Paper 3113-2019

REST Easier with SAS®: Using the LUA Procedure to Simplify REST API Interactions

Steven Major, SAS Institute Inc.

ABSTRACT

Making HTTP requests to interface with a REST API is a common necessity in modern programming. Doing this from within a SAS® program can be cumbersome. Making a simple get request requires the user to write header and body files, parse any URL parameters, and pass this information along to PROC HTTP. Authentication further complicates the problem. Once requests are made, the response needs to be somehow read in a way that is useful to the SAS programmer. This process often requires that the user be able to parse JavaScript Object Notation (JSON) strings to extract data before storing them in a SAS data set or local database. In this paper, we show how users can create Lua modules that simplify these tasks and, by using PROC LUA, how you can use these modules to easily interface with a REST API from SAS. We also show how debugging functionality can be worked into these modules and used to troubleshoot request errors. As a specific example, it is shown how to retrieve online climate data from the National Oceanic and Atmospheric Administration.

INTRODUCTION

Representational State Transfer (REST) is a web service architecture which is commonly used to provide an application program interface (API) – a method of communication between networked computers and a host system. Joseph Henry's 2016 paper on the use of the HTTP procedure to make requests to a RESTful web service (https://support.sas.com/resources/papers/proceedings16/SAS6363-2016.pdf) is an excellent introduction to the topic for readers who are not familiar.

This paper assumes you are familiar with the basic concepts of RESTful APIs and HTTP. Using the LUA procedure, it is shown how you can create an easy-to-use interface between SAS and a web service. Lua is a scripting language that you can use from within SAS by through the LUA Procedure and offers some distinct advantages to the SAS Macro language in this context. A brief summary of the LUA procedure is provided.

This paper first develops a general REST service Lua module to show how to simplify making HTTP requests in general. To show how you can leverage this for a specific web service, a small sample interface to the 'Climate Data Online' web service maintained by The National Centers for Environmental Information (NCEI) is also presented. In this example, it is shown how you can easily pull in zip code specific daily temperature data from the National Oceanic and Atmospheric Administration (NOAA) into a SAS dataset.

A BRIEF OVERVIEW OF THE LUA PROCEDURE

Traditionally, the SAS macro language has been the scripting tool of choice for SAS developers, and if you are familiar with the unique syntax of SAS macros, you may be hesitant to take on the task of learning a new tool to accomplish the same goal. However, you should consider the advantages of the Lua programming language before deciding

whether or not to invest time in learning it. First and foremost, Lua is very straightforward and easy to learn. With the help of a few bits of sample code you can generally be up writing code in Lua very quickly. Lua supports functions with multiple return arguments, data structures (everything in the macro language is a string) and associative arrays; in other words, it is a modern programming language. Finally, debugging Lua code is very straightforward. For the purposes of interacting with RESTful APIs, the Lua table is a very big plus, as it allows for an easy way to parse JSON, which is a very common way to receive data from an HTTP request.

Two very good papers for getting started with programming in Lua are Paul Tomas' 2015 paper: 'Driving SAS with Lua' (https://support.sas.com/resources/papers/proceedings15/SAS1561-2015.pdf) and the Lua reference manual available from https://www.lua.org/. The Lua concepts most relevant to this paper are discussed below.

