Ada Web Server

Brent Seidel
Phoenix, AZ

2018-09-18

1 Synopsis

This Ada web server is a simple server designed to serve a few files and provide an interface to embedded systems. It is not intended to be a highly flexible high performance server–there are plenty of systems that provide that. It also does not currently support HTTPS or any authentication methods, so it should not be used in an application that needs security. However, it does not, by default refer to any external servers so it can be used on an isolated network.

This document describes the configuration of and modifications to the Ada web server. The amount of customization needed depends on the application. For some applications, no modifications of the software may be needed. In other applications, major modifications are needed.

Note: As part of a restructuring, all packages have been prefixed with `bbs.` This is part of an effort to allow the code to be used as a library instead of being integrated into the target application.

2 Installation

This repository is stand alone and can be installed without any other repositories. It is written in Ada 2012 and built using the GNAT Ada tool set. The following Ada packages are used in various places throughout the system:

- `Ada.Characters.Latin_1`
- `Ada.Containers.Indefinite_Hashed_Maps`
- `Ada.Sequential_IO`
- `Ada.Strings`
- `Ada.Strings.Equal_Case_Insensitive`
- `Ada.Strings.Fixed`
- `Ada.Strings.Hash_Case_Insensitive`
- `Ada.Strings.Unbounded`
- `Ada.Text_IO`

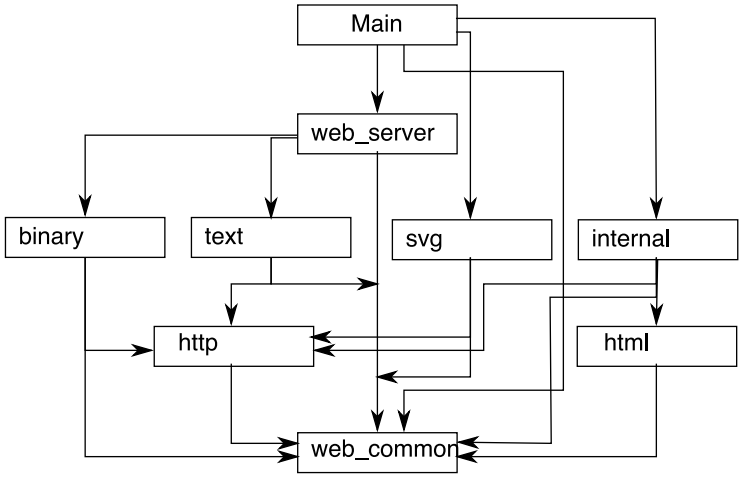

Figure 1: "With" Structure of Packages.
"With" to packages outside of this repository (Ada and GNAT standard packages) are ignored. "With" from spec and body treated the same.

- `Ada.Text_IO.Unbounded_IO`
- `GNAT.Sockets`

3 Software Structure

The software consists of a number of packages, each with a spec and a body. See figure 1.

3.1 `main.adb`

This is the main routine and it simply starts the web server. If other initializations are needed, they could be added or called from here.

3.2 `web_common`

This package contains a number of items that are available for use in other packages. Items are put here instead of in `web_server` in order to avoid circular dependancies. The primary item from `web_common` that is expected to be used by user software is the `params` hash map. The package is instantiated as follows:

```
package params is new Ada.Containers.Indefinite_Hashed_Maps
  (Element_Type => String,
   Key_Type => String,
   Hash => Ada.Strings.Hash_Case_Insensitive,
   Equivalent_Keys => Ada.Strings.Equal_Case_Insensitive);
```

If your code needs access to parameters passed in a GET or POST request, this is how you will get access to those parameters. There are some examples where this is used in the `internal` and `svg` packages. Both parameters and headers use this hash map type.

Another hash map type is instantiated for internally generated items. First, every internal item procedure has the same signature so an access type can be created as:

```
--
--  Define a type for the user procedures.  This is for a map used to map
--  the internal procedures.  The parameters are:
--    s - The stream to write output to.
--    p - Any passed parameters from the HTTP request
--    h - HTTP request headers.
--
type user_proc is access procedure (s : GNAT.Sockets.Stream_Access;
                                    p : params.Map;
                                    h : params.Map);
```

Then the hash map package is instantiated as:

```
--
--  Instantiate a hashed map indexed by a string and containing procedure
--  accesses.  Used as a table to identify which internal procedure to call
--  for internal requests.
--
package proc_tables is new Ada.Containers.Indefinite_Hashed_Maps
  (Element_Type => user_proc,
   Key_Type => String,
   Hash => Ada.Strings.Hash_Case_Insensitive,
   Equivalent_Keys => Ada.Strings.Equal_Case_Insensitive);
```

This map is populated and used in the `web_server` package.

