
Aspen

Release trunk

Chad W. L. Whitacre

today

Zeta Design & Development
<http://www.zetadev.com/software/aspn/>
Email: chad@zetaweb.com

Abstract

Aspen is a web server for highly extensible Python-based publication, application, and hybrid websites.

Introduction

Aspen is designed around the idea that there are basically two kinds of websites, publications and applications, differentiated by their organization and interface models. A *publication* website organizes information into individual pages within a hierarchical folder structure that one navigates by browsing. In an *application* website, on the other hand, data is not organized into hierarchical pages but is dealt with via a non-browsing interface such as a search box.

The HTML version of this documentation is an example of a publication website: a number of hypertext documents organized into sections. If we weren't using LaTeX (or if I knew how to use it better), the sections would probably be encoded in folders. [Gmail](#) is a pure application website, one which organizes and presents information non-hierarchically. Most websites, however, are hybrids. That is, within an overall hierarchical organization you will find both individual pages of information as well as applications such as a site search feature, or a threaded discussion forum.

Publication websites are actually a subset of application websites, of course. An application site can use any interface metaphor; a publication is an application that uses the familiar folder/page metaphor to organize and present its information. Therefore, every website is fundamentally an application.

Aspen enables the full range of websites: publications, applications, and hybrids. It uses the filesystem for the hierarchical structure of publication and hybrid websites, and provides a mechanism for including applications within that hierarchy.

An Aspen website is a collection of files, self-contained within a single directory, called the *root* of the website (cf. [Apache's DocumentRoot directive](#)). In general, URLs map directly to the filesystem. That is, given a root of:

```
/usr/local/www/example.com
```

A request for `/foo.html` would serve a file at:

```
/usr/local/www/example.com/foo.html
```

If all you want to do is serve static files, then that's most of what you need to know.

To extend an Aspen website, you use a UNIX-style userland located within a directory under the website root named `__` (that's two underscores), also called the website's *magic directory*. The existence and contents of this directory are safe from prying eyes, because Aspen will respond to any requests mapping to the magic directory with a [404 Not Found](#).

Installation

Aspen can be installed using either `distutils` or `setuptools`. That is, you can either download a tarball, unpack it, and run:

```
$ python setup.py install
```

Or you can run:

```
$ easy_install aspen
```


Tutorial

Once you have installed Aspen, here are some quick walk-throughs to get your feet wet. They are written sequentially.

3.1 "Greetings, program!"

In your home directory, make a new directory named 'aspentut'. Create a file in 'aspentut' named 'index.html', with the following contents:

```
Greetings, program!
```

At the command line in the 'aspentut' directory, type `aspen`. You should get output like this:

```
$ aspen
aspen starting on ('', 8080)
```

Now open a web browser and hit `http://localhost:8080/`. You should see "Greetings, program!" in your browser. Congratulations!

3.2 Your First Handler

Aspen uses *handlers* to process files such as your 'index.html'. Now we are going to write our own handler.

First, create a directory under 'aspentut' named '__' (that's two underscores). This is 'aspentut's *magic directory*, and it is where you configure and extend your website. Now create two directories under the magic directory: 'etc' and 'lib'. Under 'lib', create a 'python2.x' directory, where 'x' corresponds to the minor version of Python you are using. Your directory structure should now look like this:

```
aspentut
aspentut/__
aspentut/__/etc
aspentut/__/lib/python2.x
```

In '__/lib/python2.x', create a file named 'handy.py' with the following contents:

```
def handle(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return [environ['PATH_TRANSLATED']]
```

And in `__etc`, create a file named `handlers.conf` with these contents:

```
fnmatch aspen.rules:fnmatch

[handy:handle]
fnmatch *.asp
```

What we have done is we have defined a new handler, and wired it up to be used for any request for a file with the extension `.asp`. So now let's create such a file at `aspentut/handled.asp` and give it the following contents:

```
Greetings, program?
```

Restart Aspen, then hit `http://localhost:8080/handled.asp`. You should see the filesystem pathname of the file being served.

