A Case Study in Ada2012

Implementing a Static Http Server, Axios

Prepared for SOEN6411,
and presented to Dr. Constantinides

Simon J. Symeonidis
5887887

November 25, 2013

Contents

List of Figures

List of Tables

Listings

::Section 1::

Abstract

This is a small case study in the 2012 release of the Ada programming language. The motivation of this case study is to expose one to the broadest topics of the Ada programming language via a detailed example. The topic I have chosen is to write up a very simple web server. A web server typically, has to deal with the following aspects:

- Concurrency
- Sockets, and messaging
- The Protocol (in this case **http**)
- Different Filesystems support

This document is hoped to help the curious reader to learn the aspects of the language with a little more ease, introduce the http protocol slightly, and provide references where possible to direct to sources for more exploration. I also record personal notes.

There is a core aim, to present the aforementioned array of topics to the reader, in a project that is not too big, and intimidate the beginner, and not too small to not be of any learning use. We try and achieve this balance by reducing the lines of code and sacrificing aspects of software architecture[1] in order achieve this.

1.1 Disclaimer

As previously stated, this is a case study that is worked on by a complete beginner in the Ada2012 programming language. You might find that this document is referencing a lot of sources, in a collage fashion. That may be a true fact, but the purpose of this document is not to shed new light to a problem. This document is aimed as a tutorial and reference notes to the programming language, for the beginners and intermediate programmers. Please also consider using the excellent references, as they have provided a lot of help in this case study, and can be used more to broaden ones' knowledge.

1.2 The name 'Axios'

Axios stands for "Ada eXtra ineffectual Object Oriented Server". And it is just that!

[1] In the code base, you might notice some aspects that could be factored out, or areas where coupling could be reduced. This will be done on purpose in order to reduce the amount of indirection, and ease the quick navigation, reading and understanding for the reader.

::Section 2::

Topics of interest

Here we discuss in a little more detail the previously outlined topics. These topics offer us a concise and detailed tests to investigate the the Ada 2012 progrqamming language, along with other tools that can be used to supplement and extend future projects. We will be looking at the following aspects:

- The Ada2012 programming language (Language structure, strengths, weaknesses, etc)
- The Ada2012 Toolchain The **gnat** compiler, the project manager files (gpr)
- AUnit for a unit testing platform

2.1 Concurrency

Ada is considered to have a good concurrency support. We see examples later in the code listings, and some more specific (yet unpractical) examples in the appendix.

2.2 Sockets

Sockets are a good topic in this case study, since sockets are OS dependent therefore will provide some insight on how the language and application will behave on different platforms.

2.3 Cross Platform Filesystem Support

To programmers that have used languages that do not require virtual machines, concerns for cross platform availability of an application, more often arise. This case study will use some simple filesystem operations to indicate what we may and may not do with the language. We will investigate how:

- Location to resources are handled
- Permissions for inodes are handled

2.4 Protocol Implementation

A topic that I have came accross in programming languages that is somewhat interesting to see in implementation, are protocols. Notably **Erlang** makes this a breeze. Investigating this in Ada will be interesting.

For this small project, we'll be implementing 5 parts of the Http protocol. Namely the HEAD, GET, POST, PUT and DELETE methods. We refer to the RFC2616 manual [8] for the definition of behaviours of these methods. Once they are implemented in this case study using Ada, the project shall be tested by using any mainstream browser such as *Firefox* or *Chromium*.

For the reader we extract the definitions and behaviour of these four methods.

2.4.1 Overview of Http Protocol

The Http protocol is a text, request-response protocol. The RFC2616 specification is quite detailed, but we will be implementing a very small subset of the specification for this case study.

The Http version we are concerned with is **1.1**. There exists indempodent and non indempodent methods in the Http specification.

GET is described as an indempodent method. That is, if a request is being made with the same specific set of parameters, then the response should be the same each time that request is repeated. For example, if we had a function such as $f(x) = x + 1$, then each time we provided $x = 1$ we should always retrieve '2'. We present the following table, which describes the methods we will be implementing:

Method	Type
HEAD	Indempondent
GET	Indempondent
POST	Non-Indempondent
PUT	Indempondent
DELETE	Indempondent

Table 1: HTTP Method Indempodency

2.4.2 HEAD

This Http method requests the header fields of an html document. The message body is not to be retrieved. Here is an example of what http headers look like. The definition is taken straight out of the manual:

```
message-header = field-name ":" [ field-value ]
field-name     = token
field-value    = *( field-content | LWS )
field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, separators, and quoted-string>
```

Listing 1: Header Definition

Here is a more practical example. These headers are retrieved from *www.google.ca*:

```
HTTP/1.1 302 Found
Location: http://www.google.ca/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Date: Mon, 22 Apr 2013 20:26:36 GMT
Server: gws
Content-Length: 218
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
```

Listing 2: Html Header

And this is the expected html response definition from a successful request.

```
Response  = Status-Line
            *(( general-header
              | response-header
              | entity-header ) CRLF)
            CRLF
            [ message-body ]
```

Listing 3: Http Response

2.4.3 GET

A **GET** method is defined in the specification, to be used for operations that require **reading** information. On the server side, this should not tamper with the resources. A get request requires **Request-Line** field to be sent. It is up to the server to look for the resource requested, and send the response afterwards.

Taken directly from the manual, here is a demonstration of what the GET request looks like, when “accessing www.w3.org”:

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.w3.org
```

Listing 4: Http Get Request

2.4.4 POST

POST is the non-indempodent method we mentioned before. This means each time a post request is sent, a resource must be created, always guaranteeing different response, for a same request. An example for this would be creating a book with common data fields such as *author, year, title, etc...*

This is what a typical POST request looks like:

In this case study, we will be using this to write a text file on the server, with the request body as file contents. It would be possible to retrieve these contents later on.

2.4.5 PUT

PUT is a indempodent method. It is typically used in order to update resources on the sever side. For example, if we wish to update the book re

In this case study we will be using this to change a variable on the server. That way it will be possible to start the server, change this value, and get a response showing that the update was a success. It is quite possible to use this along with AUnit in order unit test these features automatically.

2.4.6 DELETE

DELETE is an indempodent method. It is typically used in order to delete a resource on a website. We will be using this to delete resources that are created by the POST definition of Axios (refer to 2.4.4).

2.5 First Steps

The first steps in an Ada application is to define the *project manager file* for the given project as outlined on the manual [2].

The GNAT project file abstracts a lot of the building details into this file. It will take care even about other language builds, meaning that if you wish to create, for example C extensions to the language, you could achieve this easily, as opposed to tedious makefiles.

Another added advantage is that GNAT Project Files also take into consideration the file naming conventions. Another automated feature in this file is automatic building of external libraries. There are various other ways to achieve this, either on a makefile level, or alternatively using features in certain SCM tools to clone and build the libraries automatically.

A notable feature of the GPR files, is that hierarchical builds are allowed, meaning that if different components require different builds, and different dependencies, that build can be isolated to that specific build alone [2]. The hierarchical layering of the building process, also exploits a similar aspect to that of a layered software architecture: different layers can be developed at parallel, as could subsystems.

2.5.1 Deployment View

The overall view of the system components can be seen in Figure 1.

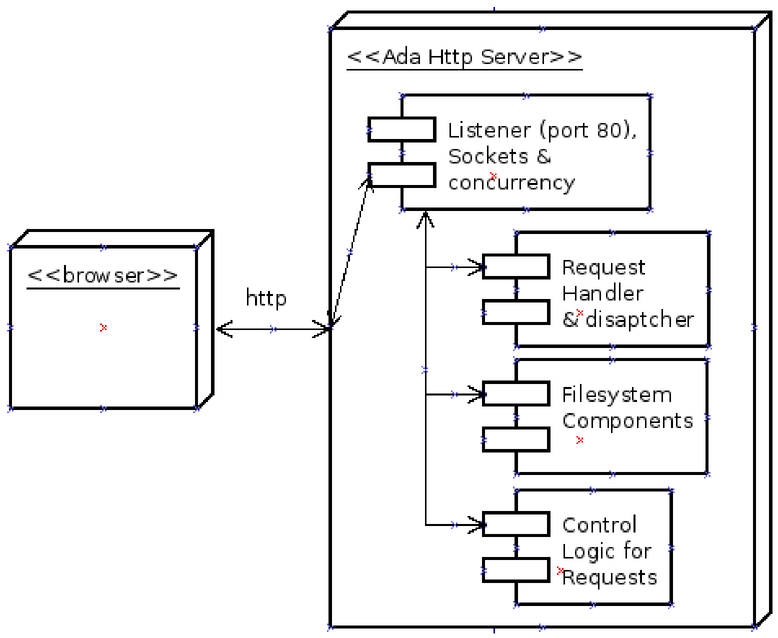

<<Ada Http Server>>
Listener (port 80), Sockets & concurrency
Request Handler & disaptcher
Filesystem Components
Control Logic for Requests
<<browser>>
http

Figure 1: Deployment Diagram

src	The directory to contain the sources
obj	The directory to contain the compiler output
bin	The directory to contain the binaries
obj/debug	The non-optimized compiled binaries with debug info are stored here. You can notice this in the GPR file, where the '-g' flag is given to the compiler.
obj/release	The optimized compiled binaries without debug information. You can notice this in the GPR file, where the 'O2' flag is given to the compiler.
tests	The directory to contain the tests

Table 2: Directory Structure Specification for GPR File

::Section 3::

Implementation

This section will deal with the more technical aspects of this case study. From now on we will list topics that have to do with the implementation of the simple Http server.

3.1 Defining the required GNAT Project File Specification

The specifications list the requirements as noted on Table 2.

3.1.1 GNAT Project File

Here is the project file we will be using.

```
-- Simon Symeonidis, 2013
-- GNAT Project file for the Http server case study.
--
-- Used Manuals:
--   http://docs.adacore.com/gprbuild-docs/html/gprbuild_ug.html
--   http://docs.adacore.com/gnat-unw-docs/html/gnat_ugn_12.html
--
-- Example use:
--   gnatmake -P axios.gpr -Xmode=debug -p
project Axios is

  -- Standard configurations
  for Main        use ("main.adb");
  for Source_Dirs use ("src/**");
  for Exec_Dir    use "bin/";

  -- Ignore git scm stuff
  for Ignore_Source_Sub_Dirs use (".git/");

  -- Objects are contained in their own directories (this is also
  -- known as an isolated build).
  for Object_Dir use "obj/" & external ("mode", "debug");
  for Object_Dir use "obj/" & external ("mode", "release");

  package Builder is
    for Executable ("main.adb")  use "axios";
  end Builder;

  -- To invoke either case, you need to set the -X flag at gnatmake in command
  -- line. You will also notice the Mode_Type type. This constrains the values
  -- of possible valid flags.
  type Mode_Type is ("debug", "release");
  Mode : Mode_Type := external ("mode", "debug");
  package Compiler is
    -- Either debug or release mode
    case Mode is
    when "debug" =>
      for Switches ("Ada") use ("-g");
    when "release" =>
      for Switches ("Ada") use ("-O2");
    end case;
  end Compiler;

  -- Not needed for this study, but listed
  package Binder is
  end Binder;

  -- Not needed for this study, but listed
  package Linker is
  end Linker;

end Axios;
```

Listing 5: GNAT Project File

Notice that some of the flags for the compiler in both release, and debug modes are the same flags used for GNU C++ compiler. We explain each section with a little more detail.

Directory Structure The directory structure is as outlined on the previously demonstrated table. We see these defined in the GPR file by the names of **Source_Dirs**, **Exec_Dir**. Because of the definition of the Object file directories for a debug and release version, we also get the "obj/release" and "obj/debug" directories, as previously required. A Simple hello world application would yield the following structure, for example:

```
.
bin
bin/hello_world
hello.gpr
obj
obj/debug
obj/debug/main.ali
obj/debug/main.o
obj/release
obj/release/main.ali
obj/release/main.o
src
src/main.adb

