cQuery Documentation

Release 0.2.2

Marcus Ottosson

March 09, 2016

Contents

Welcome to the documentation of cQuery v0.2.2.

CHAPTER 1

Getting Started

1.1 Overview

cQuery implements a decentralised method of querying file-system contents using "selectors" within a library for Python and a command-line interface.

```
$ cquery search .Asset
/path/to/asset1
/path/to/asset2
/path/to/asset3
```

1.1.1 How it works

Given a directory such as:

```
$ /projects/spiderman/assets/Peter
```

cQuery answers the questions:

- What is my Asset?
- What is my Project?
- What rigs do I have?
- Am I Orange?
- How many shaders do I contain?

1.1.2 Quick example

Here is how it might look when tagging and querying a content hierarchy for a feature animation project.

```
$ cd MyAsset
$ cquery tag .Asset
$ cquery search .Asset
/path/to/MyAsset
```

1.1.3 Decentralised

The traditional method of identifying content on a file-system is via keys and maps. E.g. a database stores a *key* along with an associated *path* in a map. The *path* can then be looked up through the map via *key*.

```
$ query MyKey
$ /path/to/corresponding_content
```

This means that all keys are stored in one spot, the map, and all content stored elsewhere, the file-system. cQuery joins maps with content in an effort to eliminate this separation and in so doing eliminates many of the added responsibilities such as synchronisation and increased barrier-to-entry.

1.1.4 Selectors

The idea of selectors are adopted from CSS3 and its use in jQuery (from which the name cQuery was derived). jQuery allows users to operate on the Document Object Model, or DOM, using CSS3 selectors to locate the appropriate Nodes. Similarly, cQuery operates on the Content Object Model, or COM, using CSS3 selectors to locate the appropriate folders.

1.1.5 Up or Down

Starting from a root directory, a query can either be made up or down. To find descending matches of a given directory, you would use DOWN. To instead query for ascending matches, you would use UP. To query one-self only, you would use NONE.

Here is how that might look when used in Python:

```
>>> # Find the associated project of the asset Peter
>>> first_match("/projects/spiderman/assets/Peter",
...             selector='.Project', direction=UP)
>>>
>>> # Find all textures
>>> for match in matches("/projects/spiderman/assets/Peter",
...             selector='.Texture', direction=DOWN):
...     print match
>>>
>>> # Is this asset a Hero?
>>> True if first_match("/projects/spiderman/assets/Peter",
...                     selector='.Hero', direction=NONE) else False
```

1.1.6 Architecture

cQuery works upon directories tagged with metadata to indicate its class, ID or name. The tagged directories may then be queried, either from outside a hierarchy looking in or from within a hierarchy looking out.

For tagging, cQuery uses the Open Metadata specification [1], the process is simple - for each subdirectory within a directory, recursively look for a file by name stored within the Open Metadata container. If a match is found, return the absolute path to said directory. The name of this file is the "selector" argument of your query.

[1] For more information on Open Metadata, see here https://github.com/abstractfactory/openmetadata

1.1.7 Performance

cQuery operates on the hard-drive and is a seek-only algorithm and as such doesn't perform any reads. Despite this however, disk-access is (seemingly) the prime bottle-neck. A cQuery prototype has been implemented in both Python and Go for performance comparisons, here are some results:

Python

```
# Scanning a hierarchy of 3601 items
# 1 queries, 7 matches in 1.494072 seconds
# 1 queries, 7 matches in 1.480471 seconds
# 1 queries, 7 matches in 1.477589 seconds
#   Average time/query: 1.484044 seconds

# Scanning a hierarchy of 47715 items
# 1 queries, 14 matches in 19.888399 seconds
# 1 queries, 14 matches in 20.078811 seconds
# 1 queries, 14 matches in 19.879660 seconds
#   Average time/query: 19.948957 seconds
```

Go

```
# Scanning a hierarchy of 3601 items
# 1 queries, 7 matches in 1.425702 seconds
# 1 queries, 7 matches in 1.420373 seconds
# 1 queries, 7 matches in 1.419541 seconds
#   Average time/query: 1.421872 seconds

# Scanning a hierarchy of 47715 items
# 1 queries, 14 matches in 18.015012 seconds
# 1 queries, 14 matches in 17.951607 seconds
# 1 queries, 14 matches in 17.994924 seconds
#   Average time/query: 17.987181 seconds
```

For some more encouraging results in file-system search and indexing, here are some resources:

- http://www.voidtools.com/
- http://rlocate.sourceforge.net/
- http://www.lesbonscomptes.com/recoll/
- http://grothoff.org/christian/doodle/
- http://xapian.org/

1.1.8 Roadmap

There are currently two mutually exclusive goals of cQuery. One is to fully implement the DOM as it exists in Javascript and XML. The DOM closely resembles that of a file-system and has undergone vast amounts of research and development in an effort to find the best method of traversing it. The other is for the development of cQuery to focus its efforts on CSS3-style selectors exclusively, making it much more nimble and easier to maintain.