If you are familiar with the WSGI specification, you will recognize that `handy.handle` is a WSGI callable. Aspen plugins all speak WSGI. Also notice that the rules for when a certain handler is invoked are themselves extensible. The `fnmatch` rule comes with Aspen, but you can also write your own.

3.3 What You've Learned

In this brief tutorial we've introduced these key facts about Aspen:

- Aspen websites use the filesystem for site hierarchy.
- Aspen websites are configured and extended via a "magic directory."
- Aspen configuration happens through plain-text configuration files.
- Aspen extensions are WSGI callables.

Besides handlers, Aspen can be extended by wiring up arbitrary WSGI apps to certain paths, and maintaining a global WSGI middleware stack. If this all fits your style of development, then check out the reference documentation that follows for the full story.

User Interface (UI)

Users interface with Aspen through three mechanisms: the command line, configuration files, and the environment. Where a program parameter is set in more than one of these contexts, they take precedence in the order given here. For example, a *mode* option on the command line will override any *mode* setting in a config file or in the environment.

4.1 Command Line

Usage:

```
aspen [options] [command]
```

Aspen takes one optional positional argument, *command*, which must be one of: *start*, *status*, *stop*, *restart*, or *runfg*. The default is *runfg*, which causes Aspen to run in the foreground, sending all messages to stdout.

start, *status*, *stop*, and *restart* control Aspen as a daemon, via a pidfile. If the website root has a directory named ‘*__*’ (that’s two underscores; the *magic directory*), then the pidfile is at ‘*__*/var/aspen.pid’. Otherwise, the pidfile is created in ‘/tmp’. When run as a daemon, stdout and stderr are redirected to ‘*__*/var/aspen.log’ if ‘*__*’ exists, and to ‘/dev/null’ otherwise. The ‘*__*/var’ directory will be created if it does not exist. The permission mode of the pidfile is set to 0600; likewise with the logfile, unless it is ‘/dev/null’.

The Aspen distribution includes a script in ‘etc/aspen_bash_completion’ that can be used to configure the bash shell to autocomplete from among Aspen’s arguments. See the source for more information.

Aspen’s command-line options are as follows:

Option	Description
-a/--address=address	The address to which Aspen should bind. If <i>address</i> begins with a dot or a forward slash, then it is interpreted as a relative path.
-m/--mode=mode	One of <i>debugging</i> , <i>development</i> , <i>staging</i> , or <i>production</i> . In <i>debugging</i> and <i>development</i> modes, Aspen will log to stdout and stderr.
-r/--root=root	The directory containing the website for Aspen to serve.

4.2 Configuration Files

Aspen obeys several configuration files, all located in ‘*__*/etc’. The comment character for these files is #, and comments can be included in-line. Blank lines are ignored, as is initial and trailing whitespace per-line. Where section names are called for, they are given in brackets.

Where a configuration file calls for a Python object to be specified, this is done in a notation derived from *setuptools*’ *entry_points* feature: a dotted module name, followed by a colon and a dotted identifier naming an object within

the module. This is referred to below as *colon notation*. The following example would import the `bar` object from `example.package.foo`, and use its `baz` attribute:

```
example.package.foo:bar.baz
```

4.2.1 apps.conf

In Aspen, an *application* or *app* refers to a WSGI application that is connected to a particular directory. Apps are set up in `'__etc/apps.conf'`.

The `'__etc/apps.conf'` file contains a newline-separated list of white-space-separated path name/object name pairs. The path names refer to URL-space, and are translated literally to the filesystem. If the trailing slash is given, then requests for that directory will first be redirected to the trailing slash before being handed off to the application. If no trailing slash is given, the application will also get requests without the slash. When choosing an application to service a request, the most specific pathname matches first.

Object names are in colon notation, and they name WSGI callables. Aspen updates the `SCRIPT_NAME` and `PATH_INFO` settings in `environ` before handing off to the relevant callable. `SCRIPT_NAME` will never end with a slash, and if `PATH_INFO` is not empty, it will always begin with a slash.

Aspen will (over)write a file called `'README.aspen'` in each directory mentioned in `'apps.conf'`, containing the relevant line from `'apps.conf'`. If the directory does not exist, it is created. Aspen will also remove any obsolete `'README.aspen'` files within your site tree.

