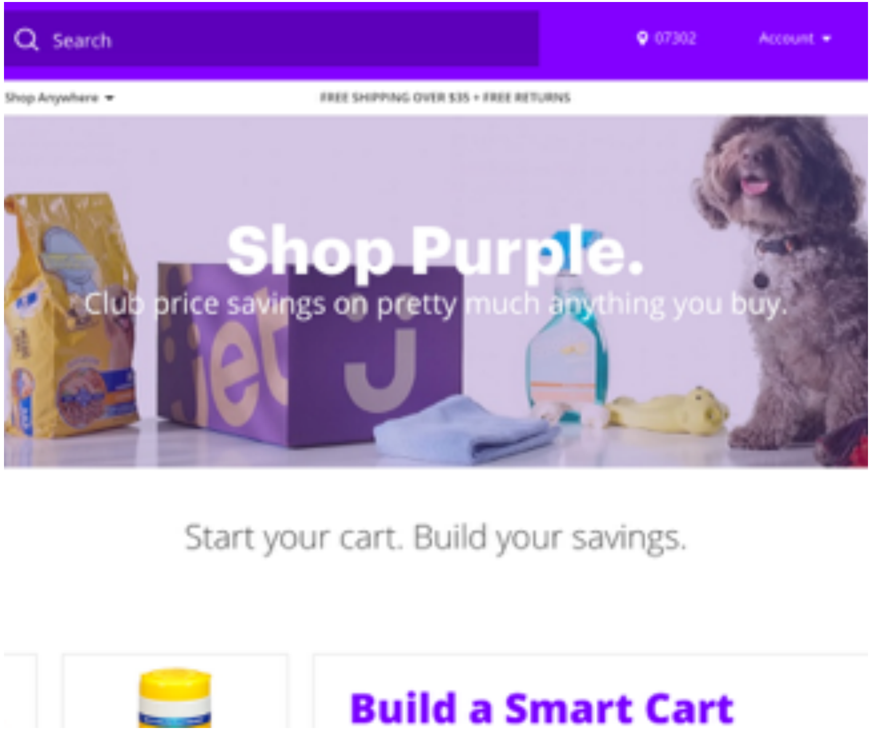
# Building reactive services using functional programming

Rachel Reese | @rachelreese | rachelree.se
Jet Technology | @JetTechnology | tech.jet.com

# Taking on Amazon!




## Launched July 22

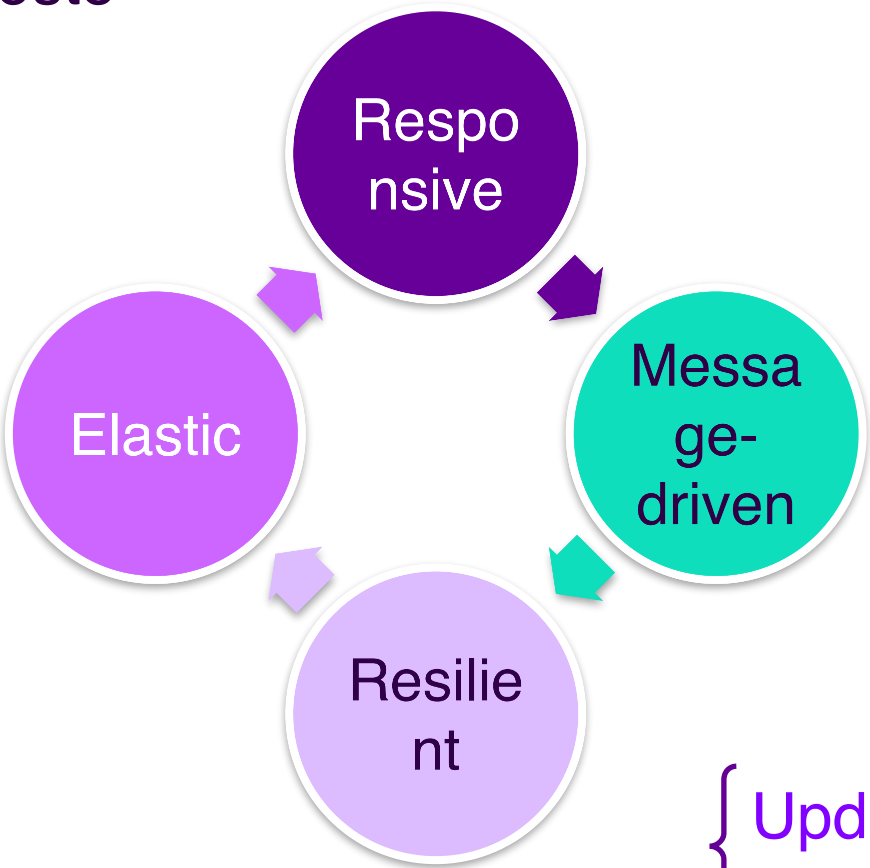
- Both Apple & Android named our app as one of their tops for 2015
- Over 20k orders per day
- Over 10.4 million SKUs
- 600k first-time buyers
- 500k mobile downloads

