

Ada Web Server User's Guide

AWS - version 2.8.0

Support for SOAP 1.1 - version 1.5.0
SMTP, POP, LDAP and Jabber protocols.

Date: 6 June 2010

AdaCore

Copyright © 2000, Pascal Obry

Copyright © 2001, Pascal Obry, Dmitriy Anisimkov

Copyright © 2002-2010, AdaCore

This document may be copied, in whole or in part, in any form or by any means, as is or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Table of Contents

1	Introduction	1
2	Building AWS	3
2.1	Requirements	3
2.2	AWS.Net.Std	3
2.3	Building	3
2.4	Building on cross-platforms	4
2.5	Demos	4
2.6	Installing	6
3	Using AWS	7
3.1	Setting up environment	7
3.1.1	Using environment variables	7
3.1.2	Using GNAT Project Files	7
3.2	Basic notions	8
3.2.1	Building an AWS server	8
3.2.2	Callback procedure	10
3.2.3	Form parameters	11
3.2.4	Distribution of an AWS server	12
3.3	Building answers	12
3.3.1	Redirection	12
3.3.2	New location for a page	12
3.3.3	Authentication required	12
3.3.4	Sending back an error message	13
3.3.5	Response from a string	13
3.3.6	Response from a Stream_Element_Array	13
3.3.7	Response from a file	13
3.3.8	Response from a stream	13
3.3.9	Response from a on-disk stream	14
3.3.10	Response from a on-disk once stream	14
3.3.11	Response from a memory stream	14
3.3.12	Response from a compressed memory stream	15
3.3.13	Split page	15
3.3.14	Response a from pipe stream	15
3.4	Configuration options	15
3.5	Session handling	19
3.6	HTTP state management	20
3.7	Authentication	21
3.8	File upload	22
3.9	Communication	22
3.9.1	Communication - client side	23
3.9.2	Communication - server side	23
3.10	Hotplug module	23
3.10.1	Hotplug module - server activation	24
3.10.2	Hotplug module - creation	24
3.11	Server Push	25
3.12	Working with Server sockets	26

3.13	Server Log	26
3.14	Secure server	28
3.14.1	Initialization	28
3.14.2	Creating a test certificate	28
3.14.3	Protocol	28
3.15	Unexpected exception handler	29
3.16	Socket log	30
3.17	Client side	30
4	High level services	33
4.1	Directory browser	33
4.2	Dispatchers	33
4.2.1	Callback dispatcher	33
4.2.2	Method dispatcher	33
4.2.3	URI dispatcher	33
4.2.4	Virtual host dispatcher	33
4.2.5	Transient pages dispatcher	34
4.2.6	Timer dispatcher	34
4.2.7	Linker dispatcher	34
4.2.8	SOAP dispatcher	34
4.3	Static Page server	35
4.4	Transient Pages	35
4.5	Split pages	36
4.6	Download Manager	36
4.7	Web Elements	38
4.7.1	Installation	38
4.7.2	Ajax	38
4.7.2.1	Steps to do Ajax	39
4.7.2.2	Basic Ajax support	39
4.7.2.3	XML based Ajax	40
4.7.2.4	Advanced Ajax	44
4.8	Web Block Context	45
4.9	Web Cross-References	47
5	Using SOAP	49
5.1	SOAP Client	49
5.2	SOAP Server	50
5.2.1	Step by step instructions	50
5.2.2	SOAP helpers	51
6	Using WSDL	53
6.1	Creating WSDL documents	53
6.1.1	Using ada2wsdl	53
6.1.2	Ada mapping to WSDL	55
6.1.3	ada2wsdl	58
6.1.4	'ada2wsdl' limitations	59
6.2	Working with WSDL documents	59
6.2.1	Client side (stub)	59
6.2.2	Server side (skeleton)	59
6.2.3	wsdl2aws	61
6.2.4	wsdl2aws behind the scene	63
6.2.5	wsdl2aws limitations	63
6.3	Using ada2wsdl and wsdl2aws together	63

7	Working with mails	65
7.1	Sending e-mail	65
7.2	Retrieving e-mail	65
8	LDAP	69
9	Jabber	71
9.1	Jabber presence	71
9.2	Jabber message	71
10	Resources	73
10.1	Building resources	73
10.2	Using resources	73
10.3	Stream resources	73
10.4	awsres tool	73
11	Status page	75
Appendix A	References	79
Appendix B	AWS API Reference	83
B.1	AWS	83
B.2	AWS.Attachments	84
B.3	AWS.Client	85
B.4	AWS.Client.Hotplug	86
B.5	AWS.Communication	87
B.6	AWS.Communication.Client	88
B.7	AWS.Communication.Server	89
B.8	AWS.Config	90
B.9	AWS.Config.Ini	91
B.10	AWS.Config.Set	92
B.11	AWS.Containers.Tables	93
B.12	AWS.Cookie	94
B.13	AWS.Default	95
B.14	AWS.Dispatchers	96
B.15	AWS.Dispatchers.Callback	97
B.16	AWS.Exceptions	98
B.17	AWS.Headers	99
B.18	AWS.Headers.Values	100
B.19	AWS.Jabber	101
B.20	AWS.LDAP.Client	102
B.21	AWS.Log	103
B.22	AWS.Messages	104
B.23	AWS.MIME	105
B.24	AWS.Net	106
B.25	AWS.Net.Buffered	107
B.26	AWS.Net.Log	108
B.27	AWS.Net.Log.Callbacks	109
B.28	AWS.Net.SSL	110
B.29	AWS.Net.SSL.Certificate	111
B.30	AWS.Parameters	112

B.31	AWS.POP	113
B.32	AWS.Resources	114
B.33	AWS.Resources.Embedded	115
B.34	AWS.Resources.Files	116
B.35	AWS.Resources.Streams	117
B.36	AWS.Resources.Streams.Disk	118
B.37	AWS.Resources.Streams.Disk.Once	119
B.38	AWS.Resources.Streams.Memory	120
B.39	AWS.Resources.Streams.Memory.ZLib	121
B.40	AWS.Resources.Streams.Pipe	122
B.41	AWS.Response	123
B.42	AWS.Server	124
B.43	AWS.Server.Hotplug	125
B.44	AWS.Server.Log	126
B.45	AWS.Server.Push	127
B.46	AWS.Server.Status	128
B.47	AWS.Services.Callbacks	129
B.48	AWS.Services.Directory	130
B.49	AWS.Services.Dispatchers	131
B.50	AWS.Services.Dispatchers.Linker	132
B.51	AWS.Services.Dispatchers.Method	133
B.52	AWS.Services.Dispatchers.URI	134
B.53	AWS.Services.Dispatchers.Virtual_Host	135
B.54	AWS.Services.Download	136
B.55	AWS.Services.Page_Server	137
B.56	AWS.Services.Split_Pages	138
B.57	AWS.Services.Split_Pages.Alpha	139
B.58	AWS.Services.Split_Pages.Alpha.Bounded	140
B.59	AWS.Services.Split_Pages.Uniform	141
B.60	AWS.Services.Split_Pages.Uniform.Alpha	142
B.61	AWS.Services.Split_Pages.Uniform.Overlapping	143
B.62	AWS.Services.Transient_Pages	144
B.63	AWS.Services.Web_Block	145
B.64	AWS.Services.Web_Block.Context	146
B.65	AWS.Services.Web_Block.Registry	147
B.66	AWS.Session	148
B.67	AWS.SMTP	149
B.68	AWS.SMTP.Client	150
B.69	AWS.Status	151
B.70	AWS.Templates	152
B.71	AWS.Translator	153
B.72	AWS.URL	154
B.73	SOAP	155
B.74	SOAP.Client	156
B.75	SOAP.Dispatchers	157
B.76	SOAP.Dispatchers.Callback	158
B.77	SOAP.Message	159
B.78	SOAP.Message.XML	160
B.79	SOAP.Parameters	161
B.80	SOAP.Types	162
Index		163

1 Introduction

AWS stand for *Ada Web Server*. It is an Ada implementation of the HTTP/1.1 protocol as defined in the RFC 2616 from June 1999.

The goal is not to build a full Web server but more to make it possible to use a Web browser (like Internet Explorer, or Netscape Navigator) to control an Ada application. As we'll see later it is also possible to have two Ada programs exchange informations via the HTTP protocol. This is possible as **AWS** also implement the client side of the HTTP protocol.

Moreover with this library it is possible to have more than one server in a single application. It is then possible to export different kind of services by using different HTTP ports, or to have different ports for different services priority. Client which must be served with a very high priority can be assigned a specific port for example.

As designed, **AWS** big difference with a standard CGI server is that there is only one executable. A CGI server has one executable for each request or so, this becomes a pain to build and to distribute when the project gets bigger. We will also see that it is easier with **AWS** to deal with session data.

AWS support also HTTPS (secure HTTP) using SSL. This is based on **OpenSSL** a very good and Open Source SSL implementation.

Major supported features are:

- HTTP implementation
- HTTPS (Secure HTTP) implementation based on SSLv3
- Template Web pages (separate the code and the design)
- Web Services - SOAP based
- WSDL support (generate stub/skeleton from WSDL documents)
- Basic and Digest authentication
- Transparent session handling (server side)
- HTTP state management (client side cookies)
- File upload
- Server push
- SMTP / POP (client API)
- LDAP (client API)
- Embedded resources (full self dependant Web server)
- Complete client API, including HTTPS
- Web server activity log

2 Building AWS

2.1 Requirements

AWS has been mainly developed with GNAT on Windows. It is built and tested regularly on GNU/Linux and Solaris, it should be fairly portable across platforms. To build AWS you need:

- GNU/Ada (GNAT compiler) ;

To build this version you need at least GNAT GPL 2009 Edition or GNAT Pro 6.2 as some Ada 2005 features (`Ada.Containers`, interfaces, overriding keyword) are used. The code should be fairly portable but has never been tested on another compiler than GNAT.

- OpenSSL (**optional**) ;

OpenSSL is an Open Source toolkit implementing the *Secure Sockets Layer* (SSL v2 and v3) and much more. You'll find libraries for Win32 into this distribution. For other platforms just download the OpenSSL source distribution from <http://www.openssl.org> and build it.

- OpenLDAP (**optional**) ;

OpenLDAP is an Open Source toolkit implementing the *Lightweight Directory Access Protocol*. If you want to use the AWS/LDAP API on UNIX based systems, you need to install properly the OpenLDAP package. On Windows you don't need to install it as the 'libldap.a' library will be built by AWS and will use the standard Windows LDAP DLL 'wldap32.dll'.

You can download OpenLDAP from <http://www.openldap.org>.

2.2 AWS.Net.Std

This package is the standard (non-SSL) socket implementation. It exists different implementations of this package:

IPv4

Version based on `GNAT.Sockets`. This is the **default implementation** used.

IPv6

As above but supporting IPv6 protocol. To select this implementation just do when building (see below):

```
$ make IPv6=true
```

2.3 Building

Before building be sure to edit 'makefile.conf', this file contains many settings important for the build. Note that it is important to run `make setup` each time you edit this file.

When you have built and configured all external libraries you must set the `ADA_PROJECT_PATH` variable to point to the GNAT Project files for the different packages. For XML/Ada support, you also need to set `XMLADA` to `true` in 'makefile.conf'.

At this point you can build AWS with:

```
$ make setup build
```

Note that by default AWS demos will be built without SSL support except on Windows. If you want to build the demos with SSL on UNIX (in this case you must have 'libssl.a' and

'`libcrypto.a`' available on your platform), open '`makefile.conf`' and set the *SOCKET* variable to `openssl`. Then rebuild with:

```
$ make setup build
```

It is possible to build *AWS* in debug mode by setting *DEBUG* make's variable in '`makefile.conf`', or just:

```
$ make DEBUG=true setup build
```

Note that by default *AWS* is configured to use the *GNAT* compiler. So, if you use *GNAT* you can build *AWS* just with:

```
$ make setup build
```

If you want to build only the *AWS* libraries and tools and do not want to build the demos you can set *DEMOS* to "false" as in:

```
$ make DEMOS=false setup
```

2.4 Building on cross-platforms

To build for a cross platform the *TARGET* makefile variable must be set with the cross toolchain to be used. The value must be the triplet of the toolchain to use.

For example, to build on VxWorks:

```
$ make TARGET=powerpc-wrs-vxworks setup build
```

2.5 Demos

During the build, the *AWS* library and demos will be compiled. The demos are a good way to learn how to use *AWS*. Yet to learn all features in *AWS* you'll need to read the documentation.

Here are a short description of the demos:

'agent'

A program using the *AWS* client interface. This simple tool can be used to retrieve Web page content. It supports passing through a proxy with authentication and basic authentication on the Web site.

'auth'

A simple program to test the Web Basic and Digest authentication feature.

'com_1'

'com_2'

Two simple programs that use the *AWS* communication service.

'dispatch'

A simple demo using the dispatcher facility. see [Section 4.2.3 \[URI dispatcher\]](#), [page 33](#).

'hello_world'

The famous Hello World program. This is a server that will always return a Web page saying "Hello World!".

<code>'main'</code>	
<code>'hotplug'</code>	A simple test for the hotplug feature.
<code>'res_demo'</code>	A demo using the resource feature. This Web Server embedded a PNG image and an HTML page. The executable is self contained.
<code>'runme'</code>	An example that test many AWS features.
<code>'multiple_sessions'</code>	A demo of two embedded servers using different sessions.
<code>'split'</code>	A demo for the transient pages and page splitter AWS's feature. Here a very big table is split on multiple pages. A set of links can be used to navigate to the next or previous page or to access directly to a given page.
<code>'jabber_demo'</code>	A simple Jabber command line client to check the presence of a JID (Jabber ID). This uses the Jabber API, see Section B.19 [AWS.Jabber] , page 101.
<code>'test_ldap'</code>	A simple LDAP demo which access a public LDAP server and display some information.
<code>'text_input'</code>	A simple demo which handle textarea and display the content.
<code>'vh_demo'</code>	Two servers on the same machine... virtual hosting demo. see Section 4.2.4 [Virtual host dispatcher] , page 33.
<code>'web_elements'</code>	A driver to browse the Web Elements library and see some examples.
<code>'web_mail'</code>	A simple Web Mail implementation that works on a POP mailbox.
<code>'ws'</code>	A static Web page server and push enabled server.
<code>'wps'</code>	A very simple static Web page server based on <code>AWS.Services.Page_Server</code> . see Section 4.3 [Static Page server] , page 35.
<code>'zdemo'</code>	A simple demo of the Gzip content encoding feature.

If XML/Ada is installed it is possible to build the SOAP binding and the SOAP demos, for this just type:

```
$ make build_soap
```

There is four demos:

`'soap_client'`

`'soap_server'`

A simple client/server program to test the SOAP protocol.

`'soap_svcs'`

A server that implements seven SOAP procedures for testing purpose.

`'soap_cvs'`

The client SOAP program that test all seven SOAP procedure of the above server.

In each case you'll certainly have to edit the makefile to correctly set the include path for the libraries `OpenSSL`, `Socket` and `XML/Ada`. For more information, look at the makefiles.

2.6 Installing

When the build is done you must install AWS at a specific location. The target directory is defined with the *prefix* `'makefile.conf'` variable. The default value is set to the compiler root directory. Note that the previously installed version is automatically removed before installing the new one. To install:

```
$ make install
```

To install AWS into another directory you can either edit `'makefile.conf'` and set *prefix* to the directory you like to install AWS or just force the make *prefix* variable:

```
$ make prefix=/opt install
```

Alternatively, with GNAT 5.03 and above it is possible to install AWS into the GNAT Standard Library location. In this case AWS is ready-to-use as there is no need to set `ADA_PROJECT_PATH`, just set *prefix* to point to GNAT root directory:

```
$ make prefix=/opt/gnatpro/6.1.1 install
```

Now you are ready to use AWS !

3 Using AWS

3.1 Setting up environment

3.1.1 Using environment variables

After installing AWS you must set the build environment to point the compiler to the right libraries. First let's say that AWS has been installed in '**awsroot**' directory.

Following are the instructions to set the environment yourself. Note that the preferred solution is to use project files. In this case there is no manual configuration.

spec files

The spec files are installed in '**<awsroot>/include/aws**'. Add this path into **ADA_INCLUDE_PATH** or put it on the command line **-AI<awsroot>/include/aws**.

components

AWS uses some components they are installed in '**<awsroot>/include/aws/components**'. Add this path into **ADA_INCLUDE_PATH** or put it on the command line **-I<awsroot>/include/aws/components**.

libraries

The GNAT library files ('**.ali**') and the AWS libraries ('**libaws.a**') are installed into '**<awsroot>/lib/aws/static**'. Add this path into **ADA_OBJECTS_PATH** or put it on the command line **-aO<awsroot>/lib/aws/static**. Furthermore for **gnatlink** to find the libraries you must add the following library path option on the **gnatmake** command line **-largS -L<awsroot>/lib/aws/static -laws**.

Note that to build SSL applications you need to add **-lssl -lcrypto** on **gnatmake**'s **-largS** section.

external libraries

You must do the same thing (setting **ADA_INCLUDE_PATH** and **ADA_OBJECTS_PATH**) for all external libraries that you will be using.

3.1.2 Using GNAT Project Files

The best solution is to use the installed GNAT Project File '**aws.gpr**'. This is supported only for GNAT 5.01 or above. You must have installed XML/Ada with project file support too.

If this is the case just set the **ADA_PROJECT_PATH** variable to point to the AWS and XML/Ada install directories. From there you just have to with the AWS project file in your GNAT Project file, nothing else to set.

```
with "aws";

project Simple is

  for Main use ("prog.adb");

  for Source_Dirs use ("src");

  for Object_Dir use "obj";

end Simple;
```

See the *GNAT User's Guide* for more information about GNAT Project Files.

3.2 Basic notions

AWS is not a Web Server like *IIS* or *Apache*, it is a component to embedded HTTP protocol in an application. It means that it is possible to build an application which can also answer to a standard browser like *Internet Explorer* or *Netscape Navigator*. Since AWS provides support client and server HTTP protocol, applications can communicate through the HTTP channel. This give a way to build distributed applications, See [Section B.3 \[AWS.Client\]](#), page 85.

An application using AWS can open many HTTP channels. Each channel will use a specific port. For example, it is possible to embedded many HTTP and/or many HTTPS channels in the same application.

3.2.1 Building an AWS server

To build a server you must:

1. declare the HTTP Web Server

```
WS : AWS.Server.HTTP;
```

2. Start the server

You need to start the server before using it. This is done by calling `AWS.Server.Start` (See [Section B.42 \[AWS.Server\]](#), page 124.)

```
procedure Start
(Web_Server      : in out HTTP;
 Name           : in   String;
 Callback       : in   Response.Callback;
 Max_Connection  : in   Positive      := Def_Max_Connect;
 Admin_URI      : in   String        := Def_Admin_URI;
 Port           : in   Positive      := Def_Port;
 Security       : in   Boolean       := False;
 Session        : in   Boolean       := False;
 Case_Sensitive_Parameters : in Boolean := True;
 Upload_Directory : in String       := Def_Upload_Dir);
-- Start the Web server. It initialize the Max_Connection connections
-- lines. Name is just a string used to identify the server. This is used
-- for example in the administrative page. Admin_URI must be set to enable
-- the administrative status page. Callback is the procedure to call for
-- each resource requested. Port is the Web server port. If Security is
-- set to True the server will use an HTTPS/SSL connection. If Session is
-- set to True the server will be able to get a status for each client
-- connected. A session ID is used for that, on the client side it is a
-- cookie. Case_Sensitive_Parameters if set to False it means that the CGI
-- parameters name will be handled without case sensitivity. Upload
-- directory point to a directory where uploaded files will be stored.
```

`Start` takes many parameters:

Web_Server

this is the Web server to start.