Example apps.conf

```
/foo      example.apps:foo      # will get both /foo and /foo/  
/bar/     example.apps:bar      # /bar will redirect to /bar/  
/bar/baz  example.apps:baz      # will 'steal' some of /bar's requests
```

4.2.2 aspen.conf

Aspen's general configuration file is at `'__etc/aspen.conf'`. It is in `.ini`-style format per the `ConfigParser` module. Aspen responds to the following settings in the `main` section. You may define additional settings and sections that are meaningful to your application, which you may access using the `aspen.conf` object described below in the "API" chapter.

Option	Description
<i>address</i>	The address to which Aspen should bind. If <i>address</i> begins with a dot or a forward slash, then it is interpreted as an <code>ADDRESS</code> .
<i>defaults</i>	A comma-separated list of names to look for when a directory is requested. Any default resource is located immediately below the directory.
<i>http_version</i>	The version of HTTP to speak, either <code>1.0</code> or <code>1.1</code> .
<i>mode</i>	One of <code>debugging</code> , <code>development</code> , <code>staging</code> , or <code>production</code> . In <code>debugging</code> and <code>development</code> modes, Aspen will log requests.
<i>threads</i>	The number of threads to maintain in the request-servicing thread pool.

Example

Here is an example `'aspen.conf'` configuration file:

```
[main]
address = :8000

[myapp]
knob = true
```

4.2.3 handlers.conf

Aspen *handlers* are WSGI applications that are associated with files and directories on the filesystem according to arbitrary rules. This provides a flexible infrastructure for many different development patterns.

The ‘`__/etc/handlers.conf`’ file begins with an anonymous ”rules” section, which is a newline-separated list of white-space-separated rule name/object name pairs. Rule names can be any string without whitespace. Each object name (in colon notation) specifies a *rule*, a callable taking a filesystem path name and an arbitrary predicate string, and returning `True` or `False`. The path argument is absolute and is guaranteed to exist; it is `PATH_TRANSLATED` from the WSGI environment, with any default resource already located.

Following the rule specification are sections specifying *handlers*, which as mentioned above are WSGI callables.

The name of each section specifies a handler (a WSGI callable) in colon notation. The body of each section is a newline-separated list of conditions under which this handler is to be called. Fundamentally, these conditions are made up of a rule name as defined at the beginning of the file, and an arbitrary predicate string (which can include whitespace) that is meaningful to the matching rule callable. If no predicate is given, then the rule callable will receive `None` for its predicate argument. Rules must be explicitly specified at the beginning of the file before being available within handler sections. After the first condition in a handler section, additional condition lines must begin with one of `AND`, `OR`, or `NOT`. These case-insensitive tokens specify how conditions are to be combined in evaluating whether to use this handler.

On each request, handlers are considered in the order given, and the first matching handler is used. Only one handler is used for any given request.

Note that if the file ‘`__/etc/handlers.conf`’ exists at all, the defaults (see the example below) disappear, and you must respecify any of the default rules in your own file if you want them.

Example handlers.conf

This is Aspen’s default handler configuration:

```
catch_all    aspen.rules:catch_all

[aspen.handlers.static:wsgi]
catch_all
```

Here is a more full-featured example:

```

catch_all    aspen.rules:catch_all
isfile       aspen.rules:isfile
fnmatch      aspen.rules:fnmatch

# Set up scripts.
# =====

[aspen.handlers.simplates:stdlib]
    isfile
AND fnmatch *.html

# Everything else is served statically.
# =====

[aspen.handlers.static:wsgi]
    catch_all

```

4.2.4 logging.conf

Aspen uses the Python logging library. You can configure logging with a `‘__etc/logging.conf’` file, which is processed by the logging library’s `fileConfig` routine. If there is no `‘logging.conf’` file, then Aspen logs all messages to `‘sys.stderr’`. Note that you are responsible to create any directories needed to store log files that you specify in `‘logging.conf’`; Aspen does not create those directories for you, and will not work if they do not exist.

