Advanced Technology Research

Quixote: A Cookbook

Revision 0.4

June 20, 2005

Document prime: Chris Hobbs
Document Number: Zig05-0006
Contributors: John Bell

Contents

CHAPTER 1

Introduction

1.1 This Document

Quixote is a web application framework written in Python and this document is intended for programmers coming fresh to Quixote without much (or any) knowledge or understanding of it. It does *not* specifically address the needs of programmers familiar with earlier versions of Quixote who need to upgrade their software.

Information about, and the source for, Quixote can be found at http://www.mems-exchange.org/software/quixote/ and http://quixote.ca/. Unfortunately, in getting to grips with Quixote, I found the code good and the documentation of such a standard that I spent a lot of time reading code. In particular it came as a surprise to find that the latest documentation of the `Form` class apparently aligned with the previous generation of the `Form` class code. This is the document that I wish had existed when I started to read the code.

This document addresses version 2.0 of the Quixote code and incorporates useful changes and corrections suggested by Mike Orr and Larry Tjoelker in emails to the Quixote mailing list http://mail.mems-exchange.org/mailman/listinfo/quixote-users dated 9th June 2005.

1.2 More Information

Other Quixote tutorials exist (e.g., http://darcs.idyll.org/~t/projects/quixote2-tutorial/) and the reader is advised to explore them all. My only criticism of many of the explanations and tutorials that I read when trying to get to grips with Quixote was that they often referred to version 1.0 of the program and the upgrade to version 2.0 was not made in a backwards-compatible manner.

1.3 Updates

This document has been prepared using LaTeXand is issued in pdf format. If anyone wishes to update the document to correct or enhance it then they should send an email to Chris Hobbs (cwlh@nortel.com) to get access to the source. Any suggestions about improvement would also be welcome to the same address.

1.4 Disclaimer

I am not a member of the Quixote design and development team and so the information given in this document may be wrong. All I can say is that it worked for me.

CHAPTER 2

A Quixotic Survey

Quixote has a number of effectively independent components:

- a mechanism for tying a URI entered by a user into a browser with a particular Python method, the method being invoked when the URI is entered. Quixote assumes an underlying architecture where the application logic is independent of the browser interface and this component of Quixote ties the two together. This mechanism is described in section 3 starting on page 5.
- a library of functions to assist with the creation of the HTML for common screen widgets (text boxes, radio buttons, etc.) and the extraction of data entered by the user into those widgets. For information about this library see section 4.1 starting on page 10.
- a library of functions to assist with the creation and analysis of an HTML form: interspersing widgets with other layout information. For information about this library see section 4.2 starting on page 12.
- an extension to the Python language, known as the Python Template Language (PTL), which makes it slightly more convenient to generate HTML. This language is described in section 5 starting on page 16.

These components can be used independently or together.

CHAPTER 3

The Quixotic Core

3.1 General Structure

Figure 3.1 illustrates the components of a system containing Quixote. There are other ways of using Quixote (see, for example, reference [WHIT]), particularly in an embedded environment where a full Apache-like web server would be inappropriate. In the remainder of this document the structure shewn in figure 3.1 is assumed.

The steps in programming a Quixote application are:

1. define a class (called a *Directory*) which describes the linkage between URLs selected by a user and the code fragments. This step is described in section 3.2 and an example is given in appendix B.1.
2. define a class (called a *Publisher*) which describes some of the configuration of the system (where to put log files, how to handle debug messages from the application, etc.) and encompasses an instance of the *Directory* class described above. This step is described in section 3.4 and an example is given in appendix B.1.
3. write a program (perhaps a CGI script) which does little else other than create and execute an instance of the *Publisher* class. This step is described in section 3.3 below and an example is given in appendix B.2.
4. write the application code (in Python or PTL). An example of such code is given in appendix B.1,

3.2 URI Mapping to Methods

One of the key building blocks of Quixote is the link between a URI typed by the user into his or her browser and the piece of code which needs to be invoked when that URI is

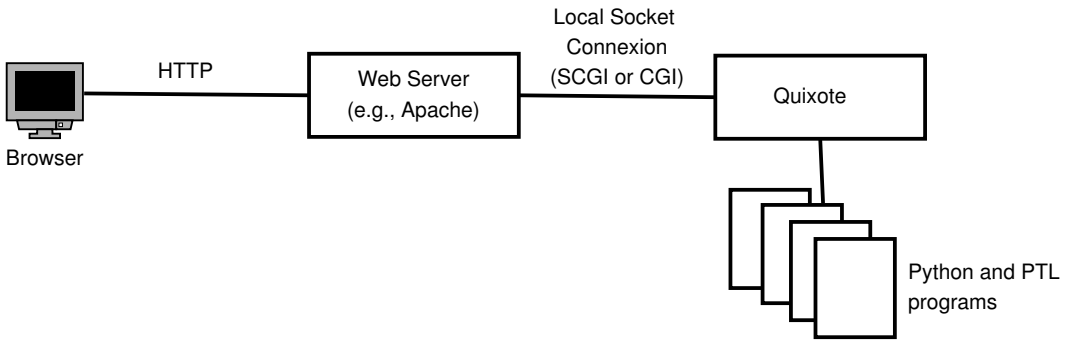

Figure 3.1: Quixote Architecture

accessed.

