F# and SignalR for a FastWeb

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

— Edsger Dijkstra

Riccardo Terrell

Agenda

- Why F# on the Web
- F# Magic
- SignalR and F#
- CQRS and F#
- Code & Code

Something about me – Riccardo Terrell

- Originally from Italy, currently living in USA ~9 years
  - Living/working in Washington DC
- +/- 16 years in professional programming
  - C++/VB → Java → .Net C# → C# & F# → ??
- Organizer of the DC F# User Group
- Software Architect @ Microsoft
- Passionate in Technology, believe in polyglot programming as a mechanism in finding the right tool for the job

Goals - Plan of the talk

F# is a *great and mature language* for Web Development

F# has built in features to develop Fast and Scalable Web App

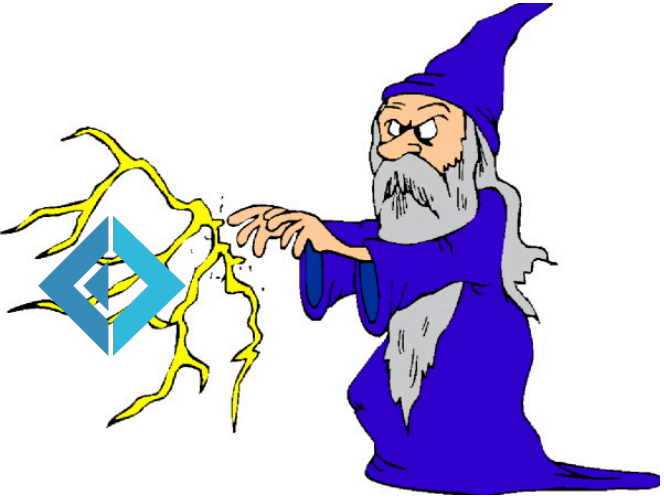
F# |> RX |> SignalR |> CQRS |> 

Technology trends

- Mobile
- Claud computing
- Real Time notification
- Single Page Apps
- Big Data

- Scalable
- Maintanable & Readable
- Reliable & Testable
- Composable & Reusable
- Concurrent

...F# *and Functional Paradigm* can help



F# helps with the mission statement simple code for complex problem…

F# great for scalability and concurrent ready

F# play well together with Web Application

Use function and composition as building block

Why F# on the web? Really?

Is functional programming and F# mostly about algorithms and calculations?

Is functional programming and F# mostly targeted at scientific or financial domains?

Is functional programming and F# just for Server Side computations?

Why F# on the web?

Fast - F# code execution is fast, using native code generation from scripted or project code

Succinct - F# is concise, readable and type-safe, for fast development of robust web solutions

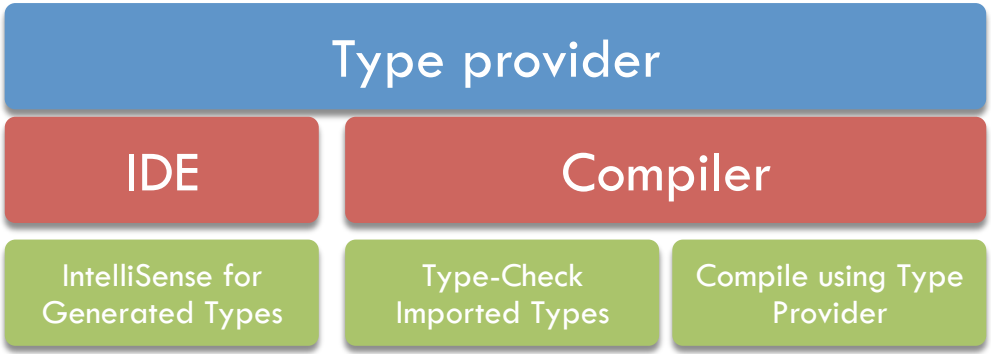
Reactive and Scalable - F# asynchronous programming simplifies scalable, reactive web programming, and Agent too!

Interoperable - F# interoperates seamlessly with languages such as C#, JavaScript and TypeScript, F# is JavaScript-ready through WebSharper and FunScript


Magic!

F# Type Providers on the Web

- JSON Type Provider
- CSV
- WSDL
- WMI
- Data Access – SQL and EF
- Funscript to Javascript
- And more, write your own



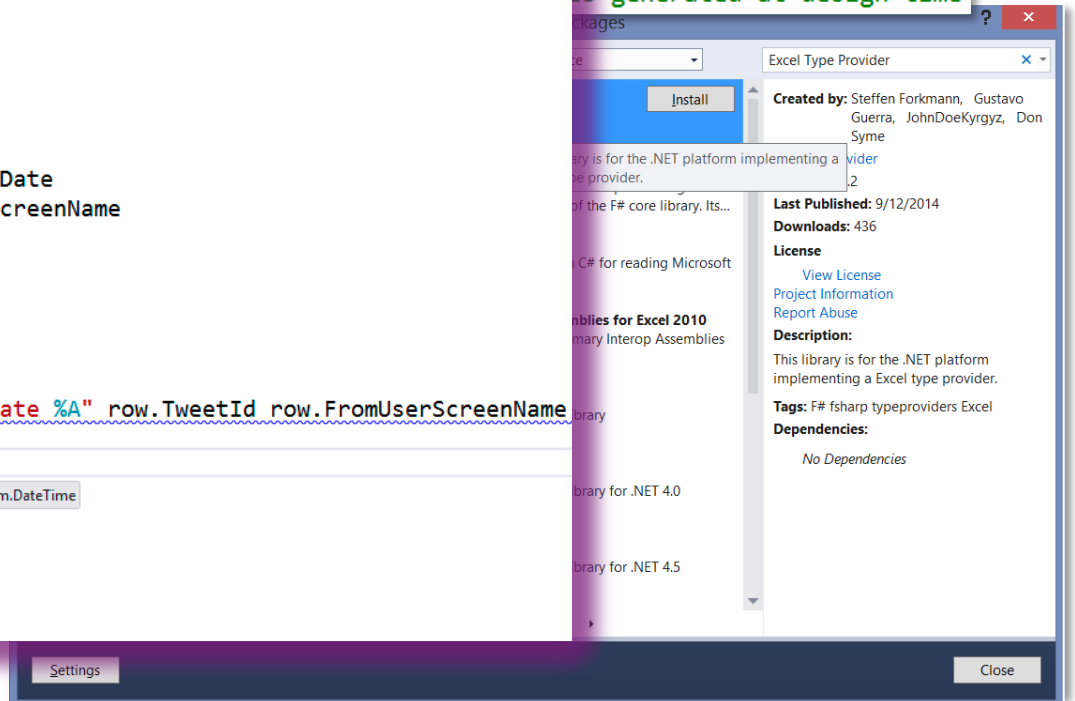
F# Type Providers on the Web


```
// A sampl                                                    is generated at design time
type Cont
```

```
let [<LiteralAttribute>] CsvFilePath = "fsharp_2013-2014.csv"

type FSharpCsv = CsvProvider<CsvFilePath>

let csv = FSharpCsv.Load(CsvFilePath)

let firstRow = csv.Rows |> Seq.head

let ``creatDate is DateTime`` = firstRow.CreatedDate
let ``creatDate is String`` = firstRow.FromUserScreenName
let ``creatDate is String`` = firstRow.Language
let ``creatDate is String`` = firstRow.Source
let ``creatDate is String`` = firstRow.Text
let ``creatDate is Int64`` = firstRow.TweetId

for row in csv.Rows |> Seq.take 10 do
    printfn "TweetId %d\t - Name %s\t - CreatedDate %A" row.TweetId row.FromUserScreenName

    row.
(*
```

CreatedDate
FromUserScreenName
Language
Source
Text
TweetId

property CsvProvider<...>.Row.CreatedDate: System.DateTime



Function C


```
if condition1 then
    doThis()
    if condition2 then
        doThat()
        if condition3 then
            doSomeOtherThing()
            if condition3 then
                aTask()
                if condition4 then
                    anotherTask()
                    if condition5 then
                        yetAnotherTask()
                        if condition5 then
                            finallyDone()
                        else handleError()
                    else handleError()
                else handleError()
            else handleError()
        else handleError()
    else handleError()
else handleError()
```

```
[<Route("buyTicker")]
member this.PostBuyTi
    match base.Mo

let validateTic
    match valid
    | Failure(e
    | Success(s

    | false -> th
```

```
pResponseMessage =

th

rice input with
 Failure e
 Success s

and connectionId

.BadRequest, "Model
```

Function Composition

Railway oriented programming

So we have a lot of these "one input -> Success/Failure output" functions -- how do we

What we want to do is connect the `Success` output of one to the input of the next, but second function in case of a `Failure` output. This diagram gives the general idea:

```
let validateTicker (input : TradingRecord) =
    if input.Symbol = "" then Failure "Ticket must not be blank"
    else Success input

let validateQuantity (input : TradingRecord) =
    if input.Quantity <= 0 || input.Quantity > 50 then Failure "Quantity must be positive
    else Success input

let validatePrice (input : TradingRecord) =
    if input.Price <= 0. then Failure "Price must be positive"
    else Success input
```

Asynchronous Workflows

- *Software is often I/O-bound, it provides notable performance benefits*
  - Connecting to the Database
  - Leveraging web services
  - Working with data on disk
- *Network and disk speeds increasing slower*
- *Not Easy to predict when the operation will complete* (no-deterministic)

Anatomy of Async Workflows

```
let getLength url =
  let wc = new WebClient()
  let  data = wc.DownloadString(url)
  data.Length
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Anatomy of Async Workflows

```
let getLength url = async {
  let wc = new WebClient()
  let  data = wc.DownloadString(url)
  data.Length }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Anatomy of Async Workflows

```
let getLength url = async {
  let wc = new WebClient()
  let  data = wc.DownloadString(url)
  return data.Length }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Anatomy of Async Workflows

```
let getLength url = async {
  let wc = new WebClient()
  let! data = wc.AsyncDownloadString(url)
  return data.Length }
```

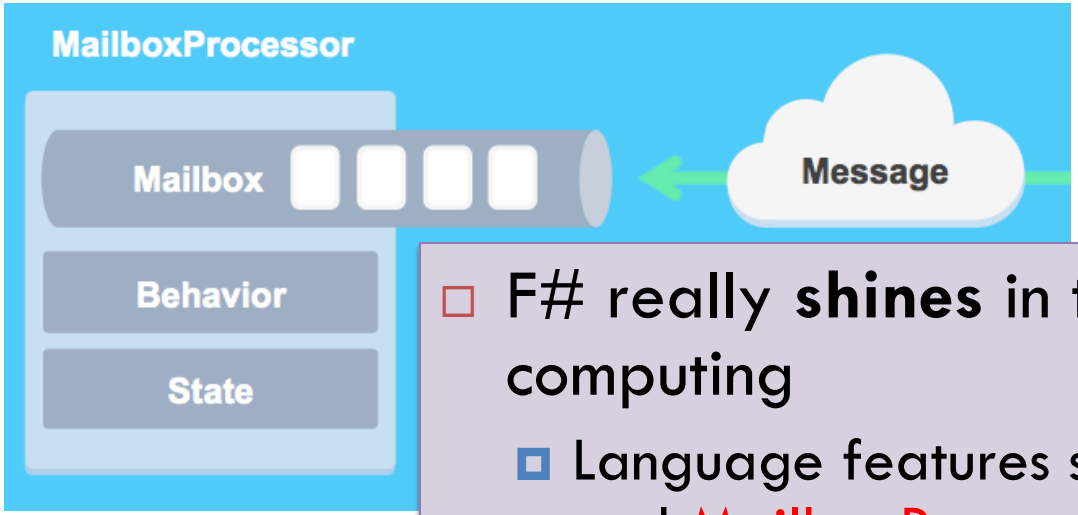
- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Asynchronous Workflows

```
let openFileAsynchronous : Async<unit>
  async { use  fs = new  FileStream(@"C:\Program Files\..., …)
          let  data = Array.create (int fs.Length) 0uy
          let!  bytesRead = fs.AsyncRead(data, 0, data.Length)
          do  printfn "Read Bytes: %i, First bytes were:
          %i %i %i ..." bytesRead data.[1] data.[2] data.[3] }
```

- Async defines a block of code we would like to run asynchronously
- We use let! instead of let
  - let! binds asynchronously, the computation in the async block waits until the let! completes
  - While it is waiting it does not block
  - No program or OS thread is blocked

F# MailboxProcessor – aka Agent



- F# really **shines** in the area of distributed computing
  - Language features such as Async Workflow and MailboxProcessor (a.k.a. *agent*) open the doors for computing that focuses on message passing concurrency
  - Scaling Up & Scaling Out easy to implement

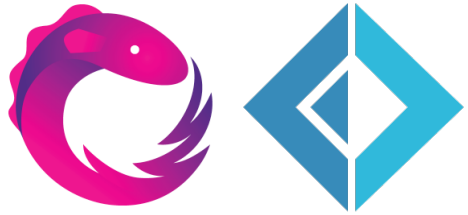
Concurrent Model Programming

```
let agent = MailboxProcessor<Message>.Start(fun inbox ->
        let rec loop n = async {
            let! msg = inbox.Receive()
            match msg with
            | Add(i) -> return! loop (n + i)
            | Get(r) -> r.Reply(n)
                        return! loop n }
        loop 0)
```

IObserver & IObservable

```
let observable = { new IObservable<string> with
                        member x.Subscribe(observer:IObserver<string>) =
                           { new IDisposable with
                                member x.Dispose() = ()
                                }
                           }

let observer = { new IObserver<string> with
                      member x.OnNext(value) = ()
                      member x.OnCompleted() = ()
                      member x.OnError(exn) = () }
```

When to use Rx



When is Rx appropriate?

Rx offers a natural paradigm for dealing with sequences of events. A sequence can contain zero or more events. Rx proves to be most valuable when composing sequences of events.

Should use Rx

Managing events like these is what Rx was built for:

- UI events like mouse move, button click
- Domain events like property changed, collection updated, "Order Filled", "Registration accepted" etc.
- Infrastructure events like from file watcher, system and WMI events
- Integration events like a broadcast from a message bus or a push event from WebSockets API or other low latency middleware like Nirvana
- Integration with a CEP engine like StreamInsight or StreamBase.

http://www.introtorx.com/content/v1.0.10621.0/01_WhyRx.html

F# to JavaScript - FunScript

- **FunScript** is Javascript compiler
  - Write F# client-side code with full intellisense
  - Leverage F# functional goodies that compiles in JS
    - Higher Order functions
    - Pattern Matching
    - Type Inference