Example logging.conf

Below is an example logging configuration file that logs all messages to a file at `‘/var/log/aspen.log’`, rotating every night at midnight, keeping old logs for six days. See more examples in [the Python logging documentation](#).

```

[loggers]
keys=root,aspen

[handlers]
keys=handler

[formatters]
keys=formatter

[logger_root]
level=NOTSET
handlers=handler

[logger_aspen]
level=NOTSET
handlers=handler
propagate=0
qualname=aspen

[handler_handler]
class=handlers.TimedRotatingFileHandler
level=NOTSET
formatter=formatter
args=('/var/log/aspen.log', 'midnight', -1, 6)
      #filename, when, interval [ignored], backupCount

[formatter_formatter]
format=%(asctime)s %(name)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter

```

See Also:

[logging](#)

The documentation for the standard Python logging library.

4.2.5 middleware.conf

Aspen allows for a full WSGI middleware stack, configured via the ‘`__etc/middleware.conf`’ file. This is simply a newline-separated list of middleware factories in colon notation. Each factory (which may be a class constructor or other callable) is called with exactly one positional argument, the next middleware on the stack. The first-mentioned middleware will therefore be the outer-most in the stack (i.e., closest to the browser).

Example middleware.conf

```

example.foo:bar # closest to browser
example.baz:buz # closest to your apps/handlers

```

4.3 The Environment

Aspen incorporates a `mode` module, which uses the `PYTHONMODE` environment variable to model the application life-cycle through four deployment modes: `debugging`, `development`, `staging`, and `production`. This module is available to your applications at `aspen.mode`, and its API is documented in the "API" chapter, below.

Aspen itself adapts to the current `PYTHONMODE`. In `debugging` and `development` modes, Aspen will restart itself any time a configuration file or module source file changes on the filesystem.

Application Programming Interface (API)

The `aspen` library exposes a number of objects. Here we document those that are probably most useful.

5.1 Applications

Aspen applications are WSGI callables that are intended to be associated with a single directory via the ‘`apps.conf`’ file. Aspen comes bundled with the following applications, in the `aspen.apps` subpackage.

5.1.1 `django_`

Aspen includes glue code for the Django web framework. In order to use this feature, you must install Django for your site:

- Put `django` on your `PYTHONPATH` (e.g., in ‘`__lib/python`’).
- Put your Django project on your `PYTHONPATH` (e.g., in ‘`__lib/python`’).
- Add a `[django]` section to ‘`aspen.conf`’, with a `settings_module` setting that points to your `settings` module.
- Configure your Django project’s ‘`settings`’ module.

The `aspen.apps.django_` module defines one function:

wsgi (*environ*, *start_response*)

This is Django’s `WSGIHandler` callable.

5.1.2 `static`

The `aspen.apps.static` module defines one function:

wsgi (*environ*, *start_response*)

This application translates the request path to the filesystem, locates any default resource for directories (per the value of `defaults` in ‘`aspen.conf`’), and hands off to `aspen.handlers.static.wsgi`.

5.2 Middleware

The Aspen package bundles the following WSGI middleware in the `aspen.middleware` subpackage.

5.2.1 raised

The `aspen.middleware.raised` module provides for ending WSGI requests by raising a `Response` object that is caught by a middleware:

class `Response` (`[code]` [`, headers`] [`, body`])

Constructs a new `Response` object. If given, `code` must be an integer; the default is `200` (see [the HTTP spec](#) for other values that will be meaningful to most HTTP clients). `headers` may be a dictionary or a list of 2-tuples. `body` may be a string or other iterable.

middleware (`next`)

WSGI middleware; `next` is the next WSGI callable on the stack. This middleware catches any `Response` objects raised by `next` and calls them (they are themselves WSGI callables) to produce the response.

Response Objects

Instances of `aspen.middleware.raised.Response` are WSGI applications, with the following data attributes. Note that values are only validated in the constructor, so it is possible to raise a malformed `Response` by setting instance attributes post-instantiation.

code

The [HTTP code](#) as an integer.

body

The message body as a string or other iterable.

headers

The message headers as an instance of the standard library's `email.Message.Message`.

