CHAPTER 2

Ressources

- Documentation: http://django-downloadview.readthedocs.org
- PyPI page: http://pypi.python.org/pypi/django-downloadview
- Code repository: https://github.com/benoitbryon/django-downloadview
- Bugtracker: https://github.com/benoitbryon/django-downloadview/issues
- Continuous integration: https://travis-ci.org/benoitbryon/django-downloadview
- Roadmap: https://github.com/benoitbryon/django-downloadview/issues/milestones

CHAPTER 3

Contents

3.1 Overview, concepts

Given:

- you manage files with Django (permissions, search, generation, ...)
- files are stored somewhere or generated somehow (local filesystem, remote storage, memory...)

As a developer, you want to serve files quick and efficiently.

Here is an overview of `django-downloadview`'s answer...

3.1.1 Generic views cover commons patterns

Choose the generic view depending on the file you want to serve:

- *ObjectDownloadView*: file field in a model;
- *StorageDownloadView*: file in a storage;
- *PathDownloadView*: absolute filename on local filesystem;
- *HTTPDownloadView*: URL (the resource is proxied);
- *VirtualDownloadView*: bytes, text, `StringIO`, generated file...

3.1.2 Generic views and mixins allow easy customization

If your use case is a bit specific, you can easily extend the views above or *create your own based on mixins*.

3.1.3 Views return DownloadResponse

Views return `DownloadResponse`. It is a special `django.http.StreamingHttpResponse` where content is encapsulated in a file wrapper.

Learn more in *Responses*.

3.1.4 DownloadResponse carry file wrapper

Views instanciate a *file wrapper* and use it to initialize responses.

File wrappers describe files: they carry files properties such as name, size, encoding...

File wrappers implement loading and iterating over file content. Whenever possible, file wrappers do not embed file data, in order to save memory.

Learn more about available file wrappers in *File wrappers*.

3.1.5 Middlewares convert DownloadResponse into ProxiedDownloadResponse

Before WSGI application use file wrapper to load file contents, middlewares (or decorators) are given the opportunity to capture `DownloadResponse` instances.

Let's take this opportunity to optimize file loading and streaming!

A good optimization it to delegate streaming to a reverse proxy, such as nginx [1] via X-Accel [2] internal redirects.

django_downloadview provides middlewares that convert `DownloadResponse` into `ProxiedDownloadResponse`.

Learn more in *Optimize streaming*.

3.1.6 Testing matters

django-downloadview also helps you *test the views you customized*.

You may also *write healthchecks* to make sure everything goes fine in live environments.

3.1.7 What's next?

Let's *install django-downloadview*.

Notes & references

3.2 Install

Note: If you want to install a development environment, please see *Contributing to the project*.

System requirements:

- Python 2.7

Install the package with your favorite Python installer. As an example, with pip:

```
pip install django-downloadview
```

Installing *django-downloadview* will automatically trigger the installation of the following requirements:

[1] http://nginx.org
[2] http://wiki.nginx.org/X-accel

```
REQUIREMENTS = ['setuptools', 'Django>=1.5', 'requests']
```

Note: Since version 1.1, django-downloadview requires Django>=1.5, which provides `StreamingHttpResponse`.

3.2.1 Known good set of versions

django-downloadview has been tested in an environment with the following set of versions. If something is going wrong with other versions, please report it in django-downloadview's bugtracker [3].

```
bpython = 0.12
collective.recipe.omelette = 0.16
coverage = 3.7
Django = 1.6
django-nose = 1.2
docutils = 0.11
evg.recipe.activate = 0.5
Jinja2 = 2.7.1
MarkupSafe = 0.18
mock = 1.0.1
nose = 1.3.0
Pygments = 1.6
python-termstyle = 0.1.10
rednose = 0.4.1
requests = 2.0.1
Sphinx = 1.1.3
sphinxcontrib-testbuild = 0.1.3
z3c.recipe.mkdir = 0.6
zc.recipe.egg = 2.0.1
zest.releaser = 3.48
```

Notes & references

See Also:

- *Configure*
- *Changelog*
- *License*

3.3 Configure

Here is the list of settings used by *django-downloadview*.

3.3.1 INSTALLED_APPS

There is no need to register this application in your Django's `INSTALLED_APPS` setting.

[3] https://github.com/benoitbryon/django-downloadview/issues

3.3.2 MIDDLEWARE_CLASSES

If you plan to setup reverse-proxy optimizations, add `django_downloadview.SmartDownloadMiddleware` to `MIDDLEWARE_CLASSES`. It is a response middleware. Move it after middlewares that compute the response content such as gzip middleware.

Example:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django_downloadview.SmartDownloadMiddleware'
]
```

3.3.3 DOWNLOADVIEW_BACKEND

This setting is used by `SmartDownloadMiddleware`. It is the import string of a callable (typically a class) of an optimization backend (typically a `BaseDownloadMiddleware` subclass).

Example:

```
DOWNLOADVIEW_BACKEND = 'django_downloadview.nginx.XAccelRedirectMiddleware'
```

See *Optimize streaming* for a list of available backends (middlewares).

When `django_downloadview.SmartDownloadMiddleware` is in your `MIDDLEWARE_CLASSES`, this setting must be explicitely configured (no default value). Else, you can ignore this setting.

3.3.4 DOWNLOADVIEW_RULES

This setting is used by `SmartDownloadMiddleware`. It is a list of positional arguments or keyword arguments that will be used to instanciate class mentioned as `DOWNLOADVIEW_BACKEND`.

Each item in the list can be either a list of positional arguments, or a dictionary of keyword arguments. One item cannot contain both positional and keyword arguments.

Here is an example containing one rule using keyword arguments:

```
DOWNLOADVIEW_RULES = [
    {
        'source_url': '/media/nginx/',
        'destination_url': '/nginx-optimized-by-middleware/',
    },
]
```

See *Optimize streaming* for details about builtin backends (middlewares) and their options.

When `django_downloadview.SmartDownloadMiddleware` is in your `MIDDLEWARE_CLASSES`, this setting must be explicitely configured (no default value). Else, you can ignore this setting.

3.4 Setup views

Setup views depending on your needs:

- *ObjectDownloadView* when you have a model with a file field;
- *StorageDownloadView* when you manage files in a storage;
- *PathDownloadView* when you have an absolute filename on local filesystem;
- *HTTPDownloadView* when you have an URL (the resource is proxied);
- *VirtualDownloadView* when you generate a file dynamically;
- *bases and mixins* to make your own.

3.4.1 ObjectDownloadView

`ObjectDownloadView` **serves files managed in models with file fields** such as `FileField` or `ImageField`.

Use this view like Django's builtin `DetailView`.

Additional options allow you to store file metadata (size, content-type, ...) in the model, as deserialized fields.

Simple example

Given a model with a `FileField`:

```
from django.db import models


class Document(models.Model):
    slug = models.SlugField()
    file = models.FileField(upload_to='object')
```

Setup a view to stream the `file` attribute:

```
from django_downloadview import ObjectDownloadView

from demoproject.object.models import Document


default_file_view = ObjectDownloadView.as_view(model=Document)
```

`ObjectDownloadView` inherits from `BaseDetailView`, i.e. it expects either `slug` or `pk`:

```
from django.conf.urls import patterns, url

from demoproject.object import views


urlpatterns = patterns(
    '',
    url(r'^default-file/(?P<slug>[a-zA-Z0-9_-]+)/$',
        views.default_file_view,
        name='default_file'),
)
```

Serving specific file field

If your model holds several file fields, or if the file field name is not "file", you can use `ObjectDownloadView.file_field` to specify the field to use.

Here is a model where there are two file fields:

```
from django.db import models


class Document(models.Model):
    slug = models.SlugField()
    file = models.FileField(upload_to='object')
    another_file = models.FileField(upload_to='object-other')
```

Then here is the code to serve "another_file" instead of the default "file":

```
from django_downloadview import ObjectDownloadView

from demoproject.object.models import Document


another_file_view = ObjectDownloadView.as_view(
    model=Document,
    file_field='another_file')
```

Mapping file attributes to model's

Sometimes, you use Django model to store file's metadata. Some of this metadata can be used when you serve the file.

As an example, let's consider the client-side basename lives in model and not in storage:

```
from django.db import models


class Document(models.Model):
    slug = models.SlugField()
    file = models.FileField(upload_to='object')
    basename = models.CharField(max_length=100)
```

Then you can configure the `ObjectDownloadView.basename_field` option:

```
from django_downloadview import ObjectDownloadView

from demoproject.object.models import Document


deserialized_basename_view = ObjectDownloadView.as_view(
    model=Document,
    basename_field='basename')
```

Note: `basename` could have been a model's property instead of a `CharField`.

See details below for a full list of options.

API reference

class `django_downloadview.views.object.`**`ObjectDownloadView`**`(**kwargs)`

Bases: `django.views.generic.detail.SingleObjectMixin`, `django_downloadview.views.base.BaseDownloadView`

Serve file fields from models.

This class extends `SingleObjectMixin`, so you can use its arguments to target the instance to operate on: `slug`, `slug_kwarg`, `model`, `queryset`...

In addition to `SingleObjectMixin` arguments, you can set arguments related to the file to be downloaded:

- `file_field`;
- `basename_field`;
- `encoding_field`;
- `mime_type_field`;
- `charset_field`;
- `modification_time_field`;
- `size_field`.

`file_field` is the main one. Other arguments are provided for convenience, in case your model holds some (deserialized) metadata about the file, such as its basename, its modification time, its MIME type... These fields may be particularly handy if your file storage is not the local filesystem.

`file_field` = 'file'
Name of the model's attribute which contains the file to be streamed. Typically the name of a FileField.

`basename_field` = None
Optional name of the model's attribute which contains the basename.

`encoding_field` = None
Optional name of the model's attribute which contains the encoding.

`mime_type_field` = None
Optional name of the model's attribute which contains the MIME type.

`charset_field` = None
Optional name of the model's attribute which contains the charset.

`modification_time_field` = None
Optional name of the model's attribute which contains the modification

`size_field` = None
Optional name of the model's attribute which contains the size.

`get_file`()
Return `FieldFile` instance.

The file wrapper is model's field specified as `file_field`. It is typically a `FieldFile` or subclass.

Raises `FileNotFound` if instance's field is empty.

Additional attributes are set on the file wrapper if `encoding`, `mime_type`, `charset`, `modification_time` or `size` are configured.

`get_basename`()
Return client-side filename.

`get`(*request*, **args*, ***kwargs*)

3.4.2 StorageDownloadView

`StorageDownloadView` **serves files given a storage and a path**.

Use this view when you manage files in a storage (which is a good practice), unrelated to a model.

Simple example

Given a storage:

```
from django.core.files.storage import FileSystemStorage


storage = FileSystemStorage()
```

Setup a view to stream files in storage:

```
from django_downloadview import StorageDownloadView


static_path = StorageDownloadView.as_view(storage=storage)
```

The view accepts a `path` argument you can setup either in `as_view` or via URLconfs:

```
from django.conf.urls import patterns, url

from demoproject.storage import views


urlpatterns = patterns(
    '',
    url(r'^static-path/(?P<path>[a-zA-Z0-9_-]+\.[a-zA-Z0-9]{1,4})$',
        views.static_path,
        name='static_path'),
)
```

Computing path dynamically

Override the `StorageDownloadView.get_path()` method to adapt path resolution to your needs.

As an example, here is the same view as above, but the path is converted to uppercase:

```
from django_downloadview import StorageDownloadView


class DynamicStorageDownloadView(StorageDownloadView):
    """Serve file of storage by path.upper()."""
    def get_path(self):
        """Return uppercase path."""
        return super(DynamicStorageDownloadView, self).get_path().upper()


dynamic_path = DynamicStorageDownloadView.as_view(storage=storage)
```

API reference

class `django_downloadview.views.storage.`**`StorageDownloadView`**`(`***kwargs*`)`

Bases: `django_downloadview.views.path.PathDownloadView`

Serve a file using storage and filename.

`storage` **= <django.core.files.storage.DefaultStorage object at 0x2f4eb90>**

Storage the file to serve belongs to.

path = None
: Path to the file to serve relative to storage.

get_path()
: Return path of the file to serve, relative to storage.

  Default implementation simply returns view's `path` attribute.

  Override this method if you want custom implementation.

get_file()
: Return `StorageFile` instance.

3.4.3 PathDownloadView

`PathDownloadView` **serves file given a path on local filesystem**.

Use this view whenever you just have a path, outside storage or model.

> **Warning:** Take care of path validation, especially if you compute paths from user input: an attacker may be able to download files from arbitrary locations. In most cases, you should consider managing files in storages, because they implement default security mechanisms.

Simple example

Setup a view to stream files given path:

```
import os

from django_downloadview import PathDownloadView


# Let's initialize some fixtures.
app_dir = os.path.dirname(os.path.abspath(__file__))
project_dir = os.path.dirname(app_dir)
fixtures_dir = os.path.join(project_dir, 'fixtures')
#: Path to a text file that says 'Hello world!'.
hello_world_path = os.path.join(fixtures_dir, 'hello-world.txt')

#: Serve ``fixtures/hello-world.txt`` file.
static_path = PathDownloadView.as_view(path=hello_world_path)
```

Computing path dynamically

Override the `PathDownloadView.get_path()` method to adapt path resolution to your needs:

```
import os

from django_downloadview import PathDownloadView


# Let's initialize some fixtures.
app_dir = os.path.dirname(os.path.abspath(__file__))
project_dir = os.path.dirname(app_dir)
fixtures_dir = os.path.join(project_dir, 'fixtures')
```

```
class DynamicPathDownloadView(PathDownloadView):
    """Serve file in ''settings.MEDIA_ROOT''.

    .. warning::

       Make sure to prevent "../" in path via URL patterns.

    .. note::

       This particular setup would be easier to perform with
       :class:'StorageDownloadView'

    """
    def get_path(self):
        """Return path inside fixtures directory."""
        # Get path from URL resolvers or as_view kwarg.
        relative_path = super(DynamicPathDownloadView, self).get_path()
        # Make it absolute.
        absolute_path = os.path.join(fixtures_dir, relative_path)
        return absolute_path


dynamic_path = DynamicPathDownloadView.as_view()
```

The view accepts a `path` argument you can setup either in `as_view` or via URLconfs:

```
from django.conf.urls import patterns, url

from demoproject.path import views


urlpatterns = patterns(
    '',
    url(r'^dynamic-path/(?P<path>[a-zA-Z0-9_-]+\.[a-zA-Z0-9]{1,4})$',
        views.dynamic_path,
        name='dynamic_path'),
)
```

API reference

class `django_downloadview.views.path.`**`PathDownloadView`**`(`***kwargs*`)`

Bases: `django_downloadview.views.base.BaseDownloadView`

Serve a file using filename.

`path` **= None**

Server-side name (including path) of the file to serve.

Filename is supposed to be an absolute filename of a file located on the local filesystem.

`path_url_kwarg` **= 'path'**

Name of the URL argument that contains path.

`get_path``()`

Return actual path of the file to serve.

Default implementation simply returns view's `path`.

Override this method if you want custom implementation. As an example, `path` could be relative and your custom `get_path()` implementation makes it absolute.

`get_file`()
Use path to return wrapper around file to serve.

3.4.4 HTTPDownloadView

`HTTPDownloadView` **serves a file given an URL.**, i.e. it acts like a proxy.

This view is particularly handy when:

- the client does not have access to the file resource, while your Django server does.
- the client does trust your server, your server trusts a third-party, you do not want to bother the client with the third-party.

Simple example

Setup a view to stream files given URL:

```
from django_downloadview import HTTPDownloadView


class SimpleURLDownloadView(HTTPDownloadView):
    def get_url(self):
        """Return URL of hello-world.txt file on GitHub."""
        return 'https://raw.github.com/benoitbryon/django-downloadview' \
               '/b7f660c5e3f37d918b106b02c5af7a887acc0111' \
               '/demo/demoproject/download/fixtures/hello-world.txt'


simple_url = SimpleURLDownloadView.as_view()
```

API reference

class `django_downloadview.views.http.`**`HTTPDownloadView`**(***kwargs*)
Bases: `django_downloadview.views.base.BaseDownloadView`

Proxy files that live on remote servers.

`url` **= u''**
URL to download (the one we are proxying).

`request_kwargs` **= {}**
Additional keyword arguments for request handler.

`get_request_factory`()
Return request factory to perform actual HTTP request.

Default implementation returns `requests.get()` callable.

`get_request_kwargs`()
Return keyword arguments for use with `get_request_factory()`.

Default implementation returns `request_kwargs`.

get_url()
Return remote file URL (the one we are proxying).

Default implementation returns `url`.

get_file()
Return wrapper which has an `url` attribute.

3.4.5 VirtualDownloadView

`VirtualDownloadView` **serves files that do not live on disk**. Use it when you want to stream a file which content is dynamically generated or which lives in memory.

It is all about overriding `VirtualDownloadView.get_file()` method so that it returns a suitable file wrapper...

Note: Current implementation does not support reverse-proxy optimizations, because there is no place reverse-proxy can load files from after Django exited.

Serve text (string or unicode) or bytes

Let's consider you build text dynamically, as a bytes or string or unicode object. Serve it with Django's builtin `ContentFile` wrapper:

```
from django.core.files.base import ContentFile

from django_downloadview import VirtualDownloadView


class TextDownloadView(VirtualDownloadView):
    def get_file(self):
        """Return :class:`django.core.files.base.ContentFile` object."""
        return ContentFile(u"Hello world!\n", name='hello-world.txt')
```

Serve StringIO

`StringIO` object lives in memory. Let's wrap it in some download view via `VirtualFile`:

```
from StringIO import StringIO

from django_downloadview import VirtualDownloadView
from django_downloadview import VirtualFile


class StringIODownloadView(VirtualDownloadView):
    def get_file(self):
        """Return wrapper on ``StringIO`` object."""
        file_obj = StringIO(u"Hello world!\n")
        return VirtualFile(file_obj, name='hello-world.txt')
```

Stream generated content

Let's consider you have a generator function (`yield`) or an iterator object (`__iter__()`):

```
def generate_hello():
    yield u'Hello '
    yield u'world!'
    yield u'\n'
```

Stream generated content using `VirtualDownloadView`, `VirtualFile` and `StringIteratorIO`:

```
from django_downloadview import VirtualDownloadView
from django_downloadview import VirtualFile
from django_downloadview import StringIteratorIO


class GeneratedDownloadView(VirtualDownloadView):
    def get_file(self):
        """Return wrapper on ``StringIteratorIO`` object."""
        file_obj = StringIteratorIO(generate_hello())
        return VirtualFile(file_obj, name='hello-world.txt')
```

API reference

class `django_downloadview.views.virtual.`**`VirtualDownloadView`**`(`***kwargs*`)`

Bases: `django_downloadview.views.base.BaseDownloadView`

Serve not-on-disk or generated-on-the-fly file.

Override the `get_file()` method to customize file wrapper.

`was_modified_since``(`*file_instance*, *since*`)`

Delegate to file wrapper's was_modified_since, or return True.

This is the implementation of an edge case: when files are generated on the fly, we cannot guess whether they have been modified or not. If the file wrapper implements `was_modified_since()` method, then we trust it. Otherwise it is safer to suppose that the file has been modified.

This behaviour prevents file size to be computed on the Django side. Because computing file size means iterating over all the file contents, and we want to avoid that whenever possible. As an example, it could reduce all the benefits of working with dynamic file generators... which is a major feature of virtual files.

3.4.6 Make your own view

DownloadMixin

The `django_downloadview.views.DownloadMixin` class is not a view. It is a base class which you can inherit of to create custom download views.

`DownloadMixin` is a base of BaseDownloadView, which itself is a base of all other django_downloadview's builtin views.

class `django_downloadview.views.base.`**`DownloadMixin`**

Bases: `object`

Placeholders and base implementation to create file download views.

Note: This class does not inherit from `django.views.generic.base.View`.

The `get_file()` method is a placeholder subclasses must implement. Base implementation raises `NotImplementedError`.

Other methods provide a base implementation that use the file wrapper returned by `get_file()`.

`response_class`
Response class, to be used in `render_to_response()`.

alias of `DownloadResponse`

`attachment` = True
Whether to return the response as attachment or not.

`basename` = None
Client-side filename, if only file is returned as attachment.

`mimetype` = None
File's mime type. If `None` (the default), then the file's mime type will be guessed via `mimetypes`.

`encoding` = None
File's encoding. If `None` (the default), then the file's encoding will be guessed via `mimetypes`.

`get_file``()`
Return a file wrapper instance.

Raises `FileNotFound` if file does not exist.

`get_basename``()`
Return `basename`.

Override this method if you need more dynamic basename.

`get_mimetype``()`
Return `mimetype`.

Override this method if you need more dynamic mime type.

`get_encoding``()`
Return `encoding`.

Override this method if you need more dynamic encoding.

`was_modified_since`(*file_instance*, *since*)
Return True if `file_instance` was modified after `since`.

Uses file wrapper's `was_modified_since` if available, with value of `since` as positional argument.

Else, fallbacks to default implementation, which uses `django.views.static.was_modified_since()`.

Django's `was_modified_since` function needs a datetime and a size. It is passed `modified_time` and `size` attributes from file wrapper. If file wrapper does not support these attributes (`AttributeError` or `NotImplementedError` is raised), then the file is considered as modified and `True` is returned.

`not_modified_response`(**response_args*, ***response_kwargs*)
Return `django.http.HttpResponseNotModified` instance.

`download_response`(**response_args*, ***response_kwargs*)
Return `DownloadResponse`.

`file_not_found_response``()`
Raise Http404.

`render_to_response`(**response_args*, ***response_kwargs*)
Return "download" response (if everything is ok).

Return `file_not_found_response()` if file does not exist.

Respects the "HTTP_IF_MODIFIED_SINCE" header if any. In that case, uses `was_modified_since()` and `not_modified_response()`.

Else, uses `download_response()` to return a download response.

BaseDownloadView

The `django_downloadview.views.BaseDownloadView` class is a base class to create download views. It inherits DownloadMixin and `django.views.generic.base.View`.

The only thing it does is to implement `get`: it triggers `DownloadMixin's render_to_response`.

class `django_downloadview.views.base.`**`BaseDownloadView`**`(`***kwargs*`)`

Bases: `django_downloadview.views.base.DownloadMixin`, `django.views.generic.base.View`

A base `DownloadMixin` that implements `get()`.

`get``(`*request*, **args*, ***kwargs*`)`

Handle GET requests: stream a file.

Handling http not modified responses

Sometimes, you know the latest date and time the content was generated at, and you know a new request would generate exactly the same content. In such a case, you should implement `was_modified_since()` in your view.

Note: Default `was_modified_since()` implementation trusts file wrapper's `was_modified_since` if any. Else (if calling `was_modified_since()` raises `NotImplementedError` or `AttributeError`) it returns `True`, i.e. it assumes the file was modified.

As an example, the download views above always generate "Hello world!"... so, if the client already downloaded it, we can safely return some HTTP "304 Not Modified" response:

```
from django.core.files.base import ContentFile
from django_downloadview import VirtualDownloadView

class TextDownloadView(VirtualDownloadView):
    def get_file(self):
        """Return :class:`django.core.files.base.ContentFile` object."""
        return ContentFile(u"Hello world!", name='hello-world.txt')

    def was_modified_since(self, file_instance, since):
        return False  # Never modified, always u"Hello world!".
```

3.5 Optimize streaming

Some reverse proxies allow applications to delegate actual download to the proxy:

- with Django, manage permissions, generate files...
- let the reverse proxy serve the file.

As a result, you get increased performance: reverse proxies are more efficient than Django at serving static files.

3.5.1 Supported features grid

Supported features depend on backend. Given the file you want to stream, the backend may or may not be able to handle it:

View / File	*Nginx*	*Apache*	*Lighttpd*
PathDownloadView	Yes, local filesystem.	Yes, local filesystem.	Yes, local filesystem.
StorageDownloadView	Yes, local and remote.	Yes, local filesystem.	Yes, local filesystem.
ObjectDownloadView	Yes, local and remote.	Yes, local filesystem.	Yes, local filesystem.
HTTPDownloadView	Yes.	No.	No.
VirtualDownloadView	No.	No.	No.

As an example, *Nginx X-Accel* handles URL for internal redirects, so it can manage `HTTPFile`; whereas *Apache X-Sendfile* handles absolute path, so it can only deal with files on local filesystem.

There are currently no optimizations to stream in-memory files, since they only live on Django side, i.e. they do not persist after Django returned a response. Note: there is a feature request about "local cache" for streamed files [4].

3.5.2 How does it work?

View return some `DownloadResponse` instance, which itself carries a *file wrapper*.

django-downloadview provides response middlewares and decorators that are able to capture `DownloadResponse` instances and convert them to `ProxiedDownloadResponse`.

The `ProxiedDownloadResponse` is specific to the reverse-proxy (backend): it tells the reverse proxy to stream some resource.

Note: The feature is inspired by `Django's TemplateResponse`

3.5.3 Available optimizations

Here are optimizations builtin *django_downloadview*:

Nginx

If you serve Django behind Nginx, then you can delegate the file streaming to Nginx and get increased performance:

- lower resources used by Python/Django workers ;
- faster download.

See Nginx X-accel documentation [5] for details.

Known limitations

- Nginx needs access to the resource by URL (proxy) or path (location).
- Thus `VirtualFile` and any generated files cannot be streamed by Nginx.

[4] https://github.com/benoitbryon/django-downloadview/issues/70
[5] http://wiki.nginx.org/X-accel

Given a view

Let's consider the following view:

```
import os

from django.conf import settings
from django.core.files.storage import FileSystemStorage

from django_downloadview import StorageDownloadView


storage_dir = os.path.join(settings.MEDIA_ROOT, 'nginx')
storage = FileSystemStorage(location=storage_dir,
                            base_url=''.join([settings.MEDIA_URL, 'nginx/']))


optimized_by_middleware = StorageDownloadView.as_view(storage=storage,
                                                      path='hello-world.txt')
```

What is important here is that the files will have an `url` property implemented by storage. Let's setup an optimization rule based on that URL.

Note: It is generally easier to setup rules based on URL rather than based on name in filesystem. This is because path is generally relative to storage, whereas URL usually contains some storage identifier, i.e. it is easier to target a specific location by URL rather than by filesystem name.

Setup XAccelRedirect middlewares

Make sure `django_downloadview.SmartDownloadMiddleware` is in `MIDDLEWARE_CLASSES` of your *Django* settings.

Example:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django_downloadview.SmartDownloadMiddleware'
]
```

Then set `django_downloadview.nginx.XAccelRedirectMiddleware` as `DOWNLOADVIEW_BACKEND`:

```
DOWNLOADVIEW_BACKEND = 'django_downloadview.nginx.XAccelRedirectMiddleware'
```

Then register as many `DOWNLOADVIEW_RULES` as you wish:

```
DOWNLOADVIEW_RULES = [
    {
        'source_url': '/media/nginx/',
        'destination_url': '/nginx-optimized-by-middleware/',
    },
]
```

Each item in `DOWNLOADVIEW_RULES` is a dictionary of keyword arguments passed to the middleware factory. In the example above, we capture responses by `source_url` and convert them to internal redirects to `destination_url`.

class `django_downloadview.nginx.middlewares.`**`XAccelRedirectMiddleware`**(*source_dir=None*, *source_url=None*, *destination_url=None*, *expires=None*, *with_buffering=None*, *limit_rate=None*, *media_root=None*, *media_url=None*)

Bases: `django_downloadview.middlewares.ProxiedDownloadMiddleware`

Configurable middleware, for use in decorators or in global middlewares.

Standard Django middlewares are configured globally via settings. Instances of this class are to be configured individually. It makes it possible to use this class as the factory in `django_downloadview.decorators.DownloadDecorator`.

`get_redirect_url`(*response*)
Return redirect URL for file wrapped into response.

`is_download_response`(*response*)
Return True for DownloadResponse, except for "virtual" files.

This implementation cannot handle files that live in memory or which are to be dynamically iterated over. So, we capture only responses whose file attribute have either an URL or a file name.

`process_response`(*request*, *response*)
Call *process_download_response()* if `response` is download.

`process_download_response`(*request*, *response*)
Replace DownloadResponse instances by NginxDownloadResponse ones.

Per-view setup with x_accel_redirect decorator

Middlewares should be enough for most use cases, but you may want per-view configuration. For *nginx*, there is `x_accel_redirect`:

`django_downloadview.nginx.decorators.`**`x_accel_redirect`**(*view_func*, **args*, ***kwargs*)
Apply `XAccelRedirectMiddleware` to `view_func`.

Proxies (`*args`, `**kwargs`) to middleware constructor.

As an example:

```
import os

from django.conf import settings
from django.core.files.storage import FileSystemStorage

from django_downloadview import StorageDownloadView
from django_downloadview.nginx import x_accel_redirect
```

```
optimized_by_decorator = x_accel_redirect(
    StorageDownloadView.as_view(storage=storage, path='hello-world.txt'),
    source_url=storage.base_url,
    destination_url='/nginx-optimized-by-decorator/')
```

Test responses with assert_x_accel_redirect

Use `assert_x_accel_redirect()` function as a shortcut in your tests.

```
import os

from django.core.files.base import ContentFile
from django.core.urlresolvers import reverse
import django.test

from django_downloadview.nginx import assert_x_accel_redirect

from demoproject.nginx.views import storage, storage_dir


def setup_file():
    if not os.path.exists(storage_dir):
        os.makedirs(storage_dir)
    storage.save('hello-world.txt', ContentFile(u'Hello world!\n'))


class OptimizedByMiddlewareTestCase(django.test.TestCase):
    def test_response(self):
        """'nginx:optimized_by_middleware' returns X-Accel response."""
        setup_file()
        url = reverse('nginx:optimized_by_middleware')
        response = self.client.get(url)
        assert_x_accel_redirect(
            self,
            response,
            content_type="text/plain; charset=utf-8",
            charset="utf-8",
            basename="hello-world.txt",
            redirect_url="/nginx-optimized-by-middleware/hello-world.txt",
            expires=None,
            with_buffering=None,
            limit_rate=None)


class OptimizedByDecoratorTestCase(django.test.TestCase):
    def test_response(self):
        """'nginx:optimized_by_decorator' returns X-Accel response."""
        setup_file()
        url = reverse('nginx:optimized_by_decorator')
        response = self.client.get(url)
        assert_x_accel_redirect(
            self,
            response,
            content_type="text/plain; charset=utf-8",
            charset="utf-8",
            basename="hello-world.txt",
            redirect_url="/nginx-optimized-by-decorator/hello-world.txt",
            expires=None,
            with_buffering=None,
            limit_rate=None)
```

`django_downloadview.nginx.tests.`**`assert_x_accel_redirect`**(*test_case*, *response*, ***assertions*)

Make `test_case` assert that `response` is a XAccelRedirectResponse.

Optional `assertions` dictionary can be used to check additional items:

- `basename`: the basename of the file in the response.
- `content_type`: the value of "Content-Type" header.
- `redirect_url`: the value of "X-Accel-Redirect" header.
- `charset`: the value of `X-Accel-Charset` header.
- `with_buffering`: the value of `X-Accel-Buffering` header. If `False`, then makes sure that the header disables buffering. If `None`, then makes sure that the header is not set.
- `expires`: the value of `X-Accel-Expires` header. If `False`, then makes sure that the header disables expiration. If `None`, then makes sure that the header is not set.
- `limit_rate`: the value of `X-Accel-Limit-Rate` header. If `False`, then makes sure that the header disables limit rate. If `None`, then makes sure that the header is not set.

The tests above assert the *Django* part is OK. Now let's configure *nginx*.

Setup Nginx

See Nginx X-accel documentation [1] for details.

Here is what you could have in `/etc/nginx/sites-available/default`:

```
charset utf-8;

# Django-powered service.
upstream frontend {
    server 127.0.0.1:8000 fail_timeout=0;
}

server {
    listen 80 default;

    # File-download proxy.
    #
    # Will serve /var/www/files/myfile.tar.gz when passed URI
    # like /optimized-download/myfile.tar.gz
    #
    # See http://wiki.nginx.org/X-accel
    # and https://django-downloadview.readthedocs.org
    #
    location /proxied-download {
        internal;
        # Location to files on disk.
        alias /var/www/files/;
    }

    # Proxy to Django-powered frontend.
    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_pass http://frontend;
    }
}
```

... where specific configuration is the `location /optimized-download` section.