Name

This is a string to identify the server. This name will be used for example in the administrative status page.

Callback

This is the procedure to call for each requested resources. In this procedure you must handle all the possible URI that you want to support. (see below).

Max_Connection

This is the maximum number of simultaneous connections. It means that Max_Connection client's browsers can get answer at the same time. This parameter must be changed to match your needs. A medium Web server will certainly need something like 20 or 30 simultaneous connections.

Admin_URI

This is a special URI recognized internally by the server. If this URI is requested the server will return the administrative page. This page is built using a specific template page (default is 'aws_status.thtml') see [Chapter 11 \[Status page\]](#), [page 75](#).

The administrative page returns many information about the server. It is possible to configure the server via two configuration files See [Section 3.4 \[Configuration options\]](#), [page 15](#).

Port

This is the port to use for the Web server. You can use any free port on your computer. Note that on some OS specific range could be reserved or needs special privileges (port 80 on Linux for example).

Security

If Security is set to True the server will use an HTTPS/SSL connection. This part uses the OpenSSL library.

Session

If Session is set to true the server will keep a session ID for each client. The client will be able to save and get variables associated with this session ID.

Case_Sensitive_Parameters

If set to True the CGI name parameters will be handled without using the case.

Note that there is other **Start** routines which support other features. For example there is a **Start** routine which use a dispatcher routine instead of the simple callback procedure. see [Section B.42 \[AWS.Server\]](#), [page 124](#). And there is also the version using a **Config.Object** which is the most generic one.

3. provides a callback procedure

The callback procedure has the following prototype:

```
function Service (Request : in AWS.Status.Data) return AWS.Response.Data;
```

This procedure receive the request status. It is possible to retrieve information about the request through the **AWS.Status** API (See [Section B.69 \[AWS.Status\]](#), [page 151](#).).

For example, to know what URI has been asked:

```
URI : constant String := AWS.Status.URI (Request);

if URI = "/whatever" then
    ...
end if;
```

Then this function should return an answer using one of the constructors in **AWS.Response** (See [Section B.41 \[AWS.Response\]](#), [page 123](#).). For example, to return an HTML message:

```
AWS.Response.Build (Content_Type => "text/html",
                    Message_Body => "<p>just a demo");
```

It is also possible to return a file. For example, here is the way to return a PNG image:

```
AWS.Response.File (Content_Type => "image/png",
                  Filename      => "adains.png");
```

Note that the main procedure should exit only when the server is terminated. For this you can use the `AWS.Server.Wait` service.

A better solution is to use a template engine like `Templates.Parser` to build the HTML Web Server answer. `Templates.Parser` module is distributed with this version of AWS.

3.2.2 Callback procedure

The callback procedure is the user's code that will be called by the AWS component to get the right answer for the requested resource. In fact AWS just open the HTTP message, parsing the HTTP header and it builds an object of type `AWS.Status.Data`. At this point it calls the user's callback procedure, passing the object. The callback procedure must returns the right response for the requested resources. Now AWS will just build up the HTTP response message and send it back to user's browser.

But what is the resource ?

Indeed in a standard Web development a resource is either a static object - an HTML page, an XML or XSL document - or a CGI script. With AWS a resource is *just a string* to identify the resource, it does not represent the name of a static object or CGI script.

So this string is just an internal representation for the resource. The callback procedure must be implemented to handle each internal resource and return the right response.

Let's have a small example. For example we want to build a Web server that will answer "Hello World" if we ask for the internal resource `/hello`, and must answer "Hum..." otherwise.

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

procedure Hello_World is

    WS : AWS.Server.HTTP;

    function HW_CB (Request : in AWS.Status.Data)
        return AWS.Response.Data
    is
        URI : constant String := AWS.Status.URI (Request);
    begin
        if URI = "/hello" then
            return AWS.Response.Build ("text/html", "<p>Hello world !");
        else
            return AWS.Response.Build ("text/html", "<p>Hum...");
        end if;
    end HW_CB;

begin
    AWS.Server.Start
        (WS, "Hello World", Callback => HW_CB'Unrestricted_Access);
    delay 30.0;
end Hello_World;
```

Now of course the resource internal name can represent a file on disk. It is not mandatory but it is possible. For example it is perfectly possible to build with AWS a simple page server.

As an example, let's build a simple page server. This server will return files in the current directory. Resources internal name represent an HTML page or a GIF or PNG image for example. This server will return a 404 message (Web Page Not Found) if the file does not exist. Here is the callback procedure that implements such simple page server:

```
function Get (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 Utils.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 Get;
```

3.2.3 Form parameters

Form parameters are stored into a table of key/value pair. The key is the form input tag name and the value is the content of the input field as filled by the user.

```
Enter your name

<FORM METHOD=GET ACTION=/get-form>"
<INPUT TYPE=TEXT NAME=name VALUE="<default>" size=15>
<INPUT TYPE=SUBMIT NAME=go VALUE="Ok">
</FORM>
```

Note that as explained above see [Section 3.2.2 \[Callback procedure\], page 10](#), the resource described in ACTION is just an internal string representation for the resource.

In this example there is two form parameters:

name The value is the content of this text field as filled by the client.

go The value is "Ok".

There is many functions (in `AWS.Parameters`) to retrieve the tag name or value and the number of parameters. Here are some examples:

```
function Service (Request : in AWS.Status.Data)
  return AWS.Response.Data
is
  P : constant AWS.Parameters.List := AWS.Status.Parameters (Request);
  ...
```

`AWS.Parameters.Get (P, "name")`
Returns the value for parameter named **name**

`AWS.Parameters.Get_Name (P, 1)`
Returns the string "name".

`AWS.Parameters.Get (P, 1)`
Returns the value for parameter named **name**

```
AWS.Parameters.Get (P, "go")
    Returns the string "Ok".
```

```
AWS.Parameters.Get_Name (P, 2)
    Returns the string "go".
```

```
AWS.Parameters.Get (P, 2)
    Returns the string "Ok".
```

`Request` is the `AWS` current connection status passed to the callback procedure. And `P` is the parameters list retrieved from the connection status data. For a discussion about the callback procedure See [Section 3.2.1 \[Building an AWS server\]](#), page 8.

3.2.4 Distribution of an AWS server

The directory containing the server program must contain the following files if you plan to use a status page See [Chapter 11 \[Status page\]](#), page 75.

```
'aws_status.shtml'
    The template HTML file for the AWS status page.
```

```
'aws_logo.png'
    The AWS logo displayed on the status page.
```

```
'aws_up.png'
    The AWS hotplug table up arrow.
```

```
'aws_down.png'
    The AWS hotplug table down arrow.
```

Note that these filenames are the current `AWS` default. But it is possible to change those defaults using the configuration files see [Section 3.4 \[Configuration options\]](#), page 15.

3.3 Building answers

We have already seen, in simple examples, how to build basic answers using `AWS.Response` API. In this section we present all ways to build answers from basic support to the more advanced support like the compressed memory stream response.

3.3.1 Redirection

A redirection is a way to redirect the client's browser to another URL. Client's won't notice that a redirection has occurs. As soon as the browser has received the response from the server it will retrieve the page as pointed by the redirection.

```
return Response.URL (Location => "/use-this-one");
```

3.3.2 New location for a page

User will receive a Web page saying that this page has moved and eventually pointing to the new location.

```
return Response.Moved
    (Location => "/use-this-one";
     Message => "This page has moved, please update your reference");
```

3.3.3 Authentication required

For protected pages you need to ask user to enter a password. see [Section 3.7 \[Authentication\]](#), page 21.

3.3.4 Sending back an error message

`Acknowledge` can be used to send back error messages. There is many kind of status code, see `Message.Status_Code` definition. Together with the status code it is possible to pass textual error message in `Message_Body` parameter.

```
return Response.Acknowledge
  (Status_Code => Messages.S503,
   Message_Body => "Can't connect to the database, please retry later.",
   Content_Type => MIME.Text_Plain);
```

3.3.5 Response from a string

This is the simplest way to build a response object. There is two constructors in `AWS.Response`, one based on a standard string and one for `Unbounded_String`.

```
return Response.Build (MIME.Text_HTML, "My answer");
```

The `Build` routine takes also a status code parameter to handle errors. By default this code is `Messages.S200` which is the standard HTTP status (no error encountered). The other parameter can be used to control caches. see [Section B.41 \[AWS.Response\]](#), page 123.

3.3.6 Response from a Stream_Element_Array

This is exactly as above but the `Build` routine takes a `Stream_Element_Array` instead of a string.

3.3.7 Response from a file

To build a `File` response there is a single constructor named `File`. This routine is very similar to the one above except that we specify a filename as the response.

```
return Response.File (MIME.Text_HTML, "index.html");
```

Again there parameters to control the status code and cache. No check on the filename is done at this point, so if `'index.html'` does not exist no exception is raised. The server is responsible to check for the file and to properly send back the 404 message if necessary.

Note that this routine takes an optional parameter named `Once` that is to be used for temporary files created on the server side for the client. With `Once` set to `True` the file will be deleted by the server after sending it (this includes the case where the download is suspended).

3.3.8 Response from a stream

Sometimes it is not possible (or convenient) to build the response in memory as a string object for example. Streams can be used to workaround this. The constructor for such response is again very similar to the ones above except that instead of the data we pass an handle to a `Resources.Streams.Stream_Type` object.

The first step is to build the stream object. This is done by deriving a new type from `Resources.Streams.Stream_Type` and implementing three abstract procedures.

Read

Must return the next chunk of data from the stream. Note that initialization if needed are to be done there during the first call to `read`.

End_Of_File

Must return `True` when there is no more data on the stream.

Close

Must close the stream and for example release all memory used by the implementation.

The second step is to build the response object:

```
type SQL_Stream is new Resources.Streams.Stream_Type;

Stream_Object : SQL_Stream;

procedure Read (...) is ...
function End_Of_File (...) return Boolean is ...
procedure Close (...) is
...

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

Note that in some cases it is needed to create a file containing the data for the client (for example a tar.gz or a zip archive). But there is no way to properly remove this file from the file system as we really don't know when the upload is terminated when using the `AWS.Response.File` constructor. To solve this problem it is possible to use a stream as the procedure `Close` is called by the server when all data have been read. In this procedure it is trivial to do the necessary clean-up.

3.3.9 Response from a on-disk stream

An ready-to-use implementation of the stream API described above where the stream content is read from an on-disk file.

3.3.10 Response from a on-disk once stream

An ready-to-use implementation of the stream API described above where the stream content is read from an on-disk file. When the transfer is completed the file is removed from the file system.

3.3.11 Response from a memory stream

This is an implementation of the standard stream support described above. In this case the stream is in memory and built by adding data to it.

To create a memory stream just declare an object of type `AWS.Resources.Streams.Memory.Stream_Type`. When created, this memory stream is empty, using the `Streams.Memory.Append` routines it is possible to add chunk of data to it. It is of course possible to call `Append` as many times as needed. When done just return this object to the server.

```
Data : AWS.Resources.Streams.Memory.Stream_Type;

Append (Data, Translator.To_Stream_Element_Array ("First chunk"));
Append (Data, Translator.To_Stream_Element_Array ("Second chunk..."));

...

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

Note that you do not have to take care of releasing the allocated memory, the default `Close` routine will do just that.

3.3.12 Response from a compressed memory stream

This is a slight variant of the standard memory stream described above. In this case the stream object must be declared as a `AWS.Resources.Streams.Memory.ZLib.Stream_Type`.

The ZLib stream object must be initialized to enable the compression and select the right parameters. This is done using the `AWS.Resources.Streams.Memory.ZLib.Deflate_Initialize` routine which takes many parameters to select the right options for the compression algorithm, all of them have good default values. When initialized the compressed stream object is used exactly as a standard stream.

```
Data : AWS.Resources.Streams.Memory.ZLib.Stream_Type;

Deflate_Initialize (Data);

Append (Data, Translator.To_Stream_Element_Array ("First chunk"));
Append (Data, Translator.To_Stream_Element_Array ("Second chunk..."));

...

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

Note that there is the reverse implementation to decompress a stream. see [Section B.39 \[AWS.Resources.Streams.Memory.ZLib\]](#), page 121. It's usage is identical.

3.3.13 Split page

AWS has a specific high level service to split a large response into a set of pages. For more information see [Section 4.5 \[Split pages\]](#), page 36.

3.3.14 Response a from pipe stream

The response sent to the server is read from the output of an external application. This kind of stream can be used to avoid writing a temporary file into the hard disk. For example it is possible to return an archive created with the `tar` tool without writting the intermediate tar achive on the disk.

3.4 Configuration options

To configure an AWS server it is possible to use a configuration object. This object can be set using the `AWS.Config.Set` API or initialized using a configuration file.

Configuration files are a way to configure the server without recompiling it. Each application can be configured using two configurations files:

`'aws.ini'`

This file is parsed first and corresponds to the configuration for all AWS server runs in the same directory.

`'<program_name>.ini'`

This file is parsed after `'aws.ini'`. It is possible with this initialization file to have specific settings for some servers. `'program_name.ini'` is looked first in the application's directory and then in the current working directory.

Furthermore, it is possible to read a specific configuration file using the `AWS.Config.Ini.Read` routine. see [Section B.9 \[AWS.Config.Ini\]](#), page 91.

Current supported options are:

`Accept_Queue_Size (positive)`

This is the size of the queue for the incoming requests. Higher this value will be and less *"connection refused"* will be reported to the client. The default value is 64.

Admin_Password (string)

This is the password used to call the administrative page. The password can be generated with 'aws_password' (the module name must be *admin*)

```
$ aws_password admin <password>
```

Admin_URI (string)

This is the URI to call the administrative page. This can be used when calling `AWS.Server.Start`. The default is ''.

Certificate (string)

Set the certificate file to be used with the secure servers. The default is 'cert.pem'. A single certificate or a certificate chain is supported. The certificates must be in PEM format and the chain must be sorted starting with the subject's certificate, followed by intermediate CA certificates if applicable and ending at the highest level (root) CA certificate. If the file contains only a single certificate, it can be followed by a private key. In this case the Key parameter (see below) must empty.

Case_Sensitive_Parameters (boolean)

If set to `True` the HTTP parameters are case sensitive. The default value is `TRUE`.

Check_URL_Validity (boolean)

Server have to check URI for validity. For example it checks that an URL does not reference a resource above the Web root. The default is `TRUE`.

Cleaner_Wait_For_Client_Timeout (duration)

Number of seconds to timeout on waiting for a client request. This is a timeout for regular cleaning task. The default is 80.0 seconds.

Cleaner_Client_Header_Timeout (duration)

Number of seconds to timeout on waiting for client header. This is a timeout for regular cleaning task. The default is 7.0 seconds.

Cleaner_Client_Data_Timeout (duration)

Number of seconds to timeout on waiting for client message body. This is a timeout for regular cleaning task. The default is 28800.0 seconds.

Cleaner_Server_Response_Timeout (duration)

Number of seconds to timeout on waiting for client to accept answer. This is a timeout for regular cleaning task. The default is 28800.0 seconds.

Directory_Browser_Page (string)

Specify the filename for the directory browser template page. The default value is 'aws_directory.shtml'.

Down_Image (string).

The name of the down arrow image to use in the status page. The default is 'aws_down.png'.

Error_Log_Filename_Prefix (string)

This is to set the filename prefix for the log file. By default the log filename prefix is the program name (without extension) followed by "_error".

Error_Log_Split_Mode [None/Each_Run/Daily/Monthly]

It indicates how to split the error logs. `Each_Run` means that a new log file is used each time the process is started. `Daily` and `Monthly` will use a new log file each day or month. The default is `NONE`.

Exchange_Certificate (boolean)

If set to True it means that the client will be asked to send its certificate to the server. The default value is FALSE.

Force_Wait_For_Client_Timeout (duration)

Number of seconds to timeout on waiting for a client request. This is a timeout for urgent request when resources are missing. The default is 2.0 seconds.

Force_Client_Header_Timeout (duration)

Number of seconds to timeout on waiting for client header. This is a timeout for urgent request when resources are missing. The default is 2.0 seconds.

Force_Client_Data_Timeout (duration)

Number of seconds to timeout on waiting for client message body. This is a timeout for urgent request when resources are missing. The default is 10800.0 seconds.

Force_Server_Response_Timeout (duration)

Number of seconds to timeout on waiting for client to accept answer. This is a timeout for urgent request when resources are missing. The default is 10800.0 seconds.

Free_Slots_Keep_Alive_Limit (positive)

This is the minimum number of remaining free slots to enable keep-alive HTTP connections. For heavy-loaded HTTP servers, the `Max_Connection` parameter should be big enough, and `Free_Slots_Keep_Alive_Limit` should be about 1-10% of the `Max_Connection` parameter depending on the duration of the average server response. Longer is the average time to send back a response bigger `Free_Slots_Keep_Alive_Limit` should be. The default is 1.

Hotplug_Port (positive)

This is the hotplug communication port needed to register and un-register an hotplug module. The default value is 8888.

Key (string)

Set the RSA key file to be used with the secure servers. The default file is ‘.’.

Line_Stack_Size (positive)

The HTTP lines stack size. The stack size must be adjusted for each applications depending on the use of the stack done by the callback procedures. The default is 1376256.

Log_Extended_Fields (string list)

Comma separated list of the extended log field names. If this parameter is empty, the HTTP log would be in the apache compatible format, otherwise log file would be in Extended format. For more information see [Section 3.13 \[Server Log\]](#), page 26.

Log_File_Directory (string)

This is to set the directory where log file must be written. This parameter will be used automatically by `AWS.Log` if logging facility is enabled. By default log files are written in the current directory. The default is ‘./’.

Log_Filename_Prefix (string)

This is to set the filename prefix for the log file. By default the log filename prefix is the program name (without extension).

Log_Split_Mode [None/Each_Run/Daily/Monthly]

It indicates how to split the logs. `Each_Run` means that a new log file is used each time the process is started. `Daily` and `Monthly` will use a new log file each day or month. The default is `NONE`.

Logo_Image (string).

The name of the logo image to use in the status page. The default is 'aws_logo.png'.

Max_Connection (positive)

This is the maximum number of simultaneous connections for the server. This can be used when calling the `AWS.Server.Start`. The default is 5.

Note that the total number of threads used by a server is:

`N = <main server thread> + <max connections> [+ <session>]`

Note: [...] means optional value

Add 1 only if the session feature is activated. This is due to the session cleaner task.

Receive_Timeout (duration)

Number of seconds to timeout when receiving chunk of data. The default is 30.0 seconds.

