Developing a Web server in Ada with AWS

J-P. Rosen
Adalog
An AXLOG Group company

rosen@adalog.fr

- Introduction
- Internet
- AWS basics
- The templates parser
- AWS advanced
- Distributed applications with AWS
- AWS in practice
- Conclusion

AWS

- Ada Web Server

Many thanks for the slides!

  - Authors: Pascal Obry, Dmitriy Anisimkov.
- History and availability
  - Project started on January 2000
  - Free Software (GMGPL)
  - 100% Ada (except SSL based on OpenSSL and LDAP based on OpenLDAP/MS LDAP)
  - Windows - GNU/Linux - FreeBSD...
  - Download:
    - http://libre.act-europe/aws/ (english)
    - http://www.obry.org/contrib.html (french)

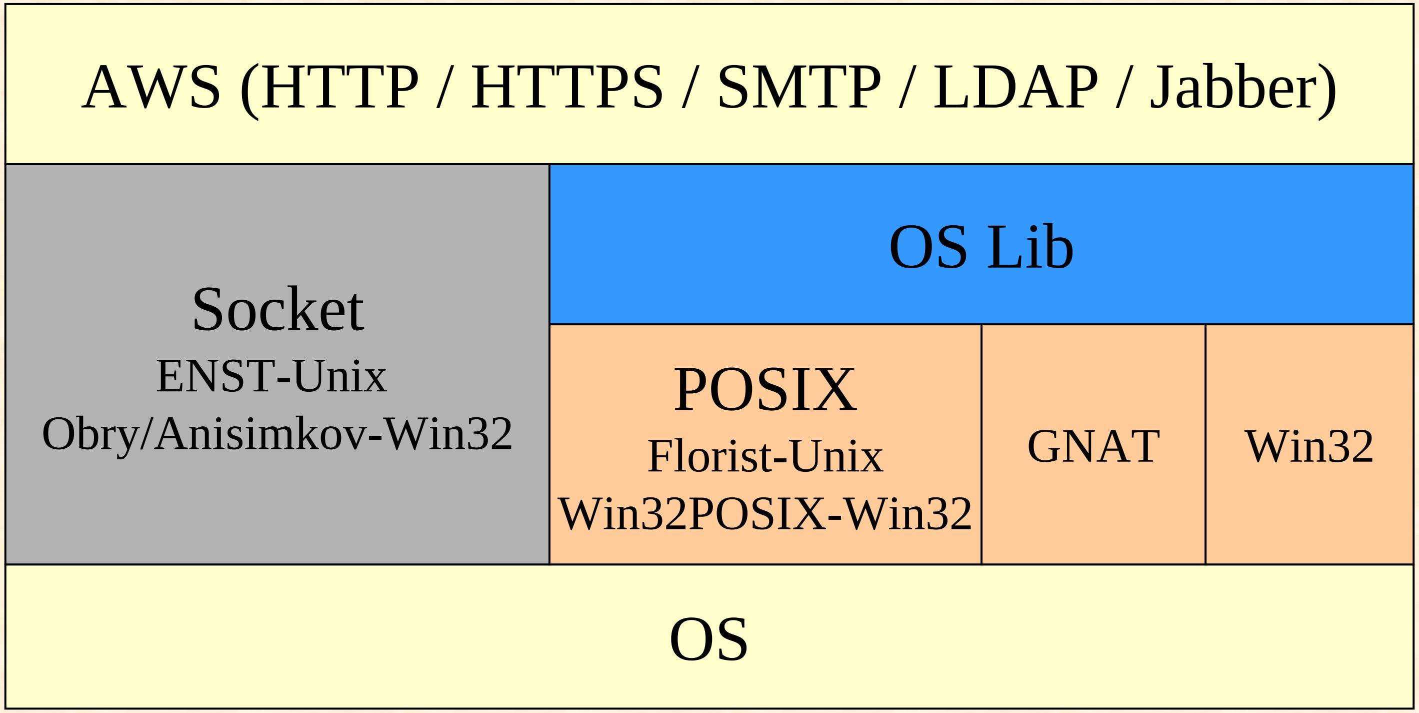
What is AWS?

82 (user) packages !

- A set of packages for managing protocols
  - http/https, SOAP, LDAP, Jabber, SMTP, POP…
  - Server side
  - Client side
- Facilities for managing pages (dispatchers)
- Facilities for building pages (templates parser)
- Facilities for making distributed applications
- Other facilities (Resources, WSDL…)

Architecture

- AWS - UNIX, Windows, ...



AWS (HTTP / HTTPS / SMTP / LDAP / Jabber)

Socket
ENST-Unix
Obry/Anisimkov-Win32

OS Lib

POSIX
Florist-Unix
Win32POSIX-Win32

GNAT

Win32

OS

- Introduction
- Internet
- AWS basics
- The templates parser
- AWS advanced
- Distributed applications with AWS
- AWS in practice
- Conclusion

Internet

- Internet protocol suite

Application layer	HTTP	SMTP	FTP	LDAP	…
Transport layer	TCP	UDP	SCTP	ICMP	…
Network layer	IP	IPv6	IPX	ARP	…
Data link layer	Ethernet	Token ring	FDDI	802.11 (Wifi)	…

- Communication needs a stack of protocols
  - For example: HTTP over TCP/IP over Ethernet

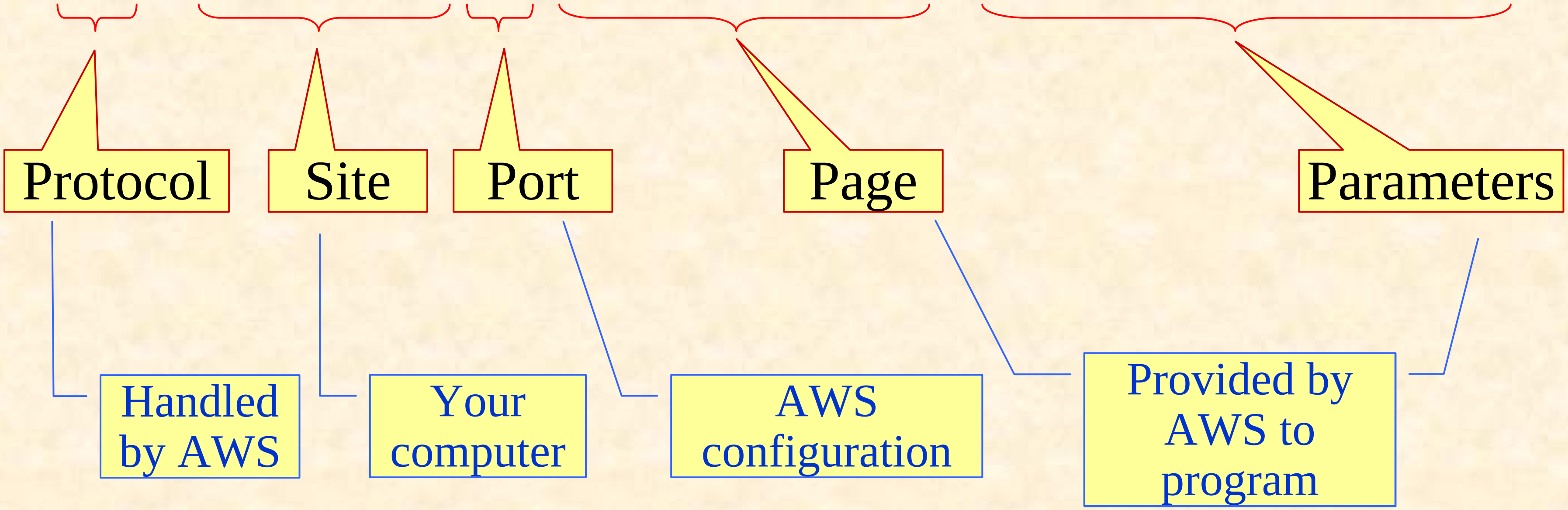
HTTP, HTTPS

- A protocol for exchanging information between a client and a server (RFC 2616)
  - HTTP is *not secure* (all messages are readable)
  - HTTPS is *secured* HTTP.
    - HTTP over SSL. SSL uses a 40-bit key size for the RC4 stream encryption algorithm, which is considered an adequate degree of encryption for commercial exchange.
- HTTP defines the form of messages exchanged between client and server
  - Headers, contents, encryption…

HTTP Fundamentals

- A server provides a response to a request applied to a URI
  - *Uniform Resource Identifier*
- Structure of a URI:

```
http://www.site.com:8650/page-dir/page-name?Param1=Value1&Param2=Value2
```



Response Code

- Each response has a code to explain what it means:
  - 1xx codes: Informational, request received, continuing.
  - 2xx codes: Action accepted
    - 200: OK
  - 3xx codes: Redirection
    - 301: Moved permanently
  - 4xx codes: Client error
    - 404: Not found
  - 5xx codes: Server error
    - 500: Internal server error

If code /= 200, the body gives more information

HTML

- A standard for representing information
  - based on *tags*

```
<b> bold text </b>
```

Unrecognized tags are *ignored*

```
<applet codebase=… code=… >
   <param name="param1" value="…">
   <param name="param1" value="…">
   Your browser does not support JAVA!
</applet>
```

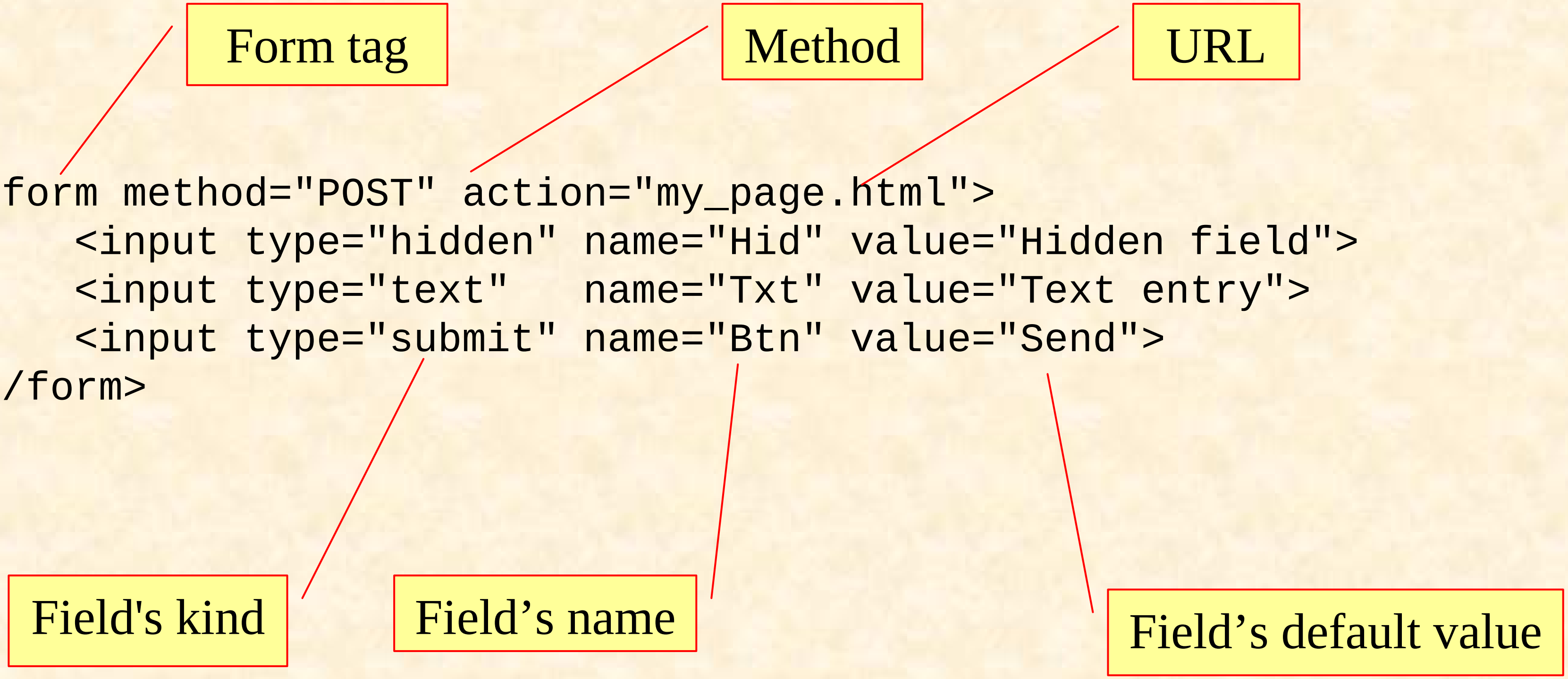
  - Line breaks, spaces are *irrelevant*

Interactive HTML

- How can the user interact with the server?
- Links
  - Leads to another page
  - Not parameterizable by the user
- Forms
  - A request (URI) parameterized from user input
  - Entry fields : Text , Password, Textarea, Radio, Checkbox, Select, File
  - Buttons: Submit, Resest, Image, Button (effect defined in JavaScript)
  - Hidden fields

A button must be in a form!

