The QDL Workspace

Introduction

The QDL interpreter may be run from the command line but there is also another supplied tool called the ***workspace***. This runs a QDL interpreter and lets you run commands interactively as well as stores state (so you can create a bunch of variables and save them for future use). Sessions may be saved and loaded.

Getting and starting the workspace

The current release of QDL is on git hub at https://github.com/ncsa/security-lib/releases and will be in qdl.jar. Download that and you should be set.

To start QDL, invoke it using java

java -jar qdl.jar

And you should get the splash screen:

```
****************************************
Welcome to the QDL Workspace
Version 1.0
Type )help for help.
****************************************
```

The prompt is simply an indent of 4 spaces. So try it out by typing the famous program

```
    say('Hello world');
Hello world
```

Congrats, you have just run your first complete QDL program. Since this is an interpreter, you issue commands in the QDL language. There is a whole other reference manual/tutorial for that. This blurb is about how to interact with the workspace.

Since it says to type)help, do it and you will see

```
   )help
This is the QDL (pronounced 'quiddle') workspace.
You may enter commands and execute them much like any other interpreter.
There are several commands available to help you manage this workspace.
Generally these start with a right parenthesis, e.g., ')off' (no quotes) exits this
program.
Here is a quick summary of what they are and do.
  )buffer -- commands relating to using buffers.
  )clear -- clear the state of the workspace. All variables, functions etc. will be
lost.
  )edit -- commands relating to running the line editor.
  )env -- commands relating to environment variables in this workspace.
  )     -- short hand to execute whatever is in the current buffer.
```

```
  )funcs -- list all of the imported and user defined functions this workspace
knows about.
  )help -- this message.
  )modules -- lists all the imported modules this workspace knows about.
  )off -- exit the workspace.
  )load file -- Load a file of QDL commands and execute it immediately in the
current workspace.
  )vars -- lists all of the variables this workspace knows about.
  )ws -- commands relating to this workspace.
```

If you type)off, you will exit the workspace.

```
    )off y
exiting...
```

Running a single script

You may run a single script with the *-run + script_path* argument if invoking via java:

```
java -jar qdl.jar -cfg config_file -name config_name -run script_path
```

or from the command line with the shell script as

```
qdl script_path
```

in which case your configuration will be loaded completely (no banner or splash screen) the script executed and the interpreter will exit. Note that environment variables and substitutions will be available. This lets you embed QDL in shell scripts, for instance with your required runtime. Note that this will load the system completely, so any boot scripts of virtual file system mounts will happen before your script is called.

Commands generally

Since the workspace is charged with managing the QDL environment for us, its commands are not QDL commands. All of the start with a right parenthesis, e.g.)help.

The basic structure of a command is

command action arguments

So this is the command to save the current workspace to the file my_space.ws

```
    )ws save my_space.ws
workspace saved at Thu Jan 30 16:32:28 CST 2020
```

-)ws is the command
- *save* is the action
- *my_space.ws* is the only argument, in this case.

Arguments may be simple, such as a single file name here, or there may be several of them. This varies per command, hence this reference.

In the command reference that follows, the heading of the section is the command and the subsections are the actions. An argument of – means that this may be omitted and is the default.

Templates and script preprocessing

You may set values directly in the interpreter environment and access them via the ***templating syntax*** of `${…}`.

Note: This is a feature of the workspace and is *not* part of QDL! It is to help harried coders reduce their typing workload and keep useful snippets of text at hand. If you develop code using templates, they should be removed if you are, e.g. going to save your code in a file and run it as a script later.

An example of how to use this: Let's set two environment variables:

```
)env set vo voPersonExternalID
)env set a @accounts.google.com
```

The template are done ***before*** any parsing or other processing. In computer languages a lot of use can be made of pre-processing or doing things to the code before actual evaluation, for instance, look up the Obfuscated C competition which is a humorous jab at abuses of the C pre-processor. This is a very useful facility to have and can make your work vastly easier at times, but it should be an aid, not a lifestyle choice.

