Fanstatic Documentation

Release 1.0.1.dev0

Fanstatic Developers

Feb 07, 2018

Contents

Contents:

CHAPTER 1

Introduction

Fanstatic is a small but powerful framework for the automatic publication of resources on a web page. Think Javascript and CSS. It just serves static content, but it does it really well.

Can you use it in your project? If you use Python, yes: Fanstatic is web-framework agnostic, and will work with any web framework that supports WSGI. Fanstatic is issued under the BSD license.

Why would you need something like Fanstatic? Can't you just add your static resources to some statically served directory and forget about them? For small projects this is certainly sufficient. But so much more is possible and useful in this modern Javascript-heavy world. Fanstatic is able to offer a lot of powerful features for projects both small and large.

Fanstatic has a lot of cool features:

1.1 Always the right resources

- **Import Javascript as easily as Python**: Javascript dependencies are a Python import statement away. Importing Python code is easy, why should it be harder to import Javascript code?
- **Depend in the right place**: do you have a lot of server-side code that assembles a web page? Want your datetime widget to pull in a datetime Javascript library, but only when that widget is on the page? Fanstatic lets you do that with one line of Python code.
- **Dependency tracking**: use a Javascript or CSS file that uses another one that in turn uses another one? Fanstatic knows about dependencies and will make sure all dependencies will appear on your page automatically. Have minified or rolled up versions available? Fanstatic can automatically serve those too.
- **Declare dependencies**: want to publish your own Javascript library? Have your own CSS? Does it depend on other stuff? Fanstatic lets you declare dependencies with a few lines of Python code.

1.2 Optimization

- **Serve the right version**: have alternative versions of your resource available? Want to serve minified versions of your Javascript during deployment? Debug versions during development? It's one configuration option away.
- **Bundle up resources**: roll up multiple resources into one and serve the combined resource to optimize page load time. Bundled resources can be generated automatically, or can automatically served when available.
- **Run compilers and minifiers**: Fanstatic knows how to run compilers like CoffeeScript, SASS or LESS, as well as minifiers like uglifyjs or cssmin on your resources. Just write your code in the language you prefer and let Fanstatic take care of the rest.
- **Optimize load times**: Fanstatic knows about tricks to optimize the load time of your Javascript, for instance by including `script` tags at the bottom of your web page instead of in the `head` section.

1.3 Smart Caching

- **Infinite caching**: Fanstatic can publish static resources on unique URLs, so that the cache duration can be set to infinity. This means that browsers will hold on to your static resources: web server only gets that resource request once per user and no more. If a front-end in cache is in use, you reduce that to once per resource; the cache will handle all other hits.
- **Automatic deployment cache invalidation**: Fanstatic can automatically update all your resource URLs if new versions of resources are released in an application update. No longer do you need to instruct your user to use shift-reload in their application to refresh their resources.
- **Automatic development cache invalidation**: you can instruct Fanstatic to run in development mode. It will automatically use new URLs whenever you change your code now. No longer do you as a developer need to do shift-reload whenever you change a resource; just reload the page.

1.4 Powerful Deployment

- **Automated deployment**: no longer do you need to tell people in separate instructions to publish Javascript libraries on a certain URL: Fanstatic can publish these for you automatically and transparently.
- **Pre-packaged libraries**: A lot of pre-packaged Javascript libraries are available on the PyPI and are maintained by the Fanstatic community. This can be installed into your project right away using easy_install, pip or buildout. No more complicated installation instructions, just reuse a Javascript library like you reuse Python libraries.

1.5 Compatible

- **Fits your web framework**: Fanstatic integrates with any WSGI-compliant Python web framework.
- **Roll your own**: Not happy with the details of how Fanstatic works? We've already split the Fanstatic WSGI component into separately usable components so you can mix and match and roll your own.

CHAPTER 2

Quickstart

This quickstart will demonstrate how you can integrate Fanstatic with a WSGI-based web application.

In this example, we will use Python to hook up Fanstatic to your WSGI application, but you could also use a WSGI configuration framework like Paste Deploy. For more information about this, see *our Paste Deploy documentation*.

2.1 A simple WSGI application

A simple WSGI application will stand in for your web application:

```
def app(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return ['<html><head></head><body></body></html>']
```

As you can see, it simply produces the following web page, no matter what kind of request it receives:

```
<html><head></head><body></body></html>
```

You can also include some code to start and run the WSGI application. Python includes `wsgiref`, a WSGI server implementation:

```
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    server = make_server('127.0.0.1', 8080, app)
    server.serve_forever()
```

For real-world uses you would likely want to use a more capable WSGI server, such as Paste Deploy as mentioned before, or for instance mod_wsgi.

2.2 Including resources without Fanstatic

Let's say we want to start using jQuery in this application. The way to do this without Fanstatic would be:

- download jQuery somewhere and publish it somewhere as a static resource. Alternatively use a URL to jQuery already published somewhere on the web using a content distribution network (CDN).
- modify the `<head>` section of the HTML in your code to add a `<script>` tag that references jQuery, in all HTML pages that need jQuery.

This is fine for simple requirements, but gets hairy once you have a lot of pages that need a variety of Javascript libraries (which may change dynamically), or if you need a larger selection of Javascript libraries with a more involved dependency structure. Soon you find yourself juggling HTML templates with lots of `<script>` tags, puzzling over what depends on what, and organizing a large variety of static resources.

2.3 Including resources with Fanstatic

How would we do this with Fanstatic? Like this:

```
from js.jquery import jquery

def app(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    jquery.need()
    return ['<html><head></head><body></body></html>']
```

You need to make sure that `js.jquery` is available in your project using a familiar Python library installation system such as pip, easy_install or buildout. This will automatically make the Javascript code available on your system.

2.4 Wrapping your app with Fanstatic

To use Fanstatic, you need to configure your application so that Fanstatic can do two things for you:

- automatically inject resource inclusion requirements (the `<script>` tag) into your web page.
- serve the static resources (such as jQuery.js) when a request to a resource is made.

Fanstatic provides a WSGI framework component called `Fanstatic` that does both of these things for you. Here is how you use it:

```
from fanstatic import Fanstatic

fanstatic_app = Fanstatic(app)
```

When you use `fanstatic_app`, Fanstatic will take care of serving static resources for you, and will include them on web pages when needed. You can import and `need` resources all through your application's code, and Fanstatic will make sure that they are served correctly and that the right script tags appear on your web page.

If you used `wsgiref` for instance, this is what you'd write to use the Fanstatic wrapped app:

```
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    server = make_server('127.0.0.1', 8080, fanstatic_app)
    server.serve_forever()
```

The resulting HTML looks like this:

```
<html>
  <head>
```

```
    <script type="text/javascript" src="/fanstatic/jquery/jquery.js"></script>
  </head>
  <body>
  </body>
</html>
```

Now you're off and running with Fanstatic!

CHAPTER 3

Concepts

To understand Fanstatic, it's useful to understand the following concepts.

3.1 Library

Static resources are files that are used in the display of a web page, such as CSS files, Javascript files and images. Often resources are packaged as a collection of resources; we call this a *library* of resources.