HTML Forms



Form tag

Method

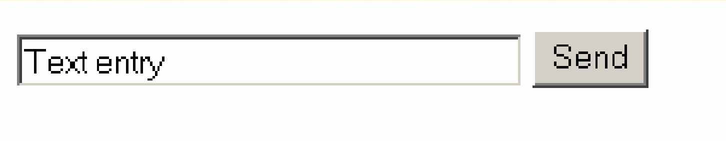
URL

```
<form method="POST" action="my_page.html">
    <input type="hidden" name="Hid" value="Hidden field">
    <input type="text"   name="Txt" value="Text entry">
    <input type="submit" name="Btn" value="Send">
</form>
```

Field's kind

Field’s name

Field’s default value



Text entry

- Result:

```
http://my_page.html?Hid="Hidden field"&Txt="Text entry"&Btn="Send"
```

Form Methods

- Define the effect of the submission to the server

GET	Request does not change the state of the server
POST	Request changes the state of the server and returns new state.
PUT	Request changes the state of the server and does not (necessarily) return new state.
HEAD	Request does not need the body of the response

- In practice:
  - Use GET for URIs
  - Use POST for forms
  - Forget others

URI Encoding

- Since the response may be part of the created URI, it can contain any character…
- But some characters have special meaning in URIs
- Spaces are encoded as '+'
- Other special characters are encoded in Hex:
  - %hh
  - '?' = %3F

JavaScript (ECMAScript)

- A scripting language
  - + source embedded in HTML page
  - + interpreted by the browser
  - + script functions can be linked to forms events
  - + useful for
    - checking data in forms, …
    - Dynamic menus
    - Displaying (foldable) trees
    - Controlling the browser

Generally not needed with AWS

Java

- A compiled language for a virtual machine
  - HTML page contains a link to the byte-code file
  - The byte code needs not be generated from the Java language
    - http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html: 186 compilers!
- Virtual machine emulated by the browser

Generally not needed with AWS

PHP, ASP, JSP…

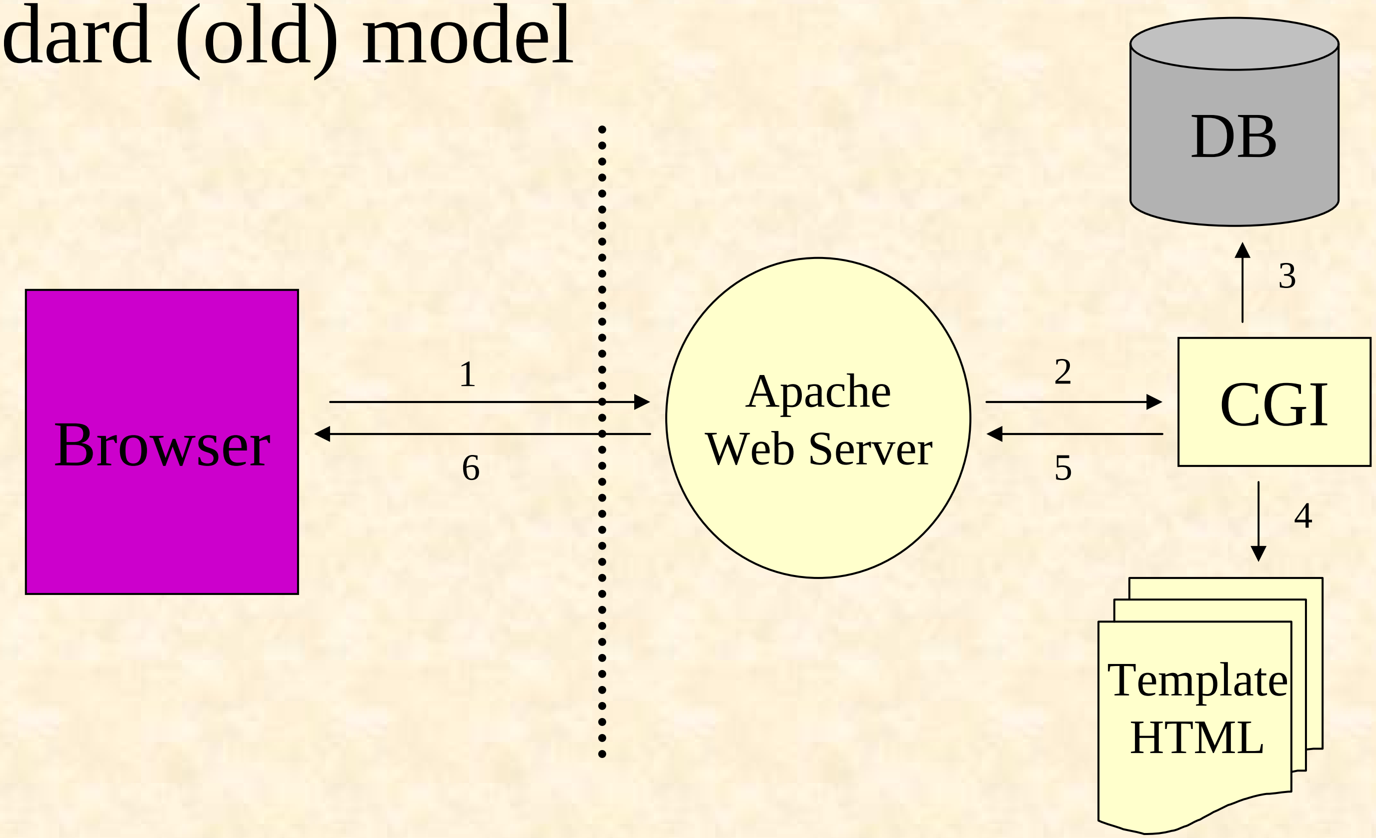
- Server Side Inserts languages
  - Code is included in a page template
  - Interpreted by the server
  - Builds the page dynamically before it is returned

Never needed with AWS

- Introduction
- Internet
- **AWS basics**
- The templates parser
- AWS advanced
- Distributed applications with AWS
- AWS in practice
- Conclusion

Web Development

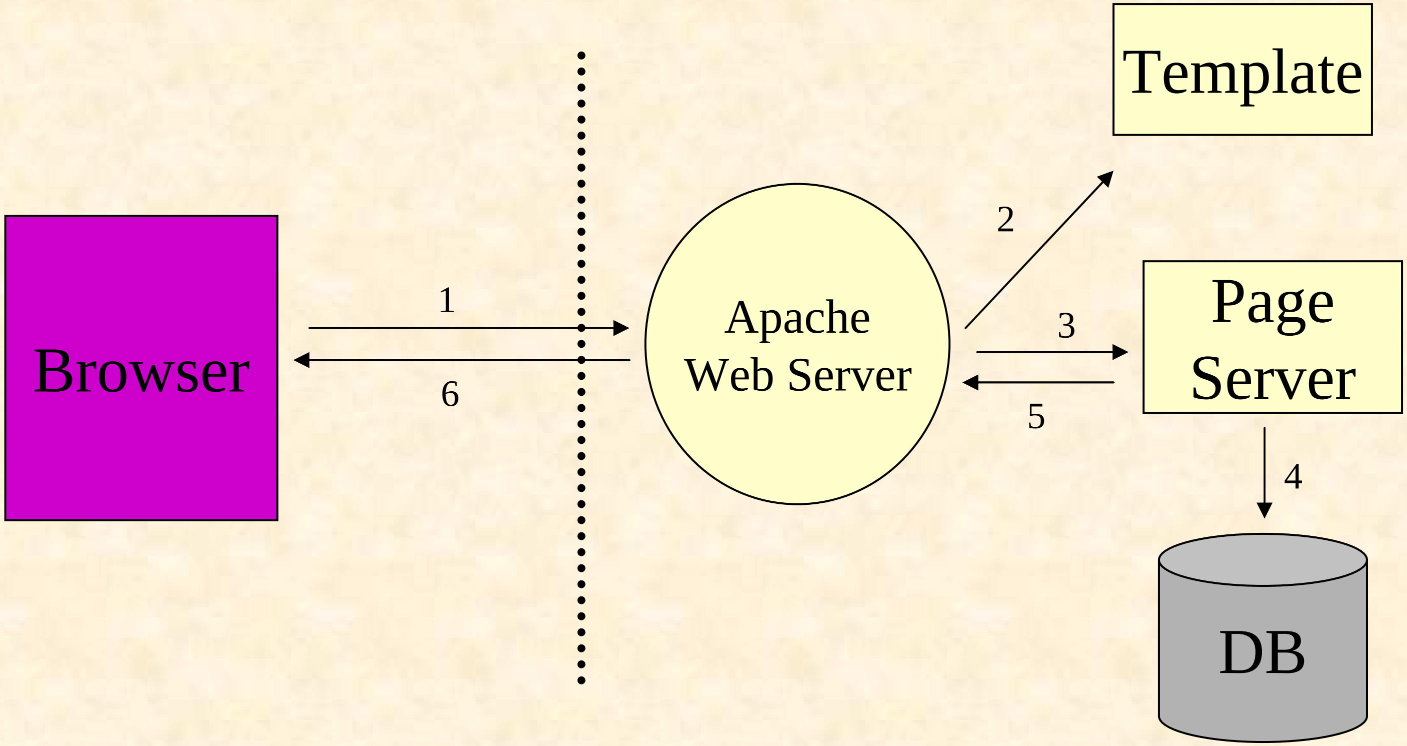
- Standard (old) model



The program is separated from the server

Web Development

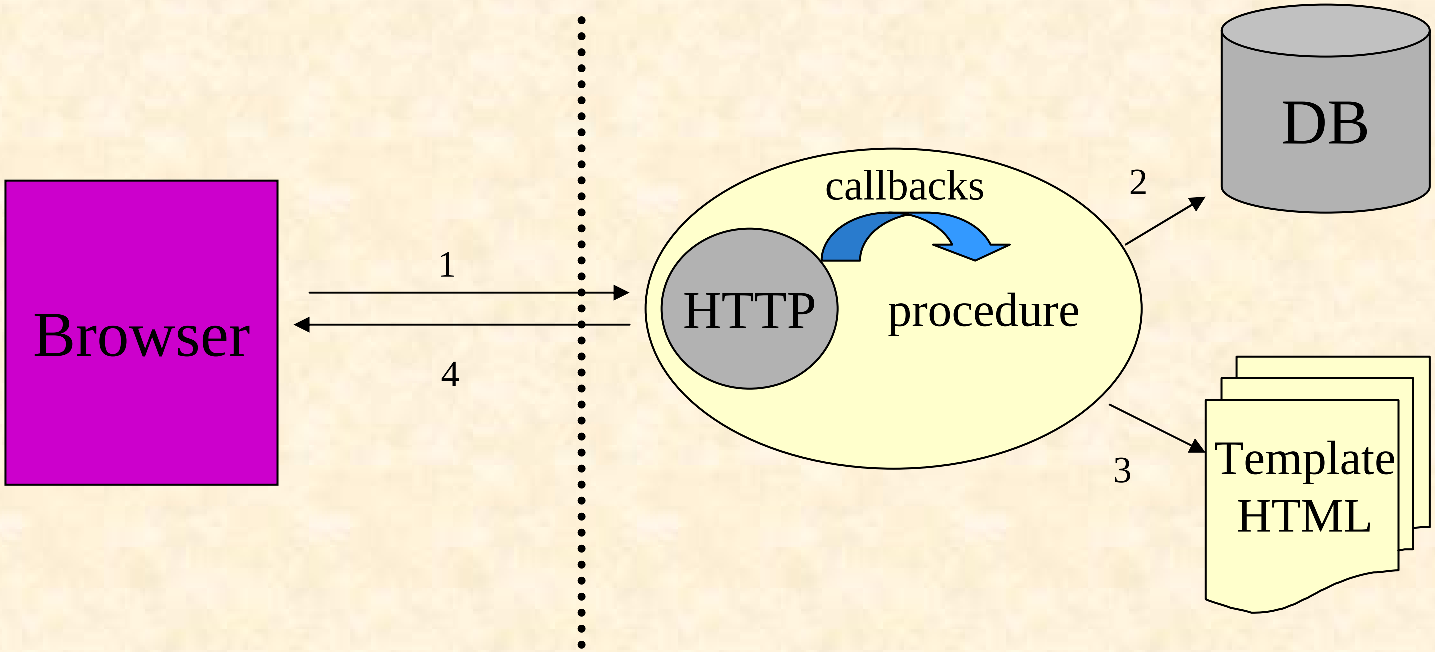
- Scripting model (Server side inserts)



