Yolk Manual

Revised July 27th. 2011

Contents

Part I
General Information

1 What is Yolk?

Yolk is a collection of packages that aim to help build solid web-applications using Ada. Yolk itself doesn't do a whole lot that can't be accomplished simply by using AWS and the GNAT Component Collection (GNATcoll), but it does make the job of building complete web-applications a bit simpler. Things like changing user for the running application, accessing multiple databases, automatically cleaning up log files, adding basic static content handlers and building ATOM syndication XML are all handled by Yolk.

A Yolk application is in reality an AWS application, with some sugar added, so you're not really building a Yolk web-application, as much as you're building an AWS web-application. What I'm getting at, is that you need to understand how to use AWS, in order for Yolk to make any kind of sense. What you get when using Yolk is the little things that AWS does not readily provide.

1.1 The Yolk demo application

Reading this manual will of course (I hope!) help you understand how to use Yolk, but please consider taking a closer look at the Yolk demo application to get a feel for how Yolk is actually used. The demo is heavily commented, so it should be fairly easy to understand what's going on. The demo application is also very suitable as a foundation for other AWS/Yolk applications.

It is much easier to show how to use Yolk, than it is to write down all possible usage scenarios. With the combination of this manual, the Yolk source files and the demo application, you should be able to make full use of the Yolk packages in your own applications.

1.2 The source code

The Yolk source code is the best documentation there is. This document is never going to be as comprehensive as the actual source, so I'll strongly suggest having the source code available as you read this document. What you will find in this document are short descriptions of what a package is meant to do and small usage examples, not a complete rundown of every type and procedure in a package.

1.3 Building and installing Yolk

See the README and INSTALL files. These are found in the Yolk root directory.

1.4 The files Yolk depend upon

When you read this document and the Yolk source code, you'll notice that quite a few packages depend on various files being available at specified locations. This is for example the case with the *Yolk.Configuration* package, that must have a *config.ini* file available in the *configuration/* directory, or the *Yolk.Whoops* package that expects its template file to be found at the path *templates/system/500.tmpl*

All such "dependencies" will of course be noted accordingly as we go along, but instead of forgetting one or more in your own application, I'd much rather encourage using the demo application as a foundation for your own applications, since all these fixed paths and files has been properly added to the demo.

I also recommend compiling and running the demo, to make sure your Yolk install is working as intended. Just read the *demo/README* and *demo/INSTALL* files for instructions on how to get it up and running.

1.5 The Yolk packages naming

The Yolk packages are pretty diverse, ranging from process control to sending email. I've tried naming them as sensibly as possible, in the hope that the package names alone give away their function. If I've failed, well, you're just going to have to refer to this document or take a look at the source for yourself.

Part II
The Yolk Packages

2 Yolk

The Yolk main package currently only contain one thing: The Yolk *Version* string. This is used in a few places, for example in the *directory.tmpl* template file, but obviously it's not something that's of vital importance to most applications.

3 Yolk.Cache.Discrete_Keys

If a piece of data doesn't change very often and it is expensive to build, then caching it might be worthwhile. Instead of going to file/database on every hit, you simply go to the cache and grab the latest version from there. This is ***very*** fast, at the cost of some memory.

If you know exactly what you want to cache, the *Yolk.Cache.Discrete_Keys* package might be just what you need.

3.1 The generic formal parameters

These are:

```
generic
   type Key_Type is (<>);
   type Element_Type is private;
   Max_Element_Age : Duration := 3600.0;
package Yolk.Cache.Discrete_Keys is
...
```

The *Max_Element_Age* defaults to one hour. You should obviously set this to whatever suits your needs. This timer is used for all content in the cache. You cannot set this individually for each element.

3.2 Instantiation

If for example we have two different sets of data (Foo and Bar) that are expensive to build, we can instantiate a *Discrete_Keys* package to handle this:

```
type Cache_Keys is (Foo, Bar);
package My_Cache is new Yolk.Cache.Discrete_Keys
  (Key_Type     => Cache_Keys,
   Element_Type => Unbounded_String);
```

And that is all. We now have a *My_Cache* object that can hold two objects: *Foo* and *Bar*. These are of the type *Unbounded_String* and they have a *Max_Element_Age* of 3600.0 seconds.

3.3 Writing to the cache

Before we can read something from the cache, we must first write something to it:

```
declare
   Foo_Value : Unbounded_String := To_Unbounded_String ("Foo");
begin
   My_Cache.Write (Key   => Foo,
                   Value => Foo_Value);
end;
```

That is all it takes: “Foo” is now safely tucked away in the *My_Cache* object, and will be so for 3600.0 seconds. Calling *Write* with the *Foo* key will always overwrite earlier written *Foo* elements, no matter their age.

3.4 Reading from the cache

A cache obviously only makes sense if you intend to read from it. In our case we want to get our hands on the previously written “Foo” value:

```
declare
   Valid : Boolean := False;
   Value : Unbounded_String;
begin
   My_Cache.Read (Key      => Foo,
                  Is_Valid => Valid,
                  Value    => Value);
   if Valid then
      -- do something interesting with the data
   else
      -- the Foo data is invalid.
   end if;
end;
```

In order for an element to be valid (the *Is_Valid* parameter is true), it must:

1. have been added to the cache in the first place
2. be younger than *Max_Element_Age*

If *Is_Valid* is *False*, then *Value* contains undefined garbage.