```
Globals.document.getElementById("btn")
    .onc
```

oncanplay
oncanplayStream
oncanplaythrough
oncanplaythroughStream
oncellchange
oncellchangeStream
onchange
onchangeStream
onclick

property HTMLElement.

Full functional Data-Structure

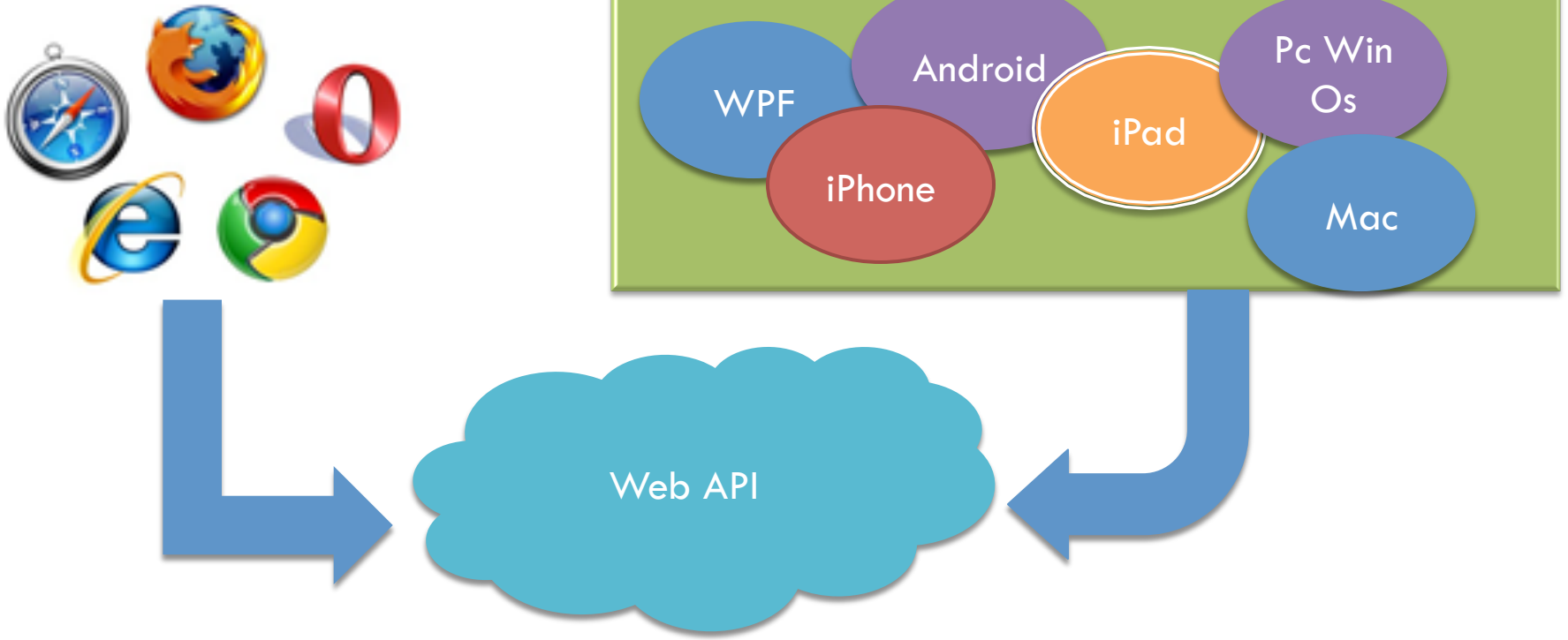
- Records
- Discriminated Union
- Tuples
- List Map Set seq
- .Net mutable Collections Array – Dictionary - List

F# to JavaScript - FunScript

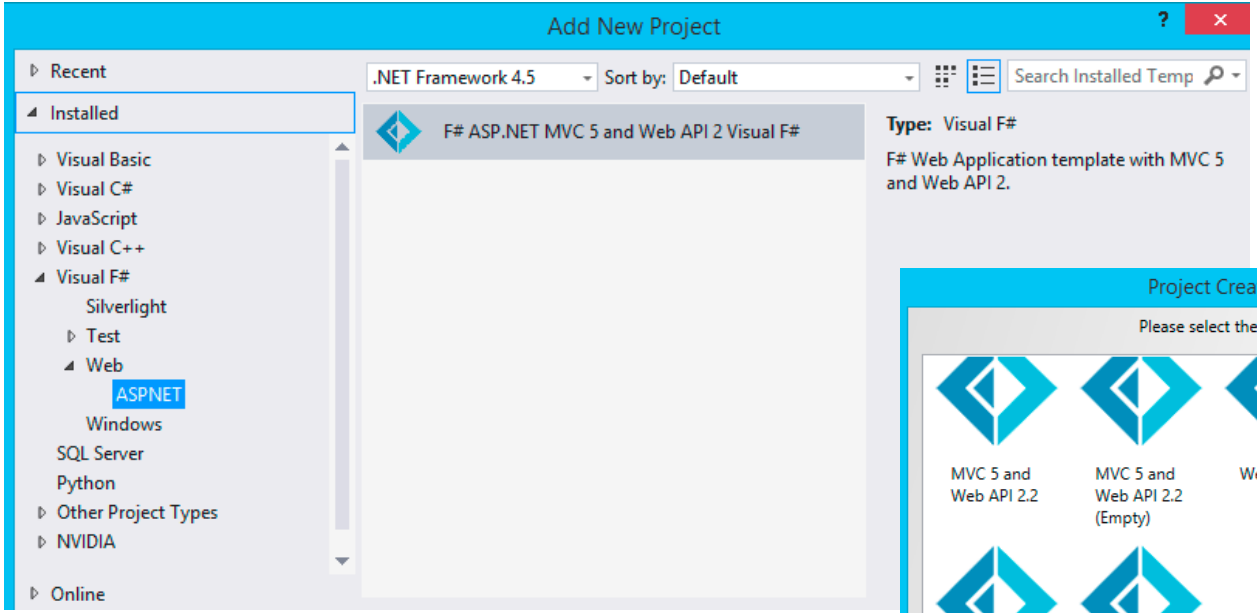
- F# unique Features
  - Async Workflow
  - Reactive Extensions
  - Type Providers
  - Computation Expression

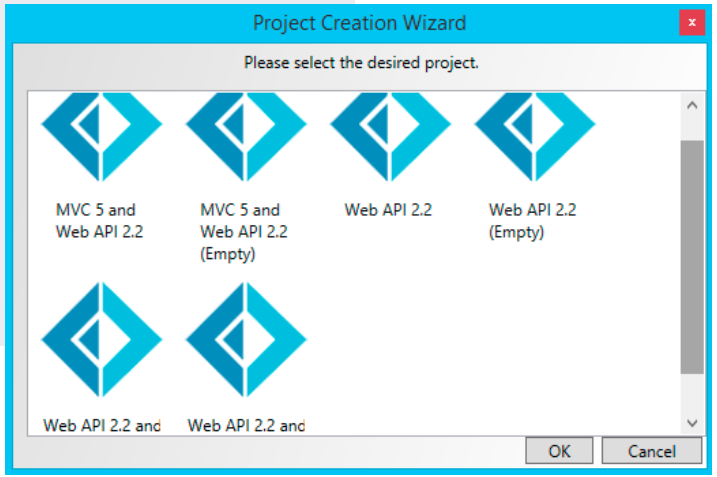
```
let orderProcess (ractive : Ractive) =
    ractive.onStream ("addTicker")
        |> Observable.add (fun (ev, arg) ->
            let tickerSymbol = j ("#tickerSymbol")
            async {
                let url = "http://localhost:48430/api/treading/addTicker"
                let req = System.Net.WebRequest.Create(url)
                do! req.AsyncPostJSONOneWay(url, tickerSymbol)
            }
            |> Async.StartImmediate)
```

F# and MVC - Web Api



F# and MVC - Web Api





F# Magic in review

- TypeProvider
- Function Composition - ROP
- Async Workflow
- MailboxProcessor – aka Agent
- Reactive Extensions
- FunScript
- Full integration with Asp.Net MVC & Web API

RX the Dual of IEnumerable

Formal definition [edit]

http://en.wikipedia.org/wiki/Dual_(category_theory)

We define the elementary language of category theory as the two-sorted first order language with objects and morphisms as distinct sorts, together with the relations of an object being the source or target of a morphism and a symbol for composing two morphisms.

Let σ be any statement in this language. We form the dual σ^{op} as follows:

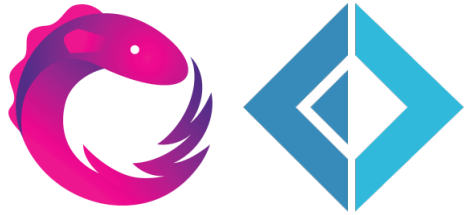
1. Interchange each occurrence of "source" in σ with "target".
2. Interchange the order of composing morphisms. That is, replace each occurrence of $g \circ f$ with $f \circ g$

Informally, these conditions state that the dual of a statement is formed by reversing arrows and compositions.

Duality is the observation that σ is true for some category *C* if and only if σ^{op} is true for C^{op}.

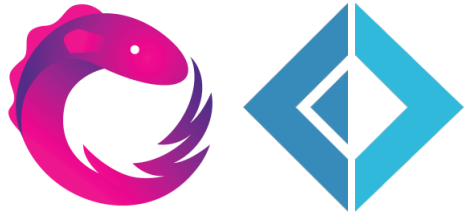
Reversing arrows
The input becomes output and <->

IObserver & IObservable



```
let observable = { new IObservable<string> with
                      member x.Subscribe(observer:IObserver<string>) =
                         { new IDisposable with
                              member x.Dispose() = ()
                              }
                         }

let observer = { new IObserver<string> with
                    member x.OnNext(value) = ()
                    member x.OnCompleted() = ()
                    member x.OnError(exn) = () }
```

When to use Rx



http://www.introtorx.com/content/v1.0.10621.0/01_WhyRx.html

When is Rx appropriate?

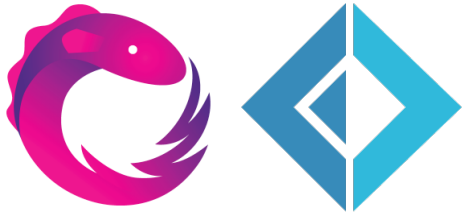
Rx offers a natural paradigm for dealing with sequences of events. A sequence can contain zero or more events. Rx proves to be most valuable when composing sequences of events.

Should use Rx

Managing events like these is what Rx was built for:

- UI events like mouse move, button click
- Domain events like property changed, collection updated, "Order Filled", "Registration accepted" etc.
- Infrastructure events like from file watcher, system and WMI events
- Integration events like a broadcast from a message bus or a push event from WebSockets API or other low latency middleware like Nirvana
- Integration with a CEP engine like StreamInsight or StreamBase.

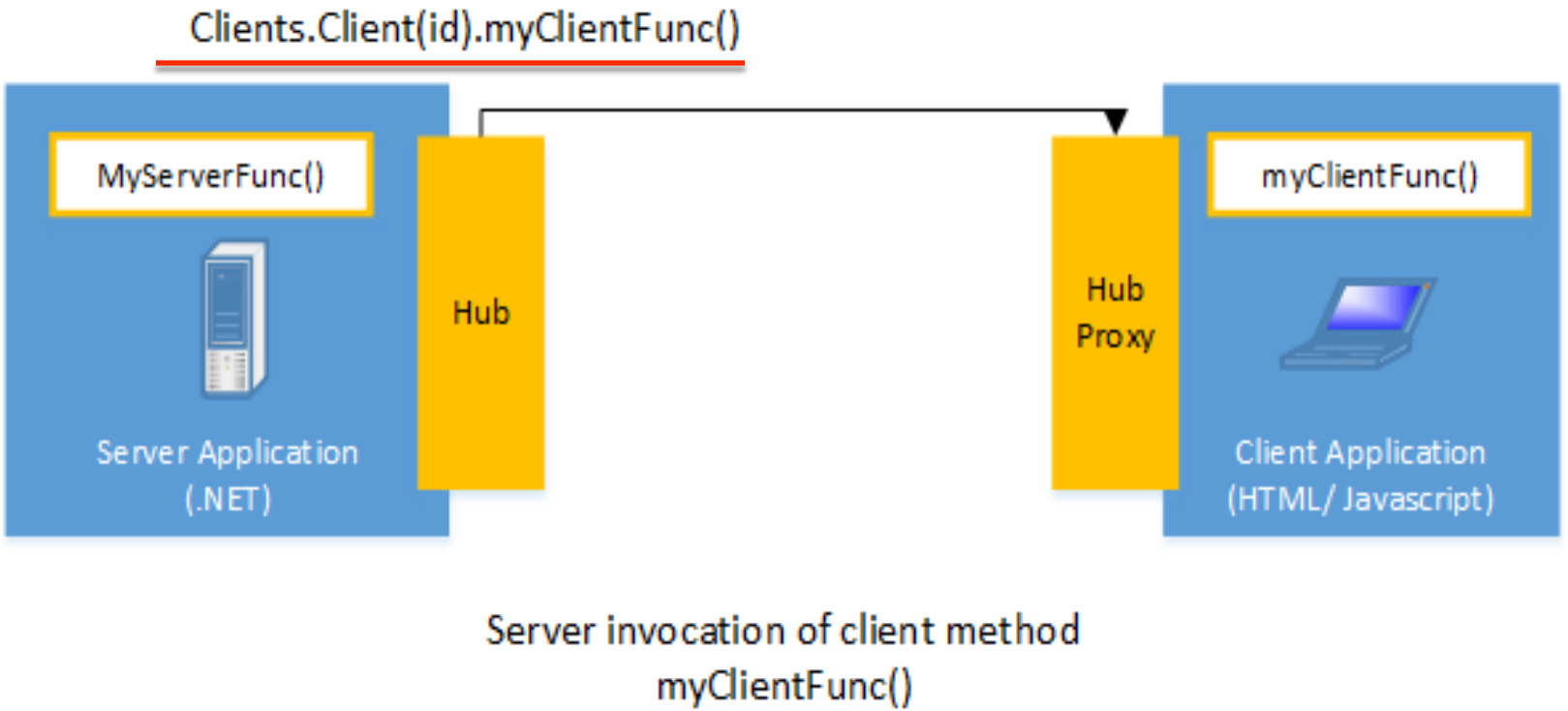
Subject<'a>.. as a bus



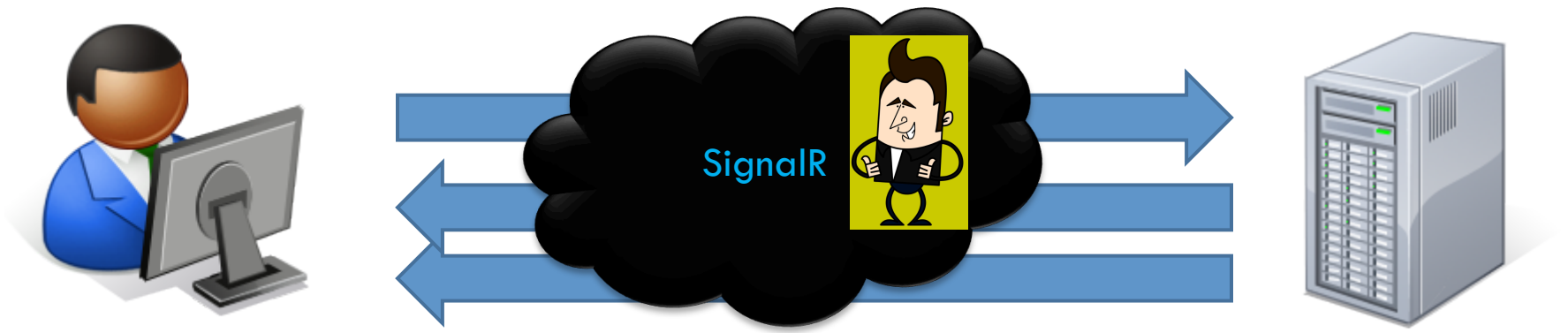
```
// Since RX is all about sequences of events/messages
// it does fit very well together with any sort of message bus or event broker.
type RxBusCommand() =
    let subject = new Subject<RxCommand>()

    member this.AsObservable() =
       subject.AsObservable()

    member this.Send(item:RxCommand) =
        subject.OnNext(item)


let busud = RxBusCommand()

busud.AsObservable().Do(fun t ->
                        match t with
                        | BuyTickets(name, quantity) -> printfn "I am buying %d tickets for %s" quantity name
                        | OrderDrink(drink) -> printfn "I am getting some %s to drink" drink).Subscribe() |> ignore

// The nice thing about this is that you get automatic Linq support since it is built into RX.
// So you can add message handlers that filters or transform messages.
busud.Send(BuyTickets("Opera", 2))
busud.Send(OrderDrink("Coke"))
```