5 directories, 7 files
```

Listing 6: Directory Structure

Flags There are many flags that can be set for the compilation and linking process. The manual covers a lot of them, but the ones used are explained.

Builder Notice the "package Builder is ..." line. We define the flags that should be used by the builder (gnatmake), when invoked.

Executable This is to specify the output name of the binary once the whole project is compiled.

Compiler Notice the "package Compiler is ..." line. This contains the flags for the actual compiler. We have two flags '-g', and '-O2' that the seasoned GNU C++ compiler user would recognize instantly. The '-g' flag would be used to produce non-optimized binaries in order to contain extra information when debugging.
The '-O2' is for optimization, and that is the reason why it is in the release case. We define a Mode_Type which can either be one of the two literals 'debug', and 'release' in order to achieve this.

3.2 Components

We now describe the components of the project. Figure 2

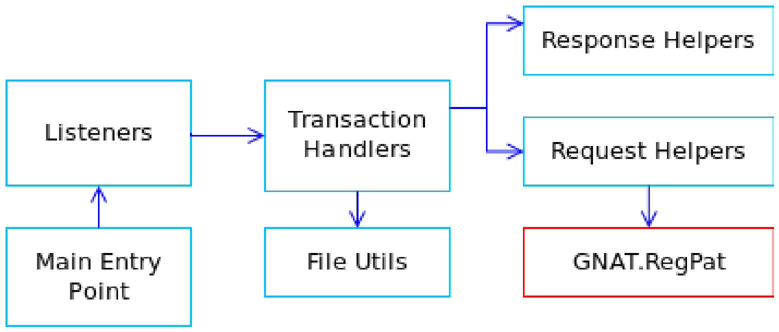


Figure 2: Overall organization of software components of Axios

3.2.1 Main Entry Point

The main entry point is where this particular software begins execution. In the case of this application the first steps that are fired up on execution, is spawning two different listeners. Listener L_1 accepts connections on port 3000, and another Listener L_2 accepts connections on port 8000. We exploit this fact, and make them point to different directories as well; L_1 to $Dir = www1/$ and L_2 to $Dir = www2/$.

```
-- @author Simon Symeonidis
-- @date Fri May  3
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Unbounded;

-- user
with Listeners, File_Utils;
procedure Main is

  -- Pointers so we pass to the task types.
  type Listener_Access is access Listeners.Listener;

  task type Launch_Listener(L : Listener_Access) is
    entry Start;
    entry Stop;
  end Launch_Listener;

  task body Launch_Listener is
  begin
    accept Start;
    L.Print_Info;
    L.Listen;
    accept Stop;
  end Launch_Listener;

  -- listeners. Notice the '
  l1 : Listener_Access :=
    new Listeners.Listener'(Port_Number  => 3000,
    Host_Name    => Ada.Strings.Unbounded.To_Unbounded_String("localhost"),
    Shutdown     => false,
    WS_Root_Path => Ada.Strings.Unbounded.To_Unbounded_String("./www1/"));

  l2 : Listener_Access :=
    new Listeners.Listener'(Port_Number  => 8000,
    Host_Name    => Ada.Strings.Unbounded.To_Unbounded_String("localhost"),
    Shutdown     => false,
    WS_Root_Path => Ada.Strings.Unbounded.To_Unbounded_String("./www2/"));

  -- listener tasks
  lt1 : Launch_Listener(L => l1);
  lt2 : Launch_Listener(L => l2);

begin
  -- Start listening for connections.
  lt1.Start;
  lt2.Start;

  -- On Graceful end
  lt1.Stop;
  lt2.Stop;
end Main;
```

Listing 7: Main Entry Point

The listeners are defined in the data block, and later in the body, their process starts by calling their ‘Start’ procedure. Next we investigate what actually happens within the listeners.

3.2.2 Listener

Specification of the listeners.

```
-- @author Simon Symeonidis
-- @date
--
-- This is the listener for http connections to port 80.
-- This should delegate the request to some handler, and
-- execute the proper logic.
with
  Ada.Text_IO
, Ada.Strings.Unbounded
, Ada.Strings.Unbounded.Text_IO
, Ada.Exceptions
, Ada.Streams
, Ada.Calendar
, Ada.Real_Time
, GNAT.Sockets
-- user
, Transaction_Handlers
;

use
  Ada.Exceptions
, GNAT.Sockets;

use type
  Ada.Streams.Stream_Element_Count
, Ada.Streams.Stream_Element_Array
, Ada.Real_Time.Time
, Ada.Real_Time.Time_Span
;

package Listeners is

  -- This is basically the specification for web servers. The way this is done
  -- allows us to create multiple listeners on the same machine.
  type Listener is tagged record
    Port_Number   : Integer range 0 .. 16#ffff#;
    Host_Name     : Ada.Strings.Unbounded.Unbounded_String;
    Shutdown      : Boolean := False;
    WS_Root_Path  : Ada.Strings.Unbounded.Unbounded_String;
  end record;

  procedure Print_Info(This : Listener);
  procedure Listen(This : Listener);
  function Tiny_Name(This : Listener) return String;
  function Response_Date return String;
end Listeners;
```

Listing 8: Simple Listener Specification

Implementation of the listeners.

```
-- @author Simon Symeonidis
-- @date   Mon May 6 2013
-- Implementation for the listener
--
-- Some nice references / supplementary material
--
-- + Simple http server in ruby using tcpserver
--       http://stackoverflow.com/questions/7540064/
-- + Using Ada get_line for unknown input size
--       http://www.radford.edu/~nokie/classes/320/stringio.html

package body Listeners is

  -- Print to the command line information such
-- as host and port number
procedure Print_Info(This : Listener) is
begin
  Ada.Text_IO.Put_Line("Listener Info: ");
  Ada.Text_IO.Put_Line("Port Number : "
    & Integer'Image(This.Port_Number));
  Ada.Text_IO.Put_Line("Hostname    : "
    & Ada.Strings.Unbounded.To_String(This.Host_Name));
  Ada.Text_IO.Put_Line("Root Dir.   : "
    & Ada.Strings.Unbounded.To_String(This.WS_Root_Path));
  Ada.Text_IO.New_Line;
end Print_Info;

function Tiny_Name(This : Listener)
return String is
  Name : String :=
    Ada.Strings.Unbounded.To_String(This.Host_Name) & "@" &
    Integer'Image(This.Port_Number);
begin
  return Name;
end Tiny_Name;

-- Return the date as string, in the format specified by the RFC2616 manual.
-- For now we just add a placeholder date.
-- TODO need to fix this
function Response_Date
return String is
begin
  return "Date: Tue, 20 Jan 2012 10:48:45 GMT";
end Response_Date;

-- Listen forever. Graceful shutdown if it receives some signal.
procedure Listen(This : Listener) is
  Socket   : Socket_Type;
  Server   : Socket_Type;
  Address  : Sock_Addr_Type;
  Channel  : Stream_Access;
  The_Host : String := Ada.Strings.Unbounded.To_String(This.Host_Name);
begin