LUA BASICS

The most basic way to use the LUA Procedure is to place your Lua code in a submit block. The following example is meant to demonstrate some basic Lua functionality.

```
PROC LUA restart;
submit;
   -- Comments can be written like this
   --[[ Or you can write block
        Comments like this
   ]]--

   -- Everyone's favorite first bit of code:
   print("Hello World")

   --[[ Basic Lua variable types are:
        nil, number, string, boolean, table, and function
   ]]--
   local num_var = 1

   local str_var = "some string value"

   local bool_var = true

   local table_var = {1,2,"three",four=4,five={6,"seven"}}

   local function func_var(arg1,arg2,arg3)
      return 1,"two"
   end

   -- Variable types are not static:
   print(type(num_var))
   num_var=str_var
   print(type(num_var))

   --Names are case sensitive
   local abc = 1
   local ABC = 2
   print(abc, ABC)
```

```
    -- Tables are lists and associative arrays, very flexible
    -- Grab the first list element, 1
    print(table_var[1])
    -- Grab the named element 'five', which is itself a table and print
    -- its second item
    print(table_var["five"][2])
    --Alternatively
    print(table_var.five[2])

    --Loop through all elements - order is uncertain
    for name,value in pairs(table_var) do
        print("name is:",name," and value is:",value)
    end

    --Loop through only the array elements, numerical order is preserved
    for index,value in ipairs(table_var) do
       print("index is:",index,"and value is:",value)
    end

    --Trying to print a table prints the address, not helpful
    print(table_var)

    --The table package has a tostring function that is nice
    print(table.tostring(table_var))

    --functions can return multiple arguments
    local x,y = func_var(1,2,3)
    -- Second returned value for our function is "two"
    print(y)

    -- Functions are variables and can be stored in tables
    table_var.new_func=func_var
    print(table_var.new_func(1,2,3))

    -- nil is similar to SAS's missing
    print(type(var_does_not_exist))

endsubmit;
RUN;
```

USING MODULES

A Lua module is a collection of Lua code stored in a file that can be loaded into a Lua environment using the require statement. SAS uses the special filepath called 'LUAPATH' to determine where to look for modules. The equivalent for SAS Macros is an autocall library. Lua modules are a convenient way to store functions that serve a similar purpose.

For the next examples, it is assumed that <lua_location> is a reference to a directory on your file system, e.g. C:\temp\lua_sample. It is also assumed that there is a subdirectory within <lua_location> called 'subdir'.

Create a file called 'my_module.lua' in <lualocation> with the following contents:

```
local module = {}
```

```
module.details={version="1.0",purpose="example"}


function module.print_info()
   print(table.tostring(module.details))
end

return module
```

You can then use the module like this:

```
filename luapath "<lua_location>";

PROC LUA restart;
submit;

   local mod = require 'my_module'
   mod.print_info()

endsubmit;
RUN;
```

Modules can reference other modules. For example, you can create a utility module that defines a function called 'tprint' to print the contents of a table. Create a file called 'utility.lua' in <lualocation> with the following contents:

```
local M = {}

function M.tprint(a_table)
   if(type(a_table) == 'table' then
      print(table.tostring(a_table))
   else
      print(a_table)
   end
end

return M
```

You can then simplify the first module (my_module.lua) as follows:

```
local module = {}

local util = require 'subdir.utility'
module.details={version="1.0",purpose="example"}


function module.print_info()
  util.tprint(module.details)
end

return module
```

A SIMPLE REST API MODULE

A RESTful API will have a base URL, with specific services having an address with this base URL as their root. For example, the base URL for accessing NOAA's web services is: https://www.ncdc.noaa.gov/cdo-web/api/v2/. The datasets service has the address: https://www.ncdc.noaa.gov/cdo-web/api/v2/datasets. If the ID of a specific dataset is known, its own services have the address https://www.ncdc.noaa.gov/cdo-web/api/v2/datasets/{ID}. As such, a REST module should allow the user to set a base url and make calls against it without having to repeatedly pass this information along.

All requests are made using HTTP, which means that users will be passing a file with the request header information and another file with the request body. The response from this request will take the form of a file with header information and a body. As such, a REST module should make reading and writing to these files simple.

The code for the rest module is developed in the following sections. It consists of 4 functions and some variables that allow for the control of different HTTP options and the base URL.