Creating this link is achieved by creating an instance of a *Directory* class which supports a number of methods. This instance is then passed as a parameter to a `Publisher` as described in section 3.3.

The Quixote release includes the class `Directory` in the file `directory.py` and this provides a skeleton which may be copied and extended or which may be used as a superclass.

The methods and properties which need to be instantiated are:

_q_exports This is a list of strings or tuples specifying which methods are to be externally visible for use in URIs. For example, the list:

```
_q_exports = [ "doit1", "doit2", ("externalName", "doit3") ]
```

indicates that `doit1` in a URI should cause the method `doit1()` to be invoked, `doit2` in a URI should cause the method `doit2()` to be invoked and `externalName` in a URI should cause the method `doit3()` to be invoked. If `_q_exports` contains an empty string then this is interpreted as an implicit `("", "_q_index")` tuple so that incoming references without a explicit path result in `_q_index()` being invoked.

_q_lookup(self, component) This method can be used to create URIs dynamically since it is called by Quixote to resolve a URI entered at the browser. It would typically return an instance of a Directory but could also return a method or even a string.

This method could, for example, be used to "create" a web page for each member of the current (May 2005) English Cricket Team. While it would be possible to create a genuine page for each of Vaughan, Bell, Flintoff, Giles, Harmison,

Hoggard, Jones (Geraint), Jones (Simon), Lewis, Strauss, Thorpe and Trescothick, it would also be possible construct URIs such as http://prefix/vaughan, http://prefix/bell, http://prefix/trescothick and intercept the call using _q_lookup. This method would be passed the component ("vaughan", "bell", etc.) and could return a dynamically-created Directory which has collected information about the particular player.

There is no end to the fun which can be had here: it would for example, as suggested by Larry Tjoelker, be possible to create "pages" with URLs comparing the performance of two players such as http://prefix/lewis/hoggard. It is this flexibility which makes it important to select a URL structure *before* beginning to write a Quixote application—advice which comes from hard experience in my case.

An example of this is given in the Quixote distribution in Quixote-2.0/-demo/extras.ptl which uses Quixote-2.0/demo/integers.ptl to "create" a web page for every (positive) integer. Not as much fun as the English Cricket Team but a useful example nevertheless.

3.3 The Publisher

In the same way that two versions of the Form class exist (see page 12), version 2.0 of Quixote comes with two versions of the Publisher class: publish.py and publish1.py. The later of these is publish.py and that is the one discussed here.

The Publisher class has a constructor with the following signature:

```
def __init__(self,
                 root_directory,
                 logger=None,
                 session_manager=None,
                 config=None,
                 **kwargs)
```

where:

root_directory is an instance of a class as described in section 3.2.

logger gives an instance of a Logger object. If none is given then a Default_Logger as defined in logger.py is used.

session_manager gives an instance of a SessionManager object as defined in session.py. Such a manager is responsible for creating sessions, setting and reading session cookies, maintaining the collection of all sessions, and so forth. There is one SessionManager instance per Quixote process. If no manager is specified then a null session manager is used. This supports memory-based sessions.

config gives an instance of a `Config` object as defined in `config.py`. As described in section 3.4 below, *either* a `Config` object *or* the kwargs may be given to define the configuration variables listed in table 3.1 but not both.

kwargs specify the configuration parameters (see section 3.4 below.

An instance of the Publisher class is created to handle each "transaction": i.e., when using plain CGI, the instance will handle exactly one HTTP request and then be destroyed, when using FastCGI, then the instance will handle every HTTP request handed to that driver script process.

Once created, the Publisher exports a number of methods:

log(msg) to write a message to the log system.

The Quixote system is started by making a call to

```
quixote.server.cgi\_server.run()
```

passing it the constructor for the Publisher.

3.4 Configuration

Configuration of the Quixote system is handled by a class called `Config` and the parameters listed in table 3.1 are supported. The values of these parameters may be set by passing a `Config` instance to the Publisher's constructor or by using the `kwargs` parameter.

3.4.1 Quixote Versions and Backward Compatibility

In addition to the `publish.py`, `publish1.py`, `publish2.py` and the `form.py`, `form1.py`, `form2.py` confusion described in other sections, there have been other non-backward compatible changes made in Quixote and, in many cases, the documentation and examples have not kept up with the changes. The Quixote 2.0 release is, however, accompanied by a file describing the necessary code changes which need to be made to earlier code to make it work with Quixote 2.0: `upgrading.txt` in the `doc` directory.