  Address.Addr := Addresses (Get_Host_By_Name (The_Host), 1);
  Address.Port := Port_Type(This.Port_Number);
  Create_Socket(Server);

  Set_Socket_Option(Server, Socket_Level, (Reuse_Address, True));
  Bind_Socket(Server, Address);
  Listen_Socket(Server);
  -- pass to other socket.
  Ada.Text_IO.Put_Line
   ("Successfully listening on: " &
    Port_Type'Image(Address.Port));

  Listening_Loop :
  while not This.Shutdown loop
    Accept_Socket(Server, Socket, Address);
    Channel := Stream(Socket);
    declare
      CRLF       : String    := ASCII.CR & ASCII.LF;
      LF         : Character := ASCII.LF;
```

```
         Counter       : Integer   := 0;

         Request       : Ada.Strings.Unbounded.Unbounded_String;
         Chara         : Character;

         RTime_Start   : Ada.Real_Time.Time := Ada.Real_Time.Clock;
         RTime_Stop    : Ada.Real_Time.Time;
         RTime_Total   : Ada.Real_Time.Time_Span;
       begin
         -- Read the request body
         Read_Request :
         loop
         Character'Read(Channel, Chara);
         Ada.Strings.Unbounded.Append(
           Source   => Request,
           New_Item => Chara);

           if Chara = ASCII.LF or Chara = ASCII.CR then
             Counter := Counter + 1;
           else -- reset
             Counter := 0;
           end if;

           exit when Counter = 4;
         end loop Read_Request;

         RTime_Stop  := Ada.Real_Time.Clock;
         RTime_Total := RTime_Stop - RTime_Start;

         Ada.Text_IO.Put_Line(
           Ada.Strings.Unbounded.To_String(
             Request));

         String'Write(Channel,
         Transaction_Handlers.Handle_Request(
           Ada.Strings.Unbounded.To_String(Request),
             Ada.Strings.Unbounded.To_String(This.WS_Root_Path)));

       exception when E : others =>
         Ada.Text_IO.Put_Line
           ("Listeners, at main listen loop: " &
           Exception_Name(E) & Exception_Message(E));

       end;
       Free(Channel);
       Close_Socket(Socket);
     end loop Listening_Loop;

   end Listen;
end Listeners;
```

Listing 9: Simple Listener Body

Next we investigate the transaction handlers. The transaction handlers help us generate the content that we need to relay to the clients (browsers).

3.2.3 Transaction Handlers

Specification of the transaction handlers.

```
with
  Request_Helpers
, Response_Helpers
, File_Utils
;

-- @author Simon Symeonidis
-- @date   Fri May 17
-- This encapsulates business logic of the http server (what to do
-- with each given request).
package Transaction_Handlers is
  function Handle_Request(R : String; Context : String) return String;
end Transaction_Handlers;
```

Listing 10: Transaction Handler Specification

It's pretty straightforward - when the client (browser) sends us a request, we want to identify what kind of request that is, and what to do with it. Therefore we streamline this to the function 'Handle_Request'. Now let's take a look how we actually handle the implementation of the request.

Implementation of the transaction handlers.

```
-- @author Simon Symeonidis
-- Implementation of the transaction handlers
package body Transaction_Handlers is
  function Handle_Request(R : String; Context : String)
  return String is
    package rh renames Response_Helpers;
    package rr renames Request_Helpers;
    R_Type : rr.Request_Type :=
             rr.Parse_Request_Type(R);
    URI    : String := rr.Parse_Request_URI(R);
  begin
  case R_Type is
    when rr.GET    =>
      return
        rh.Make_Response(
         File_Utils.Read(Context & "/" & URI));

    when rr.POST   =>
      return
        rh.Make_Response("<h1>Got Post</h1>");

    when rr.PUT    =>
      return
        rh.Make_Response("<h1>Got Put</h1>");

    when rr.DELETE =>
      return
        rh.Make_Response("<h1>Got DELETE</h1>");

    when rr.HEAD   =>
      return
        rh.Make_Response("<h1>Got HEAD</h1>");

    when others =>
      return
        rh.Make_Response("<h1>Error!</h1>"
        & "<p> You did something very weird in order to see this</p>");

  end case;
  end Handle_Request;
end Transaction_Handlers;
```

Listing 11: Transaction Handler Implementation

We see that we can handle the requests 'GET', 'POST', 'PUT', 'DELETE', 'HEAD' appropriately using the case clause. We've implemented functionality mainly for 'GET', since 'Axios' simply shares static information. We support however the rest of the 'Http' methods by returning simple strings to handle the requests at a minimum. Next we investigate the 'Request_Helpers' and 'Reponse_Helpers'.

3.2.4 Request Helpers

Specification of request helpers.

```
with GNAT.Regpat;
with Ada.Text_IO;
-- @author Simon Symeonidis
-- @date   Thu May 16 2013
package Request_Helpers is
  type Request_Type is (HEAD, GET, POST, PUT, DELETE, OTHER);

  function Parse_Request_Type(R : String) return Request_Type;
  function Parse_Request_URI(R : String)  return String;
  procedure Get_Word(S           : String;
                     First, Last : out Positive;
                     Found       : out Boolean);
end Request_Helpers;
```

Listing 12: Request Helper Specification

Implementation of request helpers.

```
-- @author Simon Symeonidis
-- @date   Thu May 16 2013
package body Request_Helpers is
  -- Parse the request type (get first line, see what type of
  -- http method it is and return the valid enumaration).
  function Parse_Request_Type(R : String)
  return Request_Type is
    First, Last : Positive;
    Found       : Boolean;
  begin
    Get_Word(R, First, Last, Found);

    if    R(First..Last) = "GET"    then
      return GET;
    elsif R(First..Last) = "POST"   then
      return POST;
    elsif R(First..Last) = "PUT"    then
      return PUT;
    elsif R(First..Last) = "HEAD"   then
      return HEAD;
    elsif R(First..Last) = "DELETE" then
      return DELETE;
    else
      return GET;
    end if;
  end Parse_Request_Type;

  -- This extracts the path requested from the first line in the http
  -- method eg:
  --
  --   GET /index.html
  --   ...
  --
  -- Would return a string with "index.html"
  function Parse_Request_URI(R : String)
  return String is
    First, Last : Positive;
    Found : Boolean;
  begin
    -- Called twice, because the second 'word' is the uri in the request
    -- header.
    Get_Word(R, First, Last, Found);
    Get_Word(R(Last+1..R'Last), First, Last, Found);

    if Found then
      return R(First..Last);
    end if;

    return "uri not found.";
  end Parse_Request_URI;