3.5 Checking if a key is valid

If you need to check whether a specific key exists in the cache and is valid, then you need to use the *Is_Valid* function.

```
if My_Cache.Is_Valid (Foo) then
   -- Foo is good!
else
   -- Foo is bad!
end if;
```

This follows the same rules as the *Is_Valid* parameter for the *Read* procedure.

3.6 Clearing keys and the entire cache

For clearing of keys and the entire cache we have, naturally, two *Clear* procedures:

```
-- First we clear the Foo key
My_Cache.Clear (Key => Foo);

-- And then we clear the entire cache
My_Cache.Clear;
```

And that's all it takes.

4 Yolk.Cache.String_Keys

This package is almost similar to the *Yolk.Cache.Discrete_Keys* package. The biggest difference is that where the *Discrete_Keys* cache package requires that you define a type for the keys, this packages takes a regular *String* as key.

The implications of this difference between the two cache packages are subtle. Both have the same *Read*, *Write*, *Is_Valid* and *Clear* procedures and functions, so in that sense the two packages are the same. The biggest difference lies in the available generic formal parameters and the addition of the *Cleanup* procedure and the *Length* function.

4.1 The generic formal parameters

These are:

```
generic
   type Element_Type is private;
   Cleanup_Size       : Positive := 200;
   Cleanup_On_Write   : Boolean  := True;
   Max_Element_Age    : Duration := 3600.0;
   Reserved_Capacity  : Positive := 100;
package Yolk.Cache.Discrete_Keys is
...
```

When the amount of elements in the cache >= *Cleanup_Size*, then the *Cleanup* procedure is called by *Write*, if *Cleanup_On_Write* is set to Boolean *True*. *Cleanup_Size* is a sort of failsafe for this cache package. Since we can't know for sure what is being added (we don't know the keys beforehand), we need to make sure it doesn't gobble up all available resources. Set this number high enough that it'll never tricker under normal circumstances, but low enough that it'll prevent resource exhaustion in case of errors.

The *Max_Element_Age* defaults to one hour. You should obviously set this to whatever suits your needs. This timer is used for all content in the cache. You cannot set this individually for each element.

Reserved_Capacity should be set as close as possible to the expected final size of the cache. If your best guestimate is 200 elements in the cache, then set this to 200. Note that this setting has no bearing has no bearing on the actual size of the cache. The cache will happily grow beyond the *Reserved_Capacity* value.

4.2 Instantiation

Instantiating *String_Keys* is done like this:

```
package My_Cache is new Yolk.Cache.String_Keys
  (Element_Type      => Unbounded_String,
   Reserved_Capacity => 200);
```

And that is all. We now have a *My_Cache* object that can hold 200 objects of the type *Unbounded_String*, all of which have a *Max_Element_Age* of 3600.0 seconds.

4.3 Writing to the cache

Before we can read something from the cache, we must first write something to it:

```
declare
   Value : Unbounded_String := To_Unbounded_String ("42");
begin
   My_Cache.Write (Key   => "Foo",
                   Value => Value);
end;
```

That is all it takes: "42" is now safely tucked away in the *My_Cache* object under the key "Foo", and will be so for 3600.0 seconds. Calling *Write* with the "Foo" String will always overwrite earlier written "Foo" elements, no matter their age.

4.4 Reading from the cache

A cache obviously only makes sense if you intend to read from it. In our case we want to get our hands on the previously written "Foo" value:

```
declare
   Valid : Boolean := False;
   Value : Unbounded_String;
begin
   My_Cache.Read (Key      => "Foo",
                  Is_Valid => Valid,
                  Value    => Value);
   if Valid then
      -- do something interesting with the data
   else
      -- the Foo data is invalid.
   end if;
end;
```

In order for an element to be valid (the *Is_Valid* parameter is true), it must:

1. have been added to the cache in the first place
2. be younger than *Max_Element_Age*

If *Is_Valid* is *False*, then *Value* contains undefined garbage.

4.5 Checking if a key is valid

If you need to check whether a specific key exists in the cache and is valid, then you need to use the *Is_Valid* function.

```
if My_Cache.Is_Valid ("Foo") then
   -- Foo is good!
else
   -- Foo is bad!
end if;
```

This follows the same rules as the *Is_Valid* parameter for the *Read* procedure.

4.6 Clearing keys and the entire cache

For clearing of keys and the entire cache we have, naturally, two *Clear* procedures:

```
-- First we clear the Foo key
My_Cache.Clear (Key => "Foo");

-- And then we clear the entire cache
My_Cache.Clear;
```

And that's all it takes.

4.7 How much is in there?

With the *Discrete_Keys* cache we obviously always now the exact amount of keys available, since we've defined the keys ourselves. This is not the case with the *String_Keys* cache, where any String can be a key. If we need to know how many elements that are currently in the cache, we call the *Length* function:

```
if My_Cache.Length > 1000 then
   -- Woa! Lots of stuff in the cache..
end if;
```

Note that *Length* count both valid and invalid elements.