You may find examples set up for earlier versions of Quixote which have items now unsupported:

- in earlier versions of the Publisher code there was also a method `read_config()` which allowed a configuration to be read from a file. This method no longer exists.
- in earlier versions of the configuration there was a parameter `DEBUG_LOG`. This is no longer supported—debug information now always goes to the error log.

Parameter	Default	Meaning
ERROR_EMAIL	None	E-mail address to which to send application errors
ACCESS_LOG	None	Filename for writing the Quixote access log
ERROR_LOG	None	Filename for logging error messages and debugging output. If `None`, everything is sent to stderr
DISPLAY_EXCEPTIONS	None	Controls what's done when uncaught exceptions occur. If set to 'plain', the traceback will be returned to the browser in addition to being logged, If set to 'html' and the cgitb module is installed, a more elaborate display will be returned to the browser, showing the local variables and a few lines of context for each level of the traceback. If set to None, a generic error display, containing no information about the traceback, will be used.
COMPRESS_PAGES	False	Compress large pages using gzip if the client accepts that encoding
FORM_TOKENS	False	If true, then a cryptographically secure token will be inserted into forms as a hidden field. The token will be checked when the form is submitted. This prevents cross-site request forgeries (CSRF). It is off by default since it doesn't work if sessions are not persistent across requests.
SESSION_COOKIE_NAME	"QX_session"	Name of the cookie that will hold the session ID string
SESSION_COOKIE_DOMAIN	None	Domain to which the session cookie is restricted.
SESSION_COOKIE_PATH	None	
MAIL_FROM	None	Default for the "From" header and the SMTP sender for all outgoing e-mail. Required if sending email otherwise system will crash.
MAIL_SERVER	"localhost"	Mail server configured to relay outgoing emails
MAIL_DEBUG_ADDR	None	If set, then all e-mail will actually be sent to this address rather than the intended recipients. This should be a single, bare e-mail address.

Table 3.1: Configuration Parameters

CHAPTER 4

Widget and Form Classes

The two classes described in this chapter are not essential to the Quixote operation: they are convenience classes for creating the HTML of widgets and forms and extracting the information entered by the user.

4.1 Widget Classes

The classes which inherit from Widget are illustrated in figure 4.1. These are mostly self-explanatory (and, with the exception of `FloatWidget`, `IntWidget`, `OptionSelectWidget` and `ListWidget`, all non-abstract classes map directly to HTML elements).

Basically these classes are just convenience functions which obviate the need for manual production of the HTML for the various widgets and manual analysis of the results when the user has entered data.

The Widget class defines a function `render()` which returns the HTML for the widget as illustrated here (linebreaks in the output have been added for the reader's convenience and do not actually occur):

```
bash-2.05b$ python
Python 2.3.5 (#2, Mar 26 2005, 17:32:32)
[GCC 3.3.5 (Debian 1:3.3.5-12)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import quixote
>>> from quixote.form import widget
>>> x = widget.StringWidget('name', size=20)
>>> print x.render()
<div class="StringWidget widget">
```