  -- @note Thanks to
  --   http://rosettacode.org/wiki/Regular_expressions#Ada
  procedure Get_Word(S           : String;
                     First, Last : out Positive;
                     Found       : out Boolean) is
    Pattern     : constant String := "(\S+)";
    Compiled    : GNAT.Regpat.Pattern_Matcher :=
      GNAT.Regpat.Compile(Pattern);
    Match_Array : GNAT.Regpat.Match_Array(0..1);
  begin
    GNAT.Regpat.Match(Compiled, S, Match_Array);
    Found := not GNAT.Regpat."="(Match_Array(1), GNAT.Regpat.No_Match);

    if Found then
      First  := Match_Array(1).First;
      Last   := Match_Array(1).Last;
    end if;
  end Get_Word;
  end Request_Helpers;
```

Listing 13: Request Helper Implementation

Here we see that the *Request Helper* provides us with some auxiliary functions that we can use in order to discern what kind of request is incoming. We are also provided with another auxiliary function to extract the requested path from ‘GET’ Http methods. We use these paths in order to look into the static resources, and if found, send the resource back to the user. So for example L_1 has a directory *www*1, so the path requested would be prefixed with *www*1, and return the contents if found. Else an error message is served back.

3.2.5 Response Helpers

Specification of response helpers.

```
-- @author Simon Symeonidis
-- @date   Fri May 17
-- This is a helper for forming the responses.
package Response_Helpers is
  function Headers return String;
  function Response_Date return String;
  function Make_Response(S : String) return String;
private
  CRLF : constant String := ASCII.CR & ASCII.LF;
end Response_Helpers;
```

Listing 14: Respone Helper Specification

The two main important functions that one should notice are ‘Headers’ and ‘Make_Response’. ‘Header’ is to ready a header for us to send back, and the ‘Make_Response’ is to combine a header, with the required HTML body, and calculate a length to provide to the ‘Content-Length’ http header field.

Implementation of response helpers.

```
with Ada.Calendar;
with GNAT.Calendar.Time_IO;
-- @author Simon Symeonidis
-- This is the response helper that readies a header to send back to
-- the client and concatenates the html body as well.
package body Response_Helpers is

  -- @note don't use this directly
  function Headers
  return String is
  begin
    return
      "Http/1.1 200 OK"                             & CRLF &
      Response_Date                                 & CRLF &
      "Server: axios"                               & CRLF &
      "Content-Type: text/html; charset=iso-8859-1" & CRLF &
      "Content-Length: ";
  end Headers;

  function Response_Date
  return String is
    package ac  renames Ada.Calendar;
    package gct renames GNAT.Calendar.Time_IO;
    Current_Time : ac.Time;
    Format       : gct.Picture_String := "%a, %d %B %Y %H:%M:%S EST";
    Field_Name   : constant String    := "Date: ";
  begin
    Current_Time := ac.Clock;
    return
      Field_Name &
      gct.Image(Current_Time, Format);
  end Response_Date;

  -- Make the response for the browser. This function includes
  -- the headers, (including the content-length field of the body),
  -- and the message body.
  function Make_Response(S : String)
  return String is
  begin
    return
      Headers &
      Positive'Image(S'Length) & CRLF & CRLF &
      S;
  end Make_Response;


end Response_Helpers;
```

Listing 15: Response Helper Implementation

And here we see the realization of the specification. 'Header' indeed provides a function to construct an http header to send back. This method is used along with 'Make_Respone' in order to send back everything with the required body containing the 'Html'. The body is in fact the requested resource from the user. Last but not least, we present some file utilities that help us achieve sending back the required resources.

3.2.6 File Utils

Specification of file utils.

```
with Ada.Strings.Unbounded;
-- @author Simon Symeonidis
-- Specification of the file utils helper
package File_Utils is
  procedure
    Write(File_Name : String; Contents : String);

  procedure
    Append(File_Name : String; Contents : String);

  function
    Read(File_Name : String)
    return String;
end File_Utils;
```

Listing 16: File Utils Specification

Implementation of file utils.

```ada
-- @author Simon Symeonidis
--
-- Common read and write utils so that we separate the job of the
-- post and put method calls to this package. This is also a shorthand
-- for pretty basic file read and write operations.
--
-- Thanks to
-- http://rosettacode.org/wiki/Read_entire_file#Ada.Direct_IO
-- http://rosettacode.org/wiki/Read_a_file_line_by_line#Ada
-- http://faculty.cs.wwu.edu/reedyc/AdaResources/LectureNotes/Text_Files.html
-- http://en.wikibooks.org/wiki/Ada_Programming/Libraries/Ada.Text_IO

with Ada.Text_IO;
with Ada.Direct_IO;
with Ada.Exceptions; use Ada.Exceptions;

package body File_Utils is

  -- Write to a file with bounded strings. Please note this will overwrite
  -- previously created files.
  procedure Write(File_Name : String; Contents : String) is
    The_File_Mode : Ada.Text_IO.File_Mode := Ada.Text_IO.Out_File;
    The_File      : Ada.Text_IO.File_Type;
  begin
    Ada.Text_IO.Create
        (File => The_File,
   Mode => The_File_Mode,
   Name => File_Name);
    Ada.Text_IO.Put
        (File => The_File,
   Item => Contents);
    Ada.Text_IO.Close(File => The_File);

  exception when E : others =>
    Ada.Text_IO.Put_Line
      (Exception_Name(E) & Exception_Message(E));
  end Write;

  -- This will read the full contents of a given file.
  -- Read a file by name, and then return the file text contents, lines
  -- delimited by a ASCII.LF
  function Read(File_Name : String)
  return String is
    The_File_Mode : Ada.Text_IO.File_Mode := Ada.Text_IO.In_File;
    The_File      : Ada.Text_IO.File_Type;
    Contents      : Ada.Strings.Unbounded.Unbounded_String;
  begin
    Ada.Text_IO.Open
      (File => The_File,
       Mode => The_File_Mode,
       Name => File_Name);

    while not Ada.Text_IO.End_Of_File (The_File) loop
      declare
        Line : String := Ada.Text_IO.Get_Line(The_File);
      begin
        Ada.Strings.Unbounded.Append(
          Source => Contents, New_Item => Line(1..Line'Last));
      end;
    end loop;