3.3 web_server

This package contains the main web server software. Since this package calls the user code, any attempt by user code to call items in this package will create a circular dependency. Items that may be needed by user code should be in `web_common`. The item in this package that gets called with user code is the following procedure:

```
--
--  This is the web server.  In initializes the network interface and enters
--  an infinite loop processing requests.  The passed paraemters are:
--    internals   - A map of the names of internal item and procedure
--    config_name - The name of the configuration file
--    port        - The port to listen on
--
--  Note that this procedure never returns.  Eventually code should be added
--       to shut the server down and exit.  It may also be turned into a task
```

```
--       which may allow multiple servers to be run simultaneously.
--
procedure server(internals : bbs.web_common.proc_tables.Map;
                 config_name : String;
                 port : GNAT.Sockets.Port_Type);
```

This procedure should be called after all other initialization is done. It starts the web server and does not return. At some point, it may get turned into a task so that other processing in the main task can proceed along with it.

3.4 html

This package contains routines to support the generation of HTML. The visible routines are:

- `html_head(s : GNAT.Sockets.Stream_Access; title : String)`
  Generate a simple HTML heading with the specified title.
- `html_head(s : GNAT.Sockets.Stream_Access; title : String; style : String)`
  Generate a simple HTML heading with the specified title and style sheet.
- `html_end(s : GNAT.Sockets.Stream_Access; name : String)`
  Generate an ending for an HTML item using the file specified in `name`.

3.5 http

This package contains routines to support HTTP. Currently GET and POST methods are supported with some minimal support for the OPTIONS method. GET methods are supported for all items while POST methods are only supported for internally generated items. If a file or an internally generated item with no parameters is being served, any supplied parameters are just ignored. Some of these routines are intended for use by user code and some are not. The visible routines are:

- `ok(s : GNAT.Sockets.Stream_Access; txt: String)`
  Return code 200 OK for normal cases.
- `not_found(s : GNAT.Sockets.Stream_Access; item: String)`
  Return code 404 NOT FOUND for when the requested item is not in the directory.
- `internal_error(s : GNAT.Sockets.Stream_Access; file: String)`
  Return code 500 INTERNAL SERVER ERROR generally when unable to open the file for the specified item. This means an item has been added to `config.txt` without adding the file to the system.
- `not_implemented_int(s : GNAT.Sockets.Stream_Access; item: String)`
  Return code 501 NOT IMPLEMENTED for a request for an internally generated item that is not yet implemented.
- `read_headers(s : GNAT.Sockets.Stream_Access;`
  `method : out request_type;`
  `item : out Ada.Strings.Unbounded.Unbounded_String;`
  `params : in out web_common.params.Map)`

The `read_headers` procedure handles both GET, POST, and OPTIONS request and returns any passed parameters. Returned values will be the requested item in `item` and a dictionary containing the parameters in `params`. If there are no parameters, the dictionary will be empty. This routine is called by the web server to read the headers of the HTTP request. By the time any user code is called, the headers have already been read.

3.6 internal

This package contains routines to generate HTML or XML for internally generated items. The generated HTML or XML is written to the `Stream_Access` that needs to be passed in. The proper HTTP headers also need to be written. None of the routines in this package are required and can be changed around as needed. They do provide some examples of how to generate items. The internal routines can have access to parameters passed in the GET or POST request. If no parameters were passed, the dictionary is empty. The internal routines cannot distinguish between a GET or a POST request. The visible routines are:

- ```
  procedure xml_count(s : GNAT.Sockets.Stream_Access;
                      h : web_common.params.Map;
                      p : web_common.params.Map);
  ```

  Sends the count of transactions as an xml message.

- ```
  procedure html_show_config(s : GNAT.Sockets.Stream_Access;
                             h : web_common.params.Map;
                             p : web_common.params.Map);
  ```

  Sends the configuration data as a HTML table.

- ```
  procedure target(s : GNAT.Sockets.Stream_Access;
                   h : web_common.params.Map;
                   p : web_common.params.Map);
  ```

  Sends the parameters provided as a HTML table.

- ```
  procedure html_reload_config(s : GNAT.Sockets.Stream_Access;
                               h : web_common.params.Map;
                               p : web_common.params.Map);
  ```

  Request that the configuration file be reloaded. This can be useful during development and debugging.