3.2 Resource inclusion

Resources can be included in a web page in several ways.

A common forms of inclusion in HTML are Javascript files, which are included using the `script` tag, for instance like this:

```
<script type="text/javascript" src="/something.js"></script>
```

and CSS files, which are included using a `link` tag, like this:

```
<link rel="stylesheet" href="/something.css" type="text/css" />
```

A common way for Javascript and CSS to be included is in `head` section of a HTML page. Javascript can also be included in script tags elsewhere on the page, such as at the bottom.

Fanstatic can generate these resource inclusions automatically for you and insert them into your web page.

Fanstatic doesn't do anything special for the inclusion of image or file resources, which could be included by the `img` or `a` tag. While Fanstatic can serve these resources for you, and also knows how to generate URLs to them, Fanstatic does not automatically insert them into your web pages: that's up to your application.

3.3 Resource definitions

Fanstatic lets you define resources and their dependencies to make the automated rendering of resource inclusions possible.

You can see a resource inclusion as a Python import: when you import a module, you import a particular file in a particular package, and a resource inclusion is the inclusion of a particular resource (`.js` file, `.css` file) in a particular library.

A resource may depend on other resources. A Javascript resource may for instance require another Javascript resource. An example of this is jQuery UI, which requires the inclusion of jQuery on the page as well in order to work.

Fanstatic takes care of inserting these resources inclusions on your web page for you. It makes sure that resources with dependencies have their dependencies inserted as well.

3.4 Resource requirements

How do you tell Fanstatic that you'd like to include jQuery on a web page? You do this by making an *resource requirement* in Python: you state you *need* a resource.

It is common to construct complex web pages on the server with cooperating components. A datetime widget may for instance expect a particular datetime Javascript library to be loaded. Pages but also sub-page components on the server may have inclusion requirements; you can effectively make resource requirements anywhere on the server side, as long as the code is executed somewhere during the request that produces the page.

CHAPTER 4

Creating a Resource Library

We've seen how to reuse existing resources, but how do you publish your own resources using Fanstatic?

Here's how:

4.1 Your project

So, you're developing a Python project. It's set up in the standard Python way, along these lines:

```
fooproject/
   setup.py
   foo/
     __init__.py
```

4.2 Making Fanstatic available in your project

In order to be able to import from `fanstatic` in your project, you need to make it available first. The standard way is to include it in `setup.py`, like this:

```
install_requires=[
    'fanstatic',
]
```

4.3 Adding the resource directory

You need to place the resources in a subdirectory somewhere in your Python code.

Imagine you have some resources in a directory called `bar_resources`. You simply place this in your package:

```
fooproject/
   setup.py
   foo/
     __init__.py
     bar_resources/
       a.css
       b.js
```

Note that `bar_resources` isn't a Python package, so it doesn't have an `__init__.py`. It's just a directory.

4.4 Declaring the Library

You need to declare a *Library* for `bar`. In `__init__.py` (or any module in the package), write the following:

```
from fanstatic import Library

bar_library = Library('bar', 'bar_resources')
```

Here we construct a fanstatic Library named `bar`, and we point to the subdirectory *bar_resources* to find them.

4.5 Hooking it up to an entry point

To let Fanstatic know that this library exists so it will automatically publish it, we need to add an *entry point* for the library to your project's `setup.py`. Add this to the `setup()` function:

```
entry_points={
    'fanstatic.libraries': [
        'bar = foo:bar_library',
        ],
    },
```

This tells Fanstatic that there is a `Library` instance in the `foo` package. What if you had defined the library not in `__init__.py` but in a module, such as `foo.qux`? You would have referred to it using `foo.qux:bar_library`.

At this stage, Fanstatic can serve the resources in your library. The default URLS are:

```
/fanstatic/bar/a.css

/fanstatic/bar/b.js
```

4.6 Declaring resources for inclusion

While now the resources can be served, we can't actually yet `.need()` them, so that we can have Fanstatic include them on web pages for us. For this, we need to create *Resource* instances. Let's modify our original `__init__.py` to read like this:

```
from fanstatic import Library, Resource

bar_library = Library('bar', 'bar_resources')

a = Resource(bar_library, 'a.css')
```

```
b = Resource(bar_library, 'b.js')
```

Now we're done!

4.7 Depending on resources

We can start using the resources in our code now. To make sure `b.js` is included in our web page, we can do this anywhere in our code:

```
from foo import b

...

def somewhere_deep_in_our_code():
    b.need()
```

4.8 An example

Need an example where it's all put together? We maintain a Fanstatic package called `js.jquery` that wraps jQuery this way:

http://bitbucket.org/fanstatic/js.jquery/src

It's also available on PyPI:

http://pypi.python.org/pypi/js.jquery

4.9 Bonus: shipping the library

You can declare any number of libraries and resources in your application. What if you want to reuse a library in multiple applications? That's easy too: you just put your library, library entry point, resource definitions and resource files in a separate Python project. You can then use this in your application projects. If it's useful to other as well, you can also publish it on PyPi! The various `js.*` projects that we are maintaining for Fanstatic, such as `js.jquery`, are already examples of this.

4.10 Bonus: dependencies between resources

What if we really want to include `a.css` whenever we pull in `b.js`, as code in `b.js` depends on it? Change your code to this:

```
from fanstatic import Library, Resource

bar_library = Library('bar', 'bar_resources')

a = Resource(bar_library, 'a.css')

b = Resource(bar_library, 'b.js', depends=[a])
```

Whenever you `.need() b` now, you'll also get `a` included on your page.

You can also use a *Group* to group Resources together:

```
from fanstatic import Group

c = Group([a, b])
```

4.11 Bonus: a minified version

What if you have a minified version of your `b.js` Javascript called `b.min.js` available in the `bar_resources` directory and you want to let Fanstatic know about it? You just write this:

```
from fanstatic import Library, Resource

bar_library = Library('bar', 'bar_resources')

a = Resource(bar_library, 'a.css')

b = Resource(bar_library, 'b.js', minified='b.min.js')
```

If you now configure Fanstatic to use the `minified` mode, it will automatically pull in `b.min.js` instead of `b.js` whenever you do `b.need()`.

The minified files can also be created automatically, see *Compilers and Minifiers* for details on that.

4.12 Bonus: preprocessing resources

If you prefer, say, CoffeeScript to JavaScript, you can have Fanstatic run the coffeescript compiler automatically:

```
from fanstatic import Library, Resource

baz_library = Library('baz', 'baz_resources')

a = Resource(baz_library, 'a.js', compilers={'js': 'coffee'})
```

See *Compilers and Minifiers* for detailed information on that.

4.13 Bonus: bundling of resources

Bundling of resources minimizes the amount of HTTP requests from a web page. Resources from the same Library can be bundled up into one, when they have the same renderer. Bundling is disabled by default. If you want bundling, set *bundle* to True:

```
from fanstatic import Library, Resource, Inclusion

qux_library = Library('qux', 'qux_resources')

a = Resource(qux_library, 'a.css')
b = Resource(qux_library, 'b.css')

needed = fanstatic.init_needed()
```

```
a.need()
b.need()

Inclusion(needed, bundle=True)
```

The resulting URL looks like this:

```
http://localhost/fanstatic/qux/:bundle:a.css;b.css
```

The fanstatic publisher knows about bundle URLs and serves a bundle of the two files.