    Ada.Text_IO.Close(File => The_File);
    return Ada.Strings.Unbounded.To_String(Contents) ;

  exception when E : others =>
    Ada.Text_IO.Put_Line("Warning: request to a non existant file was made.");
    Ada.Text_IO.Put_Line(">> Path: " & File_Name);
    return "Error";
  end Read;

  -- Append to file by given name.
  procedure Append(File_Name : String; Contents : String) is
  begin
    null;
    --exception when E : others =>
    --  Ada.Text_IO.Put_Line
    --    (Exception_Name(E) & Exception_Message(E));
  end Append;

end File_Utils;
```

Listing 17: Request Helper Implementation

Mainly the ‘Read’ function is what is of interest here. But this package is straightforward, and does not need any more explanations.

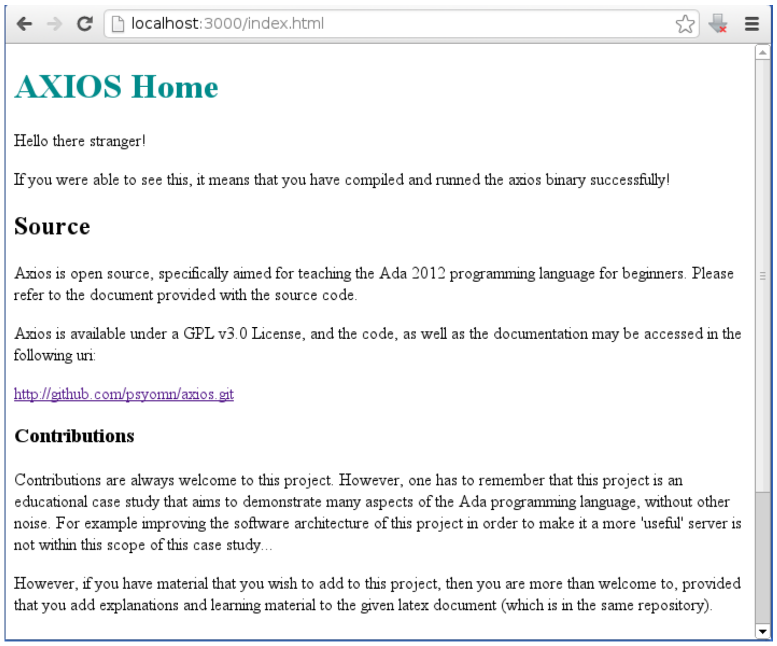
3.3 Testing

Testing is achieved by using **AUnit**, that provides us with some facilities. More information can be read here [3].

::Section 4::

Demonstration

Figure 3 is a small demo on what it looks like when accessing the *AXIOS* server. Figure 4 shows the output of the server (which is the text request sent by the browser, unless there is an error rased).



localhost:3000/index.html

AXIOS Home

Hello there stranger!

If you were able to see this, it means that you have compiled and runned the axios binary successfully!

Source

Axios is open source, specifically aimed for teaching the Ada 2012 programming language for beginners. Please refer to the document provided with the source code.

Axios is available under a GPL v3.0 License, and the code, as well as the documentation may be accessed in the following uri:

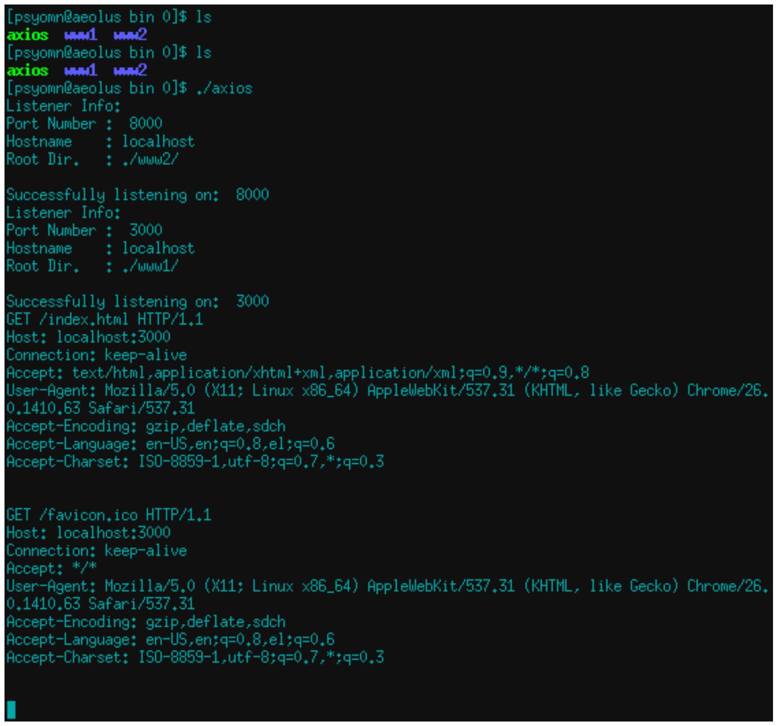
http://github.com/psyomn/axios.git

Contributions

Contributions are always welcome to this project. However, one has to remember that this project is an educational case study that aims to demonstrate many aspects of the Ada programming language, without other noise. For example improving the software architecture of this project in order to make it a more 'useful' server is not within this scope of this case study...

However, if you have material that you wish to add to this project, then you are more than welcome to, provided that you add explanations and learning material to the given latex document (which is in the same repository).

Figure 3: Using a browser to access the server



```
[psyomn@aeolus bin 0]$ ls
axios  www1  www2
[psyomn@aeolus bin 0]$ ls
axios  www1  www2
[psyomn@aeolus bin 0]$ ./axios
Listener Info:
Port Number :  8000
Hostname    : localhost
Root Dir.   : ./www2/

Successfully listening on:  8000
Listener Info:
Port Number :  3000
Hostname    : localhost
Root Dir.   : ./www1/

Successfully listening on:  3000
GET /index.html HTTP/1.1
Host: localhost:3000
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.31 (KHTML, like Gecko) Chrome/26.
0.1410.63 Safari/537.31
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,el;q=0.6
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3


GET /favicon.ico HTTP/1.1
Host: localhost:3000
Connection: keep-alive
Accept: */*
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.31 (KHTML, like Gecko) Chrome/26.
0.1410.63 Safari/537.31
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,el;q=0.6
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

Figure 4: Server output when receiving requests

::Section 5::

Appendix

The appendix will give some extra simplified information which might help understand the smaller parts of the given case study. Explanations and code listings are provided as best as possible.

