The QDL Configuration File

Version 1.3.1

Configuration File Basics

The basic format of the file is

```
<config>
   <qdl name=”X”>
      <!-- stuff →
   </qdl>
   <qdl name=”Y”>
      <!-- stuff →
   </qdl>
  <!-- more configurations -->
</config>
```

Each file consists of qdl elements which are named. Names are just strings and can be anything you like as long as they are unique within the file. There may be many of these and you use them by invoking the name. So here would be a typical user-defined

```
java -jar qdl.jar -cfg /path/to/file/cfg.xml -name X
```

So the path to the above file is /path/to/file/cfg.xml and inside the file, the qdl tag with name X is to be used. You may have many configurations in a file for different needs and just invoke them.

Aliases.

You may also make several configurations then refer them using aliases:

```
<config>
   <qdl name=”default” alias=”test-mysql-5.4.3”/>
   <qdl name=”test-mysql-5.4.3”>
      <!-- stuff →
   </qdl>
   <qdl name=”test-mysql-6.0”>
      <!-- stuff →
   </qdl>
  <!-- more configurations -->
</config>
```

In this case, there is an alias for the configuration named `test-mysql-5.4.3` and you would simply use the name “default” when referencing it. Later, if you needed to change the configuration to `test-mysql-6.0` you would just change the alias. The idea is that you can set your configuration files (which may be awkward to do on a server, *e.g.* especially if this is called by a complex set of startup scripts) and then make changes in the configuration file

Aliases are transitive in that you can have them refer to other aliases like

```
<config>
  <qdl name=″A″ alias=″B″/>
  <qdl name=″B″ alias=″C″/>
  <qdl name=″C″ …
```

Thus invoking “A” would resolve to “C”. If there were a cycle, an exception would be thrown.

File imports

You may also import another configuration file.

```
<config>
    <file include="/path/to/server-cfg.xml"/>
   <qdl name=″A″ alias=″B″/>
   <!-- stuff→>
</config>
```

In this case, all of the configurations in the indicated file are loaded and may be referred to henceforth as if they were part of the current file. A typical use might be

```
<config>
    <file include="/path/to/server-cfg-ver1.xml"/>
    <file include="/path/to/server-cfg-ver2.xml"/>
    <file include="/path/to/server-cfg-ver3.xml"/>
    <qdl name=″default-version-1″ alias=″version-1.1″/>
    <qdl name=″default-version-2″ alias=″version-2.5.4″/>
    <qdl name=″default-version-3″ alias=″version-3.0.0-alpha″/>
</config>
```

In this case, each version in question has its own set of configurations and a default from each is set. This allows for management of *extremely* complex server installs. Then again, you might not need any of it.

A complete example

This is a typical bare-bones workspace configuration’

```
<config>
    <qdl name="minimal">
        <workspace verbose="true"
                   echoModeOn="true">
            <home_dir>/home/your_name/dev/qdl</home_dir>
            <env>etc/qdl.properties</env>
        </workspace>
    </qdl>
</config>
```

Assuming that this resides in the file `/path/to/cfg.xml`, you could invoke the interpreter using this:

```
java -jar qdl.jar -cfg /path/to/cfg.xml -name minimal
```

The Configuration Reference

Each section corresponds to an entry in the configuration file.

config

This is the outermost tag and includes at least one qdl tag. This is so all of the qdl configuration can be kept separate in case there are other types of configuration in the file. It has neither attributes nor any other possible entry than qdl elements.