4.8 Cleanup - Keeping cache size in check

if *Cleanup_On_Write* is *True*, then *Cleanup* is called by *Write* whenever the size of the cache reach *Cleanup_Size*. It is of course also possible to call it manually:

```
if My_Cache.Length > 1000 then
   My_Cache.Cleanup;
end if;
```

If you've set *Cleanup_On_Write* to Boolean *False* and the String keys are coming from outside sources, then you really should make sure you call *Cleanup* on a regular basis.

5 Yolk.Config_File_Parser

This package enable you to access KEY/VALUE pairs in configuration files that are written in the style:

```
# This is a comment
-- This is also a comment
KEY VALUE
```

Keys are case-insensitive, so *FOO*, *foo* and *fOo* are all the same. Blank lines and comments are ignored and so is pre/postfixed whitespace. It is not necessary to quote values that contain whitespace, to this:

```
KEY some value with whitespace
```

is perfectly valid, and will return "*some value with whitespace*" when calling *Get (KEY)*. If VALUE is Boolean True or False (case-insensitive), then the KEY can be returned as a String or a Boolean, depending on the target type. If the target type does not match the VALUE and no sensible conversion can be made, then a *Conversion_Error* exception is raised. No dummy values are returned at any time.

To clear a default value, simply add the key to the configuration file, with no value set.

5.1 The generic formal parameters

These are:

```
generic
   use Ada.Strings.Unbounded;
   type Key_Type is (<>);
   type Defaults_Array_Type is array (Key_Type) of Unbounded_String;
   Defaults     : in Defaults_Array_Type;
   Config_File  : in String := "configuration/config.ini";
package Yolk.Config_File_Parser is
...
```

Note that the *Config_File* parameter has a default value. This path is actually where Yolk expects to find the configuration file used by the Yolk.Configuration package.

5.2 Exceptions

There are 3 different exceptions that can be raised by the *Yolk.Config_File_Parser* package. These are:

- *Unknown_Key*. This is raised if an unknown key has been found in the configuration file given when instantiating the package or when *Load_File* is called.
- *Cannot_Open_Config_File*. This is raised when a configuration file cannot be read.
- *Conversion_Error*. This is raised when a value cannot be converted to the target type, ie. the value “42” to a Boolean.

5.3 Instantiation

Yolk.Config_File_Parser is a generic package, so in order to use it, you have to instantiate it, like this:

```
package My_Configuration is

   type Keys is (Foo, Bar);
   type Defaults_Array is array (Keys) of
     Ada.Strings.Unbounded.Unbounded_String;

   Default_Values : constant Defaults_Array :=
                      (Foo => TUS ("some foo"),
                       Bar => TUS ("some bar"));
   --  TUS is a rename of the To_Unbounded_String function. It
   --  is found in the Yolk.Utilities package.

   package Config is new Yolk.Config_File_Parser
     (Key_Type => Keys,
      Defaults_Array_Type => Defaults_Array,
      Defaults => Default_Value,
      Config_File => "config.ini");

end My_Configuration;
```

Here we instantiate the *Config* package with *config.ini* as the configuration file. This means that KEY/VALUE pairs found in this file will overwrite the default values set in the *Default_Values* array. Setting a default value to *Null_Unbounded_String* means the value is empty.

5.4 Re-loading configuration files

With the *Load_File* procedure you can re-load a new configuration file into your *Config* package:

```
My_Configuration.Config.Load_File ("new_config.ini");
```

And that is all. Now the KEY/VALUE pairs of *new_config.ini* will overwrite the ones originally found in the *config.ini* file the package was instantiated with. You can do this as many times as you like. Note that you cannot change what KEY's are valid, so if the *new_config.ini* file contains unknown keys, *Load_File* will raise the *Unknown_Key* exception.

5.5 Getting values

With instantiation and loading of configuration files out of the way, it is now time to get to the configuration values. To get the value of the *Foo* key, you do:

```
My_Configuration.Config.Get (Foo);
```

There are Get functions for the following types:

- Boolean
- Duration

- Float
- Integer
- String
- Unbounded_String

Empty keys simply return an empty String or a *Null_Unbounded_String*, depending on the target type. If a key is empty and the target type is not a String or an Unbounded_String, then the *Conversion_Error* exception is raised.

5.6 Checking if a KEY has a VALUE

You can check if a key has a value with the *Has_Value* function:

```
if Has_Value (Foo) then
   Put_Line ("Foo has a value");
end if;
```

Basically all this function does is return Boolean True if the value of the given key is not a *Null_Unbounded_String*.

6 Yolk.Configuration

This package is a bit of an oddball, as all it does is instantiate the *Yolk.Config_File_Parser* generic with the default AWS and Yolk configuration values. This is used by Yolk internally, but also possibly by the AWS component of your application. The instantiation looks like this:

```
package Config is new Config_File_Parser
  (Key_Type => Keys,
   Defaults_Array_Type => Defaults_Array,
   Defaults => Default_Values,
   Config_File => "configuration/config.ini");
```

As you can see, it is required that there's a configuration file found in *configuration/config.ini*. This path is of course relative to the application. There's a fully commented *config.ini* file available in *demo/exe/configuration/*. This setup is required if you're using Yolk, and the only way to avoid having *config.ini* in the mentioned directory is if you manually change the path in the source file before compiling Yolk.

