django-drf-filepond Documentation

Jeremy Cohen

Jun 25, 2021

Contents:

CHAPTER 1

Introduction

django-drf-filepond is a Django app providing a back-end implementation for pqina's excellent filepond file upload library.

django-drf-filepond can be easily added to your Django applications to provide support for handling file uploads from a filepond client. The app includes support for filepond's `process`, `patch`, `revert`, `fetch`, `restore` and `load` endpoints allowing all the core file upload functions of the library to be used.

django-drf-filepond provides support for managing the storage of files once they have been uploaded using filepond so that they can subsequently be accessed using filepond's `load` endpoint.

Support is provided for storage on remote storage backends via integration with the django-storages library.

Support is also provided for filepond's chunked file upload, functionality.

CHAPTER 2

Installation

The app can be installed from PyPi:

```
pip install django-drf-filepond
```

or add it to your list of dependencies in a *requirements.txt* file.

2.1 Configuration

There are three required configuration updates to make within your Django application to set up django-drf-filepond. A number of additional configuration options may be specified if you're using optional features:

2.1.1 1. Add the app to INSTALLED_APPS:

Add 'django-drf-filepond' to `INSTALLED_APPS` in your Django settings file (e.g. `settings.py`):

```
...

INSTALLED_APPS = [
        ...,
        'django_drf_filepond'
]

...
```

You will need to re-run `python manage.py migrate` to update the database with the table(s) used by django-drf-filepond.

2.1.2 2. Set the temporary file upload location:

Set the location where you want django-drf-filepond to store temporary file uploads by adding the `DJANGO_DRF_FILEPOND_UPLOAD_TMP` configuration variable to your settings file, e.g.:

```
import os
...
DJANGO_DRF_FILEPOND_UPLOAD_TMP = os.path.join(BASE_DIR, 'filepond-temp-uploads')
...
```

Note: It is strongly recommended that you set `DJANGO_DRF_FILEPOND_UPLOAD_TMP`. If you do not set this variable, the app will set a default location for the storage of temporary uploads. This is `BASE_DIR/filepond_uploads` where `BASE_DIR` is the variable defined by default in an auto-generated Django settings file pointing to the top-level directory of your Django project. If your settings do not contain `BASE_DIR` the app will default to storing the `filepond_uploads` directory in the `django-drf-filepond` app directory, wherever that is located. Note that this may be within the `lib` directory of a virtualenv.

Note: If you wish to set your file upload location to an directory outside of your Django application, i.e. something that is not below BASE_DIR, you should consider any security implications that may result from letting the app write to another location on disk and set the `DJANGO_DRF_FILEPOND_ALLOW_EXTERNAL_UPLOAD_DIR` to `True` to confirm that you want to use a directory external to your application to store temporary uploads.

2.1.3 3. Include the app urls into your main url configuration

Add the URL mappings for django-drf-filepond to your URL configuration in `urls.py`:

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

urlpatterns = [
        ...
        url(r'^fp/', include('django_drf_filepond.urls')),
]
```

On the client side, you need to set the endpoints of the `process`, `revert`, `fetch`, `load` and `restore` functions to match the endpoint used in your path statement above. For example if the first parameter to `url` is `^fp/` then the endpoint for the `process` function will be `/fp/process/`. For example, your client-side configuration may include configuration similar to the following:

```
FilePond.setOptions({
    ...
    server: {
        url: 'https://<your app domain>/fp',
        process: '/process/',
        patch: '/patch/',
        revert: '/revert/',
        fetch: '/fetch/?target=',
        load: '/load/'
    }
    ...
});
```

See the filepond server configuration documentation for further examples.

2.1.4 (Optional) 4. File storage configuration

Initially, uploaded files are stored in a temporary staging area (the location you set in item 2 above, with the `DJANGO_DRF_FILEPOND_UPLOAD_TMP` parameter. At this point, an uploaded file is still shown in the filepond UI on the client and the user can choose to cancel the upload resulting in the file being deleted from temporary storage and the upload being cancelled. When a user confirms a file upload, e.g. by submitting the form in which the filepond component is embedded, any temporary uploads need to be moved to a permanent storage location.

There are three different options for file storage:

- Use a location on a local filesystem on the host server for file storage (see Section *4.1*)
- ***NEW*** Use a remote file storage backend via the django-storages library (see Section *4.2*)
- Manage file storage yourself, independently of django-drf-filepond (in this case, filepond `load` functionality is not supported)

More detailed information on handling file uploads and using the django-drf-filepond API to store them is provided in *Working with file uploads*.

4.1 Storage of filepond uploads using the local file system

To use the local filesystem for storage, you need to specify where to store files. Set the `DJANGO_DRF_FILEPOND_FILE_STORE_PATH` parameter in your Django application settings file to specify the base location where stored uploads will be placed, e.g.:

```
...
DJANGO_DRF_FILEPOND_FILE_STORE_PATH = os.path.join(BASE_DIR, 'stored_uploads')
...
```

The specified path for each stored upload will then be created relative to this location. For example, given the setting shown above, if `BASE_DIR` were `/tmp/django-drf-filepond`, then a temporary upload with the specified target location of either `/mystoredupload/uploaded_file.txt` or `mystoredupload/uploaded_file.txt` would be stored to `/tmp/django-drf-filepond/stored_uploads/mystoredupload/uploaded_file.txt`

When using local file storage, `DJANGO_DRF_FILEPOND_FILE_STORE_PATH` is the only required setting.

4.2 Remote storage of filepond uploads via django-storages

The django-storages library provides support for a number of different remote file storage backends. The django-storages documentation lists the supported backends.

To enable django-storages support for django-drf-filepond, set the `DJANGO_DRF_FILEPOND_STORAGES_BACKEND` parameter in your application configuration to the django-storages backend that you wish to use. You need to specify the fully-qualified class name for the storage backend that you want to use. This is the same value that would be used for the django-storages `DEFAULT_FILE_STORAGE` parameter and the required value can be found either by looking at the django-storages documentation for the backend that you want to use, or by looking at the code in GitHub.

For example, if you want to use the SFTP storage backend, add the following to your application settings:

```
...
DJANGO_DRF_FILEPOND_STORAGES_BACKEND = 'storages.backends.sftpstorage.SFTPStorage'
...
```

or, for the Amazon S3 backend:

```
...
DJANGO_DRF_FILEPOND_STORAGES_BACKEND = 'storages.backends.s3boto3.S3Boto3Storage'
...
```

For the Azure Storage backend, set:

```
...
DJANGO_DRF_FILEPOND_STORAGES_BACKEND = 'storages.backends.azure_storage.AzureStorage'
...
```

For the Google Cloud Storage backend, set:

```
...
DJANGO_DRF_FILEPOND_STORAGES_BACKEND = 'storages.backends.gcloud.GoogleCloudStorage'
...
```

django-storages provides support for several other storage backends including Digital Ocean and Dropbox.

For each storage backend, there are a number of additional *django-storages* configuration options that must be specified. These are detailed in the *django-storages* documentation. The specific set of parameters that you need to provide depends on your chosen storage backend configuration.

As an example, if you are using the Amazon S3 storage backend and want to store uploads into a bucket named *filepond-uploads* in the *eu-west-1* region, with the bucket and files set to be accessible only by the user specified using the access/secret key, you would provide the following set of parameters in your application's `settings.py` file:

```
DJANGO_DRF_FILEPOND_STORAGES_BACKEND = 'storages.backends.s3boto3.S3Boto3Storage'
AWS_ACCESS_KEY_ID = os.environ.get('AWS_ACCESS_KEY_ID')
AWS_SECRET_ACCESS_KEY = os.environ.get('AWS_SECRET_ACCESS_KEY')
AWS_S3_REGION_NAME = 'eu-west-1'
AWS_STORAGE_BUCKET_NAME = 'filepond-uploads'
AWS_DEFAULT_ACL = 'private'
AWS_BUCKET_ACL = 'private'
AWS_AUTO_CREATE_BUCKET = True
```

Note that the ACL for the bucket and the default ACL for files are set to private. There may well be other security-related parameters that you will want/need to set to ensure the security of the files on your chosen storage backend. The configuration here provides an example but you should read the *django-storages* docuemntation for your chosen backend and documentation for the associated storage platform to ensure that you understand the parameters that you are setting and any related potential security issues that may result from your configuration.

Note: django-storages is now included as a core dependency of django-drf-filepond. However, the different django-storages backends each have their own additional dependencies **which you need to install manually** or add to your own app's dependencies.

You can add additional dependencies using `pip` by specifying the optional *extras* feature tag, e.g. to install additional dependencies required for django-storages `boto3` support run:

```
pip install django-storages[boto3]
```

See "*Working with file uploads*" for more details on how to use the django-drf-filepond API to store files to a local or remote file store.

Note: `DJANGO_DRF_FILEPOND_FILE_STORE_PATH` is not used when using a remote file store backend. It is recommended to remove this setting or leave it set to None.

The base storage location for a remote file storage backend from django-storages is set using a setting specific to the backend that you are using - see the django-storages documentation for your chosen backend for further information.

2.1.5 Chunked uploads

`django-drf-filepond` now supports filepond chunked uploads. To use chunked uploads, you enable the functionality in your configuration of the filepond client and set the file chunk size you'd like to use. When filepond attempts to upload a file larger than the chunk size, it breaks the file up into chunks which are each uploaded in order. If the connection should fail and a chunk doesn't upload correctly, the client will retry the chunk. If the set number of retries is exceeded, the client stops attempting to retry the upload but provides the user with a retry button to manually retry the upload. `django-drf-filepond` includes all the necessary server-side functionality to support this.

There is no configuration required for `django-drf-filepond` on the server side to handle chunked uploads.

On the client side, you need to ensure that your filepond configuration specifies server endpoints for both the `process` and `patch` methods and that you have the required configuration options in place to enable chunked uploads. For example, if you want to enable `chunkUploads` and send uploads in 500,000 byte chunks, your filepond configuration should include properties similar to the following:

```
FilePond.setOptions({
    ...
    chunkUploads: true,
    chunkSize: 500000,
    server: {
        url: 'https://.../fp',
        process: '/process/',
        patch: '/patch/',
        ...
    }
    ...
});
```

2.1.6 Advanced Configuration Options

There are some optional additional configuration parameters that can be used to manage other features of the library. These are detailed in this section.

`DJANGO_DRF_FILEPOND_DELETE_UPLOAD_TMP_DIRS` (*default*: `True`):

> When a file is uploaded from a client using *filepond*, or pulled from a remote URL as a result of a call to the fetch endpoint from the filepond client, a temporary directory is created for the uploaded/fetched file to be placed into as a temporary upload. When the temporary upload is subsequently removed, either because it is cancelled or because it is moved to permanent storage, the file stored as a temporary upload is removed along with the temporary directory that it is stored in. The approach of creating a temporary directory named with a unique ID specific to the individual file being uploaded is as described in the filepond server documentation.
>
> In cases where there are large numbers of temporary uploads being created and removed, if there is a need to reduce the load on the filesystem, setting `DJANGO_DRF_FILEPOND_DELETE_UPLOAD_TMP_DIRS` to `False` will prevent the temporary directories from being removed when a temporary upload is deleted. The files within those directories will still be removed.
>
> *NOTE:* If you set `DJANGO_DRF_FILEPOND_DELETE_UPLOAD_TMP_DIRS` to `False`, you will need to have some alternative periodic "garbage collection" process in operation to remove all empty

temporary directories in order to avoid a build up of potentially very large numbers of empty directories on the filesystem.

Using a non-standard element name for your client-side filepond instance:

If you have a filepond instance on your client web page that uses an element name other than the default `filepond`, *django-drf-filepond* can now handle this. For example, if you have multiple filepond instances on a page, you will need to give each instance a different name. To take advatage of this feature, you will need to inject an additional parameter `fp_upload_field` into the HTTP upload request which provides the name of the filepond form instance to process. An example of this is shown in the issue describing the request for this feature.

2.1.7 Logging

django-drf-filepond outputs a variety of debug logging messages. You can configure logging for the app through Django's logging configuration in your Django application settings.

For example, taking a basic logging configuration such as the first example configuration in Django's logging documentation examples, adding the following to the `loggers` section of the `LOGGING` configuration dictionary will enable DEBUG output for all modules in the `django_drf_filepond` package:

```
'django_drf_filepond': {
    'handlers': ['file'],
    'level': 'DEBUG',
},
```

You can also enable logging for individual modules or set different logging levels for different modules by specifying the fully qualified module name in the configuration, for example:

```
'django_drf_filepond.views': {
    'handlers': ['file'],
    'level': 'DEBUG',
    'propagate': False,
},
'django_drf_filepond.models': {
    'handlers': ['file'],
    'level': 'INFO',
    'propagate': False,
},
```

CHAPTER 3

Using the library

3.1 Working with file uploads

When a file is uploaded from a filepond client, the file is placed into a uniquely named directory within the temporary upload directory specified by the `DJANGO_DRF_FILEPOND_UPLOAD_TMP` parameter. As per the filepond server spec, the server returns a unique identifier for the file upload. In this case, the identifier is a 22-character unique ID generated using the shortuuid library. This ID is the name used for the directory created under `DJANGO_DRF_FILEPOND_UPLOAD_TMP` into which the file is placed. At present, the file also has a separate unique identifier which hides the original name of the file on the server filesystem. The original filename is stored within the django-drf-filepond app's database. The use of a unique ID for the stored file name also allows multiple uploads with the same file name in a single upload session without causing problems with overwriting of files in the temporary upload directory.

When/if the client subsequently submits the form associated with the filepond instance that triggered the upload, the unique directory ID will be passed to the server by the client and this can be used to look up the temporary file.

There are two different approaches for handling files that need to be stored permanently on a server after being uploaded from a filepond client via django-drf-filepond. *These two approaches are not mutually exclusive and you can choose to use one approach for some files and the other approach for other files if you wish.*

3.1.1 1. Use django-drf-filepond's API to store a temporary upload to permanent storage *(recommended)*

Note: You must use this approach for storing any files that you subsequently want to access using filepond's `load` function.

Using this approach, the file is stored either to local storage or to a remote storage service depending on the file store configuration you are using.

1.1 store_upload

`store_upload` stores a temporary upload, uploaded as a result of adding it to the filepond component in a web page, to permanent storage.

If you have configured *django-drf-filepond* to use local file storage by setting the `DJANGO_DRF_FILEPOND_FILE_STORE_PATH` parameter in your application settings, the file will be stored to a location under this directory.

If you have configured a remote file store via *django-storages*, the stored upload will be sent to the configured storage backend via *django-storages*.

Parameters:

`upload_id`: The unique ID assigned to the upload by *django-drf-filepond* when the file was initially uploaded via filepond.

`destination_file_path`: The location where the file should be stored. This location will be appended to the base file storage location as defined using the *DJANGO_DRF_FILEPOND_FILE_STORE_PATH* parameter, or, for remote storage backends, the location configured using the relevant *django-storages* parameters. If you pass an absolute path beginning with `/`, the leading `/` will be removed. The path that you provide should also include the target filename.

Returns:

A `django_drf_filepond.models.StoredUpload` object representing the stored upload.

Raises `django.core.exceptions.ImproperlyConfigured` if using a local file store and *DJANGO_DRF_FILEPOND_FILE_STORE_PATH* has not been set.

Raises `ValueError` if:

- an `upload_id` is provided in an invalid format
- the `destination_file_path` is not provided
- a `django_drf_filepond.models.TemporaryUpload` record for the provided `upload_id` is not found

Example:

```
from django_drf_filepond.api import store_upload

# Given a variable upload_id containing a 22-character unique file upload ID:
su = store_upload(upload_id, destination_file_path='target_dir/filename.ext')
# destination_file_path is a relative path (including target filename.
# The path will created under the file store directory and the original
# temporary upload will be deleted.
```

1.2 get_stored_upload / get_stored_upload_file_data

Get access to a stored upload and the associated file data.

`get_stored_upload`: Given an `upload_id`, return the associated `django_drf_filepond.models.StoredUpload` object.

Throws `django_drf_filepond.models.StoredUpload.DoesNotExist` if a database record doesn't exist for the specified `upload_id`.

`get_stored_upload_file_data`: Given a StoredUpload object, return the file data for the upload as a Python file-like object.

Parameters:

`stored_upload`: A `django_drf_filepond.models.StoredUpload` object for which you want retrieve the file data.

Returns:

Returns a tuple `(filename, bytes_io)` where `filename` is a string representing the name of the stored file being returned and `bytes_io` is an `io.BytesIO` object from which the file data can be read. If an error occurs, raises an exception:

- `django_drf_filepond.exceptions.ConfigurationError`: Thrown if using a local file store and `DJANGO_DRF_FILEPOND_FILE_STORE_PATH` is not set or the specified location does not exist, or is not a directory.
- `FileNotFoundError`: Thrown if using a remote file store and the file store API reports that the file doesn't exist. If using a local file store, thrown if the file does not exist or the location is a directory and not a file.
- `IOError`: Thrown if using a local file store and reading the file fails.

Example:

```
from django_drf_filepond.api import get_stored_upload
from django_drf_filepond.api import get_stored_upload_file_data

