The SkunkWeb Developers Guide

Release 3.3

Drew Csillag and Jacob Smullyan

January 29, 2007

CONTENTS

CHAPTER

ONE

Introduction

This document is intended for the SkunkWeb developer that needs to get more into the inner guts of the SkunkWeb server. This document will describe the inner architecture of the server, the file formats used, the core APIs, the APIs of foundation services, as well as how to write SkunkWeb services.

It is assumed that you are familliar with the SkunkWeb server, i.e. you don't need to be told that what `<:component foo.comp:>` would do, and that you know Python well.

CHAPTER

TWO

Services

The crux of most of what SkunkWeb does lies in services, of which there are four kinds.

Provider Provides direct functionality, but few, if any API functions.

Linkage Mainly links in some external (from `pylibs`) code, and may provide functionality of it's own.

API Provides little, if any functionality on it's own, but provides API functions to do various things.

Foundation Services that form the foundation that services of other types are built upon.

2.1 Provider Services

Many of the provider services are protocol adaptors, those being the `aecgi`, `fcgiprot`, `scgi`, `httpd` and `remote`, whose purpose is merely to provide support for a network protocol.

There are others in this category. These are the services that provide direct functionality and really don't have any API to speak of.

The `basicauth` service, while deprecated, provides the ability to do basicauth (i.e. browser-based) authentication. It has an API inasmuch setting of the `REMOTE_USER` and `REMOTE_PASSWORD` is an API. The newer `auth` service provides the functionality that the `basicauth` service provides and more and is the replacement now that `basicauth` is deprecated.

The `rewrite` service allows for some pre-request URL rewriting. In addition, named groups in the matched regular expression will be added to `CONNECTION.args` if the appropriate configuration variable is set. See the SkunkWeb Operations Manual for details.

The `userdir` service allows you to provide simple SkunkWeb services to the users on your machine. If the URL starts with ˜*username*, it will serve the pages from the directory named by the `userDirPath` configuration variable (`public_html` by default) in their home directory.

The `extcgi` service allows SkunkWeb to directly handle CGI programs. This is useful in the event that you either aren't running Apache (or some other SkunkWeb compatible server), or that your configuration is such that having your main HTTP server handle it is either not aesthetically pleasing or just plain annoying.

The `pycgi` service also allows SkunkWeb to directly handle CGI programs, but `pycgi` only works for CGI programs that are Python scripts. It handles them in such a way that, unlike `extcgi`, does not require a call to `fork`, so should provide better performance in the case that you are running a Python CGI program.

2.2 Linkage Services

Linkage services mainly exist to link other stuff into skunkweb.

mysql Loads the MySQL pylib library to do connection caching. See Appendix B.

postgresql Loads the PostgreSql pylib library to do connection caching. See Appendix B.

oracle Loads the Oracle pylib library to do connection caching. See Appendix B.

ae_component Loads the AE pylib library that is the component rendering infrastructure, see Appendix A.

templating Hooks up the `ae_component` service up to the `web` service to make templates web accessible.

psptemplate Hooks up the psp pylib to the AE infrastructure.

2.3 API Services

The API services are those that could not (easily) be arranged to otherwise be linkage services, as they are more tightly integrated because of the tasks they perform, and they provide some direct functionality also.