```
type RxCommand =
    | BuyTickets of string * int
    | OrderDrink of string
```

What is SignalR?



What is SignalR?



Client

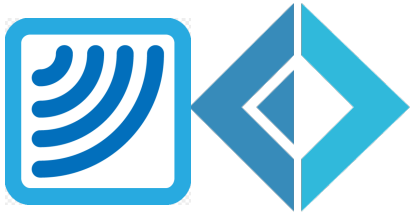
javaScript
.Net
WinRT

WebSocket
Long Polling
Server sent events
Forever Frame

Asp.Net
Self-Host
Owin

Server

Why SignalR?

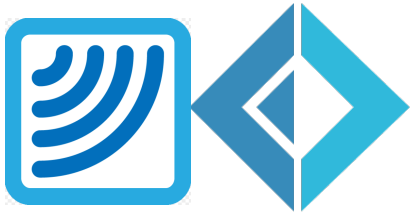


… the real questions are

- *When the Web users want their Data?*
- *When the Web user want the latest info?*

Why SignalR?

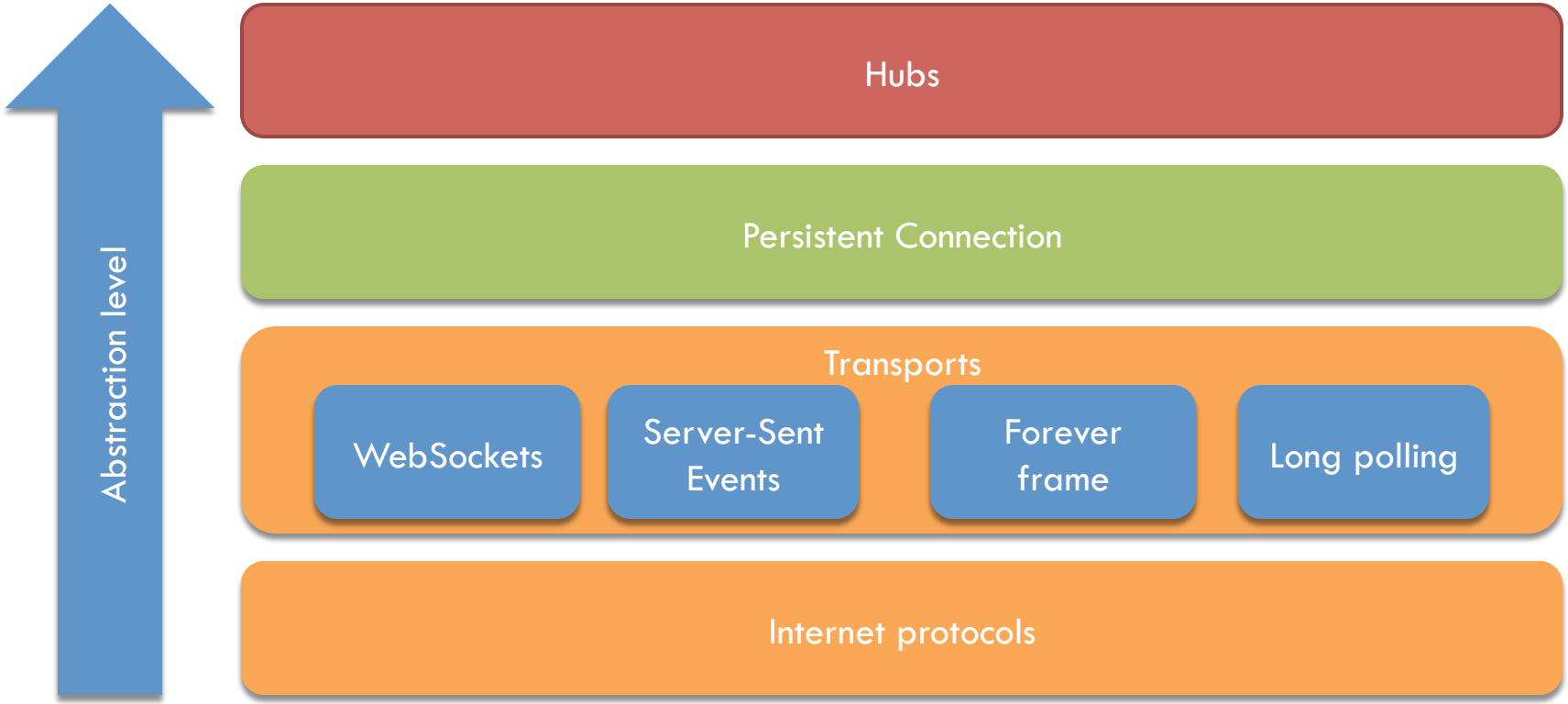
What can SignalR do?



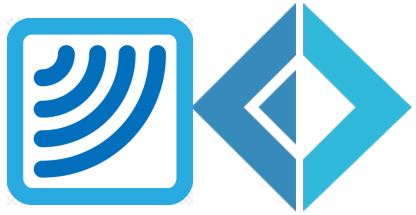
SignalR can be used to add any sort of "real-time" web functionality to your application…

- *Anything that needs live data*
  - *Chat Room application*
  - *Broadcasting (Stock Tickers, Live Scores…)*
  - *Internet Games (http://shootr.signalr.net/default.aspx#)*
  - *Social Media*
  - *Collaborative Apps*

SignalR levels of abstraction



Persistent Connection

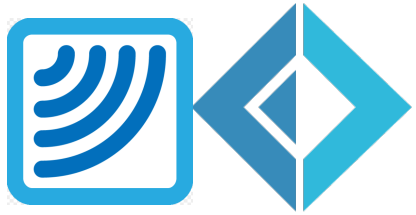


Client (javascript)

```
var conn = $.connection("myChat");
conn.start();

conn.send("Hello F#!!");

conn.receive(function(text){
    $("#log").append("<li>" + text …
```

```
type MyChat() =
        inherit PersistentConnection()

override x.OnConnected() : Task = …

override x.OnReceived(data) : Task =
        Connection.Broadcast(data)

override x.OnDisconnected() : Task =
```

Hubs

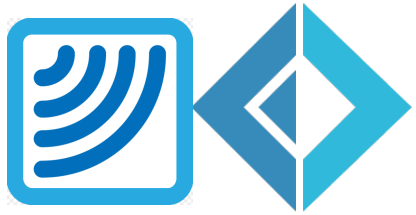


Client (javascript)

```
var chat = $.connection.myChat
$.connection.hub.start();

chat.server.message("Hello F#!!");

chat.client.notify = function(text){
    $("#log").append("<li>" + text …
```

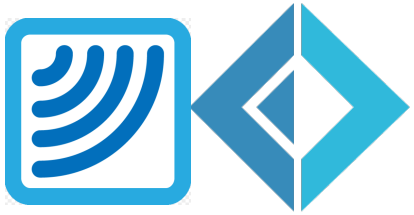
```
[<HubName("myChat")>]
type MyChat() =
        inherit Hub

  member x.Message(text) : Task =
      Clients.All.notify(text)
```

Hubs – Sending Message



```
[<HubName("ChatHub")>]
type ChatHub() =
    inherit Hub()

    member x.Send(name:string, message:string) =
        // send to all
        Clients.All.sendMessage(name, message)
        // send to specific client
        Clients.Client(Context.ConnectionId).sendMessage(message)
        // send only to caller
        Clients.Caller.sendMessage(name, message)
        // send to all but caller
        Clients.Others.sendMessage(name, message)
        // excluding some
        Clients.AllExcept(connectionId1, connectionId2).sendMessage(name, message)
        // send to a group
        Clients.Group(groupName).sendMessage(name, message)
```

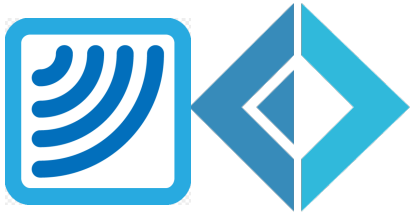
SignalR ? Dynamic



```
type IChatHubClient =
    abstract member BroadcastMessage : string -> unit

[<HubName("chatHub")>]
type ChatRoomHub() =
    inherit Hub<IChatHubClient>()

    member this.SendMessage(text:string) =
        this.Clients.All.BroadcastMessage(text)
```

SignalR & JavaScript

F# Type Provider for SignalR



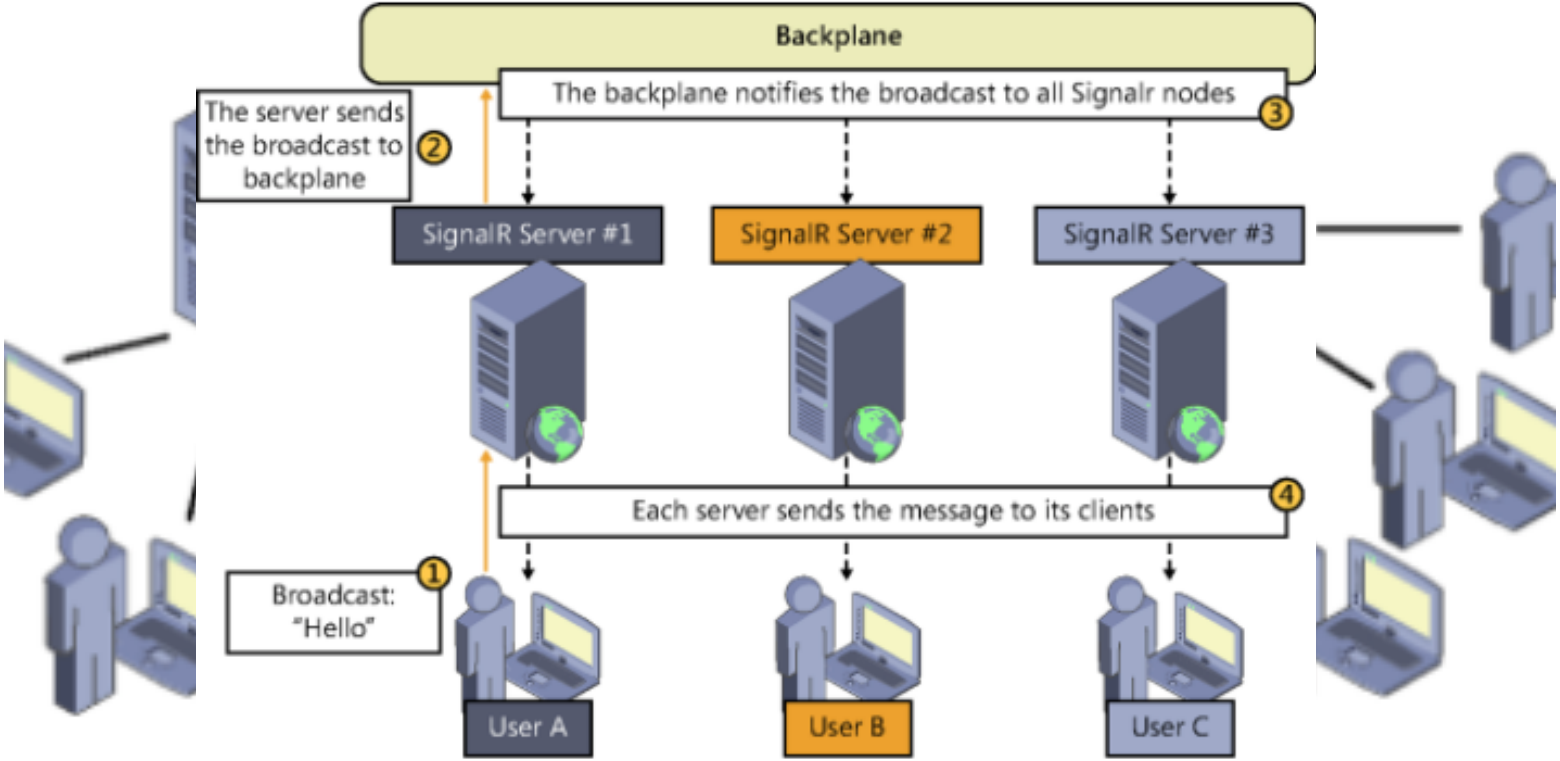
Type provider giving a typed view of a .NET SignalR server Hub to client-side code compiled from F# to JavaScript with FunScript.

```
open FunScript
open SignalRProvider

let signalR = Globals.Dollar.signalR
let serverHub = new Hubs.myhub(signalR.hub)

ignore <| serverHub.SendMessage("Hello F#")
ignore <| serverHub.SendMessage( 42 )
```

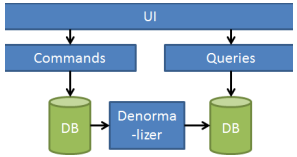
```
This expression was expected to have type
    string
but here has type
    int
```

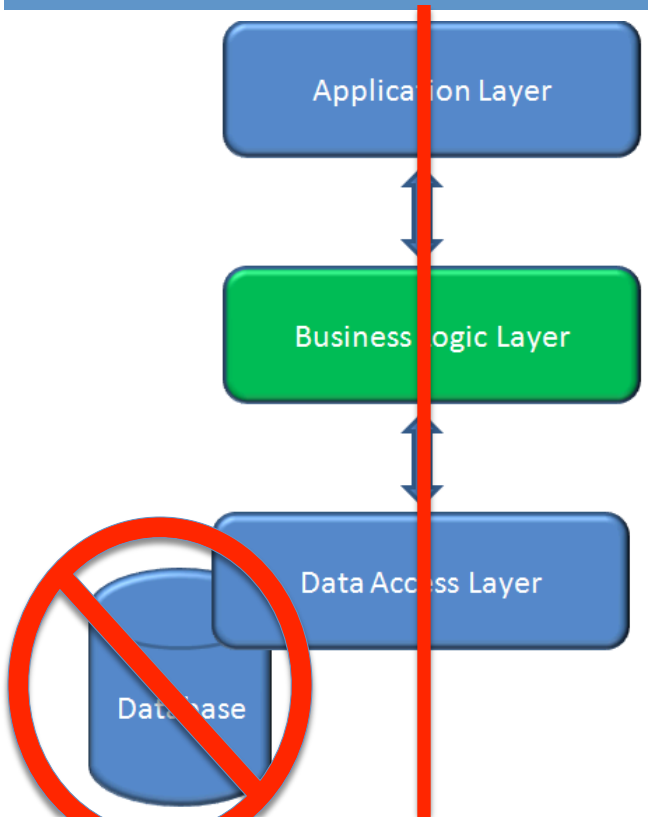
```
[<HubName("myhub")>]
type MyHub() =
    inherit Hub