The program is inside the server

Web Development

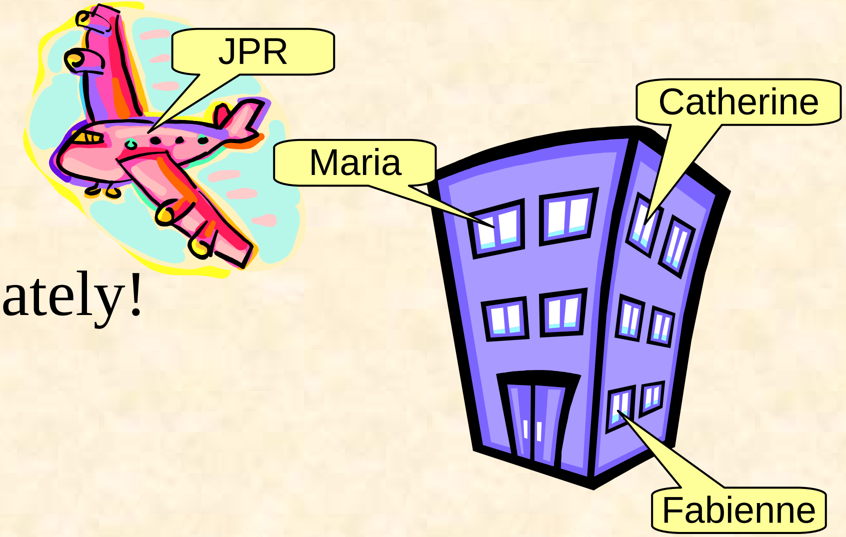
- AWS based model



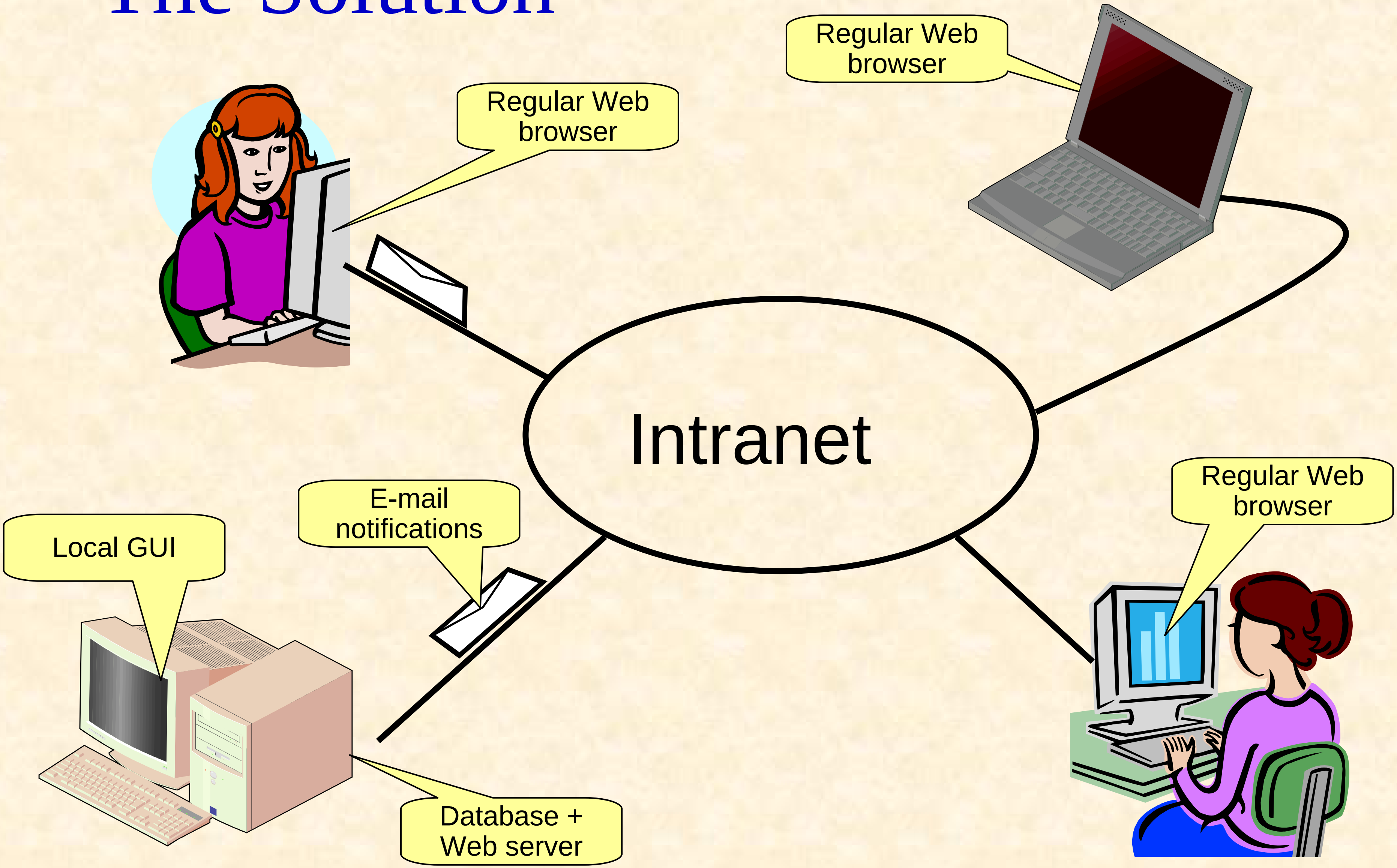
The server is inside the program

Example: Adalog's Gesem

- Managing the registration to training sessions
  - Several persons in charge
  - In various locations,
    not available at the same times
  - Must answer the phone immediately!
- Pinging people
  - Prepare hand-outs
  - Reserve restaurant
  - …
- Managing mailing
  - Classical database extraction



The Solution



Basic Behaviour

- AWS :
  - +opens the HTTP(S) message
  - +Gets answer using the user’s callback procedure
  - +Encapsulates answer and sends it back to browser

```
procedure Start(Web_Server : in out HTTP;
                Callback   : in     Response.Callback;
                Config     : in     AWS.Config.Object);

type Callback is access
   function (Request : Status.Data) return Response.Data;
```

The callback is the “script”,
but the language is full Ada.

Using AWS (1)

- User:
  - Declare server to handle the HTTP protocol.
  - Start the server (several overloaded `Start` procedures)

```
procedure Demo is
   WS : Server.HTTP;
begin
   Server.Start (WS, "demo server", Service'Access, 3,
                 "/Admin-Page", 1024,
                 Security => True, Session => True);
```

Simultaneous connections

Callback procedure

Status page

Port

HTTPS

Session handling

Using AWS (2)

- Do not exit from the main program

```
   ...
   Server.Wait (Server.Q_Key_Pressed);
   --  Wait for the Q key to be pressed

   Server.Wait (Server.Forever);
   -- Wait forever, the server must be killed

   Server.Wait (Server.No_Server);
   --  Exit when there is no server running (all of them
   --  have been stopped)
end Demo;
```

Using AWS (3)

- Stopping the server

```
procedure Demo is
   WS : Server.HTTP;
begin
   …

   Server.Shutdown (WS);
```

Shutdown can be called from a call-back function while the main program is on wait

Using AWS (4)

- Develop the callback procedure which is called by the server.
  + Used to provide answer for the requested URI.

```
function Service (Request : in Status.Data) return Response.Data
is
   URI : constant String := Status.URI (Request);
begin
   if URI =  "/givemethat" then
      return Response.Build (Content_Type => "text/html";
                             Message_Body => "<p>Hello there !");
   elsif ...
```

The callback procedure must be thread-safe.

Using AWS (5)

- The form’s parameters

```
function Service (Request : in Status.Data) return Response.Data is
  P_List : constant Parameters.List := Status.Parameters (Request);
  --  List of parameters
   N  : constant Natural := Natural'Value
                                (Parameters.Get (P_List, "count"));

   --  Numbers is a list with multiple selections enabled
   V1 : constant String := Parameters.Get (P_List, "numbers", 1)
   V2 : constant String := Parameters.Get (P_List, "numbers", 2)
begin
   …
```

Using AWS (6)

- A response is built with one of the AWS.Response constructors.
  - From a string :

```
function Build
  (Content_Type  : in String;
   Message_Body  : in String;
   Status_Code   : in Messages.Status_Code  := Messages.S200;
   Cache_Control : in Messages.Cache_Option := Messages.No_Cache)
return Data;
```

  - From a file:

```
function File
 (Content_Type : in String;
  Filename     : in String;
  Status_Code  : in Messages.Status_Code := Messages.S200)
return Data;
```

Object Oriented AWS (1)

- A tagged type can be used instead of a call-back function

```
package AWS.Dispatchers is
   type Handler is abstract new Ada.Finalization.Controlled
      with private;
   procedure Initialize (Dispatcher : in out Handler);
   procedure Adjust     (Dispatcher : in out Handler);
   procedure Finalize   (Dispatcher : in out Handler);

   function Dispatch (Dispatcher : in Handler;
                      Request    : in Status.Data)
   return Response.Data is abstract;
…

procedure Start (Web_Server : in out HTTP;
                 Dispatcher : in     Dispatchers.Handler'Class);
…
```

Object Oriented AWS (2)

- Benefit: the dispatcher can be extended
  - For example, a function to register a call-back (or another dispatcher) for pages matching a given pattern
  - An ordered set of rules with the corresponding action.
  - Helps manage the complexity of large projects.
- Provided: AWS.Dispatchers.Callback
  - A simple wrapper around the regular callback procedure
  - Adds:

```
function Create (Callback : in Response.Callback)
   return Handler;
```

- More dispatchers later…

Example : Hello_World

```ada
with AWS.Response;
with AWS.Server;
with AWS.Status;

procedure Hello_World is
   WS  : AWS.Server.HTTP;

   function Service (Request : in AWS.Status.Data)
     return AWS.Response.Data is
   begin
      return AWS.Response.Build ("text/html", "<p>Hello world !");
   end Service;

begin
   AWS.Server.Start (WS, "Hello World",
                     Callback => Service'Unrestricted_Access);
   AWS.Server.Wait (AWS.Server.Q_Key_Pressed);
end Hello_World;
```

Because the call-back is a local function

Example : A Static Page Server

```
function Service (Request : in AWS.Status.Data)
  return AWS.Response.Data
is
   URI      : constant String := AWS.Status.URI (Request);
   Filename : constant String := URI (2 .. URI'Last);
begin
   if OS_Lib.Is_Regular_File (Filename) then
      return AWS.Response.File
        (Content_Type => AWS.MIME.Content_Type (Filename),
         Filename     => Filename);
   else
      return AWS.Response.Acknowledge
        (Messages.S404, "<p>Page '" & URI & "' Not found.");
   end if;
end Service;
```

- Introduction
- Internet
- AWS basics
- The templates parser
- AWS advanced
- Distributed applications with AWS
- AWS in practice
- Conclusion

Templates Parser: Why?

- 100% code and design separation.
- Other projects : WebMacro, FreeMarker, PHP, JSP, ASP…(scripting in HTML pages).
- Velocity : W3C Project (code/design separation, based on Java introspection).
- Java Struts (maturing project...)

Ada for the code, some HTML tags to layout the data. No scripting in the HTML.

