Aspen Documentation

Release 1.0rc3-dev

Chad Whitacre et al.

Sep 27, 2017

Contents

Aspen is a filesystem dispatch library for Python web frameworks. Instead of using regular expressions, decorators, or object traversal, Aspen dispatches HTTP requests based on the natural symmetry of URL paths and filesystem paths. In the immortal words of Jeff Lindsay, "so like. a web server." Yes! ;-)

This is the documentation for the development version of the core Aspen library, describing its filesystem dispatch behavior regardless of the web framework you're using it with. For instructions on configuring Aspen with a specific web framework, see the docs for django-aspen, Flask-Aspen, or Pando. See the project homepage for an overview.

This version of Aspen has been tested with Python 2.7, 3.4, and 3.5, on both Ubuntu and Windows.

Aspen's source code is on GitHub, and is MIT-licensed.

CHAPTER 1

Installation

Aspen is available on PyPI:

```
$ pip install aspen
```

CHAPTER 2

Contents

Filesystem Dispatch Rules

Aspen dispatches web requests to the filesystem based on paths. For simple cases this is straightforward: `/foo.html` in the browser will find `foo.html` in the publishing root on your filesystem, and serve the file statically. There are a couple wrinkles, however. What about *dynamic* resources? And what about variable path parts?

Note: This is a tutorial. Please refer to our test table for the complete dispatch rules.

Dynamic Resources

Sometimes you want the URL path `/foo.html` to find a static HTML file. More frequently, you want it to serve a dynamic resource. Aspen uses the simplates file format to model dynamic resources, and Aspen knows a file is a simplate because of a `.spt` extension: `/foo.html` will find `foo.html.spt` if it exists. If you ask for `/foo.html.spt` directly, however, you'll get a 404.

But what happens if you have both of the following on your filesystem?

- `foo.html`
- `foo.html.spt`

When you ask for `/foo.html`, which one will you get? Which file will Aspen use to represent the resource? The answer is `foo.html`. The principle is "most specific wins". The dynamic resource could actually serve other content types (despite the `.html` in the filename), whereas the static resource will *only* result in an HTML representation.

Now how about this one: what happens if you ask for `/foo.html` with *these* two on your filesystem?

- `foo.html.spt`
- `foo.spt`

You guessed it: `foo.html.spt`. Even though both are dynamic resources, and both could technically result in any content type representation, the former is likely to result in just HTML. Aspen therefore considers it to be more specific, and to match it before the more general `foo.spt`.

Now let's say you only have:

- `foo.spt`

That simplate will answer for `/foo.html`. But! It will *also* answer for `/foo.json`, `/foo.csv`, `/foo.xml`, etc. One simplate can serve multiple content type representations of the same resource. The simplate docs explain how, but before we get there, let's talk about path variables.

Path Variables

It's common in web applications to use parts of the URL path to pass variables to a dynamic resource. For example, the `2016` in `/blog/2016/some-post.html` will want to end up as a `year` variable in your code, and `some-post` perhaps as `slug`. Since Aspen uses the filesystem for dispatch, you define these variables on the filesystem. You use the `%` (percent) character for this.

For the blog URL example, we might have the following simplate on our filesystem:

- `blog/%year/%slug.html.spt`

Aspen matches from `%` to the end of the path part or a file extension, whichever comes first. Now, inside your simplate, you will have access to `year` and `slug` variables containing the values from the URL path.

Typecasting

URL path parts are strings, but sometimes you want to convert to a different data type. Aspen provides for this by looking for special file extensions following the `%` variable: `.int` and `.float` are supported by default.

If our simplate for the blog example was at:

- `blog/%year.int/%slug.html.spt`

Then the `year` variable inside our simplate would be an integer instead of a string.

Ready for Simplates?

Aspen serves static files directly, and dynamic files using simplates (`.spt`), with path variables based on special `%` names on the filesystem. With those basics in place, it's clearly time to write a simplate!

How to Write a Simplate

Aspen dispatches web requests to files on the filesystem based on paths, but what kind of files does it expect to find there? The answer is simplates. Simplates are a file format combining request processing logic—like you'd find in a Django view or a Rails controller—with template code, in one file with multiple sections.