REST MODULE BASICS

The first thing we need to do is define the table object that our module will be returning. Next, define some variables to control the PROC HTTP options and the base url. The use of these variables will become clear when the HTTP request function is defined.

```
-- The table our module will return
local rest={}

-- This is the base URL for all calls.
rest.base_url=""

--- Some basic HTTP options
-- use cookies? (NO_COOKIES)
rest.cookies = true
-- clear cache after each call? (CLEAR_CACHE)
rest.clear_cache = false
-- avoid multiple headers when redirected or in other cases?
(HEADEROUT_OVERWRITE)
rest.header_overwrite = false
-- Cache connections?(NO_CONN_CACHE)
rest.cache_connection = true
```

FILE UTILITIES

Some functions for reading and writing to files can make life a lot easier as most requests will require that you pass a header and body file. You will also need to read the returned header and body in order to parse the response.

```
------------------------------------------------------------
--- Some utility functions to read to and write from files ---
------------------------------------------------------------
```

```
rest.utils={}

--- Read a file referenced by fileref into a string.
-- @param fileref [string or a fileref from sasxx.new()] - The fileref to
read from
-- @return contents [string] - The contents of the fileref, nil if nothing
was found
-- @return msg [string] - Any error message
function rest.utils.read( fileref )
   if type(fileref) == "string" then
       fileref =  sasxx.new(fileref)
   end
   local path = fileref:info().path
   if not path then
      fileref:deassign()
      return nil, "Couldn't open "..tostring(fileref).." for read."
   end

   local BUFSIZE = 2^13
   local f = io.open(path,"rb")
   if not f then
      return nil, "Couldn't open "..tostring(fileref).." for read."
   end

   local contents = ""
   while true do
      local bufread = f:read(BUFSIZE)
      if not bufread then break end
      contents = contents..bufread
   end
   f:close()
   return contents,""
end

--- Write a file referenced by fileref from a string with carriage returns
-- @param fileref [string or a fileref from sasxx.new()] - The fileref to
write to
-- @param txt  - the string being written to the file
-- @return rc [boolean] true if no error, false otherwise
function rest.utils.write( fileref, txt )
    if type(fileref) == "string" then
       fileref =  sasxx.new(fileref)
   end
   local path = fileref:info().path
   if not path then
      fileref:deassign()
      return false, "Couldn't open "..tostring(fileref).." for write."
   end
   local f = io.open(path,"wb")
   if not f then
      return false, "Couldn't open "..path.." for write."
   end
   f:write(txt)
   f:close()
   return true
end
```

```
--- Encode a URL string - need to do this when passing parameters that
contain special characters
-- @param str String containing a URL to encode
function rest.utils.urlencode( str )
    if (str) then
        str = string.gsub (str, "\n", "\r\n")
        str = string.gsub (str, "([^%w ])",
         function (c) return string.format ("%%%02X", string.byte(c)) end)
        str = string.gsub (str, " ", "+")
    end
    return str
end
```

The urlencode function replaces special characters with their URL-encoded equivalents. This is necessary if, for example, you want to pass a '/' symbol in a request.

It is also helpful to have some filerefs available to read and write to by default. The next portion of code will create four filerefs in the work directory or, if a macro variable called 'working_folder' is found, it will create them there.

```
-- Assign some default filerefs that can be used for requests and
responses.
--   _hin_  : the request header
--   _hout_ : the response header
--   _bin_  : the request body
--   _bout_ : the response body
if sas.symget('working_folder') and sas.symget('working_folder') ~= '' then
   rest.working_folder = sas.symget('working_folder')
else
   rest.working_folder = sas.getoption('work')
end
sas.submit([[
   filename _hin_  "@working_folder@/_hin_.txt";
   filename _hout_ "@working_folder@/_hout_.txt";
   filename _bin_  "@working_folder@/_bin_.txt";
   filename _bout_ "@working_folder@/_bout_.txt";
]], {working_folder = rest.working_folder})
```

