EMS Documentation

version 1.0

Thyag Sundaramoorthy

April 06, 2017

Contents

realHTML4Natural Documentation

In this Documentation you will find how the project is build, how you are getting started and a full list of command that are available in the templates.

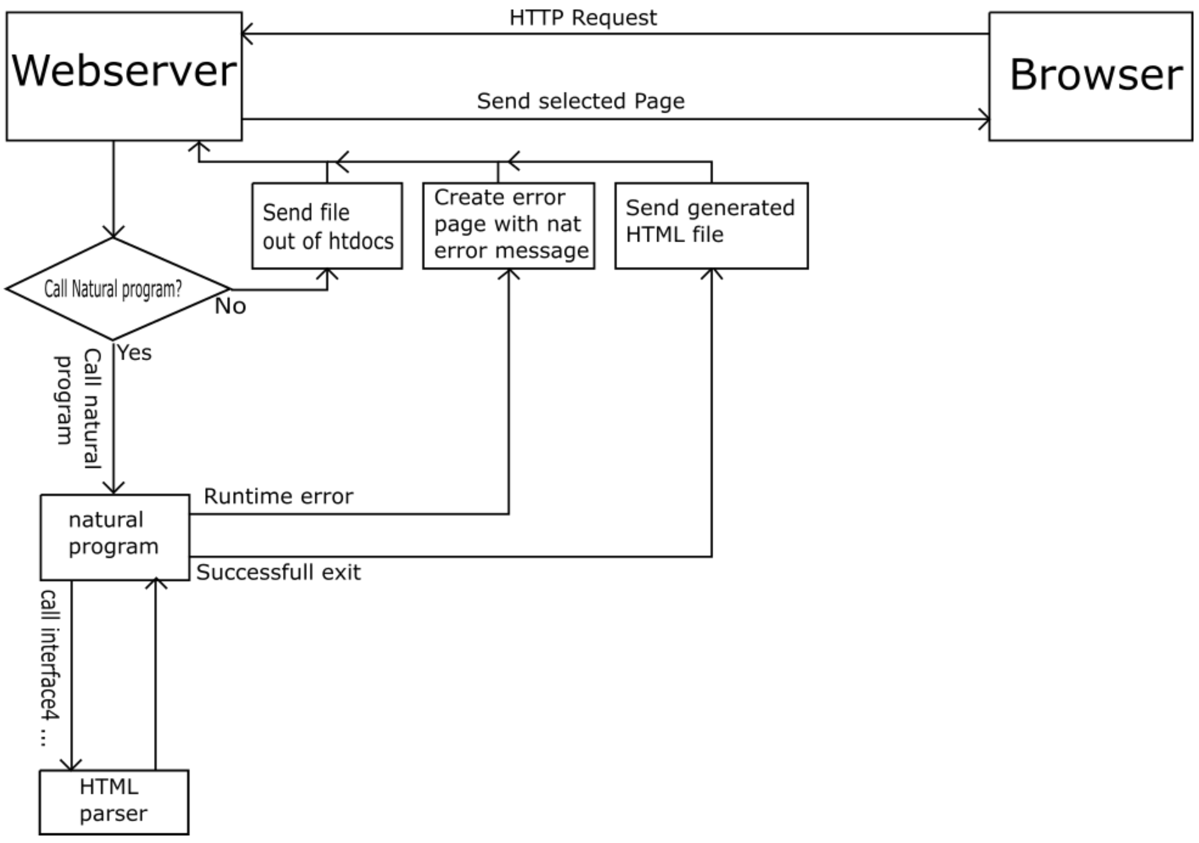
What is realHTML4Natural?

RealHTML4Natural is a web framework for Natural. It allows you to write your own HTML code where you don't have to use predefined controls. It does not rely on one big Natural Session (like N4Aj), every HTTP request (when the route it is defined to call a Natural subprogram) is a new Natural session. So it is thread save and you can make real asynchronous request with Java script. Your skills are the limit here.

Motivation and idea

I have worked a lot with the Python Framework Flask which is using the template engine Jinja and I thought: "Why can't you do this kind of web programming with Natural?" So I sat down and began to wrote my own framework where you have full control over the behavior of the web server and the HTML page.

How does it work?



