A Sane Approach to Modern Web Application Development

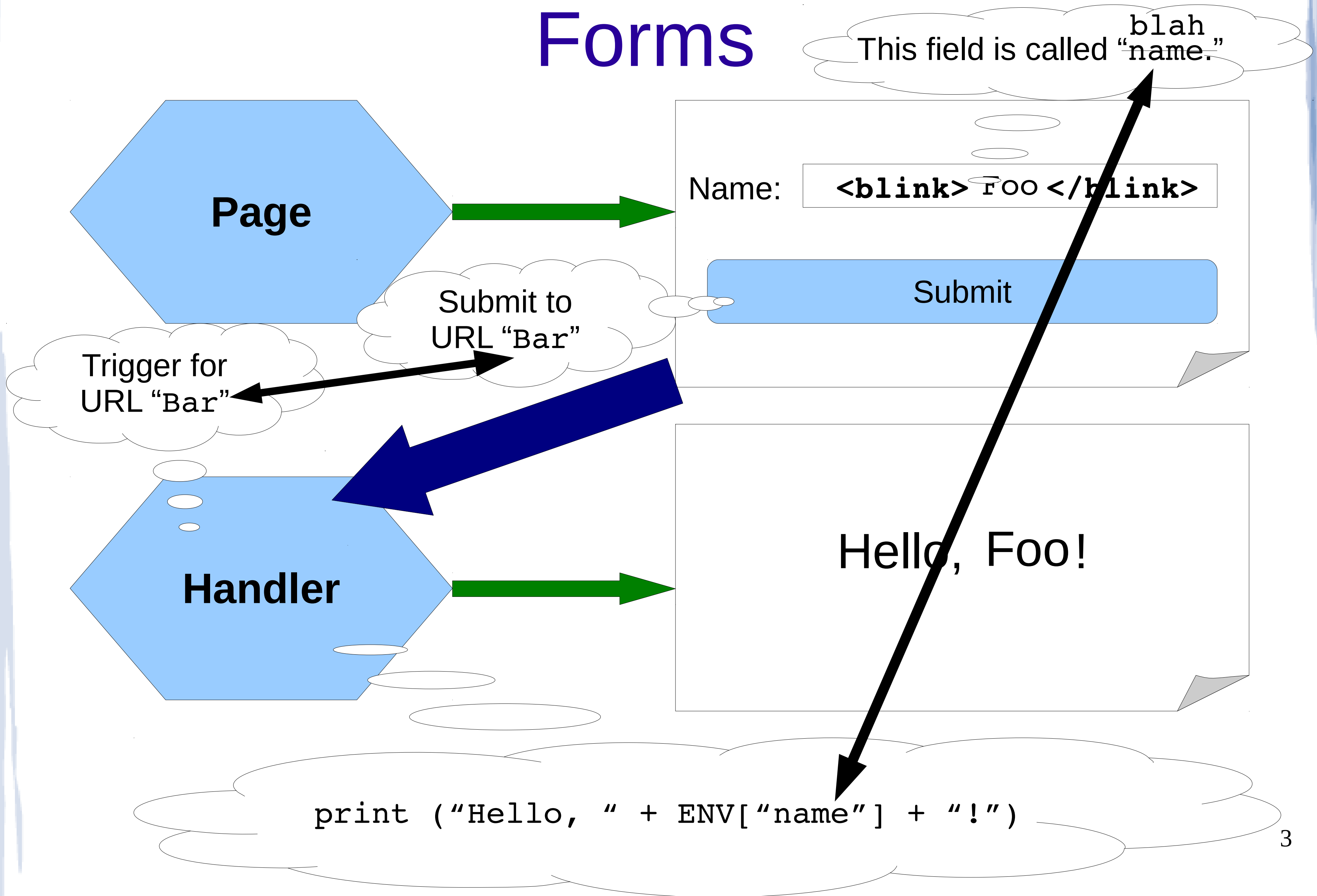
Adam Chlipala
February 22, 2010

Web 1.0

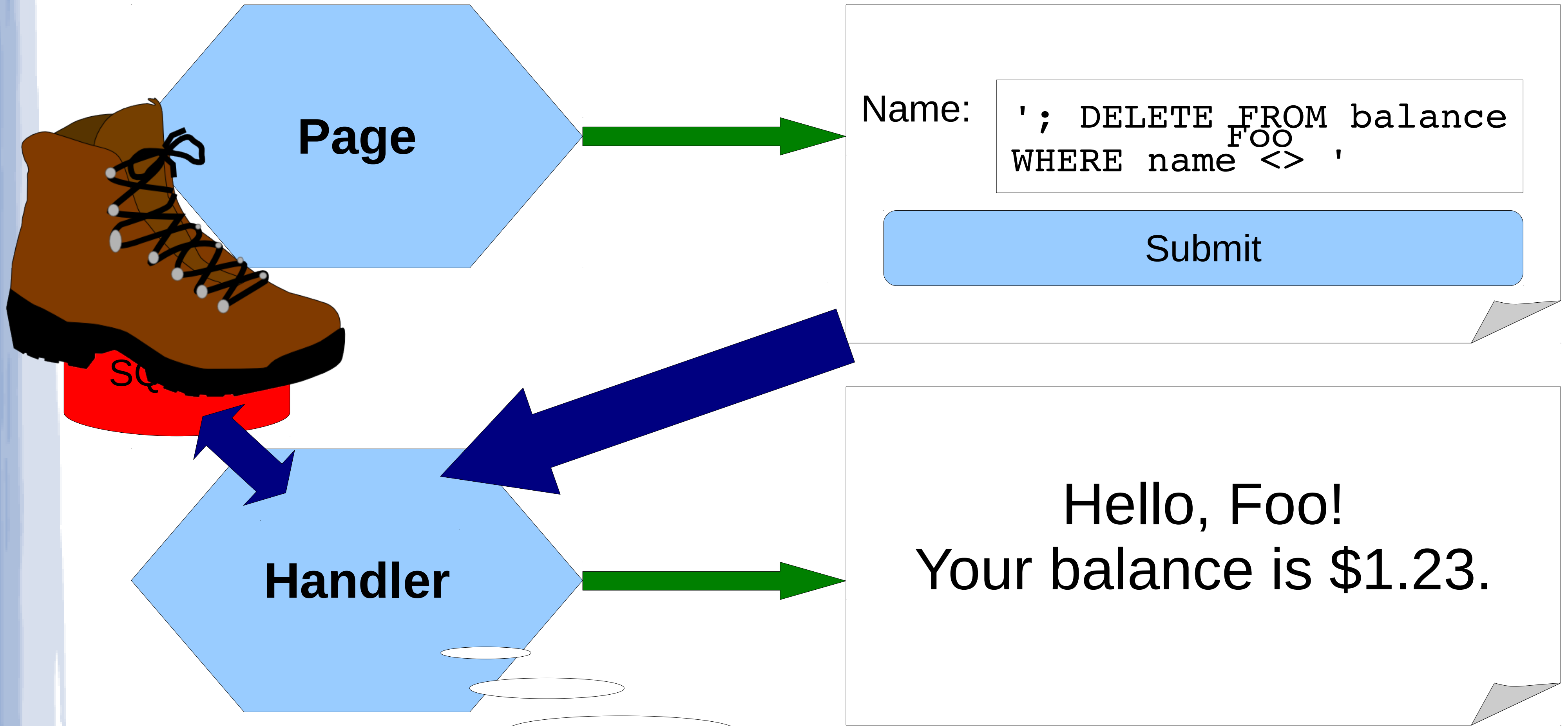
“Application”

```
<html><body>
  Hello world!
</body></html>
```