If cQuery is not the place for a full implementation of the DOM, another project will take its place shortly.

1.2 Quickstart

This page will guide you through setting up cQuery and running your first query. If you experience any problems here or are looking for more information about each step, head on the the Tutorial for a full overview or Example to experience a demo project.

1.2.1 Install

```
$ pip install cquery
```

Note: cQuery is a pure-Python library and as such will require an installation of Python.

Note: cQuery has been tested on Python 2.7.7 on Windows 8.1 and Ubuntu 13.01

1.2.2 Some Content

```
$ cd c:/projects
$ mkdir spiderman/assets/Peter
$ mkdir spiderman/assets/Goblin

$ mkdir spiderman/shots/1000
$ mkdir spiderman/shots/2000
```

1.2.3 Tag

Note: cQuery ships with an executable. On Windows, you may have to add the Python27\scripts directory to your PATH.

```
$ cd spiderman/assets
$ cquery tag .Asset --root=Peter
$ cquery tag .Asset --root=Goblin
$ cd ../shots
$ cquery tag .Shot --root=1000
$ cquery tag .Shot --root=2000
```

1.2.4 Query

```
$ cd ..
$ cd ..
$ cquery search .Asset
c:/projects/spiderman/assets/Peter
c:/projects/spiderman/assets/Goblin
```

And that's it. Now you can tag and query via the command-line.

1.2.5 Python

From Python, you could query like this:

```
import os

import cquery
for match in cquery.matches(os.getcwd(), selector='.Asset'):
    print match
```

Next we'll have a look at a more thorough version of this quickstart.

1.3 Tutorial

Note: Much of the below in in the works. Keep tabs on the github repository for more immediate updates or if you're interested in collaborating.

This page is meant as a more thourough version of the Quickstart. If you haven't been through it yet, it is recommended that you do so before proceeding.

1.3.1 Process

cQuery is simple and depends on as little knowledge and setup as possible. As such, to get started with cQuery there are three steps to fulfill:

1. Install cQuery
2. Tag content
3. Query content

Once set up, a more general workflow may look like this:

1. Tag content
2. Query content

1.3.2 Installation

To get started, install cQuery like this:

1.3.3 Content

cQuery is designed to work with tens of millions of subdirectories but for the purposes of this tutorial, let's stick with a minimal set of possible matches.

```
$ cd c:/projects
$ mkdir spiderman/assets/Peter
$ mkdir spiderman/assets/Goblin

$ mkdir spiderman/shots/1000
$ mkdir spiderman/shots/2000
```

1.3.4 Advanced

Note: The following is on the roadmap for cQuery but isn't part of it yet. We are looking for contributors interested in file-based search optimisations - if that's you, contact us. If you know anyone, spread the word.

cQuery is designed to facilitate very large content hierarchies (> 20 million individual directories) and as such provides a few alternatives for optimisation.

No Optimisations

Per default, cQuery is designed to work out-of-the-box with little or no setup. This means making every query live and will in some cases be cause for a noticable slowdown depending on the amount of directories are involved in a query. For upwards queries, this is usually not noticeable (~0.001s/level) but downwards queries could potentially touch millions of targets and as such may take several minutes to complete.

Local Daemon

The simplest level of optimisation is one that indexes results during a query. Once a query has been performed, the results are stored in the currently running process and help speed up subsequent queries.

Dedicated Daemon

The next level of optimistation involves running a dedicated daemon that performs an either live, at a fixed interval or at events. The dedicated daemon has the advantage of being persistent across runs and facilitating a multi-user setup.

Central Deamon

Finally, the central deamon, like the dedicated daemon, is persistent but as opposed to the dedicated deamon the central daemon facilitates a multi-user/multi-site usage.

1.4 Example

A project-based example of cQuery - Building a simple publishing tool.

1.4.1 What is publishing?

Et etc.

1.4.2 Requirements

1.4.3 Implementation

CHAPTER 2

Reference

2.1 Reference

cQuery consists of a single function: `cquery.matches()` and takes at least two arguments: a *root* and *selector*. The root determines from where within a hierarchy to start a query whereas the *selector* determines what to query for. Additional functions are either helpers or conveinence measures.

`matches`	Yield matches at absolute path *root* for selector *selector* given the direction *direction*.
`first_match`	Convenience function for returning a first match from `matches()`.
`convert`	Convert CSS3 selector *selector* into compatible file-path

2.1.1 cquery.lib

cQuery - Content Object Model traversal.

`cquery.lib.`**`CONTAINER`**
str

Metadata storage prefix Metadata associated with directories are prefixed with a so-called "container". In Open Metadata land, this means an additional directory by the name of *~openmetadata.Path.CONTAINER*

`cquery.lib.`**`UP`**
flag

Search direction A flag for `cquery.matches()` specifying that the content traversal should proceed up from the *root* directory. Use this to retrieve a hierarchy of matches.