Reuse_Address (boolean)

Set the socket reuse address policy. If set to True the server will be able to bind to a socket that has just been released without the need of waiting. Enabling this feature is not secure and should probably used only for debugging purpose. The default is FALSE.

Security_Mode (string)

Set the security mode to use for the secure connections. The default mode is "SSLv23". see [Section B.28 \[AWS.Net.SSL\]](#), page 110.

Send_Timeout (duration)

Number of seconds to timeout when sending chunk of data. The default is 40.0 seconds.

Server_Host (string)

The name of the host machine. This can be used if a computer has more than one IP address, it is possible to have two servers at the same port on the same machine, both being binded on different IP addresses.

Server_Name (string)

The name of the server. This can be used when calling `AWS.Server.Start`. The default is "AWS Module".

Server_Port (integer)

The port where server will wait for incoming connections requests. This can be used when calling `AWS.Server.Start`. The default is 8080.

Session (boolean)

Whether the session support must be activated or not. The default is FALSE.

Session_Name (string)

The name of the cookie session. This can be used to support sessions for multiple servers embedded into the same executable. The default is "AWS".

Session_Lifetime (duration)

Number of seconds to keep session information. After this period a session is obsoleted and will be removed at next cleanup. The default is 600.0 seconds.

Session_Cleanup_Interval (duration)

Number of seconds between each run of the session cleanup task. This task will remove all session data that have been obsoleted. The default is 300.0 seconds.

Status_Page (string)

The name of the status page to used. The default is `'aws_status.html'`.

Transient_Cleanup_Interval (positive)

Specify the number of seconds between each run of the cleaner task to remove transient pages. The default value is 180.0 seconds.

Transient_Lifetime

Specify the number of seconds to keep a transient page. After this period the transient page is obsoleted and will be removed during next cleanup. The default value is 300.0 seconds.

Up_Image (string)

The name of the up arrow image to use in the status page. The default is `'aws_up.png'`.

Upload_Directory (string)

This is to set the directory where upload files must be stored. By default uploaded files are written in the current directory. The default is `'.'`.

WWW_Root (string)

This option sets the Web Server root directory. All Web resources are referenced from this root directory. The default value is `"/"`.

Each option value can be retrieved using the `AWS.Config` unit or set using `AWS.Config.Set`.

For example to build a server where the *port* and the maximum number of *connection* can be changed via a configuration file (either `'aws.ini'` or `'<program_name>.ini'`):

```
WS    : AWS.Server.HTTP;

Conf : constant AWS.Config.Object := AWS.Config.Get_Current;

Server.Start (WS, Service'Access, Conf);
```

3.5 Session handling

AWS provides a way to keep session data while users are browsing. It works by creating transparently a session ID where it is possible to insert, delete and retrieve session data using a standard Ada API (See [Section B.66 \[AWS.Session\]](#), page 148.). Session data are key/value pair each of them being strings. These sessions data are kept on the server, for client side state management see [Section 3.6 \[HTTP state management\]](#), page 20.

- First you declare and start an HTTP channel with session enabled:

```
WS : AWS.Server.HTTP;

Server.Start (WS,
              Port      => 1234,
              Callback => Service'Access,
              Session   => True);
```

Here we have built an HTTP channel with a maximum of 3 simultaneous connections using the port 1234. A session ID will be created and sent inside a cookie to the client's browser at the first request. This session ID will be sent back to the server each time the client will ask for a resource to the server.

- Next, in the Service callback procedure that you have provided you must retrieve the Session ID. As we have seen, the callback procedure has the following prototype:

```
function Service (Request : in AWS.Status.Data) return AWS.Response.Data;
```

The Session ID is kept in the Request object and can be retrieved using:

```
Session_ID : constant AWS.Session.ID := AWS.Status.Session (Request);
```

- From there it is quite easy to get or set some session data using the provided API. For example:

```
declare
  C : Integer;
begin
  C := AWS.Session.Get (Session_ID, "counter");
  C := C + 1;
  AWS.Session.Set (Session_ID, "counter", C);
end;
```

This example first get the value (as an Integer) for session data whose key is "counter", increment this counter and then set it back to the new value.

It is also possible to save and restore all session data. It means that the server can be shutdown and launched some time after and all client data are restored as they were at shutdown time. Client will just see nothing. With this feature it is possible to shutdown a server to update its look or because a bug has been fixed for example. It is then possible to restart it keeping the complete Web server context.

3.6 HTTP state management

AWS provides a full implementation of RFC 2109 via the `AWS.Cookie` package. Using this package you set, get and expire client-side HTTP cookies.

First we set a cookie:

```
declare
  Content : AWS.Response.Data;
begin
  AWS.Cookie.Set (Content,
                  Key      => "hello",
                  Value    => "world",
                  Max_Age  => 86400);
end;
```

Here we set the cookie `hello` with the value `world`, and we tell the client to expire the cookie 86400 seconds into the future.

Getting the cookie value back is equally simple:

```
declare
  Request : AWS.Status.Data
  -- Assume that this object contain an actual HTTP request.
begin
  Put_Line (AWS.Cookie.Get (Request, "hello"));
  -- Output 'world'
end;
```

Had the cookie `hello` not existed, an empty `String` would've been returned.

In some cases it might be of value to know if a given cookie exists, and for that we have the `Exists` function available.

```

declare
  Request : AWS.Status.Data
  -- Assume that this object contain an actual HTTP request
begin
  if AWS.Cookie.Exists ("hello") then
    Put_Line ("The 'hello' cookie exists!");
  end if;
end;

```

Note that `Exists` doesn't care if the cookie contains an actual value or not. If a cookie with no value exists, `Exists` will return `True`.

And finally we might wish to tell the client to expire a cookie:

```

declare
  Content : AWS.Response.Data;
begin
  AWS.Cookie.Expire (Content,
                    Key => "hello");
end;

```

The Cookie package provide `Get` functions and `Set` procedures for `String`, `Integer`, `Float` and `Boolean` types, but since cookies are inherently strings, it's important to understand what happens when the cookie `String` value can't be converted properly to either `Integer`, `Float` or `Boolean`.

So if either conversion fails or the cookie simply doesn't exist, the following happens:

- For `Integer`, the value 0 is returned
- For `Float`, the value 0.0 is returned.
- For `Boolean`, the value `False` is returned. Note that only the string "True" is `True`. Everything else is `False`.

For more information, See [Section B.12 \[AWS.Cookie\]](#), page 94.

3.7 Authentication

AWS supports **Basic** and **Digest** authentication. The authentication request can be sent at any time from the callback procedure. For this the `AWS.Response.Authenticate` message must be returned.

The authentication process is as follow:

1. Send authentication request

From the callback routine return an authentication request when needed.

```

function Service (Request : in Status.Data) return Response.Data is
  URI  : constant String := Status.URI (Request);
  User : constant String := Status.Authorization_Name (Request);
begin
  -- URI starting with "/prot/" are protected
  if URI (URI'First .. URI'First + 5) = "/prot/"
    and then User = ""
  then
    return Response.Authenticate ("AWS", Response.Basic);
  end if;
end;

```

The first parameter is the **Realm**, it is just a string that will be displayed (on the authentication dialog box) by the browser to indicate for which resource the authentication is needed.

2. Check authentication

When an authentication has been done the callback's request data contain the user and password. Checks the values against an ACL for each protected resources.

```
function Protected_Service
  (Request : in AWS.Status.Data)
  return AWS.Response.Data
is
  User : constant String := Status.Authorization_Name (Request);
  Pwd  : constant String := Status.Authorization_Password (Request);
begin
  if User = "xyz" and then Pwd = "azerty" then
    return ...;
```

Note that the **Basic** authentication is not secure at all. The password is sent unencoded by the browser to the server. If security is an issue it is better to use the **Digest** authentication and/or an **SSL** server.

3.8 File upload

File upload is the way to send a file from the client to the server. To enable file upload on the client side the Web page must contain a **FORM** with an **INPUT** tag of type **FILE**. The **FORM** must also contain the **enctype** attribute set to *multipart/form-data*.

```
<FORM enctype="multipart/form-data" ACTION=/whatever METHOD=POST>
File to process: <INPUT NAME=filename TYPE=FILE>
<INPUT TYPE=SUBMIT NAME=go VALUE="Send File">
</FORM>
```

On the server side, AWS will retrieve the file and put it into the upload directory. AWS add a prefix to the file to ensure that the filename will be unique on the server side. The upload directory can be changed using the configuration options. See [Section 3.4 \[Configuration options\]](#), page 15.

The uploaded files are removed after the user's callback. This is done for security reasons, if files were not removed it would be possible to fill the server hard disk by uploading large files to the server. This means that uploaded files must be specifically handled by the users by either copying or renaming them.

AWS will also setup the form parameters as usual. In the above example there is two parameters (See [Section 3.2.3 \[Form parameters\]](#), page 11.)

filename This variable contains two values, one with the client side name and one with the server side name.

First value : Parameters.Get (P, "filename")

The value is the full pathname of the file on the server. (i.e. the upload directory catenated with the prefix and filename).

Second value : Parameters.Get (P, "filename", 2)

The value is the simple filename (no path information) of the file on the client side.

go The value is "Send File"

3.9 Communication

This API is used to do communication between programs using the HTTP GET protocol. It is a very simple API not to be compared with GLADE or SOAP. This communication facility is to

be used for simple request or when a light communication support is needed. For more complex communications or to achieve inter-operability with other modules it is certainly a good idea to have a look at the AWS/SOAP support, see [Section B.73 \[SOAP\], page 155](#).

In a communication there is a Client and a Server. Here is what is to be done on both sides to have programs talking together.

3.9.1 Communication - client side

On the client side it is quite simple. You just have to send a message using `AWS.Communication.Client.Send_Message`.

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

The message is sent to the specified server using the given port. A message is composed of a name which is a string and a set of parameters. There is a parameter set constructor in `AWS.Communication`. This function return a response as for any callback procedure.

3.9.2 Communication - server side

On the server side things are a bit more complex but not that difficult. You must instantiate the `AWS.Communication.Server` generic package by providing a callback procedure. This callback procedure will must handle all kind of message that a client will send.

During instantiation you must also pass a context for the communication server. This context will be passed back to the callback procedure.

```
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 := Null_Parameter_Set)
    return Response.Data;

package AWS.Communication.Server is
  ...
```

A complete example can be found in the demos directory. Look for 'com_1.adb' and 'com_2.adb'.

Note that this communication API is used by the Hotplug module facility See [Section 3.10 \[Hotplug module\], page 23](#).

3.10 Hotplug module

An **Hotplug module** is a module that can be dynamically binded to a running server. It is a Web server and the development process is very similar to what we have seen until now See

[Section 3.2.1 \[Building an AWS server\], page 8](#). The Hotplug module will register itself into a Web server by sending a message using the communication API. The Hotplug module send to the server a regular expression and an URL. The main server will redirect all URL matching the regular expression to the Hotplug module.

Note that the main server will redirect the URL to the first matching regular expression.

3.10.1 Hotplug module - server activation

The first step is to properly create the main server hotplug module registration file. This file must list all hotplug modules that can register into the main server. Each line have the following format:

```
hotplug_module_name:password:server:port
```

hotplug_module_name

The name of the hotplug module. You can choose any name you want. This name will be use during the registration process and to generate the password.

password

The MD5 password, see below.

server

The name of the server where the redirection will be made. This is for security reasons, main server will not permit to redirect requests to any other server.

port

The port to use for the redirection on **server**.

You must create a password for each hotplug modules. The generated password depends on the hotplug module name. A tool named `aws_password` is provided with AWS to generate such password. Usage is simple:

```
$ aws_password <hotplug_module_name> <password>
```

Then, after starting the main server you must activate the Hotplug feature:

```
AWS.Server.Hotplug.Activate (WS'Unchecked_Access, 2222, "hotplug_conf.ini");
```

'hotplug_conf.ini' is the hotplug module registration file described above.

3.10.2 Hotplug module - creation

Here is how to create an Hotplug module:

1. First you create a standard Web server See [Section 3.2.1 \[Building an AWS server\], page 8](#).

```
WS : AWS.Server.HTTP
    (3, 1235, False, Hotplug_CB.Hotplug'Access, False);
```

Here we have a server listening to the port 1235. This server can be used alone if needed as any Server developed with AWS.

2. Then you register the Hotplug module to the main server See [Section B.4 \[AWS.Client.Hotplug\], page 86](#).

```
Response := AWS.Client.Hotplug.Register
(Name      => "Hotplug_Module_Demo",
 Password => "my_password",
 Server    => "http://dieppe:2222",
 Regexp    => ".*AWS.*",
 URL       => "http://omsk:1235/");
```

The hotplug module `Hotplug_Module_Demo` must have been declared on the main server, the password and redirection must have been properly recorded too for security reasons see [Section 3.10.1 \[Hotplug module - server activation\]](#), page 24. This command register `Hotplug_Module_Demo` into the server running on the machine `dieppe` and ask it to redirect all URL containing `AWS` to the server running on machine `omsk` on port 1235.

3. When the Hotplug module is stopped, you must unregister it

```
Response := AWS.Client.Hotplug.Unregister
(Name      => "Hotplug_Module_Demo",
 Password => "my_password",
 Server    => "http://dieppe:2222",
 Regexp    => ".*AWS.*");
```

Here we ask to unregister `Hotplug_Module_Demo` from server `dieppe`. As for the registration process a proper password must be specified see [Section 3.10.1 \[Hotplug module - server activation\]](#), page 24.

A complete example can be found in the demos directory. Look for 'main.adb' and 'hotplug.adb'.

3.11 Server Push

Server Push is a feature that let the Web Server send continuously data to client's Web Browser or client applications. The client does not have to reload at periodic time (which is what is called client pull) to have the data updated, each time the server send a piece of data it gets displayed on the client.

To build a push server you need to build an instance of the `AWS.Server.Push` package. This package takes a set of formal parameters. Here are the step-by-step instructions to build a Push Server:

1. The data to be sent

First you must create a type that will contains the data to be sent to client's browser except if it is a standard Ada type. See `Client_Output_Type` formal parameter.

2. The data that will be streamed

This is the representation of the data that will be sent to client's browser. This will be either a `String` for Web pages or `Stream_Element_Array` for binary data like pictures. See `Stream_Output_Type` formal parameter.

3. The context

It is often nice to be able to configure each client with different parameters if needed. This can be achieved with the Context data type that will be passed as parameter of the conversion function described below. See `Client_Environment` formal parameter.

4. Provides a function to convert from the data type to be sent to the data that will be streamed.

This is a function that will transform the data described on point 1 above to the form described on point 2 above. See `To_Stream_Output` formal parameter.

5. Build the Push Server

To do so you just need to instantiate `AWS.Server.Push` with the above declarations.

6. Registering new clients

In the standard `AWS` procedure callback it is possible to register a client if requested. This is done by calling `AWS.Server.Push.Register`. It is possible to unregister a client using `AWS.Server.Push.Unregister`. Each client must be identified with a unique client ID. After registering a new client from the callback procedure you must return the `AWS.Response.Socket_Taken` message. This is very important, it tells the server to not close this socket.

7. Sending the data

At this point it is possible to send data to clients. To do so two routines are available.

`AWS.Server.Push.Send_To`

To send a piece of data to a specific client identified by its client ID.

`AWS.Server.Push.Send`

To send a piece of data to all clients registered on this server.

Very large Internet applications should use this feature carefully. A push server keeps a socket reserved for each registered clients and the number of available sockets per process is limited by the OS.

3.12 Working with Server sockets

With `AWS` it is possible to take out a socket from the server and give it back later. This feature must be used carefully but it gives a lot of flexibility. As the socket is taken away, the connection line (or slot) is released, `AWS` can then use it to handle other requests.

This can be used to better support heavy loaded servers when some requests need a long time to complete. Long time here means longer than most of the other requests which should be mostly interactivities for a Web server. Of course in such a case a keep-alive connection is kept open.

The usage in such a case is to take out the socket and put it in a waiting line. This releases the connection for the server. When the data to prepare the answer is ready you give back the socket to the server.

- Take a socket from the server

This first step is done from the callback function. A user instead of replying immediately decides to take away the socket from the server. The first step is to record the connection socket by calling `AWS.Status.Socket`. The second step is to tell the server to not release this socket by returning `AWS.Response.Socket_Taken` from the callback function. At this point the server will continue to serve other clients.

Note that this feature is used by the server push implementation see [Section 3.11 \[Server Push\]](#), page 25.

- Give back the socket to the server

Calling `AWS.Server.Give_Back_Socket` will register the socket for reuse. This socket will be placed into a pool, next time the server will check for incoming requests it will be picked up.

3.13 Server Log

It is possible to have the server activity logged into the file '`<progname>-Y-M-D.log`'. To activate the logging you must call the `AWS.Server.Log.Start`, and it is possible to stop logging by calling `AWS.Server.Log.Stop`. Note that `AWS.Server.Log.Start` have a parameter named `Auto_Flush` to control output buffering. This parameter is `False` by default. If set to `True`, the

log file will be automatically flushed after each data. If the server logging is not buffered, i.e. `Auto_Flush` is `False`, the log can still be flushed by calling the `AWS.Server.Log.Flush` routine. See [Section B.21 \[AWS.Log\], page 103](#), for more information especially about the way rotating logs can be setup. Using this feature it is possible to have automatic split of the log file each day, each month or at every run. See `AWS.Log` spec. This is very useful to avoid having very big log files.

The log format depend on `Log_Extended_Fields` configuration parameter. If this parameter is empty, the HTTP log would have fixed apache compatible format:

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

For example:

```
100.99.12.1 - - [22/Nov/2000:11:44:14] "GET /whatever HTTP/1.1" 200 1789
```

If the extended fields list is not empty, the log file format would have user defined fields set:

```
#Version: 1.0
#Date: 2006-01-09 00:00:01
#Fields: date time c-ip cs-method cs-uri cs-version sc-status sc-bytes
2006-01-09 00:34:23 100.99.12.1 GET /foo/bar.html HTTP/1.1 200 30
```

Fields in the comma separated `Log_Extended_Fields` list could be:

<i>date</i>	Date at which transaction completed
<i>time</i>	Time at which transaction completed
<i>time-taken</i>	Time taken for transaction to complete in seconds
<i>c-ip</i>	Client side connected IP address
<i>c-port</i>	Client side connected port
<i>s-ip</i>	Server side connected IP address
<i>s-port</i>	Server side connected port
<i>cs-method</i>	HTTP request method
<i>cs-username</i>	Client authentication username
<i>cs-version</i>	Client supported HTTP version
<i>cs-uri</i>	Request URI
<i>cs-uri-stem</i>	Stem portion alone of URI (omitting query)
<i>cs-uri-query</i>	Query portion alone of URI
<i>sc-status</i>	Response status code
<i>sc-bytes</i>	Length of response message body
<i>cs(<header>)</i>	Any header field name sent from client to server
<i>sc(<header>)</i>	Any header field name sent from server to client

x-<appfield>

Any application defined field name

AWS also support error log files. If activated every internal error detected by AWS will gets logged into this special file. Log file for errors would be in simple apache compatible format. See `AWS.Server.Log.Start_Error` and `AWS.Server.Log.Stop_Error`.