    member this.SendMessage(text : string) : string =
        this.Clients.Others.BroadcastMessage(text)
```

Scale-out SignalR (backplane)



Web N-Tier Architecture





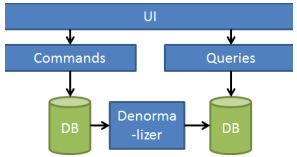
In the case of three-tier architecture

1) Presentation tier
2) Business logic tier
3) Data storage tier

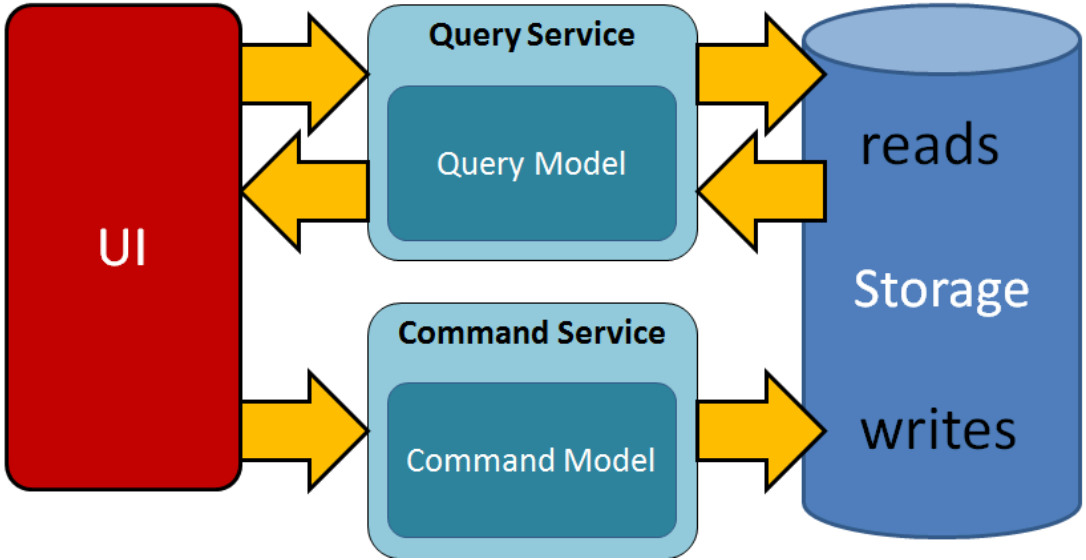
Problems:

1) The project can become very difficult to maintain
2) Scalability as only one data base handles read/write

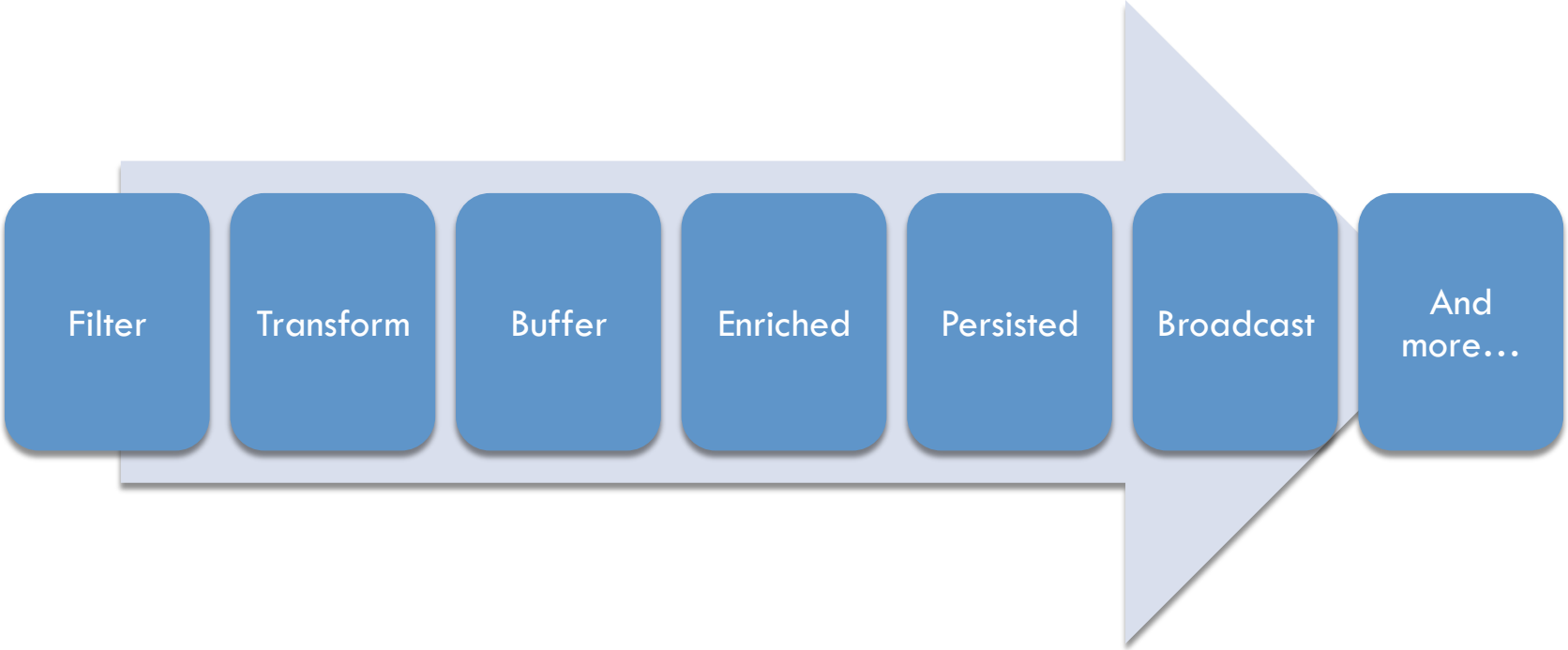
CQRS pattern



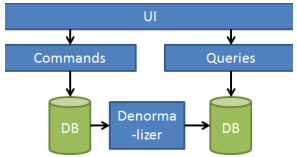
Command Query Responsibility Segregation



CQRS is sort of data flow pattern



CQRS benefits



- "Almost" infinite scalability
  - Performance and scalability are always concerns when handling a large volume of transactions over the internet
- Clear understanding what the application does
- Separation of concerns

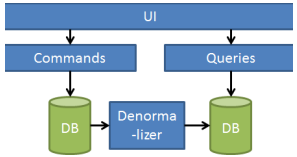
.. scalability, simplicity, and maintainability…

How to handle UI-Eventual Consistency

Building a UI for a CQRS system is challenging

- *The commands could complete fast*
- *The read model is eventually consistent*
- *The read of the data store may return stale results*

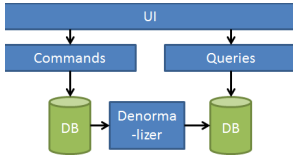
Query



How can we update the UI?

- *Reload the page*
- *Disable then refresh after editing*
- *Use a confirmation screen*
- *Fake it*
- *"Amazon Style"*

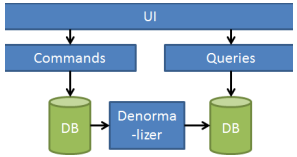
SignalR to replace Query



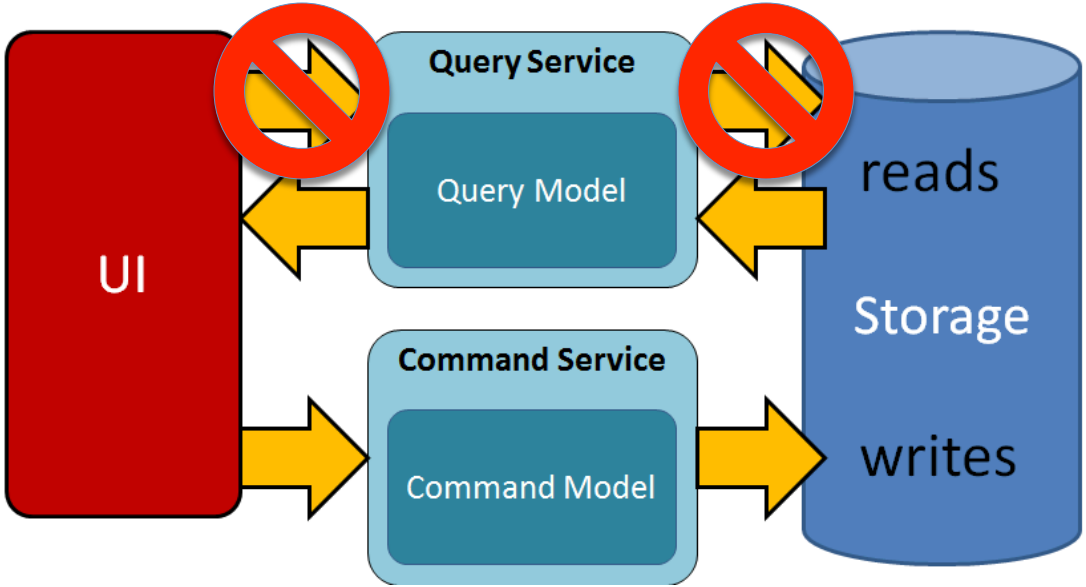
How can we update the UI?

- *Reload the page*
- *Disable then refresh after editing*
- *Use a confirmation screen*
- *Fake it*
- *"Amazon Style"*
- *SignalR can be used as a Query to add any sort of "real-time" web functionality to your application…*

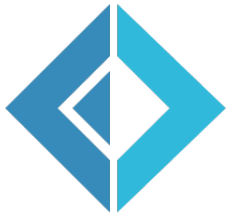
CQRS pattern



Command Query Responsibility Segregation

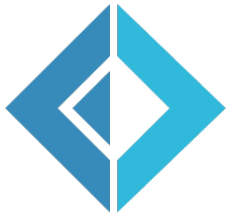


CQRS CAP



- This works when the user actually expects some sort of “background” work to happen, or that we present this to the user in a meaningful way.
- But when doing CQRS, eventual consistency is an orthogonal choice. They are two completely separate concerns. Going back to our new CQRS design:
- We have many choices here on what should be synchronous, and what should not. It can all be synchronous, all be async, it’s just a separate decision.
- What I have found though that is if we build asynchronous denormalization in the back-end, but try to *mimic* synchronous results in the front end, we’re really just choosing async where it’s not needed. Not in all cases of course, but for most of the ones I’ve seen.
- Some async-sync mimicry I’ve seen includes:
- Using Ajax from the server to ping the read store to see if denormalization is “done”
- Using SignalR to notify the client when the denormalization is “done”
- Writing to the read store synchronously, but then allowing eventual consistency to fix any mistakes

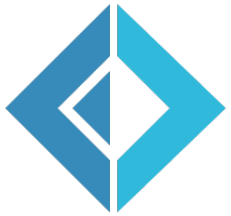
SAGA



- A Saga is a distribution of multiple workflows across multiple systems, each providing a path (fork) of compensating actions in the event that any of the steps in the workflow fails.

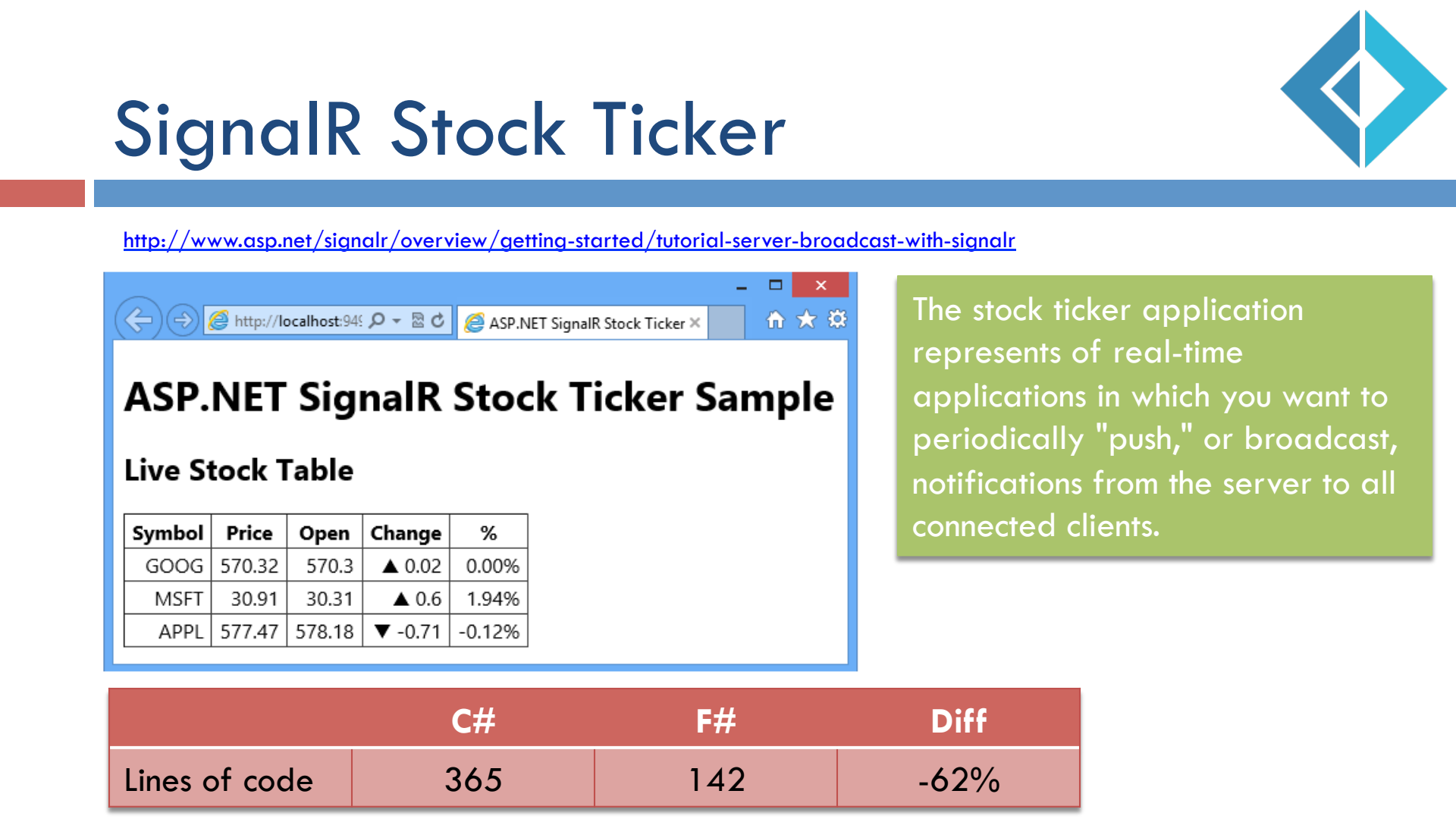
- "Sagas and persistence
- In general, a saga must be persistent and persistence of the saga is a typical responsibility of the bus. In this regard, it might completely be transparent to you if you don't write a bus class yourself. In the sample Bus class, we simulated persistence through an in-memory dictionary—whereas, for example, NServiceBus uses SQL Server. For persistence to happen, it is key that you give a unique ID to each saga instance."

When to use CQRS



- In general, the CQRS pattern could be very valuable in situations when you have highly collaborative data and large, multi-user systems, complex, include ever-changing business rules, and delivers a significant competitive advantage of business. It can be very helpful when you need to track and log historical changes.
- With CQRS you can achieve great read and write performance. The system intrinsically supports scaling out. By separating read and write operations, each can be optimized.
- CQRS can by very helpful when you have difficult business logic. CQRS forces you to not mix domain logic and infrastructural operations.
- With CQRS you can split development tasks between different teams with defined interfaces.
- When not to use CQRS
- If you are not developing a highly collaborative system where you don't have multiple writers to the same logical set of data you shouldn't use CQRS.

SignalR Stock Ticker

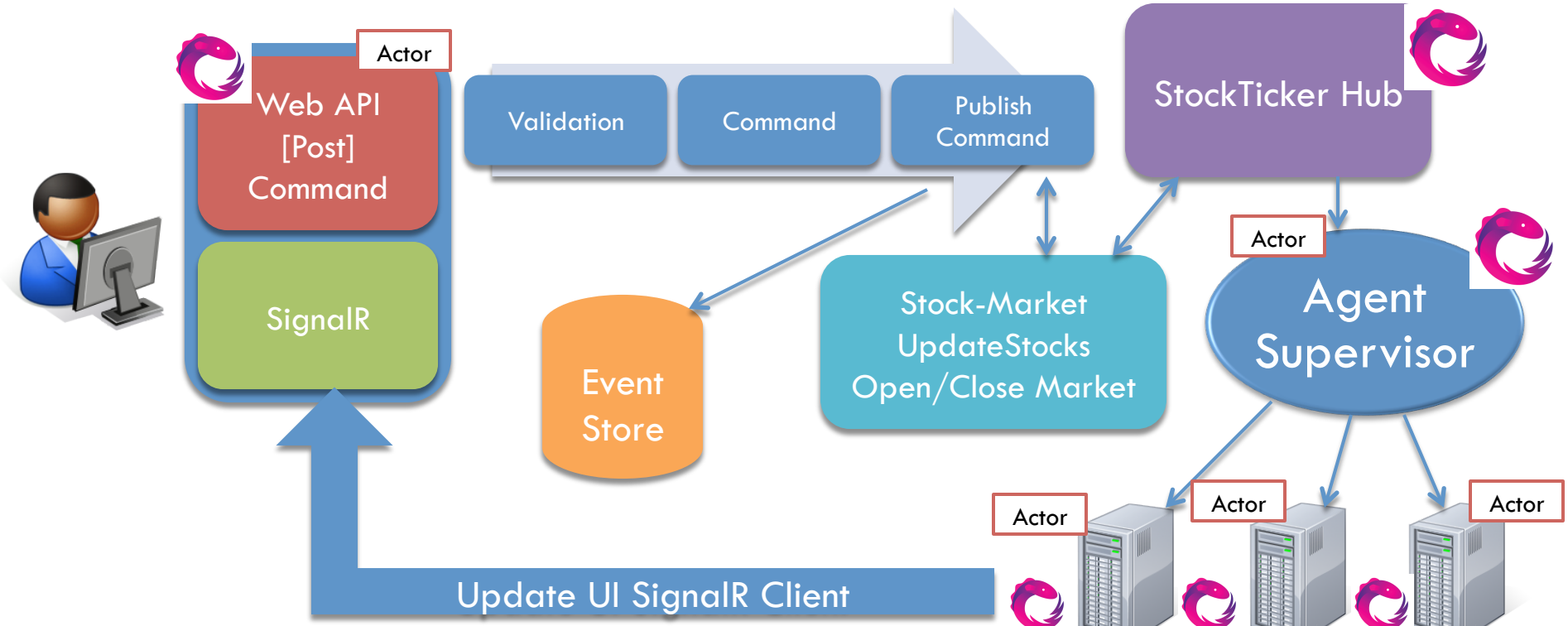


http://www.asp.net/signalr/overview/getting-started/tutorial-server-broadcast-with-signalr

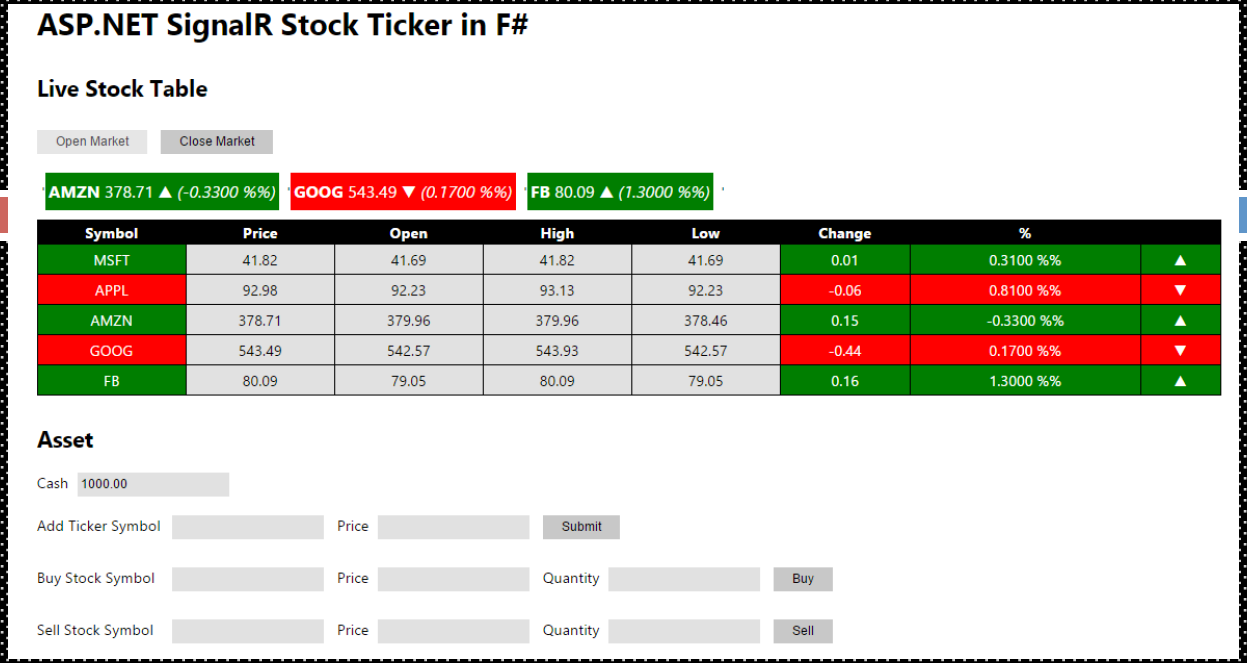
The stock ticker application represents of real-time applications in which you want to periodically "push," or broadcast, notifications from the server to all connected clients.

	C#	F#	Diff
Lines of code	365	142	-62%

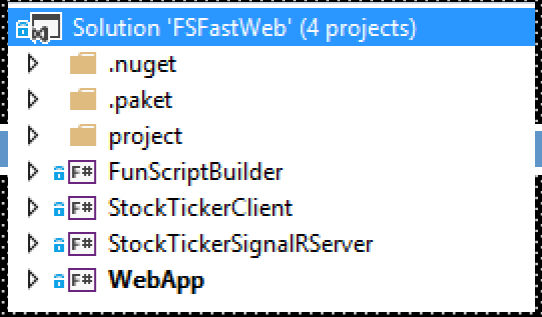
Demo Project

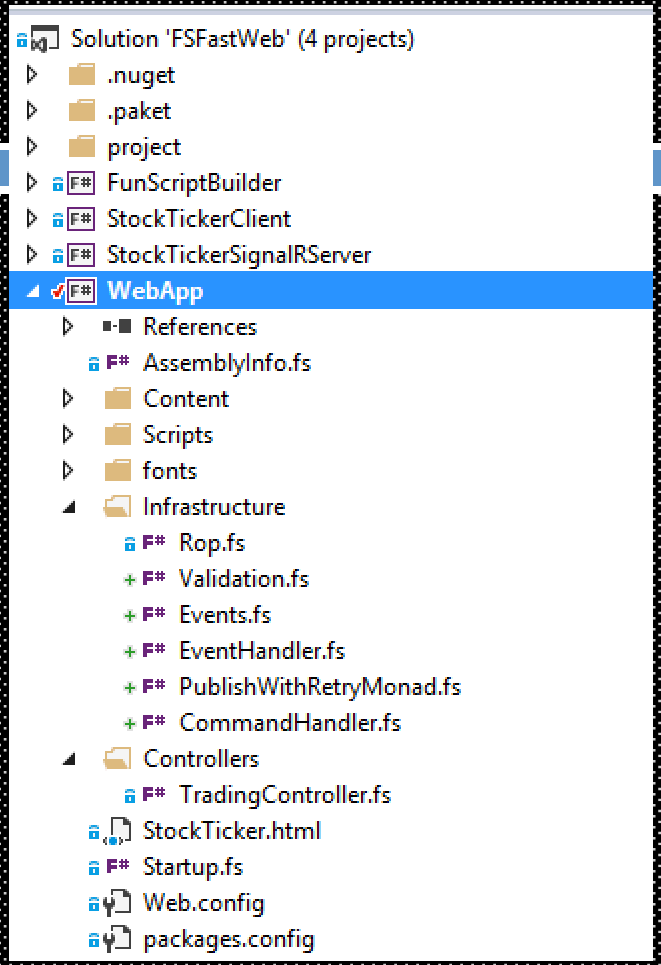



DEMO


ASP.NET SignalR Stock Ticker in F#
Live Stock Table
Open Market
Close Market
AMZN 378.71 ▲ (-0.3300 %%)
GOOG 543.49 ▼ (0.1700 %%)
FB 80.09 ▲ (1.3000 %%)
Symbol Price Open High Low Change %
MSFT 41.82 41.69 41.82 41.69 0.01 0.3100 %% ▲
APPL 92.98 92.23 93.13 92.23 -0.06 0.8100 %% ▼
AMZN 378.71 379.96 379.96 378.46 0.15 -0.3300 %% ▲
GOOG 543.49 542.57 543.93 542.57 -0.44 0.1700 %% ▼
FB 80.09 79.05 80.09 79.05 0.16 1.3000 %% ▲
Asset
Cash 1000.00
Add Ticker Symbol
Price
Submit
Buy Stock Symbol
Price
Quantity
Buy
Sell Stock Symbol
Price
Quantity
Sell

The Code!


Solution 'FSFastWeb' (4 projects)
.nuget
.paket
project
FunScriptBuilder
StockTickerClient
StockTickerSignalRServer
WebApp


Solution 'FSFastWeb' (4 projects)
.nuget
.paket
project
FunScriptBuilder
StockTickerClient
StockTickerSignalRServer
WebApp
References
AssemblyInfo.fs
Content
Scripts
fonts
Infrastructure
Rop.fs
Validation.fs
Events.fs
EventHandler.fs
PublishWithRetryMonad.fs
CommandHandler.fs
Controllers
TradingController.fs
StockTicker.html
Startup.fs
Web.config
packages.config

StockTicker.html