So you could use this to, for instance, keep typos to a minimum but issuing statements like (assuming `claims.oidc` has the value 349765)

```
    ${vo} := claims.oidc + '${a}';
    say(${vo});
349765@accounts.google.com
    say(voPersonExternalID);
349765@accounts.google.com
```

What is the advantage of this? You can embed things like code snippets (especially the one you keep mistyping) or use it to prevent spelling issues or to future proof. In this case, if there were several references to `voPersonExternalID,` then the template allows you to set this one place in your script. Note that this lets you template for things like control structures, names of functions or really anything – it is wholly outside the language. If the name of ever changes (to say `voPersonExternalID_v2` then you would need to only change where it is set in the environment and the changes would take place automatically. While fairly trivial to use, it can make a lot of coding/testing considerably easier.

Repeating the last command

If you enter a single % then the very last command you entered is repeated.

```
   date_iso()
2020-03-16T22:01:29.258Z
  %
```

```
2020-03-16T22:07:31.626Z
```

In this case, the current date and time are created and then repeated.

Listing things

For the following commands

-)funcs
-)vars
-)modules

Each allows for various formatting switches when listing. These are

- -r *regex* = apply the regex to each element of the output for matching
- -cols = list in columnar formatting
- -w *int* set the max display width. Note that -cols overrides this
- -compact = show namespaces as lists, so rather than a#q and b#q you'd see [a,b]#q.

A few examples:

Show the system functions that start with *to_* and list as a single column

```
  )funcs system -r to_.* -cols
to_hex()
to_json()
to_list()
to_lower()
to_upper()
```

Remember that for regexes the idiom **.*** means to match any character, so **to_.*** means just match anything that starts with **to_**. Also, you do not need a start of line character because each entry is not a line.

Just list the functions:

```
  )funcs
a#f(1)     arf#f(1)   b#f(1)     c#f(1)     d#f(1)     woof#f(1)
```

Now list them in compact notation

```
  )funcs -compact
[a, d]#f(1)             [b, c, arf, woof]#f(1)
```

And now, list all of the system (built-in) functions with their namespaces (-fq flag is specific here) that are in the ***io*** namespace and restrict the output to 60 characters in width:

```
  )funcs system -fq -r io#.* -w 60
```

```
io#dir()      io#print()    io#say()      io#unmount()
io#mount()    io#read()     io#scan()     io#write()
```

Command reference

)buffer

A *buffer* is just a bunch of text. A buffer on disk is just a file, hence every file is a buffer. In an interpreted language, you may want to have several statements that can be executed together. You may buffer them, then have them execute at once. There is also an editor supplied (see the **)edit** section below), though that is wholly optional.

There are two types of buffers, *local* which is kept inside the workspace and *external* which is just a fancy name for any file. You may have both or neither.

local

These actions relate to the management of the local buffer. You may set what the active buffer is using **on** and **off** here. This effects the **)edit** and quick run **)** commands.

append *file_name*

append the given file to the end of the current buffer. See *prepend* in this section for inserting a file at the beginning.

clear

clear the local buffer. All contents will be lost.

load file_name

load the given file into the local buffer, replacing its contents. This implicitly sets the focus to the local buffer

off

turn local buffering off

on

turn local buffering on

Example of how to use local buffering to put in a longer statement

```
)buffer local on
)edit
CLI editor.
Empty buffer. Type ? to see help.
edit >i
define[
    sum(a, b)
   ]body[
```

```
    >> add a pair of integers and return the sum.
    c:= a+b;
    return(c); // this terminates execution and returns the value.
   ];

say('the sum of 3 and 4 is ' + sum(3,4));
.
edit >q
done!
  )
the sum of 3 and 4 is 7
  )funcs help sum 2
add a pair of integers and return the sum.
```

So what does this do?