If you don't want your Resource to be bundled, give it the `dont_bundle` argument.:

```
c = Resource(qux_library, 'a.css', dont_bundle=True)
```

Resources are bundled based on their Library. This means that bundles don't span Libraries. If we were to allow bundles that span Libraries, we would get inefficient bundles. For an example look at the following example situation.:

```
from fanstatic import Library, Resource

foo = Library('foo', 'foo')
bar = Library('bar', 'bar')

a = Resource(foo, 'a.js')
b = Resource(bar, 'b.js', depends=[a])
c = Resource(bar, 'c.js', depends=[a])
```

If we *need()* resource b in page 1 of our application and would allow cross-library bundling, we would get a bundle of a + b. If we then need only resource c in page 2 of our application, we would render a bundle of a + c. In this example we see that cross-library bundling can lead to inefficient bundles, as the client downloads 2 * a + b + c. Fanstatic doesn't do cross-library bundling, so the client downloads a + b + c.

When bundling resources, things could go haywire with regard to relative URLs in CSS files. Fanstatic prevents this by taking the dirname of the Resource into account:

```
from fanstatic import Library, Resource

foo = Library('foo', 'foo')

a = Resource(foo, 'a.css')
b = Resource(foo, 'sub/sub/b.css')
```

Fanstatic won't bundle *a* and *b*, as *b* may have relative URLs that the browser would not be able to resolve. We *could* rewrite the CSS and inject URLs to the proper resources in order to have more efficient bundles, but we choose to leave the CSS unaltered.

CHAPTER 5

Optimization

There are various optimizations for resource inclusion that Fanstatic supports. Because some optimizations can make debugging more difficult, the optimizations are disabled by default.

We will summarize the optimization features that Fanstatic offers here. See the *configuration section* and the *API documentation* for more details.

- minified resources. Resources can specify minified versions using the mode system. You can then configure Fanstatic to preferentially serve resources in a certain mode, such as `minified`.
- rolling up of resources. Resource libraries can specify rollup resources that combine multiple resources into one. This reduces the amount of server requests to be made by the web browser, and can help with caching. This can be controlled with the `rollup` configuration parameter.
- bundling of resources. Resource bundles combine multiple resources into one. This reduces the amount of server requests to be made by the web browser, and help with caching. This can be controlled with the `bundle` configuration parameter.
- infinite caching. Fanstatic can serve resources declaring that they should be cached forever by the web browser (or proxy cache), reducing the amount of hits on the server. Fanstatic makes this safe even when you upgrade or modify resources by its versioning technology. This can be controlled with the `versioning` and `recompute_hashes` configuration parameters.
- Javascript inclusions at the bottom of the web page. This can speed up the time web pages render, as the browser can start displaying the web page before all Javascript resources are loaded. This can be controlled using the `bottom` and `force_bottom` configuration parameters.

To find out more about these and other optimizations, please read this best practices article that describes some common optimizations to speed up page load times.

CHAPTER 6

Compilers and Minifiers

Fanstatic supports running external programs to create transformations of resource files. There are two use cases for this: The first use case is compiling files written in languages like CoffeeScript or SASS into JavaScript and CSS, respectively. The second use case is automatically generating minified versions of JS and CSS files. We call programs for the first case *compilers* and those for the second case *minifiers*, and use *compiling* as the encompassing term.

6.1 Running compilers

There are two ways of running compilers, one is manually via the command-line program `fanstatic-compile`. The other way is on-the-fly when processing a request: When the *compile* option is set to True (see *Configuration options*), Fanstatic will check on each request whether the source file is older than the compiled file, and invoke the compiler if needed.

```
Usage: fanstatic-compile my.package.name
Compiles and minifies all Resources declared in the given package.
```

Fanstatic also provides a hook into setuptools to run compilers during sdist creation, so you can package and deploy the compiled resources and don't need any of the compilers in the production environment. To use this, add the following to the `setup()` call in your package's `setup.py`:

```
setup(
  ...
  cmdclass={'sdist': fanstatic.sdist_compile},
  ...
)
```

Then, run `python setup.py sdist` as usual to create your sdist.

Note: If you are using version control plugins (e.g. setuptools_hg) to collect the files to include in your sdist, and do not check in the compiled/minified files, they will not be included in the sdist. In that case, you will need to create a `MANIFEST.in` file to pick them up, for example:

```
recursive-include src *.css *.js
```

6.2 Configuring compilers

Compilers work by creating the resource file from a source file. For example, the CoffeeScript compiler creates `foo.js` from `foo.coffee`. This is configured like so:

```
from fanstatic import Library, Resource

js_library = Library('js', 'js_resources')

a = Resource(js_library, 'a.js', compiler='coffee', source='a.coffee')
```

When compilation is run and `a.js` is not present, or older than `a.coffee`, Fanstatic will run the CoffeeScript compiler on `a.coffee` to produce `a.js`.

Compilers can have knowledge what the source files are typically named, so usually you don't have to specify that explicitly on each Resource (if you do specify a `source` that of course is used, overriding what the Compiler thought).

You can also configure compilers on the level of the Library, so they apply to all Resources with a given extension:

```
from fanstatic import Library, Resource

coffee_library = Library('coffee', 'coffee_resources',
                         compilers={'.js': 'coffee'})

a = Resource(coffee_library, 'b.js')
b = Resource(coffee_library, 'plain.js', compiler=None)
```

Note that individual Resources can override the compiler set on the Library.

6.3 Configuring minifiers

Minifiers work by creating a minified version of the resource file. For example, jsmin creates `foo.min.js` from `foo.js`. This is configured like so:

```
from fanstatic import Library, Resource

js_library = Library('js', 'js_resources')

a = Resource(js_library, 'a.js', minified='a.min.js', minifier='jsmin')
```

Minifiers can have a built-in rule what the target filename looks like, so usually you don't have to explicitly specify `minified=`.

You can also configure minifiers on the level of the Library, so they apply to all Resources with a given extension:

```
from fanstatic import Library, Resource

js_library = Library('js', 'js_resources', minifiers={'.js': 'jsmin'})

a = Resource(js_library, 'a.js')
b = Resource(js_library, 'tricky.js', minifier=None, minified='tricky.min.js')
```

Note that individual Resources can override the minifier set on the Library.

6.4 Pre-packaged compilers

Fanstatic includes the following compilers:

coffee CoffeeScript, a little language that compiles to JavaScript, requires the `coffee` binary (`npm install -g coffeescript`)

less LESS, the dynamic stylesheet language, requires the `lessc` binary (`npm install -g less`)

sass SASS, Syntactically Awesome Stylesheets, requires the `sass` binary (`gem install sass`)

Fanstatic includes the following minifiers:

cssmin cssmin, A Python port of the YUI CSS compression algorithm, requires the `cssmin` package. Use the extras requirement `fanstatic[cssmin]` to install this dependency.

jsmin jsmin, A Python port of Douglas Crockford's `jsmin`, requires the `jsmin` package. Use the extras requirement `fanstatic[jsmin]` to install this dependency.

closure closure, A Python wrapper around the Google Closure Compiler. Use the extras requirement `fanstatic[closure]` to install this dependency.