For the full set of routines supporting the log facility see [Section B.44 \[AWS.Server.Log\]](#), page 126.

3.14 Secure server

It is not much difficult to use a secure server (HTTPS) than a standard one. Here we describe only what is specific to an HTTPS server.

Before going further you must check that AWS has been configured with SSL support. see [Section 2.3 \[Building\]](#), page 3. You must also have installed the `OpenSSL` library on your system. If this is done, you can continue reading this section.

3.14.1 Initialization

A server is configured as using the HTTPS protocol at the time it is started. The only thing to do is to set the `Start's Security` parameter to `True`. This will start a server and activate the SSL layer by default. A secure server must use a valid certificate, the default one is `'cert.pem'`. This certificate has been created by the `openssl` tool and is valid until year 2008. Yet, this certificate has not been signed. To build a secure server user's can rely on, you must have a valid certificate signed by one of the **Certificate Authorities**.

The certificate to be used must be specified before starting the secure server with `AWS.Server.Set_Security`.

```
AWS.Server.Set_Security (WS, Certificate_Filename => "/xyz/aws.pem");
```

3.14.2 Creating a test certificate

The goal here is not to replace the `OpenSSL` documentation but just to present one way to create a certificate for an HTTPS test server.

The RSA key

```
$ openssl genrsa -rand <filename> -out ca-key.pem
```

Filename must be point to any file, this is used to initialized the random seed.

The Certificate

```
$ openssl req -new -x509 -days 730 -key ca-key.pem -out ca-cert.pem
```

Create a single self contained file

```
$ cat ca-key.pem ca-cert.pem > aws.pem
```

At this point you can use `'aws.pem'` with your server.

3.14.3 Protocol

There are different security options, either `SSLv2`, `SSLv3` or `TLSv1`. `SSLv2` and `SSLv3` are supported by most if not all Web browsers. These are the default protocol used by AWS.

TLSv1 is not supported at this point.

3.15 Unexpected exception handler

When AWS detects an internal problem, it calls a specific handler. This handler can be used to log the error, send an alert message or build the answer to be sent back to the client's browser. Here is the spec for this handler:

```
type Unexpected_Exception_Handler is access
  procedure (E      : in      Ada.Exceptions.Exception_Occurrence;
             Log     : in out  AWS.Log.Object;
             Error   : in      Data;
             Answer  : in out  Response.Data);
```

The handler can be called in two modes:

Non fatal error (**Error.Fatal is False**)

In this case AWS will continue working without problem. A bug has been detected but it was not fatal to the thread (slot in AWS's terminology) handling. In this case it is possible to send back an application level message to the client's browser. For that you just have to fill the unexpected handler's *Answer* parameter with the right response message. The *Error* parameter receive information about the problem, see [Section B.16 \[AWS.Exceptions\], page 98](#).

Fatal error (**Error.Fatal is True**)

In this case AWS will continue working but a thread (slot number *Error.Slot* in AWS's terminology) will be killed. It means that AWS will have lost one the simultaneous connection handler. The server will continue working unless it was the last slot handler available. Note that a Fatal error means an AWS internal bug and it should be reported if possible. In this mode there is no way to send back an answer to the client's browser and *Error* value must be ignored.

The default handler for unexpected exceptions send a message to standard error for fatal errors. For non fatal errors it log a message (if the error log is activated for the server) and send back a message back to the client. The message is either a built-in one or, if present in the server's directory, the content of the '500.tmp1t' file. This templates can used the following tags:

AUTH_MODE

The authorization mode (Either NONE, BASIC or DIGEST).

EXCEPTION

Exception information with traceback if activated.

HTTP_VERSION

Either HTTP/1.0 or HTTP/1.1

METHOD The request method (Either GET, HEAD, POST or PUT)

PAYLOAD

The full XML payload for SOAP request.

PEERNAME

The IP address of the client

SOAP_ACTION

Either True or False. Set to True for a SOAP request.

URI

The complete URI

For more information see [Section B.42 \[AWS.Server\], page 124](#) and see [Section B.16 \[AWS.Exceptions\], page 98](#).

3.16 Socket log

To ease AWS applications debugging it is possible to log all data sent/received to/from the sockets. For this you need to call the `AWS.Net.Log.Start` routine by passing a write procedure callback. You have to create such procedure or use one read-to-use provided in `AWS.Net.Log.Callbacks` package.

For more information see [Section B.26 \[AWS.Net.Log\]](#), page 108 and see [Section B.27 \[AWS.Net.Log.Callbacks\]](#), page 109.

3.17 Client side

AWS is not only a server it also implement the HTTP and HTTPS protocol from the client side. For example with AWS it is possible to get a Web page content using the `AWS.Client` API, See [Section B.3 \[AWS.Client\]](#), page 85.

It also support client **Keep-Alive** connections. It is then possible to request many URI from the same server using the same connection (i.e. the same sockets).

AWS client API also support proxy, proxy authentication and Web server authentication. Only basic (and not digest) authentication is supported at this time.

Let's say that you want to retrieve the `contrib.html` Web page from Pascal Obry's homepage which is <http://perso.wanadoo.fr/pascal.obry>. The code to do so is:

```
Data := Client.Get
  (URL => "http://perso.wanadoo.fr/pascal.obry/contrib.html");
```

From there you can ask for the result's content type:

```
if Response.Content_Type (Data) = "text/html" then
  ...
end if;
```

Or using the MIME types defined in `AWS.MIME` unit:

```
if Response.Content_Type (Data) = MIME.Text_HTML then
  ...
end if;
```

And display the content if it is some kind of text data:

```
Text_IO.Put_Line (Response.Message_Body (Data));
```

If the content is some kind of binary data (executable, PNG image, Zip archive...), then it is possible to write the result to a file for example. Look at the `agent` program in the `demo` directory.

If the Web page is protected and you must pass the request through an authenticating proxy, the call will becomes:

```
Data := Client.Get
  (URL      => "http://www.mydomain.net/protected/index.html"
  User      => "me",
  Pwd       => "mypwd",
  Proxy     => "192.168.67.1",
  Proxy_User => "puser",
  Proxy_Pwd => "ppwd");
```

The client upload protocol is implemented. Using `AWS.Client.Upload` it is possible to send a file to a server which support the file upload protocol.

4 High level services

Here you will find a description of high level services. These services are ready to use with AWS and can be used together with user's callbacks.

Refer to the Ada spec for a complete API and usage description.

4.1 Directory browser

This service will help building a Web directory browser. It has a lot of options to sort directory entries and is based on the templates interface see [Section B.70 \[AWS.Templates\], page 152](#). This means that you can use the default directory template or provide your own.

see [Section B.48 \[AWS.Services.Directory\], page 130](#) for complete spec and services descriptions.

4.2 Dispatchers

In many AWS applications it is needed to check the URI to give the right answer. This means that part of the application is a big **if/elsif** procedure. Also, in standard callback it is not possible to have user data. Both of these restrictions are addressed with the Dispatchers facilities.

Working with a dispatcher is quite easy:

1. Create a new dispatcher by inheriting from the service you want to build.
2. Register a set of action based on rules (strings, regular expressions depending on the service)

4.2.1 Callback dispatcher

This is a wrapper around the standard callback procedure. It is needed to mix dispatcher based callback and access to procedure callback. Note that it is not in the `AWS.Services.Dispatchers` hierarchy but in `AWS.Dispatchers.Callback` because this is a basic service needed for the server itself. It is referenced here for documentation purpose but an AWS server can be built with using it.

see [Section B.15 \[AWS.Dispatchers.Callback\], page 97](#) for complete spec description.

4.2.2 Method dispatcher

This is a dispatcher based on the request method. A different callback procedure can be registered for the supported request methods: GET, POST, PUT, HEAD.

see [Section B.51 \[AWS.Services.Dispatchers.Method\], page 133](#) for complete spec description.

4.2.3 URI dispatcher

This is a dispatcher based on the request resource. A different callback procedure can be registered for specific resources. The resource is described either by its full name (string) or a regular expression.

see [Section B.52 \[AWS.Services.Dispatchers.URI\], page 134](#) for complete spec description.

4.2.4 Virtual host dispatcher

This is a dispatcher based on the host name. A different callback procedure can be registered for specific host. This is also known as virtual hosting.

The same computer can be registered into the DNS with different names. So all names point to the same machine. But in fact you want each name to be seen as a different Web server. This is called virtual hosting. This service will just do that, call different **callback** procedures or redirect to some **machine/port** based on the host name in the client's request.

see [Section B.53 \[AWS.Services.Dispatchers.Virtual_Host\], page 135](#) for complete spec description.

4.2.5 Transient pages dispatcher

This is a dispatcher that calls a user's callback and if the resource requested is not found (i.e. the user's callback returns status code 404) it checks if this resource is known as a transient page. see [Section 4.4 \[Transient Pages\], page 35](#).

4.2.6 Timer dispatcher

A timer dispatcher can be used to call different callback routines depending on the current date and time. Such dispatcher is composed of a set of **Period** activated. When the current date and time is inside a **Period** the corresponding callback is called. A **Period** can eventually be repeated. Here are the different kind of **Period** supported by AWS:

Once

A unique period in time. The boundaries are fully described using a year, month, day, hour, minute and second.

Yearly

A period that repeats each year. The boundaries are described using a month, day, hour, minute and second.

Monthly

A period that repeats each month. The boundaries are described using a day, hour, minute and second.

Weekly

A period that repeats each week. The boundaries are described using a day name, hour, minute and second.

Daily

A period that repeats each day. The boundaries are described using an hour, minute and second.

Hourly

A period that repeats each hour. The boundaries are described using a minute and second.

Minutely

A period that repeats each minute. The boundaries are described using a second.

4.2.7 Linker dispatcher

A dispatcher that can be used to chain two dispatchers. The response of the first dispatcher is returned except if it is a 404 (Not Found) error. In this case, the response of the second dispatcher is returned.

4.2.8 SOAP dispatcher

AWS provides also a SOAP specific dispatcher. This is a way to automatically route HTTP requests or SOAP requests to different callback routines.

see [Section 5.2.2 \[SOAP helpers\], page 51](#) for more information.

see [Section B.76 \[SOAP.Dispatchers.Callback\], page 158](#) for complete spec description.

4.3 Static Page server

This service is a ready to use static page server callback. Using it is possible to build a simple static page server, as simple as:

```
with AWS.Server;
with AWS.Services.Page_Server;

procedure WPS is
  WS : AWS.Server.HTTP;
begin
  AWS.Server.Start
    (WS, "Simple Page Server demo",
     Port      => 8080,
     Callback  => AWS.Services.Page_Server.Callback'Access,
     Max_Connection => 5);

  AWS.Server.Wait (AWS.Server.Q_Key_Pressed);

  AWS.Server.Shutdown (WS);
end WPS;
```

Build this program and launch it, it will server HTML pages and images in the current directory.

It is possible to activate the directory browsing facility of this simple page server. This is not activated by default. This feature is based on the directory browsing service see [Section 4.1 \[Directory browser\]](#), page 33.

Note that this service uses two template files:

aws_directory.thtml

The template page used for directory browsing. See [Section B.48 \[AWS.Services.Directory\]](#), page 130 for a full description of this template usage.

404.thtml

The Web page returned if the requested page is not found. This is a template with a single tag variable named PAGE. It will be replaced by the resource which was not found.

Note that on Microsoft IE this page will be displayed only if the total page size is bigger than 512 bytes or it includes at least one image.

see [Section B.55 \[AWS.Services.Page_Server\]](#), page 137 for a complete spec description.

4.4 Transient Pages

A transient page is a resource that has a certain life time on the server. After this time the resource will be released and will not be accessible anymore.

Sometimes you want to reference, in a Web page, a resource that is built in memory by the server. This resource can be requested by the client (by clicking on the corresponding link) or not, in both cases the page must be released after a certain amount of time to free the associated memory.

This is exactly what the transient pages high level service do automatically. Each transient page must be registered into the service, a specific routine named **Get_URI** can be used to create a unique URI on this server. see [Section B.62 \[AWS.Services.Transient_Pages\]](#), page 144.

A transient pages dispatcher can be used to build a transient pages aware server. see [Section 4.2.5 \[Transient pages dispatcher\]](#), page 34.

4.5 Split pages

It is not very convenient to send back a Web page with a large table. In such a case it is better to split the table in chunks (20 lines or so) and to send only the first page. This page references the next pages and can also contain an index of the pages.

The AWS's split page feature can automatically do that for you. Given template `Translate_Table` or `Translate_Set` and the max line per page it returns the first page and creates a set of transient pages for all other pages. A set of template tags are used to reference the previous and next page and also to build the page index.

There are different ways to split a set of pages and ready-to-use splitters are available:

Alpha Split in (at most) 28 pages, one for empty fields, one for all fields that start with a digit, and one for each different initial letter. see [Section B.57 \[AWS.Services.Split_Pages.Alpha\]](#), page 139.

Alpha.Bounded

Same as the alpha splitter, but pages larger than a `Max_Per_Page` value are further splitted. A secondary index is generated that gives the various pages for a given letter. see [Section B.58 \[AWS.Services.Split_Pages.Alpha.Bounded\]](#), page 140.

Uniform Split in pages of length `Max_Per_Page` (except the last one). This corresponds to the default service in `Split_Pages` package. see [Section B.59 \[AWS.Services.Split_Pages.Uniform\]](#), page 141.

Uniform.Alpha

Same as the uniform splitter, but builds in addition an alphabetical secondary index from a key field. see [Section B.60 \[AWS.Services.Split_Pages.Uniform.Alpha\]](#), page 142.

Uniform.Overlapping

Same as the uniform splitter, but pages (except the first one) repeat `Overlap` lines from the previous page in addition to the `Max_Per_Page` lines. see [Section B.61 \[AWS.Services.Split_Pages.Uniform.Overlapping\]](#), page 143.

Using the splitter abstract interface it is possible to build a customized splitter algorithm. see [Section B.56 \[AWS.Services.Split_Pages\]](#), page 138.

4.6 Download Manager

A server that needs to handle a lot of large downloads can run out of connection to answer the standard Web pages. A solution is to increase the number of simultaneous connections, but this is not really efficient as a task is created for each connection and does not ensure that all the connections will be used for the downloads anyway.

The download manager can be used for that, and provides the following feature:

- use a single task for all downloads
- can be configured to limit the number of simultaneous connections
- downloads past this limit are queued
- send messages to the client with the position in the waiting line
- send messages to the client when the download is about to start

The server must be configured to use dispatchers (standard callbacks are not supported, note that it is possible to create a dispatcher for standard callbacks. see [Section B.15 \[AWS.Dispatchers.Callback\]](#), page 97).

To start the download manager you need to pass the main server dispatcher object. The start routine will return a new dispatcher, linked with the download server specific dispatcher,

that must be used to start the standard Web server. See comment in see [Section B.54 \[AWS.Services.Download\]](#), page 136.

To queue a download request in the download manager you just need to create a stream object (can be any kind of stream, see `AWS.Resources.Streams.*`) for the resource to download.

The download manager needs two templates files:

`aws_download_manager_waiting.thtml`

This template is used for sending a message to the client when the request is on the waiting line. The tags defined in this template file are:

NAME the name of the resource to download (the filename), this is the default filename used for the client side save dialog.

RES_URI the URI used to access the resource.

POSITION the position in the waiting line (not counting the current served clients).

`aws_download_manager_start.thtml`

This template is used for sending a message to the client when the download is about to start (the request is out of the waiting line). The tags defined in this template file are:

NAME as above

RES_URI as above

It is important to note that those templates must be reloaded periodically. The best way to do that in the context of an HTML document is to use a meta-tag. For example to refresh the page every two seconds:

```
<meta http-equiv="refresh" content="2">
```

The templates could look like:

`aws_download_manager_waiting.thtml`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <meta http-equiv="refresh" content="2">
  <title>Download Manager - waiting</title>
</head>
<body>
  <p>Waiting for downloading @_NAME_@
  <p>Position in the waiting line @_POSITION_@
</body>
</html>
```

aws_download_manager_start.shtml

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <meta http-equiv="refresh" content="2">
  <title>Download Manager - waiting</title>
</head>
<body>
  <p>Waiting for downloading @_NAME_@
  <p>The download will start in a moment
</body>
</html>
```

4.7 Web Elements

AWS provides some components to help creating nice looking Web interfaces. It is possible to browse those Web Elements using the `web_elements` demo. Just launch this Web application from the demos directory and turn your Web browser to <http://localhost:2400>.

Currently AWS provides:

- Notebooks (based on CSS)
- CSS Menu
- Rounded boxes
- Ajax

All of them are based on templates to be easily reused in other applications. The three first are best described by the Web Elements demos as they are 100% design. The Ajax one is a bit more complex, we will present its use in the following section.

4.7.1 Installation

To ease integration we have used the following design:

- Sub-directories found in the AWS's `web_elements` directory are self contained. The content must be copied into the project. Note that the icons and javascripts directories contain the icons and javascripts code shared by all web elements and must also be copied, see below.
- Each graphic elements (icons) is referenced into the templates with the alias `/we_icons/<icon_name>`. So users must provide the right alias (`"/we_icons/"`) in the Web server.
- Each JavaScripts code is referenced into the templates with the alias `/we_js/<script>`. So users must provide the right alias (`"/we_js/"`) in the Web server.

4.7.2 Ajax

First of all, Ajax stand for *Asynchronous JavaScript language and XML*, and is not well defined at the moment. Ajax is on one side able to send HTTP requests to the Web server and on the other side able to manipulate directly the Web browser's DOM tree. On the DOM it can add, remove or replace XML nodes. So, it is possible to change the content of a Web page without reloading it from the server.

Most importantly, Ajax changes the way Web applications are thought from **page** based to **event** based.

As implemented into AWS, Ajax support comes as a set of JavaScript templates. Using those templates there is no need to know JavaScript (except for the JavaScript event names) and

it makes **Ajax** programming lot easier. Two actions are provided, one for replacing another for clearing part of the web page content.

4.7.2.1 Steps to do Ajax

What are the steps to do **Ajax** ?

Remember, do not think about the Web page but about a specific widget (HTML fragments) with the associated event and action.

1. Include the AWS/Ajax support file

This is the **AWS/Ajax** runtime, it contains **JavaScript** code needed for the **AWS/Ajax** support.

2. Create the Web widgets/forms

There is nothing special here, use your favorite Web designer tool.

3. Create Web area

Using some **HTML** `<div>` tags we create areas where we will place **HTML** fragments later. For example when clicking on a button (described above) in our Web interface we want to display a new form in this area.

4. Name the widgets/forms/area using `id="name"` attribute

Give a different name to the widgets using `id="name"`. This name will be later used to identify the widgets on which the event and corresponding action must be placed. We do not want to clutter the Web design with **JavaScript** code like `onclick="dothis()"` or `onchange="dothat()"`.

5. Add the proper event/action to the widgets using the **AWS/Ajax** templates

This is the interesting part. At this point we link events/actions to the widgets and specify in which area the results sent by the server will be placed.