**We're hiring!**

http://jet.com/about-us/working-at-jet

| | | | | | | |
|---|---|---|---|---|---|---|
| Azure | Web sites | Cloud services | VMs | Service bus queues | Services bus topics | Blob storage |
| Table storage | Queues | Hadoop | DNS | Active directory | SQL Azure | R |
| F# | Paket | FSharp.Data | Chessie | Unquote | SQLProvider | Python |
| Deedle | FAK E | FSharp.Async | FsBlog | Node | Angular | SAS |
| Storm | Elastic Search | EventStore | Microservices | Consul | Kafka | PDW |
| Splunk | Redis | SQL | Puppet | Jenkins | Apache Hive | Apache Tez |

# Reactive manifesto




Updated manifesto!
Sept 16, 2014

Responsive

# Responsive

The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behavior in turn simplifies error handling, builds end user confidence, and encourages further interaction.

# Events

# Events in F#

Events

| Immutable | Implement IObservable | First-class | Composable |
|---|---|---|---|

# First class! Composable!

```
let register ev =
   ev
   |> Event.map    (fun _ -> DateTime.Now)
   |> Event.scan   (fun (_, currentStamp : DateTime) lastStamp ->
                       if ((lastStamp - currentStamp).TotalSeconds >
2.0)
                       then (4, lastStamp)
                       else (1, currentStamp))
                   (0, DateTime.Now)
   |> Event.map    fst
   |> Event.scan   (+)
   |> Event.map    (sprintf "Clicks: %d")
   |> Event.add    lbl.set_Text
```

# Event combinators

| | | | |
|---|---|---|---|
| Add | Choose | Filter | Map |
| Scan | Partition | Pairwise | Merge |
| Split | | | |

```
let using f event =
  event |> Event.scan (fun state input ->
  if state <> Unchecked.defaultof<_> then
     (state :> IDisposable).Dispose()
  f input) Unchecked.defaultof<_>
```

```
colors
|> Event.using (fun clr -> new SolidBrush(clr))
|> Event.add (fun br -> frm.BackBrush <- br)
```

# Message-driven

Message-driven

Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

# Async workflows

# Async vs. concurrent vs. parallel

## Asynchronous

- Non-blocking, specifically in reference to I/O operations (not necessarily parallel, can be sequential).

## Concurrent

- Multiple operations happening at the same time (not necessarily in parallel).

## Parallel

- Multiple operations processed simultaneously.

# Computation expressions

- A single syntactic mechanism which gives a nice syntax for multiple abstract computation types.
- Monads, yes

- But also:
- Monoids
- Additive monads*
- Computations constructed using monad transformers*

# Async workflows

Reads similarly to classic, linear synchronous code

Easy to convert

Easy to understand and reason about.