The complete framework is based on the Natural native interfaces (NNI) which allows you to create a Natural process from your own program without using the Natural binary directly. The interface to the HTML parser is based on the interface4 API from the Natural native interfaces. With this API you can call external programs out of Natural and hand over parameters. Then you can read attributes like value, length or type, unfortunately not there names. So I have written a program which parses the local data areas (LDAs) from Natural and can make a connection between the parameters and the names based one the position, length and type. Based on this informations you can write a HTML template which is just HTML code with placeholder for the Natural variables. The HTML parser will then parse the template an replace these placeholder with the corresponding values. But how does the web server know when to call a Natural program? That is quite simple. In a configuration file you describe on which URL which program is called. When nothing is defined the server will deliver the requested file out of a htdocs folder. When no file is requested or the file is not found a "Page not found" (HTTP error 400) is delivered.

Guide:

Build process and Installation

Build

Unfortunately the build process is currently very manual. The libraries in the folder "libs" are build with the Makefile in the root directory. Except the library "mxml", it has its own makefile. The rest of the libraries outside of "libs" has there own makefiles. So you have to edit the following Makefiles and change the $CC variable to your compiler (where ./ is the root folder from the repository):

- ./web_server/Makefile
- ./var2names/Makefile
- ./libs/mxml/Makefile
- ./Makefile
- ./Makedyn

You also have to have the shared object "libnatural.so" from the Software AG in your "LIBPATH" Environment variable. It should be located in your Natural bin directory. Now you should be able to just type in "make all" in the root folder and the complete project will be build in the right order.

Installation

There is no real installation yet but there are just two core binaries to copy:

- ./jinja2.so
- ./web_server/miniweb

"jinja.so" is the shared object which provides the INTERFACE 4 functions "genpage" and "flipbuf". It must be mentioned in your NATUSER Env-var.

"miniweb" is the web server. It is good practice to put it into your "/usr/bin/" folder so it is in your PATH variable and can be found by just typing "miniweb". The configuration files of the server you can put everywhere. I am a fan of creating a directory in "/etc" which is called "realHTML4Natural" and putting all the configuration stuff in there.

Webserver

The webserver is based on the open source project miniweb (http://www.http://miniweb.sourceforge.net/) but this version is heavily modified.

This version can parse POST Requests where data is urlencoded and not an multipart message, read cookies and set new ones and of course calling Natural programs.

There are two configuration files that miniweb can use. A general configuration file with how the server should be configured and a file which controls when which Natural program is called.

Server configuration

The path to this file is stored in the environment variable "WEBCONFIG". It can contain multiple environments. For example a configuration for test and one for production.

Example

```
<miniweb>
    <environment type="test">
        <routes>/home/tom/C/realHtml4Natural/web_server/routes.xml</routes>
        <htdocs>/home/tom/C/realHtml4Natural/web_server/htdocs/</htdocs>
        <templates>/home/tom/C/realHtml4Natural/web_server/templates/</templates>
        <natsourcepath>/natural/natsrc/</natsourcepath>
        <port>1024</port>
        <debug>true</debug>
        <naterror>nat_error.html</naterror>
        <ldaerror></ldaerror>
        <parsererror></parsererror>
        <natparms></natparms>
    </environment>
    <environment type="production">
        <routes>/home/tom/C/realHtml4Natural/web_server/routes_prod.xml</routes>
        <htdocs>/home/tom/C/realHtml4Natural/web_server/htdocs/</htdocs>
        <templates>/home/tom/C/realHtml4Natural/web_server/templates/</templates>
    </environment>
</miniweb>
```

Explanation enviroment tag

With the environment tag you can setup multiple environments in which the server is running. It takes exactly one argument. The type attribute with the name of the environment.

Explanation enviroment childs

entry	explanation	default Value	required
routes	the path to the routes configuration file	None	yes
htdocs	the path to the htdocs folder	None	yes
templates	the path to the template folder	None	yes
natsourcepath	path to the Natural sources (must be the root directory)	None	yes
port	port on which the webserver should listen	80	no
debug	decide wether an error occurs to render a custom 500 page with the error message or a generic 500 error page	false	no
naterror	the template that should be rendered when Natural error occurs (debug must set to true)	None	no
ldaerror	the template that should be rendered when error occurs while parsing the LDA (debug must set to true)	None	no
parsererror	the template that should be rendered when error occurs while handling the template (debug must set to true)	None	no
natparms	parameter that will be passed through to Natural (the same as if you were calling Natural from the binary)	None	no

Route configuration

In the route configuration file you enter the Natural program which should run when a specific route is called.

Example

```
<realHtml>
    <route path="notizen">
        <programm>SHNOTIZ</programm>
        <alias></alias>
        <return></return>
        <library>TOMENGE</library>
    </route>
    <route path="new_notiz">
        <programm>STNOTIZ</programm>
        <library>TOMENGE</library>
    </route>
    <route path="delete_notiz">
        <programm>SDENOTIZ</programm>
        <library>TOMENGE</library>
    </route>
</realHtml>
```

Route tag

The route tag takes exactly one parameter. The path attribute. That is the URL which is requested on the server without the first slash.