Forms



Database Access

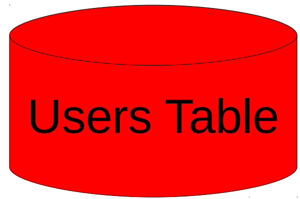


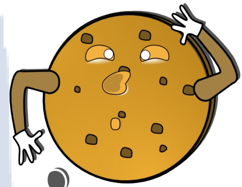

```
query ("SELECT balance FROM customers
WHERE name = '" + ENV["name"] + "'")
```

Saner Languages/Frameworks

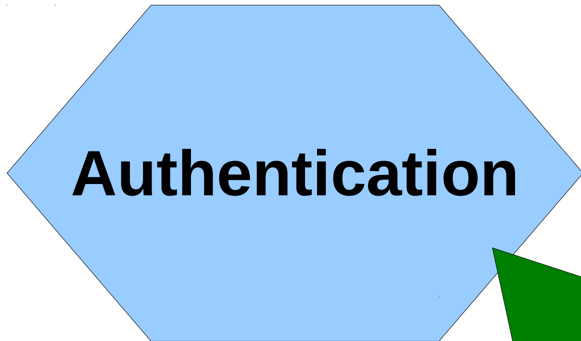
- Links
- Ocsigen
- LINQ
- WebSharper
- OPA (“One-Pot Application”)
- ...?

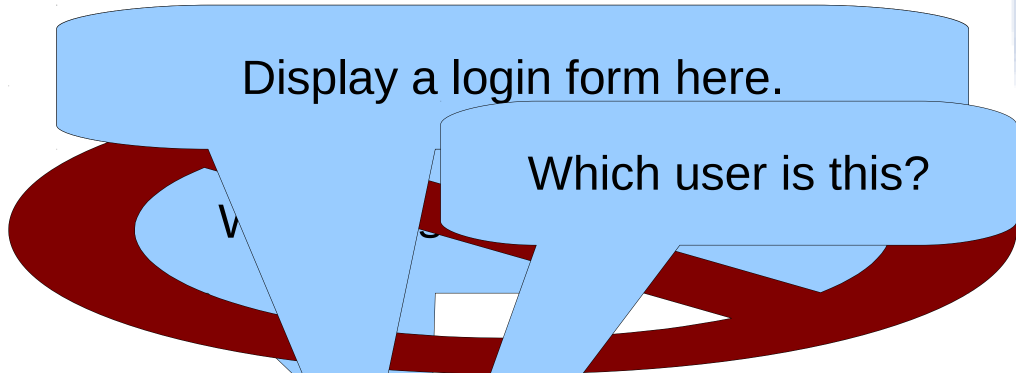
Web 1.0 Components

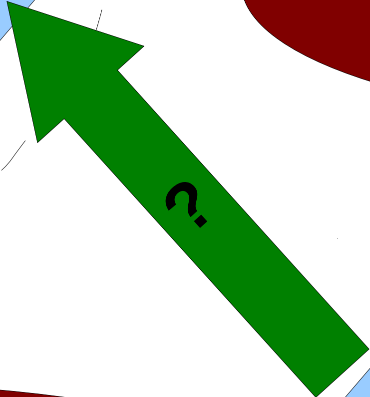




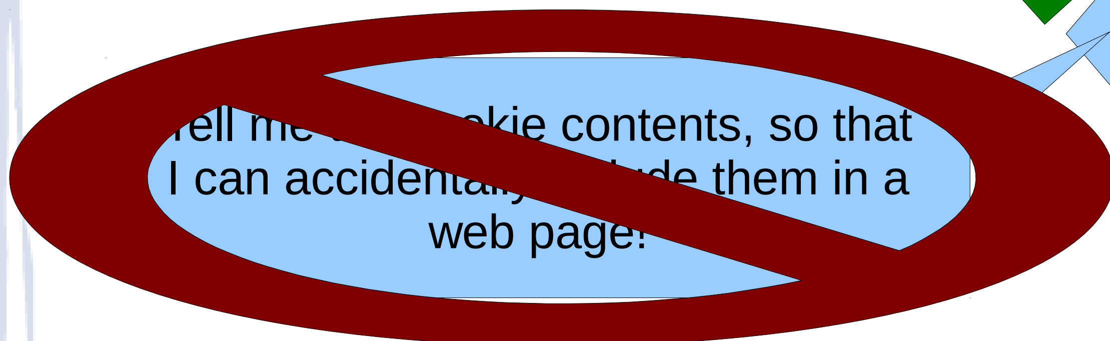
Login Cookie







Main App



Web 2.0 Components

Subtree of Dynamic Page Structure

Writable Area

Server-Side RPC Handler

?