# Given a variable upload_id containing a 22-character unique
# upload ID representing a stored upload:
su = get_store_upload(upload_id)
(filename, bytes_io) = get_store_upload_file_data(su)
file_data = bytes_io.read()
```

1.3 delete_stored_upload

`delete_stored_upload` deletes a stored upload record and, optionally, the associated file that is stored on either a local disk or a remote file storage service.

Parameters:

`upload_id`: The unique ID assigned to the upload by *django-drf-filepond* when the file was initially uploaded via filepond.

`delete_file`: `True` to delete the file associated with the record, `False` to leave the file in place.

Returns:

Returns `True` if the stored upload is deleted successfully, otherwise raises an exception:

- `django_drf_filepond.models.StoredUpload.DoesNotExist` exception if no upload exists for the specified `upload_id`.
- `django_drf_filepond.exceptions.ConfigurationError`: Thrown if using a local file store and `DJANGO_DRF_FILEPOND_FILE_STORE_PATH` is not set or the specified location does not exist, or is not a directory.
- `FileNotFoundError`: Thrown if using a remote file store and the file store API reports that the file doesn't exist. If using a local file store, thrown if the file does not exist or the location is a directory and not a file.
- `OSError`: Thrown if using a local file store and the file deletion fails.

Example:

```
from django_drf_filepond.api import delete_stored_upload

# Given a variable upload_id containing a 22-character unique
# upload ID representing a stored upload:
delete_stored_upload(upload_id, delete_file=True)
# delete_file=True will delete the file from the local
# disk or the remote storage service.
```

3.1.2 2. Manual handling of file storage

Using this approach, you move the file initially stored as a temporary upload by *django-drf-filepond* to a storage location of your choice and the file then becomes independent of *django-drf-filepond*. The following example shows how to lookup a temporary upload given its unique upload ID and move it to a permanent storage location. The temporary upload record is then deleted and *django-drf-filepond* no longer has any awareness of the file:

```
import os
from django_drf_filepond.models import TemporaryUpload

# Get the temporary upload record
tu = TemporaryUpload.objects.get(upload_id='<22-char unique ID>')

# Move the file somewhere for permanent storage
# The file will be saved with its original name
os.rename(tu.get_file_path(), '/path/to/permanent/location/%s' % tu.upload_name)

# Delete the temporary upload record and the temporary directory
tu.delete()
```

CHAPTER 4

Tutorial

This tutorial will walk you through the process of creating a basic Django application that provides server-side functionality for filepond using the django-drf-filepond app.

A simple demo web page filepond-jquery-example.html is provided for you to use as a test front-end for the demo Django application built in this tutorial. The web page uses filepond's jQuery adapter, loaded from a CDN, and is based on the Bootstrap library's starter template.

Note: This tutorial is in two parts:

Part A focuses on setting up a simple Django application to demonstrate basic use of *django-drf-filepond*. It uses local file storage (via some form of locally mounted storage on the host computer) for temporary uploads and stored files.

Part B, an advanced section at the end of the tutorial details the use of the remote file storage capabilities provided by *django-drf-filepond* via integration with the *django-storages* library.

Warning: The example given in this tutorial is purely to demonstrate how to use the *django-drf-filepond* API and to help you get started with using the library's features.

The example application built here does not address security aspects of using either local or remote file storage and you should pay particular care to this when building your own applications using *django-drf-filepond*.

For example, when using the *django-storages* S3 backend (see tutorial section B2), a number of parameters are provided to help configure security of the uploaded files and the bucket used for file storage. You should, for example, look at the various ACL options provided in the django-storages S3 documentation.

4.1 Tutorial Part A: Building a basic Django application that uses django-drf-filepond

Note: This tutorial assumes that you are using Python 3 and have virtualenv installed

The tutorial will walk you through the following steps:

1. Set up your environment - prepare an environment in which to undertake the tutorial
2. Creating the Django application - create a simple django application configured to include the django-drf-filepond app
3. Add the front-end demo web page
4. Test the service

4.1.1 A1. Set up your environment

Create a directory in which to undertake this tutorial. For example, in your home directory, create the directory `drf-filepond-tutorial`

We'll refer to this directory as `${TUTORIAL_DIR}` throughout the rest of the tutorial. If you're using a Linux or Mac OS platform with a bash shell (or a Windows-based environment that provides a bash shell such as WSL or the Git BASH shell provided with Git for Windows) you can set the environment variable `TUTORIAL_DIR` to point the tutorial directory, for example:

```
$ export TUTORIAL_DIR=$HOME/drf-filepond-tutorial
```

In `${TUTORIAL_DIR}`, create a file named `requirements.txt` containing the following content:

```
Django>=1.11
django-drf-filepond
```

Now create a *virtualenv* in `${TUTORIAL_DIR}`:

```
$ virtualenv --prompt=drf-filepond-tutorial env
$ source env/bin/activate
```

Your shell prompt should now have been modified to show `[drf-filepond-tutorial]` which shows that you're within the virtual environment.

You can now install the dependencies:

```
$ pip install -r requirements.txt
```

4.1.2 A2: Creating the Django application

In `${TUTORIAL_DIR}` with the virtualenv created in step 1 activated, use the *django-admin* command to create a new django project:

```
$ django-admin startproject drf_filepond_tutorial .
```

You should now see a `manage.py` file in your current directory as well as a `drf_filepond_tutorial` directory containing some Python source files.

As described in the Configuration section of the django-drf-filepond documentation, we'll now add the django-drf-filepond app to our Django project and then create the database to support this app and other default functionality within the Django project.

Open the file `${TUTORIAL_DIR}/drf_filepond_tutorial/settings.py` in an editor.

At the end of the `INSTALLED_APPS` section, add `'django_drf_filepond'`:

```
INSTALLED_APPS = [
        ...
    'django.contrib.staticfiles',
    'django_drf_filepond',
]
```

At the end of the file add a new configuration parameter:

```
DJANGO_DRF_FILEPOND_UPLOAD_TMP = os.path.join(BASE_DIR, 'filepond-temp-uploads')
```

Save and close the `settings.py` file.

Now open the `${TUTORIAL_DIR}/drf_filepond_tutorial/urls.py` file.

After the two existing import statements, add a new import statement:

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

There should now be three import statements at the top of the `urls.py` file.

To the `urlpatterns` list, add an additional entry to link in the filepond server URLs such that the `urlpatterns` now look as follows:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    url(r'^fp/', include('django_drf_filepond.urls')),
]
```

You can now create the database by running:

```
$ python manage.py migrate
```

4.1.3 A3. Add the front-end demo web page

We now have a very basic, but fully-configured Django project that will act as a server for filepond. In order to test this, we need a filepond client.

The filepond-jquery-example.html file in the `docs/tutorial/` directory of the django-drf-filepond GitHub repository provides a simple single-page filepond client using filepond's jQuery adapter.

We can now set up our Django project to serve this HTML file as a static file and use it to test the server-side filepond support.

NOTE: This approach uses Django's static file serving support and it should not be used for production deployment.

Create a directory called `static` in ${TUTORIAL_DIR}.

Place the `filepond-jquery-example.html` file in this directory.

Now open the `${TUTORIAL_DIR}/drf_filepond_tutorial/urls.py` file for editing. We'll add a new URL mapping to allow access to static files placed into the `${TUTORIAL_DIR}/static/`. Add the following entry to the `urlpatterns` list:

```
url(r'^demo/(?P<path>.*)$', serve, {'document_root': os.path.join(settings.BASE_DIR,
↪'static')}),
```

You will also need to add 3 new import statements to the set of existing import statements:

```
import os
from django.views.static import serve
from django.conf import settings
```

4.1.4 A4. Test the service

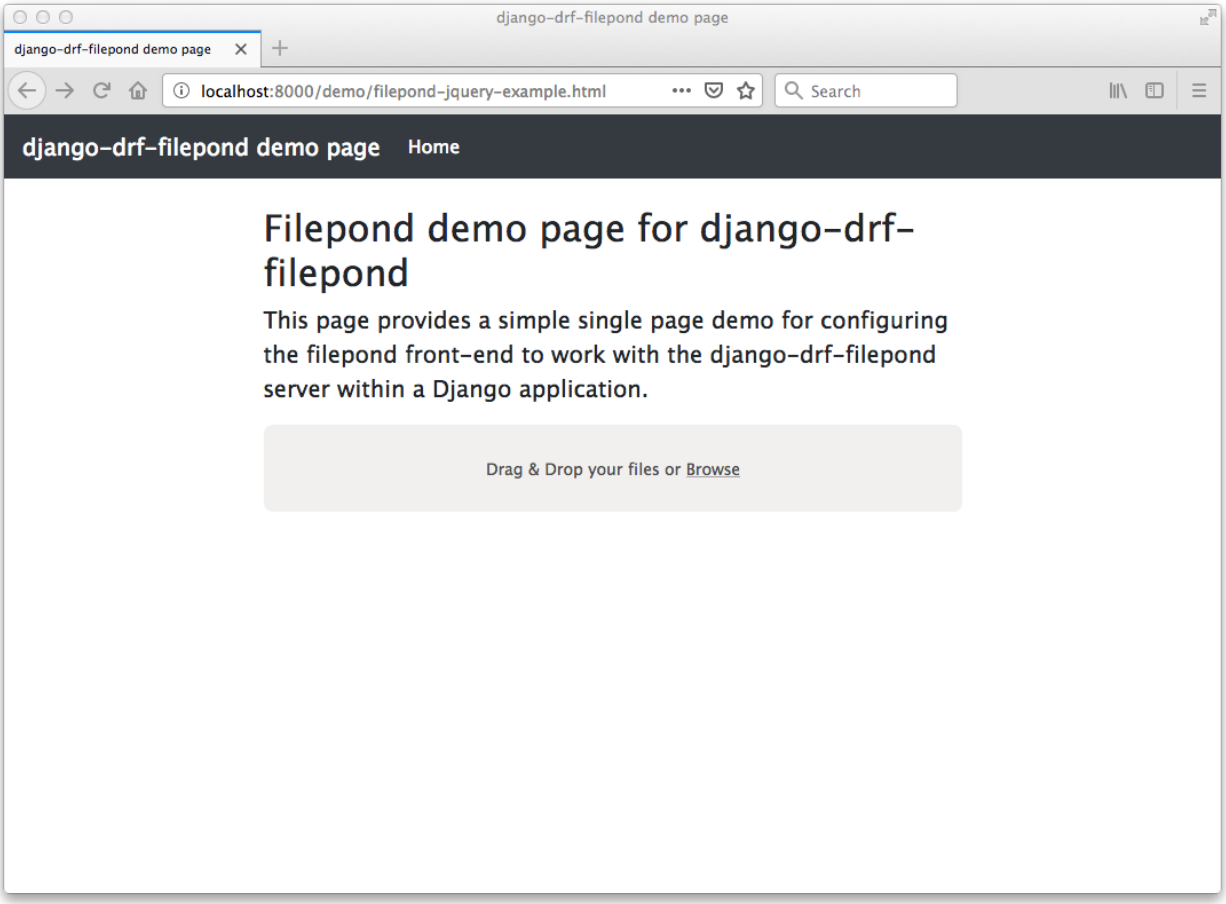
You are now in a position to test the project that you've set up.