- Creates a local buffer to store commands
- edits it (i means to input, the single period at the start of a line means exit input mode)
- *q* quits the editor so you can now run the buffer rather then edit it.
- **)** – a single left parenthesis – means to run whatever is in the buffer. The buffer has a a function defined and runs it.
- Checks that the comment in the function is reasonable by showing it with the)help command.

prepend *file_name*

insert the file at the beginning of the current buffer.

save *file_name*

saves the current buffer to the given file.

show

prints the contents to the console.

So for example to load the sample hello world file and print it to the console:

```
  )buffer local load /home/jeff/dev/qdl/examples/hello_world.qdl
Buffer updated.
  )buffer local show
/*
  The expected Hello World program.
  Jeff Gaynor
  1/26/2020
*/
say('Hello world!');
    // for fun, let's clear the buffer and show the results
  )buffer local clear
Local buffer cleared
  )buffer local show
```

A blank line is returned since the buffer is empty.

file

These relate to using an external file as the buffer. This has the great advantage that you can edit it in your favorite editor and have the system run it as you develop by using the quick run command, **)**.

file *file_name*

use this file as the currently active buffer.

If the current active buffer is set to **local** you must also change that.

)clear

See)ws clear. A typical invocation looks like

```
    )clear
workspace cleared
```

This is identical to issuing

```
)ws clear
```

) (aka *quick run*)

This is the “run the buffer” command. It is very useful shorthand. One nice application is to use this on whatever you are editing externally. So set your buffer to a file and make sure that local mode is off:

```
)buffer file file_path
)buffer local off
```

and your favorite external editor on the file. Edit for a bit, then when you are ready to test something, simply enter this command and the file will be read and executed immediately. Note that each time the file will be reloaded from disk so you always see you immediate changes.

If you have opted for a local buffer, then that will be executed instead. This is a great convenience when writing code. Here is an example where we load the hello world sample and run it

```
  )buffer local load /home/jeff/dev/qdl/examples/hello_world.qdl
Buffer updated.
  )buffer local show
/*
  The expected Hello World program.
  Jeff Gaynor
  1/26/2020
*/
say('Hello world!');

  )buffer local on
Local buffer enabled.
  )
Hello world!
```

Note that all that was needed to run this was the single character, **)**. An external file example (using an environment variable for shorthand):

```
    )env set q  /home/jeff/dev/qdl/example
    )buffer file ${q}/hello_world.qdl
Buffer set to "/home/jeff/dev/qdl/examples/hello_world.qdl.
    )
Hello world!
```

)edit

– (no argument)

edit whatever the current buffer is

local

Edit the local buffer, regardless of whether it is active or not.

file [*filename*]

Edit the file. If none is given, edit the current active external file

A note about the editor

The editor supplied with QDL is a very basic, no bones about it *line editor* meaning that you issue commands *about* the lines. You will not see what you are editing unless you tell it to show you (the **p** command). If you invoke the editor, you may see help by issuing the **?** At the **edit**> prompt. Here is an example of starting the editor and getting the help to print out.

```
  )edit
CLI editor.
edit >?
This is the line editor. It operates per line. Each command is of the form
command [start,stop,target] arg0 arg1 arg2...
where
command is a command for the editor (list below). There are long and short forms
[start,stop,target] are line numbers hence integers.
    start = the starting index, always 0 or greater
    stop = the ending index. All operations are inclusive of this line number
    target = where to apply the [start,stop] interval. E.g. the insertion point for
copying text.
arg0, arg1,... = list of strings, possibly in quotes if needed, that are arguments
to the command.

To see help on a specific command, type ? after the command, e.g. to get help on
the insert command type
i ?
List of commands
a (append) append text either before a given line or to the end of the buffer.
b (view) view the contents of the clipboard. Contents not editable.
```

```
c (copy) copies a range of lines to the clipboard.
d (delete) deletes a range of lines
e (edit) edit a range of lines. Note this will effectively replace those lines.
f (find) search lines that match a given regular expression, printing each line
found.
i (insert) insert lines starting at a give index or append lines to the end of the
buffer.
l (clear) clear the buffer (but not the clipboard)
m (move) move a block of text using the clipboard
p (print) print the buffer or a subset of it
q (quit) quit the editor
r (read) read a file into the buffer
s (replace) substitute over a range of lines using a regex
t (paste) paste the contents of the clipboard into the buffer
u (cut) cut a range of lines from the buffer and leave in the clipboard
v (verbose) turn on verbosity, i.e. print more about the functioning of the editor
w (write) write the buffer to a file
z (size) query the buffer for its current size
? print this help, or in context, print the help for a command.
edit >
```