The Templates Parser

- An independent component…
  - but extremely useful with AWS!
- The template: a text file (or a string) parameterized with
  - Commands
  - Variables (tags)

Special characters for commands and tags can be changed

- The parser replaces tags with their values and executes commands.
  - Templates parser engine is very fast
    - Templates are “compiled” in memory (semantic tree)
    - More than 20 times faster than JSP, ASP…

Tags

- A tag is a named variable
  - appears in template as `@_NAME_@`
- A translation table is an array of associations
  - Name => Value
- Associations have constructors for:
  - Scalar
    - String, Unbounded_String, Integer, Boolean (True, False)
  - Vector
    - One-dimensional array
  - Matrix
    - Two-dimensional array (actually, a vector of vector-tags)

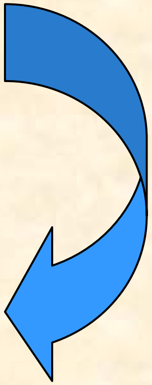
Setting Tags

```
procedure Tags is
  use type Vector_Tag;
  use type Matrix_Tag;

  B : constant Boolean    := True;
  V : constant Vector_Tag := +"10" & "30" & "5";
  M : constant Matrix_Tag := +V & V;
  S : constant String     := "a value";

  Translations : constant Translate_Table
     := (1 => Assoc ("TEST", B),
         2 => Assoc ("VECT", V),
         3 => Assoc ("MAT",  M),
         4 => Assoc ("VAL",  S));
```

Tag Substitution

Template file simple.tmplt):

```
@@-- A simple template
@@-- NAME : User’s name
<HTML>
<P>Hello @_NAME_@</P>
</HTML>
```



Resulting HTML:

```
<HTML>
<P>Hello Bill</P>
</HTML>
```

```
procedure Simple is
  Translations : Translate_Table
    := (1 => Assoc ("NAME", "Bill"));
begin
  Put_Line (Parse ("simple.tmplt",
                   Translations));
end Simple;
```

Vector and Matrix Substitution

Template file simple.tmplt):

```
@@-- A simple template
<HTML>
<P>Hello @_VECT_@
<P>Hello @_MAT_@
</HTML>
```

Resulting HTML:

```
<HTML>
<P>Hello Jean, John, Hans
<P>Hello Jean, John, Hans
Jean, John, Hans
</HTML>
```

```
procedure Simple is
  V : constant Vector_Tag := +"Jean" &
                              "John" &
                              "Hans";
  M : constant Matrix_Tag := +V & V;
  Translations : Translate_Table
      := (Assoc ("VECT", V),
          Assoc ("MAT", M));
begin
  Put_Line (Parse ("simple.tmplt",
                Translations));
end Simple;
```

Tag Modifiers

`@_{FILTER:}Tag['ATTRIBUTE]_@`

- Filters:
  - `@_UPPER:VAR_@`
  - `@_ADD(3):VAR_@`
  - `@_EXIST:VAR_@`
  - `@_MATCH("Adalog.*"):VAR_@`
  - `@_FORMAT_DATE("%H-%M-%S"):NOW_@`
  - `@_YES_NO:VAR_@`
  - `@_WEB_ESCAPE:WEB_NBSP:CAPITALIZE:TRIM:VAR_@`

And many more

- Attributes:
  - `@_VECT'LENGTH_@`
  - `@_MAT'LINE_@`
  - `@_MAT'MIN_COLUMN_@`
  - `@_MAT'MAX_COLUMN_@`

Predefined Tags

- These tags are always defined:
  - NOW
  - YEAR
  - MONTH
  - DAY
  - HOUR
  - MINUTE
  - SECOND
  - MONTH_NAME
    - January .. December
  - DAY_NAME
    - Monday .. Sunday

Templates Commands

- Comments

```
@@-- Any text
```

- Conditions

```
@@IF@@ <expression>
  …
@@ELSIF@@ <expression>
  …
@@ELSE@@
  …
@@END_IF@@
```

- Table
- Include

Expressions in "IF" Command

- Comparisons
  - =, /=, <, <=, >, >=
- Logical
  - and, or, xor, not
- Parentheses

```
@@IF@@ @_A_@ = "This chain" or (@_B_@ = 3 and @_C_@ /= 0)
```

Expressions must fit on one line
Quotes are required if the value contains spaces

Table Command

```
@@TABLE@@
   <code>
@@END_TABLE@@
```

- Is really an iterator
  - If the name of a vector tag appears in a table, it is replaced by a value from the vector tag
  - Content is repeated until all vector and matrix tags are exhausted
  - A shorter vector is completed with empty strings
- Can be nested
  - At level 1:
    - the name of a vector provides a value
    - the name of a matrix tag provides a vector
  - At level 2:
    - the name of a vector provides a value (new iteration)
    - the name of a matrix tag provides a value

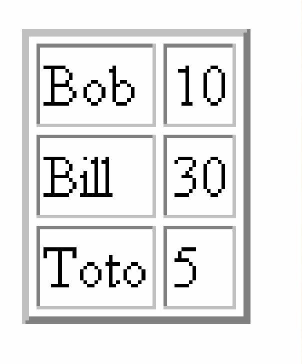
Tables and Vector Tags (1)

```
procedure Table is
  use type Vector_Tag;
  Names : constant Vector_Tag := +"Jean" & "John" & "Hans";
  Ages  : constant Vector_Tag := +"10" & "30" & "5";
  Translations : constant Translate_Table
     := (1 => Assoc ("NAME", Names),
         2 => Assoc ("AGE", Ages));
begin
   Put_Line (Parse ("table.tmplt", Translations));
end Table;
```

Tables and Vector Tags (2)

Template file (table.tmplt):

```
<TABLE border=2>
@@TABLE@@
<TR>
  <TD>@_NAME_@</TD> <TD>@_AGE_@</TD>
</TR>
@@END_TABLE@@
</TABLE>
```



Bob	10
Bill	30
Toto	5

Resulting HTML:

```
<TABLE border=2>
<TR>
  <TD>Bob</TD> <TD>10</TD>
</TR>
<TR>
  <TD>Bill</TD> <TD>30</TD>
</TR>
<TR>
  <TD>Toto</TD> <TD>5 </TD>
</TR>
</TABLE>
```

Table Sections

```
@@TABLE@@ [@@TERMINATE_SECTION@@]
   <code>
{@@SECTION@@
   <code>}
@@END_TABLE@@
```

- Each iteration uses one section in round-robin order.
- if `@@TERMINATE_SECTION@@` is specified, iteration will continue until the last section is reached
  - Matrix and vector tags are completed as necessary with empty strings

Special Tags in Tables

- `@_TABLE_LINE_@`
  - Current line number
- `@_UP_TABLE_LINE_@`
  - Line number of enclosing table
- `@_NUMBER_LINE_@`
  - Total number of lines in table
- `@_TABLE_LEVEL_@`
  - Current table depth

```
@@TABLE@@
<p>@_TABLE_LINE_@/@_NUMBER_LINE_@ : @_VECT_@
@@END_TABLE@@
```

```
1/3 : 10
2/3 : 30
3/3 : 5
```

Includes

`@@INCLUDE@@ filename` *`[parameters]`*

- Reads from another file
  - Useful for headers and other repetitive elements
- In an included file:
  - @_0_@ is the file name
  - @_1_@ .. @_n_@ is the nth parameter

`footer.thtml :`

```
Copyright @_$1_@ 2004
```

```
@@INCLUDE@@  footer.thtml Adalog
@@INCLUDE@@  footer.thtml Axlog
```

```
Copyright Adalog 2004
Copyright Axlog 2004
```

- Introduction
- Internet
- AWS basics
- The templates parser
- AWS advanced
- Distributed applications with AWS
- AWS in practice
- Conclusion

Transient pages

- Pages that do not have to stay forever
- Pages that expire after a given delay has elapsed
  - AWS.Services.Transient_Pages
    - Pages are kept in memory
    - Get a "special" URI: Get_URI
    - Build the response in a stream, and associate it to the URI, giving the lifetime: Register
    - Get the associated stream from the URI: Get
- Pages that expire after being sent
  - Set parameter "Once" to True in Response.File
    - The file is deleted after being sent

Split pages

- Pages may be very big…
  - Google can return *millions* of results
  - The result of a query may have to be split over several pages
- AWS.Services.Split_Pages
  - Parse a template with *two* translation tables
    - One for tags common to all pages
    - One for tags used by tables split over several pages
    - Extra tags added (NEXT, PREVIOUS, PAGE_INDEX, NUMBER_PAGES, OFFSET, HREFS_V, INDEXES_V)
  - Creates transient pages for all pages and returns a Response.Data object for the first page.

Sessions

- Session support is a parameter of `Start`
- Each user has a session
  - A cookie is sent to the client = session number
- Allows storing user-specific data
  - a name/value table is associated to each session

```
function Service (Request : in Status.Data) return Response.Data is
   Session_ID : constant Session.ID := Status.Session (Request);
   C : Natural := 0;
begin
   if Session.Exist (Session_ID, "counter") then
      C := Session.Get (Session_ID, "counter");
   end if;
   C := C + 1;
   Session.Set (Session_ID, "counter", C);
```

Provided Dispatchers (1)

- URI dispatcher
  - Dispatches to other functions according to the URI
  - Adds the methods:

```
procedure Register        (Dispatcher: in out Handler;
                           URI       : in String;
                           Action    : in Response.Callback);
procedure Register_Regexp (Dispatcher: in out Handler;
                           URI       : in String;
                           Action    : in Response.Callback);
```

  - similar methods with a Dispatcher parameter

```
procedure Register_Default_Callback
              (Dispatcher: in out Handler;
               Action    : in AWS.Dispatchers.Handler'Class);
procedure Unregister (Dispatcher: in out Handler;
                      URI       : in String);
```

Provided Dispatchers (2)

- Page dispatcher
  - Considers the URI as a file name and returns the corresponding file. Parses 404.thtml if not found.
- Method dispatcher
  - Dispatches to other functions according to the HTTP method.
  - Use: ???
- Virtual host dispatcher
  - Dispatches to other functions according to the host name
- Time dispatcher
  - Associates various functions to different periods of time, and dispatches according to the time of the request.

Provided Dispatchers (3)

- Transient pages dispatcher
  - Linked to another dispatcher
  - If the other dispatcher replies "404", tries to interpret the URI as a transient page.
- SOAP dispatcher
  - Provides two call-backs, one for HTTP requests, one for SOAP requests.

More on Building Responses

- Other `Build` functions
  - From Unbounded_String
  - From a Stream_Element_Array (Allows stream attributes)
- Other URL
  - Redirection (Tells the browser to request another page)

```
AWS.Response.URL (<new URI>);
```

  - New location (Tells the user that the page has moved (301)

```
AWS.Response.Moved (<new URI>, <Message>);
```

- Acknowledge
  - Can be used to return a message with any code
    - In practice, used for error messages

```
AWS.Response.Acknowledge (<Code>, <Message>, <Mime_Type>;
```

AWS Streams

- A type derived from `Resources.Streams.Stream_Type`
  - *NOT* an `Ada.Streams.Root_Stream_Type`!
- How to use it:
  - Declare your type, implement primitive operations:

    ```
    type SQL_Stream is new Resources.Streams.Stream_Type;
    procedure Read (...) is ...
    function  End_Of_File (...) return Boolean is ...
    procedure Close (...) is ...
    ```

  - Add operations to build data into the stream
  - Return response:

    ```
    return Response.Stream (MIME.Text_HTML, Stream_Object);
    ```

- Predefined streams:
  - Memory, Memory.ZLib, Disk, Disk.Once

File Upload

- Sending a file from the client to the server
  - Include a form with a "FILE" entry:

```
<FORM enctype="multipart/form-data" action="/whatever"
      method=POST>
File to process: <INPUT type=FILE name=filename >
                 <INPUT type=SUBMIT name=go value="Send file">
</FORM>
```

POST required

- AWS:
  - Transfers the file into the upload directory
  - Gives it a (local) unique name
  - Makes *two* "filename" parameters:
    - Get (P, "filename", 1) => Full local (server) pathname
    - Get (P, "filename", 2) => Full remote (client) pathname

Push

- A word of caution:
  - Push is updating client data without client request
  - Push keeps an open socket for each client
  - In general, it is better to use a refresh (client pull)
- Principle:
  - Instantiate `AWS.Server.Push` with the data types to send
  - Declare an `Push.Object` object
  - Register clients
  - Send data to clients when needed
    - To a named client
    - Broadcast (all clients)