qdl

Attributes

Name	Req ?	Default	Description
assertions_on	N	true	Enabled or disable assertions. If disabled, no assertions will be processed, effectively ignoring them.
debug	N	off	Enable debug mode level. These are off, trace, info, warn, error and severe. If you enable this then the debug() function will print debug messages to the console. This is useful in cases of, e.g., server side scripting where debug messages are usually captured and sent someplace like the syslog. generally debugging is done in QDL using the say() command. There is also the log() command that writes to the currently active log.
enabled	N	true	This enables/disables this configuration, so it will/will not be loaded.
name	Y	-	(**required**) The unique identifier for this configuration. It can be any string as long as the it is unique in this file.
numeric_digits	N	15	The default precision for decimals
server_mode	N	false	Server mode means this is running as part of a server, usually as a scripting language. This turns off things like file I/O, printing, prompting and other features. It is only needed in a specific context.
script_path	N	-	A colon separated list of paths (including virtual file system paths) to be searched for scripts. This effects the script_run and script_load commands.
module_path	N	-	A colon separated list of paths (including virtual file system paths) to be searched for modules. This effects the module_load.
lib_path	N	-	Used in conjunction with enable_library_support. This is the path that will be used to search for scripts.

enable_library_support	N	F	Enable using the lib_path to search for scripts as functions.

Note about library support: There are a couple of ways of extending QDL. One of them is to define your own functions in the workspace. Another is to write scripts and have these called seamlessly as if they were functions. So if you wrote a script **`/path/to/my/script/my_script.qdl`** you could set the lib_path to be **`/path/to/my/script`** and inside of QDL issuing something like

```
x:= my_script(arg0,arg1)
```

would find this in the libpath, load it and pass it the two arguments, *i.e.*, it is identical to

```
x:= script_run(my_script('/path/to/my/script/my_script.qdl', [arg0,arg1]));
```

Generally loading scripts this way is slow, however if you are using QDL as a general scripting language in your operating system, this is a good way to make utilities available to it. Note that the lib_path is not the same as the script_path. You can toggle this as a variable with the **)ws set** command too.

boot_script

A script to run on startup, after virtual file systems and named external modules are loaded, but before the system is available. This lets you *e.g.* do additional set up of the environment.

Logging

Attributes

Name	Req?	Default	Description
disableLog4j	N	true	Log 4 Java is a common utility. Unfortunately many projects which QDL relies on use it and if they are misconfigured, then you may get extremely strange messages about missing log4j files or inscrutable output about things you've never heard of. Setting this to **true** will hunt down all instances and terminate them.
logFileCount	N	1	The total number of files to have in the rotation. If the count it less than or equal to 1, then only a single log file is maintained.
logFileName	N	logging.xml	The default file is deposited in the invocation directory. If the entire path is given, that will be used. NOTE: the path must exist or logging will fail to initialize. I.e., no paths are created.
logName	N	qdl	The name prepended to each entry in the log file. This is especially useful if several programs are running and share a single location – a frequent occurrence on a server.
logSize	N	0	The maximum size of the file before the system rolls it over. If rotating files is enabled (by setting the logFileCount greater than 1), then logs are rotated. Otherwise, the log file is simply

			over-written
debug	N	false	Enable sending trace level messages to the currently active log.

Example

```
<logging logFileName="log/qdl.log"
         logName="qdl"
         logSize="100000"
         logFileCount="2"/>
```

This sets the relative file to **`home_dir`**`/log/qdl.log`, sets a maximum log size to 100,000 bytes (at which point it will be rotated), Entries will be prefixed with the tag “qdl”. Finally, this will allow 2 such files named qdl.log.0 and qdl.log,1 and will swap them out as needed. Look at the dates and timestamps on the files if you need to see which the current one is.

editors

This section allows you to configure external text mode editors for use with QDL. Each has a name and one may be active at once. The default is the line editor (named **line**) that works with all terminal types and all versions of Java.

Inside the *editors* tag is a list of *editor* elements with the following properties

Name	Req?	Default	Description
name	Y	-	The human readable (nick)name you want to use for this.
exec	Y	-	The path to the executable for this editor.
clear_screen	N	false	See below.

The **clear_screen** option needs some explanation. Most terminals support what is called private mode which allows a text mode application to pop over to full screen and back. Both nano and vim do this. However, this sometimes does not work (this is an implementation issue with the terminal or perhaps the terminal type does not support it right), so when you exit the editor, garbage is still on the screen. This will issue a standard (as in ISO 6429) call to the terminal to clear it. Not great but at least you get your screen back without a bunch of garbage the editor left. Sorry, but its the best we can do in the general case.

