django-filetransfers - File upload/download abstraction

With django-filetransfers you can write reusable Django apps that handle uploads and downloads in an abstract way. Django's own file upload and storage API alone is too limited because (1) it doesn't provide a mechanism for file downloads and (2) it can only handle direct uploads which eat a lot of resources and aren't compatible with cloud services like the App Engine Blobstore or asynchronous Amazon S3 uploads (where the file isn't piped through Django, but sent directly to S3). This is where django-filetransfers comes in. You can continue to use Django's `FileField` and `ModelForm` in your apps. You just need to add a few very simple API calls to your file handling views and templates and select a django-filetransfers backend via your settings.py. With this you can transparently support cloud services for file hosting or even the X-Sendfile mechanism.

Documentation Source Download Sample code Demo

Documentation

Installation

You can install the package via `setup.py install` or by copying or linking the "filetransfers" folder to your project (App Engine developers have to use the copy/link method). Then, add "filetransfers" to your `INSTALLED_APPS`.

Note for App Engine users: All nrequired backends are already enabled in the default settings. You don't need any special configuration. In order to use the Blobstore on the App Engine production server you have to enable billing. Otherwise, the Blobstore API is disabled.

Model and form

In the following we'll use this model and form:

```
class UploadModel(models.Model):
    file = models.FileField(upload_to='uploads/%Y/%m/%d/%H/%M/%S/')

class UploadForm(forms.ModelForm):
    class Meta:
        model = UploadModel
```

The `upload_to` parameter for `FileField` defines the target folder for file uploads (here, we add the date).

Note for App Engine users: When accessing a file object from `UploadedModel` you can get the file's `BlobInfo` object via `uploadedmodel.file.blobstore_info`. Use this to e.g. convert uploaded images via the Image API.

Handling uploads

File uploads are handled with the `prepare_upload()` function which takes the request and the URL of the upload view and returns a tuple with a generated upload URL and extra POST data for the upload. The extra POST data is just a `dict`, so you can pass it to your JavaScript code if needed. This is an example upload view:

```
from filetransfers.api import prepare_upload

def upload_handler(request):
    view_url = reverse('upload.views.upload_handler')
    if request.method == 'POST':
        form = UploadForm(request.POST, request.FILES)
        form.save()
        return HttpResponseRedirect(view_url)

    upload_url, upload_data = prepare_upload(request, view_url)
    form = UploadForm()
    return direct_to_template(request, 'upload/upload.html',
        {'form': form, 'upload_url': upload_url, 'upload_data': upload_data})
```

Note that it's important that you send a redirect after an upload. Otherwise, some file hosting services won't work correctly.

Now, you have to use the generated upload URL and the upload's extra POST data in the template:

```
{% load filetransfers %}
<form action="{{ upload_url }}" method="POST" enctype="multipart/form-data">
  {% csrf_token %}
  {% render_upload_data upload_data %}
  <table>{{ form }}</table>
  <input type="submit" value="Upload" />
</form>
```

Here we use the `{% render_upload_data %}` tag which generates `<input type="hidden" />` fields for the extra POST data.

Security and permissions

By default, uploads are assumed to have a publicly accessible URL if that's supported by the backend. You can tell the backend to mark the upload as private via `prepare_upload(..., private=True)`. If the backend has no control over the permissions (e.g., because it's your task to configure the web server correctly and not make private files publicly accessible) the `private=True` argument might just be ignored.

Asynchronous backends (like async S3 or even Blobstore) have to take special care of preventing faked uploads. After a successful upload to the actual server these backends have to generate a separate request which contains the POST data and a file ID identifying the uploaded file (the Blobstore automatically sends the blob key and async S3 would send the file and bucket name). The problem here is that a user can manually generate a request which matches the ID of some other user's private file, thus getting access to that file because it's now fake-uploaded to his private files, too. In order to prevent this asynchronous backends have to guarantee that no file ID is used twice for an upload.

Handling downloads

Since the actual download permissions can be out of the backend's control the download solution consists of two layers.

The `serve_file()` function primarily takes care of private file downloads, but in some configurations it might also have to take care of public downloads because the file hosting solution doesn't provide publicly accessible URLs (e.g., App Engine Blobstore). This means that you should also use that

function as a fallback even if you only have public downloads. The function takes two required arguments: the request and the Django `File` object that should be served (e.g. from `FileField`):

```
from filetransfers.api import serve_file

def download_handler(request, pk):
    upload = get_object_or_404(UploadModel, pk=pk)
    return serve_file(request, upload.file)
```

The `public_download_url` function, which is also available as a template filter, returns a file's publicly accessible URL if that's supported by the backend. Otherwise it returns `None`.

Important: Use `public_download_url` only for files that should be publicly accessible. Otherwise you should only use `serve_file()`, so you can check permissions before approving the download.

A complete solution for public downloads which falls back to `serve_file()` would look like this in a template for an instance of `UploadModel` called `upload`:

```
{% load filetransfers %}
{% url upload.views.download_handler pk=upload.pk as fallback_url %}
<a href="{% firstof upload.file|public_download_url fallback_url %}">Download</a>
```

The second line stores the `serve_file()` fallback URL in a variable. In the third line we then use the `public_download_url` template filter in order to get the file's publicly accessible URL. If that returns `None` the `{% firstof %}` template tag returns the second argument which is our fallback URL. Otherwise the public download URL is used.