`cquery.lib.`**`DOWN`**
flag

Search direction The opposite of the above UP. Use this to retrieve multiple matches within a given hierarchy, located under *root*

`cquery.lib.`**`matches`**(*root*, *selector*, *direction=4*, *depth=-1*)
Yield matches at absolute path *root* for selector *selector* given the direction *direction*.

When looking for a first match only, use *`first_match()`*

Parameters

- **`root`** (*`str`*) – Absolute path from which where to start looking
- **`selector`** (*`str`*) – CSS3-compliant selector, e.g. ".Asset"

- **direction** (*enum, optional*) – Search either up or down a hierarchy
- **depth** (*int*) – Depth of traversal; a value of -1 means infinite

Yields **path** (*str*) – Absolute path of next match.

cquery.lib.**errors**
Collection of errors occured during os.walk (NotImplemented)

Raises OSError – ENOTDIR is raised if path *root* is not a directory

Example

```
>>> import os
>>> paths = list()
>>> for match in matches(os.getcwd(), ".Asset"):
...     paths.append(match)
```

cquery.lib.**first_match**(*root*, *selector*, *direction=4*, *depth=-1*)
Convenience function for returning a first match from *matches()*.

Parameters

- **root** (*str*) – Absolute path from which where to start loo
- **selector** (*str*) – CSS-style selector, e.g. .Asset
- **direction** (*enum*) – Search either up or down a hierarchy

Returns **path** – Absolute path if successful, None otherwise.

Return type str

Example

```
>>> import os
>>> path = first_match(os.getcwd(), ".Asset")
```

cquery.lib.**convert**(*selector*)
Convert CSS3 selector *selector* into compatible file-path

Parameters **selector** (*str*) – CSS3 selector, e.g. .Asset

Returns Resolved selector

Return type str

Example:

```
$ .Asset  --> Asset.class
$ #MyId   --> MyId.id
```

2.1.2 cquery.cli

cQuery command-line interface

cquery.cli.**main**()
A group allows a command to have subcommands attached. This is the most common way to implement nesting in Click.

Parameters `commands` – a dictionary of commands.

CHAPTER 3

Additional Information

3.1 Schemaless

cQuery for schemaless directory structures

3.1.1 Why Schemaless?

cQuery doesn't solve the issues surrounding data-modeling. When defining a *schema*, you map a digital landscape onto metaphors more easily understood than their digital counterpart. You then build tools upon this map, with the intent that the landscape rarely, ideally never, changes. Although this works and has worked for a long time, change is inevitable and schemas simply doesn't cope all that well with it - i.e. a change to a schema, depending on its magnitude, may well break your tools.

Again, cQuery doesn't solve this issue, data-modeling is inherently a human problem, not a technical one. What cQuery does however is move the barrier at which change starts to affect the work you build upon it so that you are free to start building long before you know how your digital landscape will end up looking.

Generally, there is a direct analogy between a schemaless style and dynamically typed languages. And as with such languages, it is extra important to explicitly document the definition, motivation and purpose of each decision made. What cQuery allows you to do is to move this decision-making process onto a later stage. As they say, procrastination leads to wiser decisions.

See also

- Data and Reality, Kent
- Managing Data in Motion, Reeve
- Data Modeling Essentials, 3rd ed., Graeme
- http://martinfowler.com/articles/schemaless/

3.1.2 Motivation

Traditionally, prior to commencing a new project, you would spend a little time on figuring out an appropriate directory structure to encapsulate the data this project will generate. Something like:

```
o project
  o- assets
     o- peterparker
     o- loislane
  o- shots
```

```
    o- 1000
    o- 2000
    o- 3000
    o- 4000
```

Upon which you then set out to build your tools. But what if your next project also features sequences, or levels? What if the hierarchy is located on a Unix-drive or a network share depending on which computer accesses the data? The number of variables upon venturing out on any projects can never be assumed and will continuously change and the work you build on-top will have to facilitate this change.

3.2 Glossary

schema Terminology borrowed from the database industry, a schema is merely a pre-defined convention for storing data and may be as simple as a verbal agreement to store all images under /img or assets under /projects/assets.

3.3 Roadmap

We aim cQuery to be suitable for content hierarchies of any depth and width at a minimum of 10 queries/second. cQuery should work without any setup and may additionally be set up with an indexing daemon. The daemon would either run locally or remotely and maintain a live representation of either all or pre-defined hierarchies. The daemon would optimise common queries. It should be possible to query via the daemon directly, so as to allow the daemon to persist the entire index in-memory for additional performance gain.

CHAPTER 4

Indices

- genindex
- modindex

C

C

D

E

F

M

S

U