Explanation route childs

entry	explanation	default Value	required
program	the Natural program that will be called	None	no
library	the library to logon	None	no
alias	a file in the htdocs folder that will be delivered instead of calling the Natural program	None	no
return	a HTTP code that will be returned instead of a file from htdocs or calling a Natural program	None	no

Miniweb usage

-h	display this help screen
-v	log status/error info
-p	specifiy http port [default 80]
-r	specify http document directory [default htdocs]
-l	specify log file
-m	specifiy max clients [default 32]
-M	specifiy max clients per IP
-s	specifiy download speed limit in KB/s [default: none]
-n	disallow multi-part download [default: allow]
-d	disallow directory listing [default ON]
--environment	environment to load [default: none]

Tomcat Connector

Overview

When you want to use the framework with a new or current tomcat installation you can use the TomcatConnector. It consists of three parts:

- A tomcat servlet which handles the requests
- A library which provides helper function for the servlet
- A JNI (Java Native Interface) Library which calls Natural

Tomcat Servlet

The purpose of the tomcat servlet is to handle the incoming request, get the configuration and call Natural. You have to put the .class file which is located in the servlet folder into your webapps folder under "ROOT/WEB-INF/classes". When one of the folder does not exist you can just create it.

To get the servlet to get known from the tomcat you have to edit your web.xml configuration file. It should be located in your "conf" folder in your tomcat installation directory. Just add the following at the end:

```
<servlet>
   <servlet-name>realHTMLServlet</servlet-name>
   <servlet-class>realHTMLServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>realHTMLServlet</servlet-name>
   <url-pattern>/realHTML4Natural/*</url-pattern>
</servlet-mapping>
```

The servlet tag is there for getting to know the java class (located in the servlet-class tag) of the servlet to the tomcat. With the servlet-name tag under servlet you give the servlet a name for later configuration.

With the servlet-mapping tag you map the servlet to a specific path. In your case every url with "/realHTML4Natural/" at the beginning will call the servlet.

Servlet Library

The library you can find in the "realHTML/tomcat/connector" library. In there you find utility function for parsing the xml configuration files (see section "Connector configuration"), the loading of the JNI library and the call to the C-function in the JNI library. This jar file must be placed in the "lib" folder of your tomcat installation.

Why is this so?

A JNI library can only loaded once in one Java VM. Because the servlets runs in threads you are allowed to load her once. The jar files in the lib directory gets loaded on the start up process of the tomcat and so gets the JNI library loaded and you can use the functions in a servlet.

JNI library

In the JNI library happens all the magic. Here the parameter for the natural gets generated, a new process gets spawned and natural runs in this process. The library now waits until this process has exited and returns to the servlet.

Installation

The installation is rather simple:

- run the Makefile with the target "all"
- copy realHTMLconnector.jar into the lib directory of your tomcat installation
- copy the *.class file into "<path to your webapps folder>/ROOT/WEB-INF/classes/"
- copy the librealHTMLconnector.so into a folder that you choose
- edit the tomcat web.xml as show in Tomcat Servlet
- edit your setenv.sh in the bin directory and add the following lines:

```
export realHTMLconfiguration=<path to your config file for the framework>
export NATUSER=$NATUSER:<path to your ralHTMLNatural.so>
export LIBPATH=$LIBPATH:<path to the directory where your libnatural.so lies>
```

- edit your catalina.sh in the bin directory and search for a line where the variable "JAVA_OPTS" get set.
- append it with the "-Djava.library.path=<path to the directory which contains the JNI library>". For example:

```
JAVA_OPTS="$JAVA_OPTS $JSSE_OPTS -Djava.library.path=/realHTML4Natural/bin/"
```

- restart tomcat

Connector configuration

Placeholder

Natural

Parameter Data Area

A parameter data area like this has to been defined. You don't have to take this exact names but the type and length has to match.

```
1 VALUES_REQ (A2024/1:20)
1 VALUE_LENGTH (I4/1:20)
1 KEYS_REQ (A100/1:20)
1 PARM_COUNT (I4)
1 REQ_TYPE (A5)
1 REQ_FILE (A100)
1 REQ_SETTINGS (A) DYNAMIC
```

Explanation parameter data area

variable name	explanation
value_req	the variables value_req and keys_req belongs together. These two array form together a Key/Value Pair. "value_req" includes the values from the GET or POST request
value_length	the real length of the value in occurrence n
keys_req	the key to the value in occurrence n
parm_count	includes own much entry in value_req/value_length/keys_req are set
req_type	which HTTP request was received
req_file	just a internal variable. Contains the file which will be delivered to the browser
req_settings	just a long string which contains settings from the webserver configuration for the html parser

Variables for the template

You have to create a page group in an external local data area (LDA). Every variable under this group is available in the template. You can redefine variables and create new subgroups.