```
{{#stocks.length}}
<section id="main">
    <div id="stockTicker">
        <div class="inner">
            <ul>
                {{#stocks:i}}
                {{>scrollStock}}
                {{/stocks}}
            </ul>
        </div>
    </div>

    <table id="stockTable" border="1">
        <thead>
            <tr><th>Symbol</th><th>Price</th><th>Open</th>
            <th>High</th><th>Low</th><th>Change</th><th>%</th><th></th>
        </thead>
        <tbody>
            {{#stocks:i}}
            {{>stock}}
            {{/stocks}}
        </tbody>
    </table>
</section>
{{/stocks.length}}
```

StockTicker.html

```
{{#buyOrders.length}}
<h2>Orders Buy</h2>
<section id="ordersSection">
    <div class="inner">
        <table id="ordersTable" border="1">
            <thead>
                <tr><th>Symbol</th><th>Price</th><th>Quantity</th><th>Total</th></tr>
            </thead>
            <tbody>
                {{#buyOrders:i}}
                {{>order}}
                {{/buyOrders}}
            </tbody>
        </table>
    </div>
</section>
{{/buyOrders.length}}
```

StockTicker.html

```
{{#sellOrders.length}}
<h2>Orders Sell</h2>
<section id="ordersSection">
    <div class="inner">
        <table id="sellsTable" border="1">
            <thead>
                <tr><th>Symbol</th><th>Price</th><th>Quantity</th><th>Total</th></tr>
            </thead>
            <tbody>
                {{#sellOrders:i}}
                {{>order}}
                {{/sellOrders}}
            </tbody>
        </table>
    </div>
</section>
{{/sellOrders.length}}
```

StockTicker.html

```
<script id="portfolioItem" type="text/ractive">
    <tr data-symbol="{{Symbol}}">
        <td><span class="symbol">{{Symbol}}</span></td>
        <td><span class="symbol">{{OrderType}}</span></td>
        <td><span class="price">{{OrderPrice}}</span></td>
        <td><span class="price">{{Price}}</span></td>
        <td><span>{{Quantity}}</span></td>
        <td><span>{{ Quantity * Price  }}</span></td>
    </tr>
</script>
```

```
<script id="order" type="text/ractive">
    <tr data-symbol="{{Symbol}}">
        <td><span class="symbol">{{Symbol.toUpperCase()}}</span></td>
        <td><span class="price">{{Price}}</span></td>
        <td>{{Quantity}}</td>
        <td><span>{{Quantity * Price}}</span></td>
    </tr>
</script>
```

TradingController.fs

```
[<RoutePrefix("api/trading")>]
type TradingController() =
    inherit ApiController()
```

TradingController.fs

```
[<RoutePrefix("api/trading")>]
type TradingController() =
    inherit ApiController()

    // the controller act as a observable publisher of messages
    // keep the controller loosely coupled using the Reactive Extensions
    // defining a Reactive Subject which will fire up
    // requests to the subscriber(s)
    let subject = new Subject<CommandWrapper>()
```

```
  // The controller bahaves as Observable publisher and it can be registered
  interface IObservable<CommandWrapper> with
      member this.Subscribe observer = subject.Subscribe observer

  override this.Dispose disposing =
      if disposing then
          subject.Dispose()
      base.Dispose disposing
```

TradingController.fs

```fsharp
[<Route("sell"); HttpPost>]
member this.PostSell([<FromBody>] tr : TradingRequest) =
    async {
        // current connection ID from SignalR
        let connectionId = tr.ConnectionID

        // create TradingCommand
        // validate
        // log
        // publish
        // return response
        return // TradingCommand
            {   Symbol = tr.Symbol.ToUpper()
                Quantity = tr.Quantity
                Price = tr.Price
                Trading = TradingType.Sell }
            |> tradingdValidation // validation using function composition
            |> log
            |> publish connectionId
            |> toResponse this.Request

    } |> Async.StartAsTask // can easily make asynchronous controller methods.
```

TradingController.fs

```fsharp
[<Route("sell"); HttpPost>]
member this.PostSell([<FromBody>] tr : TradingRequest) =
    async {
        // current connection ID from SignalR
        let connectionId = tr.ConnectionID

        // create TradingCommand
        // validate
        // log
        // publish
        // return response
        return // TradingCommand
            Symbol = tr.Symbol.ToUpper()
            Quantity = tr.Quantity
            Price = tr.Price
            Trading = TradingType.Sell }
            |> tradingValidation // validation using function composition
            |> log
            |> publish connectionId
            |> toResponse this.Request

    } |> Async.StartAsTask // can easily make asynchronous controller methods.
```

```fsharp
[<CLIMutable>]
type TradingRecord =
    { Symbol : string
      Quantity : int
      Price : float
      Trading : TradingType }
and TradingType =
    | Buy
    | Sell
```

TradingController.fs

```fsharp
[<Route("sell"); HttpPost>]
member this.PostSell([<FromBody>] tr : TradingRequest) =
    async {
        // current connection ID from SignalR
        let connectionId = tr.ConnectionID

        // create TradingCommand
        // validate
        // log
        // publish
        // return response
        return // TradingCommand
               Symbol = tr.Symbol.ToUpper()
               Quantity = tr.Quantity
               Price = tr.Price
               Trading = TradingType.Sell }
             > tradingdValidation // validation using function composition
             |> log
             |> publish connectionId
             |> toResponse this.Request

    } |> Async.StartAsTask // can easily make asynchronous
```

```fsharp
let validateTicker  (input : TradingRecord) =
    if input.Symbol = "" then Failure "Ticket must not be blank"
    else Success input

let validateQuantity (input : TradingRecord) =
    if input.Quantity <= 0 || input.Quantity > 50
        then Failure "Quantity must be positive and not be more than 50"
    else Success input

let validatePrice (input : TradingRecord) =
    if input.Price <= 0.
        then Failure "Price must be positive"
    else Success input

let tradingdValidation =
    validateTicker >> bind validatePrice >> bind validateQuantity
```

TradingController.fs

```fsharp
[<Route("sell"); HttpPost>]
member this.PostSell([<FromBody>] tr : TradingRequest) =
    async {
        // current connection ID from SignalR
        let connectionId = tr.ConnectionID

        // create TradingCommand
        // validate
        // log
        // publish
        // return response
        return // TradingCommand
            {   Symbol = tr.Symbol.ToUpper()
                Quantity = tr.Quantity
                Price = tr.Price
                Trading = TradingType.Sell }
            |> tradingdValidation // validation using function composition
            |> log
            |> publish connectionId
            |> toResponse this.Request

    } |> Async.StartAsTask // can easily m
```

```fsharp
[<CLIMutable>]
type CommandWrapper =
    { ConnectionId : string
      Id : Guid
      Created : DateTimeOffset
      Command : TradingCommand }

    static member CreateTradingCommand connectionId (item : TradingRecord) =
        let command =
            match item.Trading with
            | Buy -> BuyStockCommand(connectionId, item)
            | Sell -> SellStockCommand(connectionId, item)
        { Id = (Guid.NewGuid())
          Created = (DateTimeOffset.Now)
          ConnectionId = connectionId
          Command = command }
```

TradingController.fs

```fsharp
[<Route("sell"); HttpPost>]
member this.PostSell([<FromBody>] tr : TradingRequest) =
    async {
        // current connection ID from SignalR
        let connectionId = tr.ConnectionID

        // create TradingCommand
        // validate
        // log
        // publish
        // return response
        return // TradingCommand
            {   Symbol = tr.Symbol.ToUpper()
                Quantity = tr.Quantity
                Price = tr.Price
                Trading = TradingType.Sell }
            |> tradingdValidation // validation using function composition
            |> log
            |> publish connectionId
            |> toResponse this.Request

    } |> Async.StartAsTask // can easily make asynchronous c
```

```fsharp
let log res =
    let msg =
        match res with
        | Success(m) -> sprintf "Validation successful - %A" m
        | Failure(f) -> sprintf "Validation failed - %s" f
    System.Diagnostics.Debug.WriteLine("[LOG]" + msg)
    res

let toResponse (request : HttpRequestMessage) result =
    let response =
        match result with
        | Success(_) -> request.CreateResponse(HttpStatusCode.OK)
        | _ -> request.CreateResponse(HttpStatusCode.BadRequest)
    response
```

Startup.fs

```
/// Route for ASP.NET Web API applications
type HttpRoute = {
    controller : string
    id : RouteParameter }
```

```
[<Sealed>]
type Startup() =

    member __.Configuration(builder : IAppBuilder) =

        let config =
            let config = new HttpConfiguration()
            // Configure routing
            config.MapHttpAttributeRoutes()

            // Configure serialization
            config.Formatters.XmlFormatter.UseXmlSerializer <- true
            config.Formatters.JsonFormatter.SerializerSettings.ContractResolver
                            <- Newtonsoft.Json.Serialization.CamelCasePropertyNamesContractResolver()

            config.Routes.MapHttpRoute("tradingApi", "api/trading/{id}",
                { controller = "Trading"; id = RouteParameter.Optional }) |> ignore


            config.Routes.MapHttpRoute("DefaultApi", "api/{controller}/{id}",
                { controller = "{controller}"; id = RouteParameter.Optional }) |> ignore

            // replace the defaulr controller activator
            config.Services.Replace(typeof<IHttpControllerActivator>,...

            config
```

Startup.fs

```
let configSignalR = new HubConfiguration(EnableDetailedErrors = true)

GlobalHost.HubPipeline.AddModule(new ErrorHandlingPipelineModule()) |> ignore

Owin.CorsExtensions.UseCors(builder, Microsoft.Owin.Cors.CorsOptions.AllowAll) |> ignore

//  MapSignalR() is an extension method of IAppBuilder provided by SignalR to facilitate mapping
//  and configuration of the hub service.
//  The generic overload MapSignalR<TConnection> is used to map persistent connections,
//  that we do not have to specify the classes that implement the services
builder.MapSignalR(configSignalR) |> ignore

builder.UseWebApi(config) |> ignore
```

Startup.fs

```
// Transform the web api controller in an Observable publisher,
// register the controller to the command dispatcher.