When called, `Response` instances call `start_response` with adaptations of `code` and `headers`, and return `body`, or a one-item list containing `body` if `body` is a string.

Example

Here is an example:

```
Python 2.5 (r25:52005, Sep 25 2006, 21:37:36)
[GCC 3.4.4 [FreeBSD] 20050518] on freebsd6
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from aspen import raised
>>> def app(env, start):
>>>     raise raised.Response(200, "Greetings, program!")
>>> app = raised.middleware(app)
>>>
>>> from wsgiref.simple_server import make_server
>>> server = make_server('', 8080, app)
>>> server.serve_forever() # now hit http://localhost:8080/
>>>
192.168.1.100 - - [09/Nov/2006 23:52:45] "GET / HTTP/1.1" 200 19
```

5.3 Configuration

Aspen parses and harmonizes all command-line, configuration file, and environment settings before it loads your plugins. This information is then available to your modules via several objects which are dynamically placed in the global `aspen` namespace before your plugins are loaded—`conf`, `configuration`, and `paths`—and via the `mode` module.

5.3.1 `conf`

The `aspen.conf` object is an instance of `aspen._configuration.ConfFile`, which subclasses the standard library's `ConfigParser.RawConfigParser` class to represent the `'__etc/aspen.conf'` file. In addition to the `RawConfigParser` API, the object supports both attribute and key read-only access; either returns a dictionary corresponding to a section of the `'aspen.conf'` file. If the named section does not exist, an empty dictionary is returned.

Your application is free and encouraged to use the `'aspen.conf'` file for its own configuration, and to access that information via this object.

To illustrate, here is a minimal `'aspen.conf'` file:

```
[my_settings]
foo = bar
```

Such a file could support code like this:

```
import aspen

def wsgi_app(environ, start_response):
    my_setting = aspen.conf.my_settings.get('foo', 'default')
    start_response('200 OK', [])
    return ["My setting is %s" % my_setting]
```

See Also:

[RawConfigParser](#)

In addition to the API above, `aspen.conf` also exposes the `RawConfigParser` API.

5.3.2 `configuration`

The `aspen.configuration` object provides raw access to the parser objects used to configure your server, and a number of basic settings.

Parsers

The various parsers and raw settings are exposed as these members:

args

An argument list as returned by `optparse.OptionParser.parse_args`.

conf

An instance of `aspen._configuration.ConfFile`; see above.

optparser

An `optparse.OptionParser` instance.

opts

An `optparse.Values` instance per `optparse.OptionParser.parse_args`.

paths

An instance of `aspen._configuration.Paths`; see below.

Settings

Furthermore, `aspen.configuration` exposes specific configuration settings as these members:

address

A *(hostname, port)* tuple (for `AF_INET` and `AF_INET6` address) or string (for `AF_UNIX`) giving the address to which Aspen is bound.

command

A string giving the command line argument (*start, stop, etc.*).

daemon

A boolean indicating whether Aspen is acting as a daemon.

defaults

A tuple listing the default resource names to look for in a directory.

http_version

A string indicating the HTTP version to speak, either `1.0` or `1.1`.

sockfam

One of `socket.AF_INET`, `socket.AF_INET6`, and `socket.AF_UNIX`.

threads

A non-zero positive integer; the number of threads in the server's request-handling thread pool.

All members are intended to be read-only.

See Also:[ConfigParser](#)

The naming is not PEP 8, but the documentation is fine.

[optparse](#)

On the other hand, the documentation for `optparse` is rather, um, convoluted. Good luck!

5.3.3 mode

It is often valuable to maintain a distinction between various phases of an application's lifecycle. The `mode` module calls these phases *modes*, and identifies four of them, given here in conceptual life-cycle order:

Mode	Description
debugging	The application is being actively debugged; exceptions may trigger an interactive debugger.
development	The application is being actively developed; however, exceptions should not trigger interactive debugging.
staging	The application is deployed in a mock-production environment.
production	The application is in live use by its end users.

The expectation is that various aspects of the application—logging, exception handling, data sourcing—will adapt to the current mode. The mode is set in the `PYTHONMODE` environment variable. This module provides API for

interacting with this variable. If PYTHONMODE is unset, it will be set to development when this module is imported.

