Seitegeist Documentation

Release 0.1

Izeni, inc.

February 17, 2015

Contents

Seitegeist (Git repository on Izeni's GitLab) is a library that provides a management command, `fetch_pages`, that loads a URL, executes it with PhantomJS (including all JavaScript), and then outputs that HTML. It provides a few configurable backends to store that HTML to disk or to Amazon AWS.

CHAPTER 1

Raison d'être

Many search engines do not execute JavaScript and if your website only renders its content client-side then these search engines and other bots such as social networking sites will not be able to "see" the rendered content of your page. Many search engines and bots support a technique called the "escaped fragment", which allows the site to signal to the bot that there is an alternative, pre-rendered version of the page that it can fetch.

Detailed documentation for Google's implementation of "escaped fragment" is available online.

Seitegeist provides the basic tools needed to render a static version of the page so that non-JavaScript version of your page can be made available via the "escaped fragment" technique. This isn't it's only use but it is a primary use case.

CHAPTER 2

Installation

Having a Python 2.7 or Python 3 installation, use `pip` to install:

```
pip install git+https://dev.izeni.net/open_source/seitegeist.git@master#egg=seitegeist
```

In the future `pip install seitegeist` may work.

Also, the main dependency is PhantomJS, which is required for it to run. This can most easily be installed with NodeJS.:

```
npm install -g phantomjs
```

This may require root privileges, or you can set the environment variable `PREFIX` to something like `$HOME/.local` or to `$VIRTUAL_ENV` and it will work.

CHAPTER 3

Usage

3.1 Backend Settings

First, determine the backend you with to use. There are currently three backends. `FilelikeBackend` simply writes the HTML to a file-like object provided by your code. `DirectoryPathBackend` write the HTML to disk under a provided base path. `S3Backend` writes to an Amazon S3 bucket.

To set your backend, you need to add a value to your Django settings module.:

```
SEITEGEIST = {
    "BACKEND": "sitegeist.backends.DirectoryPathBackend",
    "BACKEND_ARGS": {"path": "somepath"},
}
```

The `SEITEGEIST` object must be a dictionary and must have the `BACKEND` key with a string as the value, and the `BACKEND_ARGS` must exist and be a dictionary.

Whatever the `BACKEND_ARGS` has to be is dependent on the code within the backend, and is documented for each one.

3.2 App code: list_pages

Next you need to write code for your app. In each Django app you wish to use Seitegeist with, you need to add a module `seites`, with a function `list_pages()`. The `list_pages` function must take an argument that will be a boolean, specifying whether the user desires to run a "full" update or not. This function must return an iterable of dictionaries (it can be a generator if you like); the values in the dictionary must be `url`, the URL that will be fetched, `dest`, which will be passed to the backed as the target value, and optionally `callback` which is a callable that will be executed after the call is done.

Example file `myapp/seites.py`:

```
from .models import MyModel

def updateobj(obj):
    def inner():
        obj.seitegeistified = True
        obj.save()

def list_pages(full=False):
    if full:
        queryset = MyModel.objects.all()
```

```
    else:
        queryset = MyModel.objects.filter(updated=True)
    for obj in queryset:
        url = obj.get_absolute_url()
        yield {'url': 'http://mypublicsite.example.com'+url,
               'dest': url.strip("/"),
               'callback': updateobj(obj)}
```

This assumes your model has a `get_absolute_url()` method. It will instruct Seitegeist to load and render the page from a fully-qualified URL and then have the backend save it to a file path. Returning a file path isn't the only option, you could use the `FilelikeBackend` and instead return a write-able file-like, like perhaps `sys.stdout`, just to test it. The `updateobj` function is a closure that returns a callback function that will make a change to the object after it Seitegeist has been processed.

3.3 Run the management command

Now that Seitegeist is configured and a list of pages to render is provided, you can run the management command `./manage.py fetch_pages myapp`.

The list of app names is optional, Seitegeist will the `seites.list_pages()` function of all apps that have one if not provided.

What this does is run PhantomJS for each page, render its HTML and execute its JavaScript, then dump the DOM as HTML to stdout, which is captured and then saved to the configured backend.

There is an optional flag to `fetch_pages`, `--full`, which will be passed to the `list_pages()` function in each app, and means the app will process this run of fetch pages with an alternate list, semantically meaning a "full" list while a normal one is just an "incremental" list.

CHAPTER 4

nginx Example

As an example of using the escaped fragment technique, suppose you have a site with an AngularJS single-page app where widget pages live under `/app#!/widgets/[slug]/`. The hash-bang pattern is part of the escaped fragment specification, but you can use HTML5 URLs as well. You can then have the S3Backend in Seitegeist configured to write to the bucket `mysite-rendered`.