THE REQUEST FUNCTION

Finally, we can define a request function that uses all of these pieces. The function first parses the options that were defined at the beginning and then determines what (if anything) will be passed as the input body, header, and content type. The actual request is then made by calling PROC HTTP, and the response is parsed to determine if the call was successful.

```
----- Submit a request to rest.base_url
-- ARGUMENTS:
-- action       - string - 'GET', 'POST', 'PUT', etc.
-- request      - string - the portion of the URL that follows
rest.base_url
-- body_in         - string - [OPTIONAL]fileref for the request body.
```

```
-- header_in    - string - [OPTIONAL]fileref for the request header.
-- content_type - string - [OPTIONAL]passed to PROC HTTP's 'ct' parameter
(content type)
-- body_out          - string - [OPTIONAL] the fileref to write the output
to. Defaults to '_bout_'
-- header_out         - string - [OPTIONAL] the fileref to write the
returned header to. Defaults to '_hout_'
--RETURNS:
-- pass        - boolean - whether or not the HTTP return code was in the
200's (200 OK, 201 CREATED, etc)
-- code        - number  - the actual http return code
function rest.request( action, request, body_in, header_in, content_type,
body_out, header_out )

   -- Handle HTTP options
   local http_options = ""
   if not rest.cookies then http_options = http_options.." NO_COOKIES" end
   if rest.clear_cache then http_options = http_options.." CLEAR_CACHE" end
   if rest.header_overwrite then http_options = http_options.."
HEADEROUT_OVERWRITE" end
   if not rest.cache_connection then http_options = http_options.."
NO_CONN_CACHE" end

   -- Make sure a / separates the base url and request
   if rest.base_url:sub(-1) ~= '/' and request:sub(1,1) ~= '/' then request
= '/'..request end
   local url_str = rest.base_url..request

   -- Set the content type and out arguments
   local body_in_arg, header_in_arg, ct_arg
   if not content_type then ct_arg = "" else ct_arg = "ct =
'"..content_type.."'" end
   if not body_in then body_in_arg = "" else body_in_arg = "in =
"..tostring(body_in) end
   if not header_in then header_in_arg = "" else header_in_arg = "headerin
= "..tostring(header_in) end
   if not body_out then body_out = "_bout_" end
   if not header_out then header_out = "_hout_" end

   -- Initialize the contents of the response files so that if something
   -- goes wrong, we don't accidentally get the contents from a previous
   -- request
   rest.utils.write(body_out,"_NOT_SET_BY_PROC_HTTP_")
   rest.utils.write(header_out,"_NOT_SET_BY_PROC_HTTP_")

   sas.submit([[
     PROC HTTP
          @http_options@
          url=%nrstr('@url_str@')
          @header_in_arg@
          headerout=@header_out@
          @body_in_arg@
          out=@body_out@
          @ct_arg@
          method='@action@';
      RUN;
```

```
    ]])

    --Grab the response and pull out the http return code
    local header_string = rest.utils.read(header_out)
    local header_line_one = sas.scan(header_string,1,'\n\r')
    local code = tonumber(sas.scan(header_line_one,2,' '))
    local pass  = (code >= 200 and code < 300)
    return pass, code
end

return rest
```

REST MODULE EXAMPLE

In the following example, a GET request to http://httpbin.org/ip is made, which will return the IP address of the requester. Notice that only three lines of lua code are needed for this; one to load the module, another to set the base url, and a third to make the actual request.

```
filename luapath "<location of the REST lua module>";
/* Set this if you want to store the header and body files somewhere
other than work */
%let working_folder=;
PROC LUA restart;
submit;
      local rest = require 'rest'

      rest.base_url = 'http://httpbin.org/'

      local pass,code = rest.request('get','ip')
      print(pass,code)
      print(rest.utils.read('_bout_'))


endsubmit;
RUN;
```

PARSING JSON

Once the response is read, it will generally need to be parsed. A common format for request and response bodies is Javascript Object Notation (JSON). Lua makes parsing JSON easy, since any JSON object can be expressed as a Lua table. Note that the same is not true for SAS datasets in general. Jeffrey Friedl's publicly available Lua module for encoding and decoding JSON in Lua is located here: http://regex.info/blog/lua/json. All references to 'json.lua' in this paper are referring to that file.