6.5 Hiding source files

You can prevent the Fanstatic publisher from serving the source files in by using the *ignores* configuration option.

6.6 Writing compilers

A compiler is a class that conforms to the following interface:

class `fanstatic.compiler.`**Compiler**

Generates a target file from a source file.

__call__(*resource*, *force=False*)

Perform compilation of `resource`.

Parameters **force** – If True, always perform compilation. If False (default), only perform compilation if `should_process` returns True.

__weakref__

list of weak references to the object (if defined)

available

Whether this compiler is available, i.e. necessary dependencies like external commands or third-party packages are installed.

should_process(*source*, *target*)

Determine whether to process the resource, based on the mtime of the target and source.

source_path(*resource*)

Return an absolute path to the source file (to use as input for compilation)

target_path(*resource*)

Return an absolute path to the target file (to use as output for compilation)

Fanstatic provides generic base classes for both compilers and minifiers, as well as helper classes for compilers that run external commands or depend on other Python packages (`fanstatic.compiler.CommandlineBase`, `fanstatic.compiler.PythonPackageBase`).

To make a compiler or minifier known to Fanstatic, it needs to be declared as an *entry point* in its packages' `setup.py`:

```
entry_points={
    'fanstatic.compilers': [
        'coffee = fanstatic.compiler:COFFEE_COMPILER',
        ],
    'fanstatic.minifiers': [
        'jsmin = fanstatic.compiler:JSMIN_MINIFIER',
        ],
    },
```

CHAPTER 7

Injector plugins

Fanstatic allows you to write your own injector plugins. Injector plugins take care of injecting the needed resources into the HTML of the response.

The default injector plugin is the "TopBottomInjector", which injects resources into the top (the head section) and bottom (before the closing body tag) of the page.

To write your own injector plugin, you need to do the following:

```
from fanstatic.injector import InjectorPlugin

class MyInjector(InjectorPlugin):

    name = 'mine'

    def __init__(self, options):
        """Optionally, you can control the configuration of the injector
        plugin here. The options are taken from the local_conf of the paste
        deploy configuration. Don't forget to super()."""

    def __call__(self, html, needed, request=None, response=None):
        """Render the needed resources into the html.
           The request and response arguments are
           webob Request and Response objects that may be relevant for how
           you want to inject the resources.

           You may want to group the resources in the needed resources.
           For every group call self.make_inclusion(), which will return an
           Inclusion object. Calling render() on an Inclusion object,
           will return an html snippet, which you can then include in the
           html.
        """
        needed_html = self.make_inclusion(needed).render()
        return html.replace('<head>', '<head>%s' % needed_html, 1)
```

After writing the plugin code, register the plugin through the "fanstatic.injectors" entry point.

An example of an injector plugin with configuration taken from paste deploy can be found in the sylva.fanstatic package.

CHAPTER 8

Configuration options

Fanstatic makes available a number of configuration options. These can be passed to the `Fanstatic` WSGI component as keyword arguments. They can also be configured using Paste Deploy configuration patterns (see *our Paste Deploy documentation* for more information on that).

8.1 versioning

If you turn on versioning, Fanstatic will automatically include a version identifier in the resource URLs it generates and injects into web pages. This means that for each version of your Javascript resource its URL will be unique. The Fanstatic publisher will set cache headers for versioned resource URLs so that they will be cached forever by web browsers and caching proxies[1].

By default, versioning is disabled, because it needs some extra explanation. We highly recommend you to enable it however, as the performance benefits are potentially huge and it's usually entirely safe to do so. See also `recompute_hashes` if you want to use versioning during development.

The benefit of versioning is that all resources will be cached forever by web browsers. This means that a web browser will never talk to the server to request a resource again once it retrieved it once, as long as it is still in its cache. This puts less load on your web application: it only needs to publish the resource once for a user, as long as the resource remains in that user's cache.

If you use a server-side cache such as Squid or Varnish, the situation is even better: these will hold on to the cached resources as well, meaning that your web application needs to serve the resource exactly *once*. The cache will serve them after that.

But what if you change a resource? Won't users now get the wrong, old versions of the changed resource? No: with versioning enabled, when you change a resource, a *new* URL to that resource will be automatically generated. You never will have to instruct users of your web application to do a "shift-reload" to force all resources to reload – the browser will see the resource URL has changed and will automatically load a new one.

How does this work? There are two schemes: explicit versioning and an automatically calculated hash-based versioning. An explicit version looks like this (from the js.jquery package):

[1] Well, for 10 years into the future at least.

```
/fanstatic/jquery/:version:1.4.4/jquery.js
```

A hash-based version looks like this:

```
/fanstatic/my_library/:version:d41d8cd98f00b204e9800998ecf8427e/my_resource.js
```

The version of Resource depends on the version of the python package in which the Library is defined: it takes the explicit version information from this. If no version information can be found or if the python package is installed in development mode, we still want to be able to create a unique version that changes whenever the content of the resources changes.

To this end, the most recent modification time from the files and directories in the Library directory is taken. Whenever you make any changes to a resource in the library, the hash version will be automatically recalculated.

The benefit of calculating a hash for the Library directory is that resource URLs change when a referenced resource changes; If resource A (i.e. `logo.png`) in a library that is referenced by resource B (i.e. `style.css`) changes, the URL for resource A changes, not because A changed, but because the contents of the library to which A and B belong has changed.

Fanstatic also provides an MD5-based algorithm for the Library version calculation. This algorithm is slower, but you may use if you don't trust your filesystem. Use it through the `versioning_use_md5` parameter.

8.2 recompute_hashes

If you enable `versioning`, Fanstatic will automatically calculate a resource hash for each of the resource directories for which no version is found.

During development you want the hashes to be recalculated each time you make a change, without having to restart the application all the time, and having a little performance impact is no problem. The default behavior is to recompute hashes for every request.

Calculating a resource hash is a relatively expensive operation, and in production you want Fanstatic to calculate the resource hash only once per library, by setting `recompute_hashes` to false. Hashes will then only be recalculated after you restart the application.

8.3 bottom

While CSS resources can only be included in the `<head>` section of a web page, Javascript resources can be included in `<script>` tags anywhere on the web page. Sometimes it pays off to do so: by including Javascript resources at the bottom of a web page (just before the `</body>` closing tag), the page can already load and partially render for the user before the Javascript files have been loaded, and this may lead to a better user experience.

Not all Javascript files can be loaded at this time however: some depend on being included as early as possible. You can mark a *`Resource`* as "bottom safe" if they are safe to load at the bottom of the web page. If you then enable `bottom`, those Javascript resources will be loaded there. If `bottom` is turned off (the default), all Javascript resources will be included in the `<head>` section.