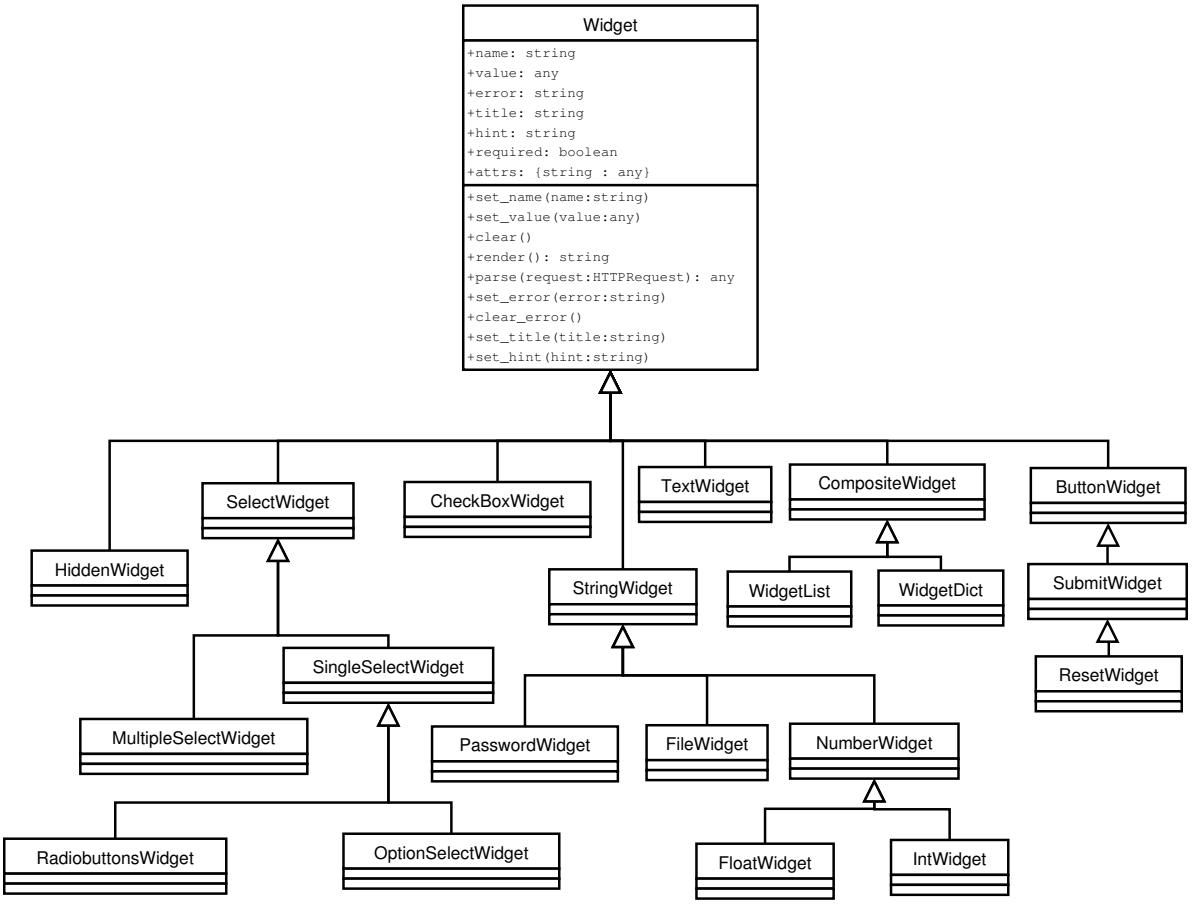

Figure 4.1: Widget Classes

```
  <div class="content">
    <input type="text" name="name" size="20" />
  </div>
</div>
<br class="StringWidget widget" />
```

Each widget has a number of attributes stored with it and these may be set and changed as required. Generally they form part of the HTML generated on a call to `render()`:

- an error string. This is typically displayed close to the field and contains details of an error condition. It is set by calling the `set_error(error : string)` method and cleared by calling `clear_error`.
- a title string. This is typically displayed close to the field and contains a title for the field. It is set by calling the `set_title(title : string)`

- a hint string. This is typically displayed when the user passes the cursor over the field. It is set by calling the `set_hint(title : string)`

The use of these functions is illustrated in the exchange below (again, line-breaks in the output have been artificially inserted):

```
bash-2.05b$ python
Python 2.3.5 (#2, Mar 26 2005, 17:32:32)
[GCC 3.3.5 (Debian 1:3.3.5-12)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import quixote
>>> from quixote.form import widget
>>> x = widget.StringWidget('name', size=20)
>>> x.set_error("This is the error string")
>>> x.set_title("This is the title string")
>>> x.set_hint("This is the hint")
>>> print x.render()
<div class="StringWidget widget">
  <div class="title">
    This is the title string
  </div>
  <div class="content">
    <input type="text" name="name" size="20" />
    <div class="hint">
      This is the hint
    </div>
    <div class="error">
      This is the error string
    </div>
  </div>
</div>
<br class="StringWidget widget" />
```

4.2 Form Class

If you have just downloaded version 2 of Quixote and do not intend to look at older documentation and examples, then the remainder of this paragraph can be ignored. In fact, trying to get to grips with version 2, I found that I needed this explanation. There are actually two Form libraries and their designation has changed throughout the history of Quixote:

- in the beginning there was `Form`

- then a second, replacement library was created, called `Form2`, and `Form` and `Form2` coëxisted.
- then, with the move to Quixote release 2.0, `Form` was renamed to `Form1` and `Form2` was renamed to `Form`.

This document assumes that the later library is being used.

The Form class represents a form as presented to, and completed by, the user. It has the following attributes:

- the widgets (other than subclasses of `SubmitWidget` or `HiddenWidget`) which should appear on the screen, held in as a list of widgets in `self.widgets`.
- the widgets subclassed from `SubmitWidget` which are held as a list of widgets in `self.submit_widgets`.
- the widgets subclassed from `HiddenWidget` which are held as a list of widgets in `self.hidden_widgets`.
- the names of the widgets held as a dictionary containing entries of the form { name : widget } in `self._names`.

The constructor of a Form has the signature:

```
def __init__(self,
             name=None,
             method="post",
             action_url=None,
             enctype=None,
             use_tokens=True,
             **attrs):
```

where:

name is the HTML name of the form

method is `"post"` or `"get"`

action_url is the URL of the method that will action the form

encType must be `"application/x-www-form-urlencoded"` or `"multi-part/form-data"`

use_tokens is a Boolean and, if set to `True`, indicates that a unique token should be generated for each form. This prevents many cross-site attacks and prevents a form from being submitted twice.

attr contains other keyword arguments for conversion to additional HTML attributes in the `<form>` tag.