INTERFACING WITH A REST API

In this section, the REST module defined in the previous section is used to create a simple interface to a REST API; NOAA's 'Climate Data Online' web service. Because there are many REST enpoints for this service, an attempt to cover all (or even many) of them is not made. Instead, an interface to the datasets and data endpoints is developed as an example. The API is documented here: https://www.ncdc.noaa.gov/cdo-web/webservices/v2. Before any requests can be made to the API, a token must be obtained by going to

https://www.ncdc.noaa.gov/cdo-web/token and providing an email address. This token is used for all requests.

As with the REST module, the code for the module will be developed in components. Functions for interacting with the datasets API, used for getting a list of available datasets and their IDs, as well as the data API, which provides specific data values, are developed. A generic request function for any of the NOAA API endpoints is also provided.

NOAA API BASICS

The NOAA module will use the rest and json modules to make requests and parse the responses. It will allow for the setting of the token used for requests and will handle all of the interactions with the rest module. A utility function used internally to parse the various filters is defined, but note that it is not returned as part of the module since it has no real use outside of the context of the NOAA API.

```
local noaa = {}
local rest = require 'rest'
local json = require 'json'

-- The token that will be used for requests.
noaa.token=false

-- Set the base url
rest.base_url = 'https://www.ncdc.noaa.gov/cdo-web/api/v2/'

--- Take a table of parameters and create a string to append to a url
-- @params parms_table [table] - table of name,value pairs
-- @return parms_string [string] - parameter table parsed as a string
function parms_table_to_string(parms_table)
  local has_parm = false
  local parms_string = ""
  for key,value in pairs(parms_table or {}) do
     if value then
        if has_parm then parms_string = parms_string..'&' end
        parms_string = parms_string..key..'='..value
          has_parm = true
     end
  end
  if has_parm then parms_string='?'..parms_string end
  return parms_string
end
```

DATASETS API

The datasets API takes an optional ID and a series of filters. The description of these parameters can be found here: https://www.ncdc.noaa.gov/cdo-web/webservices/v2#datasets. The function first checks that the token is defined and then parses the parameters to construct a proper HTTP query string. Once the request is made, the response is converted to a lua table and returned.

```
-----------------------------------------------------
--                          #DATASETS               --
-----------------------------------------------------
noaa.datasets = {}
--- Get a specific dataset or pass nil for id and get a list of all
datasets. Use filters to filter the results. See"
---  https://www.ncdc.noaa.gov/cdo-web/webservices/v2#datasets
-- @param id [string] - Optional, if nil a list of all tables is returned,
otherwise querry a specific table by id
-- @param filters [table] - Optional, table with entries for any filters to
apply. See help for the list.
-- @return pass [boolean] - true if call was successful, false if error
occurred.
-- @return datasets [table] - Table with all available datasets
function noaa.datasets.get(id,filters)
    if not noaa.token then
       print('ERROR: You need to specify a token first')
       return false,nil
    end
    local id_str = ""
    if id then id_str = '/'..tostring(id) end
    local parms_str = parms_table_to_string(filters)
    rest.utils.write('_hin_','token:'..noaa.token)
    local pass,code = rest.request('get','datasets'..id_str..parms_str
,nil,'_hin_')
    if not pass then
       return false, nil
    end
    return true, json:decode(rest.utils.read('_bout_'))
end
```