Save the contents to the server.

Restore the contents.

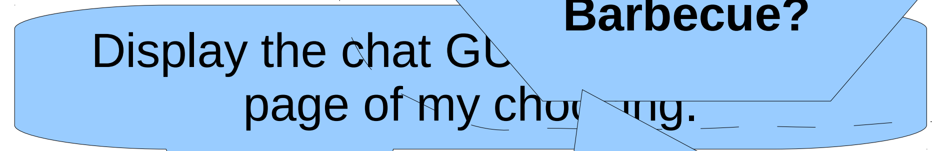
Main App

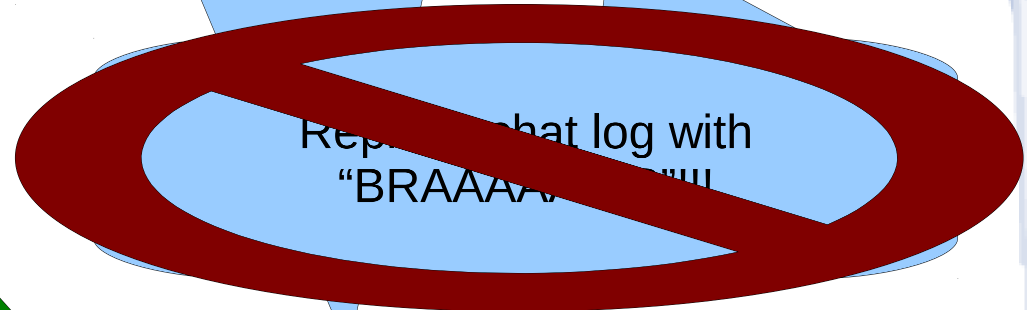
Let me ... e RPC manually, so I
can save cont... at I couldn't
add the honest w...

Comet Components

Subtree of Dynamic Page Structure

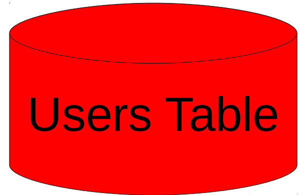
Persistent Channel for Server Comm.

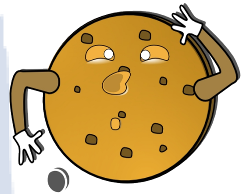




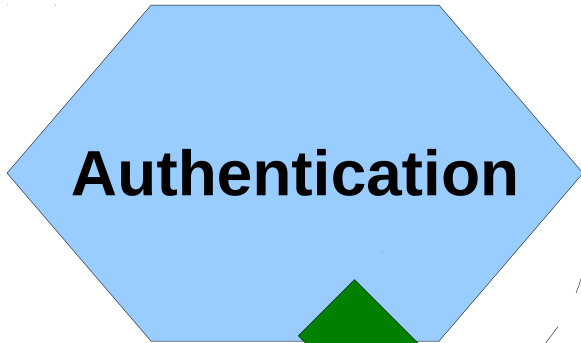
?

Back to Basics....





Login Cookie



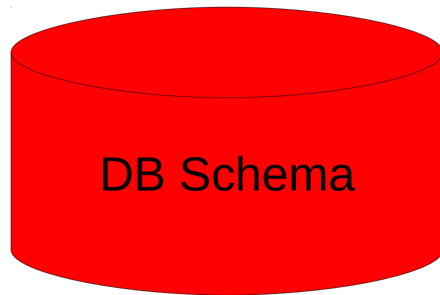
Subtree of
Dynamic
Page
Structure

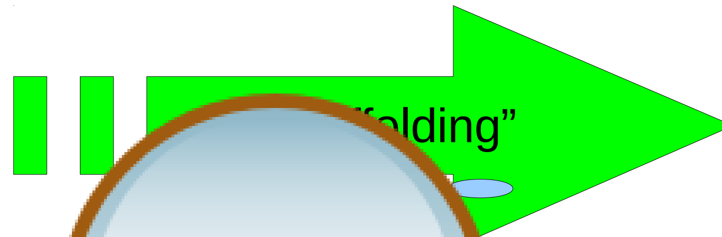
Server-Side RPC
Handler

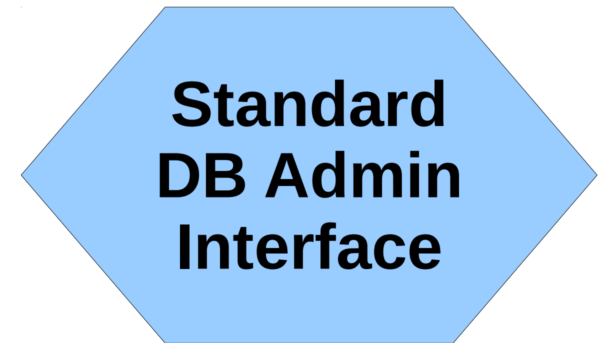
How do I dispatch
from URLs to all of
my different
components?