3.7 svg

This package contains routines to generate SVG for internally generated items. The generated SVG is written to the `Stream_Access` that needs to be passed in. The proper HTTP headers also need to be written. The visible routines are:

- ```
  procedure thermometer(s : GNAT.Sockets.Stream_Access;
                        h : web_common.params.Map;
                        p : web_common.params.Map);
  ```
  ```

Send SVG code to display a thermometer showing the value parameter. This procedure handles getting and checking the parameters. The following parameters are supported:

- `min` – The minimum displayed value
- `max` – The maximum displayed value
- `value` – The value to display.

The value is clamped to be between `min` and `max`. If any exceptions occur in parsing the parameters or if `min` is greater than `max`, a red "X" will be presented as the graphic to indicate an error condition. Using the default configuration, this could be requested as `/Thermometer?min=0&max=100&value=50`

- ```
  procedure dial(s : GNAT.Sockets.Stream_Access;
                 h : web_common.params.Map;
                 p : web_common.params.Map);
  ```

  Send SVG code to display a round dial with a pointer to the appropriate value. The following parameters are supported:

  - `min` – The minimum displayed value
  - `max` – The maximum displayed value
  - `value` – The value to display.

  The value is clamped to be between `min` and `max`. If any exceptions occur in parsing the parameters or if `min` is greater than `max`, a red "X" will be presented as the graphic to indicate an error condition. Using the default configuration, this could be requested as `/Dial?min=0&max=100&value=50`

3.8 text

This package consists of a spec and body and is used to support the serving of text files. The visible routines are:

- ```
  send_file_with_headers(s : GNAT.Sockets.Stream_Access;
                         mime : String; name : String)
  ```

  This procedure sends a text file to the client with headers. The file name is contained in the parameter `name` and the MIME type of the file is contained in the parameter `mime`.

- ```
  send_file_without_headers(s : GNAT.Sockets.Stream_Access;
                            name : String)
  ```

  This procedure sends a text file to the client without headers. The file name is contained in the parameter `name`. No HTTP headers are sent.
  ```

3.9 `binary`

This package consists of a spec and body and is used to support the serving of binary files. The one visible routine is:

- `send_file_with_headers(s : GNAT.Sockets.Stream_Access;`
  `mime : String; name : String);`

  This procedure sends a binary file to the client with headers. The file name is contained in the parameter `name` and the MIME type of the file is contained in the parameter `mime`.

4 Configuration

Generally before being used, the web server needs to be configured. An example configuration is included and may be modified as needed.

4.1 Configuration File

The primary means of configuration is the `config.txt` file in the repository root. This file is used to translate from the URL requested to the actual page served. Both external files and internally generated responses are supported.

The format of the configuration file is fairly simple. A line that has a pound sign, "#" (octothorp) as the first character is a comment. Non comment lines consist of three fields separated by a single space.

- The first field is the requested URL minus the server specification. Due to the nature of URLs, the first character will always be a slash, "/". The web server uses a simple dictionary data structure for the URLs so the structure is technically flat. However, any sort of hierarchical structure can be simulated.
- The second field identifies the item to be served. It may be a file or it may be an arbitrary string to identify an internally generated item. Files served are passed unchanged and both text and binary files are supported.
- The third field identifies the MIME type of the file being served. If the third field is "`internal`", the item being served will be generated internally. The value of the second field is used to select the proper internal routine to call.

4.2 Modifying the Software

To change the port used by the server, specify a different value when calling `web_server.server`.

To change the number of tasks (threads) available for serving, change the `num_handlers` constant in `web_server.ads`. The default value is 10, which should be adequate. If memory is tight, it can be reduced. If higher performance is needed, this number can be increased.

5 Modifications

To add or change internally generated items, the code will need to be modified and recompiled. In both cases, the first place to look in the software is the `build_internal_map` procedure in the `web_server.adb` file.

5.1 Modifications to Existing Items

First, identify the routine to modify by looking at the `build_internal_map` procedure. Then the appropriate files can be edited and the routine located. Once the routine is located, any necessary modifications can be made.

5.2 Adding New Items

The very first thing is to decide what you can your item to do. The existing software has examples of routines that generate HTML, XML, and SVG. These can be used as models. These are contained in the `internal` (for HTML and XML) and `svg` (for SVG) packages. If you need to interface with other hardware or software, it will probably be best to add new packages for these interfaces.