I recommend taking a look at the Yolk demo application to see how the *Yolk.Configuration* package is used.

6.1 Get the AWS specific configuration settings

When you start an AWS server, you need to give it an *AWS.Config.Object* with all the necessary configuration settings. This is already handled for you in *Yolk.Configuration*, all you have to do is call the *Get_AWS_Configuration* function, like this:

```
AWS_Config : constant AWS.Config.Object :=
  Yolk.Configuration.Get_AWS_Configuration;
AWS.Server.Start (Web_Server => Web_Server,
                  Dispatcher => Resource_Handlers,
                  Config     => AWS_Config);
```

You're not forced to use this, if you don't want to. But even if you don't, you'll still have to make a valid configuration file available in the configuration directory defined in the *Config* instantiation, so you might as well use this as the configuration foundation for the AWS HTTP server component of your application. Again, see the demo application for examples on how this is done.

7 Yolk.Connect_To_DB

This is a small wrapper for the *GNATCOLL.SQL* database connection functionality. Once you've connected to the database, you use plain *GNATCOLL.SQL* tools to query/update your database. The ability to stay connected to several databases in the same application was what drove the development of the *Yolk.Connect_To_DB* package, as the *GNATCOLL.SQL.Exec.Get_Task_Connection* function did not support that out of the box.

At the time of writing, GNATcoll supports PostgreSQL and SQLite, but Yolk only supports connecting to PostgreSQL. The reason for this is simple: I don't use SQLite now and I don't plan on using it in the future, and if I can't test something, I don't feel like adding it to Yolk.

7.1 The Connection_Mapping_Method type

To define how AWS threads are mapped to the database connections, you need to understand the *Connection_Mapping_Method* type. There are two different methods of connecting to a database:

1. AWS_Tasks_To_DB
2. DB_Conn_Tasks_To_DB

The first method is the simplest and it's also the method that requires fewest resources. Using method 1. we simply map each AWS thread to a database connection, meaning that if you have 5 AWS threads, you're going to end up with 5 open connections to your database. Whenever an AWS thread ask for a database connection, it always get the same connection. If no connection has been made yet, one is started and mapped to the AWS thread that started it.

This works flawlessly as long as you don't need to connect to more than one database, but it fails miserably if you have more than one database in play. For such cases we need option 2.

When option 2. is used, one set of database connection tasks are started for every database you connect to. So if you have an AWS server with 5 threads and you need to connect to two databases, you're going to end up with 5 small tasks whose only purpose in life is to open and maintain a connection to the database. These 5 *DB_Conn* tasks are then mapped to the AWS threads using the *Ada.Task_Attributes* package.

This also means that you can only instantiate once using *AWS_Tasks_To_DB* whereas you can instantiate as many times as you like with *DB_Conn_Tasks_To_DB*. If you need to connect to several databases, make sure that the one you use the most is setup with the *AWS_Tasks_To_DB* connection mapping. This is the fastest and most resource efficient method, as we don't have to spend time going through one of the *DB_Conn* tasks that are necessary with the *DB_Conn_Tasks_To_DB* method.

7.2 Database credentials

The *Set_Credentials* function is responsible for setting the credentials necessary to connect to the database.

```
Yolk.Connect_To_DB.Set_Credentials
  (Host     => "localhost",
   Database => "some_database",
   User     => "username",
   Password => "secret");
```

This call return a *Yolk.Connect_To_DB.Credentials* type which is then used when instantiating the *Yolk.Connect_To_DB.PostgreSQL* generic.

8 Yolk.Connect_To_DB.PostgreSQL

When instantiated, this package provides a single function: *Connection*. This function return a *GNATCOLL.SQL.Exec.Database_Connection* type, which is then used to interact with your database.

8.1 The generic formal parameters

When you're actually going to connect to your PostgreSQL database(s), this is the package you want. It's a generic with two formal parameters:

```
generic
   DB_Credentials : Credentials;
   Task_To_DB_Mapping_Method : Connection_Mapping_Method;
package Yolk.Connect_To_DB.PostgreSQL is
   function Connection return GNATCOLL.SQL.Exec.Database_Connection;
end Yolk.Connect_To_DB.PostgreSQL;
```

Instantiation is pretty straightforward:

```
package My_DB is new Connect_To_DB.PostgreSQL
  (DB_Credentials              => Connect_To_DB.Set_Credentials
     (Host      => My.Config.Get (My.DB_Host),
      Database  => My.Config.Get (My.DB_Name),
      User      => My.Config.Get (My.DB_User),
      Password  => My.Config.Get (My.DB_Password)),
   Task_To_DB_Mapping_Method  => Connect_To_DB.AWS_Tasks_To_DB);
```

In this case we've used the *AWS_Tasks_To_DB* mapping method, so each AWS thread will be mapped to a specific database connection.

8.2 Connecting to the database

With the *My_DB* package up and running, connecting to the database is done like this:

```
DB_Conn : constant Database_Connection := My_DB.Connection;
```

You can now use the *DB_Conn* connection just as if you'd created the *Database_Connection* object using the method described in the GNAT Component Collection (GNATcoll) manual. There's also a usage example in the Yolk demo. Take a look at the *View.DB_Test* package.