Example

```
<editors>
    <editor name="nano"
            exec="/bin/nano"
            clear_screen="false"/>
    <editor name="vim"
            exec="/usr/bin/vim"/>
   </editor>
</editors>
```

In this case, there are two editors (the active one can be set either in the workspace section of this configuration of in the active workspace itself.) The first is the *nano* editor, the second is *vim*.

virtual_file_systems

This tag has no attributes but server to hold all of the vfs elements.

vfs

This is an entry for a virtual file system. All virtual file systems have the following attributes and entries in common:

Attributes

Name	Req?	Default	Description
type	Y	false	The type of file system. This refers to the implementation. See below
access	N	r	The permissions for this file system. “r” refers to readable and “w” to writeable, so to have read and write access, you need to set this to “rw”.

Supported virtual file system types

The supported types are

- pass_through – passes through to the underlying file system.
- memory – only in memory
- mysql – backed by a mysql database
- zip – a mounted zip file. This is always read only, but allows you to treat the file as if it were just another directory.

Each type has specific configuration parameters that are detailed below.

Scheme

This is a specific name for the virtual file system. All paths prefixed with it plus a “#” will refer to this. There is no requirement that, *e.g.* all database VFS have the same scheme. Schemes are logical so you can keep them straight.

mount_point

Where in the virtual file system you want this mounted. So if this is mounted at /A/B then all paths that start with scheme#/A/B will be resolved in to this file system

Supported virtual file system types

The following are the specific configuration options for each type of file system.

pass_through

A pass-through virtual file system effectively treats a directory on the system as if it were its own separate file system. Note that since in server mode there is not direct access to the underlying file system (servers usually run with enhanced permissions, so letting a regular user be the administrator for the system is a terrible idea), but a directory may be set aside and access only to that (read only access too, if desired) may be granted. The only configuration parameter is the directory that is to server as the root. A common use is to have a library of scripts on disk and add a passthrough in read-only mode to it. Scripts can then be loaded (and managed easily by the admin) without granting any access otherwise.

root_dir

The absolute path to the directory that is the root of this VFS.

Example

```
<vfs type="pass_through"
     access="r">
    <root_dir>/home/ncsa/dev/qdl</root_dir>
    <scheme><![CDATA[qdl-vfs]]></scheme>
    <mount_point>/pt</mount_point>
</vfs>
```

In this case, the directory `/home/ncsa/dev/qdl` is on the server and that will be mounted in the VFS at `/pt`. The system is read-only. A note about the <![CDATA[]]> tag. This lets you put any text inside a tag, so special characters and such are allowed. The reason that scheme and such are elements and not just attributes is because attributes have to go in between double quotes and are very limited in the sorts of characters they allow. Use as needed, though some people always use them.

memory

A file system that is wholly in memory. This always starts up completely empty and lasts for exactly the duration that the system is running. It may be used for extremely fast in-memory only access of files, for instance, it can be used to initially cache files from a slower source (such as downloading over the web) and stashing them. This is effectively a ramdisk.

Example

```
<vfs type="memory"
     access="rw">
   <scheme><![CDATA[qdl-vfs]]></scheme>
   <mount_point>/ramdisk</mount_point>
</vfs>
```

In this case, a in-memory VFS will be created and mounted at /ramdisk.

mysql

This is for a VFS backed by a mysql database. This has a separate connection element whose attributes are

Name	Req?	Default	Description
username	Y	-	The name of the user with permission to access this database
password	Y	-	The password of the user
schema	Y	-	The schema (name of the database) that the table resides in
tablename	Y	-	The name of the table
parameters	N	-	Additional connection parameters. These may or may not be required and vary based on how the database system is configured.