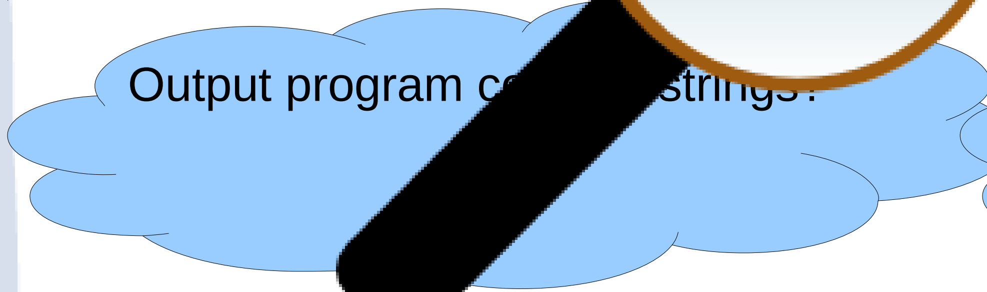
Combined Application

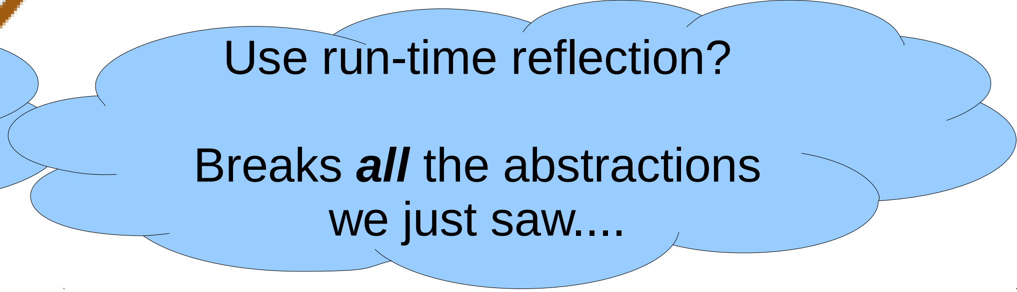
Code Generation











What is this and how do I use it?

The manual? Yeah, it's sometimes even accurate!
The source code? :-O
Examples? This is what happens today....

My Solutions

- Strongly-encapsulated web components?
  - First-class language support for the key pieces of web applications
  - No loopholes!
  - *Now use the standard abstractions of functional programming.*
- Principled metaprogramming?
  - *Statically-typed metaprograms*, using language ideas from the world of *dependent types*.

Ur/Web

Ur/Web, a special *standard library* and *compiler*
Supporting modern web app development

Web 1.0: Links, forms, etc.
Web 2.0: AJAX and Comet
SQL database access

Ur, a new general purpose language
Inspired by **ML** and **Haskell**,
but with even richer type system features

Advantages

Productivity

Important pieces of web applications are *first-class*.

New ways of structuring programs.

New approach to *meta-programming*.

Security

Any code to be interpreted is *not "just a string."*

Rules out:

Cross-site scripting
Code injection
Malicious file exec.
Cross-site req. forgery
...

Performance

Optimizing compiler produces fast & memory-efficient native code.

No garbage collection!

Domain-specific optimizations

Hello World!

```
fun main () = return <xml>
  <head>
    <title>Hello world!</title>
  </head>

  <body>
    <h1>Hello world!</h1>
  </body>
</xml>
```

Hello ____!

```
functor Hello(M : sig
                    val text : string
                  end)
  : sig
      val main : unit → page
    end = struct
  fun main () = return <xml>
    <head>
      <title>Hello {[M.text]}!</title>
    </head>

    <body>
      <h1>Hello {[M.text]}!</h1>
    </body>
  </xml>
end
```

Two Hellos, Living in Harmony

```
structure World =
   Hello(struct val text = “world” end)
structure Boston =
   Hello(struct val text = “Boston” end)

fun main () = return <xml>
   <p>Pick your poison:</p>
   <li><a link={World.main ()}>World</a></li>
   <li><a link={Boston.main ()}>Boston</a></li>
</xml>
```

Encapsulation

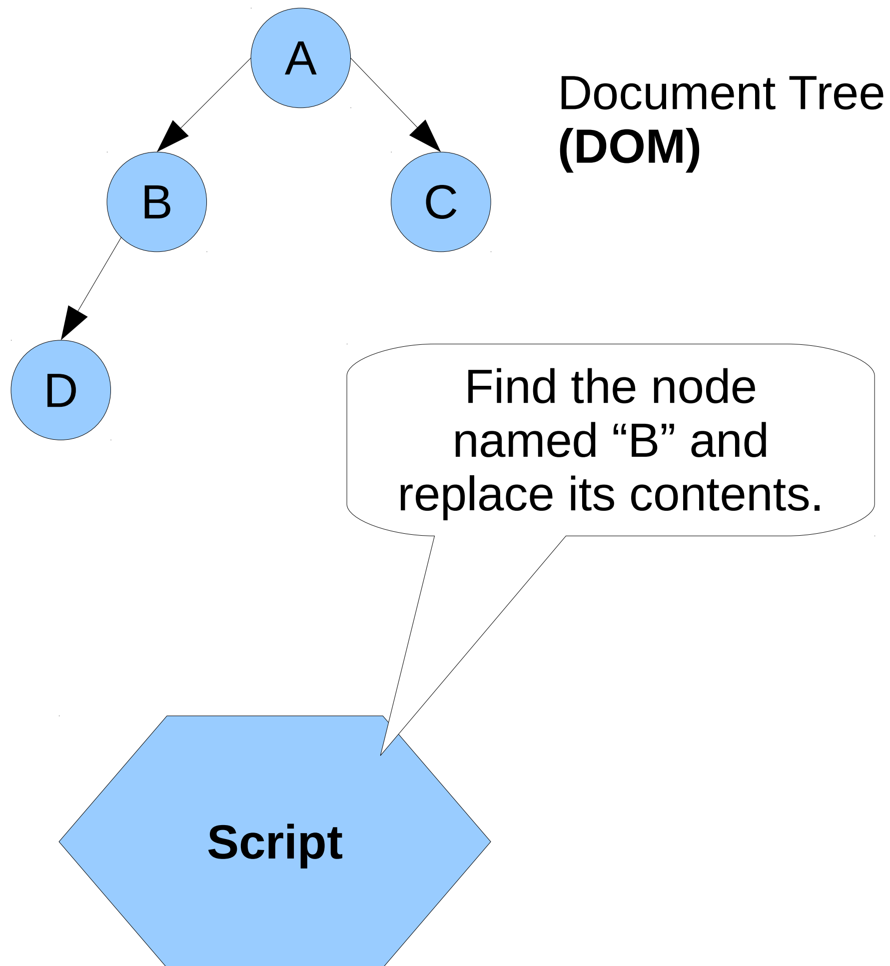
```
structure Auth
    : sig
        val loginForm : unit → xbody
        val whoami : unit → string
      end = struct
    table users : {Nam : string, Pw : string}
    cookie auth : {Nam : string, Pw : string}

    fun rightInfo r =
        oneRow (SELECT COUNT(*) FROM users
                WHERE Nam = {[r.Nam]}
                  AND Pw = {[r.Pw]}) = 1

    fun login r = if rightInfo r then setCookie auth r
                  else error “Wrong info!”

    fun loginForm () = (* Form handled by 'login'... *)

    fun whoami () = let r = getCookie auth in
                    if rightInfo r then r.Nam
                    else error “Wrong info!”
end
```