In the `${TUTORIAL_DIR}` directory, with the virtualenv that was created in step 1 activated, start the Django development server:

```
$ python manage.py runserver
```

If there are any errors with your configuration, these will be shown in the terminal when you attempt to start the development server.

You should now be able to open the demo page in your browser. Point the browser to http://localhost:8000/demo/filepond-jquery-example.html and you should see the demo page shown in the figure below:



You can now try uploading a file to the server. Drag and drop a file onto the grey filepond panel on the web page or click *Browse* and select a file. The file should be uploaded successfully. If you look in the `${TUTORIAL_DIR}/filepond-temp-uploads` directory, the directory we set as the temporary upload directory using the `DJANGO_DRF_FILEPOND_UPLOAD_TMP` parameter in part A2 above, you will see that the file has been uploaded to this location and placed in a directory named using a unique ID. The file is also saved using a unique ID to replace it's original filename, which is preserved in the database. If you click the 'X' icon for the uploaded file

that appears alongside the "Upload complete" message, the temporary upload is cancelled and you should be able to verify that the file has now disappeared from the `${TUTORIAL_DIR}/filepond-temp-uploads` directory.

You can also test programmatically uploading a file from a remote URL. You can use your browser's developer console while on the django-drf-filepond demo page to call the filepond object's addFile method to get filepond to retrieve the file and add it. Place a test text file with some content in it into the `${TUTORIAL_DIR}/static/` directory. Call the file `test.txt`.

In your browser console, enter the following JavaScript code:

```
testFile = null;
result = $('.pond').filepond('addFile', 'http://localhost:8000/demo/test.txt').then(
        function(file) { testFile = file; }
);
```

You will now see that the value of `testFile.serverId` contains the ID generated for the upload from the URL. The file upload should have appeared in the filepond panel in the webpage and it can be cancelled by clicking the cancel button in the UI in the same way as a file uploaded from the local system by browsing or drag and drop.

4.1.5 Chunked file uploads

If you'd like to test the use of chunked file uploads, you can modify the filepond-jquery-example.html file to include the necessary configuration to enable chunked uploads as described in the "*Chunked uploads*" section of the "*Installation*" part of the main django-drf-filepond documentation.

4.2 Tutorial Part B: Using remote file storage

django-drf-filepond's remote file storage functionality enables you to place stored uploads on a remote file store. You can then use filepond's load endpoint to load a stored file directly from the remote storage. You can make use of any of the storage backends supported by django-storages. This includes, for example, Amazon S3 or Azure Storage.

Note: Remote storage is currently only supported for stored uploads. Temporary uploads are still stored locally in the location defined by the `DJANGO_DRF_FILEPOND_UPLOAD_TMP` parameter in your Django application's settings.

It is planned to add remote storage for temporary uploads in a future release.