Note: Check the Aspen homepage for links to simplate support for your favorite text editor.

Sections of a Simplate

What are the sections of a simplate? Let's illustrate by example:

```
import random

[-----------------------------------]
program = querystring['program']
excitement = '!' * random.randint(1, 10)

[----] text/html via stdlib_template
<h1>Greetings, $program$excitement</h1>

[-----] text/plain via stdlib_format
Greetings, {program}{excitement}

[---] application/json via json_dump
{"program": program, "excitement": excitement}
```

The first thing to notice is that the file is separated into multiple sections using lines that begin with the characters `[---]`. There must be at least three dashes, but more are fine.

Sections in a simplate are either "logic sections" or "content sections". Content sections may have a "specline" after the `[---]` separator. The format of the specline is `content-type via renderer`. The syntax of the content sections depends on the renderer. The logic sections are Python.

Note: Simplates under Python 2.7 use the following `__future__` features: `absolute_import`, `division`, `print_function`, and `unicode_literals`.

A simplate may have one or more sections. Here are the rules for determining which section is what:

1. **If a simplate only has one section**, it's a content section.
2. **If a simplate has two sections**, the first is *request* logic (runs for every request), and the second is a content section.
3. **If a simplate has more than two sections**:

1. If the second section has a specline, then the first is request logic, and the rest are content sections.
2. If the second section has no specline, then the first is *initialization* logic (runs once when the page is first hit), the second is request logic, and the rest are content sections.

Putting that all together, we see that the above example has five sections:

1. a logic section containing Python that will run once when the page is first hit,
2. a request section containing Python that will run every time the page is hit,
3. a section for rendering `text/html` via Python templates,
4. a section for rendering `text/plain` via new-style Python string formatting, and
5. a section for rendering `application/json` via Python's `json` library.

Context

The power of simplates is that objects you define in the logic sections are automatically available to the templates in your content sections. The above example illustrates this with the `program` and `excitement` variables. Moreover,

Aspen makes various objects available to the logic sections of your simplates (besides the Python builtins).

Here's what you get:

- `path`—a representation of the URL path
- `querystring`—a representation of the URL querystring
- `request_processor`—a *`RequestProcessor`* instance
- `resource`—a representation of the HTTP resource
- `state`—the dictionary that contains the request state

Framework wrappers will add their own objects, as well.

Standard Renderers

Aspen includes five renderers out of the box:

- `json_dump`—takes Python syntax, runs it through `eval` and then `json.dumps`
- `jsonp_dump`—takes Python syntax, runs it through `eval` and `json.dumps`, and then wraps it in a JSONP callback if one is specified in the querystring (as either `callback` or `jsonp`)
- `stdlib_format`—takes a Python string, runs it through format-style string replacement
- `stdlib_percent`—takes a Python string, runs it through percent-style string replacement
- `stdlib_template`—takes a Python string, runs it through template-style string replacement

---

Note: Check the Aspen homepage for links to plugins for other renderers.

Specline Defaults

Speclines are optional. The defaults ... I guess we should point to the API reference for this. And the framework wrappers will have something to say about this, as well.

Content Negotiation

Aspen negotiates with clients to determine how to best represent a resource for a given request. Aspen models resources using simplates, and the content sections of the simplate determine the available representations. Here are the rules for negotiation:

1. **If the URL path includes a file extension**, Aspen looks in the Python mimetypes registry for a content type associated with the extension. If the extension is not in the registry, Aspen responds with `404 Not Found`. If the extension *is* in the registry, Aspen looks for a match against the corresponding type. If no content section provides the requested representation, Aspen again responds with `404 Not Found`.
2. **If the URL path does not include a file extension and there are multiple available types**, Aspen turns to the `Accept` header. If the `Accept` header is missing or malformed, Aspen responds using the first available content section. If the `Accept` header is valid, Aspen looks for a match. If no content section provides an acceptable representation, Aspen responds with `406 Not Acceptable`.
3. **If the URL path includes a file extension but there is only one available type**, then Aspen ignores the `Accept` header (as the spec allows), responding with the only available representation.

Note: Aspen delegates to the python-mimeparse library to determine the best available match for a given media range.