Example

```
<vfs type="mysql"
     access="rw">
   <scheme><![CDATA[qdl-vfs]]></scheme>
   <mount_point>/mysql</mount_point>
   <mysql username="qdl-user"
          password="w00fw00f"
          schema="qdl"
          tablename="vfs_table"
          parameters="useJDBCCompliantTimezoneShift=true"/>
</vfs>
```

In this example, the VFS is mounted at /mysql and the scheme is qdl-vfs.. The system is readable and writeable. The connection parameters are also provided.

zip

This mounts a zip (compressed file, also includes java jar files). Due to the internal structure of such files, they cannot be made writeable. (In point of fact, the entire file has to be recompressed and rewritten with each update because zip files keep a table of things that have been compressed. This means while we *could* make one writeable, the overhead and performance are truly miserable as an active file system.) One use is to zip up a library of modules and scripts, then send mount it. This allows for instance, configuring several systems with identical tools easily. There is one additional parameter required:

zip_file

This is the absolute path to the zip file.

Example

```
          <vfs type="zip"
                 access="r">
                <zip_file>/home/ncsa/dev/resources/vfs-test/vfs-test.zip
                </zip_file>
                <scheme><![CDATA[qdl-vfs]]></scheme>
                <mount_point>/zip</mount_point>
            </vfs>
```

In this case, the file located at `/home/ncsa/dev/resources/vfs-test/vfs-test.zip` will be mounted in the VFS at `/zip`.

modules

This element simply holds module elements and has no attributes or other elements.

Module

A module is an encapsulated unit of code. You may write them in QDL using the **`module[]body[]`** construct or you can also write them in Java and import them.

Attributes

Name	Req?	Default	Description
type	Y	-	The type of the module. Options are **java** or **qdl**.
import_on_start	N	F	Load this when the system starts. This means it will be instantly available to the user. Otherwise it will have to wait for use until the user calls the import function.

class_name

(Java only modules) the name of the module. This must have a no-argument constructor and conform to the requirements for a java module.

path

(For native QDL modules only) the path to the file that contains a QDL module.

Example

```
      <modules>
            <module type="java"
                    import_on_start="true">
```

```
            <class_name>edu.uiuc.ncsa.qdl.extensions.QDLLoaderImp</class_name>
        </module>
        <module type="qdl">
            <path>relative/path/module.qdl</path>
        </module>
    </modules>
```

In this case, two modules are loaded. The java module is imported but the qdl module is not.

workspace

This element corresponds to the workspace and its function (as opposed to configuring the interpreter.)

Attributes

Name	Req?	Default	Description
autosaveInterval	N	600000	The length of time between automatic saves of the workspace. This may be given with no units (assumed to be milliseconds) or may have either “ms” or “sec” appended.
autosaveMessagesOn	N	true	Whether ot not to print out the regular save messages whenever the workspace is autosaved.
AutosaveOn	N	false	Enable the autosave feature for the workspace. You must configure saves to work right
echoModeOn	N	true	Enable echoing of all single lines. Effectively this means when you enter a single line, any needed ; is added to the end and if there is output, the result is displayed
verbose	N	false	Be chatty during operation. This prints out extra information.
showBanner	N	true	Show the banner at workspace startup. Only a minimal startup message will be shown so you know the workspace started normally.
prettyPrint	N	false	Turn on vertical printing of stems by default.
use_editor	N	false	If an external is to be used. You must set the editor_name if this is enabled.
editor_name	N	-	the (nick)name you gave to the editor in the **editors** section of this file.
save_dir	N	-	The directory to use for saved workspaces. If this is not set, then a default of $QDL_HOME/var/ws will be used. You can also set this in the workspace.

home_dir

This is the root directory for this session. All relative files paths will be resolved against it.

env

This is the environment file. It is a standard java properties file and once the workspace is up and running, the contents can be seen by issuing

)env

Read up in the qdl workspace reference more about how the environment functions.

Example

```
<workspace verbose="true"
           echoModeOn="true">
     <home_dir>/home/ncsa/dev/qdl</home_dir>
     <env>etc/qdl.properties</env>
</workspace>
```

Note that the home directory is set, so that the environment file is resolved against it, *viz.*, the environment file, being relative (so it does not start with a “/”) is assumed to be at `/home/ncsa/dev/qdl/etc/qdl.properties`