Note: `/proxied-download` has the `internal` flag, so this location is not available for the client, i.e. users are not able to download files via `/optimized-download/<filename>`.

Assert everything goes fine with healthchecks

Healthchecks are the best way to check the complete setup.

Common issues

`Unknown charset "utf-8" to override` Add `charset utf-8;` in your nginx configuration file.

`open() "path/to/something" failed (2: No such file or directory)` Check your `settings.NGINX_DOWNLOAD_MIDDLEWARE_MEDIA_ROOT` in Django configuration VS `alias` in nginx configuration: in a standard configuration, they should be equal.

References

Apache

If you serve Django behind Apache, then you can delegate the file streaming to Apache and get increased performance:

- lower resources used by Python/Django workers ;
- faster download.

See Apache mod_xsendfile documentation [6] for details.

Known limitations

- Apache needs access to the resource by path on local filesystem.
- Thus only files that live on local filesystem can be streamed by Apache.

Given a view

Let's consider the following view:

[6] https://tn123.org/mod_xsendfile/

```
import os

from django.conf import settings
from django.core.files.storage import FileSystemStorage

from django_downloadview import StorageDownloadView


storage_dir = os.path.join(settings.MEDIA_ROOT, 'apache')
storage = FileSystemStorage(location=storage_dir,
                            base_url=''.join([settings.MEDIA_URL, 'apache/']))


optimized_by_middleware = StorageDownloadView.as_view(storage=storage,
                                                      path='hello-world.txt')
```

What is important here is that the files will have an `url` property implemented by storage. Let's setup an optimization rule based on that URL.

Note: It is generally easier to setup rules based on URL rather than based on name in filesystem. This is because path is generally relative to storage, whereas URL usually contains some storage identifier, i.e. it is easier to target a specific location by URL rather than by filesystem name.

Setup XSendfile middlewares

Make sure `django_downloadview.SmartDownloadMiddleware` is in `MIDDLEWARE_CLASSES` of your *Django* settings.

Example:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django_downloadview.SmartDownloadMiddleware'
]
```

Then set `django_downloadview.apache.XSendfileMiddleware` as `DOWNLOADVIEW_BACKEND`:

```
DOWNLOADVIEW_BACKEND = 'django_downloadview.apache.XSendfileMiddleware'
```

Then register as many `DOWNLOADVIEW_RULES` as you wish:

```
DOWNLOADVIEW_RULES = [
    {
        'source_url': '/media/apache/',
        'destination_dir': '/apache-optimized-by-middleware/',
    },
]
```

Each item in `DOWNLOADVIEW_RULES` is a dictionary of keyword arguments passed to the middleware factory. In the example above, we capture responses by `source_url` and convert them to internal redirects to `destination_dir`.

class `django_downloadview.apache.middlewares.`**`XSendfileMiddleware`**(*source_dir=None*, *source_url=None*, *destination_dir=None*)

Bases: `django_downloadview.middlewares.ProxiedDownloadMiddleware`

Configurable middleware, for use in decorators or in global middlewares.

Standard Django middlewares are configured globally via settings. Instances of this class are to be configured individually. It makes it possible to use this class as the factory in `django_downloadview.decorators.DownloadDecorator`.

`process_download_response`(*request*, *response*)
Replace DownloadResponse instances by XSendfileResponse ones.

`get_redirect_url`(*response*)
Return redirect URL for file wrapped into response.

`is_download_response`(*response*)
Return True for DownloadResponse, except for “virtual” files.

This implementation cannot handle files that live in memory or which are to be dynamically iterated over. So, we capture only responses whose file attribute have either an URL or a file name.

`process_response`(*request*, *response*)
Call *process_download_response()* if `response` is download.

Per-view setup with x_sendfile decorator

Middlewares should be enough for most use cases, but you may want per-view configuration. For *Apache*, there is `x_sendfile`:

`django_downloadview.apache.decorators.`**`x_sendfile`**(*view_func*, **args*, ***kwargs*)
Apply `XSendfileMiddleware` to `view_func`.

Proxies (`*args`, `**kwargs`) to middleware constructor.

As an example:

```python
import os

from django.conf import settings
from django.core.files.storage import FileSystemStorage

from django_downloadview import StorageDownloadView
from django_downloadview.apache import x_sendfile


optimized_by_decorator = x_sendfile(
    StorageDownloadView.as_view(storage=storage, path='hello-world.txt'),
    source_url=storage.base_url,
    destination_dir='/apache-optimized-by-decorator/')
```

Test responses with assert_x_sendfile

Use `assert_x_sendfile()` function as a shortcut in your tests.

```python
import os

from django.core.files.base import ContentFile
from django.core.urlresolvers import reverse
import django.test

from django_downloadview.apache import assert_x_sendfile

from demoproject.apache.views import storage, storage_dir


def setup_file():
    if not os.path.exists(storage_dir):
        os.makedirs(storage_dir)
    storage.save('hello-world.txt', ContentFile(u'Hello world!\n'))


class OptimizedByMiddlewareTestCase(django.test.TestCase):
    def test_response(self):
        """'apache:optimized_by_middleware' returns X-Sendfile response."""
        setup_file()
        url = reverse('apache:optimized_by_middleware')
        response = self.client.get(url)
        assert_x_sendfile(
            self,
            response,
            content_type="text/plain; charset=utf-8",
            basename="hello-world.txt",
            file_path="/apache-optimized-by-middleware/hello-world.txt")


class OptimizedByDecoratorTestCase(django.test.TestCase):
    def test_response(self):
        """'apache:optimized_by_decorator' returns X-Sendfile response."""
        setup_file()
        url = reverse('apache:optimized_by_decorator')
        response = self.client.get(url)
        assert_x_sendfile(
            self,
            response,
            content_type="text/plain; charset=utf-8",
            basename="hello-world.txt",
            file_path="/apache-optimized-by-decorator/hello-world.txt")
```

`django_downloadview.apache.tests.`**`assert_x_sendfile`**(*test_case*, *response*, ***assertions*)

Make `test_case` assert that `response` is a XSendfileResponse.

Optional `assertions` dictionary can be used to check additional items:

- `basename`: the basename of the file in the response.
- `content_type`: the value of “Content-Type” header.
- `file_path`: the value of “X-Sendfile” header.

The tests above assert the *Django* part is OK. Now let’s configure *Apache*.

Setup Apache

See Apache mod_xsendfile documentation [1] for details.

Assert everything goes fine with healthchecks

Healthchecks are the best way to check the complete setup.

References

Lighttpd

If you serve Django behind *Lighttpd*, then you can delegate the file streaming to *Lighttpd* and get increased performance:

- lower resources used by Python/Django workers ;
- faster download.

See Lighttpd X-Sendfile documentation [7] for details.

Note: Currently, *django_downloadview* supports `X-Sendfile`, but not `X-Sendfile2`. If you need `X-Sendfile2` or know how to handle it, check X-Sendfile2 feature request on django_downloadview's bugtracker [8].

Known limitations

- Lighttpd needs access to the resource by path on local filesystem.
- Thus only files that live on local filesystem can be streamed by Lighttpd.

Given a view

Let's consider the following view:

```python
import os

from django.conf import settings
from django.core.files.storage import FileSystemStorage

from django_downloadview import StorageDownloadView


storage_dir = os.path.join(settings.MEDIA_ROOT, 'lighttpd')
storage = FileSystemStorage(location=storage_dir,
                            base_url=''.join([settings.MEDIA_URL, 'lighttpd/']))


optimized_by_middleware = StorageDownloadView.as_view(storage=storage,
                                                      path='hello-world.txt')
```

[7] http://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file
[8] https://github.com/benoitbryon/django-downloadview/issues/67

What is important here is that the files will have an `url` property implemented by storage. Let's setup an optimization rule based on that URL.

Note: It is generally easier to setup rules based on URL rather than based on name in filesystem. This is because path is generally relative to storage, whereas URL usually contains some storage identifier, i.e. it is easier to target a specific location by URL rather than by filesystem name.

Setup XSendfile middlewares

Make sure `django_downloadview.SmartDownloadMiddleware` is in `MIDDLEWARE_CLASSES` of your *Django* settings.

Example:

```
MIDDLEWARE_CLASSES = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django_downloadview.SmartDownloadMiddleware'
]
```

Then set `django_downloadview.lighttpd.XSendfileMiddleware` as `DOWNLOADVIEW_BACKEND`:

```
DOWNLOADVIEW_BACKEND = 'django_downloadview.lighttpd.XSendfileMiddleware'
```

Then register as many `DOWNLOADVIEW_RULES` as you wish:

```
DOWNLOADVIEW_RULES = [
    {
        'source_url': '/media/lighttpd/',
        'destination_dir': '/lighttpd-optimized-by-middleware/',
    },
]
```

Each item in `DOWNLOADVIEW_RULES` is a dictionary of keyword arguments passed to the middleware factory. In the example above, we capture responses by `source_url` and convert them to internal redirects to `destination_dir`.

class `django_downloadview.lighttpd.middlewares.`**`XSendfileMiddleware`**(*source_dir=None*, *source_url=None*, *destination_dir=None*)

Bases: `django_downloadview.middlewares.ProxiedDownloadMiddleware`

Configurable middleware, for use in decorators or in global middlewares.

Standard Django middlewares are configured globally via settings. Instances of this class are to be configured individually. It makes it possible to use this class as the factory in `django_downloadview.decorators.DownloadDecorator`.

`process_download_response`(*request*, *response*)
Replace DownloadResponse instances by XSendfileResponse ones.

`get_redirect_url`(*response*)
Return redirect URL for file wrapped into response.

is_download_response(*response*)
Return True for DownloadResponse, except for "virtual" files.

This implementation cannot handle files that live in memory or which are to be dynamically iterated over. So, we capture only responses whose file attribute have either an URL or a file name.

process_response(*request*, *response*)
Call *process_download_response()* if `response` is download.

Per-view setup with x_sendfile decorator

Middlewares should be enough for most use cases, but you may want per-view configuration. For *Lighttpd*, there is `x_sendfile`:

`django_downloadview.lighttpd.decorators.`**x_sendfile**(*view_func*, **args*, ***kwargs*)
Apply `XSendfileMiddleware` to `view_func`.

Proxies (`*args`, `**kwargs`) to middleware constructor.

As an example:

```
import os

from django.conf import settings
from django.core.files.storage import FileSystemStorage

from django_downloadview import StorageDownloadView
from django_downloadview.lighttpd import x_sendfile


optimized_by_decorator = x_sendfile(
    StorageDownloadView.as_view(storage=storage, path='hello-world.txt'),
    source_url=storage.base_url,
    destination_dir='/lighttpd-optimized-by-decorator/')
```

Test responses with assert_x_sendfile

Use `assert_x_sendfile()` function as a shortcut in your tests.

```
import os

from django.core.files.base import ContentFile
from django.core.urlresolvers import reverse
import django.test

from django_downloadview.lighttpd import assert_x_sendfile

from demoproject.lighttpd.views import storage, storage_dir


def setup_file():
    if not os.path.exists(storage_dir):
        os.makedirs(storage_dir)
    storage.save('hello-world.txt', ContentFile(u'Hello world!\n'))


class OptimizedByMiddlewareTestCase(django.test.TestCase):
    def test_response(self):
        """'lighttpd:optimized_by_middleware' returns X-Sendfile response."""
        setup_file()
        url = reverse('lighttpd:optimized_by_middleware')
        response = self.client.get(url)
        assert_x_sendfile(
            self,
            response,
            content_type="text/plain; charset=utf-8",
            basename="hello-world.txt",
            file_path="/lighttpd-optimized-by-middleware/hello-world.txt")


class OptimizedByDecoratorTestCase(django.test.TestCase):
    def test_response(self):
        """'lighttpd:optimized_by_decorator' returns X-Sendfile response."""
        setup_file()
        url = reverse('lighttpd:optimized_by_decorator')
        response = self.client.get(url)
        assert_x_sendfile(
            self,
            response,
            content_type="text/plain; charset=utf-8",
            basename="hello-world.txt",
            file_path="/lighttpd-optimized-by-decorator/hello-world.txt")
```

`django_downloadview.lighttpd.tests.`**`assert_x_sendfile`**(*test_case*, *response*, ***assertions*)

Make `test_case` assert that `response` is a XSendfileResponse.

Optional `assertions` dictionary can be used to check additional items:

- `basename`: the basename of the file in the response.
- `content_type`: the value of “Content-Type” header.
- `file_path`: the value of “X-Sendfile” header.

The tests above assert the *Django* part is OK. Now let’s configure *Lighttpd*.

Setup Lighttpd

See Lighttpd X-Sendfile documentation [1] for details.

Assert everything goes fine with healthchecks

Healthchecks are the best way to check the complete setup.

References

Note: If you need support for additional optimizations, tell us [9]!

[9] https://github.com/benoitbryon/django-downloadview/issues?labels=optimizations

Notes & references

3.6 Write tests

django_downloadview embeds test utilities:

- `temporary_media_root()`
- `assert_download_response()`
- `setup_view()`
- `assert_x_accel_redirect()`

3.6.1 temporary_media_root

`django_downloadview.test.`**`temporary_media_root`**`(`***kwargs*`)`

Temporarily override settings.MEDIA_ROOT with a temporary directory.

The temporary directory is automatically created and destroyed.

Use this function as a context manager:

```
>>> from django_downloadview.test import temporary_media_root
>>> from django.conf import settings
>>> global_media_root = settings.MEDIA_ROOT
>>> with temporary_media_root():
...     global_media_root == settings.MEDIA_ROOT
False
>>> global_media_root == settings.MEDIA_ROOT
True
```

Or as a decorator:

```
>>> @temporary_media_root()
... def use_temporary_media_root():
...     return settings.MEDIA_ROOT
>>> tmp_media_root = use_temporary_media_root()
>>> global_media_root == tmp_media_root
False
>>> global_media_root == settings.MEDIA_ROOT
True
```

3.6.2 assert_download_response

`django_downloadview.test.`**`assert_download_response`**`(`*test_case*, *response*, ***assertions*`)`

Make `test_case` assert that `response` meets `assertions`.

Optional `assertions` dictionary can be used to check additional items:

- `basename`: the basename of the file in the response.
- `content_type`: the value of "Content-Type" header.
- `mime_type`: the MIME type part of "Content-Type" header (without charset).
- `content`: the contents of the file.
- `attachment`: whether the file is returned as attachment or not.

Examples, related to *StorageDownloadView demo*:

```
from django.core.files.base import ContentFile
from django.core.urlresolvers import reverse
import django.test

from django_downloadview import assert_download_response, temporary_media_root

from demoproject.storage import views


# Fixtures.
file_content = 'Hello world!\n'


def setup_file(path):
    views.storage.save(path, ContentFile(file_content))


class StaticPathTestCase(django.test.TestCase):
    @temporary_media_root()
    def test_download_response(self):
        """'storage:static_path' streams file by path."""
        setup_file('1.txt')
        url = reverse('storage:static_path', kwargs={'path': '1.txt'})
        response = self.client.get(url)
        assert_download_response(self,
                                 response,
                                 content=file_content,
                                 basename='1.txt',
                                 mime_type='text/plain')


class DynamicPathIntegrationTestCase(django.test.TestCase):
    """Integration tests around ``storage:dynamic_path`` URL."""
    @temporary_media_root()
    def test_download_response(self):
        """'dynamic_path' streams file by generated path.

        As we use ``self.client``, this test involves the whole Django stack,
        including settings, middlewares, decorators... So we need to setup a
        file, the storage, and an URL.

        This test actually asserts the URL ``storage:dynamic_path`` streams a
        file in storage.