DATA API

The data API requires start and end dates and a dataset ID. It is documented here: https://www.ncdc.noaa.gov/cdo-web/webservices/v2#data. The function takes an optional argument called 'dataset' which allows the user to specify a SAS Dataset to have the results written to. Like the datsets function, it checks that the token is specified and parses the arguments to form a proper query string.

```
-----------------------------------------------
--                          #DATA              --
-----------------------------------------------
noaa.data = {}
--- Get a specific dataset or pass nil for id and get a list of all
datasets. Use filters to filter the results. This APi will be deprecated
soon (as of March 2019)
---  https://www.ncdc.noaa.gov/cdo-web/webservices/v2#data
-- @param id [string] - dataset id to get data for. Use datasets.get() to
see a list of all IDs
-- @param startdate [string] - required, Accepts valid ISO formated date
(YYYY-MM-DD) or date time (YYYY-MM-DDThh:mm:ss)
-- @param enddate [string] - required, Accepts valid ISO formated date
(YYYY-MM-DD) or date time (YYYY-MM-DDThh:mm:ss)
```

```
-- @param filters [table] - Optional, table with entries for any filters to
apply. See help for the list.
-- @param dataset_name [string] - Optional. If nothing is specified, no
dataset is created. Otherwise, a dataset called 'dataset_name' is
--                                  created with the contents of the
requested data. If you specify a libname (e.g. pass  'mylib.dataset') then
--                                  the specified library needs to exist
-- @return pass [boolean] - true if call was successful, false if error
occurred.
-- @return data [table] - Table with the requested data
function noaa.data.get(id,startdate,enddate,filters,dataset_name)
    if not noaa.token then
       sas.print('%1zYou need to specify a token first')
       return false,nil
    end
    if not id then
       sas.print('%1zAn id is required for this API')
       return false,nil
    end
   --default output_type to table only
   dataset_name = dataset_name or false
    filters = filters or {}
    filters.datasetid = tostring(id)
    filters.enddate = tostring(enddate)
    filters.startdate = tostring(startdate)
    local parms_str = parms_table_to_string(filters)
    rest.utils.write('_hin_','token:'..noaa.token)
    local pass,code = rest.request('get','data'..parms_str ,nil,'_hin_')
    if not pass then
       return false, nil
    end
   local output = json:decode(rest.utils.read('_bout_'))
   if dataset_name then
      local sasds = output.results
      sasds.vars={station    = {length=32, type="C"},
                  date       = {length=32, type="C"},
                  value      = {length=8,  type="N"},
                  attributes = {length=32, type="C"},
                  datatype   = {length=8,  type="C"}
                  }
      sas.write_ds(sasds, tostring(dataset_name))
   end
   return true, output
end
```

A GENERIC NOAA API REQUEST

```
------------------------------------------------
--                      #GENERIC              --
------------------------------------------------
--- Generic request function for endpoints not covered in this module for
the v2 API
function noaa.request(method,endpoint,filters)
    if not noaa.token then
       sas.print('%1zYou need to specify a token first')
```

```
        return false,nil
    end
    filters = filters or {}
      endpoint = endpoint or ''
      method = method or 'get'
    local parms_str = parms_table_to_string(filters)
    rest.utils.write('_hin_','token:'..noaa.token)
    local pass,code = rest.request(method,endpoint..parms_str ,nil,'_hin_')
    if not pass then
       return false, nil
    end
    return true, rest.utils.read('_bout_')
end

return noaa
```

EXAMPLE

You can now easily interact with the NOAA API. In this example, a list of all datasets is retrieved. Next, the summary data for Cary, NC for the month of January, 2018 is pulled into a SAS dataset. Finally, the list of all stations for a given zipcode are retrieved.

```
filename luapath "<location of the NOAA module>";
PROC LUA restart;
submit;
    local noaa = require 'noaa'

    noaa.token='jfFOguhJNpUDOhhVvoHdVmXMhlCneimp';

    -- Get a list of all datasets
      local pass, datasets = noaa.datasets.get()

    --Get some actual data
    local filter = {limit=25,locationid='ZIP:27603'}
    local pass, data = noaa.data.get('GHCND','2018-01-01','2018-01-
31',filter,'work.weather')

    --Use the generic request function to access the stations API
    local pass, output =
noaa.request('get','stations',{locationid='ZIP:27603'})

endsubmit;
RUN;
```

