Aspen

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 [https://twitter.com/progrium/status/773694289033383937] 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 [http://django.aspen.io/], Flask-Aspen [http://flask.aspen.io/], or Pando [http://pando.aspen.io/]. See the project homepage [http://aspen.io/] for an overview.

This version of Aspen has been tested with Python 3.6, 3.7, 3.8 and 3.9, on both Ubuntu and Windows.

Aspen's source code is on GitHub [https://github.com/AspenWeb/aspen.py], and is MIT-licensed [https://github.com/AspenWeb/aspen.py/blob/master/COPYRIGHT].

Installation

Aspen is available on PyPI [https://pypi.python.org/pypi/aspen]:

```
$ pip install aspen
```

Contents

See Also

The Keystone [http://keystone.readthedocs.org/] web framework was inspired by Aspen.

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 [https://raw.githubusercontent.com/AspenWeb/aspen.py/master/tests/dispatch_table_data.rst].

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 [https://docs.djangoproject.com/en/1.10/topics/http/views/] or a Rails controller [http://guides.rubyonrails.org/action_controller_overview.html]—with template code, in one file with multiple sections.

Note

Check the Aspen homepage for links to simplate support for your favorite text editor [http://aspen.io/].

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.

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**:

   a. If the second section has a specline, then the first is request logic, and the rest are content sections.
   b. 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`** [https://docs.python.org/3/library/json.html#module-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 [https://docs.python.org/3/library/string.html#format-string-syntax] string replacement
- `stdlib_percent`—takes a Python string, runs it through percent-style [https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting] string replacement
- `stdlib_template`—takes a Python string, runs it through template-style [https://docs.python.org/3/library/string.html#template-strings] string replacement

Note

Check the Aspen homepage for links to plugins for other renderers [http://aspen.io/].

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 [https://tools.ietf.org/html/rfc7232#section-5.3.2]), responding with the only available representation.

Note

Aspen delegates to the python-mimeparse [https://pypi.python.org/pypi/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.http

class `aspen.http.mapping.`**`Mapping`**

Base class for HTTP mappings.

Mappings in HTTP differ from Python dictionaries in that they may have one or more values. This dictionary subclass maintains a list of values for each key. However, access semantics are asymmetric: subscript assignment clobbers to list, while subscript access returns the last item. Think about it.

Warning

This isn't thread-safe.

`keyerror`(*key*)

Called when a key is missing. Default implementation simply reraises.

`pop`(*name*, *default=<object object>*)

Given a name, return a value.

This removes the last value from the list for name and returns it. If there was only one value in the list then the key is removed from the mapping. If name is not present and default is given, that is returned instead. Otherwise, self.keyerror is called.

`popall`()

D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised

`all`(*name*)

Given a name, return a list of values, possibly empty.

get(*name*, *default=None*)

Override to only return the last value.

add(*name*, *value*)

Given a name and value, clobber any existing values with the new one.

ones(**names*)

Given one or more names of keys, return a list of their values.

class aspen.http.request.**PathPart**

Represents a segment of a URL path.

params

extra data attached to this segment

Type: Mapping

aspen.http.request.**extract_rfc2396_params**(*path*)

This function implements parsing URL path parameters, per section 3.3 of RFC2396 [https://tools.ietf.org/html/rfc3986#section-3.3].

- path should be raw so we don't split or operate on a decoded character
- output is decoded

Example:

```
>>> path =
'/frisbee;color=red;size=small/logo;sponsor=w3c;color=black/i
mage.jpg'
>>> extract_rfc2396_params(path) == [
...     PathPart('frisbee', params={'color': ['red'], 'size':
['small']}),
...     PathPart('logo', params={'sponsor': ['w3c'], 'color':
['black']}),
...     PathPart('image.jpg', params={})
... ]
True
```

`aspen.http.request.`**`split_path_no_params`**(*path*)

This splits a path into parts on "/" only (no split on ";" or ",").

class `aspen.http.request.`**`Path`**(*raw*, *split_path=<function extract_rfc2396_params>*)

Represent the path of a resource.

`raw`

the unparsed form of the path - `str` [https://docs.python.org/3/library/stdtypes.html#str]

`decoded`

the decoded form of the path - `str` [https://docs.python.org/3/library/stdtypes.html#str]

`parts`

the segments of the path - `list` [https://docs.python.org/3/library/stdtypes.html#list] of `PathPart` objects

class `aspen.http.request.`**`Querystring`**(*raw*, *errors='replace'*)

Represent an HTTP querystring.