Members

The module defines the following functions:

get ()

Return the current PYTHONMODE setting as a lowercase string; will raise `EnvironmentError` if the (case-insensitive) setting is not one of `debugging`, `development`, `staging`, or `production`.

set (mode)

Given a mode, set the PYTHONMODE environment variable and refresh the module's boolean members. If given a bad mode, `ValueError` is raised.

setAPI ()

Refresh the module's boolean members. Call this if you ever change PYTHONPATH directly in the `os.environ` mapping.

The module also defines a number of boolean attributes reflecting the current mode setting, including abbreviations and combinations. Uppercase versions of each of the following are also defined (e.g., `DEBUGGING`).

debugging, deb

True if PYTHONMODE is set to debugging.

development, dev

True if PYTHONMODE is set to development.

staging, st

True if PYTHONMODE is set to staging.

production, prod

True if PYTHONMODE is set to production.

debugging_or_development, debdev, devdeb

True if PYTHONMODE is set to debugging or development.

staging_or_production, stprod

True if PYTHONMODE is set to staging or production.

Example

Example usage:

```
>>> import mode
>>> mode.set('development') # can set the mode at runtime
>>> mode.get()             # and access the current mode
'development'
>>> mode.development      # module defines boolean constants
True
>>> mode.PRODUCTION       # uppercase versions are also defined
False
>>> mode.dev              # as are abbreviations
True
>>> mode.DEBDEV, mode.stprod # and combinations
(True, False)
```

5.3.4 paths

The `aspen.paths` object is an instance of `aspen._configuration.Paths`; it is simply a container for various paths, all normalized and absolute:

root

the website's filesystem root

--

the magic directory

lib

the site's local Python library. First we look for `'__lib/python'`, then for `'__lib/pythonx.y'`, using only the first found

pkg

'site-packages' under the site's local Python library, `'__lib/pythonx.y/site-packages'`

plat

the local platform-specific Python library, `'__lib/plat-x'`

If there is no magic directory, then `__`, `lib`, `pkg` and `plat` are all `None`. If there is, then `lib`, `pkg` and `plat` are added to `sys.path`.

5.4 Handlers

Aspen *handlers* are WSGI callables that are intended to be associated with multiple files or directories via the 'handlers.conf' file. Aspen comes bundled with the following handlers, in the `aspen.handlers` subpackage.

5.4.1 autoindex

The `aspen.handlers.autoindex` module defines one function:

wsgi (*environ*, *start_response*)

This handler displays an HTML listing of the files in the directory at `environ['PATH_TRANSLATED']`. If it is associated with a non-directory, it will raise `AssertionError`. The listing will not include the magic directory, nor files named `'README.aspen'`, nor hidden files (those whose name begins with `'.'`).

The static handler can be configured to automatically call the autoindex handler for all directories. See below for details.

5.4.2 http

The `aspen.handlers.http` module provides three handlers:

HTTP400 (*environ*, *start_response*)

Responds to every request with 400 Bad Request.

HTTP403 (*environ*, *start_response*)

Responds to every request with 403 Forbidden.

HTTP404 (*environ*, *start_response*)

Responds to every request with 404 Not Found.

HTTP500 (*environ*, *start_response*)

Responds to every request with 500 Internal Server Error.

5.4.3 `simplates`

Aspen comes bundled with a handler called `simplates`. In basic terms, a *simplat*e is a single-file web template with an initial pure-Python section that populates the context for the template. `Simplates` are a way to keep logic and presentation as close together as possible without actually mixing them.

In more detail, a *simplat*e is a template with two optional Python components at the head of the file, delimited by ASCII form feeds (this character is also called a page break, FF, `ctrl-L`, `0xc, 12`). If there are two initial Python sections, then the first is `exec`'d when the *simplat*e is first loaded, and the namespace it populates is saved for all subsequent invocations of this *simplat*e. This is the place to do imports and set constants; it is referred to as the *simplat*e's *import section* (be sure the objects defined here are thread-safe). The second Python section, or the first if there is only one, is `exec`'d within the *simplat*e namespace each time the *simplat*e is invoked; it is called the *run-time Python section*. The third section is parsed according to one of the various web templating languages. The namespace for the template section is a copy of the *import section*'s namespace, further modified by the *run-time Python section*. If a *simplat*e has no Python sections at all, then the template section is rendered with an empty context. `SyntaxError` is raised when parsing a *simplat*e that has more than two form feeds.

