django-resto Documentation

Release 1.2

Aymeric Augustin

January 29, 2016

Contents

DEPRECATED: while django-resto was a fun hack, using a public or private cloud storage service coupled to a CDN is a better choice these days.

WARNING: the architecture described below isn't a best practice anymore.

Get in touch if you're using it and would like to maintain it in the future!

CHAPTER 1

Introduction

django-resto (REplicated STOrage) provides file storage backends that can store files coming into a Django site on several servers in parallel, using HTTP. `HybridStorage` and `AsyncStorage` will store the files locally on the filesystem and remotely, while `DistributedStorage` will only store them remotely.

This works for files uploaded by users through the admin or through custom Django forms, and also for files created by the application code, provided it uses the standard storage API.

django-resto is useful for sites deployed in a multi-server environment, in order to accept uploaded files and have them available on all media servers for subsequent web requests that could be routed to any machine.

django-resto is a fork of django_dust with a strong focus on consistency, while django_dust is more concerned with availability.

CHAPTER 2

How to

2.1 Recommended setup

In an infrastructure for a Django website, each server has one (or several) of the following roles:

- Frontend servers handle directly HTTP requests for media and static files, and forward other requests to the **application servers**. For django-resto, the interesting part is that they are serving media files, so we call them **media servers**.
- Application servers run the Django application. This is where django-resto is installed.
- Database servers support the database.

If you have several application servers, you should store a **master copy** of your media files on a NAS or a SAN attached to all your application servers. If you have a single application server, you can also store the master copy on the application server itself.

In both cases, use `HybridStorage` to replicate uploaded files on all the media servers. Serving the media files from the local filesystem is more efficient than serving them from a NAS or a SAN. This is the main advantage of django-resto.

2.2 A little bit of theoretical background

Django's built-in `FileSystemStorage` goes to great lengths to avoid race conditions and ensure data integrity.

It is difficult for django-resto to provide the same guarantees, because of the CAP theorem. Instead, its storage backends can be configured to adjust the trade-off between the following properties:

- **Consistency**: this is a intrisic weakness of django-resto, because it uses plain HTTP, which doesn't provide transaction support, all the more with several hosts. However, it targets *eventual consistency*.
- **Availability**: django-resto generally improves the system's availability by replicating the data on several servers. It parallelizes storage actions to minimize response time. It also provides an asynchronous mode.
- **Partition tolerance**: django-resto is designed to cope with hardware or network failure. You can choose to maximize availability or consistency when such problems occur.

2.3 Media directories synchronization

You can configure the behavior of django-resto when a media server is unavailable:

- If RESTO_FATAL_EXCEPTIONS is True, which is the default value, django-resto will raise an exception whenever an operation doesn't succeed on all media servers. From the user's point of view, this usually results in an HTTP 500 error, unless you have some advanced error handling. This ensures that a failure won't go unnoticed.
- If RESTO_FATAL_EXCEPTIONS is False, django-resto will log a message at level ERROR for each failed upload. This is useful if you want high availability: if one media server is down, you can still upload and delete files.

In either case, since each operation is run in parallel on all media servers, it may succeed on some and fail on others. This results in an inconsistent state on the media servers. When you bring a broken server back online, you must re-synchronize the contents of its MEDIA_ROOT from the master copy, for instance with rsync. You can also set up a cron if you get random failures during load peaks. This provides eventual consistency.

Obviously, if you bring an additional media server online, you must synchronize the content of its MEDIA_ROOT from the master copy.

django_dust keeps a queue of failed operations to repeat them afterwards. This feature was removed in django-resto. It was prone to data loss, because the order of PUT and DELETE operations matters, and retrying failed operations later breaks the order. So, use rsync instead, it's fast enough.

2.4 Asynchronous operation

Once the master copy of a file is saved, you may prefer to upload it to the media servers in the background and continue your processing in the meantime. This behavior is implemented by AsyncStorage.

It improves response times, but it has two drawbacks:

- Replication lag: a client might be unable to access a file that was just uploaded because it hasn't been transferred to the media servers yet.
- No error handling: RESTO_FATAL_EXCEPTIONS is ignored and upload errors are always logged.

This works best in combination with a task queue. To use django-resto with a task queue, all you need is to subclass AsyncStorage and override its execute_one method. For instance, the following should work with rq:

```
from django_resto.storage import AsyncStorage
from redis import Redis
from rq import Queue

queue = Queue(connection=Redis())

class RqAsyncStorage(AsyncStorage):

    def execute_one(self, func, *args, **kwargs):
        return queue.enqueue(func, *args, **kwargs)
```

2.5 Low concurrency situations

You may have several servers for high availability or read performance, but still expect a low concurrency on write operations. This is a common pattern for editorial websites. In such circumstances, you can decide not to store a master copy of your media files on the application server. This behavior is implemented by DistributedStorage.