auth product remote_client sessionHandler

2.3.1 The `auth` Service

```
class authorizer:
    def __init__(self, ......):
    """
    The ...... will be filled with the contents of
    Configuration.authAuthorizerCtorArgs when this object is instantiated.
    """

    def checkCredentials(self, conn):
    """
    Examine the connection however you see fit to see if the
    connection has the credentials needed to view the page
    return true to accept, false to reject
    """

    def login(self, conn, username, password):
    """
    Take the username and password, validate them, and if they are
    valid, give the connection credentials to validate them in the
    future (i.e. make it so checkCredentials() returns true).  If your
    authorizer is doing basicauth-style authentication, this method is
    not required.
    """

    def logout(self, conn):
    """
    Remove any credentials from the browser.  This method is not required
    for methods (specifically, basicauth) that do not have the concept of
    logging out.
    """

    def authFailed(self, conn):
    """
    If the connections credentials are rejected, what to do.  For
    this, unless you are using a basic-auth means, inheriting from
    RespAuthBase is probably sufficient (it will bring up a login
    page), otherwise inheriting from BasicAuthBase and providing an
    appropriate validate function is sufficient.
    """

    #this method is required only if using one of the base classes that
    #require it
    def validate(self, username, password):
    """
    Return true if the username/password combo is valid, false otherwise
    """
```

```
class RespAuthBase:
    """
    This is a base class for those authorizers that need to use a login page.
    That’s pretty much any of them, with the exception of BasicAuth, as it
    doesn’t need it since the browser "puts up a login page".

    The loginPage ctor arg is the page to show.  It *can* be in the protected
    area.
    """


class AuthFileBase: #base class for those that auth against a basicauth file
    """
    Base class for authorizers that will use a simple basicauth file to
    validate user/password combinations

    The authFile ctor argument is the file that we will validate against
    """


class BasicAuthBase: #base class that does basicauth
    """
    This class does browser based basicauth authentication where it pops up
    the login box by itself.

    This is a mixin class and is not usable by itself.  You must provide a
    way to validate (by subclassing BasicAuthBase and defining a validate
    function) the username and password that is obtained."""

class CookieAuthBase: #class that does basic cookie authentication
    """
    This class implements a simple cookie based authentication.  Depending
    on how you want to do things, you may not (and probably don’t) want to
    have the username and password encoded directly in the cookie, but
    definitely *do* use the armor module to protect whatever you do decide to
    put into the cookie as it will make them virtually tamperproof.

    As with BasicAuthBase, thisis a mixin class and is not usable by
    itself.  You must provide a way to validate (by subclassing CookieAuthBase
    and defining a function) the username and password that is obtained.
    """
```

```
class SessionAuthBase: #class to do auth using sessions
    """
    This class uses the session object (you must have the sessionHandler
    service loaded) to handle authentication.  This is also probably not
    something that you would really want to use in a production setup as:
    a) you probably don’t want to store the password in the session.  Not that
       it’s inherently evil, but it’s one less thing you have to worry about
       leaking in the event that something gets screwed up.
    b) checking the credentials might just be to check if the username is set
       since all of the session info (minus the session key) is stored on
       the server and not the browser, we don’t have to worry as much that
       somebody fooled around with us

    But what is here is usable as a starting point and will run.

    As with BasicAuthBase, thisis a mixin class and is not usable by
    itself.  You must provide a way to validate (by subclassing SessionAuthBase
    and defining a function) the username and password that is obtained.

    """
```

2.3.2 The `product` Service

The `product` service is a service to support SkunkWeb products, which can be installed as archive files (zip, tar, or tgz), or as directories in the SkunkWeb product directory. This service requires that the documentRootFS be a MultiFS (at least prior to scoping); the assumption is that MultiFS will become the default fs.

The product loader is configured to load by default all product archives in the product directory, but can be configured to load any arbitrary subset.

The loader opens the MANIFEST file in each archive, looks for dependencies stated therein, and loads any products therein listed, raising an error for circular or missing dependencies. For each load, it adds mount points to the MultiFS for the docroot and, by means of an import hook in the vfs which permits python modules to be imported from the vfs, adds the libs (if any) to sys.path. Services specified in the MANIFEST are then imported.

The MANIFEST is essentially an init file for the product which integrates it into the SkunkWeb framework. It must contain the product version, and can also contain product dependencies and services (python modules that use SkunkWeb hooks, to be imported at product load time). See manifest.py for details.

By default, a product’s docroot will be mounted at products/¡product-name¿, relative to the SkunkWeb documentRoot. This can be altered by modifying Configuration.defaultProductPath, which will affect all products, or by adding the product-name to the Configuration.productPaths mapping.