// Hook into the Web API framework where it creates controllers
type CompositionRoot(tradingRequestObserver:IObserver<CommandWrapper>) =
    interface IHttpControllerActivator with
        member this.Create(request, controllerDescriptor, controllerType) =
            if controllerType = typeof<TradingController> then
                let c = new TradingController()
                c   // c.Subscribe  agent.Post
                |> Observable.subscribeObserver tradingRequestObserver
                |> request.RegisterForDispose
                c :> IHttpController
            else
                raise
                <| ArgumentException(
                    sprintf "Unknown controller type requested: %O" controllerType,
                    "controllerType")
```

```
// replace the default controller activator
config.Services.Replace(typeof<IHttpControllerActivator>,
                        // I create a subscription controller to the Agent
                        // Each time a message come in
                        // the publisher send the message (OnNext)
                        // to all the subscribers, in this case the Agent
                        CompositionRoot( Observer.Create(fun x -> agent.Post(x)) ))
```

Startup.fs

```
// Dispatch eterogenous message depending on the incoming
// Using Pub/Sub API of Reactive Extensions used
// to pipe messages to an Agent
// The Agent only dependd on IObserver implementation
// reducing the dependencies
let agent = new Agent<CommandWrapper>(fun inbox ->
        let rec loop () =
            async {
                let! cmd = inbox.Receive()
                do! cmd |> AsyncHandle
                return! loop() }
        loop())
do agent.Start()
```

Startup.fs

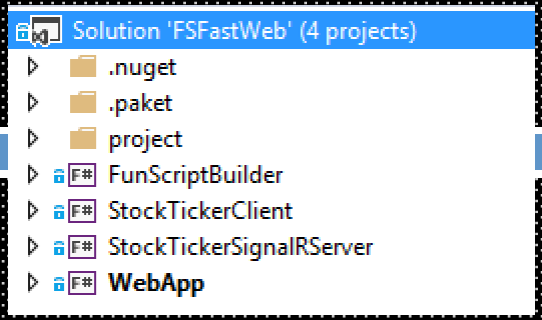
```
// Dispatch eterogenous message depending on the incoming
// Using Pub/Sub API of Reactive Extensions used
// to pipe messages to an Agent
// The Agent only depends on IObservable implementation
// r
let
```

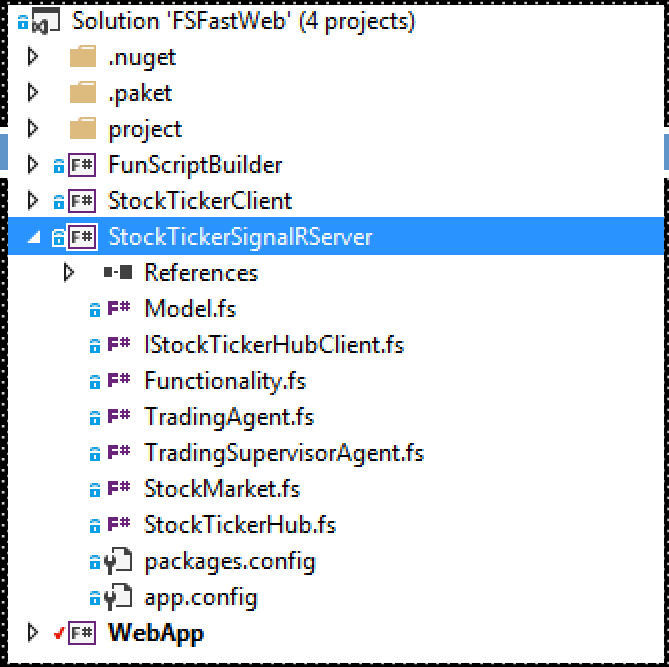
```
module CommandHandler =

    let AsyncHandle (cmd:CommandWrapper) =
        async { // async workflow

            let { ConnectionId = connectionId
                  Id = id
                  Created = created
                  Command = tradingCommand } = cmd

            retryPublish {
                do! SendCommand( connectionId, cmd)
            }
        }
```

```
do a
```

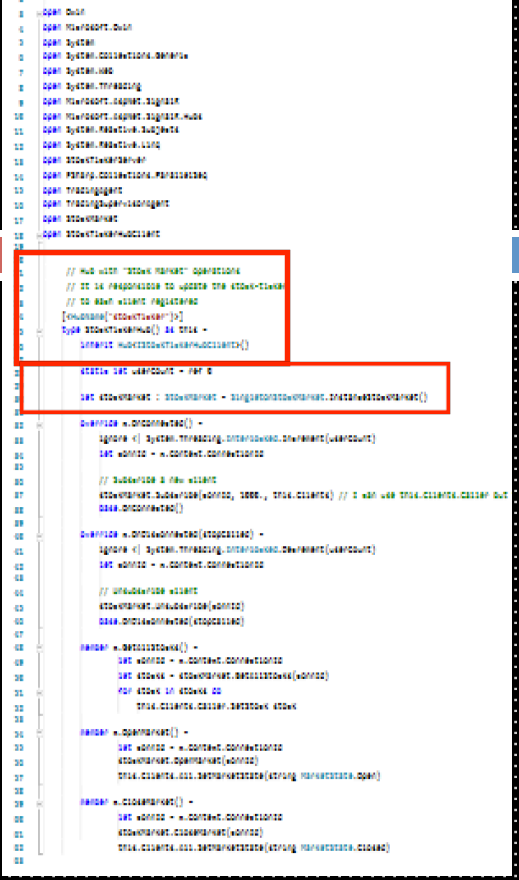
Startup.fs

```fsharp
[<AutoOpenAttribute>]
module RetryPublishMonad =
    let stockMarket = SingletonStockMarket.InstanceStockMarket()

    let Storage = new EventStorage()

    type SendCommand<'a> =
        | SendCommand of string * 'a

    type RetryPublishBuilder(max, sleepMilliseconds : int) =
        member x.Return(a) = a
        member x.Bind(SendCommand(connId, commandWrapper:CommandWrapper) : SendCommand<_>, fn) =

            let rec loop n (error:exn option) =
                async {
                    if n = 0 then ...
                    else
                        try
                            // publish the command
                            stockMarket.PublishCommand(connId, commandWrapper)

                            let event = ...

                            // and persiste the event into the Event-Storage
                            (Guid(connId), Event.CreateEventDescriptor(commandWrapper.Id, event))
                            ||> Storage.SaveEvent

                        with ex ->
                            sprintf "Call failed with %s. Retrying." ex.Message |> printfn "%s"
                            do! Async.Sleep sleepMilliseconds
                            return! loop (n - 1) (Some(ex))
                }
            loop max None |> Async.RunSynchronously

    let retryPublish = RetryPublishBuilder(3, 1000)
```


Solution 'FSFastWeb' (4 projects)
.nuget
.paket
project
FunScriptBuilder
StockTickerClient
StockTickerSignalRServer
WebApp


Solution 'FSFastWeb' (4 projects)
.nuget
.paket
project
FunScriptBuilder
StockTickerClient
StockTickerSignalRServer
References
Model.fs
IStockTickerHubClient.fs
Functionality.fs
TradingAgent.fs
TradingSupervisorAgent.fs
StockMarket.fs
StockTickerHub.fs
packages.config
app.config
WebApp

StockTickerHub.fs

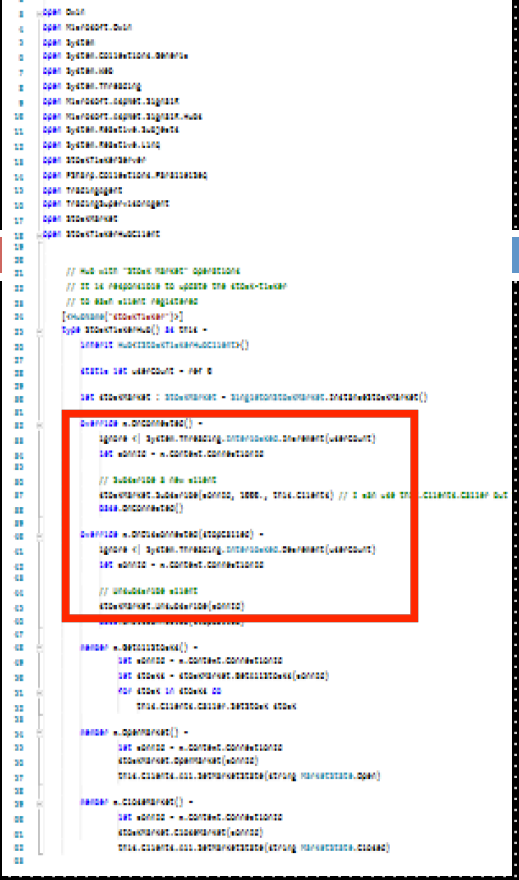


```
// interface StockTickerHub SignalR
type IStockTickerHubClient =
    abstract SetMarketState : string -> unit
    abstract UpdateStockPrice : Stock -> unit
    abstract SetStock : Stock -> unit
    abstract UpdateOrderBuy : OrderRecord -> Unit
    abstract UpdateOrderSell : OrderRecord -> Unit
    abstract UpdateAsset : Asset -> Unit
    abstract SetInitialAsset : float -> Unit
```

```
// Hub with "Stock Market" operations
// It is responsible to Subscribe and
// Unsubscribe clients
// and update the stock-ticker
// for each clients
[<HubName("stockTicker")>]
type StockTickerHub() as this =
    inherit Hub<IStockTickerHubClient>()
```

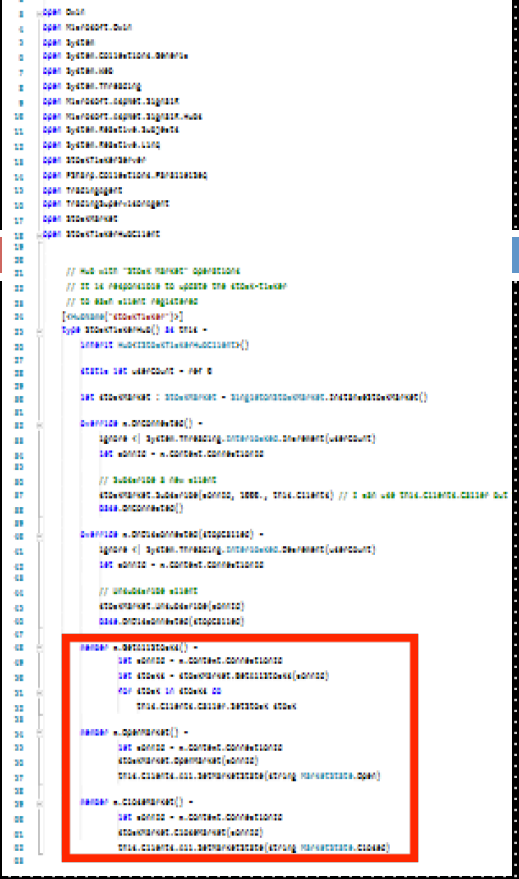
```
static let userCount = ref 0

let stockMarket : StockMarket = SingletonStockMarket.InstanceStockMarket()
```

StockTickerHub.fs



```
override x.OnConnected() =
    ignore <| System.Threading.Interlocked.Increment(userCount)
    let connId = x.Context.ConnectionId

    // Subscribe a new client
    stockMarket.Subscribe(connId, 1000., this.Clients)
    base.OnConnected()

override x.OnDisconnected(stopCalled) =
    ignore <| System.Threading.Interlocked.Decrement(userCount)
    let connId = x.Context.ConnectionId

    // Unsubscribe client
    stockMarket.Unsubscribe(connId)
    base.OnDisconnected(stopCalled)
```

StockTickerHub.fs



```
member x.GetAllStocks() =
        let connId = x.Context.ConnectionId
        let stocks = stockMarket.GetAllStocks(connId)
        for stock in stocks do
            this.Clients.Caller.SetStock stock

member x.OpenMarket() =
        let connId = x.Context.ConnectionId
        stockMarket.OpenMarket(connId)
        this.Clients.All.SetMarketState(string MarketState.Open)

member x.CloseMarket() =
        let connId = x.Context.ConnectionId
        stockMarket.CloseMarket(connId)
        this.Clients.All.SetMarketState(string MarketState.Closed)
```

StockTickerMarket.fs

```
//  Main "Controller" that run the Stock Market
type StockMarket (?initStocks : Stock seq) =
    let initStocks = defaultArg initStocks Seq.empty<Stock>

    let tradingSupervisorAgent = new TradingSupervisorAgent()
```

```
// Simulates "Stock-Ticker" updating prices every n
// ~is good enough~
let startTicker (stockAgent : Agent<StockTickerMessage>) =
    let rxTimer =
        Observable.Interval(TimeSpan.FromMilliseconds 85.0)
        |> Observable.subscribe(fun _ -> stockAgent.Post UpdateStockPrices)
    rxTimer
```

StockTickerMarket.fs

```
type StockTickerMessage =
    | UpdateStock of Stock
    | UpdateStockPrices
    | PublishCommand of connId : string * CommandWrapper
    | OpenMarket of string
    | CloseMarket of string
    | GetMarketState of string * AsyncReplyChannel<MarketState>
    | GetAllStocks of string * AsyncReplyChannel<Stock list>
and MarketState =
    | Closed
    | Open
```

```
// Agent that updates the stocks
// open and close the market
let stockAgent =
    Agent<StockTickerMessage>.Start(fun inbox ->
        let rec marketIsOpen (stocks : ResizeArray<Stock>) (stockTicker
```

```
and marketIsClosed (stocks : ResizeArray<Stock>) = async {
```

StockTickerMarket.fs

```
type StockTickerMessage =
    | UpdateStock of Stock
    | UpdateStockPrices
    | PublishCommand of connId : string * CommandWrapper
    | OpenMarket of string
    | CloseMarket of string
    | GetMarketState of string * AsyncReplyChannel<MarketState>
    | Ge
and Mark
    | Cl
    | Op
```

```
[<CLIMutableAttribute>]
type Stock =
    { Symbol : string
      DayOpen : float
      DayLow : float
      DayHigh : float
      LastChange : float
      Price : float
      Index : int }
    member x.PriceChanged = x.Price - x.DayOpen

    member x.PercentChanged = double (Math.Round(x.PriceChanged / x.Price, 4))
```

```
// Agent th
// open and
let stockAg
    Agent<S
        let
```

```
and marketIsClosed (stocks : ResizeArray<Stock>) = async {
```

StockTickerMarket.fs

```
and marketIsClosed (stocks : ResizeArray<Stock>) = async {
        let! msg = inbox.Receive()
        match msg with
        | GetMarketState(c, reply) ->  reply.Reply(MarketState.Closed)
                                       return! marketIsClosed stocks

        | GetAllStocks(c,reply) -> reply.Reply((stocks |> Seq.toList))
                                   return! marketIsClosed stocks

        | OpenMarket(c) -> return! marketIsOpen stocks (startTicker (inbox))

        | _ -> return! marketIsClosed stocks }
marketIsClosed (new ResizeArray<Stock>(initStocks)))
```

StockTickerMarket.fs

```
let rec marketIsOpen (stocks : ResizeArray<Stock>) (stockTicker : IDisposable) =
    async {
        let! msg = inbox.Receive()
        match msg with
        | GetMarketState(c, reply) ->  reply.Reply(MarketState.Open)
                                       return! marketIsOpen stocks stockTicker

        | GetAllStocks(c, reply) ->    reply.Reply(stocks |> Seq.toList)
                                       return! marketIsOpen stocks stockTicker

        | StockTickerMessage.UpdateStock(stock) ->

            let isStockChanged = updateStocks stock stocks
            match isStockChanged with
            | Some(updatedStocks) ->    tradingSupervisorAgent.OnUpdateStock(stock)
                                        return! marketIsOpen updatedStocks stockTicker
            | None -> return! marketIsOpen stocks stockTicker
```

StockTickerMarket.fs

```
| UpdateStockPrices ->
    for stock in stocks do
        inbox.Post(StockTickerMessage.UpdateStock(stock))

    return! marketIsOpen stocks stockTicker

| PublishCommand(connId, command) ->
    match command.Command with

    | TradingCommand.BuyStockCommand(connId, trading) ->
        let buy = Buy(connId, trading.Symbol,
            {Quantity=trading.Quantity; Price=trading.Price; TradingType = "Buy"})
        tradingSupervisorAgent.OnNotify(buy)

    | TradingCommand.SellStockCommand(connId, trading) ->
        let sell = Sell(connId, trading.Symbol,
            {Quantity=trading.Quantity; Price=trading.Price; TradingType = "Sell"})
        tradingSupervisorAgent.OnNotify(sell)

    | TradingCommand.AddStockCommand(connId, trading) ->
                        if not <| (stocks |> Seq.exists (fun s -> s.Symbol.ToUpper() =
                                                                  trading.Symbol.ToUpper())) then
                            let stock = Stock.CreateStock (trading.Symbol.ToUpper())
                                                          trading.Price (stocks.Count)
                            stocks.Add(stock)
                            tradingSupervisorAgent.OnNotify(AddStock(connId, stock))
    return! marketIsOpen stocks stockTicker
```

StockTickerMarket.fs

```fsharp
member x.GetAllStocks(connId) =
    stockAgent.PostAndReply(fun ch -> GetAllStocks(connId, ch))

member x.GetMarketState(connId) =
    stockAgent.PostAndReply(fun ch -> GetMarketState(connId, ch))

member x.OpenMarket(connId) =
    stockAgent.Post(OpenMarket(connId))

member x.CloseMarket(connId) =
    stockAgent.Post(CloseMarket(connId))

member x.UpdateStock(stock:Stock) =
        stockAgent.Post(StockTickerMessage.UpdateStock(stock))

member x.PublishCommand(connId, order:CommandWrapper) =
                stockAgent.Post(PublishCommand(connId, order))

member x.Subscribe(connId, initialAmount,
                   caller:IHubCallerConnectionContext<IStockTickerHubClient>) =
        tradingSupervisorAgent.Subscribe(connId, initialAmount, caller) |> ignore

member x.Unsubscribe(connId) =
        tradingSupervisorAgent.Unsubscribe(connId)
```

TradingSupervisorAgent.fs