Note that the end of this you are returned to the prompt. **p** prints the buffer, **i** lets you insert text until you enter a single period. **q** quits. It is actually quite nice – it has a clipboard, supports regexs for certain operations and the list goes on.

Why have such an editor in this great day and age? Because not all keyboards are created equal (*e.g.*, does your cellphone let you hit the escape key? Can't use the estimable old unix editor *vi* without that.) Also, it seems that invariably if I need to log in to a server (which never have GUIs, since they are servers) remotely during an emergency, the terminal type gets screwed up and it all defaults back to 7 bit ASCII – pretty much the exact character set QDL recognizes. (Quick reminder that variables and functions use a small set of characters, but strings are fully UTF-8.) This editor is *not* intended to be the go to. The intent is that you set an external file, use whatever you like and when you need to run your code, just hit **)** in the workspace. The editor, however, is pretty darned useful if you need it and since I had it lying around, why not?

One last little quirk is that the editor is indeed a completely stand alone program invoked by the workspace, so you can save your work in it to a file. If you are using it to edit the local buffer, all you need to do is exit (enter **q** at the prompt) and don't choose save (unless you want another copy of the buffer in an external file), the workspace knows how to get the resulting changes back all on its own.

)env

The workspace allows for environmental variables The environment

drop *variable*

remove the named variable from the environmental

get *variable*

display the current value of the variable.

list, –

show all currently defined variables.

load *filename*

load the given variables file, adding it to the current environment. Note that this may overwrite currently defined values. The format of this file is a standard java properties file. This file becomes the active one and save will go to it unless a different file is specified.

save [*filename*]

save the current environment to the file

set *variable value*

Set the *variable* to have the given *value*. Note that everything to the right of the variable is considered the value, so embedded spaces are possible. If you need to, you may surround the value in double quotes to ensure that all spaces are faithfully preserved.

Examples

Set a variable

```
  )env set www "once upon   a   midnight"
  )env get www
once upon   a   midnight
```

Show all the defined variables. We could supply the value of *list* but don't since it is optional.

```
  )env
Current environment variables:
{
 qdl_root=/home/bob/apps/qdl
 arf=abc d goldfish
 a=armstrong(600);
 d=load_module('/tmp/boot.qdl');
 foo=12345
}
```

)funcs

list, – [*switches*]

list all of the current functions in your workspace See the note above for command line switches that apply to this action.

Note that if you want to list specific functions, you *must* use the list command plus a regex. So let us say you had

```
    define[f(x)]body[return(x+1);];
    define[f(x,y)]body[return(x+y);];
```

```
    define[foo(x)]body[return(x-2);];
```

To to list any functions named exactly *g* you would issue

```
  )funcs list -r g.*
```

(No such functions exist.) So now let us list the ones called *f*:

```
  )funcs list -r f.*
f(1), f(2)
```

To list every function that starts with *f* we use the regex *f.** the '.*' means to take any character (the period) and match any. So can be read as *f* + anything:

```
    )funcs list -r f.*
f(1), f(2), foo(1)
```

By the same token, to just get the functions that start with *fo* you would use *fo.** like so

```
    )funcs list fo.*
foo(1)
```

drop *function*

remove a function. Note that this only applies to functions that have been defined (but not imported) in the workspace.

help [*name arg_count*]

If no arguments, print out every function and its simple description.

```
  )funcs
a#f(1), armstrong(1), b#f(1), check_parser(1), f(1), f2(1), f3(1), fac(1)
```