This service will also contain a utility for creating products, creating the MANIFEST file, byte-compiling the python modules, and creating the archive file.

2.3.3 The `remote_client` Service

The `remote_client` service exists to provide access to components exposed on other SkunkWeb servers via the `AE.Component.callComponent` function, and by extension the `<:component:>` tag. What this service does is make it so that if you specify the component name as `swrc://`*host*`:`*port/path/to/component.comp*, it will call the component */path/to/component.comp* by contacting a (presumably) existing SkunkWeb installation that has the `remote` service configured to listen on *host*`:`*port*.

2.3.4 The `sessionHandler` Service

The `sessionHandler` service adds three functions to the `CONNECTION` object that the top-level component sees. The three functions are:

`getSession(create=0, **cookieParams)` if create is true, will create (and return) a new session (in the cookie, with extra paraters in `cookieParams`), otherwise will return an existing session, if it exists, otherwise returns `None`.

`getSessionID(create=1)` get session id from cookie, or create it if `create` is true, otherwise returns `None`.

`removeSession()` kills the session cookie.

2.4 Foundation Services

Foundation services are those which are used to build other services, but in and of themselves, provide no directly useable functionality.

2.4.1 The `requestHandler` Service

The `requestHandler` service is used as the basis of all of the network request based services (directly, or indirectly) and handles the communication to the client (in whatever form that may take).

PreemptiveResponse (responseData) – respdata will be sent via protocol.marshalResponse

from requestHandler/requestHandler.py Hook sequences

BeginSession Just after connection is gotten, scoped for IP,PORT or UNIXPATH (job, sock, sessionDict)

while 1: InitRequest after the request data has been received (job, reqdata, sessiondict) HandleRequest called to actually handle the request (job, reqdata, sessiondict) PostRequest called after response sent (job, reqdata, sessionDict) CleanupRequest right after PostRequest (job, reqdata, sessionDict)

EndSession called when session ends.

has DocumentTimeout stuff

requestHandler.requestHandler.addRequestHandler(handler – should be subclass (or provide the same interface as) requestHandler.protocol.Protocol, ports)

requestHandler.protocol.Protocol

```
class Protocol:
    """
    abstract class for protocols used in handling request and response
    """

    def marshalRequest(self, socket, sessionDict):
        '''
        should return the marshalled request data
        '''
        raise NotImplementedError

    def marshalResponse(self, response, sessionDict):
        '''
        should return response data
        '''
        raise NotImplementedError

    def marshalException(self, exc_text, sessionDict):
        '''
        should return response data appropriate for the current exception.
        '''
        raise NotImplementedError

    def keepAliveTimeout(self, sessionDict):
        '''
        how long to keep alive a session.  A negative number will terminate the
        session.
        '''
        return -1
```

2.4.2 The web Service

from web/protocol.py

–CONNECTION object is constructed from reqHandler.reqdata HaveConnection (job, conn, sessionDict) – Configuration is scoped PreHandleConnection (job, conn, sessionDict) HandleConnection (job, conn, session)

has Redirect exception

Contains the CONNECTION definition

CHAPTER

THREE

Templating Languages

STML SkunkWeb Template Markup Language

PSP Python Server Pages

Cheetah The Cheetah templating system (http://www.cheetahtemplate.org)

Python Not really a templating language, but included for completeness

3.1 Common to All

`CONNECTION` exists in all of them (in toplevel) in the global namespace. URL (GET and POST) arguments show up in `CONNECTION.args`. Can import SkunkWeb API modules (might be services). Can be components (regular and data), cacheable, etc. Component args show up in global namespace.

3.2 PSP Specifics

Only <% and %> for now, no <%= %> pairs ... yet? No other PSP-style tags either.

3.3 Cheetah Specifics

Precompiled Cheetah templates are executed via the normal Python code execution path. Direct support for Cheetah templates without compilation (i.e. being able to go to http://foo.bar/a_cheetah_doc.tmpl) is nonpresent in SkunkWeb right now, but if there is demand, it could, of course, be added.