If the parameter is not given, then `Form.__init__()` tries to set it to

```
{ 'class': 'quixote' }
```

to set the CSS `class` attribute. Indeed, if the parameter *is* set but does not contain the key "class" then this entry is added to the parameter.[1] For information about such attributes (and the `class` attribute in particular), see section 7.5.2 of the W3C document which can be found at http://www.w3.org/TR/REC-html40/struct/global.html#adef-class

[1] As of June 2005, this is recognised as a bug in `Form.__init__`—the class attribute cannot, in fact, be set.

Method	Parameter	Notes
is_submitted		returns True if the form has been submitted by the user
has_key	name	returns True if the named widget is in the form
get	name	returns the value of the named widget or value of the default parameter if widget does not exist
	default=None	
get_widget	name	return the named widget or None if widget doesn't exist
get_submit_widgets		return a list of the submit widgets
get_all_widgets		return a list of all the widgets
set_error	widget name error message	set the error display for a particular widget
has_errors		cause the widgets to parse themselves and return True if any has an error
clear_errors		cause all the widgets to parse themselves and clear any outstanding errors (see also set_error)
get_submit		get the name of the submit button which was used to submit the form. If the form is submitted but not by a known submit button then True is returned.
add	widget_class	create a new widget and add it to the class. widget_class is the widget's class
	name	name of the widget.
	*args	arguments for creating widget.
	**kwargs	arguments for creating widget.
add_*****	name value=None **kwargs	call add() for a ****Widget
render		render the form as an HTML string

Table 4.1: Useful Form Methods

CHAPTER 5

The Python Templating Language

PTL is a variant of Python designed to make the preparation of HTML a little easier. PTL files are compiled into standard Python `.pyc` files. Once in this form, files which originated as PTL can be used as if they had originated from normal Python code.

If PTL is to be used without pre-compilation then a call must be made to `quixote.enable_ptl()` as illustrated in section B.3.

A PTL function is identified by having `[plain]` or `[html]` inserted into the function definition as shewn in `Login.ptl` in appendix B.1 on starting on page 22:

```
def success [html] ():
    widgets = self.form.get_all_widgets()
    for widget in widgets:
        print widget,widget.parse()
    '<html>'
    '<head><title>Nortel Community Network</title>'
    '</head>'
    '<body>'
    '<h1>Thanks, that looks good'
```

Basically, the difference between a normal Python function and a PTL function is that, instead of discarding unassigned expressions, it applies `str()` to them[1] and appends them to a string which forms the return value from the function.

Thus instead of writing:

```
def doit(x,y):
    ans = "The sum is %d" % (x+y)
    return ans
```

[1] actually it's a lot cleverer than that but this is a useful simplification

the function can simply be written:

```
def doit [plain] (x,y):
    "The sum is %d" % (x+y)
```

The only exception to this rule is expressions (normally function calls) which resolve to `None`. `None` is not appended to the returned value.

The difference between using `[plain]` and `[html]` is that, when `[html]` is used, special HTML characters are correctly escaped (e.g., & becomes &). Note that this substitution only occurs with strings given to a function as a parameter, global variables, global variables, return values, etc—anything that's not a literal (it is assumed that these are the dangerous strings since they may be being used by an external attacker as a "cross-site scripting" bug). This example illustrates the principle:

```
Python 2.3.5 (#2, Feb  9 2005, 00:38:15)
[GCC 3.3.5 (Debian 1:3.3.5-8)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from quixote import enable_ptl
>>> enable_ptl()
>>> import tryit
>>> x = 'Here is <HEAD> some "text"'
>>> print x
Here is <HEAD> some "text"
>>> tryit.doit(x)
<htmltext 'abcdHere is &lt;HEAD&gt; some &quot;text&quot;a&b<>xyz'>
```

where the `tryit.ptl` file contains:

```
def doit [html] (x):
    "abcd%s" % x
    "a&b<>"
    "xyz"
```

Note that the less-than and greater-than signs passed into the function `doit()` are correctly escaped to `<` and `>` but that the less-than, greater-than and ampersand built into the `doit()` function are not escaped.

If `tryit.ptl` were precompiled to `tryit.pyc` then the above example can be simplified to:

```
Python 2.3.5 (#2, Feb  9 2005, 00:38:15)
[GCC 3.3.5 (Debian 1:3.3.5-8)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import tryit
>>> x = 'Here is <HEAD> some "text"'
```

```
>>> print x
Here is <HEAD> some "text"
>>> tryit.doit(x)
<htmltext 'abcdHere is &lt;HEAD&gt; some &quot;text&quot;a&b<>xyz'>
```

APPENDIX A

Sequence of Events

Analysing the sequence of events when a request is received and processed by Quixote is not trivial (and it varies depending on the type of session being used, etc.). This appendix gives a summary of a very simple interaction wherein a user at a browser requests a particular URI, is returned a form to complete, submits the form and the fields are checked in some way.