Note that they are qualified as needed and have their argument counts included. To list a specific function, you would issue something like this:

```
  )funcs help armstrong 1
An Armstrong number is a 3 digit number that is equal to the sum of its cubed
digits. This computes them for 100 < n < 1000.
So for example 407 is an Armstrong number since 407 = 4^3 + 0^3 + 7^3
```

In this case, the request was for anything about armstrong functions the workspace knows so the complete description from the function definition is returned.

system [*-fq*] [switches]

This will list the built in system functions in tabular format. Note that none of these has argument counts listed, because most of these have variable length arguments. This is intended to let you look up a name. You may also show the fully qualified names with the *-fq* switch. There is also no help (at this

point) for each of these available in the workspace – mostly because keeping it in sync with the reference manual would be a huge ongoing task. Consult the reference manual for all the details. If you need to have it display for a certain width, pass that in. The default is 120 characters.

```
  )funcs system -w 80
abs()             for_keys()        is_list()         raise_error()     substring()       vfs_unmount()
box()             for_next()        keys()            random()          to_hex()          write_file()
break()           from_hex()        list_append()     random_string()   to_json()
check_after()     from_json()       list_copy()       read_file()       to_list()
common_keys()     has_keys()        list_insert_at()  remove()          to_lower()
contains()        hash()            list_keys()       rename_keys()     to_upper()
continue()        import()          list_subset()     replace()         tokenize()
date_iso()        include_keys()    load_module()     return()          trim()
date_ms()         index_of()        load_script()     run_script()      unbox()
decode_b64()      indices()         mask()            say()             union()
dir()             insert()          mod()             scan()            vdecode()
encode_b64()      is_defined()      numeric_digits()  set_default()     vencode()
exclude_keys()    is_function()     print()           size()            vfs_mount()
```

And just to show what all the fully qualified names look like

```
 )funcs system -fq -w 80
io#dir()              math#numeric_digits()  stem#list_keys()       sys#for_next()       sys#trim()
io#mount()            math#random()          stem#list_subset()     sys#import()         sys#vdecode()
io#print()            math#random_string()   stem#mask()            sys#index_of()       sys#vencode()
io#read()             math#to_hex()          stem#remove()          sys#insert()
io#say()              stem#box()             stem#rename_keys()     sys#is_defined()
io#scan()             stem#common_keys()     stem#set_default()     sys#is_function()
io#unmount()          stem#exclude_keys()    stem#size()            sys#load_module()
io#write()            stem#from_json()       stem#to_json()         sys#load_script()
math#abs()            stem#has_keys()        stem#to_list()         sys#raise_error()
math#date_iso()       stem#include_keys()    stem#unbox()           sys#replace()
math#date_ms()        stem#indices()         stem#union()           sys#return()
math#decode_b64()     stem#is_list()         sys#break()            sys#run_script()
math#encode_b64()     stem#keys()            sys#check_after()      sys#substring()
math#from_hex()       stem#list_append()     sys#contains()         sys#to_lower()
math#hash()           stem#list_copy()       sys#continue()         sys#to_upper()
math#mod()            stem#list_insert_at()  sys#for_keys()         sys#tokenize()
```

Normally you do not need to fully qualify system names, but if you really had to, for example, name a function *load_module* you could do so and just qualify which you want to use, yours or the system command.

)help

This takes no action and, as you saw above, just prints out a summary.

)modules

list, –

list the modules that are known to the workspace. This does not mean they have been imported.

imports

list the imports (URN and current alias) that are active.

```
  )modules
a:a  b:b  qdl:/java
```

This lists that there are three modules. To see if these are associated with aliases, use the imports actions:

```
  )modules imports -cols
a:a->[a, d]
b:b->[b, c, arf, woof]
```

These are the modules that have been explicitly imported. In this case, there is one, *qdl:/java* which has not.

Note that a synonym for this is just

```
)imports
```

)off [*y*]

This causes the system to exit. If you supply the argument **y** then the system will exit without saving. Otherwise, you will be prompted, but if you agree to exit, it will be immediate and with no save.