Status Page

- There is a special status page which is processed directly by AWS.
  - Its name can be chosen or configured
  - Response is built by parsing the template "aws_status.thtml" (redefinable)
  - Provides information about the state of AWS itself
- Use package `AWS.Server.Status`

Configuration

- Many things can be configured…
  - Important parameters can be given in the Start procedure
  - An alternate Start procedure uses a configuration object (`AWS.Config.Object`)
    - All parameters in a configuration object can be set or queried
    - There are defaults for everything
- Configuration is initialized from:
  - aws.ini: for all applications started from the same directory
  - <progname>.ini: for application <progname>
  - A configuration object can be initialized from a file

Configuration data

- Some examples of what can be configured:
  - Admin_URI: the status page name
  - Certificate: name of certificate file for secure servers
  - Down_Image: Name of the "down" image in the status page
  - Log_File_Directory: where to store log files
  - Max_Connection: number of simultaneous connections
  - Server_Port: the port to connect to
  - Upload_Directory: where to store uploaded files
  - And many more…

Authentication

- Identify a user with a Name/Password
- If a page requires authentication:
  - Check if request includes authentication data
    - User name not empty (function `Authorization_Name`)
  - If not:
    - return a 401 response (function `Response.Authenticate`)
    - the response includes a "realm" (a root URL)
    - browser will show a login box and resubmit request
  - All subsequent requests under the "realm" will include authentication data

Two Kinds of Authentication

- Basic (insecure), HTTP 1.0
  - passwords are transmitted without encryption
  - can be considered secure with HTTPS
  - functions `Authorization_Name` and `Authorization_Password`
- Digest (secure), HTTP 1.1
  - passwords are not transmitted
  - an MD5 checksum of Name, Password (and other fields) is transmitted
  - functions `Authorization_Name` and `Check_Digest`

Logging

- package AWS.Log
  - facilities for logging Status and Response data
  - Start, Stop, Flush (or use Auto_Flush)
  - Modes: None, Each_Run, Daily, Monthly
  - File <prefix>-Y-M-D.log
- package AWS.Server.Log
  - Used to automatically log AWS requests
- Log file format

```
<client IP> - <auth name> - [date-time] "<request>" <code> <size>
```

  - For example:

```
100.99.12.1 -  - [14/Jun/2004:11:44:14] "GET /myserver" 200 2347
```

  - This is the format used by Apache!

Secure Server (HTTPS)

- Just set Security to True in the call to "Start"
  - Uses a default certificate
  - To use another certificate:

```
AWS.Server.Set_Security (Certificate_Filename => "/xyz/aws.cert"
```

- Protocols
  - Supported : SSLv2, SSLv3
  - Unsupported : TLSv1
- Why use HTTP?
  - HTTPS is slightly slower
  - HTTPS is very hard to configure… with Apache!

Mailing (SMTP)

- Packages AWS.SMTP, AWS.SMTP.Client
  - Declare server to handle the SMTP protocol.
  - Send the mail

```
   My_Mailer : SMTP.Receiver
      := SMTP.Client.Initialize ("mailhost.axlog.fr");
   Result : SMTP.Status;
begin
   SMTP.Client.Send
        (Server  => My_Mailer,
         From    => SMTP.E_Mail ("Rosen", "rosen@adalog.fr"),
         To      => SMTP.E_Mail ("Obry",  "pascal@obry.org"),
         Subject => "Latest AWS news",
         Message => "The tutorial is doing fine!",
         Status  => Result);
```

  - Other procedures for attachments, multiple recipients…

Mailing (POP)

- Package AWS.POP
  - Declare a Mailbox object (initialize function to set server, user name and password)
  - Various operations to:
    - Get the number of messages, total size of messages
    - Get an message by number (with/without deleting)
    - Delete a message from server
    - Iterate (passive iterators) over all messages, all headers.
    - Get attachments (individually, passive iterator)
    - Get various parts of a message (Header, Content, From, CC…)
- A simple Webmail server is provided as an AWS callback

Miscellaneous Services

- Directory browser (`Services.Directory`)
  - Builds a translate table containing directory information
  - Builds a page containing directory information
    - A template must be specified
    - A default template is provided
- URL
  - Operations to parse the various parts of a URI
  - URI encoding
- MIME
  - Constants for common MIME types.
  - Function to guess the MIME type of a file name (from extension)
- Translator
  - Base 64 Encode/Decode
  - zlib Compress/Decompress
- Exceptions
  - Call-back for unexpected exceptions caught by AWS

Deploying an AWS Server

- Resources
  - It is possible to include any file (HTML, Images, icons, templates…) used by the Web server into the server executable.
  - Resources are compiled with awsres.
    - Creates a hierarchy of packages, one for each resource
    - Resources can be compressed
    - Just "with" the root package
- No Web server is easier to distribute, install and launch !

A single, self contained Web server executable

- Introduction
- Internet
- AWS basics
- The templates parser
- AWS advanced
- Distributed applications with AWS
- AWS in practice
- Conclusion

AWS for Distributed Computing

- Exchanging simple data:
  - Simple communication
  - HTTP
- Distributed server:
  - Hotplugs
- Remote services:
  - SOAP
  - LDAP
  - JABBER
- And you can still use Annex E in addition…

Simple Communication (1)

- Simple exchange of (string) data over HTTP/GET

- Client side (AWS.Communication.Client):

```
function Send_Message
  (Server     : in String;
   Port       : in Positive;
   Name       : in String;
   Parameters : in Parameter_Set := Null_Parameter_Set)
return Response.Data;
```

Array of Unbounded_String

- Sends a message like:

```
http://<Server>:<Port>/AWS_Com?HOST=<host>&NAME=<name>
&P1=<param1>&P2=<param2>
```

Simple Communication (2)

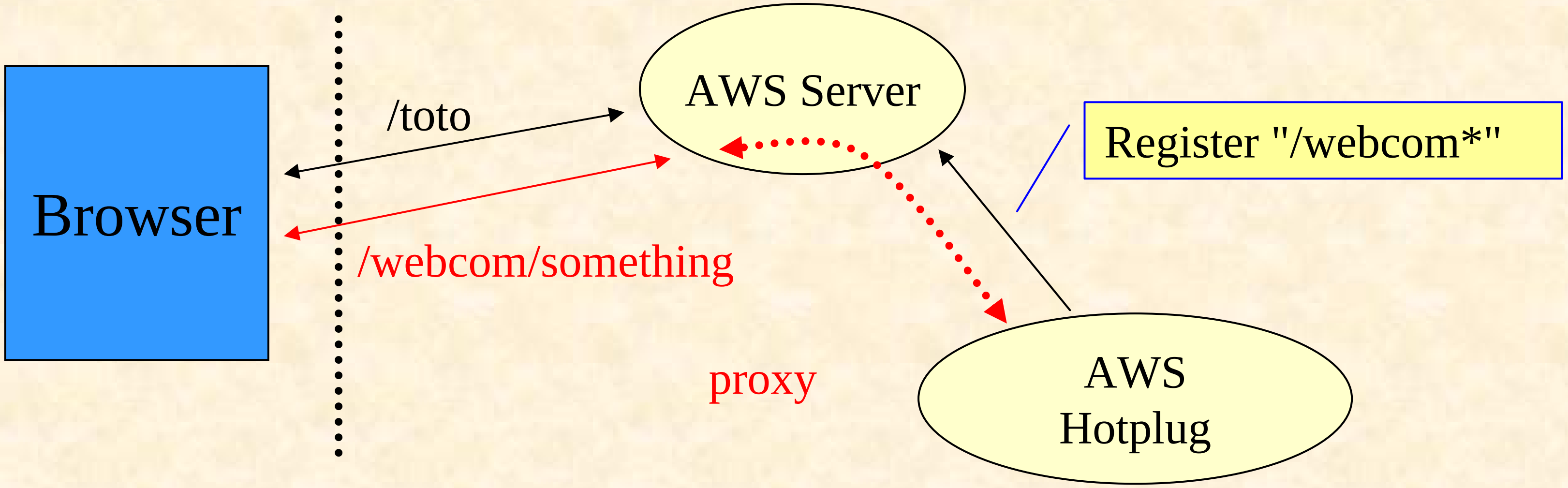
- Server side (AWS.Communication.Server):
  - Instantiate:

```
with AWS.Response;
generic
   type T (<>) is limited private;
   type T_Access is access T;
   with function Callback(Server     : in String;
                          Name       : in String;
                          Context    : in T_Access;
                          Parameters : in Parameter_Set)
      return Response.Data;
package AWS.Communication.Server is
   procedure Start (Port    : in Positive;
                    Context : in T_Access);
   procedure Shutdown;
end AWS.Communication.Server;
```

  - The context is used to keep information between calls

Hotplugs (1)

- A way to have a Web server split on multiple machines.
  - Managing several databases
  - Load balancing…



Hotplugs (2)

- Server side:
  - Activate the functionality

```
AWS.Server.Hotplug.Activate(WS'Access, 2222);
```

Port used for communication

- Client side:
  - Register by sending a message with the simple communication

```
Response := AWS.Communication.Client.Send_Message
               ("The_Server", 2222,
                AWS.Server.Hotplug.Register_Message,
                AWS.Communication.Parameters
                     ("/webcom*", "http://The_Client:1235"));
```

  - It is possible to Unregister at any time:

```
Response := AWS.Communication.Client.Send_Message
               ("The_Server", 2222,
                AWS.Server.Hotplug.Unregister_Message,
                AWS.Communication.Parameters ("/webcom*"));
```

HTTP Client

- AWS.Client

```
function Get    (…) return Response.Data;
function Head   (…) return Response.Data;
function Put    (…) return Response.Data;
function Post   (…) return Response.Data;
function Upload (…) return Response.Data;
```

- Authentication parameters can be passed
  - Only basic authentication supported currently
- Facilities for Keep_Alive
  - Define an HTTP_Connection object
  - All requests use the same connection object

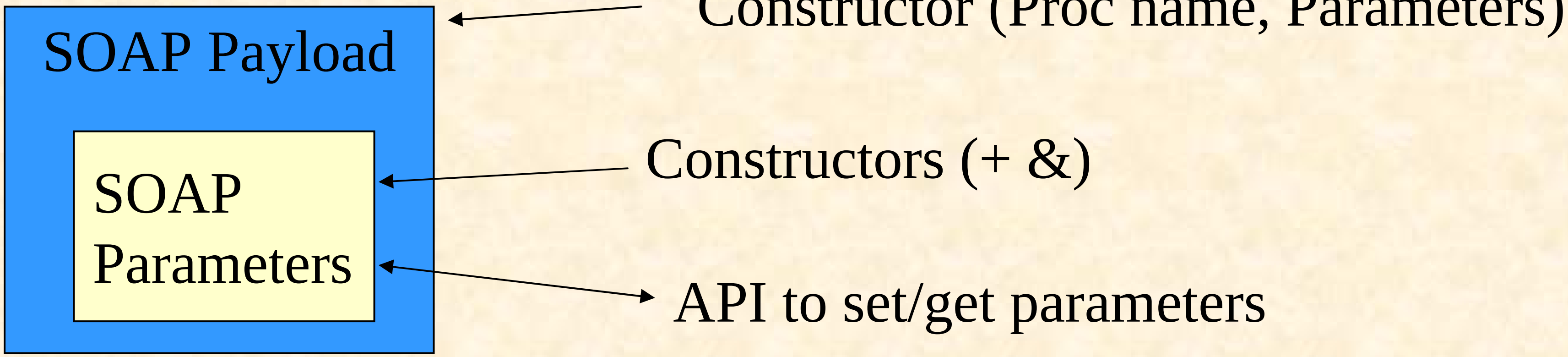
SOAP

- Simple Object Access Protocol
- Provide Request/Response protocol
  - Typically a client/server protocol
  - SOAP 1.1 implementation
  - HTTP protocol for transport
  - Validation via http://validator.soapware.org/
- Message is in XML format
  - We don't care, AWS does the job for us
- SOAP is simple, binding is not !

Designed for ease of use.

AWS/SOAP Inside

- A SOAP message is sent to a URI
- An HTTP header identifies a SOAPAction
- A SOAP message includes a payload
  - + a procedure name
  - + parameters.