Example:

```
define data local
1 page
   2 daten (A128/1:200)
   2 isnnummer (A29/1:200)
   2 maxdatenid (I4)
end-define
```

Supported data types

Guide:

type	length
Integer (I)	1/2/4
Logical (L)	----
Alphanumeric (A)	every/Dynamic
Numeric (N) (Floating points are not supported yet)	every/Dynamic
Unicode (U)	every/Dynamic

Every type can be also defined as 1D/2D and 3D arrays but then the dynamic length is not supported. This is due to a bug in the INTERFACE 4 API from the SAG.

Other

You also need two alphanumeric variable with the length of 20. One of them has to contain the LDA name which you are using to store the page group and the second must contain the name of the template you want to use.

Example

```
define data
    local using your_lda
    local
        1 realhtml_ldaname(A20)
        1 realhtml_template(A20)
 end-define

 move "your_lda" to realhtml_ldaname
 move "the_template.html" to realhtml_template
 end
```

The "flipbuf" function

The "flipbuf" function shift a specific number of digits from the beginning of the buffer to the end.

For example you want to search for a ISN number with a POST argument. The number in the POST value will be standing on the beginning of the buffer but the ISN number is sitting on the right. With the "value_length" (from the pda. See Parameter Data Area) variable you know how long the number is.

Arguments

number	type	purpose
1	Numeric (N)	input buffer
2	Numeric (N)	output buffer
3	Integer (I4)	length of the number

Example

```
define data local
   1 post_isn(N19) inti <"1200000000000000000"> /*The value from the post argument
   1 post_length (I4) init <2>
   1 right_isn(N19)
end-define

call interface4 "flipbuf" post_isn right_isn post_length
* Now right_isn contains the following: "0000000000000000012"
end
```

The "setcookie" function

Nothing to see here. Currently under development. This function will be used to set new cookies and send them to the browser.

The "genpage" function

The "genpage" function generate the html page out of the template.

Arguments

number	type	purpose
1	Alphanumeric (A20)	name of the LDA
2	Alphanumeric (A20)	name of the template
3	Alphanumeric (A100)	file path which the template engine should write to
4	Alphanumeric (A Dynamic)	settings string
5	see Supported data types	the page group from your LDA

Example

```
define data local
   1 lda_name (A20) init <"mylda">
   1 template_name (A20) init <"mytemplate">
   1 output_file (A100) init <"/tmp/outout.html">
   1 settings (A) DYNAMIC init <"templatepath=/tmp/templates/;debug=true">
   1 page /* this would be in your external lda
      2 field-1 (A10)
      2 field-2 (A10)
end-define

call interface4 "genpage" lda_name template_name output_file settings page
end
```

Complete skeleton

The subprogram

```
define data
    parameter using reqinfos /*request information
    local using testlda
end-define

move "testlda" to realhtml_ldaname
move "testtemplate.html" to realhtml_template

/* your code goes here

call interface4 "genpage" realhtml_ldaname realhtml_template
                    req_file req_settings page
```

The LDA

```
define data local
   1 realhtml_ldaname (A20)
   1 realhtml_template (A20)
```

Guide:

```
    1 page
        2 field-1 (A10)
        2 field-2 (A20)
end-define
```

The PDA

```
DEFINE DATA PARAMETER
    1 VALUES_REQ (A2024/1:20)
    1 VALUE_LENGTH (I4/1:20)
    1 REQ_KEYS (A100/1:20)
    1 PARM_COUNT (I4)
    1 REQ_TYPE (A5)
    1 REQ_FILE (A100)
    1 REQ_SETTINGS (A) DYNAMIC
END-DEFINE
```

Html Parser

Overview

The syntax of the templates are inspirited from the template engine jinja which is written in Python.

In contrast to natural4ajax you will write here your own HTML code. In your HTML code you can use two types of blocks to refer to variables or check them if they equals to a specific condition.