5.1 Networking

A lot of the networking will seem very familiar to you if you have used BSD C Sockets. Provided are two Ada procedures. One listener, and one sender. The Listener listens for incoming connections. If a message is intercepted, the message is uppercased, and sent back to the client. The client simply sends a string message to the listener. The listener listens at port 3000 (it is assumed these procedures run on your local machine, together).

```ada
-- @author Simon Symeonidis
-- A concise example of a listener procedure, adapted from the
-- ping pong example in the GNAT.Sockets specification file.
with Ada.Text_IO;
with Ada.Strings;
with Ada.Strings.Fixed;
with Ada.Strings.Maps;
with Ada.Strings.Maps.Constants;
with GNAT.Sockets; use GNAT.Sockets;
with Ada.Exceptions; use Ada.Exceptions;

procedure Simple_Listener is
  Address  : Sock_Addr_Type;
  Server   : Socket_Type;
  Socket   : Socket_Type;
  Channel  : Stream_Access;
begin
  -- Init socket stuff.
  Address.Addr := Addresses (Get_Host_By_Name (Host_Name), 1);
  Address.Port := 3000;
  Create_Socket (Server);

  Set_Socket_Option(Server, Socket_Level, (Reuse_Address, True));
  Bind_Socket (Server, Address);
  Listen_Socket(Server);
  Accept_Socket(Server,Socket,Address);
  Channel := Stream (Socket);
  declare
    Message : String := String'Input (Channel);
  begin
    Ada.Text_IO.Put_Line("Got: " & Message);
    String'Output (Channel,
      Ada.Strings.Fixed.Translate(Message,
        Ada.Strings.Maps.Constants.Upper_Case_Map));
  end;

  Close_Socket (Server);
  Close_Socket (Socket);

  exception when E : others =>
    Ada.Text_IO.Put_Line
      (Exception_Name (E) & Exception_Message(E));
```

```
end Simple_Listener;
```

Listing 18: Simple Listener

```
-- @author Simon Symeonidis
-- A concise example of a client procedure, adapted from the
-- ping pong example in the GNAT.Sockets specification file.

with Ada.Text_IO;
with GNAT.Sockets; use GNAT.Sockets;

procedure Simple_Sender is
  Address : Sock_Addr_Type;
  Socket  : Socket_Type;
  Channel : Stream_Access;
begin
  -- Init socket stuff
  Address.Addr := Addresses (Get_Host_By_Name (Host_Name), 1);
  Address.Port := 3000;
  Create_Socket(Socket);
  Set_Socket_Option(Socket, Socket_Level, (Reuse_Address, True));
  Connect_Socket (Socket, Address);
  Channel := Stream (Socket);
  String'Output (Channel, "hack the planet.");

  declare
    Message : String := String'Input(Channel);
  begin
    Address := Get_Address(Channel);
    Ada.Text_IO.Put_Line ("Server @ " & Image(Address)
                          & ": " & Message);
  end;

  -- Cleanup
  Close_Socket(Socket);
end Simple_Sender;
```

Listing 19: Simple Sender

Also, here is an example http get program that helped a lot in this case study. It is one of the GNAT Sockets examples that can be found in the following location [1]. I have modified the source a little, and provided it in listing 20.

```
with Ada.Text_IO;               use Ada.Text_IO;
with GNAT.Sockets;              use GNAT.Sockets;
with Ada.Strings.Unbounded;    use Ada.Strings.Unbounded;

with Ada.Streams;
use type Ada.Streams.Stream_Element_Count;

-- Thanks to
-- http://en.wikibooks.org/wiki/Ada_Programming/Libraries/Ada.Streams#
    Read_attribute
procedure Vcv is
   Client  : Socket_Type;
   Address : Sock_Addr_Type;
   Channel : Stream_Access;

   CRLF    : String := (1 => ASCII.CR, 2 => ASCII.LF);
   Send    : String := CRLF & CRLF;
   Offset : Ada.Streams.Stream_Element_Count;
   Data    : Ada.Streams.Stream_Element_Array (1 .. 256);
begin
   GNAT.Sockets.Initialize;  -- initialize a specific package
   Create_Socket (Client);
   Address.Addr := Addresses (Get_Host_By_Name("www.google.com"));
   Address.Port := 80;

   Connect_Socket (Client, Address);
   Channel := Stream (Client);

   String'Write (Channel, "GET / HTTP/1.1" & CRLF &
                          "Connection: close " & Send);
   loop
      Ada.Streams.Read (Channel.All, Data, Offset);
      exit when Offset = 0;
      for I in 1 .. Offset loop
         Ada.Text_IO.Put (Character'Val (Data (I)));
      end loop;
   end loop;
end Vcv;
```

Listing 20: VCV Get Http Client

5.2 Task Oriented Concurrency Model

Concurrency in Ada uses tasks. Here is a small example to demonstrate this feature of the language that exploits a very interesting way of achieving concurrent tasks. If more detail is required, refer to [10]

5.2.1 Simple Concurrent Program

This is the simplest example in using tasks in Ada. The example was adapted from notes found in [6], and was also the first concurrent program I wrote for testing and understanding tasks, the first time around.

```
with Ada.Text_IO;

procedure Task_Types is
  task type A (Label : Character) is
    entry Start;
  end A;

  task body A is
  begin
    accept Start;

    for I in Integer range 1..10 loop
```

```
      Ada.Text_IO.Put_Line(Character'Image(Label) & " is working...");
      delay 0.4;
    end loop;
  end A;

  My_Task_01 : A(Label => 'A');
  My_Task_02 : A(Label => 'B');
  My_Task_03 : A(Label => 'C');

begin
  My_Task_01.Start;
  My_Task_02.Start;
  My_Task_03.Start;
end Task_Types;
```

Listing 21: Task Types in Ada

This procedure would have the following output:

```
[psyomn@aeolus concurrency 0]$ ./task_types
'A' is working...
'B' is working...
'C' is working...
'A' is working...
'B' is working...
'C' is working...
'A' is working...
'B' is working...
'C' is working...
'A' is working...
'C' is working...
'B' is working...
'A' is working...
'B' is working...
'C' is working...
'B' is working...
'C' is working...
'A' is working...
'A' is working...
'B' is working...
'C' is working...
'A' is working...
'B' is working...
'C' is working...
'A' is working...
'B' is working...
'C' is working...
'A' is working...
'B' is working...
'C' is working...
```

Listing 22: Output for the Task Types in Ada

5.3 Text Manipulation

We list some text manipulation example code in this section, if better understanding is required. First it would be better to familiarize oneself with the *String* type; one can refer to [9], and another excellent tutorial of Java vs Ada String type comparison [7].