This is not the only way to do **Ajax**, we just presented here a simple approach that works well with the **AWS/Ajax** templates.

4.7.2.2 Basic Ajax support

This section describes the **AWS/Ajax** support where the answer from the server is an **HTML** fragment. This basic support is designed to be used for migration of a Web server to **Ajax**. For new applications, it is worth considering using the **XML** based **Ajax** support, see [Section 4.7.2.3 \[XML based Ajax\]](#), page 40.

Let's have a very simple example:

- The **AWS/Ajax** runtime support

```
@@INCLUDE@@ aws.tjs
```

- The widget: a button

```
<input id="clickme" type="button" value="Clik Me">
```

- The result area: a div

```
<div id="placeholder">... result here ...</div>
```

- The **AWS/Ajax**

```
@@INCLUDE@@ aws_action_replace.tjs onclick clickme placeholder
```

Basically it places an **onclick** attribute (the event) in the HTML **<input>** identified as **clickme** (the action) above. Here is what happen when the button is clicked:

- send the `"/clickme"` HTTP request to the server
- asynchronously wait for the answer, when received place the message body into the `<div>` **placeholder**.

On the server side the code would look like this:

```
function Callback (Request : in Status.Data) return Response.Data is
  URI : constant String := Status.URI (Request);
begin
  if URI = "/clickme" then
    return Response.Build (MIME.Text_HTML, "you click me!");
  ...
```

So when the button is clicked the string **"you click me!"** will replace the **"... result here ..."** string of the place holder div above.

This is a simple and very limited example as there is no parameter passed to the HTTP request. In real Web applications it is necessary to send a context with the request. This can be either the value of other widgets or all values of widgets' form.

References to widgets or forms can be passed to the `'aws_action_replace.tjs'` template starting with the 5th parameter.

```
<input id="field" type="text" value="default value">
...
@@INCLUDE@@ aws_action_replace.tjs (onclick clickme placeholder 5=>field)
```

OR

```
<form id="small_form" name="small_form">
...
</form>
@@INCLUDE@@ aws_action_replace.tjs (onclick clickme placeholder 5=>small_form)
```

Note that the **onclick** event is only one of the possible JavaScript event on a **button**. It is possible to used any supported event, for example on an HTML **<select>** widget it is common to map the action to the **onchange** event.

AWS also provides support for clearing an area or a widget content (like an input).

```
@@INCLUDE@@ aws_action_clear.tjs (onclick, clear, field)
```

This simple action adds the **onclick** event to the **clear** button to erase the content of the **field** widget.

4.7.2.3 XML based Ajax

In many cases you'll like to update and/or clear multiple areas in your Web interface. With the templates above only a single action is possible. AWS provides support for XML based answers. In this XML documents it is possible to:

- replace an area with a new content

```
<replace id="item_id">new text</replace>
```

- clear an area

```
<clear id="item_id"/>
```

- add an item into a select widget

```
<select action="add" id="item_id"  
option_value="value" option_content="content"/>
```

- remove an item from a select widget

```
<select action="delete" id="item_id" option_value="value"/>
```

- select a specific item in a select widget

```
<select action="select" id="item_id" option_value="value"/>
```

- clear a select widget (remove all items)

```
<select action="clear" id="item_id"/>
```

- select a radio button

```
<radio action="select" id="item_id"/>
```

- check a checkbox

```
<check action="select" id="item_id"/>
```

- clear a checkbox

```
<check action="clear" id="item_id"/>
```

- call another URL

```
<get url="http://thishost/action">  
  <parameters value="name=Ajax"/>  
  <field id="input1"/>  
</get>
```

This will send the following request:

```
http://thishost/action?name=Ajax&input1=<val_input1>
```

Where **val_input1** is the current value of the **input1** input widget. The result must be an XML/Ajax document that will be parsed.

- make a list sortable

```
<make_sortable>
  <list id="firstlist"/>
  <list id="secondlist"/>
</make_sortable>
```

Here **firstlist** and **secondlist** are **id** of UL elements. It is possible to specify as many list id as needed. A drag and drop is then possible for all elements in those lists. It is then possible to reference such list by passing the list id as a field to the template. Items on those list will be serialized and passed to the **AWS** callback. Note that for the serialization to work properly, each LI elements must be given the id of the list and then the value we want to pass.

```
<ul id="firstlist">
  <li id="firstlist_red">Red</li>
  <li id="firstlist_green">Green</li>
  <li id="firstlist_blue">Blue</li>
</ul>
```

The serialization will send each value on this list using a multi-valued parameter named **firstlist[]**.

```
http://server?firstlist[]=red&firstlist[]=green&firstlist[]=blue
```

- make a list not sortable

```
<destroy_sortable>
  <list id="firstlist"/>
  <list id="secondlist"/>
</destroy_sortable>
```

Remove the sortable properly from the specified lists.

- redirect to another URL

```
<location url="http://thishost/go_there"/>
```

Redirect the browser to the specified URL.

- refresh the current page

```
<refresh/>
```

Refresh the current page as if the Web Browser refresh button was pressed.

- Add a CSS style to a given node

```
<apply_style id="node_id">
  <attribute id="display" value="none"/>
</apply_style>
```


Add the CSS style `display:none` to the `node_id` element. It is possible to specify multiple attributes if needed.

Here is an example of such XML document:

```
<response>
  <replace id="xml_status_bar">Fill Widgets...</replace>
  <replace id="text1">Response from XML</replace>
  <replace id="text2">Another response for text2</replace>
  <replace id="input1">tag is input1</replace>
  <replace id="input2">tag is input2</replace>
  <select action="add" id="xmlsel" option_value="one" option_content="1"/>
  <select action="add" id="xmlsel" option_value="two" option_content="2"/>
  <select action="add" id="xmlsel" option_value="three" option_content="3"/>
  <select action="select" id="xmlsel" option_value="two"/>
  <radio action="select" id="radio1"/>
  <check action="select" id="check1"/>
  <check action="select" id="check3"/>
  <check action="clear" id="check2"/>
</response>
```

The template to use this feature is `'aws_action_xml.tjs'`. The usage is similar to what is described in the previous section (see [Section 4.7.2.2 \[Basic Ajax support\]](#), page 39) expect that in this case we do not have to pass the placeholder.

Let's revisit the first example above to use the XML Ajax support.

- The AWS/Ajax runtime support

```
@@INCLUDE@@ aws.tjs
```

- The widget: a button

```
<input id="clickme" type="button" value="Clik Me">
```

- The result area: a div

```
<div id="placeholder">... result here ...</div>
```

- The AWS/Ajax

```
@@INCLUDE@@ aws_action_xml.tjs onclick clickme
```

Basically it places an **onclick** attribute (the event) in the HTML `<input>` identified as **clickme** (the action) above. Here is what happen when the button is clicked:

- send the `"/clickme"` HTTP request to the server
- asynchronously wait for the XML answer, when received parse the answer and perform the actions according to the XML content.

To set the placeholder with **"new text"**, the XML document returned by the server must be:

```
<response>
  <replace id="placeholder">new text</replace>
</response>
```

If we want also to clear the input field named **field** and to select the radio button named **radio1** we must return:

```
<response>
  <replace id="placeholder">new text</replace>
  <clear id="field"/>
  <radio action="select" id="radio1"/>
</response>
```

This is by far the most flexible solution as it is possible to return, from the server, a structured answer.

A final comment, if the text returned by the server to replace a specific area is an HTML fragment, the content must be placed into a CDATA tag:

```
<response>
  <replace id="item_id">
    <![CDATA[ HTML CODE HERE ]]>
  </replace>
</response>
```

4.7.2.4 Advanced Ajax

Finally, if this is not enough because you need to use some specific JavaScript code, AWS provides a template to add an event to a specific widget, the action being the name of a JavaScript routine, see 'aws_action_js.tjs'.

This template together with 'aws_func_replace.tjs', 'aws_func_clear.tjs' and 'aws_func_xml.tjs' can be used to chain multiple actions. Those templates are the function body used by the corresponding templates 'aws_action_replace.tjs', 'aws_action_clear.tjs' and 'aws_action_xml.tjs'.

Let say you want to clear a widget, change the content of another one and calling one of your specific JavaScript routine when clicking on a button. It is not possible to have mutiple onclick events on the same widget, the solution is the following:

- Create the JavaScript routine to do the job

For this in the the body of the clear_replace() JavaScript routine we place:

```
function clear_replace()
{
```

```
  @@INCLUDE@@ aws_func_replace.tjs (clickme placeholder 4=>field)
  @@INCLUDE@@ aws_func_clear.tjs (area)
  call_this_routine();
```

```
}
```

Then to add the event on the widget:

```
@@INCLUDE@@ aws_action_js.tjs (onclick clickme clear_replace)
```

Furthermore, it is possible to pass (as the parameter number 20) a routine to call after a specific action to all templates. This is another way to chain multiple actions for a single event.

Note that all AWS/Ajax templates have a set of comments at the start explaining in details the usage of each parameter.

4.8 Web Block Context

The `AWS.Services.Web_Block` hierarchy contains an API useful for keeping context on Web pages. It has been designed to be able to split a Web application into a set of independent blocks that can be put together in the same Web page. The context is then useful as it is passed and known by each individual block. Note that this is different than the session as a session is global to the current Web browser whereas the context can be different for each individual web pages opened.

Instead of parsing a whole page using `AWS.Templates` API the web blocks are registered independently using `AWS.Services.Web_Block.Registry`. The block is registered together with its templates and a callback to use to get user's data for this specific block with the given context.

So using this API instead of having a set of callbacks returning an `AWS.Response.Data` and where the final rendering is to be done by the client code we have a set of callbacks that returns a `Translate_Set`. The client just have to fill the set with the data corresponding to the actual request and possibly using the context. The final rendering is done by the provided services in `Web_Block.Registry`.

It is possible to use the `Templates_Parser`'s `templates2ada` tool for generating the callbacks register calls. This ensures that all tags on the application Web Pages have a corresponding callback.

Let's have a simple example, a block with a single tag (`@_LAZY_COUNTER_@`) that is incremented by one each time it is used.

First create the following simple HTML fragment and place it into '`templates/counter.thtml`'. Note that by convention the lazy tags (see `Templates_Parser` documentation) start with the prefix `LAZY_`.

```
<p>@_LAZY_COUNTER_@</p>
```

The `Web_Callbacks` package contains the application callbacks.

```
with AWS.Status;
with AWS.Templates;
with AWS.Services.Web_Block.Context;

package Web_Callbacks is

  use AWS;
  use AWS.Services;

  procedure Counter
    (Request      : in      Status.Data;
     Context      : access Web_Block.Context.Object;
     Translations : in out AWS.Templates.Translate_Set);

end Web_Callbacks;
```

We need also modify the standard '`templates.tads`' as distributed with the `Templates_Parser` to add the following template code:

```

with AWS.Services.Web_Block.Registry;
with Web_Callbacks;

package body @_PACKAGE_@ is

  package body Lazy is

    procedure Register;
    -- Register lazy tags into the AWS's Web Block framework

    -----
    -- Register --
    -----

    procedure Register is
      use AWS.Services;
    begin
    @@TABLE@@
    @@IF@@ @_UPPER:SLICE(1..5):VARIABLE_LIST_@ = "LAZY_"
      Web_Block.Registry.Register
        ("@_VARIABLE_LIST_@",
         "templates/@_LOWER:REPLACE_ALL(LAZY_/):VARIABLE_LIST_@.thtml",
         Web_Callbacks. @_CAPITALIZE:REPLACE_ALL(LAZY_/):VARIABLE_LIST_@'Access);
    @@END_IF@@
    @@END_TABLE@@
    end Register;

    begin
      Lazy.Register;
    end Lazy;

  end @_PACKAGE_@;

```

Basically this is to write a register call for every template's tag. The registration is done during elaboration. The templates can be tailored as needed, for example it is also possible to generate all callback's specs.

Now let's parse the template HTML fragment and create the corresponding Ada spec:

```

$ templates2ada -d templates -o code.ada -r templates.tads
$ gnatchop code.ada

```

Look at the generated code, it properly register the **Counter** callback to be used for rendering **LAZY_COUNTER** using the 'templates/counter.thtml'. So we have a tight coupling between the code and the template file. If the tag is renamed in the template file the application will not compile anymore. This greatly helps keeping the application code synchronized.

```

procedure Register is
  use AWS.Services;
begin
  Web_Block.Registry.Register
    ("LAZY_COUNTER",
     "templates/counter.thtml",
     Web_Callbacks.Counter'Access);
end Register;

```

Last part is to actually implement the **Counter** callback. Here is a possible implementation making use of the context to keep the counter state.

```

with AWS.Utils;
with Templates;

package body Web_Callbacks is

  procedure Counter
    (Request      : in      Status.Data;
     Context      : access Web_Block.Context.Object;
     Translations : in out AWS.Templates.Translate_Set)
  is
    N : Natural := 0;
  begin
    if Context.Exist ("N") then
      N := Natural'Value (Context.Get_Value ("N"));
    end if;

    N := N + 1;
    Context.Set_Value ("N", Utils.Image (N));

    AWS.Templates.Insert
      (Translations,
       AWS.Templates.Assoc (Templates.Lazy.Lazy_Counter, N));
  end Counter;

end Web_Callbacks;

```

4.9 Web Cross-References

When building an Ajax Web applications it is required to give ids to web elements to be able to reference them. It is also quite common to use CSS to give such and such item a specific style. After some time it is quite difficult to keep track of all those ids. Are they all used ? Don't we reference an id that does not exist anymore ?

webxref has been designed to help finding such problems.

The files kinds handled are:

‘.css’

‘.tcss’

A CSS (or template CSS file). Ids and classes inside are recorded as CSS definitions.

‘.xml’

‘.html’

‘.thtml’

A meta-language document. Ids and classes inside are recorded as referencing a CSS definition and meta-language definition.

‘.txml’

An Ajax response file. Ids declared inside are recorded as referencing a meta-language definition.

The features are:

cross-references

By default **webxref** output all the references to ids and classes.

finding unused items

Output the ids/classes that are defined but not used. For example an id declared in a CSS but never referenced into an HTML document or an HTML id never referenced in an Ajax response file ‘.txml’ document.

finding undeclared items

Output ids/classes that are referenced but never defined. This is for example an id inside an Ajax response file which is never defined into an HTML document.

enforcing a naming scheme for ids and classes

It can enforce a specific prefix for ids and classes. The id prefix can be based on the filename (using filename's first character and all character before an underscore). This make it less likely to find the same id on multiple files.

Note that all references are in a format recognized by tools like **GPS** and **Emacs**. It is then possible to navigate inside them easily.

All **webxref** options are listed using the **-h** option.

5 Using SOAP

SOAP can be used to implement Web Services. The SOAP implementation uses AWS HTTP as the transport layer. SOAP is platform and language independent, to ensure a good inter-operability, AWS/SOAP implementation has been validated through <http://validator.software.org/>, the version number listed on this server corresponds to the AWS version string (AWS.Version) concatenated with the SOAP version string (SOAP.Version).

This SOAP implementation is certainly one with the higher level of abstraction. No need to mess with a serializer, to know what is a payload or be an XML expert. All the low level stuffs are completely hidden as the SOAP type system has been binded as much as possible to the Ada type system.

The SOAP type system has been relaxed to be compatible with WSDL based SOAP implementation. In these implementations, types are generally (as in the Microsoft implementation) not part of the payload and should be taken from the WSDL (Web Services Description Language). AWS/SOAP is not WSDL compliant at this stage, all such types are binded into the Ada type system as strings. It is up to the programmer to convert such strings to the desired type.

5.1 SOAP Client

The SOAP client interface is quite simple. Here are the step-by-step instructions to call a SOAP Web Service:

1. Build the SOAP parameters

As for the SOAP servers, the SOAP parameters are built using a SOAP.Parameters.List object.

```
Params : constant Parameters.List
:= +I (10, "v1") & I (32, "v2");
```

2. Build the SOAP Payload

The Payload object is the procedure name and the associated parameters.

```
declare
  Payload : Message.Payload.Object;
begin
  Payload := Message.Payload.Build ("Add", Params);
```

3. Call the SOAP Web Service

Here we send the above Payload to the Web Server which handles the Web Service. Let's say that this server is named myserver, it is listening on port 8082 and the SOAPAction is soapdemo.

```
Resp : constant Message.Response.Object'Class :=
  SOAP.Client.Call ("http://myserver:8082/soapdemo", Payload);
```

4. Retrieve the result

Let's say that the answer is sent back into the parameter named "myres", to get it:

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

In the above example we have called a Web Service whose spec could be described in Ada as follow:

```
function Add (V1, V2 : in Integer) return Integer;
-- Add V1 and V2 and returns the result. In SOAP the result is named "myres"
```

5.2 SOAP Server

A SOAP server implementation must provides a callback procedure as for standard Web server see [Section 3.2.2 \[Callback procedure\]](#), page 10. This callback must checks for the SOAP Action URI to handle both standard Web requests and SOAP ones. The `SOAPAction` is sent with the HTTP headers and can be retrieved using `AWS.Status.SOAPAction`.

5.2.1 Step by step instructions

Here are the step-by-step instructions to be followed in the SOAP callback procedure:

1. Retrieve the SOAP Payload

The SOAP Payload is the XML message, it contains the procedure name to be called and the associated parameters.

```
function SOAP_CB (Request : in AWS.Status.Data)
  return AWS.Response.Data
is
  use SOAP.Types;
  use SOAP.Parameters;

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

`AWS.Status.Payload` returns the XML Payload as sent by the SOAP Client. This XML Payload is then parsed using `SOAP.Message.XML.Load_Payload` which returns a `SOAP.Message.Payload.Object` object.

2. Retrieve the SOAP Parameters

The SOAP procedure's parameters.

```
Params : constant SOAP.Parameters.List
  := SOAP.Message.Parameters (Payload);
```

`SOAP.Parameters.List` is a structure which holds the SOAP parameters. Each parameter can be retrieved using a `SOAP.Parameters` API, see [Section B.79 \[SOAP.Parameters\]](#), page 161. For example to get the parameter named `myStruc` which is a SOAP struct:

```
My_Struct : constant SOAP_Record
  := SOAP.Parameters.Get (Params, "myStruct");
```

Another example, to get the parameter named `myInt` which is a SOAP integer:

```
My_Int : constant Integer := SOAP.Parameters.Get (Params, "myInt");
```

3. Implements the Web Service

This is the real job, as for any procedure you can do whatever is needed to compute the result.

4. Build the SOAP answer

This is the procedure answer. A SOAP answer is built from the SOAP Payload and by setting the returned parameters.


```

declare
    Resp          : SOAP.Message.Response.Object;
    Resp_Params   : SOAP.Parameters.List;
begin
    Resp := SOAP.Message.Response.From (Payload);

    Resp_Params := +I (My_Int * 2, "answer");

    SOAP.Message.Set_Parameters (Resp, Resp_Params);

```

This build a response which is a single integer value named **answer** with the value **My_Int * 2**.

5. Returns the answer back to the client

This last step will encode the response object in XML and will returns it as the body of an HTTP message.