For Cheetah templates, `$var` won't fetch item from global namespace by default, so put this at the top of your Cheetah templates and you'll be good to go.

```
#silent self._searchList.append(globals())
```

Caching in Cheetah templates probably doesn't work in SkunkWeb, or if it does, is not as effective as SkunkWeb caching because SkunkWeb does not use threads. If SkunkWeb did use threads, then they would be equally effective in a single-machine environment. In a multi-machine environment, SkunkWeb caching is more effective because the cache can be shared between the machines as it is disk-based.

CHAPTER

FOUR

Writing a Service

Hooks reload gotchas

CHAPTER

FIVE

Writing a New STML Tag

empty tags block tags debugging and vicache.py metadata

CHAPTER

SIX

Adding a Templating Type

1. templating compiler (preferably) or interpreter
2. cached compiled vsn fetcher, or interpretable form instantiator
3. AE Executable
4. mime.types entries
5. adding applicable mime types to AE.Executables.executableByTypes, perhaps overriding those for text/html and text/plain
6. add to interpretMimeTypes somehow

APPENDIX

A

The AE Package

Mention configuration if used outside of SkunkWeb, specifically that the default paths are different (since there is no SkunkRoot)

A.1 Components

A.1.1 `callComponent`

`callComponent``(`*name, argDict, cache = 0, defer = None, compType = DT_REGULAR, srcModTime = None*`)`
Return the output of calling a component.

`name` The path to the component.

`argDict` A dictionary of arguments to the component.

`cache` A boolean specifying that we should attempt to get the output from cache, and failing that, execute it and write the output to cache.

`defer` A boolean specifying that

1. If a recently expired cached version of the component exists give me that for now, and
2. Evaluate the component after the response has been sent and write it to cache.

Obviously, deferral only applies if the `cache` argument is true.

`compType` Either `DT_REGULAR` meaning a regular textual component, `DT_INCLUDE` meaning a textual component that runs in the namespace of the calling component, or `DT_DATA` for a data component. The `DT_` contants are in the `AE.Component` module.

`srcModTime` Modification time of the component source, if known.

A.1.2 The Component Stack

`componentStack`
A list used as the component stack (`componentStack[0]` being the bottom of the stack).

`topOfComponentStack`
The current top of the component stack. You should use `componentStack[topOfComponentStack]` as the top of the stack, not `componentStack[-1]`, since if an exception has occurred, `componentStack[-1]` will point to the stack frame where the exception occurred, and in the case where the exception was handled, the stack may not have been cleaned up yet

`resetComponentStack``()`
Should be called after the top level call to `callComponent` has returned. It clears out the component stack.

Attributes of the ComponentStackFrame

name
The name of the component being executed.

namespace
The namespace dictionary that the component is being executed in.

executable
The execution object of the component (from `AE.Executables`).

argDict
The explict arguments passed to the component.

auxArgs
The auto-arguments passed to the component.

compType
The kind of component this is (data or textual). See Section A.1.1, page 19.

A.2 Caching

Compiled Memory Caching

A.2.1 Filesystem Interface

_statDocRoot(*path*)
If talking to a normal filesystem, this would return `os.stat(path)`.

_getDocRootModTime(*path*)
If talking to a normal filesystem, this would return `os.stat(path)[stat.ST_MTIME]`.

_readDocRoot(*path*)
If talking to a normal filesystem, this would return `open(path).read()`.

A.2.2 The Compile Cache

_getCompiledThing(*name, srcModTime, legend, compileFunc, version, reconstituteFunc = None, unconstituteFunc = None*)
A generic way to get and cache compiled things from cache/disk/memory, returns the compiled thing.