# Consider classic synchronous code

```
let webClient = new WebClient()
let data = webClient.DownloadData(uri)
outputStream.Write(data, 0, data.Length)
```

```
> Read 53813 characters for Google
Read 1020 characters for Microsoft.com
Read 25816 characters for MSDN
```

```
let processingAgent() =
  Agent<string * string>.Start(fun inbox ->
    async { while true do
             let  name,url = inbox.Receive()
             let uri = new System.Uri(url)
             let webClient = new WebClient()
             let! html = webClient.AsyncDownloadString(uri)
             printfn "Read %d characters for %s" html.Length name })
```

# Async workflow

```
let rec loop count =
    async {
        let! ev = Async.AwaitEvent lbl.MouseDown
        lbl.Text <- sprintf "Clicks: %d" count
        do! Async.Sleep 1000
        return! loop <| count + 1
    }

let start = Async.StartImmediate <| loop 1
```

# The actor model

The actor model is a model of concurrent computation using actors which is characterized by:

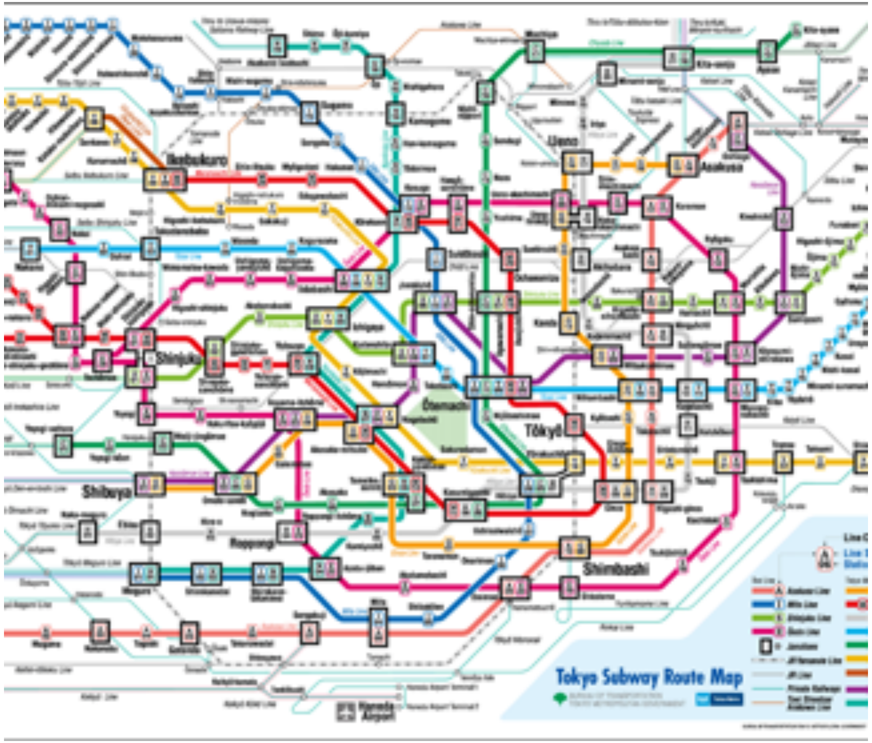
Dynamic creation of actors → Inclusion of actor addresses in messages → Interaction only through direct asynchronous message passing → → No restriction on message arrival order.

# What is an actor?

An independent computational entity which contains a queue, and receives and processes messages.

# F# Agents

Actor = Agent = MailboxProcessor



# Agents: basic syntax

```
let alloftheagents =
    [ for i in 0 .. 100000 ->
        MailboxProcessor<string>.Start(fun inbox ->
            async { while true do
                    let! msg = inbox.Receive()
                    if i % 10000 = 0 then
                        printfn "agent %d got message '%s'" i
msg })]

for agent in alloftheagents do
    agent.Post "ping!"
```

```
> agent 0 got message 'ping!'
agent 10000 got message 'ping!'
agent 20000 got message 'ping!'
agent 30000 got message 'ping!'
agent 40000 got message 'ping!'
agent 50000 got message 'ping!'
agent 60000 got message 'ping!'
agent 70000 got message 'ping!'
agent 80000 got message 'ping!'
agent 90000 got message 'ping!'
agent 100000 got message 'ping!'
```