Refer to figure A.1 when reading this sequence.

1. The browser requests a particular URI from the web server, assumed here to be Apache.

2. Apache knows nothing of Quixote and does its normal action of translating the URL into a path and invoking the cgi script on the path, here assumed to be `village.cgi`.

3. `village.cgi` passes the function to create an instance of the project-specific `Publisher` class to Quixote by calling `run()` with the function as a parameter.

4. Quixote, running in the context of `village.cgi`, invokes the function and creates an instance of the project-specific `Publisher` class. This class instance creates project-specific instances of the `Directory` class and the `Config` class containing respectively information about linking URIs to modules and details of how logging should occur, etc.

5. Using the publisher, Quixote invokes the class which will create the HTML for the form to be displayed to the user (here assumed to be `Login.ptl`).

6. `Login.ptl` creates an instance of the `Form` class but `Login.ptl` is, at this time, unaware of whether it has been called to process the user's input into the form or, as here, to present the form to the user.

7. `Login.ptl` invokes the `is_submitted()` function on the created form to determine whether or not the form has been completed. This strikes this programmer as an unusual invocation: `Login.ptl` has just created the form instance and, strictly speaking, that form instance can have no idea of whether "it" has been filled in by the user.

8. The form object uses the publisher to determine whether or not the request from the user contained a completed form with the appropriate fields and finds that it did not. The form object therefore returns from `is_submitted()` with a value of False.

9. Knowing that it has been invoked by an original request from the user, `Login.ptl` invokes the `render()` function on the form to generate the HTML necessary to display the form to the user. This HTML is passed back to Apache by Quixote and control is returned to `village.cgi` which exits.

10. Apache sends the HTML to the originating browser which displays it for the user.

11. **Note that, at this point, no state is being held on the server.**

12. The user completes the fields in the form (e.g., name and password) and sends the information back to the same URL.

13. Steps 2 to 7 occur again but this time the call to `is_submitted()` returns True to indicate that the form has been completed by the user.

14. `Login.ptl` retrieves the entered values from the form and checks them for correctness. Assume that the password is, in fact, not correct.

15. `Login.ptl` sets an error message against the widget holding the password and again renders the form into HTML.

16. The form, together with error message, is sent back to the user for correction and the process starts again at step 12.