)vars

drop

list – [*switches*]

List all of the variables (but not their content) in the current workspace. See the switches supported by this action above.

size

List the total number of symbols in use in the workspace. This does not tell you how much memory is in use. See the workspace memory command for that.

system [switches]

List the system constants. Note that these are not kept in the symbol table and can only be accessed in fully qualified form.

```
  )vars system
sys#constants.
  sys#constants.var_types.
{boolean=1, string=3, null=0, integer=2, decimal=5, stem=4, undefined=-1}
```

)ws

echo [*on* | *off*] [-pp on|off]

Turns *echo mode* on or off. If no argument, the current mode is displayed. If the switch **-pp** is used, then pretty print mode is turned on or off. This refers to printing all stems vertically.

Echo mode is the default on workspace start up. It will take any expression (so not control statements, like if, switch, etc.) and wrap it in a *say();* call. It will also add any implied but missing final ";".

```
  )ws echo
echo mode currently on
  2 + 2
4
  )ws echo off
echo mode off
// Without echo mode on, you would have to fully write out every expressions
// So this will fail:
  2 + 2
syntax error:

// now turn echo back on so it works
  )ws echo on
echo mode on

  mod(3*4*5*6,11)
8
// same as
// print(mod(360, 11));

  a. := indices(6)+10
  a.
{0=10, 1=11, 2=12, 3=13, 4=14, 5=15}
```

Echo mode lets you use QDL like a big calculator.

Another example. Turn on pretty print

```
  )ws echo on -pp off
echo mode on, pretty print = off
  sys#constants()
{var_types.={boolean=1, string=3, null=0, integer=2, decimal=5, stem=4, undefined=-
1}}
)ws echo on -pp on
echo mode on, pretty print = on
  sys#constants()
{
 var_types.= {
  boolean=1,
  string=3,
  null=0,
  integer=2,
  decimal=5,
  stem=4,
  undefined=-1
 }
```

```
}
```

load *file*

Load a saved workspace from the given file. Relative references are resolved against the qdl root directory (which you can see since it is saved in the environment).

memory

Reports the amount of used memory, remaining memory and number of processors.

```
  )ws memory
memory used = 479MB, free = 461MB, processors = 8
  a. := indices(100000);
  )ws memory
memory used = 479MB, free = 441MB, processors = 8
  size(a.);
100000
```

It should be noted that the amount of memory is not fixed: if you decide to write

```
a. := random(1000000)^5
```

You may get something like this

```
  )ws memory
memory used = 761MB, free = 408MB, processors = 8
```

Which simply means that the system allocated more memory. The standard limit is 2 GB on 32 bit systems and unlimited on 64 bit system. You can change the initial amount at startup by configuring the Java virtual machine at system startup.

vfs

Reports on any installed virtual file systems

```
  )ws vfs
Installed virtual file systems
  type:pass_through access:rw  scheme: qdl-vfs  mount point:/pt/      current dir:(none)
  type:zip          access:r   scheme: qdl-vfs  mount point:/zip/     current dir:(none)
  type:mysql        access:rw  scheme: qdl-vfs  mount point:/mysql/   current dir:(none)
  type:memory       access:rw  scheme: qdl-vfs  mount point:/ramdisk/ current dir:(none)
```

(Font size adjusted to show the formatting.) In this case there are four and e.g. the zip VFS it is mounted with read only (**r**) access. See the reference manual on how to mount them and unmount them and use them generally.

save *file*

Save the current workspace to the given file. Relative references are resolved against the qdl root directory (which you can see since it is saved in the environment).

clear

This command will remove all state and return it to the exact condition of starting a clean workspace, except that environment variables are not affected. All variables, functions, imports and any other state will be lost. This is useful if, for instance, you need to start a different project or if, e.g.something went wrong and you really need to just get rid of everything. A typical invocation looks like

```
    )ws clear
workspace cleared
```