# Getting replies from an agent

```
let agent =
  MailboxProcessor<string * AsyncReplyChannel<string>>.Start(fun
inbox ->
   let rec loop () =
      async {
         let! (message, replyChannel) = inbox.Receive()
         replyChannel.Reply(String.Format("Received message: {0}",
message))
         do! loop ()
      }
   loop ())


let messageAsync = agent.PostAndAsyncReply(fun rc-> input, rc)
```

# Scanning an agent's queue

```
let inprogressAgent = new MailboxProcessor<Job>(fun _ -> async { ()
})

let completeAgent = MailboxProcessor<Message>.Start(fun inbox ->
        let rec loop n =
            async {
                let! (id, result) = inbox.Receive()
                printfn "The result of job #%d is %f" id result
                do! loop <| n + 1
            }
        loop 0)
```

# Scanning an agent’s queue

```
let cancelJob(cancelId) =
        Async.RunSynchronously(
            inprogressAgent.Scan(fun (jobId, result, source) ->
                let action =
                    async {
                        printfn "Canceling job #%d" cancelId
                        source.Cancel()
                    }

                if (jobId = cancelId) then
                    Some(action)
                else
                    None))
```

# One major difference from Erlang

F# agents do not work across process boundaries, only within the same process.

→ Enter Cricket (previously FSharp.Actor)

# Cricket

```
let greeter =
    actor {
        name "greeter"
        body (
            let rec loop() = messageHandler {
                let! msg = Message.receive()

                match msg with
                | Hello ->  printfn "Hello"
                | HelloWorld -> printfn "Hello World"
                | Name name -> printfn "Hello, %s" name
                return! loop()
            }
            loop())
    } |> Actor.spawn
```

```
greeter <-- Name("From F# Actor")
```

# Cricket remoting

```
let System =
  ActorHost.Start()
           .SubscribeEvents(fun (evt:ActorEvent) -> printfn "%A" evt)
           .EnableRemoting(
             [new 
TCPTransport(TcpConfig.Default(IPEndPoint.Create(12002)))],
             new TcpActorRegistryTransport(
               TcpConfig.Default(IPEndPoint.Create(12003))),
             new UdpActorRegistryDiscovery(UdpConfig.Default(), 1000)
  )
```

# Cricket remoting

```
let ping count =
  actor {
    name "ping"
    body (
      let pong = !~ "pong"
      let rec loop count =
        messageHandler {
          let! msg = Message.receive()
          match msg with
          | Pong when count > 0 ->
            if count % 1000 = 0 then printfn "Ping: ping %d" count
              do! Message.post pong.Value Ping
              return! loop (count - 1)
          | Ping -> failwithf "Ping: received a ping message, panic..."
          | _ -> do! Message.post pong.Value Stop
        }
      loop count )}
```

# Resilient

The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

# Error handling

Tasks that are run using Async.RunSynchronously report failures back to the controlling thread as an exception. Use Async.Catch or try/catch to handle.

Async.StartWithContinuations has an exception continuation.

Supervisor pattern.

# Async.Catch

```
// Start these next two operations asynchronously, forcing an exception due
// to trying to access the file twice simultaneously.
Async.Start(readFile filename numBytes)
let result2 = writeToFile filename numBytes
             |> Async.Catch
             |> Async.RunSynchronously
match result2 with
| Choice1Of2 buffer -> printfn "Successfully read from file."
| Choice2Of2 exn ->
    printfn "Exception occurred reading from file %s: %s" filename (exn.Message)
```

```
Reading from file BigFile.dat.
val it : unit = ()
> Exception occurred reading from file BigFile.dat: The process cannot access the file
val it : unit = ()
>
```

# Async.StartWithContinuations

```
//PostAndReply blocks
let messageAsync = agent.PostAndAsyncReply(fun replyChannel -> input, replyChannel)

Async.StartWithContinuations(messageAsync,
        (fun reply -> printfn "Reply received: %s" reply), //continuation
        (fun _ -> ()), //exception
        (fun _ -> ())) //cancellation
```