        """
        setup_file('1.TXT')
        url = reverse('storage:dynamic_path', kwargs={'path': '1.txt'})
        response = self.client.get(url)
        assert_download_response(self,
                                 response,
                                 content=file_content,
                                 basename='1.TXT',
                                 mime_type='text/plain')
```

3.6.3 setup_view

`django_downloadview.test.`**`setup_view`**(*view*, *request*, **args*, ***kwargs*)

Mimic `as_view()`, but returns view instance.

Use this function to get view instances on which you can run unit tests, by testing specific methods.

This is an early implementation of https://code.djangoproject.com/ticket/20456

view A view instance, such as `TemplateView(template_name='dummy.html')`. Initialization arguments are the same you would pass to `as_view()`.

request A request object, typically built with `RequestFactory`.

args and kwargs "URLconf" positional and keyword arguments, the same you would pass to `reverse()`.

Example, related to *StorageDownloadView demo*:

```
import unittest

from django_downloadview import setup_view

from demoproject.storage import views


class DynamicPathUnitTestCase(unittest.TestCase):
    """Unit tests around ``views.DynamicStorageDownloadView``."""
    def test_get_path(self):
        """DynamicStorageDownloadView.get_path() returns uppercase path.

        Uses :func:`~django_downloadview.test.setup_view` to target only
        overriden methods.

        This test does not involve URLconf, middlewares or decorators. It is
        fast. It has clear scope. It does not assert ``storage:dynamic_path``
        URL works. It targets only custom ``DynamicStorageDownloadView`` class.

        """
        view = setup_view(views.DynamicStorageDownloadView(),
                          django.test.RequestFactory().get('/fake-url'),
                          path='dummy path')
        path = view.get_path()
        self.assertEqual(path, 'DUMMY PATH')
```

3.7 Write healthchecks

In the previous *testing* topic, you made sure the views and middlewares work as expected... within a test environment.

One common issue when deploying in production is that the reverse-proxy's configuration does not fit. You cannot check that within test environment.

Healthchecks are made to diagnose issues in live (production) environments.

3.7.1 Introducing healthchecks

Healthchecks (sometimes called "smoke tests" or "diagnosis") are assertions you run on a live (typically production) service, as opposed to fake/mock service used during tests (unit, integration, functional).

See hospital [10] and django-doctor [11] projects about writing healthchecks for Python and Django.

3.7.2 Typical healthchecks

Here is a typical healthcheck setup for download views with reverse-proxy optimizations.

When you run this healthcheck suite, you get a good overview if a problem occurs: you can compare expected results and learn which part (Django, reverse-proxy or remote storage) is guilty.

Note: In the examples below, we use "localhost" and ports "80" (reverse-proxy) or "8000" (Django). Adapt them to your configuration.

Check storage

Put a dummy file on the storage Django uses.

The write a healthcheck that asserts you can read the dummy file from storage.

On success, you know remote storage is ok.

Issues may involve permissions or communications (remote storage).

Note: This healthcheck may be outside Django.

Check Django VS storage

Implement a download view dedicated to healthchecks. It is typically a public (but not referenced) view that streams a dummy file from real storage. Let's say you register it as `/healthcheck-utils/download/` URL.

Write a healthcheck that asserts `GET http://localhost:8000/healtcheck-utils/download/` (notice the *8000* port: local Django server) returns the expected reverse-proxy response (X-Accel, X-Sendfile...).

On success, you know there is no configuration issue on the Django side.

Check reverse proxy VS storage

Write a location in your reverse-proxy's configuration that proxy-pass to a dummy file on storage.

Write a healthcheck that asserts this location returns the expected dummy file.

On success, you know the reverse proxy can serve files from storage.

Check them all together

We just checked all parts separately, so let's make sure they can work together. Configure the reverse-proxy so that */healthcheck-utils/download/* is proxied to Django. Then write a healthcheck that asserts `GET http://localhost:80/healthcheck-utils/download` (notice the *80* port: reverse-proxy server) returns the expected dummy file.

On success, you know everything is ok.

[10] https://pypi.python.org/pypi/hospital
[11] https://pypi.python.org/pypi/django-doctor

On failure, there is an issue in the X-Accel/X-Sendfile configuration.

Note: This last healthcheck should be the first one to run, i.e. if it passes, others should pass too. The others are useful when this one fails.

Notes & references

3.8 File wrappers

A view return `DownloadResponse` which itself carries a file wrapper. Here are file wrappers distributed by Django and django-downloadview.

3.8.1 Django's builtins

Django itself provides some file wrappers [12] you can use within `django-downloadview`:

- `django.core.files.File` wraps a file that live on local filesystem, initialized with a path. `django-downloadview` uses this wrapper in *PathDownloadView*.
- `django.db.models.fields.files.FieldFile` wraps a file that is managed in a model. `django-downloadview` uses this wrapper in *ObjectDownloadView*.
- `django.core.files.base.ContentFile` wraps a bytes, string or unicode object. You may use it with *VirtualDownloadView*.

3.8.2 django-downloadview builtins

`django-downloadview` implements additional file wrappers:

- `StorageFile` wraps a file that is managed via a storage (but not necessarily via a model). *StorageDownloadView* uses this wrapper.
- `HTTPFile` wraps a file that lives at some (remote) location, initialized with an URL. *HTTPDownloadView* uses this wrapper.
- `VirtualFile` wraps a file that lives in memory, i.e. built as a string. This is a convenient wrapper to use in *VirtualDownloadView* subclasses.

3.8.3 API reference

StorageFile

class `django_downloadview.files.`**`StorageFile`**(*storage*, *name*, *file=None*)

Bases: `django.core.files.base.File`

A file in a Django storage.

This class looks like `django.db.models.fields.files.FieldFile`, but unrelated to model instance.

[12] https://docs.djangoproject.com/en/1.5/ref/files/file/

file
Required by django.core.files.utils.FileProxy.

open (*mode='rb'*)
Retrieves the specified file from storage and return open() result.

Proxy to self.storage.open(self.name, mode).

save (*content*)
Saves new content to the file.

Proxy to self.storage.save(self.name).

The content should be a proper File object, ready to be read from the beginning.

path
Return a local filesystem path which is suitable for open().

Proxy to self.storage.path(self.name).

May raise NotImplementedError if storage doesn't support file access with Python's built-in open() function

delete ()
Delete the specified file from the storage system.

Proxy to self.storage.delete(self.name).

exists ()
Return True if file already exists in the storage system.

If False, then the name is available for a new file.

size
Return the total size, in bytes, of the file.

Proxy to self.storage.size(self.name).

url
Return an absolute URL where the file's contents can be accessed.

Proxy to self.storage.url(self.name).

accessed_time
Return the last accessed time (as datetime object) of the file.

Proxy to self.storage.accessed_time(self.name).

created_time
Return the creation time (as datetime object) of the file.

Proxy to self.storage.created_time(self.name).

modified_time
Return the last modification time (as datetime object) of the file.

Proxy to self.storage.modified_time(self.name).

HTTPFile

class `django_downloadview.files.`**HTTPFile** (*request_factory=<function get at 0x2c44758>, url='', name=u'', **kwargs*)
Bases: `django.core.files.base.File`

Wrapper for files that live on remote HTTP servers.

Acts as a proxy.

Uses https://pypi.python.org/pypi/requests.

Always sets "stream=True" in requests kwargs.

request

file

size

Return the total size, in bytes, of the file.

Reads response's "content-length" header.

VirtualFile

class `django_downloadview.files.`**`VirtualFile`**(*file=None*, *name=u''*, *url=''*, *size=None*)

Bases: `django.core.files.base.File`

Wrapper for files that live in memory.

size

Notes & references

3.9 Responses

Views return `DownloadResponse`.

Middlewares (and decorators) are given the opportunity to capture responses and convert them to `ProxiedDownloadResponse`.

3.9.1 DownloadResponse

class `django_downloadview.response.`**`DownloadResponse`**(*file_instance*, *attachment=True*, *basename=None*, *status=200*, *content_type=None*, *file_mimetype=None*, *file_encoding=None*)

Bases: `django.http.response.StreamingHttpResponse`

File download response (Django serves file, client downloads it).

This is a specialization of `django.http.StreamingHttpResponse` where `streaming_content` is a file wrapper.

Constructor differs a bit from `HttpResponse`:

`file_instance` A *file wrapper instance*, such as `File`.

`attachement` Boolean. Whether to return the file as attachment or not. Affects `Content-Disposition` header.

`basename` Unicode. Client-side name of the file to stream. Only used if `attachment` is `True`. Affects `Content-Disposition` header.

`status` HTTP status code.

content_type Value for Content-Type header. If None, then mime-type and encoding will be populated by the response (default implementation uses mimetypes, based on file name).

file_mimetype Value for file's mimetype. If None (the default), then the file's mimetype will be guessed via Python's mimetypes. See get_mime_type().

file_encoding Value for file's encoding. If None (the default), then the file's encoding will be guessed via Python's mimetypes. See get_encoding().

Here are some highlights to understand internal mechanisms and motivations:

- Let's start by quoting **PEP 3333** (WSGI specification):

  For large files, or for specialized uses of HTTP streaming, applications will usually return an iterator (often a generator-iterator) that produces the output in a block-by-block fashion.

- Django WSGI handler (application implementation) return response object.
- django.http.HttpResponse and subclasses are iterators.
- In StreamingHttpResponse, the __iter__() implementation proxies to streaming_content.
- In DownloadResponse and subclasses, streaming_content is a *file wrapper*. File wrapper is itself an iterator over actual file content, and it also encapsulates access to file attributes (size, name, ...).

default_headers
Return dictionary of automatically-computed headers.

Uses an internal _default_headers cache. Default values are computed if only cache hasn't been set.

Content-Disposition header is encoded according to RFC 5987. See also http://stackoverflow.com/questions/93551/how-to-encode-the-filename-parameter-of-content-disposition-header-in-http.

items()
Return iterable of (header, value).

This method is called by http handlers just before WSGI's start_response() is called... but it is not called by django.test.ClientHandler! :'(

get_basename()
Return basename.

get_content_type()
Return a suitable "Content-Type" header for self.file.

get_mime_type()
Return mime-type of the file.

get_encoding()
Return encoding of the file to serve.

get_charset()
Return the charset of the file to serve.

3.9.2 ProxiedDownloadResponse

class django_downloadview.response.**ProxiedDownloadResponse**(*content=''*, **args*, ***kwargs*)

Bases: django.http.response.HttpResponse

Base class for internal redirect download responses.

This base class makes it possible to identify several types of specific responses such as `XAccelRedirectResponse`.

3.10 Migrating from django-sendfile

django-sendfile [13] is a wrapper around web-server specific methods for sending files to web clients. See *Alternatives and related projects* for details about this project.

django-downloadview provides a `port of django-sendfile's main function`.

> **Warning:** *django-downloadview* can replace the following *django-sendfile*'s backends: `nginx`, `xsendfile`, `simple`. But it currently cannot replace `mod_wsgi` backend.

Here are tips to migrate from *django-sendfile* to *django-downloadview*...

1. In your project's and apps dependencies, replace `django-sendfile` by `django-downloadview`.
2. In your Python scripts, replace `import sendfile` and `from sendfile` by `import django_downloadview` and `from django_downloadview`. You get something like `from django_downloadview import sendfile`
3. Adapt your settings as explained in *Configure*. Pay attention to:
   - replace `sendfile` by `django_downloadview` in `INSTALLED_APPS`.
   - replace `SENDFILE_BACKEND` by `DOWNLOADVIEW_BACKEND`
   - setup `DOWNLOADVIEW_RULES`. It replaces `SENDFILE_ROOT` and can do more.
   - register `django_downloadview.SmartDownloadMiddleware` in `MIDDLEWARE_CLASSES`.
4. Change your tests if any. You can no longer use *django-senfile*'s `development` backend. See *Write tests* for *django-downloadview*'s toolkit.
5. Here you are! ... or please report your story/bug at django-downloadview's bugtracker [14] ;)

3.10.1 API reference

`django_downloadview.shortcuts.`**`sendfile`**`(`*request*, *filename*, *attachment=False*, *attachment_filename=None*, *mimetype=None*, *encoding=None*`)`

Port of django-sendfile's API in django-downloadview.

Instantiates a `PathDownloadView` to stream the file by `filename`.

References

3.11 Demo project

Demo folder in project's repository [15] contains a Django project to illustrate *django-downloadview* usage.

[13] http://pypi.python.org/pypi/django-sendfile
[14] https://github.com/benoitbryon/django-downloadview/issues
[15] https://github.com/benoitbryon/django-downloadview/tree/master/demo/demoproject/

3.11.1 Documentation includes code from the demo

Almost every example in the documentation comes from the demo:

- discover examples in the documentation;
- browse related code and tests in demo project.

Examples in documentation are tested via demo project!

3.11.2 Browse demo code online

See demo folder in project's repository [1].

3.11.3 Deploy the demo

System requirements:

- Python [16] version 2.7, available as `python` command.

  Note: You may use Virtualenv [17] to make sure the active `python` is the right one.

- `make` and `wget` to use the provided `Makefile`.

Execute:

```
git clone git@github.com:benoitbryon/django-downloadview.git
cd django-downloadview/
make runserver
```

It installs and runs the demo server on localhost, port 8000. So have a look at http://localhost:8000/

Note: If you cannot execute the Makefile, read it and adapt the few commands it contains to your needs.

Browse and use `demo/demoproject/` as a sandbox.

3.11.4 References

3.12 About django-downloadview

3.12.1 Vision

django-downloadview tries to simplify the development of "download" views using Django [18] framework. It provides generic views that cover most common patterns.

Django is not the best solution to serve files: reverse proxies are far more efficient. *django-downloadview* makes it easy to implement this best-practice.

Tests matter: *django-downloadview* provides tools to test download views and optimizations.

[16] http://python.org
[17] http://virtualenv.org
[18] https://django-project.com

Notes & references

See Also:

- *Alternatives and related projects*
- roadmap

3.12.2 Alternatives and related projects

This document presents other projects that provide similar or complementary functionalities. It focuses on differences with django-downloadview.

There is a comparison grid on djangopackages.com: https://www.djangopackages.com/grids/g/file-streaming/.

Here are additional highlights...

Django's static file view

django.contrib.staticfiles provides a view to serve files [19]. It is simple and quite naive by design: it is meant for development, not for production. See Django ticket #2131 [20]: advanced file streaming is left to third-party applications.

django-downloadview is such a third-party application.

django-sendfile

django-sendfile [21] is a wrapper around web-server specific methods for sending files to web clients.

Note: `django_downloadview.shortcuts.sendfile()` is a port of *django-sendfile*'s main function. See *Migrating from django-sendfile* for details.

`django-senfile`'s main focus is simplicity: API is made of a single `sendfile()` function you call inside your views:

```
from sendfile import sendfile

def hello_world(request):
    """Send 'hello-world.pdf' file as a response."""
    return sendfile(request, '/path/to/hello-world.pdf')
```

The download response type depends on the chosen backend, which could be Django, Lighttpd's X-Sendfile, Nginx's X-Accel... depending your settings:

```
SENDFILE_BACKEND = 'sendfile.backends.nginx'  # sendfile() will return
                                              # X-Accel responses.
# Additional settings for sendfile's nginx backend.
SENDFILE_ROOT = '/path/to'
SENDFILE_URL = '/proxied-download'
```

Here are main differences between the two projects:

[19] https://docs.djangoproject.com/en/1.6/ref/contrib/staticfiles/#static-file-development-view
[20] https://code.djangoproject.com/ticket/2131
[21] http://pypi.python.org/pypi/django-sendfile

- `django-sendfile` supports only files that live on local filesystem (i.e. where `os.path.exists` returns `True`). Whereas `django-downloadview` allows you to serve or proxy files stored in various locations, including remote ones.
- `django-sendfile` uses a single global configuration (i.e. `settings.SENDFILE_ROOT`), thus optimizations are limited to a single root folder. Whereas `django-downloadview`'s `DownloadDispatcherMiddleware` supports multiple configurations.

References

3.12.3 License

3.12.4 Authors & contributors

Maintainer: Benoît Bryon <benoit@marmelune.net>

Original code by Novapost team:

- Nicolas Tobo <https://github.com/nicolastobo>
- Lauréline Guérin <https://github.com/zebuline>
- Gregory Tappero <https://github.com/coulix>
- Rémy Hubscher <remy.hubscher@novapost.fr>
- Benoît Bryon <benoit@marmelune.net>

Developers: https://github.com/benoitbryon/django-downloadview/graphs/contributors