9 Yolk.Email

Using *Yolk.Email* and the child package *Yolk.Email.Composer* you can build and send more or less any kind of email:

- Plain text
- Multipart/Alternative
- Multipart/Mixed

The package supports adding multiple SMTP servers, meaning you can add as many as you need, and the email will then be send via the first one that accepts it.

The *Yolk.Email* package define 4 exceptions and 3 types. The facilities for actually constructing and sending the email are found in *Yolk.Email.Composer*.

9.1 Exceptions

These are:

- *Attachment_File_Not_Found*. Is raised if a file attachment is not found at the given path.
- *No_Address_Set*. Is raised if the address component of a To, Reply-To, From, Bcc/Cc header is missing.

- *No_Sender_Set_With_Multiple_From*. Is raised when an email contains multiple From headers but no Sender header, as per RFC-5322, 3.6.2. http://tools.ietf.org/html/rfc5322
- *No_SMTP_Host_Set*. Is raised if the SMTP host list is empty, ie. no SMTP host has been set for sending the email.

9.2 The Yolk.Email types

When using *Yolk.Email.Composer* to build and send emails, three types declared in *Yolk.Email* are central:

1. *Character_Set*
2. *Recipient_Kind*
3. *Structure*

The *Character_Set* type define what character set is used when data is added to an email *Structure* object. For example looking at the *Yolk.Email.Composer.Add_From* procedure, we see that the *Charset* parameter defaults to *US_ASCII*:

```
procedure Add_From
  (ES        : in out Structure;
   Address   : in     String;
   Name      : in     String := "";
   Charset   : in     Character_Set := US_ASCII);
```

This does not mean that *Yolk.Email.Composer.Add_From* will encode *Name* as *US_ASCII*, instead it means that the data given in *Name* is already encoded as *US_ASCII*. So if *Name* had contained an ISO-8859-1 encoded String, then the call would've looked like this:

```
declare
   use Yolk.Email;

   Email : Structure;
begin
   Composer.Add_From
     (ES        => Email,
      Address   => "thomas@12boo.net",
      Name      => "Thomas Løcke",
      Charset   => ISO_8859_1);
end;
```

In this case you will end up with a From header looking like this:

```
From: =?ISO-8859-1?Q?Thomas_L=F8cke?= <thomas@12boo.net>
```

So bear in mind that it is your responsibility to encode that data as you need, and then set the *Character_Set* parameters accordingly.

The *Recipient_Kind* type define the kind of recipient that is being added to an email. If you've worked with email, these three should be familiar to you:

1. Bcc
2. Cc
3. To

When adding recipients to an email *Structure* the default is To, but since not all recipients are equal, you have the

The *Structure* type is at the core of it all. You declare an object to be of the *Structure* type, and then you use the *Yolk.Email.Composer* facilities to build and send the email.

10 Yolk.Email.Composer

The actual tools for building and sending an email is found in this package. Here are tools for building emails from the ground up and there are a few convenience procedures if you just need to send a simple email with no bells or whistles.

I'm not going to go through ever procedure in this package, instead I'll show an example on how to build an email from the ground up and how to use one of the convenience procedures.

10.1 Building and sending an email, the easy way

There are two convenience procedures in *Yolk.Email.Composer* for sending emails without having to do a whole lot of work/thinking. They are both named *Send* and they both look like this:

```
procedure Send
   (ES            : in out Structure;
    From_Address  : in     String;
    From_Name     : in     String := "";
    To_Address    : in     String;
    To_Name       : in     String := "";
    Subject       : in     String;
    Text_Part     : in     String;
    SMTP_Server   : in     String := "localhost";
    SMTP_Port     : in     Positive := 25;
    Charset       : in     Character_Set := US_ASCII);

procedure Send
   (ES            : in out Structure;
    From_Address  : in     String;
    From_Name     : in     String := "";
    To_Address    : in     String;
    To_Name       : in     String := "";
    Subject       : in     String;
    Text_Part     : in     String;
    HTML_Part     : in     String;
    SMTP_Server   : in     String := "localhost";
    SMTP_Port     : in     Positive := 25;
    Charset       : in     Character_Set := US_ASCII);
```

As you can see, the only difference between these two is that the first one sends plain text emails, while the second one sends *multipart/alternative* with both plain text and HTML parts. Usage is as simple as:

```
declare
   use Yolk.Email;

   Email : Structure;
begin
   Composer.Send (ES           => Email,
                  From_Address => "thomas@12boo.net",
                  From_Name    => "Thomas Løcke",
                  To_Address   => "tl@ada-dk.org",
                  To_Name      => "Me",
                  Subject      => "Is this thing on?",
                  Text_Part    => "Hey you!",
                  Charset      => ISO_8859_1);

   if Composer.Is_Send (Email) then
      --  Success!
   else
      --  Failure!
   end if;
end;
```

It is possible, and allowed, to call some of the various other procedures prior to calling one of these. If for example you want to add a custom header, it can be done like this:

```
declare
   use Yolk.Email;

   Email : Structure;
begin
   Composer.Add_Custom_Header (ES    => Email,
                               Name  => "User-Agent",
                               Value => "My User Agent");

   Composer.Send (ES           => Email,
                  From_Address => "thomas@12boo.net",
                  From_Name    => "Thomas Løcke",
                  To_Address   => "tl@ada-dk.org",
                  To_Name      => "Me",
                  Subject      => "Is this thing on?",
                  Text_Part    => "Hey you!",
                  Charset      => ISO_8859_1);

   if Composer.Is_Send (Email) then
      --  Success!
   else
      --  Failure!
   end if;
end;
```

And with that, the header *User-Agent:* is now added to the email:

```
User-Agent: My User Agent
```

It hardly gets any easier than that. Lets move on and see how the above is accomplished the hard way.