In debugging and development modes, `simplates` are loaded for each invocation of the resource. In staging and production modes, `simplates` are loaded and cached until the filesystem modification time of the underlying file changes. If parsing the file into a *simplat*e raises an `Exception`, then that is cached as well, and will be raised on further calls until the entry expires as usual.

`Simplates` obey an *encoding* key in a `[simplates]` section of `'aspen.conf'`: this is the character encoding used when reading `simplates` off the filesystem, and it defaults to `'UTF-8'`.

For all `simplates`, the full filesystem path of the *simplat*e is placed in its namespace as `__file__` before the *import section* is executed.

NB: `Simplates` are never used in the abstract. Rather, one always uses a particular flavor of *simplat*e that obeys the above general rules but which provides slightly different semantics corresponding to the web framework upon which each flavor is based.

The Aspen distribution currently bundles two flavors of *simplat*e: Django-flavored and `stdlib`-flavored. The WSGI callables for each are defined in the `aspen.handlers.simplates` module:

`django` (*environ*, *start_response*)

Serve `environ['PATH_TRANSLATED']` as a Django-flavored *simplat*e.

`stdlib` (*environ*, *start_response*)

Serve `environ['PATH_TRANSLATED']` as a `stdlib`-flavored *simplat*e.

Django-flavored

In addition to the `aspen.apps.django_app`, which serves Django in usual monolithic fashion, we also provide a handler that integrates the Django web framework with the *simplat*e pattern. As mentioned, this callable is available as *django* in the `aspen.handlers.simplates` module.

Installation To use Django `simplates`, first install the Django framework in your site:

- Put `django` on your `PYTHONPATH` (e.g., in `'__lib/python'`).
- Put your Django project on your `PYTHONPATH` (e.g., in `'__lib/python'`).
- Add a `[django]` section to `'aspen.conf'`, with a `settings_module` key that points to your `settings` module.
- Configure any database settings, etc., in your Django project's `'settings'` module.

Then tell Aspen to use the django simplate handler for various files via the `__etc/handlers.conf` file. For example, the following `handlers.conf` would serve files ending in `.html` as Django simplates, and would serve all other resources statically:

```
fnmatch      aspen.rules:fnmatch
catch_all    aspen.rules:catch_all

[aspen.handlers.simplates:django]
fnmatch *.html

[aspen.handlers.static:wsgi]
catch_all
```