`raw`

the unparsed form of the querystring - `str` [https://docs.python.org/3/library/stdtypes.html#str]

`decoded`

the decoded form of the querystring - `str` [https://docs.python.org/3/library/stdtypes.html#str]

`aspen.http.resource.`**`open_resource`**(*request_processor*, *resource_path*)

Open a resource in read-only binary mode, after checking for symlinks.

Raises: AttemptedBreakout – if the **`resource_path`** points to a file that isn't inside any of the known `resource_directories`

This function doesn't fully protect against attackers who have the ability to create and delete symlinks inside the resource directories whenever they want, but it makes the attack more difficult and detectable.

`aspen.http.resource.`**`check_resource_path`**(*request_processor*, *resource_path*)

Return the "real" path of a file (i.e. a path without symlinks).

Raises: **AttemptedBreakout** – if the **`resource_path`** points to a file that isn't inside any of the known **`resource_directories`**

class `aspen.http.resource.`**`Static`**(*request_processor*, *fspath*)

Model a static HTTP resource.

`render`(**ignored*)

Returns the file's content as **`bytes`** [https://docs.python.org/3/library/stdtypes.html#bytes].

If the `store_static_files_in_ram` configuration option was set to **`False`** [https://docs.python.org/3/library/constants.html#False] (the default), then the file is read from the filesystem, otherwise its content is returned directly.

class `aspen.http.resource.`**`Dynamic`**

Model a dynamic HTTP resource.

`render`(*context*, *dispatch_result*, *accept_header*)

Render the resource.

Before rendering we need to determine what type of content we're going to send back, by trying to find a match between the media types the client wants and the ones provided by the resource.

The two sources for what the client wants are the extension in the request URL (**`dispatch_result.extension`**), and the `Accept` header (**`accept_header`**). If the former fails to match we raise **`NotFound`** (404), if the latter fails we raise **`NegotiationFailure`** (406).

Note that we don’t always respect the `Accept` header (the spec says we can ignore it: <https://tools.ietf.org/html/rfc7231#section-5.3.2>).

Parameters:

- **context** (*dict* [https://docs.python.org/3/library/stdtypes.html#dict]) – the variables you want to pass to the resource
- **dispatch_result** (*DispatchResult*) – the object returned by the dispatcher
- **accept_header** (*str* [https://docs.python.org/3/library/stdtypes.html#str]) – the requested media types

Returns: an **`Output`** object.

aspen.request_processor

The request processor dispatches requests to the filesystem, typecasts URL path variables, loads the resource from the filesystem, and then renders and encodes the resource (if it’s dynamic).

class `aspen.request_processor.`**`RequestProcessor`**(***kwargs*)

A core request processor designed for integration into a host framework.

The `kwargs` are for configuration, see **`DefaultConfiguration`** for valid keys and default values.

`dispatch`(*path*)

Call the dispatcher and inject the path variables into the given path object.

Parameters: **path** (*Path*) – the requested path, e.g. **`'/foo'`**

Returns: A **`DispatchResult`** object.

`process`(*path*, *querystring*, *accept_header*, *context*)

Process a request.

Parameters:

- **path** (*Path*) – the requested path, e.g. **`Path('/foo')`**
- **querystring** (*Querystring*) – the query parameters, e.g. **`Querystring('?bar=baz')`**
- **accept_header** (*str* [https://docs.python.org/3/library/stdtypes.html#str]) – the value of the HTTP header `Accept`
- **context** (*dict* [https://docs.python.org/3/library/stdtypes.html#dict]) – the context variables passed to dynamic resources

Returns:

A 3-tuple `(dispatch_result, resource, output)`. The latter two are set to **`None`** [https://docs.python.org/3/library/constants.html#None] if dispatching failed.

`is_dynamic`(*fspath*)

Given a filesystem path, return a boolean.

`get_resource_class`(*fspath*)

Given a filesystem path, return a resource class.

`guess_media_type`(*filename*)

Guess the media type of a file by looking at its extension.

This method is a small wrapper around **`mimetypes.guess_type()`** [https://docs.python.org/3/library/mimetypes.html#mimetypes.guess_type]. It returns **`media_type_default`** if the guessing fails.

class `aspen.request_processor.`**`DefaultConfiguration`**

Default configuration values.

`changes_reload` = *False*

Reload files on every request if they've been modified. This can be costly, so it should be turned off in production.

`charset_static` = *None*

The charset of your static files. It ends up as a `charset=` parameter in `Content-Type` HTTP headers (if the framework on top of Aspen supports that).