```

return SOAP.Message.Response.Build (Resp);

```

5.2.2 SOAP helpers

There is two ways to help building the SOAP callbacks. AWS provides a SOAP specific callback, the spec is :

```

function SOAP_Callback
(SOAPAction : in String;
Payload      : in Message.Payload.Object;
Request      : in AWS.Status.Data)
return AWS.Response.Data;

```

With both solutions exposed below, AWS retrieve the SOAPAction and the Payload from the SOAP request. This is transparent to the user.

1. Using Utils.SOAP_Wrapper

It is possible to dispatch to such callback by using the SOAP.Utils.SOAP_Wrapper generic routine.

```

generic
    with function SOAP_CB
        (SOAPAction : in String;
         Payload      : in Message.Payload.Object;
         Request      : in AWS.Status.Data)
        return AWS.Response.Data;
    function SOAP_Wrapper
        (Request : in AWS.Status.Data)
        return AWS.Response.Data;
-- From a standard HTTP callback call the SOAP callback passed as generic
-- formal procedure. Raise Constraint_Error if Request is not a SOAP
-- request.

```

For example, from the standard HTTP callback CB we want to call SOAP_CB for all SOAP requests:

```

function SOAP_CB
(SOAPAction : in String;
 Payload    : in Message.Payload.Object;
 Request    : in AWS.Status.Data)
return AWS.Response.Data
is
begin
    -- Code here
end SOAP_CB;

procedure SOAP_Wrapper is new SOAP.Utls.SOAP_Wrapper (SOAP_CB);

function CB (Request : in AWS.Status.Data)
return AWS.Response.Data
is
    SOAPAction : constant String := Status.SOAPAction (Request);
begin
    if SOAPAction /= "" then
        SOAP_Wrapper (Request);
    else
        ...
    end if;
end CB;

```

2. Using a SOAP Dispatcher

AWS provides also a SOAP specific dispatcher. This dispatcher will automatically calls a standard HTTP or SOAP callback depending on the request. If `SOAPAction` is specified (i.e. it is a SOAP request), the dispatcher will call the SOAP callback otherwise it will call the standard HTTP callback. This is by far the easiest integration procedure. Using dispatcher the above code will be written:

```

function SOAP_CB
(SOAPAction : in String;
 Payload    : in Message.Payload.Object;
 Request    : in AWS.Status.Data)
return AWS.Response.Data
is
begin
    -- Code here
end SOAP_CB;

function CB (Request : in AWS.Status.Data)
return AWS.Response.Data
is
    SOAPAction : constant String := Status.SOAPAction (Request);
begin
    -- Code here
end CB;

-- In the main procedure
begin
    AWS.Server.Start
    (WS,
     Dispatcher =>
        SOAP.Dispatchers.Callback.Create (CB'Access, SOAP_CB'Access),
     Config     =>
        AWS.Config.Default_Config);
end;

```

The dispatcher is created using `SOAP.Dispatchers.Callback.Create`. This routine takes two parameters, one is the standard HTTP callback procedure and the other is the SOAP callback procedure.

6 Using WSDL

WSDL (Web Service Definition Language) is an XML based document which described a set of Web Services either based on SOAP or XML/RPC. By using a WSDL document it is possible to describe, in a formal way, the interface to any Web Services. The WSDL document contains the end-point (URL to the server offering the service), the SOAPAction (needed to call the right routine), the procedure names and a description of the input and output parameters.

AWS provides two tools to work with WSDL documents:

`'ada2wsdl'`

which creates a WSDL document from an Ada package spec.

`'wsdl2aws'`

which create the interfaces to use a Web Service or to implement Web Services. With this tool the SOAP interface is completely abstracted out, users will deal only with Ada API. All the SOAP marshaling will be created automatically.

6.1 Creating WSDL documents

Note that this tool is based on ASIS.

6.1.1 Using ada2wsdl

`ada2wsdl` can be used on any Ada spec file to generated a WSDL document. The Ada spec is parsed using ASIS.

The simplest way to use it is:

```
$ ada2wsdl simple.ads
```

Given the following Ada spec file:

```
package Simple is
  function Plus (Value : in Natural) return Natural;
end Simple;
```

It will generate the following WSDL document:

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Simple"
  targetNamespace="urn:aws:Simple"
  xmlns:tns="urn:aws:Simple"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="Plus_Request">
    <part name="Value" type="xsd:int"/>
  </message>

  <message name="Plus_Response">
    <part name="Result" type="xsd:int"/>
  </message>

  <portType name="Simple_PortType">
    <operation name="Plus">
      <input message="tns:Plus_Request"/>
      <output message="tns:Plus_Response"/>
    </operation>
  </portType>

  <binding name="Simple_Binding" type="tns:Simple_PortType">
    <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>

    <operation name="Plus">
      <soap:operation soapAction="Plus"/>
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:aws:Simple"
          use="encoded"/>
      </input>
      <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:aws:Simple"
          use="encoded"/>
      </output>
    </operation>
  </binding>

  <service name="Simple_Service">
    <port name="Simple_Port" binding="tns:Simple_Binding">
      <soap:address location="http://.../">
    </port>
  </service>
</definitions>

```

In bold are marked the important parts from a spec point of view. The first item is the name of the WSDL document (the name of the Ada spec package). On the **portType** section we have the description of the Ada **Plus** function. Something important to note is that in Ada a function does not have a named return parameter, ‘ada2wsdl’ use **Result** for the response. Both the input and output parameter are mapped to SOAP **xsd:int** type.

Note that the SOAP address generated by default (<http://.../>) must be edited manually or specified using ‘ada2wsdl’'s -a option.

This is of course a very simple example. `ada2wsdl` does support lot more complex specs and will map Ada records, arrays, enumerations, derived types to a corresponding XML schema definition. See section below for a description of the mapping.

6.1.2 Ada mapping to WSDL

`ada2wsdl` parse Ada records, arrays, derived types, enumerations, procedures and functions and generate the corresponding WSDL document. In this section we describe the mapping between Ada and WSDL.

Integer Mapped to `xsd:int`.

Float Mapped to `xsd:float`.

Long_Float
Mapped to `xsd:double`

Long_Long_Float
Mapped to `xsd:double`, not supported by SOAP, mapped for convenience but precision cannot be guaranteed.

Boolean Mapped to `xsd:boolean`

String Mapped to `xsd:string`

Unbounded_String
Mapped to `xsd:string`, note that `Unbounded_String` should be used only inside a record for full interoperability. This is a current limitation.

Character Mapped to a Character schema definition.

```
<simpleType name="Character">
  <restriction base="xsd:string">
    <length value="1"/>
  </restriction>
</simpleType>
```

SOAP.Utills.SOAP_Base64

Mapped to `xsd:base64Binary`. `SOAP.Utills.SOAP_Base64` is a subtype of string which is recognized by `ada2wsdl` to generate the proper SOAP type.

SOAP.Types.Byte

Mapped to `xsd:byte`. `SOAP.Types.Byte` is a type which is recognized by `ada2wsdl` to generate the proper SOAP type.

SOAP.Types.Short

Mapped to `xsd:short`. `SOAP.Types.Short` is a type which is recognized by `ada2wsdl` to generate the proper SOAP type.

SOAP.Types.Long

Mapped to `xsd:long`. `SOAP.Types.Long` is a type which is recognized by `ada2wsdl` to generate the proper SOAP type.

SOAP.Types.Unsigned_Byte

Mapped to `xsd:unsignedByte`. `SOAP.Types.Unsigned_Byte` is a type which is recognized by `ada2wsdl` to generate the proper SOAP type.

SOAP.Types.Unsigned_Short

Mapped to `xsd:unsignedShort`. `SOAP.Types.Unsigned_Short` is a type which is recognized by `ada2wsdl` to generate the proper SOAP type.

SOAP.Types.Unsigned_Int

Mapped to `xsd:unsignedInt`. `SOAP.Types.Unsigned_Int` is a type which is recognized by `ada2wsdl` to generate the proper SOAP type.

SOAP.Types.Unsigned_Long

Mapped to `xsd:unsignedLong`. `SOAP.Types.Unsigned_Long` is a type which is recognized by `ada2wsdl` to generate the proper SOAP type.

Derived types

Mapped to a type schema definition.

```
type Number is new Integer;
```

is defined as:

```
<simpleType name="Number"
  targetNamespace="http://soapaws/WSDL_C_pkg/">
  <restriction base="xsd:int"/>
</simpleType>
```

User's types

Mapped to a type schema definition.

```
type Small is range 1 .. 10;
```

is defined as:

```
<simpleType name="Small"
  targetNamespace="http://soapaws/WSDL_C_pkg/">
  <restriction base="xsd:byte"/>
</simpleType>
```

Enumerations

Mapped to an enumeration schema definition. For example:

```
type Color is (Red, Green, Blue);
```

is defined as:

```
<simpleType name="Color">
  <restriction base="xsd:string">
    <enumeration value="Red"/>
    <enumeration value="Green"/>
    <enumeration value="Blue"/>
  </restriction>
</simpleType>
```

Records

Mapped to a struct schema definition. For example:

```

type Rec is record
  A : Integer;
  B : Float;
  C : Long_Float;
  D : Character;
  E : Unbounded_String;
  F : Boolean;
end record;

```

is defined as:

```

<complexType name="Rec">
  <all>
    <element name="A" type="xsd:int"/>
    <element name="B" type="xsd:float"/>
    <element name="C" type="xsd:double"/>
    <element name="D" type="tns:Character"/>
    <element name="E" type="xsd:string"/>
    <element name="F" type="xsd:boolean"/>
  </all>
</complexType>

```

Arrays

Mapped to an array schema definition. For example:

```

type Set_Of_Rec is array (Positive range <>) of Rec;

```

is defined as:

```

<complexType name="Set_Of_Rec">
  <complexContent>
    <restriction base="soap-enc:Array">
      <attribute ref="soap-enc:arrayType" wsdl:arrayType="tns:Rec[]" />
    </restriction>
  </complexContent>
</complexType>

```

Array inside a record

This part is a bit delicate. A record field must be constrained but a SOAP arrays is most of the time not constrained at all. To support this AWS use a safe access array component. Such a type is built using a generic runtime support package named `SOAP.Utills.Safe_Pointers`. This package implements a reference counter for the array access and will release automatically the memory when no more reference exists for a given object.

For example, let's say that we have an array of integer that we want to put inside a record:

```

type Set_Of_Int is array (Positive range <>) of Integer;

```

The first step is to create the safe array access support:

```

type Set_Of_Int_Access is access Set_Of_Int;

package Set_Of_Int_Safe_Pointer is
  new SOAP.Utills.Safe_Pointers (Set_Of_Int, Set_Of_Int_Access);

```

Note that the name `Set_Of_Int_Safe_Pointer` (*<type>_Safe_Pointer*) is mandatory (and checked by 'ada2wsdl') to achieve interoperability with 'wsdl2aws'. see [Section 6.2 \[Working with WSDL documents\]](#), page 59.

From there the safe array access can be placed into the record:

```
type Complex_Rec is record
  SI : Set_Of_Int_Safe_Pointer.Safe_Pointer;
end record;
```

To create a `Safe_Pointer` given a `Set_Of_Int` one must use `Set_Of_Int_Safe_Pointer.To_Safe_Pointer` routine. Accessing individual items is done with `SI.Item (K)`.

These Ada definitions are fully recognized by 'ada2wsdl' and will generate standard array and record WSDL definitions as seen above.

```
<complexType name="Set_Of_Int">
  <complexContent>
    <restriction base="soap-enc:Array">
      <attribute ref="soap-enc:arrayType" wsdl:arrayType="xsd:int[]" />
    </restriction>
  </complexContent>
</complexType>

<complexType name="Complex_Rec">
  <all>
    <element name="SI" type="tns:Set_Of_Int"/>
  </all>
</complexType>
```

6.1.3 ada2wsdl

```
Usage: ada2wsdl [options] ada_spec
```

ada2wsdl options are:

- a url** Specify the URL for the Web Server address. Web Services will be available at this address. A port can be specified on the URL, `http://server[:port]`. The default value is `http://.../`.
- f** Force creation of the WSDL file. Overwrite exiting file with the same name.
- I path** Add path option for the ASIS compilation step. This option can appear any number of time on the command line.
- noenum** Do not generate WSDL representation for Ada enumerations, map them to standard string. see [Section 6.1.2 \[Ada mapping to WSDL\]](#), page 55.
- o file** Generate the WSDL document into file.
- q** Quiet mode (no output)
- s name** Specify the Web Service name for the WSDL document, by default the spec package's name is used.
- v** Verbose mode, display the parsed spec.

6.1.4 ‘ada2wsdl’ limitations

- Do not handle constraint arrays into a records.
- Unbounded_String are supported with full interoperability only inside a record.
- Only unconstraint arrays are supported
- Arrays with multiple dimentions not supported

6.2 Working with WSDL documents

6.2.1 Client side (stub)

This section describe how to use a Web Service. Let’s say that we want to use the Barnes & Noble Price Quote service. The WSDL document for this service can be found at <http://www.xmethods.net/sd/2001/BNQuoteService.wsdl>. In summary this document says that there is a service named `getPrice` taking as input a string representing the ISBN number and returning the price as floating point.

The first step is to generate the client interface (stub):

```
$ wsdl2aws -noskel http://www.xmethods.net/sd/2001/BNQuoteService.wsdl
```

This will create many files, the interesting one at this point is ‘bnquoteservice-client.ads’, inside we have:

```
function getPrice
  (isbn : in String)
  return Float;
--  Raises SOAP.SOAP_Error if the procedure fails
```

Let’s call this service to find out the price for *The Sword of Shannara Trilogy* book.

```
with Ada.Text_IO;
with BNQuoteService.Client;

procedure Price is
  use Ada;

  ISBN : constant String := "0345453751";
  --  The Sword of Shannara Trilogy ISBN

  package LFIO is new Text_IO.Float_IO (Float);

begin
  Text_IO.Put_Line ("B&N Price for The Sword of Shannara Trilogy");
  LFIO.Put (BNQuoteService.Client.getPrice (ISBN), Aft => 2, Exp => 0);
end Price;
```

That’s all is needed to use this Web Service. This program is fully functional, It is possible to build it and to run it to get the answer.

6.2.2 Server side (skeleton)

Building a Web Service can also be done from a WSDL document. Let’s say that you are Barnes & Noble and that you want to build Web Service `getPrice` as described in the previous section. You have created the WSDL document to specify the service spec. From there you can create the skeleton:

```
$ wsd12aws -nostub http://www.xmethods.net/sd/2001/BNQuoteService.wsdl
```

This will create many files, the interesting one here is `'bnquoteservice-server.ads'`, inside we have:

```
Port : constant := 80;

generic
  with function getPrice
    (isbn : in String)
    return Float;
function getPrice_CB
  (SOAPAction : in String;
   Payload    : in SOAP.Message.Payload.Object;
   Request    : in AWS.Status.Data)
  return AWS.Response.Data;
```

This is a SOAP AWS's callback routine that can be instantiated with the right routine to retrieve the price of a book given its ISBN number. A possible implementation of such routine could be:

```
function getPrice
  (isbn : in String)
  return Float is
begin
  if isbn = "0987654321" then
    return 45.0;
  elsif ...
end getPrice;

function SOAP_getPrice is new BNQuoteService.Server.getPrice_CB (getPrice);
```

`SOAP_getPrice` is a SOAP AWS's callback routine (i.e. it is not a standard callback). To use it there is different solutions:

Using `SOAP.Utils.SOAP_Wrapper`

This generic function can be used to translate a standard callback based on `AWS.Status.Data` into a SOAP callback routine.

```
function getPrice_Wrapper is new SOAP.Utils.SOAP_Wrapper (SOAP_getPrice);
```

The routine `getPrice_Wrapper` can be used as any other AWS's callback routines. Note that inside this wrapper the XML payload is parsed to check the routine name and to retrieve the SOAP parameters. To call this routine the payload needs to be parsed (we need to know which routine has been invoked). In this case we have parsed the XML payload twice, this is not efficient.

Building the wrapper yourself

This solution is more efficient if there is many SOAP procedures as the payload is parsed only once.

```

function CB (Request : in Status.Data) return Response.Data is
  SOAPAction : constant String := Status.SOAPAction (Request);
  Payload     : constant SOAP.Message.Payload.Object
    := SOAP.Message.XML.Load_Payload (AWS.Status.Payload (Request));
  Proc        : constant String
    := SOAP.Message.Payload.Procedure_Name (Payload);
begin
  if SOAPAction = "..." then
    if Proc = "getPrice" then
      return SOAP_getPrice (SOAPAction, Payload, Request);
    elsif ...
      ...
    end if;
  else
    ...
  end if;

```

Note that the port to be used by the AWS server is described into the server spec.

6.2.3 wsdl2aws

Usage: `wsdl2aws [options] <file|URL>`

It is possible to pass a WSDL file or direct 'wsdl2aws' to a WSDL document on the Web by passing it's URL.

wsdl2aws options are:

- q** Quiet mode (no output)
- d** Generate debug code. Will output some information about the payload to help debug a Web Service.
- a** Generate using Ada style names. For example `getPrice` will be converted to `Get_Price`. This formatting is done for packages, routines and formal parameters.
- f** Force creation of the file. Overwrite any exiting files with the same name.
- e** Specify the default endpoint to use instead of the one found in the WSDL document.
- s** Skip non supported SOAP routines. If **-s** is not used, `wsdl2aws` will exit with an error when a problem is found while parsing the WSDL document. This option is useful to skip routines using non supported types and still be able to compile the generated files.
- o name** Specify the name of the local WSDL document. This option can be used only when using a Web WSDL document (i.e. passing an URL to `wsdl2aws`).
- doc** Handle document style binding as RPC ones. This is sometimes needed because some WSDL document specify a document style binding even though it is really an RPC one.
- v** Verbose mode, display the parsed spec.
- v -v** Verbose mode, display the parsed spec and lot of information while parsing the WSDL document tree.
- wsdl** Add WSDL document as comment into the generated root unit.
- cvs** Add CVS d tag in every generated file.
- nostub** Do not generated stubs, only skeletons are generated.
- noskel** Do not generated skeletons, only stubs are generated.

-cb Generate a SOAP dispatcher callback routine for the server. This dispatcher routine contains the code to handle all the operations as described in the WSDL document. You need also to specify the **-spec** and/or **-types** options, see below.

-x operation

Add **operation** to the list of SOAP operations to skip during the code generation. It is possible to specify multiple **-x** options on the command line.

-spec spec Specify the name of the spec containing the Ada implementation of the SOAP routines. This is used for example by the **-cb** option above to instantiate all the server side SOAP callbacks used by the main SOAP dispatcher routine. If **-types** is not specified, the type definitions are also used from this spec.

-types spec

Specify the name of the spec containing the Ada types (record, array) used by SOAP routines specified with option **-spec**. If **-spec** is not specified, the spec definitions are also used from this spec.

-main filename

Specify the name of the server's procedure main to generate. If file '**<filename>.amt**' (Ada Main Template) is present, it uses this template file to generate the main procedure. The template can reference the following variable tags:

SOAP_SERVICE

The name of the service as described into the WSDL document. This tag can be used to include the right units

```
with @_SOAP_SERVICE_@.Client;
with @_SOAP_SERVICE_@.CB;
```

SOAP_VERSION

The AWS's SOAP version.