10.2 Building and sending email, the hard way

It is possible to build an email from the ground up, which obviously allows for a more fine grained control over what is added. It is also a bit more complicated, but not much. Lets try and mimick the easy examples, the “hard” way:

```
declare
   use Yolk.Email;

   Email : Structure;
begin
   Composer.Add_Custom_Header (ES    => Email,
                               Name  => "User-Agent",
                               Value => "My User Agent");

   Composer.Add_From (ES      => Email,
                      Address => "thomas@12boo.net",
                      Name    => "Thomas Løcke",
                      Charset => ISO_8859_1);

   Composer.Add_Recipient (ES      => Email,
                           Address => "tl@ada-dk.org",
                           Name    => "Me");

   Composer.Set_Subject (ES      => Email,
                         Subject => "Is this thing on?");

   Composer.Set_Text_Part (ES => Email,
                           Part => "Hey you!");

   Composer.Add_SMTP_Server (ES   => Email,
                             Host => "localhost");

   Composer.Send (ES => Email);

   if Composer.Is_Send (Email) then
      -- Success!
   else
      -- Failure!
   end if;
end;
```

Harder yes, but really not all that much more difficult.

11 Yolk.Handlers

Most web applications will need to handle various static content, such as PNG, HTML and CSS files. *Yolk.Handlers* helps you accomplish that, so you don't have to build your own handlers for these kinds of files.

The following filetypes are supported by *Yolk.Handlers*:

- CSS
- GIF
- HTML
- ICO
- JPG
- JS
- PNG
- SVG
- XML
- XSL

The filetypes that are textual, are compressed according to the *Yolk.Configuration* setting *Compress_Static_Content* parameter, and the various regular expressions for identifying these filetypes are also defined in *Yolk.Configuration* by the *Handler_** parameters. These regular expressions are registered by the *AWS.Services.Dispatchers.URI.Register_Regexp* procedure.

There's only one procedure in the *Yolk.Handlers* package:

```
procedure Set (RH : out AWS.Services.Dispatchers.URI.Handler);
```

You can see an example on how this is used in the demo file *my_handlers.adb*. There's really very little reason not to use this package for handling of static content, but it is of course not mandatory.

This package makes use of the *Yolk.Static_Content* package for the actual delivery of the content to the user.

12 Yolk.Log_File_Cleanup

House-keeping is a very important task, and when dealing with HTTP servers, one of those tasks involve keeping an eye on the ever growing amount of access and error log files. To help automate this task we have the *Yolk.Log_File_Cleanup* package. This package contains one procedure:

```
procedure Delete
  (Config_Object          : in AWS.Config.Object;
   Web_Server             : in AWS.Server.HTTP;
   Amount_Of_Files_To_Keep : in Positive);
```

Whenever the *Delete* procedure is called, the directory where AWS keeps its log files is traversed and if more log files than *Amount_Of_Files_To_Keep* is found, then the oldest of these are deleted until the amount equals *Amount_Of_Files_To_Keep*.

Note that *Yolk.Log_File_Cleanup* only deals with the AWS acces and error log files. In the demo file *demo/src/yolk_demo.adb* there's an example on how to call this procedure using a task that wakes up according to the configuration setting *Log_File_Cleanup_Interval*.

Be sure to read the *Logging - Rotating And Regular* section in the *demo/exe/configuration/-config.ini* file for more information on the configuration parameters that affect this system.

13 Yolk.Not_Found

This job of this package is to return a HTTP 404 status code and an accompanying simple not found HTML page. It's sole function *Generate* is about as simple as they come:

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

It relies on the template file *demo/exe/templates/system/404.tmpl* to generate the generic 404 HTML page, so if you want to use *Yolk.Not_Found* in your own application, then remember to bring along this file. Where the *404.tmpl* is placed is defined in the configuration parameter *System_Templates_Path*.

Also worth noting is that the *Yolk.Not_Found.Generate* function is used as the default callback in the demo application. This means that all requested resources that doesn't match a registered dispatcher, is served by *Yolk.Not_Found.Generate* ie. a 404 is returned. See the *demo/src/my_handlers.adb* file for more information.

14 Yolk.Process_Control

With *Yolk.Process_Control* you get the ability to control your application using the SIGINT, SIGPWR and SIGTERM signals. You also get a PID file placed next to your executable, which coupled with the *demo/tools/rc.yolk* script gives you a rather nice and simple way of starting and stopping your application.

14.1 Exceptions

These are:

- *Cannot_Create_PID_File*. Is raised if the PID file cannot be created, eg. if the application lacks permissions to write to the directory where the executable is located.