Here is an simple example:

```
 <html>
     <head>
         <title>Hello World</title>
     </head>
     <body>
         <table>
             <thead>
                 <th>id</th>
                 <th>name</th>
                 <th>lastname</th>
            </thead>
            <tbody>
            {% for user in username %}
                <tr>
                     <td>{{ isnnumber[loop.i]</tr>
                     <td>{{ user }}</tr>
                     <td>{{ firstname[loop.i] }}</tr>
                </tr>
            {% end-for %}
            </tbody>
        </table>
    </body>
</html>
```

With the LDA:

```
define data local
1 page
   2 isnnumber (N19/1:20)
   2 username (A20/1:20)
   2 firstname (A20/1:20)
end-define
```

There are two types of blocks. First the variable block ({{ ... }}) which will just be replaced with the value of the variable and second the command block ({% ... %}). The command block reacts like a if or for block in natural. It have a command line (the first one) where you specify the break condition and a end line where the block ends.

Variable blocks

As in the section Variables for the template explained you have available every variable that is defined under the page group in your LDA. If a variable is not known the HTML parser will exit with an error message and will be render a error page instead. Currently (Version 1.0) it is not possible to add two variable blocks in one line. The parser would just print out the complete block without parsing it. I am working on this problem.

Plain variable:

Let's assume that you have defined a variable with the name "varname" which has the type alphanumeric with the length of 20 bytes. When you just write {{ varname }} it would be replaced with the value. But you can also handle it as an array by adding square brackets after the name: {{ varname[0] }}. Now you would get the value saved at the position zero of the variable. If you have spaces at the end the parser will automatically trim the value to the last non space character.

Arrays:

Now let's chance the variable to array with 10 occurrences. In this situation the behavior of the variable block change a little bit. It will be still replace the complete block with the value but when you don't specify a index a JSON string will be generated and printed out. But when you don't what that you can specify a index in square brackets ({{ varname[0] }}). Even here you can access the individual characters with a second index ({{ varname[0][1] }}).

Example

The variable: 2 varname(A20/1:10)

The HTML-code: {{ varname }}

The output: ['value 1', 'value 2', 'value 3', 'value 4', 'value 5', 'value 6', 'value 7', 'value 8', 'value 9', 'value 10']

The HTML-code: {{ varname[0] }}

The output: value 1

The HTML-code: {{ varname[0][1] }}

The output: a

Did you noticed that I accessed the first occurrence of the array with the index of zero even we declared it with an offset of one? That is because the HTML parser begins counting from zero. So whatever your offset is the first occurrence will be zero.

The explanation above also applies to 2D or 3D arrays. You just add more square brackets after the variable name to access the different dimensions.

Command blocks

Command blocks are control structures which influences the render process of the template. So you can iterate through arrays, or render different HTML code when a condition is true.

For

Can loop over each entry in a array. For example the user list from above: Overview. In the for loop you can access a special variable: loop.i. This variable contains the current iteration of the loop with an index of zero. In the loop you can use the {% break %} command to break out of the loop and the {% continue %} command to go into the next iteration of the loop.

If

The if statement is based on the Python if. But this one have much less functionality (at least in version 1.0). So you can compare variables with each other or a hard coded value. With the `{% else %}` command you can switch to the else branch of the if statement.

Here is an example where we just want five iterations through the array "varname" from the Array Example.

Example

```
<ul>
{% for entry in varname %}
    {% if loop.i == 4 %}
        {% break %}
    {% else %}
        <li>{{ entry }}</li>
    {% end-if %}
{% end-for %}
</ul>
```

printVars

This function is just for debugging purposes. It will print out a pretty formated list of the defined variables with there corresponding values. It is recommended to wrap this command in the HTML tag `<pre>` so you can read it in your browser.

Example

`<pre>{% printVars() %}</pre>`

typeof

Prints out the type of the variable which is given as a argument.

Example

`{% typeof(varname) %}`

import

Imports another template that is provided after the import statement with quotation marks. This path is relative to the template_path defined in the Server configuration. The new template is parsed with the variable scope of the root template. It is planed to provide a second parameter which makes it possible to call a second natural program for this template.

Example

`{% import "navbar.html" %}`

macro

Macros are like function you define them in a block and call them later on. You can define two types of parameter required and optional. The optional ones must be defined at the end and have a default value.

Example

```
{% macro renderButton(btn_text, btn_type="submit") %}
    <button type="{{ btn_type }}">btn_text</button>
{% end-macro %}

{% renderButton("Hello World", "reset") %}
{% renderButton("Hello World") %}
```

Output:

```
<button type="reset">Hello World</button>
<button type="submit">Hello World</button>
```

Workflow

Placeholder

Indices and tables

- genindex
- search