Your `list_pages()` renders each `/app#!/widgets/slug/` URL to the path "widgets/slug". Now, all you need is an nginx config like this

```
http {
    ...
    server ... {
        ...
        # Numbered error code that returned calls the fragment_s3 location.
        error_page 601 = @rendered_s3;
        location @rendered_s3 {
            rewrite ^(.*)/$ /mysite-rendered$1? break;
            proxy_pass http://s3.amazonaws.com;
            proxy_http_version 1.1;
            proxy_set_header Host 's3.amazonaws.com';
            proxy_set_header Cookie '';
            proxy_read_timeout 60;
            add_header Content-Type 'text/html; charset=utf-8';
        }

        location / {
            # If the URL arguments include "_escaped_fragment_" then return a
            # rendered version.
            if ($args ~ "_escaped_fragment_=") {
                return 601;
            }
            # When using HTML5 URLS instead of hash-bang, Facebook.com doesn't
            # play nice. Use User-Agent sniffing instead in that case.
            if ($http_user_agent ~* "^facebookexternalhit") {
                return 601;
            }
            ...
            uwsgi_pass 127.0.0.1:3000;
            ...
        }
        ...
    }
}
```

This server will pass requests to uWSGI unless they either have `_escaped_fragment_` in their GET arguments or are from Facebook.com, in which case they instead sent a pre-rendered HTML file, served from the S3 bucket using a reverse-proxy.

And that's that. More examples will be written soon.

CHAPTER 5

API Documentation

5.1 Backends

Backend classes which all implement a save method with the following signature:

```
save(html, target)
```

The target can be a string specifying a target path, or some other kind of object. The arguments to save are fields in the two-tuples returned by the `seites.list_pages()` function in your app.

class `seitegeist.backends.`**`AbstractSeitegeistBackend`**
Abstract definition of the interface for a Seitegeist backend. Please subclass this and implement your own.

class `seitegeist.backends.`**`FilelikeBackend`**
FilelikeBackend writes the HTML to the target, which is a file-like object, meaning it has a write(str) method.

Settings required for this backend:

```
SEITGEIST = {
    "BACKEND": "seitegeist.backends.FilelikeBackend",
    "BACKEND_ARGS": {},
}
```

When using this backend, the `seites.list_pages()` function in your app should return something like this:

```
[('http://full.url/path/', StringIO.StringIO())]
```

Of course it doesn't have to be a `StringIO`, but anything that can be written to will work. Maybe you use `sys.stdout` to test your system, or you write your own class with a `write(str)` method which writes the rendered HTML to a remote service.

class `seitegeist.backends.`**`DirectoryPathBackend`**(*path*)
DirectoryPathBackend writes the HTML to a target file, provided by a path string.

Settings required for this backend:

```
SEITGEIST = {
    "BACKEND": "seitegeist.backends.DirectoryPathBackend",
    "BACKEND_ARGS": {"path": "/path/to/prerendered/files"}
}
```

When using this backend, the `seites.list_pages()` function in your app should return something like this:

```
[('http://full.url/a/path/', 'a/path')]
```

This will render the HTML from the URL and write it to the provided path below the base path, in this case /path/to/prerendered/files/a/path.

class seitegeist.backends.**S3Backend**(*access_key_id*, *secret_access_key*, *bucket*)
S3Backend writes the HTML to a target key path in an Amazon S3 bucket.

Settings required for this backend:

```
SEITGEIST = {
    "BACKEND": "seitegeist.backends.S3Backend",
    "BACKEND_ARGS": {
        "access_key_id": "AMZID",
        "secret_access_key": "THISISSECRET",
        "bucket": "name-of-bucket",
    },
}
```

The `access_key_id` arg should be your `AWS_ACCESS_KEY_ID`, and the `secret_access_key` be the `AWS_SECRET_ACCESS_KEY`, and the `bucket` is the name of the S3 bucket to upload to.

When using this backend, the `seites.list_pages()` function in your app should return something like this:

```
[('http://full.url/a/path/', '/a/path')]
```

This will write the HTML from the URL to your S3 bucket as https://s3.amazonaws.com/name-of-bucket/a/path.

The path will be the name of a key in your bucket. There is no provision at this time to prefix the key.

5.2 Tools

Assorted tools for importing related data and interfacing with phantomjs.

seitegeist.tools.**render_html**(*url*)
Process a given url through phantomjs and return a string of the page content.

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

S

A

D

F

R

S