Some Client Code

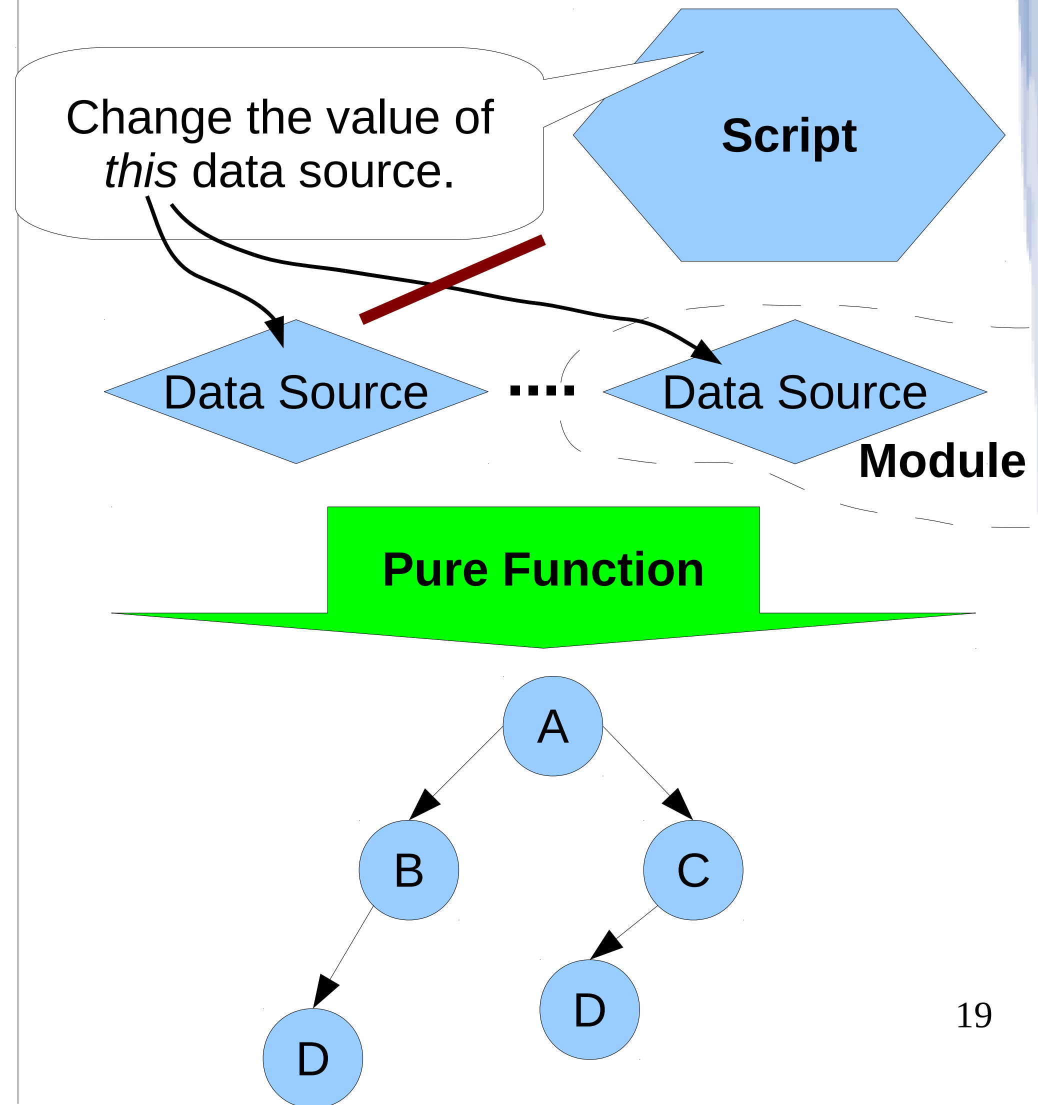
```
fun main () = return <xml>
   {Auth.loginForm ()}

   <p>Welcome.  You could
   <a link={somewhere ()}>go somewhere</a>.</p>
</xml>

and somewhere () =
   user ← Auth.whoami ();
   if user = “Fred Flintstone” then
      return <xml>Yabba dabba doo!</xml>
   else
      return <xml>Boring</xml>

                  Row (SELECT Pw
                                 Auth.user
                           WHERE Nam
```

Functional-Reactive GUIs

The Status Quo:



The Reactive Way:



A Client-Side Counter

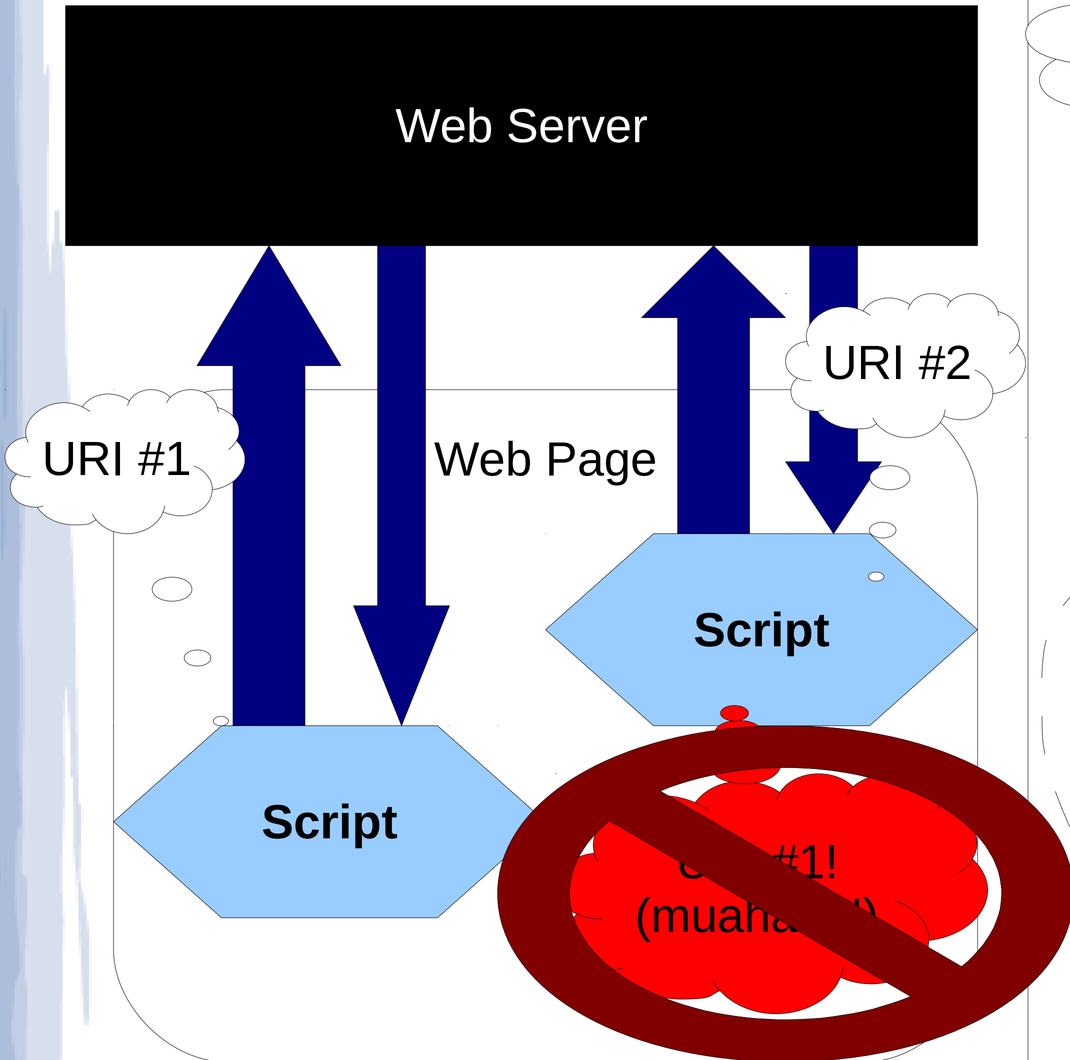
```
structure Counter : sig
                        type t
                        val new : unit → t
                        val increment : t → unit
                        val render : t → xbody
                      end = struct
  type t = source int

  fun new () = source 0

  fun increment c = n ← get c; set c (n + 1)

  fun render c = <xml>
    <dyn signal={n ← signal c;
                      return <xml><b>{[n]}</b></xml>}/>
  </xml>
end
```

Counter Client

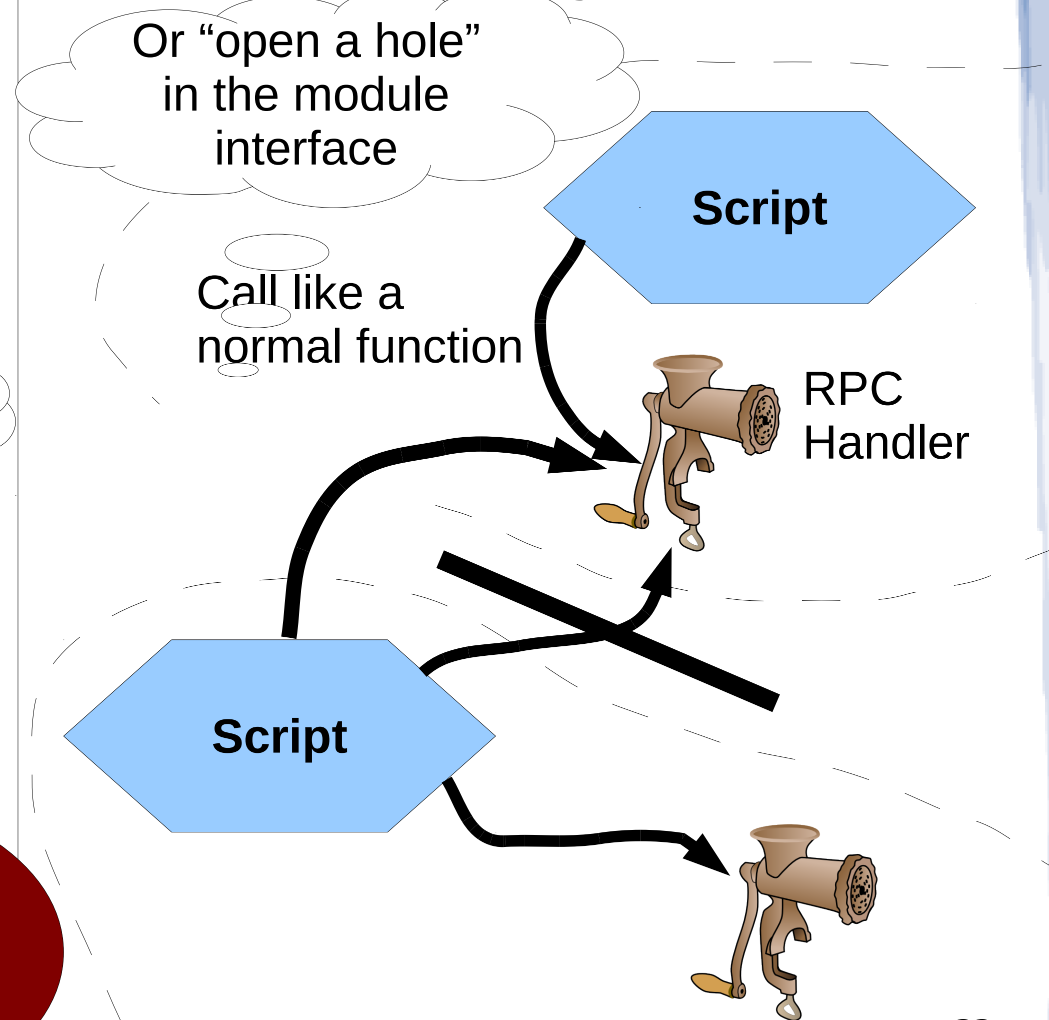
```
fun main () =
   c ← Counter.new ();
   return <xml>
      {Counter.render c}
      <button onclick={Counter.increment c}/>
      Backup copy: {Counter.render c}
      <button onclick={set c -1}/>
   </xml>
```