8.4 force_bottom

If you enable `force_bottom` (default it's disabled) then if you enable `bottom`, *all* Javascript resources will be included at the bottom of a web page, even if they're not marked "bottom safe".

8.5 minified and debug

By default, the resource URLs included will be in the normal human-readable (and debuggable) format for that resource.

When creating *Resource* instances, you can specify alternative modes for the resource, such as minified and debug versions. The argument to `minified` and `debug` are a resource path or resource that represents the resource in that alternative mode.

You can configure Fanstatic so that it prefers a certain mode when creating resource URLs, such as `minified`. In this case Fanstatic will preferentially serve minified alternatives for resources, if available. If no minified version is available, the default resource will be served.

8.6 ignores

You can prevent the Fanstatic publisher from publishing certain files and directories by using the `ignores` option. You can leave the source files of your graphics and client side logic near the result files without worrying about Fanstatic 'leaking' this information. The `ignores` option accepts a list of glob patterns.

8.7 rollup

A performance optimization to reduce the amount of requests sent by a client is to roll up several resources into a bundle, so that all those resources are retrieved in a single request. This way a whole collection of resources can be served in one go.

You can create special *Resource* instances that declare they supersede a collection of other resources. If `rollup` is enabled, Fanstatic will serve a combined resource if it finds out that all individual resources that it supersedes are needed.

8.8 base_url

The `base_url` URL will be prefixed in front of all resource URLs. This can be useful if your web framework wants the resources to be published on a sub-URL. By default, there is no `base_url`, and resources are served in the script root.

Note that this can also be set using the `set_base_url` method on a *NeededResources* instance during run-time, as this URL is generally not known when *NeededResources* is instantiated.

8.9 publisher_signature

The default publisher signature is `fanstatic`. What this means is that the *Fanstatic()* WSGI component will look for the string `/fanstatic/` in the URL path, and if it's there, will take over to publish resources. If you would like the root for resource publication to be something else in your application (such as `resources`), you can change this to another string.

8.10 bundle

Bundling of resources minimizes HTTP requests from the client by finding efficient bundles of resources. In order to configure bundling of resources, set the `bundle` argument to True.

8.11 compile

To automatically run compilers and minifiers when needed, set the `compile` argument to True. (This argument is only about running compilers automatically; you can always compile your resources manually via the `fanstatic-compile` command-line program.)

CHAPTER 9

Paste Deployment

Fanstatic has support for Paste Deployment, a system for configuring WSGI applications and servers. You can configure the Fanstatic WSGI components using Paste Deploy.

9.1 Fanstatic WSGI component

If you have configured your application with Paste, you will already have a configuration `.ini` file, say `deploy.ini`. You can now wrap your application in the *`Fanstatic()`* WSGI component:

```
[server:main]
use = egg:Paste#http

[app:my_application]
use = egg:myapplication

[pipeline:main]
pipeline = fanstatic my_application

[filter:fanstatic]
use = egg:fanstatic#fanstatic
```

The *`Fanstatic()`* WSGI framework component actually itself combines three separate WSGI components - the *`Injector`*, the *`Delegator`* and the *`Publisher`* - into one convenient component.

The `[filter:fanstatic]` section accepts several configuration directives (see also the *configuration documentation*):

Turn recomputing of hashes on or off with “true” or “false”:

```
recompute_hashes = true
```

To turn versioning on or off with “true” or “false”:

```
versioning = true
```

You can also configure the URL segment that is used in generating URLs to resources and to recognize "serve-able" resource URLs:

```
publisher_signature = foo
```

To allow for bottom inclusion of resources:

```
bottom = true
```

To force *all* Javascript to be included at the bottom:

```
force_bottom = true
```

To serve `minified` resources where available:

```
minified = True
```

To serve `debug` resources where available:

```
debug = True
```

Use rolled up resources where possible and where they are available:

```
rollup = true
```

Use bundling of resources:

```
bundle = true
```

Use compilation or resources:

```
compile = true
```

Configure an injector plugin, by name:

```
injector = foo
```

A complete `[filter:fanstatic]` section could look like this:

```
[filter:fanstatic]
use = egg:fanstatic#fanstatic
recompute_hashes = false
versioning = true
bottom = true
minified = true
```

The Fanstatic WSGI component is all you should need for normal use cases. Next, we will go into the details of what the sub-components that this component consists of. These should only be useful in particular use cases when you want to take over some of the task of Fanstatic itself.

9.2 Injector WSGI component

If you don't want to use the Publisher component as you want to serve the libraries yourself, you can still take care of injecting URLs by configuring the *Injector* WSGI component separately:

```
[server:main]
use = egg:Paste#http

[app:my_application]
use = egg:myapplication

[pipeline:main]
pipeline = injector my_application

[filter:injector]
use = egg:fanstatic#injector
```

The `[filter:injector]` section accepts the same set of configuration parameters as the `[filter:fanstatic]` section. A complete section therefore could look like this:

```
[filter:injector]
use = egg:fanstatic#injector
recompute_hashes = false
versioning = false
bottom = true
minified = true
```

9.3 Publisher WSGI component

It is also possible to set up the `Publisher` component separately. The publisher framework component is actually a combination of a *`Delegator`* and a *`Publisher`* component. The delegator is responsible for recognizing what URLs are in fact URLs to "serve-able" resources, passing along all other URLs to be handled by your application.

The delegator recognizes URLs that contain the `publisher_signature` as a path segment are recognized as "serve-able". Configuring only the publisher component for your application implies that there is some other mechanism that injects the correct resources URLs into, for example, web pages.

The publisher component accepts one configuration directive, the `publisher_signature` (default it's set to `fanstatic`):

```
[server:main]
use = egg:Paste#http

[app:my_application]
use = egg:myapplication

[pipeline:main]
pipeline = publisher my_application

[filter:publisher]
use = egg:fanstatic#publisher
publisher_signature = bar
```

9.4 Combining the publisher and the injector

As explained before, the *`Fanstatic()`* component combines the publisher and injector components. An equivalent configuration using the separate components would look like this:

```
[server:main]
use = egg:Paste#http

[app:my_application]
use = egg:myapplication

[pipeline:main]
pipeline = publisher injector my_application

[filter:publisher]
use = egg:fanstatic#publisher
publisher_signature = baz

[filter:injector]
use = egg:fanstatic#injector
recompute_hashes = false
versioning = true
bottom = true
minified = true
publisher_signature = baz
```

CHAPTER 10

Serf: A standalone Fanstatic WSGI application

During development of Javascript code it can be useful to test your Javascript code in a very simple HTML page. Fanstatic contains a very simple WSGI application that allows you to do this: Serf.

The *Serf* class is a WSGI application that serves a very simple HTML page with a <head> and <body> section. You can give the Serf class a single resource to include. If you wrap the Serf WSGI application in a *Fanstatic* WSGI framework component, the resource and all its dependencies will be included on the web page.

10.1 Paste Deployment of Serf

Serf is mostly useful in combination with Paste Deployment, as this makes it very easy to configure a little test web application. You configure Fanstatic as discussed in the *our Paste Deploy documentation* section. You then add a serf app in a `app:` section and tell it what resource to include using the `py:<dotted_name>` notation.