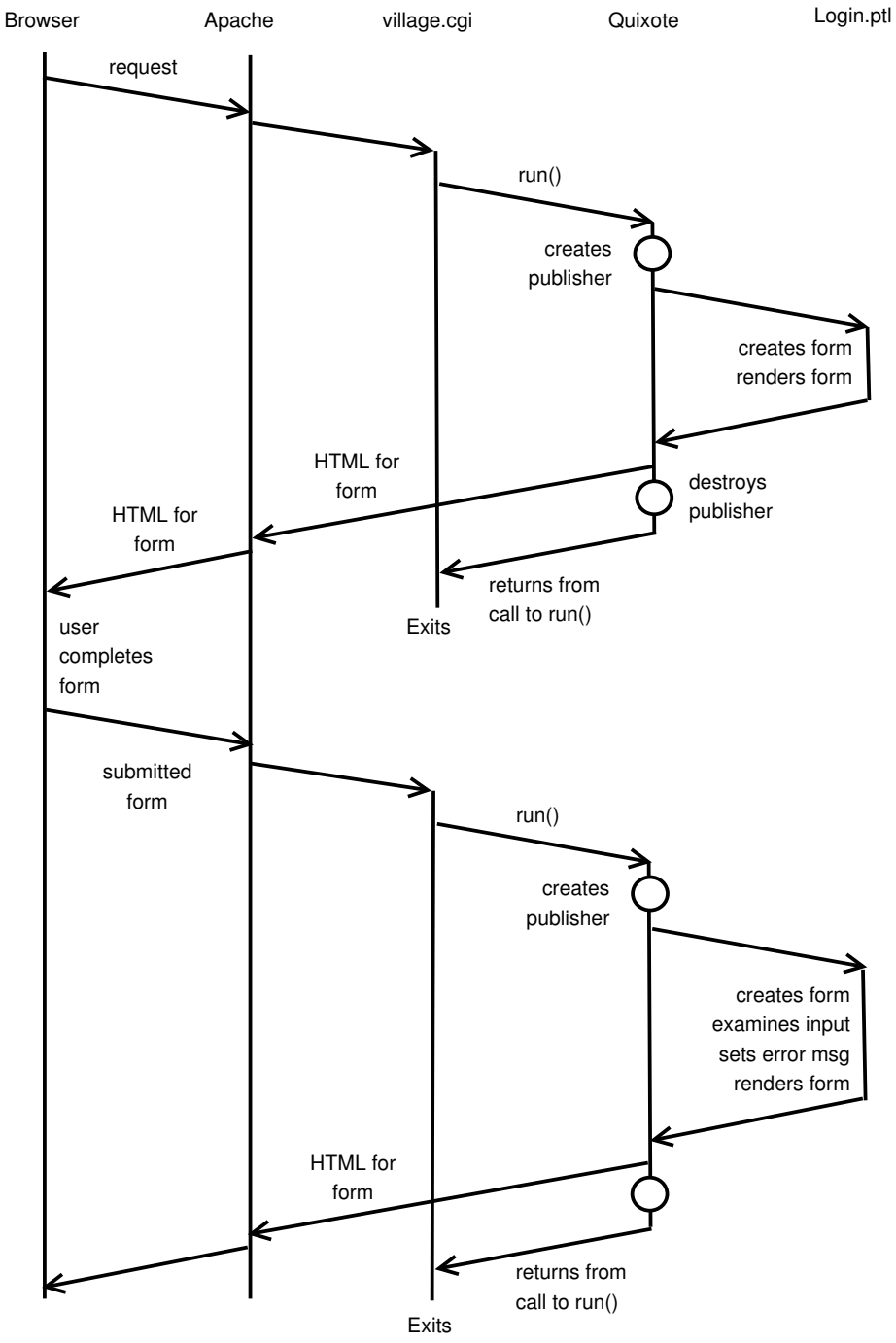

Figure A.1: Example Interaction

APPENDIX B

Example

This appendix contains the code required to display a simple form (username and password) to a web browser and accept and verify the information entered by the user. It is assumed that there is a simple text file, called `passwords.txt` which contains a list of names and md5-encrypted passwords, separated by a colon. For example,

```
Python:$1$ab$flouUiJtpLHNgkgpN7JR60
cwlh:$1$ab$5RYdb3usKBFXhbyJEtl7U.
```

defines two users (*Python* and *cwlh*) with their associated (encrypted) passwords.

The remaining sections in this appendix give sample code laid out in the same order as defined in section 3.1 on page 5.

B.1 Creating a Root Directory Instance

The code for a simple directory instance is given below. As the comment on myDirectory says, for pedagogical reasons the code has been split between `myDirectory.py` and `Login.ptl`. Obviously, the application code in `Login.ptl` which actually checks the entered username and password is not useful for understanding Quixote but it is included here to ensure that this example contains all of the code necessary for a working system.

```
#!/usr/bin/python

# ************************************************
#   module  myDirectory.py
#   purpose definition of the mapping from URI to
#               method xxxxxxxxxxxxxxxxxx
#   note    for pedagogical reasons this file has
```

```
#               been split into two: Login.ptl
#               contains the other part of the
#               script
#   author  chris hobbs
#   written april 2005
# ************************************************

import quixote
from quixote.directory import Directory

import Login

class myDirectory(Directory):
    _q_exports = ['', 'hello', 'login']

    def _q_index(self):
        return '''<html>
                    <body>Welcome to Chris' Demo.  Here is a
                    <a href="login">link</a>.
                    </body>
                  </html>
                '''

    def hello(self):
        return '<html><body>Hello world!</body></html>'

    login = Login.LoginDirectory()
```

```
#!/usr/bin/python

# ************************************************
#   module  Login.ptl
#   purpose ptl code for the login screen
#               for xxxxxxxxxxxxxxxx
#   author  chris hobbs
#   written april 2005
# ************************************************

import time
import string
import md5crypt
import quixote
```

```
from quixote.directory import Directory
from quixote.directory import Resolving
from quixote.form import widget
from quixote.form import Form
from quixote.form import StringWidget
from quixote.form import PasswordWidget
from quixote.form.css import BASIC_FORM_CSS

# ***********************************************
#   class   LoginDirectory
#   purpose display the prompts for a username and
#               password
# ***********************************************

class LoginDirectory(Resolving, Directory):

    _q_exports = [ '', 'login', 'handleInput' ]

    def _q_index [html] (self):

        # ************************************************
        #   method      render
        #   purpose     write out the HTML for the login
        #                   screen
        # ************************************************

        def render [html] ():
            '<html>'
            '   <HEAD>'
            '       <TITLE>Zigamorph Configuration</TITLE>'
            '   </HEAD>'
            '   <BODY BGCOLOR = white>'
            '       <CENTER>'
            '       <H1>Nortel xxxxxxxxx Network '
            '         <IMG SRC="/gifDirectory/nortelLogo.gif"></H1>'
            '       <P>'
            '       <H3>Welcome to the configuration interface. <H3>'
            '       <P> <H3>Please enter your username and password below</H3>'
            '       <FORM METHOD = post>'
            self.form.get_widget('name').render()
            self.form.get_widget('password').render()
```

```
        '           <P>'
        self.form.get_widget("ok").render()
        '       </FORM>'
        '       </CENTER>'
        '   </BODY>'
        '</HTML>'

    # ************************************************
    #   method      success
    #   purpose     handle the condition whereby the
    #                   user typed in a valid password
    # ************************************************

    def success [html] ():
        widgets = self.form.get_all_widgets()
        for widget in widgets:
            print widget,widget.parse()
        '<html>'
        '<head><title>Nortel Community Network</title>'
        '</head>'
        '<body>'
        '<h1>Thanks, that looks good'

    # ************************************************
    #   method      checkPassword
    #   purpose     check whether a given
    #                   username/password combination is
    #                   valid
    #   input       username
    #               password
    #   output      True if the combination is valid,
    #                   False otherwise
    # ************************************************

    def checkPassword(userid, passwd):
        # if there is already someone logged in (and their
        # session hasn't timed out) then the password might
        # as well be bad because we're not going to let them in

        passwd_file = open('passwords.txt', 'r')
        allPasswords = passwd_file.readlines()
```

```
        passwd_file.close()
        for password in allPasswords:
            combo = string.split(password, ":")
            if userid == combo[0]:
                encrypted_pw = md5crypt.unix_md5_crypt(passwd, 'ab')
                if encrypted_pw[0:20] == combo[1][0:20]:
                    return True
        return False

    # ************************************************
    #   Mainline code for this form
    # ************************************************

    # We don't know whether we're here to render (display)
    # the form or to check the input entered by the user.
    # Either way, we'll need the form so we'll create it.

    self.form = Form(enctype="application/x-www-form-urlencoded")
    self.form.add(StringWidget, "name", title="Name",
                                size=20, required=True)
    self.form.add(PasswordWidget, "password", title="Password",
                                size=20, maxlength=20, required=True)
    self.form.add_hidden('time', value=time.time())
    self.form.add_submit("ok")

    if not self.form.is_submitted() or self.form.has_errors():
        return render()
    else:
        if checkPassword(self.form.get_widget('name').parse(),
                         self.form.get_widget('password').parse()):
            return success()
        else:
            self.form.set_error('password',"Invalid Username/Password")
            return render()
```