SOAP Payload
SOAP Parameters
Constructor (Proc name, Parameters)
Constructors (+ &)
API to set/get parameters

Supported SOAP Types

- Support all base types
  - xsd:int, xsd:float, xsd:string, xsd:boolean, xsd:timeInstant, xsd:null
- Support base64 type
  - SOAP-ENC:Base64
- Support Struct
- Support Array
  - SOAP-ENC:Array

SOAP Parameters

- Constructors in SOAP.Types

`I (value, "name")` to build a xsd:int

`F (value, "name")` to build a xsd:float

…

- SOAP.Parameters.List
  - Operator "+" to convert a type to a list
  - Operator "&" to add parameters to the list

SOAP Example

- Ada function :

```
function This_Proc(P1 : in Integer;
                   P2 : in Integer;
                   P3 : in Float)
return Integer;
```

- Translated to :
  - A SOAP Message
  - A SOAP Response

SOAP Message

```
POST /examples HTTP/1.1
User-Agent: Radio UserLand/7.0 (WinNT)
Host: localhost:81
Content-Type: text/xml; charset=utf-8
Content-length: 474
SOAPAction: "/examples"

<?xml version="1.0"?>
<SOAP-ENV:Envelope
   SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
   xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">

   <SOAP-ENV:Body>
      <m:This_Proc xmlns:m="http://www.soapware.org/">
         <p1 xsi:type="xsd:int">10</p1>
         <p2 xsi:type="xsd:int">32</p2>
         <p3 xsi:type="xsd:float">12.4</p3>
      </m:This_Proc>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

procedure name

parameters

SOAP Message Response

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/xml; charset=utf-8
Content-length: 420
Date: Wed, 28 Mar 2001 05:05:04 GMT
Server: UserLand Frontier/7.0-WinNT

<?xml version="1.0"?>
<SOAP-ENV:Envelope
   SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
   xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">

   <SOAP-ENV:Body>
      <m:This_ProcResponse xmlns:m="http://www.soapware.org/">
         <myres xsi:type="xsd:int">42</myres>
      </m:This_ProcResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

result

SOAP Client

```
   P_Set : Parameters.List := +I (10, "p1") & I (32, "p2")
                                            & F (12.4, "p3");
   P     : Message.Payload.Object;
begin
   P := Message.Payload.Build ("This_Proc", P_Set);

   declare
      R : constant Message.Response.Object'Class
         := SOAP.Client.Call ("http://host:8080/soapdemo", P);

      P : constant Parameters.List := SOAP.Message.Parameters (R);

      My_Res : constant Integer := SOAP.Parameters.Get (P, "myres");
```

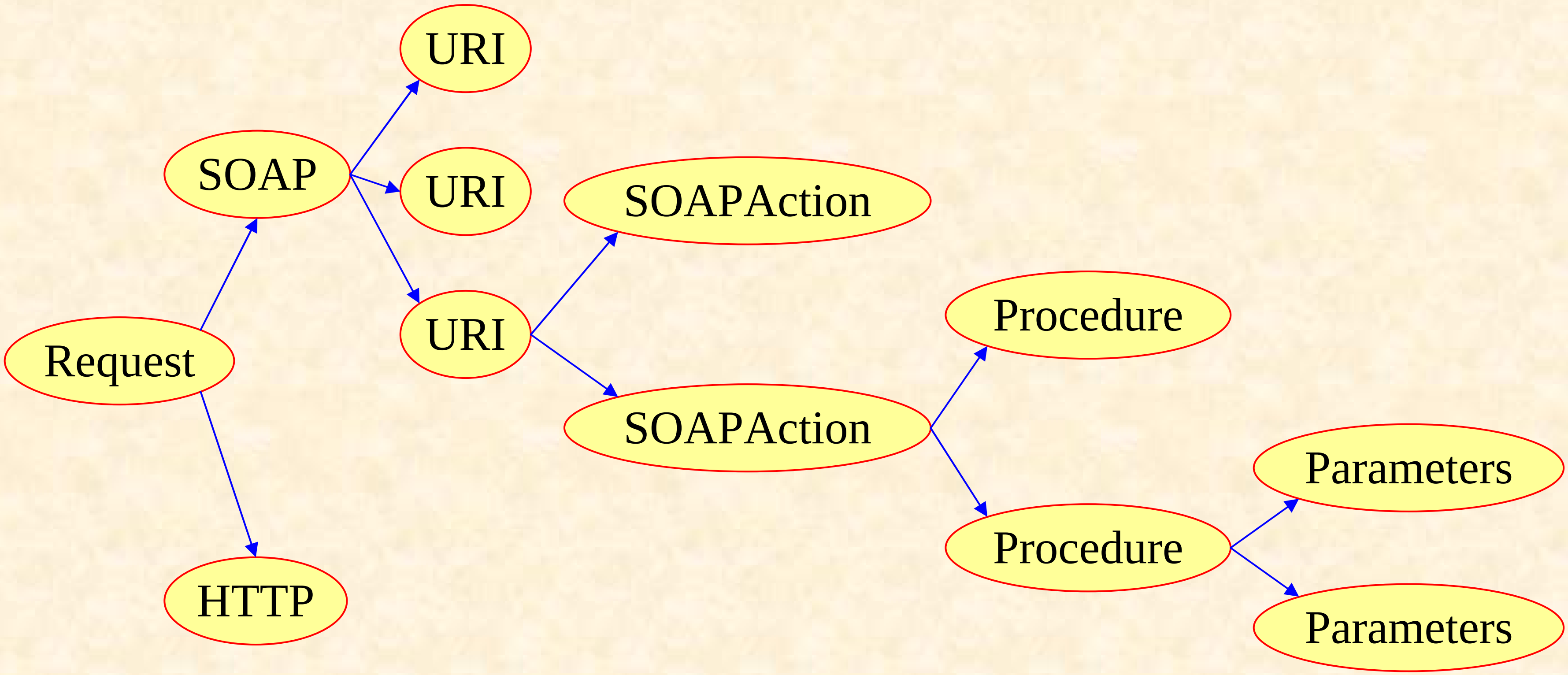
Default value for SOAPAction is **URL#PROC**
(http://host:8080/soapdemo#This_Proc)

SOAP Server

```
function SOAP_CB (Request : in AWS.Status.Data)
   return AWS.Response.Data
is
   use SOAP, SOAP.Types, SOAP.Parameters;
   PL : constant Message.Payload.Object
        := Message.XML.Load_Payload (AWS.Status.Payload (Request));
   P  : constant Parameters.List := Message.Parameters (PL);
   R  : Message.Response.Object;
   RP : Parameters.List;
begin
   R := Message.Response.From (PL);
   declare
      P1 : constant Integer := SOAP.Parameters.Get (P, "p1");
      P2 : constant Integer := SOAP.Parameters.Get (P, "p2");
   begin
      RP := +I (P1 + P2, "myres");
   end;

   SOAP.Message.Set_Parameters (R, RP);
   return Message.Response.Build (R);
```

SOAP dispatching

- Many degrees of freedom!
  - It is not necessary to consider all of them…



SOAP Server dispatching

1. Determine that it is a SOAP request
   - can use the SOAP dispatcher
2. Dispatch according to URI
   - can use a regular URI dispatcher

Often not necessary

3. Get the SOAPAction from the status object and dispatch.

```
SOAPAction : constant String := Status.SOAPAction (Request);
```

   - SOAPAction can be interpreted as an "object" name
4. In the SOAP routine, retrieve the SOAP procedure name and dispatch to the appropriate routine.

```
Payload : constant SOAP.Message.Payload.Object
   := SOAP.Message.XML.Load_Payload(Status.Payload (Request));
Proc : constant String
   := SOAP.Message.Payload.Procedure_Name (Payload);
```

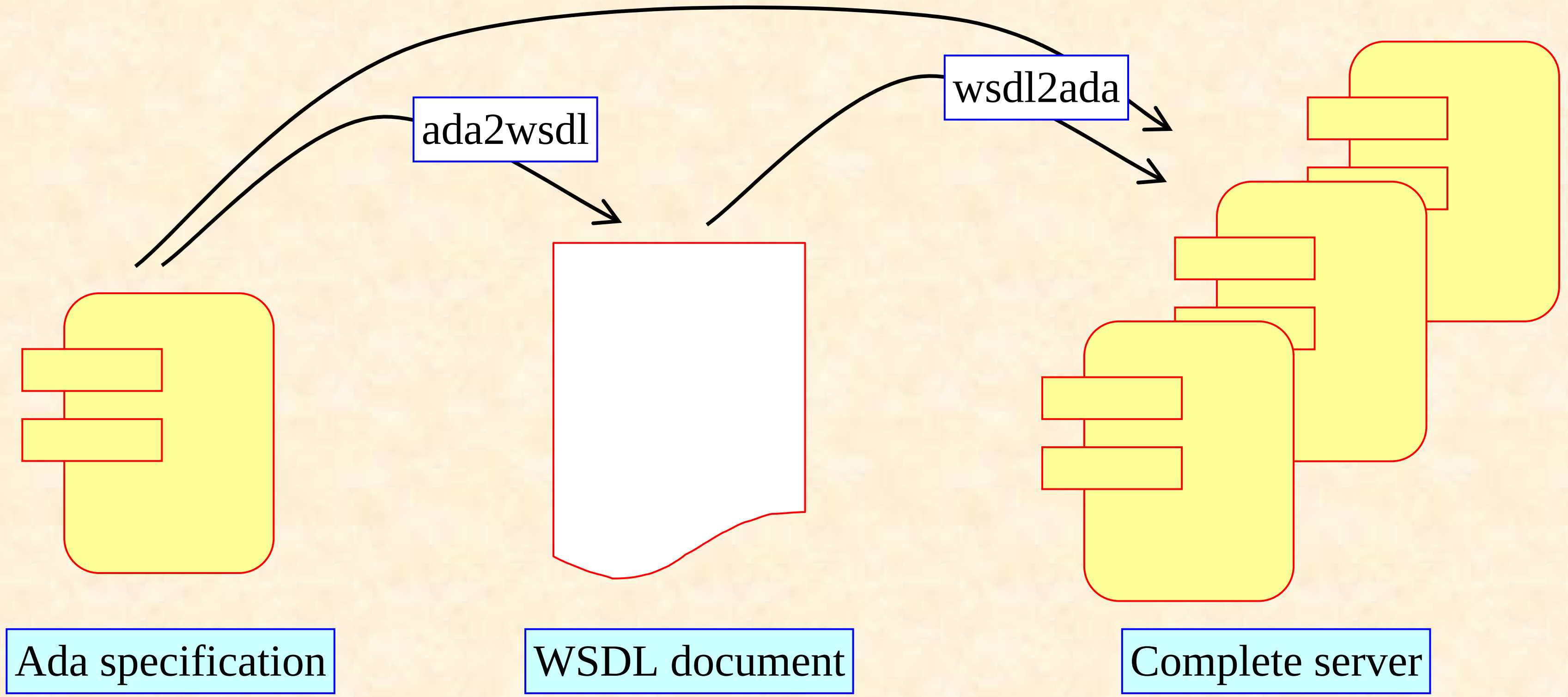
5. The routine deals with the parameters

WSDL Interface

- WSDL
  - An XML based document describing a SOAP service
  - Developed jointly by Microsoft and IBM.
    - To be endorsed (not yet) by W3C
  - Binding to SOAP 1.1, HTTP GET/POST, and MIME
- wsdl2aws
  - Automatically generates client stubs to provide access to a service.
  - Automatically generates server skeletons to create a service.
- aws2wsdl
  - Automatically generates a WSDL document from an Ada specification

Writing a SOAP/WSDL server

- aws2wsdl and wsdl2aws work together!



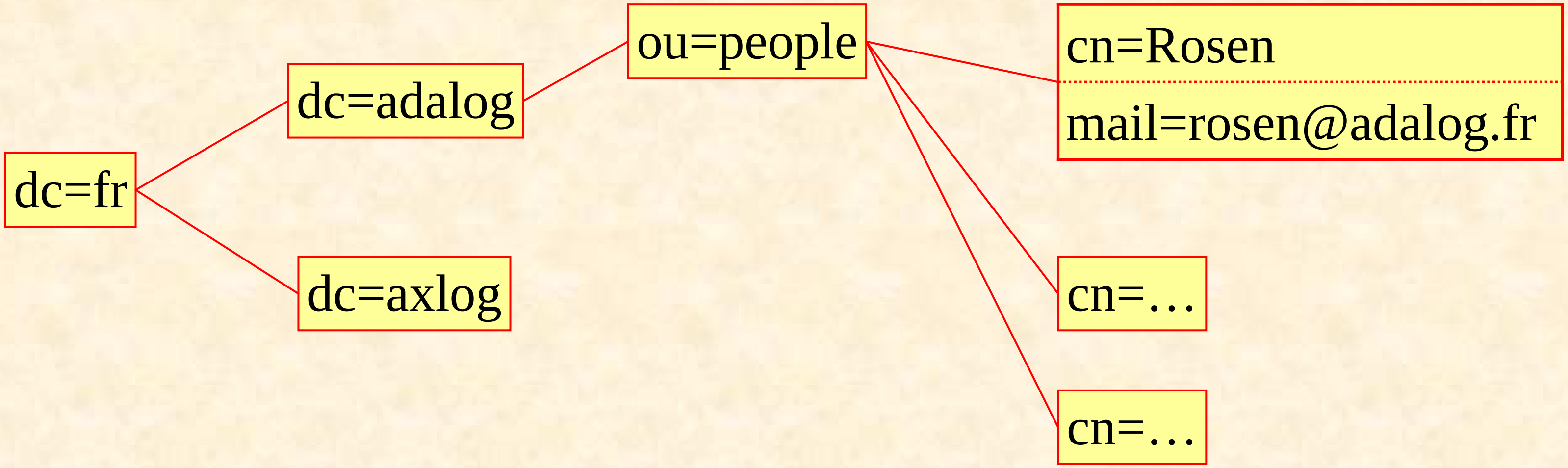
LDAP (1)

- Lightweight Directory Access Protocol
  - A lightweight subset of DAP (X.500)
  - A means of serving data on individuals, system users, network devices and systems over the network
  - Example: DNS
- A remotely callable database interface
  - Based on *entries*:

```
cn = test entry
cn = another commonName value for test entry
mail = entry@someHost.someDomain
```

LDAP (2)

- A hierarchical database:



- Entries are retrieved by giving the value of an attribute: the DN (Distinguished Name)
  - `"cn=rosen, ou=people, dc=adalog, dc=fr"`

LDAP Client

- AWS implementation:
  - +Client only, no modification or deleting of data
  - +If someone volunteers to provide more functionalities…
- Usage summary:

```
   Directory : LDAP.Client.Directory := Init (Host);
begin
   Bind (Directory, "", "");
   declare
      Response_Set := Search (Directory,
                              Base_DN,
                              Filter,
                              LDAP_Scope_Subtree,
                              Attributes ("cn", "mail"));
   begin
      -- Iterate through responses
         -- Iterate through attributes
```

Username, password

JABBER

- A "chat" protocol (immediate messaging)
  - Exchange messages between users connected to a JABBER server
- AWS implementation:
  - Check presence of a user

```
procedure Check_Presence (Server : in      Jabber.Server;
                          JID    : in      String;
                          Status :    out Presence_Status);
```

  - Send message to a user

```
procedure Send_Message (Server  : in Jabber.Server;
                        JID     : in String;
                        Subject : in String;
                        Content : in String);
```

  - If someone volunteers to provide more functionalities…

Sufficient to send alerts to a connected user

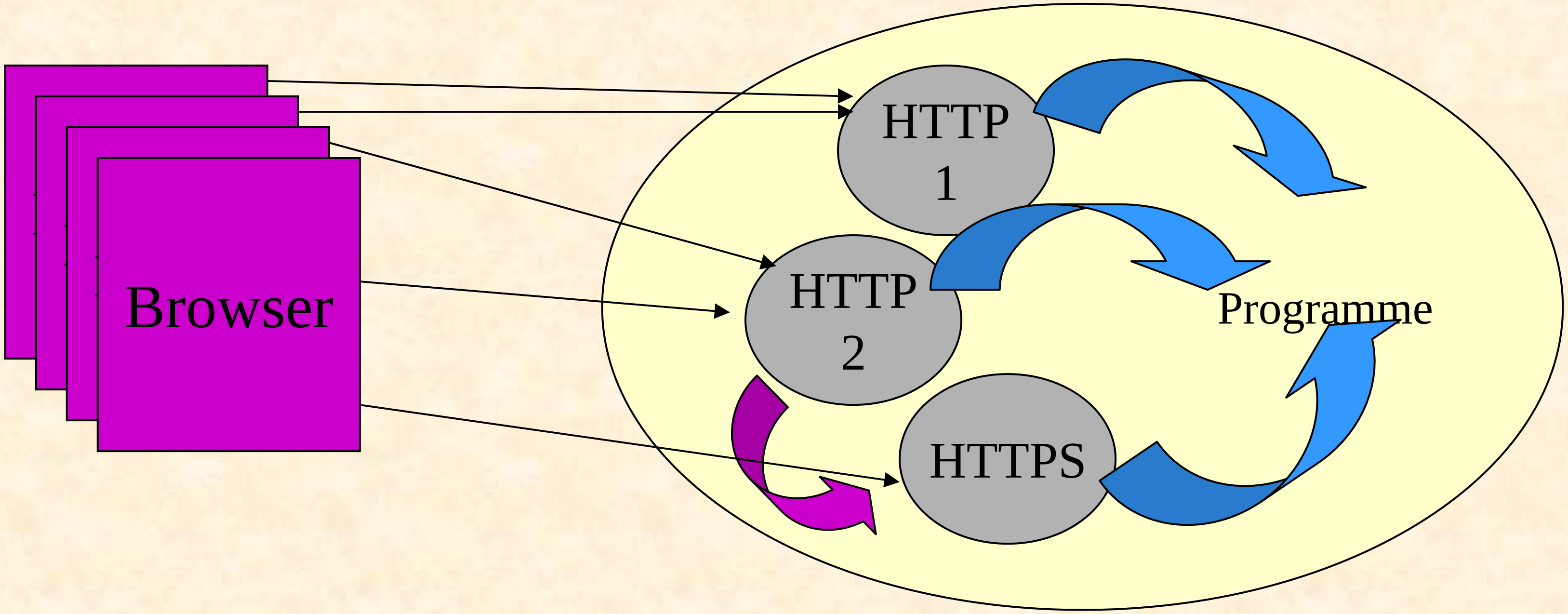
- Introduction
- Internet
- AWS basics
- The templates parser
- AWS advanced
- Distributed applications with AWS
- **AWS in practice**
- Conclusion

What Can AWS Be Used For?

- HTTP services
  - Lightweight page server
    - A full web server is another story…
  - Virtual site
- HTML as a Graphical User Interface
- Regular application with Web access
  - Remotely monitoring a process, an experiment…
- Client-server applications
  - HTTP communication
  - SOAP

Why a Single Server ?

- It is possible to have more than one server in the same program.



Maintaining User State (1)

State as page parameters

- Each "page" has a different URI
- Pages can be bookmarked
- "Back" button works
- Allows direct links
- No global state in the program
- But
  - Data not really hidden
  - User can provide "bad" URIs
  - URI can become too long

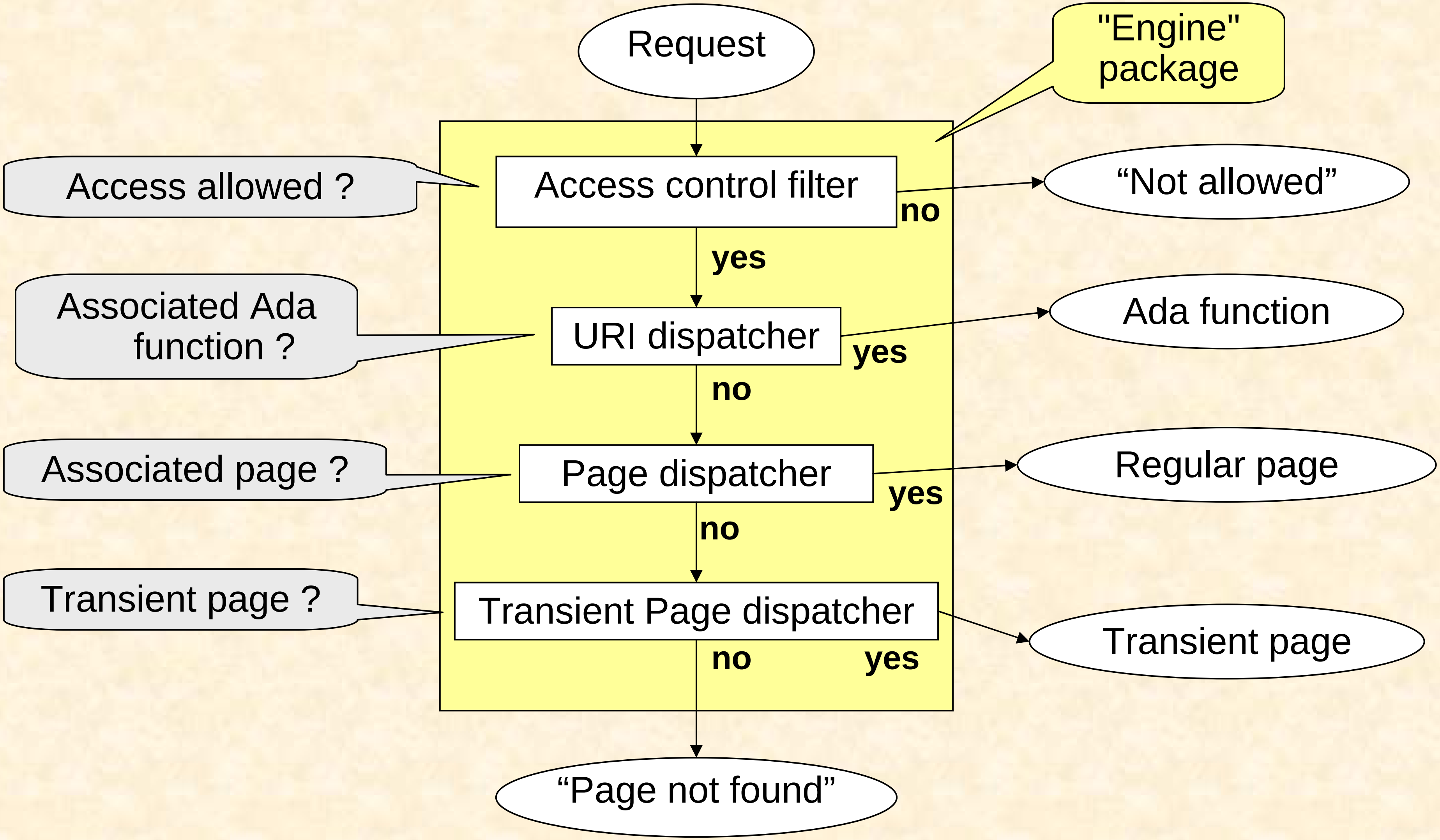
Maintaining User State (2)

- State as session data
  - + Only one URI appears in the browser
  - + Better control over user's behaviour
  - + Session data can be kept when the server is restarted
    - "Hot" restart
  - + But
    - Client must accept cookies
    - "Surprising" behaviour with "back" button

Gesem's Implementation

- Unusual constraints:
  - Use free software
  - User interface usable by casual users
  - Availability on Windows and Linux
  - Independent of any particular DBMS
  - Easily modifiable
  - Deal with concurrent accesses
  - Efficiency *is not* a concern
  - Reliability *is* a concern

Gesem Filters and Dispatchers



The Page Design Pattern

```
with AWS.Response;
package Pages.Some_Page is
  function URI (<parameters>)return String;
end Pages.Some_Page;


package body Pages.Some_Page is
  My_Name : constant String := "some_page";

  function Build (<Parameters>)
    return Response.Data is ...

  function Buttons (Request : in AWS.Status.Data)
    return Response.Data is ...

  function Page (Request : in AWS.Status.Data)
    return Response.Data is ...