A dotted name is a string that refers to a Python object. It consists of a packages, modules and objects joined together by dots, much as you can write them in Python `import` statements. `js.jquery.jquery` for instance refers to the `jquery` resource in the `js.jquery` package. This way you can refer to any package on your Python path (controlled by buildout or virtualenv).

Finally, you also must include the Serf application in the WSGI pipeline.

Here is a full example which includes the jquery resource on a HTML page:

```
[server:main]
use = egg:Paste#http

[app:serf]
use = egg:fanstatic#serf
resource = py:js.jquery.jquery

[filter:fanstatic]
use = egg:fanstatic#fanstatic

[pipeline:main]
pipeline = fanstatic serf
```

CHAPTER 11

API

11.1 WSGI components

fanstatic.**Fanstatic**(*app*, *publisher_signature='fanstatic'*, *injector=None*, ***config*)

Fanstatic WSGI framework component.

Parameters

- **app** – The WSGI app to wrap with Fanstatic.
- **publisher_signature** – Optional argument to define the signature of the publisher in a URL. The default is `fanstatic`.
- **injector** – A injector callable.
- ****config** – Optional keyword arguments. These are passed to `NeededInclusions` when it is constructed.

fanstatic.**Serf**(*resource*)

Serf WSGI application.

Serve a very simple HTML page while needing a resource. Can be configured behind the *Fanstatic()* WSGI framework component to let the resource be included.

Parameters **resource** – The *Resource* to include.

class fanstatic.**Injector**(*app*, *injector=None*, ***config*)

Fanstatic injector WSGI framework component.

This WSGI component takes care of injecting the proper resource inclusions into HTML when needed.

This WSGI component is used automatically by the *Fanstatic()* WSGI framework component, but can also be used independently if you need more control.

Parameters

- **app** – The WSGI app to wrap with the injector.

- ****config** – Optional keyword arguments. These are passed to *NeededResources* when it is constructed. It also makes sure that when initialized, it isn't given any configuration parameters that cannot be passed to `NeededResources`.

class fanstatic.**Publisher**(*registry*)
Fanstatic publisher WSGI application.

This WSGI application serves Fanstatic *Library* instances. Libraries are published as `<library_name>/<optional_version>/path/to/resource.js`.

All static resources contained in the libraries will be published to the web. If a step prefixed with `:version:` appears in the URL, this will be automatically skipped, and the HTTP response will indicate the resource can be cached forever.

This WSGI component is used automatically by the *Fanstatic()* WSGI framework component, but can also be used independently if you need more control.

Parameters **registry** – an instance of *LibraryRegistry* with those resource libraries that should be published.

class fanstatic.**LibraryPublisher**(*library*)
Fanstatic directory publisher WSGI application.

This WSGI application serves a directory of static resources to the web.

This WSGI component is used automatically by the *Fanstatic()* WSGI framework component, but can also be used independently if you need more control.

Parameters **library** – The fanstatic library instance.

class fanstatic.**Delegator**(*app*, *publisher*, *publisher_signature='fanstatic'*)
Fanstatic delegator WSGI framework component.

This WSGI component recognizes URLs that point to Fanstatic libraries, and delegates them to the *Publisher* WSGI application.

In order to recognize such URLs it looks for occurrences of the `publisher_signature` parameter as a URL step. By default it looks for `/fanstatic/`.

This WSGI component is used automatically by the *Fanstatic()* WSGI framework component, but can also be used independently if you need more control.

Parameters

- **app** – The WSGI app to wrap with the delegator.
- **publisher** – An instance of the *Publisher* component.
- **publisher_signature** – Optional argument to define the signature of the publisher in a URL. The default is `fanstatic`.

11.2 Python components

class fanstatic.**Library**(*name*, *rootpath*, *ignores=None*, *version=None*, *compilers=None*, *minifiers=None*)
The resource library.

This object defines which directory is published and can be referred to by *Resource* objects to describe these resources.

Parameters

- **name** – A string that uniquely identifies this library.
- **rootpath** – An absolute or relative path to the directory that contains the static resources this library publishes. If relative, it will be relative to the directory of the module that initializes the library.
- **ignores** – A list of globs used to determine which files and directories not to publish.

init_library_nr()
This can only be called once all resources are known.

i.e. once sort_resources is called this can be called. once library numbers are calculated once this will be done very quickly.

path = None
The absolute path to the directory which contains the static resources this library publishes.

register(*resource*)
Register a Resource with this Library.

A Resource knows about its Library. After a Resource has registered itself with its Library, the Library knows about the Resources associated to it.

signature(*recompute_hashes=False*, *version_method=None*)
Get a unique signature for this Library.

If a version has been defined, we return the version.

If no version is defined, a hash of the contents of the directory indicated by `path` is calculated. If `recompute_hashes` is set to `True`, the signature will be recalculated each time, which is useful during development when changing Javascript/css code and images.

class fanstatic.**Resource**(*library*, *relpath*, *depends=None*, *supersedes=None*, *bottom=False*, *renderer=None*, *debug=None*, *dont_bundle=False*, *minified=None*, *minifier=<object object>*, *compiler=<object object>*, *source=None*, *mode_parent=None*)
A resource.

A resource specifies a single resource in a library so that it can be included in a web page. This is useful for Javascript and CSS resources in particular. Some static resources such as images are not included in this way and therefore do not have to be defined this way.

Parameters

- **library** – the *Library* this resource is in.
- **relpath** – the relative path (from the root of the library path) that indicates the actual resource file.
- **depends** – optionally, a list of resources that this resource depends on. Entries in the list are *Resource* instances.
- **supersedes** – optionally, a list of *Resource* instances that this resource supersedes as a rollup resource. If all these resources are required for render a page, the superseding resource will be included instead.
- **bottom** – indicate that this resource is "bottom safe": it can be safely included on the bottom of the page (just before `</body>`). This can be used to improve the performance of page loads when Javascript resources are in use. Not all Javascript-based resources can however be safely included that way, so you have to set this explicitly (or use the `force_bottom` option on *NeededResources*).

- **renderer** – optionally, a callable that accepts an URL argument and returns a rendered HTML snippet for this resource. If no renderer is provided, a renderer is looked up based on the resource's filename extension.
- **dont_bundle** – Don't bundle this resource in any bundles (if bundling is enabled).

mode(*mode*)
Get Resource in another mode.

If the mode is `None` or if the mode cannot be found, this `Resource` instance is returned instead.

Parameters **mode** – a string indicating the mode, or `None`.

need(*slots=None*)
Declare that the application needs this resource.

If you call `.need()` on `Resource` sometime during the rendering process of your web page, this resource and all its dependencies will be inserted as inclusions into the web page.

Parameters **slots** – an optional dictionary mapping from *Slot* instances to *Resource* instances. This dictionary describes how to fill in the slots that this resource might depend on (directly or indirectly). If a slot is required, the dictionary must contain an entry for it.

class fanstatic.**Slot**(*library*, *extension*, *depends=None*, *required=<object object>*, *default=None*)
A resource slot.