```
type TradingSupervisorMessage =
    | Subscribe of  id : string * initialAmount : float *
                    caller:IHubCallerConnectionContext<IStockTickerHubClient>
    | Unsubscribe of   id : string
    | Buy of id : string * Ticker : string * TradingDetails
    | Sell of id : string * Ticker : string * TradingDetails
    | AddStock of connId:string * stock:Stock
```

```
// Subscribes and Unsubscribes TradingAgent
// Uses a mix of RX and Agent.Post just for demo purpose
// (TradingAgent : IOboservable) and (TradingSuperviserAgent : IObservable)
type internal TradingSupervisorAgent() =

    let subject = new Subject<TradingMessage>()

    let agent =
        Agent<TradingSupervisorMessage>.Start(fun inbox ->
```

TradingSupervisorAgent.fs

```
let rec loop n (agents : Map<string, (IObserver<TradingMessage> * IDisposable)>) =
    async {
        let! msg = inbox.Receive()

        match msg with
        | Subscribe(id, initialAmount, caller) ->
            let agent = Map.tryFind id agents

            match agent with
            | Some(a) -> return! loop n agents
            | None ->
                let observerAgent = new TradingAgent(id, initialAmount, caller)

                let dispObsrever = subject.Subscribe(observerAgent)
                observerAgent.Agent |> reportErrorsTo id supervisor |> startAgent

                caller.Client(id).SetInitialAsset(initialAmount)
                return! loop (n + 1) (Map.add id
                    (observerAgent :> IObserver<TradingMessage>, dispObsrever) agents)

        | Unsubscribe(id) ->
            let agent = Map.tryFind id agents
            match agent with
            | Some(a, d) ->     a.OnCompleted()
                                d.Dispose() // Agent dispsed
                                return! loop (n - 1) (Map.remove id agents)
            | None -> return! loop n agents
```

TradingSupervisorAgent.fs

```
| Buy(id, symbol, trading) ->
    let agent = Map.tryFind id agents
    match agent with
    | Some(a, _) ->
        a.OnNext(TradingMessage.Buy(symbol, trading))
        return! loop n agents
    | None -> return! loop n agents

| Sell(id, symbol, trading) ->
    let agent = Map.tryFind id agents
    match agent with
    | Some(a, _) ->
        a.OnNext(TradingMessage.Sell(symbol, trading))
        return! loop n agents
    | None -> return! loop n agents


| AddStock(id, stock) ->
    let agent = Map.tryFind id agents
    match agent with
    | Some(a, _) ->
        a.OnNext(TradingMessage.AddStock(stock))
        return! loop n agents
    | None -> return! loop n agents
    }
```

TradingSupervisorAgent.fs

```fsharp
// exposes and IObservable
let obs = subject :> IObservable<TradingMessage>

member this.OnUpdateStock(stock:Stock) =
    // in this case the OnNext is bypassing the local Agent
    // and all the clients will be notify
    subject.OnNext(TradingMessage.UpdateStock(stock))

member this.OnNotify(msg:TradingSupervisorMessage) =
        agent.Post(msg)

member x.Subscribe(id : string, initialAmount : float,
                   caller:IHubCallerConnectionContext<IStockTickerHubClient>) =
    agent.Post(Subscribe(id, initialAmount, caller))

member x.Unsubscribe(id : string) = agent.Post(Unsubscribe(id))

interface IObservable<TradingMessage> with
    member x.Subscribe(obs) =
        subject.Subscribe(obs)
```

TradingAgent.fs

```fsharp
// Agent per Subscriber
// Keeps track orders and portfolio
type TradingAgent(id : string, initialAmount : float,
                  caller:IHubCallerConnectionContext<IStockTickerHubClient>) =

    let connId = id
    let caller = caller
```

```fsharp
let agent =
    new Agent<TradingMessage>(fun inbox ->
        // single thread safe no sharing
        let rec loop cash (portfolio : Portfolio) (buyOrders : Treads) (sellOrders : Treads)
            async {
                let! msg = inbox.Receive()
                match msg with
                | Kill(reply) -> reply.Reply()
                | Error(exn) -> raise exn
```

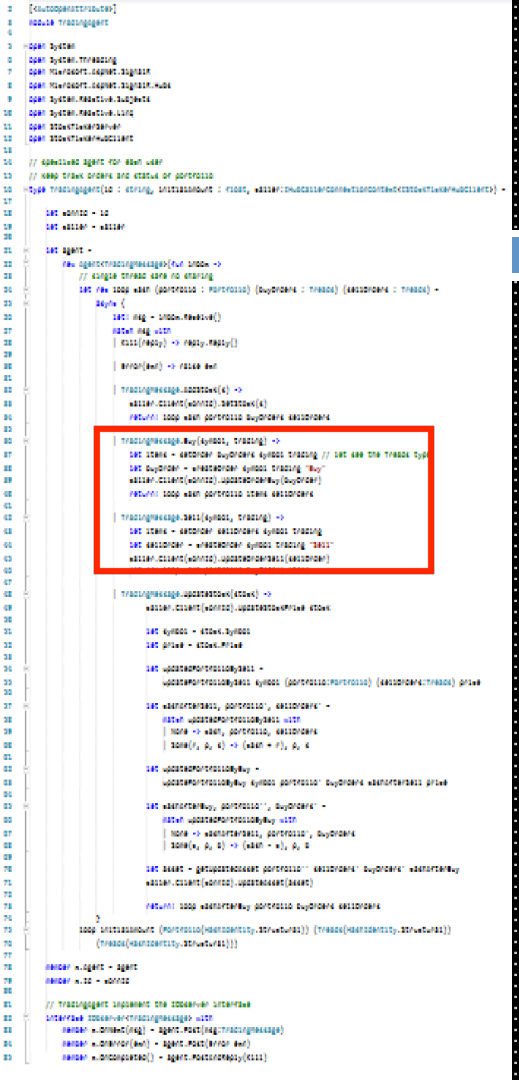
TradingAgent.fs

```
let agent =
    new Agent<TradingMessage>(fun inbox ->
        // single thread safe no sharing
        let rec loop cash (portfolio : Portfolio) (buyOrders : Treads) (sellOrders : Treads)
            async {
                let! msg = inbox.Receive()
                match msg with
                | Kill(reply) -> reply.Reply()
                | Error(exn) -> raise exn
```

```
type TradingMessage =
    | Kill of AsyncReplyChannel<unit>
    | Error of exn
    | Buy of symbol : string * TradingDetails
    | Sell of symbol : string * TradingDetails
    | UpdateStock of Stock
    | AddStock of Stock

and TradingDetails =
    { Quantity : int
      Price : float
      TradingType:string }
```

```
and Treads = Dictionary<string, ResizeArray<TradingDetails>>

and Portfolio = Dictionary<string, TradingDetails>
```

TradingAgent.fs



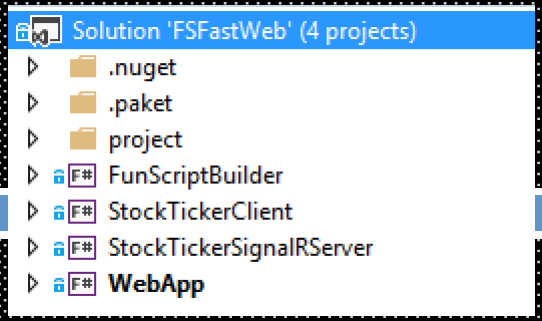
```
| TradingMessage.Buy(symbol, trading) ->
    let items = setOrder buyOrders symbol trading
    let buyOrder = createOrder symbol trading "Buy"
    caller.Client(connId).UpdateOrderBuy(buyOrder)
    return! loop cash portfolio items sellOrders

| TradingMessage.Sell(symbol, trading) ->
    let items = setOrder sellOrders symbol trading
    let sellOrder = createOrder symbol trading "Sell"
    caller.Client(connId).UpdateOrderSell(sellOrder)
    return! loop cash portfolio buyOrders items
```

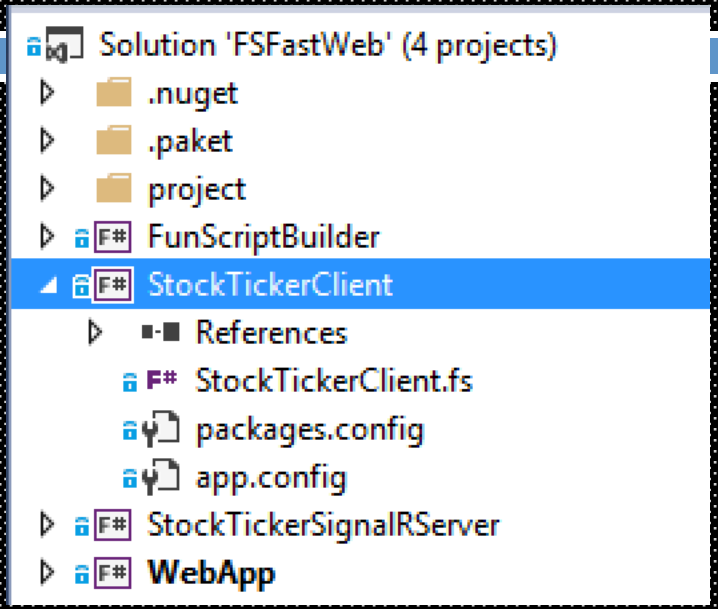
TradingAgent.fs

```fsharp
// Treads -> string -> TradingDetails -> Treads
let setOrder (orders : Treads) symbol trading =
    let items = orders |> tryGetValues symbol
    match items with
    | Some(items) ->
        let index =
            items
            |> Seq.tryFindIndex
                    (fun p -> p.Price = trading.Price)
        match index with
        | Some(i) ->
            orders.[symbol].[i] <-
                    { items.[i]
                        with Quantity = (items.[i].Quantity + trading.Quantity) }
        | None ->
            orders.[symbol].Add({ Quantity = trading.Quantity
                                  Price = trading.Price
                                  TradingType = trading.TradingType })
    | None ->
        let treads = ResizeArray<TradingDetails>([ trading ])
        orders.Add(symbol, treads)
    orders
```

TradingAgent.fs

```
| TradingMessage.UpdateStock(stock) ->
        caller.Client(connId).UpdateStockPrice stock

        let symbol = stock.Symbol
        let price = stock.Price

        let updatedPortfolioBySell =
            updatePortfolioBySell symbol (portfolio:Portfolio) (sellOrders:Treads) price

        let cashAfterSell, portfolio', sellOrders' =
            match updatedPortfolioBySell with
            | None -> cash, portfolio, sellOrders
            | Some(r, p, s) -> (cash + r), p, s

        let updatedPortfolioByBuy =
            updatePortfolioByBuy symbol portfolio' buyOrders cashAfterSell price

        let cashAfterBuy, portfolio'', buyOrders' =
            match updatedPortfolioByBuy with
            | None -> cashAfterSell, portfolio', buyOrders
            | Some(c, p, b) -> (cash - c), p, b

        let asset = getUpdatedAsset portfolio'' sellOrders' buyOrders' cashAfterBuy
        caller.Client(connId).UpdateAsset(asset)

        return! loop cashAfterBuy portfolio buyOrders sellOrders
```


Solution 'FSFastWeb' (4 projects)
.nuget
.paket
project
FunScriptBuilder
StockTickerClient
StockTickerSignalRServer
WebApp


Solution 'FSFastWeb' (4 projects)
.nuget
.paket
project
FunScriptBuilder
StockTickerClient
References
StockTickerClient.fs
packages.config
app.config
StockTickerSignalRServer
WebApp

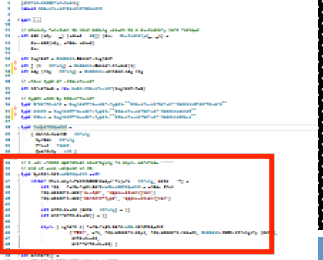
StockTickerClient.fs

```
[<ReflectedDefinition>]
module StockTickerClientFSClient

open FunScript.TypeScript
open FunScript
open SignalRProvider
open System.Collections.Generic
open FunScript.TypeScript
open FunScript.HTML
```

```
let signalR = Globals.Dollar.signalR
let j (s : string) = Globals.Dollar.Invoke(s)
let log (msg : string) = Globals.console.log msg

// check type of .stockTicker
let serverHub = new Hubs.stockTicker(signalR.hub)

// types used by Stock-Ticker
type OrderRecord = SignalRProvider.Types.``StockTickerServer!Models+OrderRecord``
type Asset = SignalRProvider.Types.``StockTickerServer!Models+Asset``
type Stock = SignalRProvider.Types.``StockTickerServer!Models+Stock``
```

```
type TikcerRequest =
    { ConnectionID : string
      Symbol : string
      Price : float
      Quantity : int }
```

StockTickerClient.fs



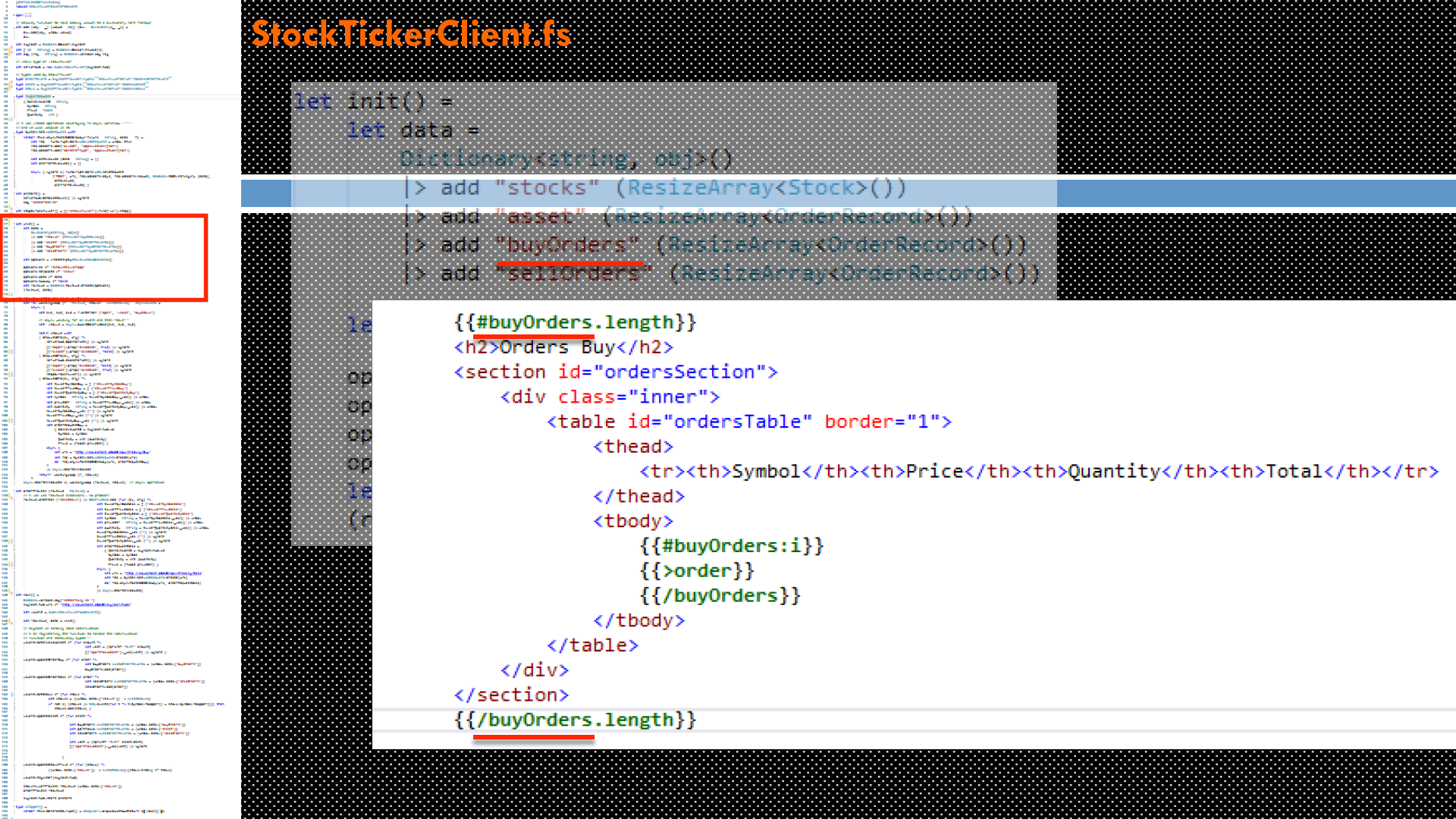
```
type System.Net.WebRequest with
    member this.AsyncPostJSONOneWay<'T>(url : string, data : 'T) =
        let req : FunScript.Core.Web.WebRequest = unbox this
        req.Headers.Add("Accept", "application/json")
        req.Headers.Add("Content-Type", "application/json")

        let onReceived (data : string) = ()
        let onErrorReceived() = ()

        async { ignore <| FunScript.Core.Web.sendRequest
                    ("POST", url, req.Headers.Keys,
                     req.Headers.Values,
                     Globals.JSON.stringify (data),
                     onReceived,
                     onErrorReceived) }
```

StockTickerClient.fs