AWS_VERSION

The AWS's version.

UNIT_NAME

The name of the generated unit. This is the name of the procedure that will be created.

```
procedure @_UNIT_NAME_@ is
begin
  ...
```

-proxy name|IP

Use this proxy to access the WSDL document and generate code to access to these Web Services via this proxy. The proxy can be specified by its DNS name or IP address.

-pu name User name for the proxy if proxy authentication required.

-pp password

User password for the proxy if proxy authentication required.

-timeouts [timeouts | connect_timeout,send_timeout,receive_timeout]

Set the timeouts for the SOAP connection. The timeouts is either a single value used for the connect, send and receive timeouts or three values separated by a colon to set each timeout independently.

6.2.4 wsdl2aws behind the scene

The `wsdl2aws` tool read a WSDL document and creates a root package and a set of child packages as described below:

<root> This is the main package, it contains eventually the full WSDL in comment and the description of the services as read from the WSDL document.

<root>.Types

This package contains the definitions of the types which are not SOAP base types. We find here the definitions of the SOAP structs and arrays with routines to convert them between the Ada and SOAP type model. A subtype definition is also created for every routine's returned type. In fact, all definitions here are only alias or renaming of types and/or routines generated in other packages. The real definitions for structs, arrays, enumerations and derived types are generated into a package whose name depends on the name space used for these entities. This package act as a container for all definitions and it is the only one used in the other generated packages.

<root>.Client

All spec to call Web Services.

<root>.Server

All spec to build Web Services. These specs are all generic and must be instantiated with the right routine to create the web services.

<root>.CB The SOAP dispatcher callback routine.

6.2.5 wsdl2aws limitations

It is hard to know all current limitations as the WSDL and SOAP world is quite complex. We list there all known limitations:

- Some SOAP base types are not supported : *date*, *time*, *xsd:hexBinary*, *decimal*. All these are easy to add (except decimal), it is just not supported with the current version.
- Multi-dimension arrays are not supported.
- abstract types are not supported.
- SOAP MIME attachments are not supported.
- WSDL type inheritance not supported.
- Only the RPC/Encoded SOAP messages' style is supported (the Document/Literal is not)

6.3 Using ada2wsdl and wsdl2aws together

Using both tools together is an effective way to build rapidly a SOAP server. It can be said that doing so is quite trivial in fact. Let's take the following spec:

```
package Graphics is

  type Point is record
    X, Y : Float;
  end record;

  function Distance (P1, P2 : in Point) return Float;
  -- Returns the distance between points P1 and P2

end Graphics;
```

We do not show the body here but we suppose it is implemented. To build a server for this service it is as easy as:

```
$ ada2wsdl -a http://localhost:8787 -o graphics.wsdl graphics.ads
```

The server will be available on localhost at port 8787.

```
$ wsdl2aws -cb -main server -types graphics graphics.wsdl
$ gnatmake server -larges ...
```

Options

- `-cb` is to create the SOAP dispatcher callback routine,
- `-main server` to generate the main server procedure in `'server.adb'`,
- `-types graphics` to use `'graphics.ads'` to get references from user's spec (reference to `Graphics.Point` for example).

7 Working with mails

7.1 Sending e-mail

AWS provides a complete API to send e-mail using SMTP protocol. You need to have access to an SMTP server to use this feature. The API covers sending simple mail with text message and/or with MIME attachments (base64 encoded). Here are the steps to send a simple e-mail:

- Initialize the SMTP server

```
SMTP_Server : SMTP.Receiver
:= SMTP.Client.Initialize ("smtp.hostname");
```

Here AWS uses the default SMTP port to create an SMTP mail server but it is possible to specify a different one. The hostname specified must be a valid SMTP server.

- Send the e-mail

To send an e-mail there is many different API. Let's send a simple text mail.

```
Status : SMTP.Status;

SMTP.Client.Send
(SMTP_Server,
 From    => SMTP.E_Mail ("Pascal Obry", "p.obry@wanadoo.fr"),
 To      => SMTP.E_Mail ("John Doe", "john.doe@here.com"),
 Subject => "About AWS SMTP protocol",
 Message => "AWS can now send mails",
 Status  => Status);
```

Here Status will contain the SMTP returned status.

- Check that everything is ok

Using above status data it is possible to check that the message was sent or not by the server. The status contain a code and an error message, both of them can be retrieved using specific routines, See [Section B.67 \[AWS.SMTP\], page 149](#). It is also possible to check that the call was successful with `SMTP.Is_Ok` routine.

```
if not SMTP.Is_Ok (Status) then
  Put_Line ("Can't send message: " & SMTP.Status_Message (Status));
end if;
```

In the above example, the message content was given as a string but it is possible to specify a disk file. AWS can also send MIME messages either from disk files or with in memory base64 encoded binary data. The API provides also a way to send messages to multiple recipients at the same time and to send messages with alternative contents (text and HTML for example). These features are not described here, complete documentation can be found on the spec see [Section B.67 \[AWS.SMTP\], page 149](#) and see [Section B.68 \[AWS.SMTP.Client\], page 150](#).

7.2 Retrieving e-mail

AWS provides an API to retrieve e-mails from a POP mailbox. POP stands for *Post Office Protocol* and is the main protocol used by Internet Service Providers around the world. IMAP is another well known protocol in this area but it is not supported by AWS.

We describes here the POP API. For a complete description see see [Section B.31 \[AWS.POP\], page 113](#).

- Opening the mailbox

The first step is to authenticate using a user name and password. AWS supports two methods one called `Clear_Text` which is the most used and another one `APOP` which is more secure but almost not supported by ISP for the moment (and will probably never be supported as a more secure protocol named `SPA` -Secure Password Authentication- could be used instead).

```
Mailbox : POP.Mailbox
:= POP.Initialize ("pop.hostname", "john.does", "mysuperpwd");
```

The default Authentication method is `Clear_Text`.

- Getting mailbox information

When the connection is opened it is possible to get information about the mailbox like the number of messages or the total number of bytes in the mailbox.

```
N      : constant Natural := POP.Message_Count (Mailbox);

Bytes : constant Natural := POP.Size (Mailbox);
```

- Retrieving individual e-mail

Each message is numbered starting from 1. A function named `Get` will return a message given its mailbox's number.

```
Message : constant POP.Message := POP.Get (Mailbox, 2, Remove => True);
```

`Remove` can be set to `False` for the message to stay on the mailbox. The default value is `False`.

- Iterating through the mailbox content

Another way to retrieve message is by using an iterator.

```
procedure Print_Subject
(Message : in POP.Message
 Index   : in Positive;
 Quit    : in out Boolean) is
begin
  Text_IO.Put_Line (POP.Subject (Message));
end Print_Message;

procedure Print_All_Subjects is new POP.For_Every_Message (Print_Subject);

...

Print_All_Subjects (Mailbox, Remove => True);
```

It exists a set of routines on a `POP.Message` object to get the subject the content, the date or any headers. It is also possible to work with attachments. See point below.

- Working with attachments

A message can have a set of MIME attachments. The number of attachments can be retrieved using `Attachment_Count`.

```
Message : constant POP.Message := ...;

A_Count : constant Natural := POP.Attachment_Count (Message);
```


As for messages it is possible to get a single attachment using its index in the message or by using an iterator.

```
First_Attachment : constant POP.Attachment := POP.Get (Message, 1);

procedure Write_Attachment
  (Attachment : in     POP.Attachment
   Index       : in     Positive;
   Quit        : in out Boolean) is
begin
  POP.Write (Attachment, Directory => ".");
end Print_Message;

procedure Write_All_Attachments is new POP.For_Every_Attachment (Write_Attachment);

...

Write_All_Attachments (Message);
```

It is also possible to retrieve the attachment's filename, the content as a memory stream. see [Section B.31 \[AWS.POP\]](#), page 113.

- Closing the connection

```
POP.Close (POP_Server);
```


8 LDAP

AWS provides a complete API to retrieve information from LDAP servers. Note that there is no support for updating, modifying or deleting information only to read information from the server.

The AWS/LDAP implementation is based on `OpenLDAP`. To build an LDAP application you need to link with the `'libldap.a'` library. This library is built by AWS on Windows based system and will use the `'wldap32.dll'` as provided with Windows NT/2000/XP. On UNIX based systems, you must install properly the `OpenLDAP` package.

The steps required to read information from an LDAP server are:

Initialize the LDAP directory

```
declare
  Directory : LDAP.Client.Directory;
begin
  Directory := LDAP.Client.Init (Host);
```

Host is the hostname where the LDAP directory is running. It is possible to specify the port if the LDAP server does not use the default one.

Bind to the LDAP server

This step is the way to pass a login/password if the LDAP server required an authentication. If not, the login/password must be empty strings.

```
LDAP.Client.Bind (Directory, "", "");
```

Do the search

For the search you must specify the base name, a filter, the scope and a set of attributes to retrieve.

```
Response_Set := LDAP.Client.Search
(Directory, Base_DN, Filter, LDAP.Client.LDAP_Scope_Subtree,
 LDAP.Client.Attributes
 ("cn", "sn", "telephonenumber"));
```

Attributes The set of attributes to retrieve from the directory.

Filter A set of values for some attributes. A filter is `<attribute_name>=<value>` where value can contain `'*'` at the end. For example `"(cn=DUPON*)"` will look for all entries where the common name is starting by the string `"DUPON"`.

Scope Define how far in the hierarchical directory the search will operate. It is either one level, all subtrees or on the base of the tree.

For more information see [Section B.20 \[AWS.LDAP.Client\]](#), page 102.

Iterate through the response set

For this there is two iterators. `First_Entry/Next_Entry` or the generic high level iterator `For_Every_Entry`.

```

declare
  Message : LDAP.Client.LDAP_Message;
begin
  Message := LDAP.Client.First_Entry (Directory, Response_Set);

  while Message /= LDAP.Client.Null_LDAP_Message loop

    Do_Job (Message);

    Message := LDAP.Client.Next_Entry (Directory, Message);
  end loop;
end;

```

Read attributes for each entry

Each entry has an associated set of attributes. To retrieve attributes values there is two iterators. `First_Attribute` / `Next_Attribute` or the generic high level iterator `For_Every_Attribute`.

```

declare
  BER : aliased LDAP.Client.BER_Element;

  Attr : constant String := LDAP.Client.First_Attribute
    (Directory, Message, BER'Unchecked_Access);
begin
  Do_Job (Attr);

  loop
    declare
      Attr : constant String := LDAP.Client.Next_Attribute
        (Directory, Message, BER);
    begin
      exit when Attr = "";
      Do_Job (Attr);
    end;
  end loop;
end;

```

Cleanup

At the end of the processing it is important to release memory associated with LDAP objects.

```

LDAP.Client.Free (Message);
LDAP.Client.Unbind (Directory);

```

see [Section B.20 \[AWS.LDAP.Client\]](#), page 102 for all high level supported API and documentation.

Note that for complete information about AWS/LDAP you should read an LDAP API description. AWS/LDAP is only a binding and follow the LDAP API closely.

9 Jabber

AWS support part of the Jabber protocol. At this stage only two kind of messages are supported:

1. Presence

To check the presence status of a specific JID (Jabber ID)

2. Message

To send messages to a specific JID (Jabber ID)

Note that if you want an application to check the presence or send message to users it is recommended to create a specific Jabber ID on the server for this application and ask users to accept this specific user to check their presence status.

9.1 Jabber presence

To check for the presence of another JID you must first have the right to do so. The jabber server won't let you see presence of another JID unless the JID have permitted you to see its presence.

- First declare the server and status objects

```
Server : AWS.Jabber.Server;  
Status : AWS.Jabber.Presence_Status;
```

- Connect to the server, you must have an account created and must know the login and password

```
AWS.Jabber.Connect  
(Server, "jabber.domain.org", "joe", "mysuperpwd");
```

- Then, to check the presence of user "john"

```
AWS.Jabber.Check_Presence  
(Server, "john@jabber.domain.org", Status);
```

- Then, you just have to close the server

```
AWS.Jabber.Close (Server);
```

9.2 Jabber message

To send a message to a specific JID, you must connect to the server as above and close the server when you don't need to communicate with it anymore. The only different part is to send the message, here is an example:

```
Send_Message  
(Server,  
  JID    => "john@jabber.domain.org",  
  Subject => "Hello there!",  
  Content => "Are you using AWS ?");
```

It is as simple as that !

10 Resources

AWS support embedded resources. It means that it is possible to build a fully self dependent executable. This is useful when distributing a server. The server program contains the code but also the images (PNG, JPEG, GIF), the templates, the HTML pages... more generally any file the Web Server must serve to clients.

10.1 Building resources

To embed the files into the executable you must build a resource tree. This task is greatly simplified using ‘AWSRes’ tool. For example let’s say that you want to build a simple server with a single page containing some text and one PNG image. The text is handled directly in the callback procedure and contain a reference to the image ‘logo.png’. To build the resource tree:

```
$ awsres logo.png
```

This will create a set of packages whose root is the unit **res** by default. The resource tree is created. see [Section 10.4 \[awsres tool\]](#), page 73 for the complete AWS’s usage description.

awsres can also compress the resource files. This can be done by using **awsres**’s **-z** option. Compressed resources are handled transparently. If the Web client supports compression the resource is sent as-is otherwise a decompression stream will be created for the resource to be decompressed on-the-fly while sending it.

10.2 Using resources

This is really the simplest step. The resource tree must be linked with your executable, to do so you just have to “with” the resource tree root into one of your program unit. This will ensure that the resource tree will be compiled and linked into the executable. **AWS** and **Templates_Parser** know about resource files and will pick them up if available.

Note that this is transparent to users. It is possible to build the very same server based on standard files or resources files. The only change in the code is to “with” or not the resource tree.

Note that **AWS** supports only a single resource tree. If more than one resource tree is included into a program only one will be seen.

10.3 Stream resources

Users can build a response directly from a stream. In this case the callback answer is built using **AWS.Response.Stream**. It creates a resource object whose operations have been inherited from **AWS.Resource.Stream.Stream_Type** and redefined by the user. So the **Read** operation can dynamically create the result stream data, the **End_Of_File** operation must returns **True** when the stream data is out and so on. This feature is useful to let users completely create and control dynamically AWS’s response content.

see [Section B.35 \[AWS.Resources.Streams\]](#), page 117.

10.4 awsres tool

AWSRes is a tool to build resource files. It creates a root package named ‘**res**’ by default and a child package for each resource file.

Usage: **awsres** [-hrquz] file1 [-uz] [file2...]

-h Display help message.

- `-q` Quiet mode.
- `-r name` Set the root unit name. Default is **res**.
- `-u` Add following files as uncompressed resources.
- `-z` Add following files as compressed resources.

11 Status page

The status page gives information about the AWS internal status. For example it returns the server socket ID, the number of simultaneous connection, the number of time a connection has been used...

To display the information AWS use a template file. The template file (default is 'aws_status.thtml') is an HTML file with some specific tags recognized by the parser. For more information about how the template parser works, please look for the template parser documentation distributed with AWS.

Here are the tag variables recognized by AWS status page:

ABORTABLE_V (vector tag)

A list of boolean. One for each connection. True means that this connection can be aborted if none is available. This is to be inserted in a template table.

ACCEPT_QUEUE_SIZE

see [Section 3.4 \[Configuration options\]](#), page 15.

ACCEPTOR_LENGTH

Number of sockets in the internal socket set.

ACTIVITY_COUNTER_V (vector tag)

A list of natural. One for each connection. This is the number of request the connection has answered. This counter is reset each time the connection is closed. In other word this is the number of request a keep-alive connection has processed.

ACTIVITY_TIME_STAMP_V (vector tag)

A list of date. One for each connection. This is the time of the latest request answered.

ADMIN

URI to the administrative page.

CASE_SENSITIVE_PARAMETERS

see [Section 3.4 \[Configuration options\]](#), page 15.

CHECK_URL_VALIDITY

see [Section 3.4 \[Configuration options\]](#), page 15.

CLEANER_CLIENT_DATA_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

CLEANER_CLIENT_HEADER_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

CLEANER_SERVER_RESPONSE_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

CLEANER_WAIT_FOR_CLIENT_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

CURRENT_CONNECTIONS

Number of current connections to the server.

ERROR_LOG (boolean tag)

This is set to true if error logging is active.

ERROR_LOG_FILE

The error log file full pathname.

ERROR_LOG_FILENAME_PREFIX

see [Section 3.4 \[Configuration options\]](#), page 15.

ERROR_LOG_SPLIT_MODE

see [Section 3.4 \[Configuration options\]](#), page 15.

FORCE_CLIENT_DATA_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

FORCE_CLIENT_HEADER_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

FORCE_SERVER_RESPONSE_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

FORCE_WAIT_FOR_CLIENT_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

FREE_SLOTS_KEEP_ALIVE_LIMIT

see [Section 3.4 \[Configuration options\]](#), page 15.

LINE_STACK_SIZE

see [Section 3.4 \[Configuration options\]](#), page 15.

KEYS_M (matrix tag)

A list of set of keys (for each key correspond a value in the tag `VALUES_L`, see below). Each key in the vector tag start with an HTML "<td>" tag. This is to be able to display the key/value in column.

LOG (boolean tag)

This is set to true if logging is active.

LOG_FILE

The log file full pathname.

LOG_FILENAME_PREFIX

see [Section 3.4 \[Configuration options\]](#), page 15.

LOG_FILE_DIRECTORY

see [Section 3.4 \[Configuration options\]](#), page 15.

LOG_MODE

The rotating log mode, this is either `NONE`, `DAILY`, `MONTHLY` or `EACH_RUN`.

LOGO

A string to be placed in an img HTML tag. This is the name of the AWS logo image.

MAX_CONCURRENT_DOWNLOAD

see [Section 3.4 \[Configuration options\]](#), page 15.

MAX_CONNECTION

see [Section 3.4 \[Configuration options\]](#), page 15.

PEER_NAME_V (vector tag)

A list of peer name. One for each connection. This is the name of the last peer connected to the slot.

PHASE_V (vector tag)

What is the slot currently doing, for example `Server.Processing` or `Closed`.

RECEIVE_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

REUSE_ADDRESS

see [Section 3.4 \[Configuration options\]](#), page 15.

SECURITY

A boolean set to True if this is a secure socket (HTTPS/SSL).

SECURITY_MODE

see [Section 3.4 \[Configuration options\]](#), page 15.

SEND_TIMEOUT

see [Section 3.4 \[Configuration options\]](#), page 15.

SERVER_HOST

see [Section 3.4 \[Configuration options\]](#), page 15.

SERVER_NAME

see [Section 3.4 \[Configuration options\]](#), page 15.

SERVER_PORT

see [Section 3.4 \[Configuration options\]](#), page 15.

SERVER SOCK

Server socket ID.

SESSION

see [Section 3.4 \[Configuration options\]](#), page 15.

SESSION_CLEANUP_INTERVAL

Number of seconds between each run of the session cleanup task. This task will remove all session data that have been obsoleted.