Sometimes only the application has knowledge on how to fill in a dependency for a resource, and this cannot be known at resource definition time. In this case you can define a slot, and make your resource depend on that. This slot can then be filled in with a real resource by the application when you `.need()` that resource (or when you need something that depends on the slot indirectly).

Parameters

- **library** – the *Library* this slot is in.
- **ext** – the extension of the slot, for instance '.js'. This determines what kind of resources can be slotted in here.
- **required** – a boolean indicating whether this slot is required to be filled in when a resource that depends on a slot is needed, or whether it's optional. By default filling in a slot is required.
- **depends** – optionally, a list of resources that this slot depends on. Resources that are slotted in here need to have the same dependencies as that of the slot, or a strict subset.

class fanstatic.**Group**(*depends*)
A resource used to group resources together.

It doesn't define a resource file itself, but instead depends on other resources. When a Group is depended on, all the resources grouped together will be included.

Parameters **depends** – a list of resources that this resource depends on. Entries in the list can be *Resource* instances, or *Group* instances.

need(*slots=None*)
Need this group resource.

If you call `.need()` on `Group` sometime during the rendering process of your web page, all dependencies of this group resources will be inserted into the web page.

Parameters **slots** – an optional dictionary mapping from *Slot* instances to *Resource* instances. This dictionary describes how to fill in the slots that this resource might depend on (directly or indirectly). If a slot is required, the dictionary must contain an entry for it.

class fanstatic.**NeededResources**(*versioning=False*, *versioning_use_md5=False*, *recompute_hashes=True*, *base_url=None*, *script_name=None*, *publisher_signature='fanstatic'*, *resources=None*)

The current selection of needed resources..

The `NeededResources` instance maintains a set of needed resources for a particular web page.

Parameters

- **versioning** – If `True`, Fanstatic will automatically include a version identifier in all URLs pointing to resources. Since the version identifier will change when you update a resource, the URLs can both be infinitely cached and the resources will always be up to date. See also the `recompute_hashes` parameter.
- **versioning_use_md5** – If `True`, Fanstatic will use and md5 algorithm instead of an algorithm based on the last modification time of the Resource files to compute versions. Use md5 if you don't trust your filesystem.
- **recompute_hashes** – If `True` and versioning is enabled, Fanstatic will recalculate hash URLs on the fly whenever you make changes, even without restarting the server. This is useful during development, but slower, so should be turned off during deployment. If set to `False`, the hash URLs will only be calculated once after server startup.
- **base_url** – This URL will be prefixed in front of all resource URLs. This can be useful if your web framework wants the resources to be published on a sub-URL. By default, there is no `base_url`, and resources are served in the script root. Note that this can also be set with the set_base_url method on a `NeededResources` instance.
- **script_name** – The script_name is a fallback for computing library URLs. The base_url parameter should be honoured if it is provided.
- **publisher_signature** – The name under which resource libraries should be served in the URL. By default this is `fanstatic`, so URLs to resources will start with `/fanstatic/`.
- **resources** – Optionally, a list of resources we want to include. Normally you specify resources to include by calling `.need()` on them, or alternatively by calling `.need()` on an instance of this class.

has_base_url()

Returns True if base_url has been set.

has_resources()

Returns True if any resources are needed.

library_url(*library*)

Construct URL to library.

This constructs a URL to a library, obey `versioning` and `base_url` configuration.

Parameters **library** – A *Library* instance.

need(*resource*, *slots=None*)

Add a particular resource to the needed resources.

This is an alternative to calling `.need()` on the resource directly.

Parameters

- **resource** – A *Resource* instance.
- **slots** – an optional dictionary mapping from *Slot* instances to *Resource* instances. This dictionary describes how to fill in the slots that the given resource might depend on (directly or indirectly). If a slot is required, the dictionary must contain an entry for it.

resources()
Retrieve the list of resources needed.

This returns the needed *Resource* instances. Resources are guaranteed to come earlier in the list than those resources that depend on them.

Resources are also sorted by extension.

set_base_url(*url*)
Set the base_url. The base_url can only be set (1) if it has not been set in the NeededResources configuration and (2) if it has not been set before using this method.

class fanstatic.**LibraryRegistry**(*items=()*)
Bases: fanstatic.registry.Registry

A dictionary-like registry of libraries.

This is a dictionary that maintains libraries. A value is a *Library* instance, and a key is its library `name`.

Normally there is only a single global LibraryRegistry, obtained by calling `get_library_registry()`.

Parameters **libraries** – a sequence of libraries

class fanstatic.**ConfigurationError**
Bases: exceptions.Exception

Impossible or illegal configuration.

class fanstatic.**UnknownResourceError**
Bases: exceptions.Exception

Resource refers to non-existent resource file.

class fanstatic.**UnknownResourceExtensionError**
Bases: exceptions.Exception

A resource has an unrecognized extension.

class fanstatic.**LibraryDependencyCycleError**
Bases: exceptions.Exception

Dependency cycles between libraries aren't allowed.

A dependency cycle between libraries occurs when the file in one library depends on a file in another library, while that library depends on a file in the first library.

class fanstatic.**SlotError**
Bases: exceptions.Exception

A slot was filled in incorrectly.

If a slot is required, it must be filled in by passing an extra dictionary parameter to the `.need` method, containing a mapping from the required *Slot* to *Resource*.

When a slot is filled, the resource filled in should have the same dependencies as the slot, or a subset of the dependencies of the slot. It should also have the same extension as the slot. If this is not the case, it is an error.

11.3 Functions

fanstatic.**register_inclusion_renderer**(*self*, *extension*, *renderer*, *order=None*)
Register a renderer function for a given filename extension.

Parameters

- **extension** – the filename extension to register the renderer for.
- **renderer** – a callable that should accept a URL argument and return a rendered HTML snippet for this resource.
- **order** – optionally, to control the order in which the snippets are included in the HTML document. If no order is given, the resource will be included after all other resource inclusions. The lower the order number, the earlier in the rendering the inclusion will appear.

fanstatic.**set_resource_file_existence_checking**(*v*)

Set resource file existence checking to True or False.

By default, this is set to True, so that resources that point to non-existent files will result in an error. We recommend you keep it at this value when using Fanstatic. An *UnknownResourceError* will then be raised if you accidentally refer to a non-existent resource.

When running tests it's often useful to make fake resources that don't really have a filesystem representation, so this is set to False temporarily; for the Fanstatic tests this is done. Inside a test for this particular feature, this can temporarily be set to True.

CHAPTER 12

Pre-packaged libraries

A lot of pre-packaged CSS and Javascript libraries are available on the PyPI and are maintained by the Fanstatic community. These can be installed into your project right away using `easy_install`, `pip`, `buildout` or by specifying them in `setup_requires` in `setup.py` within your `setuptools`-compatible project. No more complicated installation instructions, just reuse a CSS or Javascript library like you reuse Python libraries.