  function URI (<parameters>)return String is ...
begin
  Engine.Register(My_Name, (Root    => Page'Access,
                            Buttons => Buttons'Access));
end Pages.Some_Page;
```

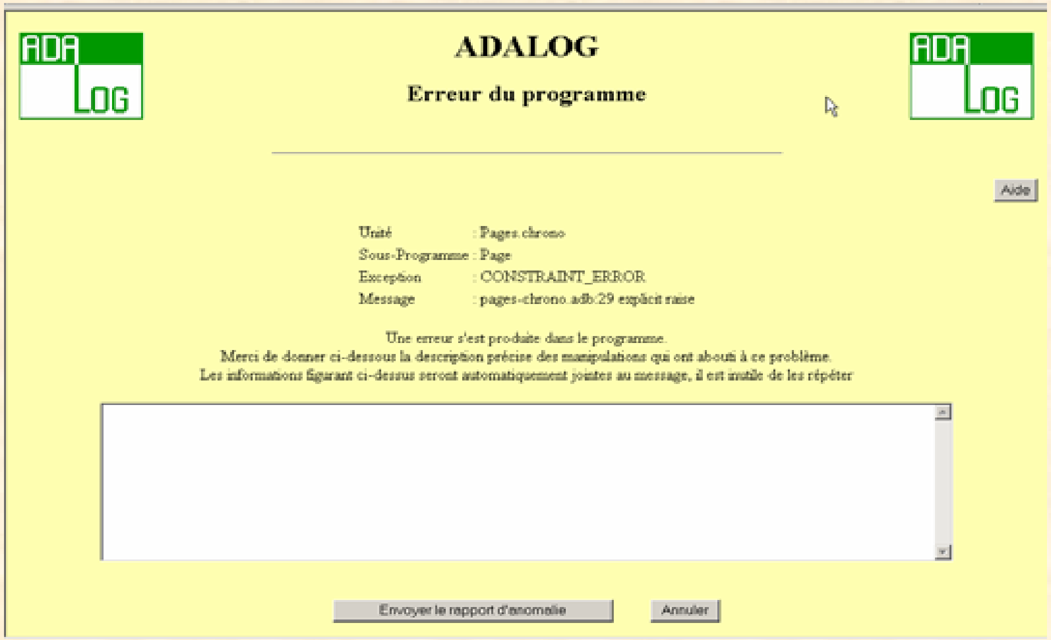
some_page.html

some_page.btns

Reliability

- Every page has an exception handler:

```
exception
   when Occur : others =>
      return URL (Pages.Error.Build
                       (Unit       => "pages." & My_Name,
                        Subprogram => “Name of subprogram",
                        Occur      => Occur));
```



ADALOG

Erreur du programme

Unité : Pages.chrono
Sous-Programme : Page
Exception : CONSTRAINT_ERROR
Message : pages-chrono.adb:29 explicit raise

Une erreur s'est produite dans le programme.
Merci de donner ci-dessous la description précise des manipulations qui ont abouti à ce problème.
Les informations figurant ci-dessus seront automatiquement jointes au message, il est inutile de les répéter

Concurrency

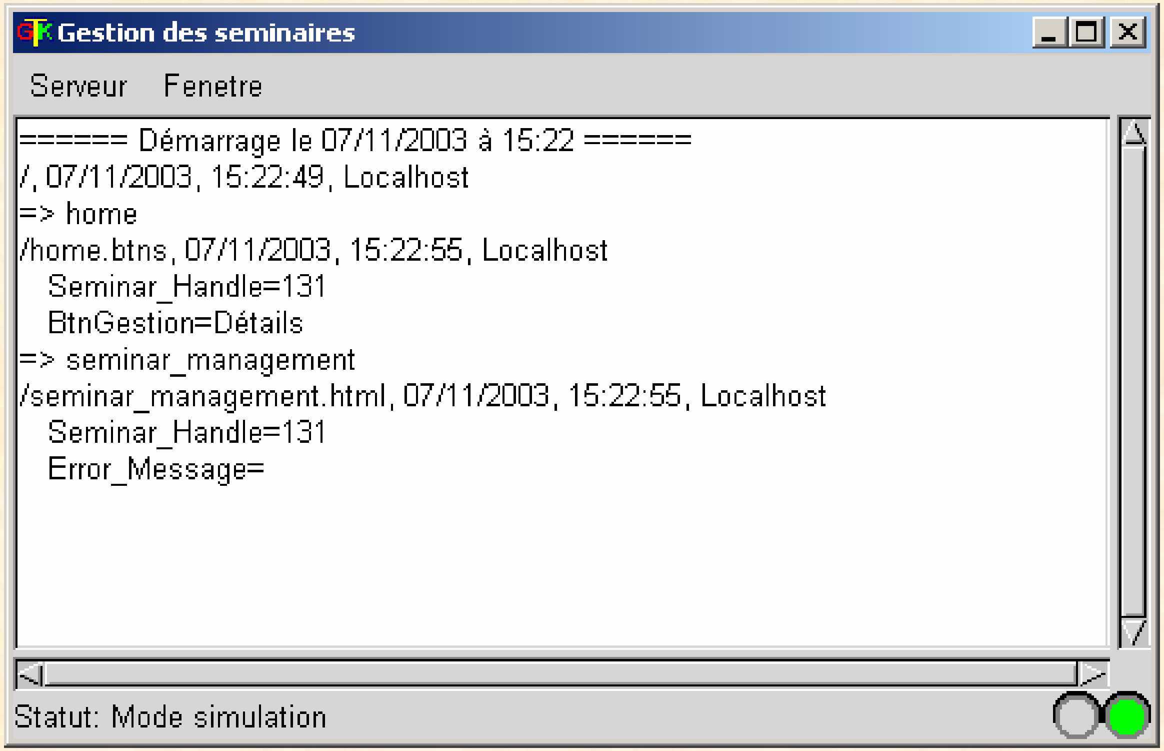
- Concurrent access is extremely unlikely, but possible
  - Recognize users from their IP address
  - Use a global lock:
    - Only one user can modify at any one time
    - "Modify" button on each page to grab the lock
- But beware of "back" button
  - Display a page
  - Modify it (get lock)
  - Validate (release lock)
  - Back page: the page is modifiable, but the user doesn't own the lock !
  - Checked by the access control filter => page expired

Local Interface

- Manages the application
  - Stop, lock database…
  - Shows uncommitted transactions
- Monitors requests
  - Clear window
  - Save content to file

•Plain GTK

•Generated automatically with GLADE



```
====== Démarrage le 07/11/2003 à 15:22 ======
/, 07/11/2003, 15:22:49, Localhost
=> home
/home.btns, 07/11/2003, 15:22:55, Localhost
   Seminar_Handle=131
   BtnGestion=Détails
=> seminar_management
/seminar_management.html, 07/11/2003, 15:22:55, Localhost
   Seminar_Handle=131
   Error_Message=
```

Statut: Mode simulation

Objects Design Pattern

```
with Globals, Data_Manager, AWS.Templates;
use Globals;
package Objects.Abstraction is
   type Data is
         record
            …
         end record;
      -- Operations on Abstraction.Data

   function Image (Item : Data)return Array_Of_Unbounded;
   function Value (Item : Array_Of_Unbounded)return Data;
   package Manager is new Data_Manager
     (Data      => Data,
      Data_Name => "my_data",
      Columns   => "col1, col2, col3");
   subtype Handle is Manager.Handle;
   type List is array (Positive range <>)of Handle;

   function Associations (Item : Handle) return Translate_Table;
   function Associations (Item : List)   return Translate_Table;
   function Extract (Param : AWS.Parameters.List) return Data;
end Objects.Abstraction;
```

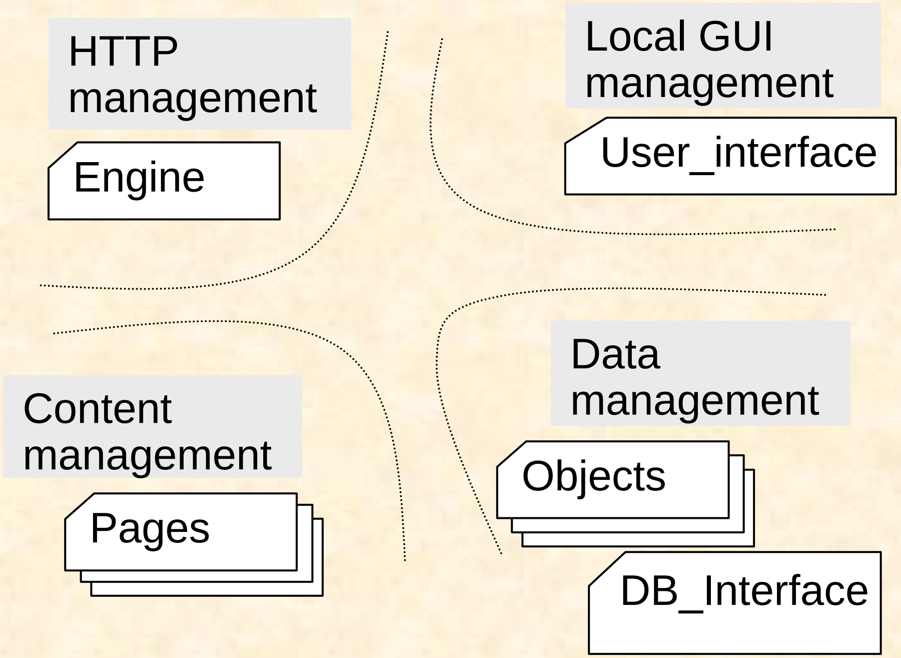
Ada view

Database view

Templates (HTML) view

Lessons learned (1)

- Separate concerns



HTTP management

Engine

Local GUI management

User_interface

Content management

Pages

Data management

Objects

DB_Interface

- Reliability
  - + Exceptions are great!
- AWS is powerful enough
  - + No Javascript, no Java
  - + The template parser is great!

Lessons learned (2)

- A web interface is difficult to manage
  - User can close the browser at any time (even with uncommitted transactions), but the application is not aware!
  - User can call "previous page" at any time: no global state
- Portability
  - > 10_000 SLOC in 81 compilation units
  - Network interface + GUI + Database interface
  - **No** difference between Linux and Windows version
  - Ada is great!

- Introduction
- Internet
- AWS basics
- The templates parser
- AWS advanced
- Distributed applications with AWS
- AWS in practice
- Conclusion

Installing AWS

- Prerequisites:
  - Gnat (other compilers in AWS 2.0)
  - Windows: cygwin shell or equivalent (bash)
  - Unix: OpenSSL (optional), OpenLDAP (optional)
  - For SOAP: XMLAda
- Installation:
  - Read the document (rather than INSTALL file!)
  - Set variables in makefile.conf
    - Windows: use Dos syntax for file names
  - make build
  - make install

AWS vs. Other Technologies (1)

- The application is a single executable, not a set of scripts
  - Must recompile when functionnalities are added/changed
  - NOT when presentation changes (thanks to templates)
- Separate processing from display
  - Unlike servlets
- Easy to deal with concurrent access
  - Thanks to protected types!
- What's difficult with Apache made easy
  - HTTPS, logs, …

AWS vs. Other Technologies (2)

- Efficiency
  - No need to start a process for each request
- Ease of distribution
  - Simplified deployment (no Web server to install and configure, a single executable to install).
- Mixed applications
  - When the Web interface is only part of the application
  - Possibility of having a control panel

AWS Usage (1)

- Users
  - EDF/R&D
    - WORM (shared bookmark)
    - Internet share
  - Adalog
    - Gesem
  - SETI@Home module
    - Ted Dennison (Open Source) – 1 to 3 millions users.
  - ACT
    - Gnat tracker
  - Ada-Russia (http://www.ada-ru.org)

AWS Usage (2)

- More users
  - Frontend to access Oracle via a Web interface.
  - DOCWEBSERVER and OESM
    - Overall Equipment Status Monitoring - Wiljan Derks (Philips).
  - Currency change
    - Dmitriy Anisimkov. (40 to 50 requests per second !).
- Statistics
  - ≅ 300 users
  - A mailing-list with 87 people.

Conclusion

- A mature technology
- AWS is more than a Web server
  - Full HTTP API
    - Communication (client/server).
    - Sessions
    - PUSH
  - Other protocols:
    - SOAP
    - SMTP / POP / LDAP / Jabber
  - More than a simple server
    - Several servers, hotplugs
    - Virtual hosts
    - distributed computing

A complete Web development framework

Questions