AJAX

The Status Quo:



The Modular Way:



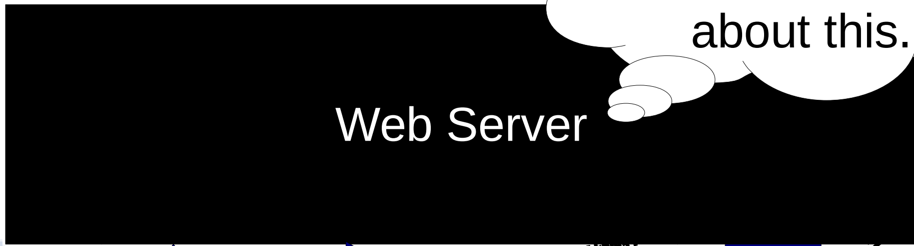
A Server-Side List

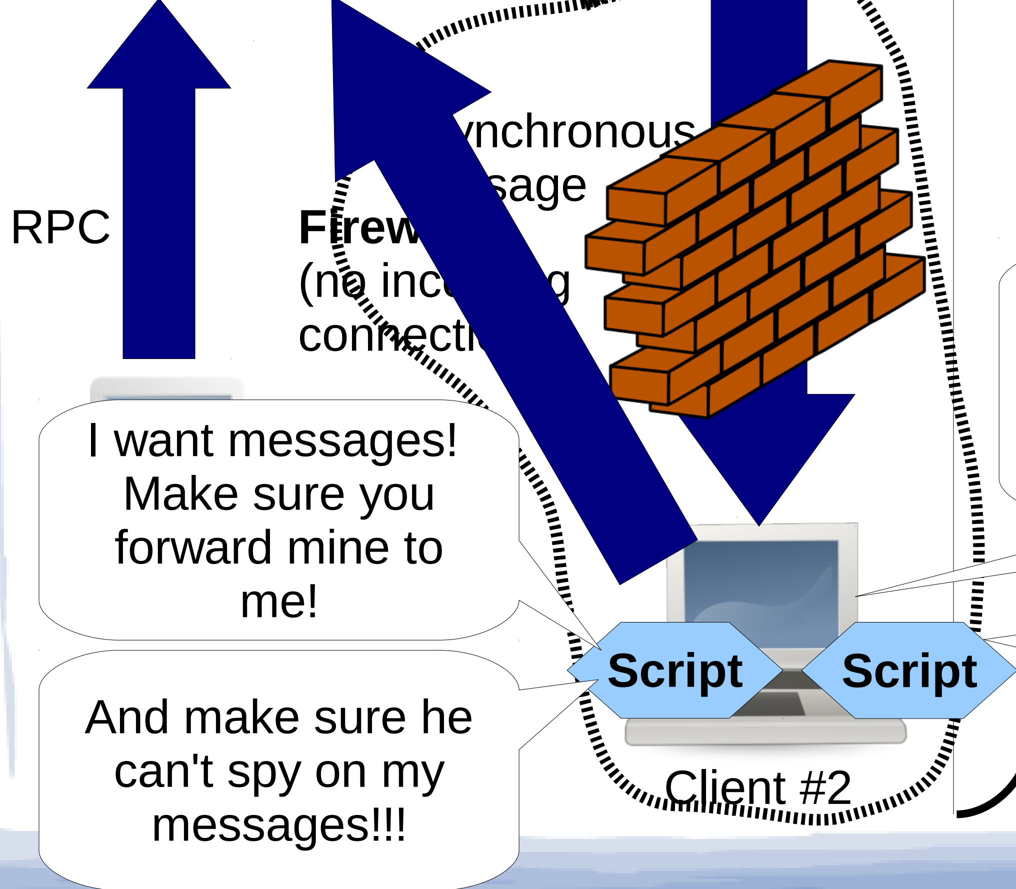
```
structure ServList : sig
                         type view
                         val new : unit → view
                         val render : view → xbody
                         val add : string → unit
                       end = struct
  table items : { It : string }
  fun allItems () = query (SELECT * FROM items)

  type view = source (list string)
  fun new () = its ← rpc (allItems ()); source its

  fun render v = return <xml>
    <dyn signal={its ← signal v;
                 (* Format this list for HTML display *)}/>
    <button value="Refresh" onclick={its ← rpc (allItems ());
                                      set v its}/>
  </xml>

  fun doAdd s = dml (INSERT INTO items VALUES ({[s]}))
  fun add s = rpc (doAdd s)
end
```