CONCLUSION

As RESTful APIs continue to grow in prevalence, the ability to easily interact with a REST API from SAS will allow you to interact with these services, perform any data analyses, and generate reports all from within the SAS environment. PROC LUA simplifies this task by allowing you to create a simple set of functions, managed within lua modules, to streamline these interactions.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Steven Major
100 SAS Campus Dr
Cary, NC 27513
919-531-1467
Steven.major@sas.com

BASIC INSTRUCTIONS

WRITING GUIDELINES

Trademarks and product names

To find correct SAS product names (including use of trademark symbols), if you are a SAS employee, see the Master Name List. Otherwise, see SAS Trademarks.

- Use superscripted trademark symbols in the first use in title, first use in abstract, and in graphics, charts, figures, and slides.
- Do not abbreviate product names. For example, you cannot use "EM" for SAS® Enterprise Miner™. After having introduced a SAS product name, you can occasionally omit "SAS" for certain products, provided that your editor agrees. For example, after you have introduced SAS® Simulation Studio, you can occasionally use "Simulation Studio."

Writing style

- Use active voice. (Use passive voice only if the recipient of the action needs to be emphasized.) For example:

  The product creates reports. (active)
  Reports are created by the product. (passive)

- Use second person and present tense as much as possible. For example:

  You get accurate results from this product. (second person, present tense)
  The user will get accurate results from this product. (future tense)

- Run spellcheck, and fix errors in grammar and punctuation.

Citing references

All published work that is cited in your paper must be listed in the REFERENCES section.

If you include text or visuals that were written or developed by someone other than yourself, you must use the following guidelines to cite the sources:

- If you use material that is copyrighted, you must mention that you have permission from the copyright holder or the publisher, who might also require you to include a copyright notice. For example: "Reprinted with permission of SAS Institute Inc. from *SAS® Risk Dimensions®: Examples and Exercises*. Copyright 2004. SAS Institute Inc."
- If you use information from a previously printed source from which you haven't requested copyright permission, you must cite the source in parentheses after the paraphrased text. For example: "The minimum variance defines the distance between cluster (Ward 1984, p. 23)

TIPS FOR USING WORD

These instructions are written for MS Word 2007 and MS Word 2010. The steps are similar for MS Word 2003.

To select a paragraph style

1. Click the HOME tab. The most common styles in your document are displayed in the top right area of the Microsoft ribbon. If you don't see a style that you want, click the slanted down arrow at the bottom right corner of the Styles area, and scroll through the list. The main styles for this template are headings 1 through 4, PaperBody, and Caption. Avoid using other styles.
2. To change a paragraph style, click the paragraph to which you want to apply a style, and then click the style that you want in the ribbon.
3. PaperBody (used for most text) is automatically applied when you press Enter at the end of any heading style or the Caption style.

To insert a caption

1. Click **REFERENCES** on the main Word menu.
2. Click **Insert Caption**.

3. Select the **Label** type that you want.
4. Click **OK**.

To insert a cross-reference

1. Click **REFERENCES** on the main Word menu.
2. Click **Cross-reference**.
3. In the **Reference type** list box, select Heading, Figure, Table, Display, or Output.
4. For a heading:
    a. In the **For which heading** list, select the heading that you want.
    b. From the **Insert reference to** list, select **Heading text.**
5. For a figure, table, display, or output:
    a. In the **For which caption** list, select the caption that you want.
    b. From the **Insert reference to** list, select **Only label and number**.

To insert a graphic from a file

1. Click **INSERT** on the main Word menu.
2. Click **Picture**.
3. In the Insert Picture dialog box, navigate to the file that you want to insert.
4. When the name of the file that you want to insert is displayed in the **File name** box, click **Insert**.