Lastly, close the loop by telling Django about simplates via the `urls.py` file in your Django project package, like so:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', include('aspen.handlers.simplates.django_'))
)
```

Admittedly, that is a fair amount of wiring. The main benefits to using Django via Aspen simplates are first, that your view code and template code are together in the same file (without being mixed); and second, that you get filesystem- rather than regex-based URL layouts.

Distinctives Django-flavored simplates have these distinctives:

- The simplate namespace is a [Django RequestContext](#).
- The template section is compiled as a [Django template](#).
- If the run-time Python section raises `SystemExit` with a `Django HttpResponse` object as its argument, and no other *response* object is defined in the section, then the `SystemExit` response is sent back and the templating section is skipped entirely.
- If the run-time Python section raises `SystemExit` without a `Django HttpResponse` object as its argument, and no other *response* object is defined in the section, then processing of the run-time section ends and processing proceeds to rendering the template.
- If the run-time Python section defines a *response* object, then (whether or not the run-time section is terminated with `SystemExit`) this object is assumed to be a `Django HttpResponse` object, and the template is rendered to it.

Standard Library-flavored

Aspen includes a simplate flavor that has no dependencies outside the standard library, effectively giving you a raw WSGI interface. The handler for this is named `stdlib` and is defined in the `aspen.handlers.simplates` module. Here are its distinctives:

- The run-time Python section has two additional names in its namespace, `environ` and `start_response`, corresponding to the parameters of the handler as specified above.
- If the run-time Python section raises `SystemExit`, this is silently ignored.

- If the run-time Python section defines a *response* object, this is assumed to be an iterable per the WSGI specification and is returned as such. The template section is skipped.
- The template section is rendered using mapping-based string interpolation.

5.4.4 static

The `aspen.handlers.static` module defines one function:

wsgi (*environ*, *start_response*)

This handler serves `environ['PATH_TRANSLATED']` as a static resource. The `Content-Type` is set using the standard library's `mimetypes.guess_type` function, defaulting to `text/plain`. In staging and production mode, we obey any `If-Modified-Since` header.

This handler adapts to the *autoindex* setting in the `[static]` section of `'aspen.conf'`. If set to `yes` (the default), then the `aspen.handlers.autoindex.wsgi` handler will be used to serve requests for directories. If set to `no`, the `aspen.handlers.http.HTTP403` handler is used instead. The *autoindex* value is case-insensitive, but if other than `yes` or `no` is given, `ConfigError` is raised at start-up.

5.5 Rules

An Aspen *rule* is a callable that takes a filesystem path and a predicate string, and returns `True` or `False`. Rules are part of Aspen's handler infrastructure; see the documentation for `'handlers.conf'` for how to use them.

catch_all (*path*, *predicate*)

Always return `True`.

fnmatch (*path*, *predicate*)

Match if *path* matches the pattern *predicate* per the standard library's `fnmatch.fnmatchcase`.

hashbang (*path*, *predicate*)

Match if the file at *path* starts with `'#!'`.

isdir (*path*, *predicate*)

Match if *path* points to a directory.

isfile (*path*, *predicate*)

Match if *path* points to a file.

isexecutable (*path*, *predicate*)

Match if the file at *path* is executable.

mimetype (*path*, *predicate*)

Match if the mimetype of the file at *path* is *predicate*.

rematch (*path*, *predicate*)

Match if *path* matches the regular expression *predicate*.

Additional Programs

Aspen comes with two helper programs.

6.1 `aspen.mod_wsgi`

You can serve an Aspen website using the `mod_wsgi` extension module for Apache via the **`aspen.mod_wsgi`** script. Since the same Aspen website can be served by the main **`aspen`** program as by `mod_wsgi`, this gives you a compelling development/deployment scenario.

To use `mod_wsgi` with Aspen, add the following two directives to your `'httpd.conf'`:

```
WSGIScriptAlias / "\path\to\aspen.mod_wsgi"  
SetEnv aspen.root "\path\to\website-root"
```

See Also:

[mod_wsgi](#)

The `mod_wsgi` Apache extension module

6.2 `aspen.monitord`

Aspen includes a daemon called **`aspen.monitord`** that launches the main **`aspen`** daemon, and restarts it if it ever goes down. It takes one command-line argument, the root filesystem path of the website to be monitored. It uses the `'__/var/aspen.pid'` file to keep track of the main **`aspen`** process, and it stores its own pid in `'__/var/aspen.monitord.pid'`. **`aspen.monitord`** logs via the *USER* syslog facility with an ident of *aspen.monitord*.

Credits and Legalese

All original work is copyright (c) 2006-2007 Chad Whitacre and contributors, all rights reserved, and is released under the [MIT license](#):

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

7.1 aspen_bash_completion

The 'aspen_bash_completion' script is based on a similar script from Django. The original is copyright (c) 2005 by the Lawrence Journal-World, all rights reserved, and is used in modified form under the BSD license:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of Django nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

7.2 daemon.py

The 'daemon.py' module is copyright 2006 by LivingLogic AG, Bayreuth/Germany and Walter Drwald. It is used without change under the following license:

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of LivingLogic AG or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

LIVINGLOGIC AG AND THE AUTHOR DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL LIVINGLOGIC AG OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

7.3 wsgiserver.py

The 'wsgiserver.py' module is copyright (c) 2004-2006 by the CherryPy Team (team@cherrypy.org), all rights reserved, and is used without change under this BSD license:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the CherryPy Team nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.