Configuration

There are three backend types which are supported by django-filetransfers: one for uploads, one for downloads via `serve_file()`, and one for public downloads. You can specify the backends in your settings.py:

```
PREPARE_UPLOAD_BACKEND = 'filetransfers.backends.default.prepare_upload'
SERVE_FILE_BACKEND = 'filetransfers.backends.default.serve_file'
PUBLIC_DOWNLOAD_URL_BACKEND = 'filetransfers.backends.default.public_download_url'
```

The default upload backend simply returns the URL unmodified. The default download backend transfers the file in chunks via Django, so it's definitely not the most efficient mechanism, but it uses only a small amount of memory (important for large files) and requires less resources than passing a file object directly to the response. The default public downloads backend simply returns `None`. This default configuration should work with practically all servers, but it's not the most efficient solution. Please take a look at the backends which are shipped with django-filetransfers to see if something fits your solution better.

Private download backends

xsendfile.serve_file

Many web servers (at least Apache, Lighttpd, and nginx) provide an "X-Sendfile" module which allows for handing off the actual file transfer to the web server. This is much more efficient than the default download backend, so you should install the required module for your web server and then configure the xsendfile download backend in your settings.py:

```
SERVE_FILE_BACKEND = 'filetransfers.backends.xsendfile.serve_file'
```

url.serve_file

We also provide a backend which simply redirects to `file.url`. You have to make sure that `file.url` actually generates a private download URL, though. This backend should work with the Amazon S3 and similar storage backends from the [django-storages] project. Just add the following to your settings.py:

```
SERVE_FILE_BACKEND = 'filetransfers.backends.url.serve_file'
```

Public download backends

url.public_download_url

If `file.url` points to a public download URL you can use this backend:

```
PUBLIC_DOWNLOAD_URL_BACKEND = 'filetransfers.backends.url.public_download_url'
```

base_url.public_download_url

Alternatively, there's also a simple backend that merely points to a different URL. You just need to

specify a base URL and the backend appends `file.name` to that base URL.

```
PUBLIC_DOWNLOAD_URL_BACKEND = 'filetransfers.backends.base_url.public_download_url'
PUBLIC_DOWNLOADS_URL_BASE = '/downloads/'
```

Upload backends

delegate.prepare_upload

This backend delegates the upload to some other backend based on `private=True` or `private=False`. This way you can, for instance, use the App Engine Blobstore for private files and Amazon S3 for public files:

```
# Configure "delegate" backend
PREPARE_UPLOAD_BACKEND = 'filetransfers.backends.delegate.prepare_upload'
PRIVATE_PREPARE_UPLOAD_BACKEND = 'djangoappengine.storage.prepare_upload'
PUBLIC_PREPARE_UPLOAD_BACKEND = 's3backend.prepare_upload'

# Use S3 for public_download_url and Blobstore for serve_file
SERVE_FILE_BACKEND = 'djangoappengine.storage.serve_file'
PUBLIC_DOWNLOAD_URL_BACKEND = 'filetransfers.backends.base_url.public_download_url'
PUBLIC_DOWNLOADS_URL_BASE = 'http://s3.amazonaws.com/my-public-bucket/'
```

Reference: `filetransfers.api` module

prepare_upload(request, url, private=False, backend=None)

Returns a tuple with a target URL for the upload form and a `dict` with additional POST data for the upload request.

Required arguments:

- `request`: The view's request.
- `url`: The target URL where the files should be sent to.

Optional arguments:

- `private`: If `False` the backend will try to make the upload publicly accessible, so it can be served via the `public_download_url` template filter. If `True` the backend will try to make the upload non-accessible to the public, so it can only be served via `serve_file()`.
- `backend`: If defined, you can override the backend specified in settings.py.

serve_file(request, file, backend=None, save_as=False, content_type=None)

Serves a file to the browser. This is used either for checking permissions before approving a downoad or as a fallback if the backend doesn't support publicly accessible URLs. So, you always have to provide a view that uses this function.

Required arguments:

- `request`: The view's request.
- `file`: The `File` object (e.g. from `FileField`) that should be served.

Optional arguments:

- `save_as`: Forces the browser to save the file instead of displaying it (useful for PDF documents, for example). If this is True the file object's `name` attribute will be used as the file name in the download dialog. Alternatively, you can pass a string to override the file name. The default is to let the browser decide how to handle the download.
- `content_type`: Overrides the file's content type in the response. By default the content type will be detected via `mimetypes.guess_type()` using `file.name`.
- `backend`: If defined, you can override the backend specified in settings.py.

public_download_url(file, backend=None)

Tries to generate a publicly accessible URL for the given file. Returns `None` if no URL could be generated. The same function is available as a template filter.

Required arguments:

- `file`: The `File` object (e.g. from `FileField`) that should be served.

Optional arguments:

- `backend`: If defined, you can override the backend specified in settings.py.

Reference: `filetransfers` template library

{% render_upload_data upload_data %}

Renders `<input type="hidden" ... />` fields for the extra POST data (`upload_data`) as returned by `prepare_upload()`.

public_download_url
This template filter does the same as the `public_upload_url()` function in the `filetransfers.api` module: It returns a publicly accessible URL for the given file or `None` if it no such URL exists.

It takes the `File` object (e.g. from `FileField`) that should be served and optionally a second parameter to override the backend specified in settings.py.

Powered by Django-nonrel, App Engine, and All Buttons Pressed