`compileFunc`	takes the name and the source and returns compiled form
`unconstituteFunc`	takes the object and produces a marshal-friendly form
`reconstituteFunc`	takes the unmarshalled form and reconstitutes into the object
`name`	is the documentRoot relative path to the thing
`srcModTime`	the modification time of the source, if known, if not known, say `None`.
`legend`	the label to use in debug messages
`version`	marhallable thing that if the one in the compile cache doesn't match, we recompile

A.2.3 The Component Cache

getCachedComponent(*name, argDict, auxArgs, srcModTime = None*)
returns cached component (or None), source Mod time

`name` Name under which the cached output is stored.

`argDict` Dictionary of explicit component arguments.

`auxArgs` Dictionary of automatic component arguments.

`srcModTime` Generally, modification time of source.

`putCachedComponent`(*name, argDict, auxArgs, out, cache_exp_time*)
Puts the cached form of a component's output to storage.

`name` Name under which the cached output is stored.

`argDict` Dictionary of component arguments.

`auxArgs` Dictionary of automatic component arguments.

`out` The output of the component.

`cache_exp_time` The number of seconds past the epoch when the cached version will expire.

`clearCache`(*name, arguments, matchExact = None*)
Selectively removes cached components from cache.

`name` Name of the component you want to clear.

`arguments` Argument dictionary to match.

`matchExact` If true, will clear **only** if the argument dictionary supplied matches the argument dictionary used to store the component exactly, otherwise, any cached version whose argument dictionary is a superset of `arguments` will be cleared.

A.3 Misc Functions

`AE.Component.rectifyRelativeName`(*path*)
Will return the full path of `path` as taken as relative to the directory of the currently executing component. For example: if the currently executing component is `/foo/bar/baz.comp` calling

```
AE.Component.rectifyRelativeName('.')
```

will return `/foo/bar/` and

```
AE.Component.rectifyRelativeName('doofus')
```

will return `/foo/bar/doofus` and

```
AE.Component.rectifyRelativeName('/absolute/path')
```

will return `/absolute/path`.

A.4 File Formats

Serialized tuple of (*item*, *version*). Cached compiled objs use the `marshal` module for serialization and cached component outputs use `cPickle` for serialization.

A.4.1 Compiled Templates

item is tuple of (*filename*, *python source text*, *python code object*, *metadata*)

A.4.2 Compiled Message Catalogs

item is tuple of (*message dict*, *source file name*)

A.4.3 Compiled Python Code

item is tuple of (*python code object*, *python source text*)

A.4.4 Cached components

Generation of md5 hash (see cachekey.c).

describe file layout and MD5 hash stuff path_to_src/firstbyte/secondbyte/fullhash.(cache—key)

item is dict of

`exp_time` Seconds past the epoch at which this cached representation expires.

`defer_time` when using deferred rendering, the expired component's lifetime is temporarily extended so that it gives the process time to render it and write it back to the cache. When this is the case, this is the time at which it expires again, otherwise -1.

`output` Serialized form of component output.

`full_key` Serialized form of full cache key.

A.4.5 Component Cache Distribution

Take last 16 bits of md5 hash and mod numServers. Insert # between component cache root and normal path to cache. If access to server fails abnormally, don't try it again for failoverRetry seconds, and use failoverComponentCacheRoot in meantime for only the section(s) of the component cache that failed recently.

Database APIs

The `mysql`, `oracle` and `postgresql` services exist to merely import their pylib counterparts (`MySQL`, `Oracle` and `PostgreSql` respectively). The services themselves provide no api, they just provide configuration options to setup the connection caches that the pylib modules hold.

The APIs to the modules are essentially identical. There is really only one function of interest (though some contain other function which you should *NOT* rely on to exist in future releases), and that is `getConnection`. This function takes a connection name (as defined in the `sw.conf` file) and returns the connection associated with it. The connections are lazily made, i.e. they are not created when the SkunkWeb child process starts, but when the first call of `getConnection` with the connection name is made in a process.

Here's an example of typical usage using PostgreSQL:

```
import PostgreSql
db = PostgreSql.getConnection('test') # <--- as defined in sw.conf
cur = db.cursor()
cur.execute('select * from testTable')
result=cur.fetchall()
cur.close()
```

INDEX