```
let onstart() =
    serverHub.GetAllStocks() |> ignore
    log "##Started!##"

let stopScrollTicker() = j("#stockTicker").find("ul").stop()
let startScrollTicker() = j("#stockTicker").find("ul").scroll()
```

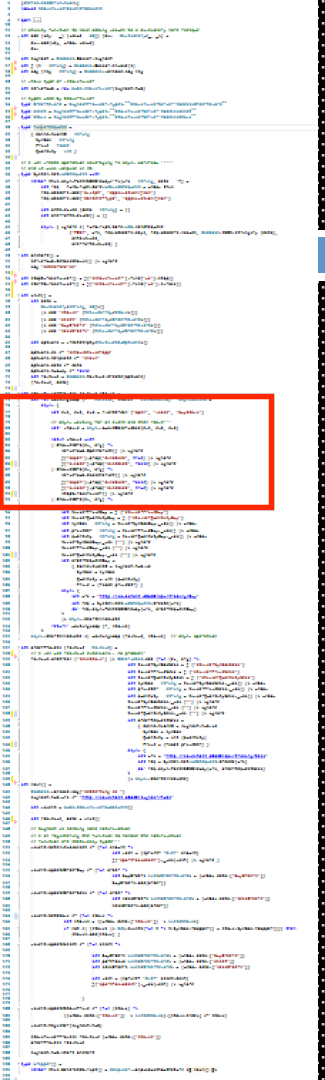
StockTickerClient.fs

```
let init() =
    let data =
        Dictionary<string, obj>()
        |> add "stocks" (ResizeArray<Stock>())
        |> add "asset" (ResizeArray<OrderRecord>())
        |> add "buyOrders" (ResizeArray<OrderRecord>())
        |> add "sellOrders" (ResizeArray<OrderRecord>())

    let options = createEmpty<RactiveNewOptions>()

    options.el <- "#stockStickerApp"
    options.template <- "#main"
    options.data <- data
    options.twoway <- false
    let ractive = Globals.Ractive.Create(options)
    (ractive, data)
```


StockTickerClient.fs
let init() =
let data =
Dictionary<string, obj>()
|> add "stocks" (ResizeArray<Stock>())
|> add "asset" (ResizeArray<OrderRecord>())
|> add "buyOrders" (ResizeArray<OrderRecord>())
|> add "sellorders" (ResizeArray<OrderRecord>())
{{#buyOrders.length}}
<h2>Orders Buy</h2>
<section id="ordersSection">

<table id="ordersTable" border="1">
<thead>
<tr><th>Symbol</th><th>Price</th><th>Quantity</th><th>Total</th></tr>
</thead>
<tbody>
{{#buyOrders:i}}
{{>order}}
{{/buyOrders}}
</tbody>
</table>

</section>
{{/buyOrders.length}}

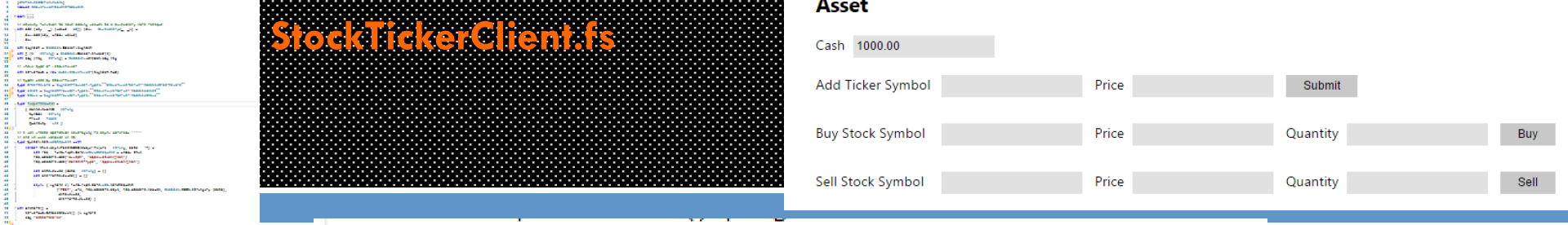


StockTickerClient.fs

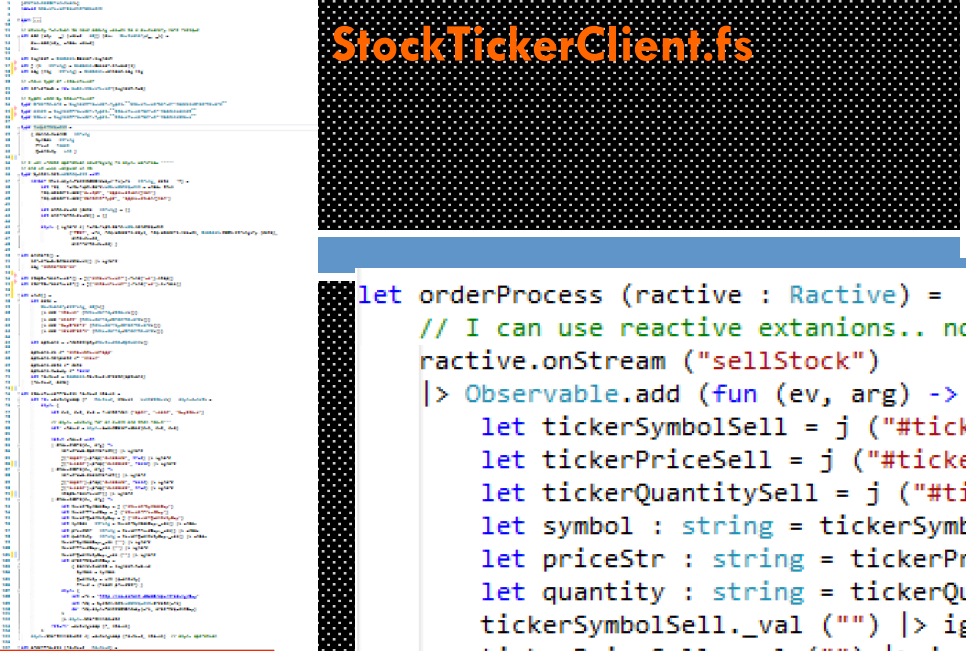
Live Stock Table

Open Market | Close Market

```
let stockTickerProcess ractive stocks =
    let rec waitingLoop (r : Ractive, stocks : List<Stock>) : Async<unit> =
        async {
            let ev1, ev2, ev3 = r.onStream ("open", "close", "buyStock")

            // Async waiting for an event and then react!!!
            let! choice = Async.AwaitObservable(ev1, ev2, ev3)

            match choice with
            | Choice1Of3(ev, arg) ->
                serverHub.OpenMarket() |> ignore
                j("#open").prop("disabled", true) |> ignore
                j("#close").prop("disabled", false) |> ignore
            | Choice2Of3(ev, arg) ->
                serverHub.CloseMarket() |> ignore
                j("#open").prop("disabled", false) |> ignore
                j("#close").prop("disabled", true) |> ignore
                stopScrollTicker() |> ignore
```




```
        | Choice3Of3(ev, arg) ->
            let tickerSymbolBuy = j ("#tickerSymbolBuy")
            let tickerPriceBuy = j ("#tickerPriceBuy")
            let tickerQuantityBuy = j ("#tickerQuantityBuy")
            let symbol : string = tickerSymbolBuy._val() |> unbox
            let priceStr : string = tickerPriceBuy._val() |> unbox
            let quantity : string = tickerQuantityBuy._val() |> unbox
            tickerSymbolBuy._val ("") |> ignore
            tickerPriceBuy._val ("") |> ignore
            tickerQuantityBuy._val ("") |> ignore
            let orderRequestBuy =
                { ConnectionID = signalR.hub.id
                  Symbol = symbol
                  Quantity = int (quantity)
                  Price = (float priceStr) }
            async {
                let url = "http://localhost:48430/api/trading/Buy"
                let req = System.Net.WebRequest.Create(url)
                do! req.AsyncPostJSONOneWay(url, orderRequestBuy)
            }
            |> Async.StartImmediate
        return! waitingLoop (r, stocks)
    }
Async.StartImmediate <| waitingLoop (ractive, stocks)  // Async operation
```



StockTickerClient.fs

Asset

Cash 1000.00

Add Ticker Symbol | Price | Submit

Buy Stock Symbol | Price | Quantity | Buy

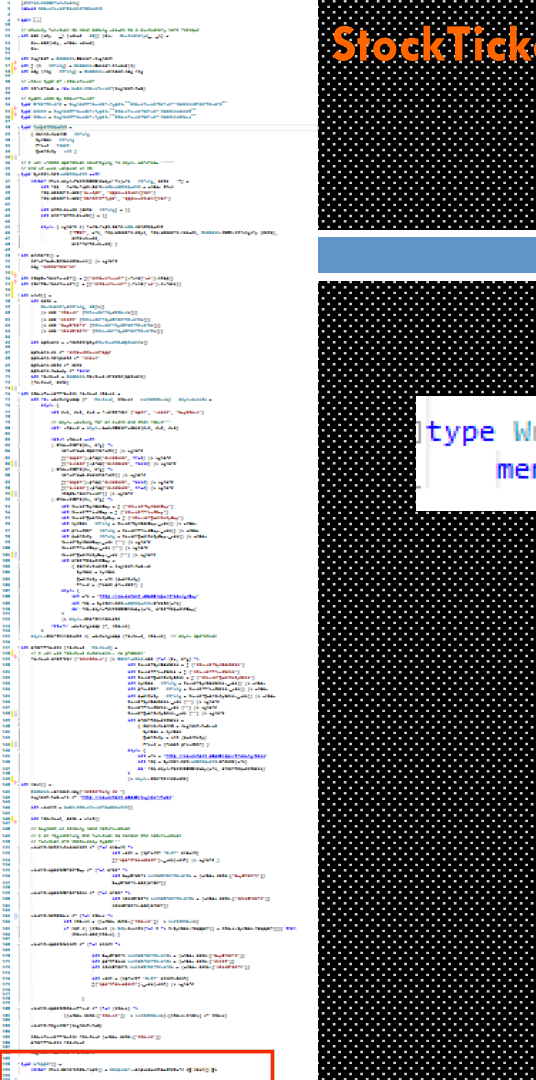
Sell Stock Symbol | Price | Quantity | Sell

```
let orderProcess (ractive : Ractive) =
    // I can use reactive extanions.. no problem!
    ractive.onStream ("sellStock")
    |> Observable.add (fun (ev, arg) ->
        let tickerSymbolSell = j ("#tickerSymbolSell")
        let tickerPriceSell = j ("#tickerPriceSell")
        let tickerQuantitySell = j ("#tickerQuantitySell")
        let symbol : string = tickerSymbolSell._val() |> unbox
        let priceStr : string = tickerPriceSell._val() |> unbox
        let quantity : string = tickerQuantitySell._val() |> unbox
        tickerSymbolSell._val ("") |> ignore
        tickerPriceSell._val ("") |> ignore
        tickerQuantitySell._val ("") |> ignore
        let orderRequestSell =
            {   ConnectionID = signalR.hub.id
                Symbol = symbol
                Quantity = int (quantity)
                Price = (float priceStr) }
        async {
            let url = "http://localhost:48430/api/trading/Sell"
            let req = System.Net.WebRequest.Create(url)
            do! req.AsyncPostJSONOneWay(url, orderRequestSell)
        }
        |> Async.StartImmediate)
```

StockTickerClient.fs

```fsharp
let main() =
    Globals.console.log("##Starting:## ")
    signalR.hub.url <- "http://localhost:48430/signalr/hubs"

    let client = Hubs.StockTickerHubClient()

    let reactive, data = init()

    // SignalR is sending some notification
    // Statically typed!!!
    client.SetInitialAsset <- (fun amount ->
                                    let cash = (sprintf "%.2f" amount)
                                    j("#portfolioCash")._val(cash) |> ignore )

    client.UpdateOrderBuy <- (fun order ->
                                    let buyOrders:List<OrderRecord> = (unbox data.["buyOrders"])
                                    buyOrders.Add(order))

    client.UpdateOrderSell <- (fun order ->
                                    let sellOrders:List<OrderRecord> = (unbox data.["sellOrders"])
                                    sellOrders.Add(order))
```

StockTickerClient.fs



```
type Wrapper() =
    member this.GenerateScript() = Compiler.compileWithoutReturn <@ main() @>
```

Summary

F# is a *great and mature language* for Web Development

F# has built in features to develop Fast and Scalable Web App

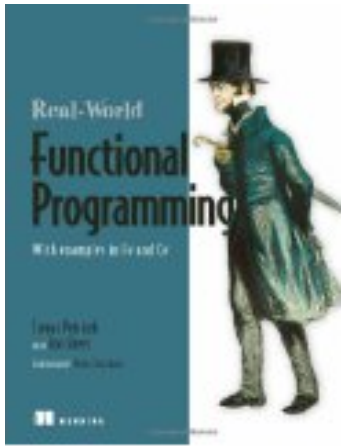
F# |> RX |> SignalR |> CQRS |> 

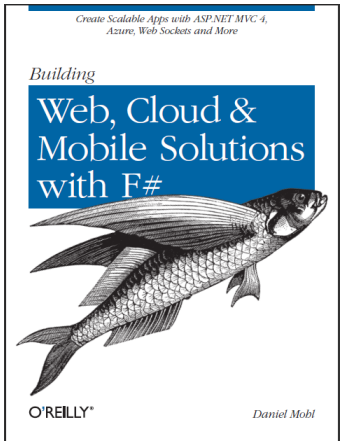
The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra

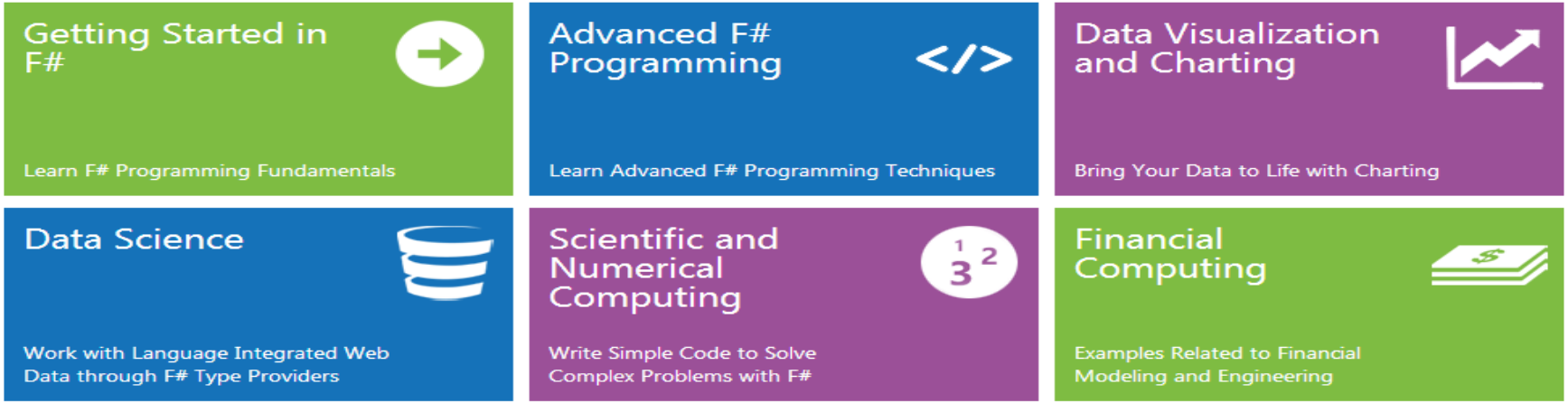
References

- https://wizardsofsmart.wordpress.com
- http://www.todobackend.com
- http://funscript.info





Online resources



- www.fsharp.org — Information & community
  www.tryfsharp.org — Interactive F# tutorials

How to reach me

https://github.com/rikace/FS-SignalR

http://meetup.com/DC-fsharp

@DCFsharp @TRikace

rterrell@microsoft.com


That's all Folks!