves a number of critical problems that hinder effective web development. WebSharper™

Allows to quickly prototype web applications in F#
Interoperates with core F# and .NET libraries seamlessly
Offers a uniform developer story (write F# code for your entire app)
Eliminates the need to write JavaScript code by hand

And at the same time, WebSharper™ applications are

stunningly short and concise
easy to develop and maintain

About IntelliFactory

At **IntelliFactory**, we specialize in:

- **Building robust .NET applications** in F#
- **Migrating** to and **extending legacy .NET applications** in F#
- Customer-friendly, **agile management** of software development projects
- **F# trainings**, from basic to advanced, from individual to enterprise-wide
- Designing and implementing **domain-specific languages**
- Building tools for **functional web application development**

IntelliFactory – in a nutshell

At **IntelliFactory**, we firmly believe in:

- **Expertise**: constantly seeking to push the limits and apply FP to the fullest
- **Diversity**: we bring talents from all around the world; currently we have staff from Hungary, USA, Sweden, Ukraine, Colombia.
- Solid **academic** and **FP professional background**
- **Bridging academia and industry -**
  - Interns – EPFL, Caltech, EPITA, Eafit
  - Sponsorship – Central European Summer School in FP (CEFP 2009)
  - Industry partners – Microsoft, local and multi-national firms
- A **challenging place to work at** – but with lots of freedom

http://www.websharper.com/products/wsp/Tutorial.aspx

Google

Getting Started

- Main Page
- News
- Download
- How It Works
- FAQ
- Demos
- Supported Libraries
- API Reference
- Licensing

Demos

- Hello World
- Factorial
- Calculator
- Toggle Panel
- Autocomplete
- Reactive Programming
- HTML5 Drawing
- HTML5 Canvas
- Remoting
- Live Chat
- Charting
- **Plot functions**

Support

- Submit a Bug
- Ask a Question

Screenshots

Plot functions

Description | F# Source | HTML | Generated JavaScript | Test

Enter a function to plot

This example runs entirely on the client - no server communication is taking place.

Variable The function variable: x

From The first X value: -3.14

To The last X value: 3.14

Formula The function to plot:

Plot | Reset

Available at:
http://www.websharper.com