Here's a list of currently available libraries:

package	library	source
css.css3githubbuttons	CSS3 GitHub Buttons	GitHub
js.ace	Ajax.org Cloud9 Editor	Bitbucket
js.amcharts	amCharts	GitHub
js.backbone	Backbone	GitHub
js.bootstrap	Bootstrap, from Twitter	GitHub
js.chosen	Chosen	?
js.ckeditor	CKEditor	?
js.classy	Classy - Classes for JavaScript	Bitbucket
js.d3	D3.js (Data Driven Documents)	GitHub
js.d3_cloud	D3-Cloud (Wordle-style layout for D3)	GitHub
js.extjs	ExtJS: http://www.sencha.com/products/js/	Bitbucket
js.galleriffic	Galleriffic	Bitbucket
js.leaflet	Leaflet	GitHub
js.jquery_datalink	the jQuery plugin Datalink	Bitbucket
js.jquery_datatables	the jQuery plugin DataTable	Bitbucket
js.jquery_expandbox	jquery.expandBox	Bitbucket
js.jquery_form	the jQuery plugin Form	Bitbucket
js.jquery_jcrop	JCrop - Image Cropping Plugin for JQuery	GitHub
js.jquery_jgrowl	jGrowl	Bitbucket
js.jquery_jqote2	jquery.jqote2	Bitbucket
js.jquery_json	the jQuery plugin jquery-json	Bitbucket
js.jquery_jstree	the jQuery plugin JsTree	Bitbucket
js.jquery_metadata	jQuery Metadata	Bitbucket

Continued on next page

Table 12.1 – continued from previous page

js.jquery_qtip	jquery.qTip	Bitbucket
js.jquery_qunit	the jQuery plugin QUnit	Bitbucket
js.jquery_slimbox	the jQuery plugin Slimbox	Bitbucket
js.jquery_tablesorter	the jQuery plugin tablesorter	Bitbucket
js.jquery_textchildren	the jQuery plugin Text Children	Bitbucket
js.jquery_tinyscrollbar	the jQuery plugin Tiny Scrollbar	Bitbucket
js.jquery_tools	jQuery tools	Bitbucket
js.jquery_tooltip	the jQuery plugin Tooltip	Bitbucket
js.jquery_utils	jQuery Utils	Bitbucket
js.jquery	jQuery	Bitbucket
js.jqueryui	jQuery UI	Bitbucket
js.knockback	Knockback.js	Bitbucket
js.knockout	Knockout	Bitbucket
js.lesscss	less.js	Bitbucket
js.lightbox	jquery lightbox	GitHub
js.mochikit	Mochikit	Bitbucket
js.modernizr	Modernizr	?
js.raphael	Raphael	?
js.spin	spin.js	?
js.sugar	Sugar	GitHub
js.tinymce	TinyMCE	Bitbucket
js.underscore	underscore.js	?
js.yui	the YUI Library	Bitbucket

Follow the instructions in the *development section* to learn how to package your own library.

CHAPTER 13

Integration

Fanstatic can be integrated with a number of web frameworks:

- Zope/Grok through zope.fanstatic
- Pyramid through pyramid_fanstatic
- Flask through Flask-Fanstatic
- Django through django_fanstatic.

In order to integrate Fanstatic with your web framework, make sure the following conditions are met:

- **base_url**: if your web framework supports virtual hosting, make sure to set the `base_url` attribute on the `NeededResources` object.
- **Error pages**: if your web framework renders error pages, make sure to clear the `NeededResources` before rendering the error page, in order to prevent resources from the original page 'leaking' onto the error page.
- **URL calculation**: Fanstatic can also serve non-JavaScript and non-CSS resources (such as images) that you link to from the views in your application. In order to do so, we advise to support rendering URLs to resources from the view/page templates in your web framework.

CHAPTER 14

Community

14.1 Mailing list

Please talk to us on the Fanstatic mailing list: fanstatic@googlegroups.com

You can subscribe here: https://groups.google.com/group/fanstatic

You can also participate in the discussions through the Gmane group: gmane.comp.python.wsgi.fanstatic

14.2 IRC

Come to the `#fanstatic` IRC channel on FreeNode.

CHAPTER 15

Developing Fanstatic

You want to contribute to Fanstatic? Great!

Please talk to us our on our *mailing list* about your plans!

15.1 Sources

Fanstatic's source code is maintained on bitbucket: http://bitbucket.org/fanstatic

You can check out fanstatic using Mercurial (hg); see the bitbucket documentation for more information as well.

Feel free to fork Fanstatic on bitbucket if you want to hack on it, and send us a pull request when you want us to merge your improvements.

15.2 Development install of Fanstatic

Fanstatic requires Python 2.6. We believe that the Fanstatic development installation is a good example of how to install a lot of useful tools into a project's sandbox automatically; read on.

To install Fanstatic for development, first check it out, then run the buildout:

```
$ python bootstrap.py -d
$ bin/buildout
```

This uses Buildout. The buildout process will download and install all dependencies for Fanstatic, including development tools.

Don't worry, that's all you need to know about buildout to get going – you only need to run `bin/buildout` again if something changes in Fanstatic's `buildout.cfg` or `setup.py`.

The `-d` option is to instruct buildout to use Distribute instead of Setuptools and is optional.

15.3 Tests

To run the tests:

```
$ bin/py.test
```

This uses py.test. We love tests, so please write some if you want to contribute. There are many examples of tests in the `test_*.py` modules.

15.4 Test coverage

To get a test coverage report:

```
$ bin/py.test --cov fanstatic
```

To get a report with more details:

```
bin/py.test --cov-report html --cov fanstatic
```

The results will be stored in a subdirectory `htmlcov`. You can point a web browser to its `index.html` to get a detailed coverage report.

15.5 pyflakes

To run pyflakes, you can type:

```
$ bin/pyflakes fanstatic
```

15.6 Building the documentation

To build the documentation using Sphinx:

```
$ bin/sphinxbuilder
```

If you use this command, all the dependencies will have been set up for Sphinx so that the API documentation can be automatically extracted from the Fanstatic source code. The docs source is in `doc`, the built documentation will be available in `doc/_build/html`.

15.7 Python with Fanstatic on the sys.path

It's often useful to have a project and its dependencies available for import on a Python prompt for experimentation:

```
$ bin/devpython
```

You can now import fanstatic:

```
>>> import fanstatic
```

You can also run your own scripts with this custom interpreter if you like:

```
$ bin/devpython somescript.py
```

This can be useful for quick experimentation. When you want to use Fanstatic in your own projects you would normally include it in your project's `setup.py` dependencies instead.

15.8 Releases

The buildout also installs zest.releaser which can be used to make automatic releases to PyPI (using `bin/fullrelease`).

15.9 Pre-packaged libraries

If you want to make an existing JS library into a fanstatic package, use the fanstatic paster template from the fanstatictemplate package.

The pre-packaged libraries live in the http://bitbucket.org/fanstatic account.

In order to add a new library, ask one of the fanstatic administrators to create a repository for you. In the new repository, run fanstatictemplate and push your changes.

Register the newly created package on PyPI and add the fanstatic administrators (currently *faassen*, *jw* and *janjaapdriessen*) as owners. After that, add your library to the list of *Pre-packaged libraries*.

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

Symbols

A

C

D

F

G

H

I

L

M

N

P

R

S

T

U