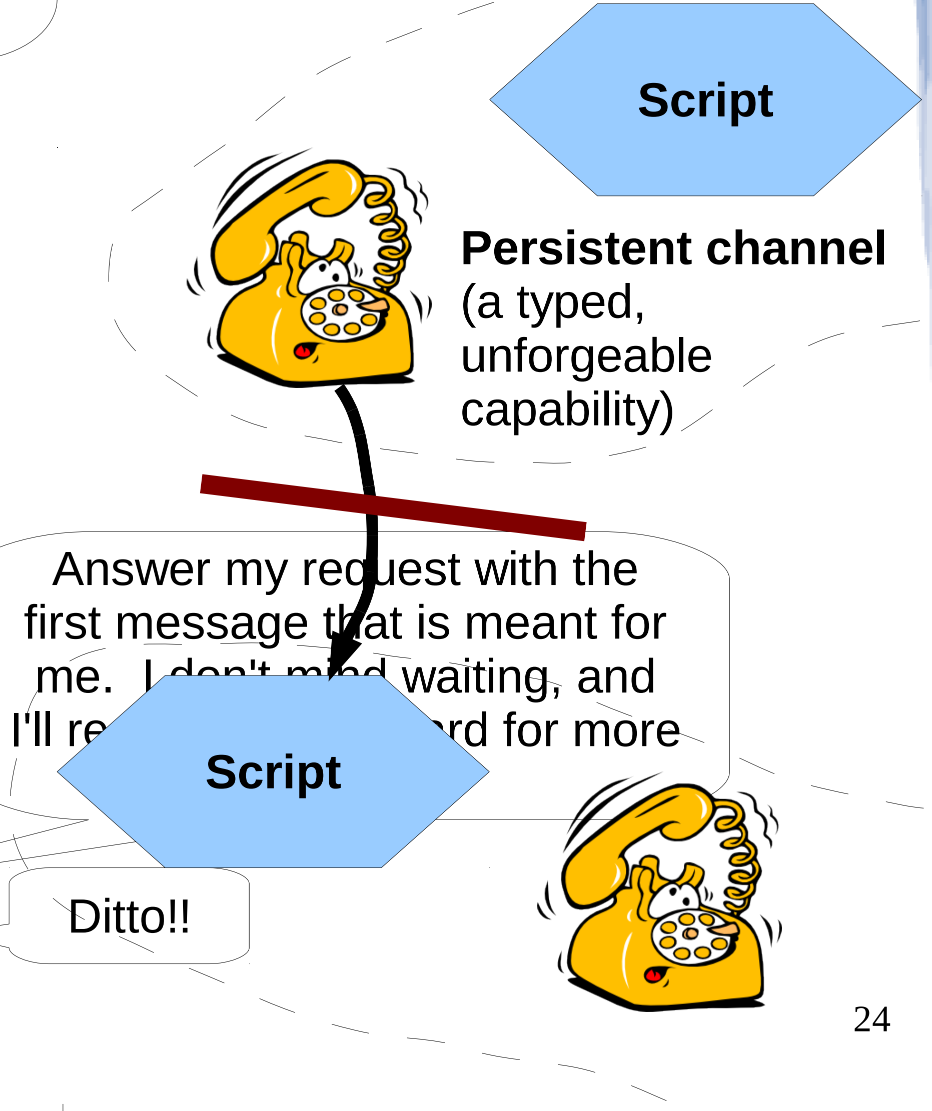
Comet

The Status Quo:

The Modular Way:







A More Proactive List

```
table listeners : { Chan : channel unit }
fun newListener () =
  ch ← channel;
  dml (INSERT INTO listeners VALUES ({[ch]}));
  return ch

fun new () =
  ch ← rpc (newListener ());
  its ← rpc (allItems ());
  v ← source its;
  let fun loop () =
        dummy ← recv ch;
        its ← rpc (allItems ());
        set v its;
        loop ()
  in spawn (loop ());
     return v
  end

fun doAdd s = dml (INSERT INTO items VALUES ({[s]}));
              foreach (SELECT * FROM listeners)
                (fn ch => send ch ())
```

Metaprogramming: An Executive Summary

Compile-time Programming Language
(AKA, type system)
(not Turing complete, but includes basic lambda calculus)
Crucial feature: **type-level records!**

Included in

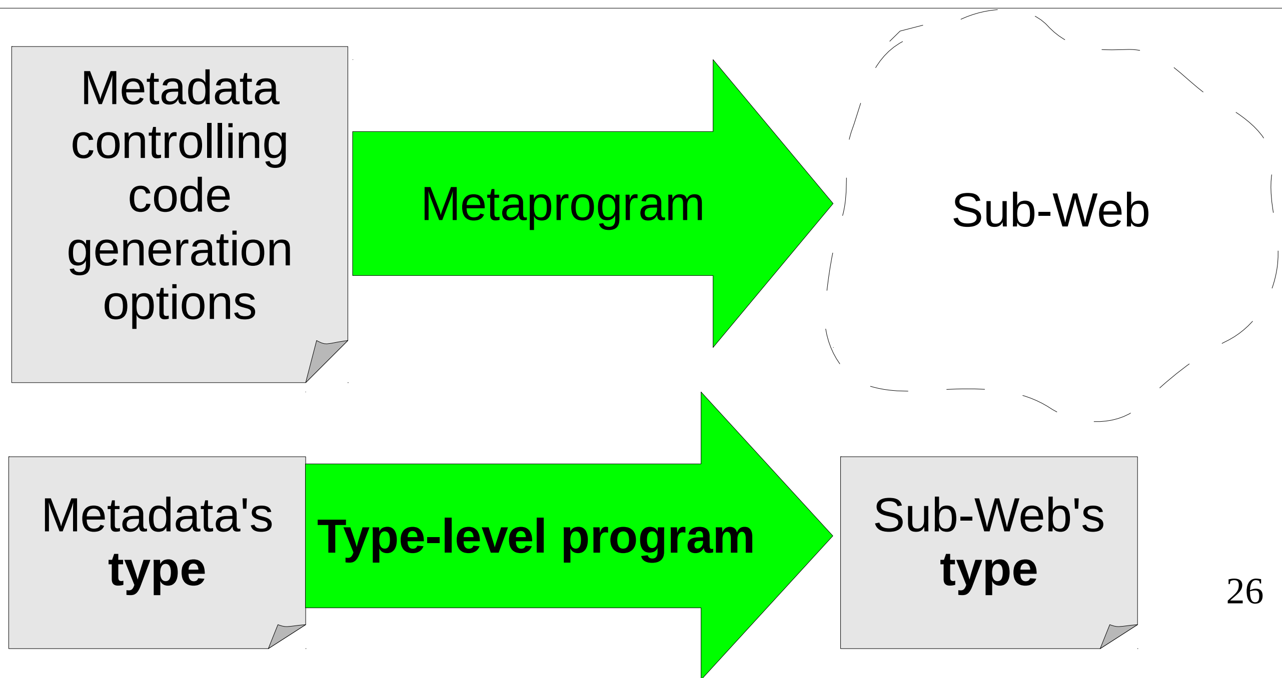
Run-time Programming Language
(Turing complete)

Key Property 1:

These types are expressive enough to guarantee absence of code injection, abstraction violation, etc..

Key Property 2:

Effective static checking that the metaprogram really has the claimed type



Demo

For more information...

http://www.impredicative.com/ur/