`dispatcher_class` = *None*

The kind of dispatcher that will be used to route requests to files. By default **`UserlandDispatcher`** is used, unless `changes_reload` is set to `True`, then **`HybridDispatcher`** is used.

`encode_output_as` = *'UTF-8'*

The encoding to use for dynamically-generated output.

`indices` = *['index.html', 'index.json', 'index', 'index.html.spt', 'index.json.spt', 'index.spt']*

List of file names that will be treated as directory indexes. The order matters.

`media_type_default` = *'text/plain'*

If the `Content-Type` of a response can't be determined, then this one is used.

`media_type_json` = *'application/json'*

The media type to use for the JSON format.

`project_root` = *None*

The root directory of your project.

`renderer_default` = *'stdlib_percent'*

The default renderer for simplates.

`resource_directories` = *[]*

The list of directories containing resource files. Aspen will refuse to load any resource located outside these directories. The *www_root* directory is automatically added to this list. The *project_root* directory is added as well if it exists.

`store_static_files_in_ram` = *False*

If set to `True`, store the contents of static files in RAM.

`typecasters` = *{'float': <function <lambda>>, 'int': <function <lambda>>}*

See `aspen.request_processor.typecasting`.

`www_root` = *None*

The root directory of your web app, containing the files it will serve. Defaults to the current directory.

aspen.request_processor.dispatcher

This module implements finding the file that matches a request path.

`aspen.request_processor.dispatcher.`**`strip_matching_ext`**(*a*, *b*)

Given two names, strip a trailing extension iff they both have them.

class `aspen.request_processor.dispatcher.`**`DispatchStatus`**

The attributes of this class are constants that represent dispatch statuses.

`okay` = *okay*

Found a matching file.

`missing` = *missing*

No match found.

`unindexed` = *unindexed*

Found a matching node, but it's a directory without an index.

class `aspen.request_processor.dispatcher.`**`DispatchResult`**(*status*, *match*, *wildcards*, *extension*, *canonical*)

The result of a dispatch operation.

`status`

A `DispatchStatus` constant encoding the overall result.

`match`

The matching filesystem path (if status != 'missing').

`wildcards`

A dict whose keys are wildcard names, and values are as supplied by the path.

`extension`

A file extension, e.g. `json` when `foo.spt` is matched to `foo.json`.

`canonical`

The canonical path of the resource, e.g. `/` for `/index.html`.

class `aspen.request_processor.dispatcher.`**`FileNode`**(*fspath*, *type*, *wildcard*, *extension*)

Represents a file in a dispatch tree.

`fspath`

The absolute filesystem path of this node.

`type`

'dynamic' or 'static'.

Type: The node's type

`wildcard`

The name of the path variable if the node is a wildcard.

`extension`

The sub-extension of a dynamic file, e.g. `json` for `foo.json.spt`.

class `aspen.request_processor.dispatcher.`**`DirectoryNode`**(*fspath*, *wildcard*, *children*)

Represents a directory in a dispatch tree.

`fspath`

The absolute filesystem path of this node.

`wildcard`

The name of the path variable if the node is a wildcard.

`children`

The node's children as a dict (keys are names and values are nodes).

class `aspen.request_processor.dispatcher.`**`LiveDirectoryNode`**(*fspath*, *wildcard*, *children*, *mtime*, *dispatcher*)

Dynamically represents a directory in a dispatch tree.

`fspath`

The absolute filesystem path of this node.

`wildcard`

The name of the path variable if the node is a wildcard.

`mtime`

The last modification time of the directory, in nanoseconds.

`dispatcher`

Points to the `Dispatcher` object that created this node.

`aspen.request_processor.dispatcher.`**`legacy_collision_handler`**(*slug*, *node1*, *node2*)

Ignores all collisions, like `SystemDispatcher` does.

`aspen.request_processor.dispatcher.`**`strict_collision_handler`**(**args*)

A sane collision handler, it doesn't allow any.

`aspen.request_processor.dispatcher.`**`hybrid_collision_handler`**(*slug*, *node1*, *node2*)

This collision handler allows a static file to shadow a dynamic resource.

Example: `/file.js` will be preferred over `/file.js.spt`.

`aspen.request_processor.dispatcher.`**`skip_hidden_files`**(*name*, *dirpath*)

Skip all names starting with a dot, except `.well-known`.

`aspen.request_processor.dispatcher.`**`skip_nothing`**(*name*, *dirpath*)

Always returns `False` [https://docs.python.org/3/library/constants.html#False].

class `aspen.request_processor.dispatcher.`**`Dispatcher`**(*www_root*, *is_dynamic*, *indices*, *typecasters*, *file_skipper=<function skip_hidden_files>*, *collision_handler=<function hybrid_collision_handler>*)

The abstract base class of dispatchers.