SESSION_LIFETIME

Number of seconds to keep session information. After this period a session is obsoleted and will be removed at next cleanup.

SESSION_NAME

see [Section 3.4 \[Configuration options\]](#), page 15.

SESSIONS_TERMINATE_V (vector tag)

A list of time. Each item correspond to the time when the session will be obsoleted.

SESSIONS_TS_V (vector tag)

A list of time stamp. Each item correspond to a session last access time.

SESSIONS_V (vector tag)

A list of session ID.

SLOT_ACTIVITY_COUNTER_V (vector tag)

A list of natural. One for each connection. This is the total number of requests the slot has answered. This counter is never reseted.

SOCK_V (vector tag)

A list of sockets ID. One for each connection.

STATUS_PAGE

see [Section 3.4 \[Configuration options\]](#), page 15.

START_TIME

A timestamp in YYYY-MM-DD HH:MM:SS format. When the server was started.

TRANSIENT_CLEANUP_INTERVAL

see [Section 3.4 \[Configuration options\]](#), page 15.

TRANSIENT_LIFETIME

see [Section 3.4 \[Configuration options\]](#), page 15.

UPLOAD_DIRECTORY

see [Section 3.4 \[Configuration options\]](#), page 15.

UPLOAD_SIZE_LIMIT

see [Section 3.4 \[Configuration options\]](#), page 15.

VALUES_M (matrix tag)

A list of set of values (for each value correspond a key in the vector tag KEYS_L, see above). Each key in the vector tag start with an HTML "<td>" tag. This is to be able to display the key/value in column.

VERSION

AWS version string.

WWW_ROOT

see [Section 3.4 \[Configuration options\]](#), page 15.

There is also all `Templates_Parser` specific tags. This is not listed here please have a look at the `Templates_Parser` documentation distributed with AWS.

Appendix A References

Here is a list of documents used to implement AWS, the SOAP support and associated services:

RFC 0821

SIMPLE MAIL TRANSFER PROTOCOL

Jonathan B. Postel
August 1982

Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, California 90291

RFC 1867

Network Working Group
Request For Comments: 1867
Category: Experimental

E. Nebel
L. Masinter
Xerox Corporation
November 1995

Form-based File Upload in HTML

RFC 1939

Network Working Group
Request for Comments: 1939
STD: 53
Obsoletes: 1725
Category: Standards Track

J. Myers
Carnegie Mellon
M. Rose
Dover Beach Consulting, Inc.
May 1996

Post Office Protocol - Version 3

RFC 1945

Network Working Group
Request for Comments: 1945
Category: Informational

T. Berners-Lee
MIT/LCS
R. Fielding
UC Irvine
H. Frystyk
MIT/LCS
May 1996

Hypertext Transfer Protocol -- HTTP/1.0

RFC 2049

Network Working Group
Request for Comments: 2049
Obsoletes: 1521, 1522, 1590
Category: Standards Track

N. Freed
Innosoft
N. Borenstein
First Virtual
November 1996

Multipurpose Internet Mail Extensions
(MIME) Part Five:
Conformance Criteria and Examples

RFC 2109

Network Working Group
Request for Comments: 2109
Category: Standards Track

D. Kristol
Bell Laboratories, Lucent Technologies
L. Montulli
Netscape Communications
February 1997

HTTP State Management Mechanism

RFC 2195

Network Working Group
 Request for Comments: 2195
 Category: Standards Track
 Obsoletes: 2095

J. Klensin
 R. Catoe
 P. Krumviede
 MCI
 September 1997

IMAP/POP AUTHorize Extension for Simple Challenge/Response

RFC 2554

Network Working Group
 Request for Comments: 2554
 Category: Standards Track

J. Myers
 Netscape Communications
 March 1999

SMTP Service Extension
for Authentication**RFC 2616**

Network Working Group
 Request for Comments: 2616
 Obsoletes: 2068
 Category: Standards Track

R. Fielding
 UC Irvine
 J. Gettys
 Compaq/W3C
 J. Mogul
 Compaq
 H. Frystyk
 W3C/MIT
 L. Masinter
 Xerox
 P. Leach
 Microsoft
 T. Berners-Lee
 W3C/MIT
 June 1999

Hypertext Transfer Protocol -- HTTP/1.1

RFC 2617

Network Working Group
 Request for Comments: 2617
 Obsoletes: 2069
 Category: Standards Track

J. Franks
 Northwestern University
 P. Hallam-Baker
 Verisign, Inc.
 J. Hostetler
 AbiSource, Inc.
 S. Lawrence
 Agranat Systems, Inc.
 P. Leach
 Microsoft Corporation
 A. Luotonen
 Netscape Communications Corporation
 L. Stewart
 Open Market, Inc.
 June 1999

HTTP Authentication: Basic and Digest Access Authentication

Transport Layer Security Working Group
INTERNET-DRAFT
Expire in six months

Alan O. Freier
Netscape Communications
Philip Karlton
Netscape Communications
Paul C. Kocher
Independent Consultant
November 18, 1996

The SSL Protocol
Version 3.0

SOAP (W3C Note 08 May 2000)

Simple Object Access Protocol (SOAP) 1.1

W3C Note 08 May 2000

This version:

<http://www.w3.org/TR/2000/NOTE-SOAP-20000508>

Latest version:

<http://www.w3.org/TR/SOAP>

Authors (alphabetically):

Don Box, DevelopMentor
David Ehnebuske, IBM
Gopal Kakivaya, Microsoft
Andrew Layman, Microsoft
Noah Mendelsohn, Lotus Development Corp.
Henrik Frystyk Nielsen, Microsoft
Satish Thatte, Microsoft
Dave Winer, UserLand Software, Inc.

Copyright 2000 DevelopMentor, International Business Machines Corporation,
Lotus Development Corporation, Microsoft, UserLand Software

<http://www.w3.org/TR/SOAP/>

A Busy Developer's Guide to SOAP 1.1

By Dave Winer, Jake Savin, UserLand Software, 4/2/01.

<http://www.soapware.org/bdg>

Appendix B AWS API Reference

B.1 AWS

B.2 AWS.Attachments

B.3 AWS.Client

B.4 AWS.Client.Hotplug

B.5 AWS.Communication

B.6 AWS.Communication.Client

B.7 AWS.Communication.Server

B.8 AWS.Config

B.9 AWS.Config.Ini

B.10 AWS.Config.Set

B.11 AWS.Containers.Tables

B.12 AWS.Cookie

B.13 AWS.Default

B.14 AWS.Dispatchers

B.15 AWS.Dispatchers.Callback

B.16 AWS.Exceptions

B.17 AWS.Headers

B.18 AWS.Headers.Values

B.19 AWS.Jabber

B.20 AWS.LDAP.Client

B.21 AWS.Log

B.22 AWS.Messages

B.23 AWS.MIME

B.24 AWS.Net

B.25 AWS.Net.Buffered

B.26 AWS.Net.Log

B.27 AWS.Net.Log.Callbacks

B.28 AWS.Net.SSL

B.29 AWS.Net.SSL.Certificate

B.30 AWS.Parameters

B.31 AWS.POP

B.32 AWS.Resources

B.33 AWS.Resources.Embedded

B.34 AWS.Resources.Files

B.35 AWS.Resources.Streams

B.36 AWS.Resources.Streams.Disk

B.37 AWS.Resources.Streams.Disk.Once

B.38 AWS.Resources.Streams.Memory

B.39 AWS.Resources.Streams.Memory.ZLib

B.40 AWS.Resources.Streams.Pipe

B.41 AWS.Response

B.42 AWS.Server

B.43 AWS.Server.Hotplug

B.44 AWS.Server.Log

B.45 AWS.Server.Push

B.46 AWS.Server.Status

B.47 AWS.Services.Callbacks

B.48 AWS.Services.Directory

B.49 AWS.Services.Dispatchers

B.50 AWS.Services.Dispatchers.Linker

B.51 AWS.Services.Dispatchers.Method

B.52 AWS.Services.Dispatchers.URI

B.53 AWS.Services.Dispatchers.Virtual_Host

B.54 AWS.Services.Download

B.55 AWS.Services.Page_Server

B.56 AWS.Services.Split_Pages

B.57 AWS.Services.Split_Pages.Alpha

B.58 AWS.Services.Split_Pages.Alpha.Bounded

B.59 AWS.Services.Split_Pages.Uniform

B.60 AWS.Services.Split_Pages.Uniform.Alpha

B.61 AWS.Services.Split_Pages.Uniform.Overlapping

B.62 AWS.Services.Transient_Pages

B.63 AWS.Services.Web_Block

B.64 AWS.Services.Web_Block.Context

B.65 AWS.Services.Web_Block.Registry

B.66 AWS.Session

B.67 AWS.SMTP

B.68 AWS.SMTP.Client

B.69 AWS.Status

B.70 AWS.Templates

B.71 AWS.Translator

B.72 AWS.URL

B.73 SOAP

B.74 SOAP.Client

B.75 SOAP.Dispatchers

B.76 SOAP.Dispatchers.Callback

B.77 SOAP.Message

B.78 SOAP.Message.XML

B.79 SOAP.Parameters

B.80 SOAP.Types

Index

A

ABORTABLE_V	75
Accept_Queue_Size	15
ACCEPT_QUEUE_SIZE	75
ACCEPTOR_LENGTH	75
ACTIVITY_COUNTER_V	75
ACTIVITY_TIME_STAMP_V	75
ada2wsdl	53
ada2wsdl limitations	59
ADMIN	75
Admin_Password	16
Admin_URI	9, 16
Ajax	38
authentication	21
AWS	83
AWS.Attachments	84
AWS.Client	85
AWS.Client.Hotplug	86
AWS.Communication	87
AWS.Communication.Client	88
AWS.Communication.Server	89
AWS.Config	90
AWS.Config.Ini	91
AWS.Config.Set	92
AWS.Containers.Tables	93
AWS.Cookie	94
AWS.Default	95
AWS.Dispatchers	96
AWS.Dispatchers.Callback	97
AWS.Exceptions	98
AWS.Headers	99
AWS.Headers.Values	100
aws.ini	15
AWS.Jabber	101
AWS.LDAP.Client	102
AWS.Log	103
AWS.Messages	104
AWS.MIME	105
AWS.Net	106
AWS.Net.Buffered	107
AWS.Net.Log	108
AWS.Net.Log.Callbacks	109
AWS.Net.SSL	110
AWS.Net.SSL.Certificate	111
AWS.Net.Std	3
AWS.Parameters	112
AWS.POP	113
AWS.Resources	114
AWS.Resources.Embedded	115
AWS.Resources.Files	116
AWS.Resources.Streams	117
AWS.Resources.Streams.Disk	118
AWS.Resources.Streams.Disk.Once	119
AWS.Resources.Streams.Memory	120
AWS.Resources.Streams.Memory.ZLib	121
AWS.Resources.Streams.Pipe	122
AWS.Response	123
AWS.Server	124
AWS.Server.Hotplug	125
AWS.Server.Log	126
AWS.Server.Push	127

AWS.Server.Status	128
AWS.Services.Callbacks	129
AWS.Services.Directory	130
AWS.Services.Dispatchers	131
AWS.Services.Dispatchers.Linker	132
AWS.Services.Dispatchers.Method	133
AWS.Services.Dispatchers.URI	134
AWS.Services.Dispatchers.Virtual_Host	135
AWS.Services.Download	136
AWS.Services.Page_Server	137
AWS.Services.Split_Pages	138
AWS.Services.Split_Pages.Alpha	139
AWS.Services.Split_Pages.Alpha.Bounded	140
AWS.Services.Split_Pages.Alpha.Uniform	141
AWS.Services.Split_Pages.Alpha.Uniform.Alpha	142
AWS.Services.Split_Pages.Alpha.Uniform.Overlapping	143
AWS.Services.Transient_Pages	144
AWS.Services.Web_Block	145
AWS.Services.Web_Block.Context	146
AWS.Services.Web_Block.Registry	147
AWS.Session	148
AWS.SMTP	149
AWS.SMTP.Client	150
AWS.Status	151
AWS.Templates	152
AWS.Translator	153
AWS.URL	154
aws_action_clear.tjs	38
aws_action_replace.tjs	38
awsres	73

B

basic	21
Building	3, 4
Building resources	73

C

Callback	8, 10
Callback procedure	10
Callback, dispatcher	33
Callback, dispatcher API	97
Case_Sensitive_Parameters	9, 16
CASE_SENSITIVE_PARAMETERS	75
certificate	28
Certificate (string)	16
Check_URL_Validity	16
CHECK_URL_VALIDITY	75
Cleaner_Client_Data_Timeout	16
CLEANER_CLIENT_DATA_TIMEOUT	75
Cleaner_Client_Header_Timeout	16
CLEANER_CLIENT_HEADER_TIMEOUT	75
Cleaner_Server_Response_Timeout	16
CLEANER_SERVER_RESPONSE_TIMEOUT	75
Cleaner_Wait_For_Client_Timeout	16
CLEANER_WAIT_FOR_CLIENT_TIMEOUT	75
client HTTP	30
Client protocol	30

Communication	22
Communication, Client	23
Communication, Server	23
Configuration options	15
Cookies	20
cross-platforms	4
CURRENT_CONNECTIONS	75

D

digest	21
Directory browser	33
Directory_Browser_Page	16
Dispatchers	33
Dispatchers callback	33
Dispatchers Linker	34
Dispatchers method	33
Dispatchers SOAP	34
Dispatchers Timer	34
Dispatchers Transient pages	34
Dispatchers URI	33
Dispatchers virtual host	33
Distributing	12
Down_Image	16
Download Manager	36
draft 302	80

E

ERROR_LOG	75
ERROR_LOG_FILE	75
ERROR_LOG_FILENAME_PREFIX	76
ERROR_LOG_SPLIT_MODE	76
exception handler	29
Exceptions, Unexpected exceptions, Exceptions handler	98
Exchange_Certificate	17

F

File upload	22
Force_Client_Data_Timeout	17
FORCE_CLIENT_DATA_TIMEOUT	76
Force_Client_Header_Timeout	17
FORCE_CLIENT_HEADER_TIMEOUT	76
Force_Server_Response_Timeout	17
FORCE_SERVER_RESPONSE_TIMEOUT	76
Force_Wait_For_Client_Timeout	17
FORCE_WAIT_FOR_CLIENT_TIMEOUT	76
Form parameters	11
Free_Slots_Keep_Alive_Limit	17
FREE_SLOTS_KEEP_ALIVE_LIMIT	76

G

GNAT	3
GNU/Ada	3

H

Hello world	10
hotplug	23
Hotplug_Port	17
HTML File Upload	79
HTTP Authentication	80
HTTP declaration	8
HTTP state	20
HTTP/1.0	79
HTTP/1.1	80
HTTPS	28

I

IMAP/POP	80
ini file	15
Installing	6

J

Jabber	71
Jabber Binding	101
Jabber message	71
Jabber presence	71

K

Key	17
KEYS_M	76

L

LDAP	69
LDAP Binding	102
LDAP Directory	69
libcrypto.a	4
libssl.a	4
Lightweight Directory Access Protocol	69
Line_Stack_Size	17
LINE_STACK_SIZE	76
linker, dispatcher	34
LOG	76
Log.Flush	26
Log.Start	26
Log.Start_Error	26
Log.Stop	26
Log.Stop_Error	26
Log_Extended_Fields	17
LOG_FILE	76
Log_File_Directory	17
LOG_FILE_DIRECTORY	76
Log_Filename_Prefix	16, 17
LOG_FILENAME_PREFIX	76
LOG_MODE	76
Log_Split_Mode	16, 17
LOGO	76
Logo_Image	18
logs	26

M

MAX_CONCURRENT_DOWNLOAD	76
Max_Connection	9, 18
MAX_CONNECTION	76
method, dispatcher	33
MIME	79

O

OpenLDAP	3
OpenSSL	3

P

Page server	10, 35
pages, split	36
pages, transient	35
Parameters	11
Parameters Get	11
Parameters Get_Name	11
Payload	50
PEER_NAME_V	76
PHASE_V	76
POP	65, 79
Port	9
Post Office Protocol	65
program_name.ini	15
Push	25

R

Receive_Timeout	18
RECEIVE_TIMEOUT	76
References	79
resources	10
Resources	73
Retrieving e-mail	65
Reuse_Address	18
REUSE_ADDRESS	77
RFC 0821	79
RFC 1867	79
RFC 1939	79
RFC 1945	79
RFC 2049	79
RFC 2109	79
RFC 2195	80
RFC 2554	80
RFC 2616	80
RFC 2617	80

S

Secure server	28
Security	9
SECURITY	77
Security_Mode	18
SECURITY_MODE	77
Self dependant	73
Send	26
Send_Timeout	18
SEND_TIMEOUT	77
Send_To	26
Sending e-mail	65
Sending message	22

Server Push	25
Server_Host	18
SERVER_HOST	77
Server_Name	18
SERVER_NAME	77
Server_Port	18
SERVER_PORT	77
SERVER SOCK	77
Session	9, 18, 19
SESSION	77
SESSION_CLEANUP_INTERVAL	77
Session.Cleanup.Interval (duration)	18
SESSION_LIFETIME	77
Session.Lifetime (duration)	18
Session_Name	18
SESSION_NAME	77
SESSIONS_TERMINATE_V	77
SESSIONS_TS_V	77
SESSIONS_V	77
Simple Mail Transfer Protocol	65
Simple Object Access Protocol	49
Simple Page server	35
Simple server	10
SLOT_ACTIVITY_COUNTER_V	77
SMTP	65, 79
SMTP Authentication	80
SOAP	49
SOAP (API)	155
SOAP 1.1	81
SOAP Client	49
SOAP Dispatcher	52
SOAP Server	50
SOAP, dispatcher	34
SOAP.Client	156
SOAP.Dispatchers	157
SOAP.Dispatchers.Callback	52, 158
SOAP.Message	159
SOAP.Message.XML	160
SOAP.Parameters	161
SOAP.Types	162
SOAPAction	50
SOCK_V	77
Socket log	30
split pages	36
SSL	28, 80
START_TIME	77
starting server	8
Static Page server	35
Status	75
Status_Page	19
STATUS_PAGE	77
Stream resources	73

T

timer, dispatcher	34
TLS	28
transient pages	35
transient pages, dispatcher	34
Transient.Cleanup.Interval	19
TRANSIENT_CLEANUP_INTERVAL	77
Transient.Lifetime	19
TRANSIENT_LIFETIME	78

U

Up_Image	19
upload, client	31
upload, server	22
Upload_Directory	19
UPLOAD_DIRECTORY	78
UPLOAD_SIZE_LIMIT	78
URI, dispatcher	33
Using resources	73
Utils.SOAP_Wrapper	51

V

VALUES_M	78
VERSION	78
virtual host, dispatcher	33

W

we_icons	38
we_js	38
Web Block Context	45
web cross-references	47
Web Elements	38
Web Service Definition Language	49, 53
Web_Server	8
webxref	47
Working with Server sockets	26
WSDL	49, 53
WSDL, Client	59
WSDL, Server	59
wsdl2aws	53, 61
wsdl2aws limitations	63
WWW_Root	19
WWW_ROOT	78