How to Write a Plugin

This document is for people who want to write a plugin for Aspen. If you only want to use Aspen with existing plugins, then ... what?

Negotiated and rendered resources have content pages the bytes for which are transformed based on context. The user may explicitly choose a renderer per content page, with the default renderer per page computed from its media type. Template resources derive their media type from the file extension. Negotiated resources have no file extension by definition, so they specify the media type of their content pages in the resource itself, on the so-called "specline" of each content page, like so:

```
[---]
[---] text/plain
Greetings, program!
[---] text/html
<h1>Greetings, program!</h1>
```

A Renderer is instantiated by a Factory, which is a class that is itself instantied with one argument:

```
configuration   an Aspen configuration object
```

Instances of each Renderer subclass are callables that take five arguments and return a function (confused yet?). The five arguments are:

```
factory         the Factory creating this object
filepath        the filesystem path of the resource in question
raw             the bytestring of the page of the resource in question
media_type      the media type of the page
offset          the line number at which the page starts
```

Each Renderer instance is a callable that takes a context dictionary and returns a bytestring of rendered content. The heavy lifting is done in the render_content method.

Here's how to implement and register your own renderer:

```
from aspen.simplates.renderers import Renderer, Factory

class Cheese(Renderer):
    def render_content(self, context):
        return self.raw.replace("cheese", "CHEESE!!!!!!")

class CheeseFactory(Factory):
    Renderer = Cheese

request_processor.renderer_factories['excited-about-cheese'] = CheeseFactory(request_
↪processor)
```

Put that in your startup script. Now you can use it in a negotiated or rendered resource:

```
[---] via excited-about-cheese
I like cheese!
```

Out will come:

```
I like CHEESE!!!!!!!
```

If you write a new renderer for inclusion in the base Aspen distribution, please work with Aspen's existing reloading machinery, etc. as much as possible. Use the existing template shims as guidelines, and if Aspen's machinery is inadequate for some reason let's evolve the machinery so all renderers behave consistently for users. Thanks.

How to Write a Framework Wrapper

This document is for people who want to write a framework wrapper for Aspen. If you only want to use Aspen with an existing framework wrapper, then the Dispatch and Simplates documents should cover what you need.

API Reference

The primary class that the `aspen` library provides is *`RequestProcessor`*. See *`testing`* for helpers to integrate Aspen into your framework's testing infrastructure, and see the *`exceptions`* module for all exceptions that Aspen raises.

aspen.request_processor

The request processor dispatches requests to the filesystem (typecasting URL path variables), loads the resource from the filesystem, and then renders and encodes the resource.

class `aspen.request_processor.`**`RequestProcessor`**`(`***kwargs*`)`
Define a parasitic request processor.

It depends on a host framework for real request/response objects.

`process``(`*path*, *querystring*, *accept_header*, *raise_immediately=None*, *return_after=None*, ***kw*`)`
Given a path, querystring, and Accept header, return a state dict.

`configure``(`***kwargs*`)`
Takes a dictionary of strings/unicodes to strings/unicodes.

`is_dynamic``(`*fspath*`)`
Given a filesystem path, return a boolean.

`get_resource_class``(`*fspath*`)`
Given a filesystem path, return a resource class.

aspen.testing

This module provides helpers for testing applications that use Aspen.

`aspen.testing.`**`teardown`**`()`
Standard teardown function.

- reset the current working directory
- remove FSFIX = %{tempdir}/fsfix
- reset Aspen's global state
- clear out sys.path_importer_cache

class aspen.testing.**Harness**
A harness to be used in the Aspen test suite itself. Probably not useful to you.

simple(*contents=u'Greetings, program!'*, *filepath=u'index.html.spt'*, *uripath=None*, *querystring=u''*, *request_processor_configuration=None*, ***kw*)
A helper to create a file and hit it through our machinery.

aspen.exceptions

This module defines all of the custom exceptions used across the Aspen library.

exception aspen.exceptions.**ConfigurationError**(*msg*)
This is an error in any part of our configuration.

exception aspen.exceptions.**LoadError**
Represent a problem loading a resource.

CHAPTER 3

See Also

The Keystone web framework was inspired by Aspen.

Python Module Index

a

Index