Parameters:

- **www_root** – the path to a filesystem directory
- **is_dynamic** – a function that takes a file name anc returns a boolean
- **indices** – a list of filenames that should be treated as directory indexes
- **typecasters** – a dict of typecasters, keys are string and values are functions
- **file_skipper** – a function that takes a file name an a directory path and returns a boolean
- **collision_handler** – a function that takes 3 arguments (*slug, node1, node2*) and returns a strin

`build_dispatch_tree`()

Called to build the dispatch tree.

Subclasses **must** implement this method.

`dispatch`(*path*, *path_segments*)

Dispatch a request.

Parameters:

- **path** (*str* [https://docs.python.org/3/library/stdtypes.html#str]) – the request path, e.g. `'/'`
- **path_segments** (*list* [https://docs.python.org/3/library/stdtypes.html#list]) – the path split into segments, e.g. `['']`

Subclasses **must** implement this method.

`find_index`(*dirpath*)

Looks for an index file in a directory.

Returns: the full path of the first index file, or `None` [https://docs.python.org/3/library/constants.html#None] if no index was found

`split_wildcard`(*wildcard*, *is_dir*)

Splits a wildcard into its components.

Parameters:
- **wildcard** (*str* [https://docs.python.org/3/library/stdtypes.html#str]) – the string to split, e.g. `'year.int'`
- **is_dir** (*bool* [https://docs.python.org/3/library/functions.html#bool]) – `True` [https://docs.python.org/3/library/constants.html#True] if the wildcard is from a directory name

Returns: a 3-tuple `(varname, vartype, extension)`

class `aspen.request_processor.dispatcher.`**`SystemDispatcher`**(*www_root*, *is_dynamic*, *indices*, *typecasters*, *file_skipper=<function skip_hidden_files>*, *collision_handler=<function hybrid_collision_handler>*)

Aspen's original dispatcher, it's very inefficient.

class `aspen.request_processor.dispatcher.`**`UserlandDispatcher`**(*www_root*, *is_dynamic*, *indices*, *typecasters*, *file_skipper=<function skip_hidden_files>*, *collision_handler=<function hybrid_collision_handler>*)

A dispatcher optimized for production use.

This dispatcher builds a complete and static tree when it is first created. It then uses this dispatch tree to route requests without making any system call, thus avoiding FFI and context switching costs.

This is the default dispatcher (when the `changes_reload` configuration option is `False`).

class `aspen.request_processor.dispatcher.`**`HybridDispatcher`**(*www_root*, *is_dynamic*, *indices*, *typecasters*, *file_skipper=<function skip_hidden_files>*, *collision_handler=<function hybrid_collision_handler>*)

A dispatcher optimized for development environments.

This dispatcher is almost identical to **`UserlandDispatcher`**, except that it does make some system calls to check that the matched filesystem directories haven't been modified. If changes are detected, then the dispacth tree is updated accordingly.

This is the default dispatcher when the `changes_reload` configuration option is set to `True`.

class `aspen.request_processor.dispatcher.`**`TestDispatcher`**(**args*, ***kw*)

This pseudo-dispatcher calls all the other dispatchers and checks that their results are identical. It's only meant to be used in Aspen's own tests.

aspen.request_processor.typecasting

This module handles the parsing of path variables.

`aspen.request_processor.typecasting.`**`defaults`** = *{'float': <function <lambda>>, 'int': <function <lambda>>}*

Aspen’s default typecasters.

`aspen.request_processor.typecasting.`**`apply_typecasters`**(*typecasters*, *path_vars*, *context*)

Perform typecasting (in-place!).

Parameters:

- **typecasters** – a `dict` [https://docs.python.org/3/library/stdtypes.html#dict] of type names to typecast functions
- **path_vars** – a `Mapping` of path variables
- **context** – a `dict` [https://docs.python.org/3/library/stdtypes.html#dict] passed to typecast functions as second argument

Raises: **`TypecastError`** – if a typecast function raises an exception

`aspen.simplates`

class `aspen.simplates.simplate.`**`Simplate`**(*request_processor*, *fspath*)

A simplate is a dynamic resource with multiple syntaxes in one file.

Parameters: **fspath** (*str* [https://docs.python.org/3/library/stdtypes.html#str]) – the absolute filesystem path of this simplate

`render_for_type`(*media_type*, *context*)

Render the simplate.