Getting replies from an agent

# Async.StartWithContinuations

```
// This agent starts each job in the order in which it is received.
let runAgent = MailboxProcessor<Job>.Start(fun inbox ->
    let rec loop () =
        async {
            let! (id, job, source) = inbox.Receive()
            printfn "Starting job #%d" id

            // Post to the in-progress queue.
            inprogressAgent.Post(id, job, source)
            // Start the job.
            Async.StartWithContinuations(job,
                (fun result -> completeAgent.Post(id, result)),
                (fun _ -> ()),
                (fun cancelException -> printfn "Canceled job #%d" id),
                source.Token)
            do! loop ()
            }
    loop ())
```

Scanning an agent

# Supervisors

```
// Fail if agent ID contains "99".
module Supervisors =
    type Agent<'T> = MailboxProcessor<'T>
    // error handling
    let errorAgent =
        Agent<int * System.Exception>.Start(fun inbox ->
            async { while true do
                    let! (agentId, err) = inbox.Receive()
                    printfn "an error '%s' occurred in agent %d" err.Message agentId })

    let agents =
        [ for agentId in 0 .. 10000 ->
            let agent =
                new Agent<string>(fun inbox ->
                    async { while true do
                                let! msg = inbox.Receive()
                                if msg.Contains("agent 99") then
                                    failwith "fail!" })
            agent.Error.Add(fun error -> errorAgent.Post (agentId,error))
            agent.Start()
            (agentId, agent) ]

    for (agentId, agent) in agents do
        agent.Post (sprintf "message to agent %d" agentId )
```

# Supervisors

```
/// ERROR QUEUE
let rec errorAgent = Agent.Start(fun errorInbox ->
    let rec loop () = async {
        errorInbox.Scan(fun (message:string, number) ->
            let replaceAn = setEmail (fun x -> x.Replace("an", "a"))
            let actionAn = async {
                Data.GetData
                    |> Array.choose (fun x -> if (number = getId x) then Some(x) else None)
                    |> Array.map replaceAn // because "failure" is simulated by checking for 'an'
                    |> (filterAgent:Agent<Message array>).Post }
            let actionEn = async { printfn "Message %d failed permanently." number }

            if (message.Contains "Cannot send") then
                printfn "Retrying failure: %s" message
                Some(actionAn)
            else if (message.Contains "Templating") then
                Some(actionEn)
            else
                None
            ) |> Async.RunSynchronously
        return! loop ()
        }
    loop ())
```

# Elastic

The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

# Scaling agents on demand

```
type Agent<'T> = MailboxProcessor<'T>

let urlList = [ ("Microsoft.com", "http://www.microsoft.com/");
                ("MSDN", "http://msdn.microsoft.com/");
                ("Google", "http://www.google.com") ]

let processingAgent() = Agent<string * string>.Start(fun inbox ->
                        async { while true do
                                let! name,url = inbox.Receive()
                                let uri = new System.Uri(url)
                                let webClient = new WebClient()
                                let! html = webClient.AsyncDownloadString(uri)
                                printfn "Read %d characters for %s" html.Length name })

let scalingAgent : Agent<(string * string) list> = Agent.Start(fun inbox ->
                              async { while true do
                                      let! msg = inbox.Receive()
                                      msg
                                      |> List.iter (fun x ->
                                                    let newAgent = processingAgent()
                                                    newAgent.Post x )})

scalingAgent.Post urlList
```

# General resources & additional reading

## General resources

http://fsharp.org/

Twitter: #fsharp

F# channel on Functional Programming Slack

## Additional reading on F#

F# for Fun and Profit

F# Weekly

## Additional reading for F# MailboxProcessors & Async

Concurrency in F#

FSharp.Control.Reactive

Cricket

C# async gotchas

# Building reactive services using functional programming

Rachel Reese | @rachelreese | rachelree.se
Jet Technology | @JetTechnology | tech.jet.com