B.2 Creating a Publisher Instance

```
#!/usr/bin/python

# ************************************************
```

```
#   module  myPublisher.py
#   purpose definition of the publisher for the
#               xxxxxxx  system
#   author  chris hobbs
#   written april 2005
# ************************************************

import quixote
from quixote.publish import Publisher
from quixote.config import Config

import myDirectory

def createPublisher():

    # define our configuration

    conf = Config(display_exceptions='plain',
                            access_log = 'access.txt',
                            error_log = 'error.txt')

    # Create a directory instance

    directory = myDirectory.myDirectory()

    # and then create our publisher

    pub = Publisher(directory,config=conf)

    return pub
```

B.3 Running Quixote

When the classes have been defined for a publisher and directory, then Quixote can be run. Sample code to make this happen is given below.

The only real point to notice is the call to `enable_ptl()` which is the call which allows PTL programs to be run in the Python environment.

```
#!/usr/bin/python
```

```
# ************************************************
#   module  Main cgi module for the xxxxxxxx system
#   author  chris hobbs
#   written april 2005
# ************************************************

import quixote
from quixote import enable_ptl
from quixote.server.cgi_server import run

# Install the import hook that enables PTL modules.
enable_ptl()

import myPublisher

# Enter the publishing main loop
run(myPublisher.createPublisher)
```

APPENDIX C

Licence

Quixote is *not* “public-domain” software. It is available for use in accordance with the terms of its licence. Before using Quixote, ensure that the licence terms (below) are acceptable.

CNRI OPEN SOURCE LICENSE AGREEMENT FOR QUIXOTE-2.0

IMPORTANT: PLEASE READ THE FOLLOWING AGREEMENT CAREFULLY. BY COPYING, INSTALLING OR OTHERWISE USING QUIXOTE-2.0 SOFTWARE, YOU ARE DEEMED TO HAVE AGREED TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT.

1. This LICENSE AGREEMENT is between Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (“CNRI"), and the Individual or Organization (“Licensee") copying, installing or otherwise using Quixote-2.0 software in source or binary form and its associated documentation (“Quixote-2.0").

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Quixote-2.0 alone or in any derivative version, provided, however, that CNRI’s License Agreement and CNRI’s notice of copyright, i.e., “Copyright ©2005 Corporation for National Research Initiatives; All Rights Reserved" are retained in Quixote-2.0 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Quixote-2.0, or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Quixote-2.0.

4. CNRI is making Quixote-2.0 available to Licensee on an “AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF QUIXOTE-2.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF QUIXOTE-2.0 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING QUIXOTE-2.0, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia’s conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Quixote-2.0 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Quixote-2.0, Licensee agrees to be bound by the terms and conditions of this License Agreement.

Bibliography

[WHIT] “White Paper: Quixote for Web Development”, available at http://www.quixote.ca/overview/paper.html

Index

>THIS IS **THE WAY**

>THIS IS **NORTEL**