Parameters:

- **media_type** (*str* [https://docs.python.org/3/library/stdtypes.html#str]) – the media type of the page to render
- **context** (*dict* [https://docs.python.org/3/library/stdtypes.html#dict]) – execution context values you wish to supply

Returns: an **`Output`** object.

`parse_into_pages`(*decoded*)

Given a bytestring that is the entire simplate, return a list of pages.

If there's one page, it's a template.

If there's more than one page, the first page is always python and the last is always a template.

If there's more than two pages, the second page is python *unless it has a specline*, which makes it a template.

`compile_pages`(*pages*)

Given a list of pages, replace the pages with objects.

Page 0 is the 'run once' page - it is executed and the resulting context stored in **`self.pages[0]`**.

Page 1 is the 'run every' page - it is compiled for easier execution later, and stored in **`self.pages[1]`**.

Subsequent pages are templates, so each one's content type and respective renderer are stored as a tuple in **`self.pages[n]`**.

`compile_page`(*page*)

Given a **`Page`**, return a **`(renderer, media_type)`** pair.

`aspen.simplates.renderers.`**`factories`**(*configuration*)

return a dict of render factory names to the factories themselves

class `aspen.simplates.renderers.`**`Renderer`**(*factory*, *filepath*, *raw*, *media_type*, *offset*)

The base class of renderers.

`compile`(*filepath*, *padded*)

Override.

Whatever you return from this will be set on self.compiled the first time the renderer is called. If changes_reload is True then this will be called every time the renderer is called. You can then use self.compiled in your render_content method as needed.

`render_content`(*context*)

Override. Context is a dict.

You can use these attributes:

```
self.raw        the raw bytes of the content page
self.compiled   the result of self.compile (generally a
template in
                 compiled object form)
self.meta       the result of Factory.compile_meta
self.media_type the media type of the page
self.offset     the line number at which the page starts
```

class `aspen.simplates.renderers.`**`Factory`**(*configuration*)

The base class of renderer factories.

class **`Renderer`**(*factory*, *filepath*, *raw*, *media_type*, *offset*)

The base class of renderers.

`compile`(*filepath*, *padded*)

Override.

Whatever you return from this will be set on self.compiled the first time the renderer is called. If changes_reload is True then this will be called every time the renderer is called. You can then use self.compiled in your render_content method as needed.

`render_content`(*context*)

Override. Context is a dict.

You can use these attributes:

```
self.raw         the raw bytes of the content page
self.compiled    the result of self.compile (generally a
template in
                  compiled object form)
self.meta        the result of Factory.compile_meta
self.media_type  the media type of the page
self.offset      the line number at which the page
starts
```

`compile_meta`(*configuration*)

Takes a configuration object. Override as needed.

Whatever you return from this will be set on self.meta the first time the factory is called, or every time if changes_reload is True. You can then use self.meta in your Renderer class as needed.

aspen.output

class `aspen.output.`**`Output`**(*body=None*, *media_type=None*, *charset=None*)

The result of rendering a resource.

`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
- 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='Greetings, program!'*, *filepath='index.html.spt'*, *uripath=None*, *querystring=''*, *request_processor_configuration=None*, ***kw*)

A helper to create a file and hit it through our machinery.

`aspen.testing.`**`chdir`**(*path*)

A context manager that temporarily changes the working directory.

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.`**`NegotiationFailure`**(*accept*, *available_types*)

The requested media type isn't available (HTTP status code 406).

exception `aspen.exceptions.`**`TypecastError`**(*extension*)

Parsing a segment of the request path failed (HTTP status code 404).

exception `aspen.exceptions.`**`NotFound`**(*message=''*)

The requested resource isn't available (HTTP status code 404).

exception `aspen.exceptions.`**`AttemptedBreakout`**(*sym_path*, *real_path*)

Raised when a request is dispatched to a symlinked file which is outside **`www_root`**.

exception `aspen.exceptions.`**`PossibleBreakout`**(*sym_path*, *real_path*)

A warning emitted when a symlink that points outside **`www_root`** is detected.

exception `aspen.exceptions.`**`SlugCollision`**(*slug*, *node1*, *node2*)

Raised if two files claim the same URL path.

Example: `foo.html` and `foo.html.spt` both claim `/foo.html`.

exception `aspen.exceptions.`**`WildcardCollision`**(*varname*, *fspath*)

Raised if a filesystem path contains multiple wildcards with the same name.

Examples: `www/%foo/%foo/index.spt`, `www/%foo/bar/%foo.spt`.

Python Module Index

Index