Be aware of the consequences:

- It is very highly discouraged to set RESTO_FATAL_EXCEPTIONS to False, because you could lose uploaded files entirely without an exception. As a consequence, you can't have high availability for write operations.

- Race conditions become possible: if two people upload different files with the same name at the same time, you may randomly end up with one file or the other on each media server.
- Checking if a file exists becomes more expensive, because it requires an HTTP request.

CHAPTER 3

Setup

3.1 Installation guide

django-resto is tested with:

- Django 1.4 (LTS) and 1.8 (LTS),
- all supported Python versions (except Python 2.5 for Django 1.4).

1. Download and install the package from PyPI:

```
$ pip install django-resto
```

2. Set a default file backend, if you want all your models to use it:

```
DEFAULT_FILE_STORAGE = 'django_resto.storage.HybridStorage'
```

   This is optional. You can also enable a backend only for selected fields in your models.

3. Define the list of your media servers:

```
RESTO_MEDIA_HOSTS = ['media-%02d:8080' % i for i in range(12)]
```

   OK, maybe you don't have 12 servers just yet.

4. Make sure you have configured `MEDIA_ROOT` and `MEDIA_URL`.
5. Set up your media servers to enable file uploads. See *Configuring the media servers* for some examples.

3.2 Backends

django-resto defines three backends in `django_resto.storage`.

3.2.1 `HybridStorage`

With this backend, django-resto will run all file storage operations on `MEDIA_ROOT` first, then replicate them to the media servers.

3.2.2 AsyncStorage

With this backend, django-resto will run all file storage operations on `MEDIA_ROOT` and lanch their replication to the media servers in the background. See *Asynchronous operation*.

3.2.3 DistributedStorage

With this backend, django-resto will only store the files on the media servers. See *Low concurrency situations*.

3.3 Settings

3.3.1 RESTO_MEDIA_HOSTS

Default: `()`

List of host names for the media servers.

The URL used to upload or delete a given media file is built using `MEDIA_URL`. It is the same URL used by the end user to download the file, except that the host name changes. It isn't possible to use HTTPS at this time.

3.3.2 RESTO_FATAL_EXCEPTIONS

Default: `True`

Whether to throw an exception when an operation fails on a media server.

Failed operations are always logged.

3.3.3 RESTO_SHOW_TRACEBACK

Default: `False`

Whether to include a traceback when logging an exception during an operation.

3.3.4 RESTO_TIMEOUT

Default: `2`

Timeout in seconds for HTTP operations.

This controls the maximum amount of time an upload operation can take. Note that all uploads run in parallel.

3.4 Configuring the media servers

The backend uses HTTP to transfer files to media servers. The HTTP server must support the `PUT` and `DELETE` methods according to RFC 2616.

In practice, these methods are often provided by an external module that implements WebDAV (RFC 2518). Unfortunately, WebDAV adds the concept of "collections" and changes the specification of the `PUT` methods, making it necessary to create a collection with `MKCOL` before creating a resource with `PUT`. Currently, django-resto requires a server that just implements HTTP/1.1 (RFC 2616).

It's critical to enable file uploads only from trusted IPs. Otherwise, anyone could write or delete files on your media servers.

Here is an example of lighttpd config:

```
server.modules += (
  "mod_webdav",
)

$HTTP["remoteip"] ~= "^192\.168\.0\.[0-9]+$" {
  "webdav.activate = "enable"
}
```

Here is an example of nginx config, assuming the server was compiled `--with-http_dav_module`:

```
server {
    listen 192.168.0.10;
    location / {
        root /var/www/media;
        dav_methods PUT DELETE;
        create_full_put_path on;
        dav_access user:rw group:r all:r;
        allow 192.168.0.1/24;
        deny all;
    }
}
```

CHAPTER 4

Advanced use

4.1 Extending

django-resto provides a robust base for distributing uploaded files. However, sites requiring this level of optimization often have custom requirements, and django-resto cannot cover every use case.

It would be impractical to provide settings to control every variation of the upload behavior, and it would still allow only a limited set of behaviors.

Instead, the recommended way to extend or modify the behavior of django-resto is to pick the storage class that best matches your requirements and write a subclass.

This approach is more flexible. You can to take advantage of the testing tools provided by django-resto to validate your customizations.

4.2 API stability

Functions or methods that have a docstring are considered stable. Their behavior won't change unless absolutely necessary, and if it does, the changes will be documented. They may be used or overridden in subclasses to tweak django-resto's behavior.

The stable APIs are:

- the `execute*` methods of the storage classes,
- all methods of `django_resto.storage.DefaultTransport`,
- all methods of `django_resto.http_server.TestHttpServer`,
- the function `django_resto.settings.get_setting`.

CHAPTER 5

History

5.1 1.2

- Fix unicode handling bugs on Python 2.

5.2 1.1

- Document the public API.
- Support asynchronous upload to media servers.

5.3 1.0

- Initial stable release.
- Support hybrid and distributed upload to media servers.