5.3.1 Splitting

A common task one wants to do with strings is split them. Refer to this [5], to read upon string splitting. Notice that this was added on the GNAT platform.

5.3.2 Regular Expressions

Regular expressions don't quite exist in the Ada programming language. However the GNAT platform provides some libraries with similar effects. A simple example can be found at [4].

5.4 Random Notes

Some other miscellanious random notes, that might help in the future.

5.4.1 ADS / ADB file location

If the *gcc-ada* compiler is being used, the specification and body files can typically be found in the following directory

$$/usr/lib/gcc/\$MACHTYPE/\$GCC-VERSION/adainclude/$$

On another platform, namely *Lubuntu*, the absolute path included another directory right before the ada include one.

$$/usr/lib/gcc/\$MACHTYPE/\$GCC-VERSION/rts-native/adainclude/$$

5.4.2 Sanity Checking

I wanted to test out if things were working properly every now and then. Here are a few commands that I used in order to do this.

5.4.3 Using Telnet to conenct to a Http Server

You can use telnet to connect to a http server, and write the headers manually, and receive the actual response. This helps at understanding a little more how the protocol works, and was used in order to debug the server.

```
[psyomn@aeolus ~ 0]$ telnet www.concordia.ca 80
Trying 132.205.244.34...
Connected to www.concordia.ca.
Escape character is '^]'.
HEAD / Http/1.1
Host: www.concordia.ca
Connection: close

HTTP/1.1 200 OK
Date: Sun, 28 Apr 2013 23:08:49 GMT
Server: Apache
X-Powered-By: PHP/5.1.6
Vary: Accept-Encoding
Connection: close
```

```
Content-Type: text/html; charset=UTF-8

Connection closed by foreign host.
```

Listing 23: Telnet to an Http Server

5.4.4 Using ncat to dump packets to a binary file

On the various stages of debugging, *ncat* was used in order to dump the packets to a file, in order for inspection. This helped when I was facing trouble when the socket would send random garbage due to mistakes to the http servers, and have undesired consequences.

```
# First we start ncat on localhost, and make it listen at port 3000
[psyomn@aeolus ~ 0]$ ncat -l localhost 3000 > log.txt

# Then we telnet to ncat, at port 3000
[psyomn@aeolus ~ 0]$ telnet localhost 3000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello there.
These are messages that are sent to you via telnet
I hope you like them.
Connection closed by foreign host.

# The log has the messages sent. The log file can also store byte values, not
# just text messages, hence a means to see what the application is writing at
# said sockets.
[psyomn@aeolus ~ 0]$ cat log.txt
Hello there.
These are messages that are sent to you via telnet
I hope you like them.
```

Listing 24: Using ncat to dump packets

Here is an investigation example of data sent through sockets from a test script. This is a GET request.

```
[psyomn@aeolus 423 0]$ xxd /tmp/loggy
0000000: 4745 5420 2f20 4874 7470 2f31 2e31 0d0a  GET / Http/1.1..
0000010: 486f 7374 3a20 6c6f 6361 6c68 6f73 740d  Host: localhost.
0000020: 0a43 6f6e 6e65 6374 696f 6e3a 2063 6c6f  .Connection: clo
0000030: 7365 0d0a 0d0a                           se....
```

Listing 25: Using ncat a test script and xxd to hex dump

You can notice the CRLFs at the end of each request (that is the two bytes on the left, 0x0A, and 0x0D respectively).

And here is a small, priceless one-liner that has helped me debug some of my code during this case study. It uses ncat, and pipes the output to xxd, which in turn makes a hex dump.

```
[psyomn@aeolus 423 0]$ for (( ;; )) do ncat -l localhost 8080 | xxd; done
0000000: 0100 0000 0500 0000 4865 6c6c 6f          ........Hello
0000000: 0100 0000 0100 0000 48                    ........H
0000000: 0100 0000 0100 0000 48                    ........H
0000000: 4745 5420 2f20 4874 7470 2f31 2e31 0d0a  GET / Http/1.1..
0000010: 486f 7374 3a20 6c6f 6361 6c68 6f73 740d  Host: localhost.
0000020: 0a43 6f6e 6e65 6374 696f 6e3a 2063 6c6f  .Connection: clo
0000030: 7365 0d0a 0d0a                           se....
0000000: 2a00 0000 0100 0000 0100 0000 42          *...........B
0000000: 2a00 0000 0100 0000 0100 0000 42          *...........B
0000000: 2a00 0000                                 *...
0000000: 2a00 0000 2a00 0000 2a00 0000             *...*...*...
```

Listing 26: One liner to hex dump dynamically

References

[1] GNAT Sockets examples, 2013. [Online Fri May 3 2013].

[2] adacore. 2013. http://docs.adacore.com/gnat-unw-docs/html/gnat_ugn_12.html.

[3] adacore. AUnit, Ada Testing, 2013. http://docs.adacore.com/aunit-docs/aunit.html#AUnit_002eSimple_005fTest_005fCases.

[4] Rosetta Code. Regular Expressions for Ada, 2013. http://rosettacode.org/wiki/Regular_expressions#Ada.

[5] commons.ada.cx. Substrings using Gnat.String_Split, 2013. `http://bit.ly/107SVuN`.

[6] M. J. Feldman. Chapter 15: Introduction to Concurrent Programming, 1996. http://www.seas.gwu.edu/ mfeldman/cs2book/chap15.html.

[7] Edward G. (Ned) Okie. Differences Between Strings in Ada and Java, 2013. http://www.radford.edu/ nokie/classes/320/strings.html.

[8] The Internet Society. 1999. http://tools.ietf.org/pdf/rfc2616.pdf.

[9] Wikibooks. Ada Programming: String Type, 2013. http://en.wikibooks.org/wiki/Ada_Programming/Strings.

[10] Wikibooks. Ada Programming: Tasking, 2013. http://en.wikibooks.org/wiki/Ada_Programming/Tasking.