- *Cannot_Delete_PID_File*. Is raised if the PID file cannot be deleted, eg. if the application lacks permissions to write to the directory where the executable is located, or to the PID file itself.
- *PID_File_Exists*. Is raised when the PID file already exists, ie. the application is already running or it was shutdown incorrectly.

14.2 Using Yolk.Process_Control

When you use the *Yolk.Process_Control* package the *Unreserve_All_Interrupts* pragma is used. This means that depending on the compiler used one or more interrupt signals may be affected. In the case of the GNAT compiler, this is specifically mentioned in the source of the *Ada.Interrupts.Names* package:

> – The pragma Unreserve_All_Interrupts affects the following signal(s):
>
> – SIGINT: made available for Ada handler

Since neither SIGPWR or SIGTERM are reserved by the compiler, the *Yolk.Process_Control* package is able to assume control of these signals. You can read more about the pragma Unreserve_All_Interrupts here. If you compile Yolk with a different compiler than GNAT, then please check if one of the affected signals are reserved.

There are two procedures in the *Yolk.Process_Control* package:

```
procedure Stop;

procedure Wait;
```

When you call the *Wait* procedure, you basically enter an endless loop that doesn't stop until

1. The *Stop* procedure is called
2. The application receives a SIGINT, SIGPWR or SIGTERM signal

This is quite handy for applications, that need some sort of loop to keep them from terminating. You can see an example on how this can be done in the *demo/src/yolk_demo.adb* file.

When *Wait* is called, subsequent calls to *Wait* are ignored, unless a call to *Stop* has been made or the application has received one of the SIGINT, SIGPWR or SIGTERM signals. So it's perfectly valid to do:

```
Wait;
--  Stop called from somewhere in the app
--  Do something...
Wait;
--  The app receives a SIGINT signal
--  Do something...
Wait;
```

Whether or not this is actually useful I don't know, but it is possible.

15 Yolk.Process_Owner

When it is necessary to change the owner of a process, the *Yolk.Process_Owner* package is the solution. Obviously this can also be done when starting the application, using various shell tricks, but I find it it much cleaner to just let the application handle it by itself.

15.1 Exceptions

There's only one:

1. *Username_Does_Not_Exist*. This is raised if the given username doesn't exist on the system.

15.2 Using Yolk.Process_Owner

There's only a single procedure in this package and its specification looks like this:

```
procedure Set_User
  (Username : in String);
  --  Set the process owner to Username.
```

Please note that when changing the user ID of the application with *Set_User*, the group ID is changed to the first group the given user is a member of.

Usage is as simple as expected:

```
declare
begin
   Set_User (Username => "billybob");
exception
   when Username_Does_Not_Exist =>
      --  User is missing. Do something!
end;
```

In the file *demo/src/yolk_demo.adb* you'll find that *Yolk.Process_Owner.Set_User* is used in conjunction with the *Yolk.Configuration.Yolk_User* parameter.

16 Yolk.Rotating_Log

This package provides the ability to write log data to three predefined log traces:

1. Error
2. Info
3. SQL

These traces correspond with the components of the *Yolk.Rotating_Log.Trace_Handles* enumeration type. Data written to one of these traces are written to files and these files are automatically rotated when they reach a specified size. The rotation happens according to a "slot" system, so you don't have to worry about cleaning away old log files on a regular basis. The slot system works by appending a slot number to each log file, so if you set the configuration parameter *Max_Slot_Count* to 3, then when starting the rotating log system you'll get a file named *Foo-rotating-INFO-1.log* for the *Info* trace (if your application is named Foo) and when that file reaches the *Rotating_Log_Size_Limit* it is rotated to *Foo-rotating-INFO-2.log* and so on and so forth. When the last slot rotates, it starts all over with 1.

This system has the obvious advantage of never growing beyond a known limited size. On the other hand it has the disadvantage of not being very good at keeping old log data readily at hand, as everything is going to get overwritten sooner or later.

For more information, take a look at the *Logging - Rotating And Regular* section of the *demo/exe/configuration/config.ini* file.

Note that this package handles all log data generated by the *GNATCOLL.SQL* packages. See the *demo/exe/configuration/config.ini* for how to enable/disable the different levels of SQL related log data.

16.1 Exceptions

These are:

- *Cannot_Create_Log_File*. Is raised when it's not possible to create one of the rotating log files.
- *Cannot_Write_To_Log_File*. Is raised when it's not possible to write to one of the rotating log files.

16.2 Starting the rotating log files

It's not enough to simply *with* the package, you have to start the rotating log system before you can actually use it. This is done with the aptly named *Start_Rotating_Logs* procedure:

```
procedure Start_Rotating_Logs
  (Emit_Warning_If_Already_Running : Boolean := True);
```

Setting the *Emit_Warning_If_Already_Running* to *False* will make sure that a second call to *Start_Rotating_Logs* won't emit a warning message on the *Info* trace.

16.3 Writing log data

With the rotating log system started, writing data to it is a simple matter of calling the *Trace* procedure. It looks like this:

```
procedure Trace
  (Handle     : in Trace_Handles;
   Log_String : in String);
```

It should be obvious how this procedure is used. Note that the configuration setting *Immediate_Flush* controls whether or not the *Log_String* is written to file immediately or if a buffer is used.

17 Yolk.Static_Content

Most web applications have a lot of static content, ie. stuff that doesn't change. Things like PNG's, HTML, CSS and Javascript. These are content types that are common for most websites, so a application is going to have to handle these in some way. This is where *Yolk.Static_Content* comes in. Two kinds of files are handled by *Yolk.Static_Content*:

- Compressable (XML, HTML, CSS, JS and so on)
- Non-compressable (PNG, JPG, GIF, ICO and so on)

It is up to you, the user, to decide whether a specific kind of file is compressable or not - the package does not make any such assumptions. The difference between the two, is that compressable files are compressed prior to delivery, if the clients HTTP request includes a *Accept-Encoding: gzip* header. Non-compressable files are simply returned as is. For both kinds, a generic 404 is returned if the requested file doesn't exist.

Yolk.Static_Content is affected by four configuration settings:

- Compressed_Cache_Directory
- Compressed_Max_Age
- Compress_Minimum_File_Size
- WWW_Root

You should carefully read the *demo/exe/configuration/config.ini* file for information on what exactly these do.

17.1 Initializing the compressed cache directory

The configuration parameter *Compressed_Cache_Directory* defines where the compressed version of the static content is saved. When your application is started, this directory may or may not exist, and it may or may not contain various compressed files. To make sure that the directory exists and is empty, you should call the *Initialize_Compressed_Cache_Directory* procedure:

```
procedure Initialize_Compressed_Cache_Directory
  (Log_To_Info_Trace : in Boolean := True);
```

This procedure creates the compressed cache directory, if it doesn't already exist, and if it exists, it deletes everything in it. So basically you're left with an empty directory after a call to this procedure. Calling it repeatedly will simply wipe out the compressed cache directory. This is a threadsafe operation.

If *Log_To_Info_Trace* is *False* then no information about the deletion and creation of the compressed cache directory is logged to the rotating log Info trace.

Calling *Initialize_Compressed_Cache_Directory* is of course only necessary if you plan on using the *Yolk.Static_Content.Compressable* function.

17.2 Yolk.Static_Content.Compressable

If your application contains a bunch of compressable files of significant size, you can save a lot of bandwidth by using the *Compressable* function to serve them:

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

A natural place to call this function would be where you define your content handlers, as seen in this, slightly altered, excerpt from the *Yolk.Handlers* package:

```
declare

   package SC renames Yolk.Static_Content;

   Resource_Handlers : AWS.Services.Dispatchers.URI.Handler;

begin

   AWS.Services.Dispatchers.URI.Register_Regexp
     (Dispatcher  => Resource_Handlers,
      URI         => "\.css$",
      Action      => Create (Callback => SC.Compressable'Access));

   AWS.Services.Dispatchers.URI.Register_Regexp
     (Dispatcher  => Resource_Handlers,
      URI         => "\.html$",
      Action      => Create (Callback => SC.Compressable'Access));

end;
```

Here you can see that we've defined some content handlers for CSS and HTML files based on a few simple regular expressions, and whenever a resource is requested that matches one of these, the corresponding *Action* callback is called, which in this example is the *Compressable* function. The following steps are then taken:

1. Does the requested resource exist and is it an ordinary file?
   (a) If no, then return a 404 message
   (b) if yes, then proceed to 2
2. Does the client support compressed content?
   (a) If no, then return the requested resource un-compressed

   (b) if yes, then proceed to 3

3. Is there a compressed version of the resource available on disk?
   (a) if no, then make one, save it to disk, and return it
   (b) If yes, then proceed to 4

4. Is the age of the compressed file <= Compressed_Max_Age?
   (a) If no, then delete the current file, build a new one and return it
   (b) If yes, then return it

And that's really all there is to it. Note that *Compressable* always looks for requested content in the *WWW_Root* directory, so if the user requests the file */css/index.css*, then the path is going to be:

```
WWW_Root & "/css/index.ss";
```

Using the default configuration value for *WWW_Root*, we'll end up with the path:

```
static_content/css/index.css
```

The compressed version of *index.css* is saved as:

```
Compressed_Cache_Directory & "/css/index.css" & ".gz";
```

Using the default configuration value for *Compressed_Cache_Directory*, we'll end up with the path:

```
static_content/compressed_cache/css/index.css.gz
```

In these two cases the paths are relative to the executable, but you can of course also define *WWW_Root* and *Compressed_Cache_Directory* as absolute paths in the configuration file.

17.3 Yolk.Static_Content.Non_Compressable

A lot of static content really doesn't benefit from any further compression. This is the case for PNG files, JPEG files and a whole lot of other kinds of files. Static content that has already been compressed, should be handled by the *Non_Compressable* function:

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

A call to *Non_Compressable* is much simpler compared to a call to it's sibling function *Compressable*:

1. Does the requested resource exist and is it an ordinary file?

(a) If no, then return a 404 message

(b) if yes, then return the file

And that's it. Either the resource is there, or it isn't. If the requested resource is */css/index.css*, then the *Non_Compressable* function searches for the resource in:

```
WWW_Root & "/css/index.ss";
```

Using the default configuration value for *WWW_Root*, we'll end up with the path:

```
static_content/css/index.css
```

Just as with *Compressable* a natural place to use *Non_Compressable* is where you define you content handlers, but it can obviously be used wherever your application handles requests for static content.