This section of the tutorial assumes that you have completed part A and builds on the Django application developed there. To support this part of the tutorial, a separate demo HTML page is provided. This HTML file (filepond-jquery-example-advanced.html) includes a more advanced design to demonstrate the storage and retrieval of uploads and also the removal of stored uploads.

Note: Not all features detailed here are supported on all *django-storages* backends. Support depends directly on whether *django-storages* provides support for a given feature. For example, if *django-storages* doesn't support file deletion for a particular storage backend, *django-drf-filepond* will not support file deletion for that platform.

4.2.1 B1. Add a new web interface and REST endpoint to the demo app

Part B of the tutorial begins with updating the demo application that you set up in part A with a new HTML page, filepond-jquery-example-advanced.html, that contains a more advanced interface with additional functionality. Ob-

tain the HTML file directly from GitHub or copy it from your clone of the *django-drf-filepond* repository into the `${TUTORIAL_DIR}/static/` directory.

As demonstrated in part A of the tutorial, the initial upload of a file, where it is uploaded to the server as a temporary upload and shown in green within the filepond component, is handled directly by the filepond server API as implemented by *django-drf-filepond*. In the case of the temporary upload, this is handled by the `process` endpoint. After one or more files have been uploaded, when the form containing the filepond component is submitted, this must be handled by your application rather than by *django-drf-filepond*. In the case of this tutorial, the *drf-filepond-tutorial* app needs to handle the submission of the form that triggers the permanent storage of the file upload.

`filepond-jquery-example-advanced.html` contains an HTML form in which the filepond component is embedded. Clicking the "Store uploads" button triggers submission of the form. This form submission is handled by a view in the *drf_filepond_tutorial* app. In part A of the tutorial, there were no views within the *drf_filepond_tutorial* app itself. File uploads were handled by the views provided by *django-drf-filepond*. We now need a view in the *drf_filepond_tutorial* app to handle the form submission. A `views.py` file containing the implementation of a view class to handle requests from the web page is provided in the `docs/tutorial` directory of the *django-drf-filepond* repository.

Copy `docs/tutorial/views.py` from your clone of the *django-drf-filepond* repository and place it in `${TUTORIAL_DIR}/drf_filepond_tutorial/`.

Alternatively, download views.py directly from GitHub and place it in the `${TUTORIAL_DIR}/drf_filepond_tutorial/` directory.

It is now necessary to modify `${TUTORIAL_DIR}/drf_filepond_tutorial/urls.py` to link an endpoint URL to the form processing view in `views.py`. Add the following entry to the `urlpatterns` list in `urls.py`:

```
url(r'^submitForm/$', views.SubmitFormView.as_view(), name='submit_form'),
```

and add the following additional import statement below the existing import statements towards the top of the top of the `urls.py` file:

```
from drf_filepond_tutorial import views
```

This will ensure that all incoming requests to the `/submitForm/` URL are handled by the `SubmitFormView` class in the `views.py` file that you just added.

4.2.2 B2. Configure your storage backend

A Django class-based view is now in place that will handle calling the *django-drf-filepond* API to store a temporary upload to remote storage. However, at this stage we don't have any configuration in place to tell *django-drf-filepond* which storage backend to use and the settings for communicating with that backend and authenticating with it.

The storage backends provided by *django-storages* each include a number of configuration options. This includes a way to define the base location on the remote storage platform where files should be stored.

Note: If you have extended your demo app from part A before starting this part of the tutorial and have added the `DJANGO_DRF_FILEPOND_FILE_STORE_PATH` setting into your `${TUTORIAL_DIR}/drf_filepond_tutorial/settings.py` file, you should set it to `None` or remove it altogether from the settings file since this parameter is not used for remote file storage.

We'll now add some storage backend settings to `settings.py`. For the example here, we'll use the Amazon S3 storage backend in *django-storages* to talk to the open source, Amazon S3-compatible MinIO storage service. You can download and run MinIO within a docker container on your local system or you can use the same approach detailed here to target Amazon S3 directly.

To begin with, it will be necessary to add additional dependencies required by *django-storages*. The basic *django-storages* library is a required dependency of *django-drf-filepond* but different storage backends may have additional dependencies that need to be installed. These additional dependencies can be installed using the `pip` package manager. For details of any additional dependencies required by a given backend you can look in the `extras_require` section of the django-storages setup.py file. This shows, for example, that the `sftp` backend requires the `paramiko` library. boto3 is the library used for accessing Amazon Web Services and we'll require `boto3` to be installed to use the Amazon S3 storage backend in this example.

Ensuring that you have first activated the Python virtualenv virtual environment (set up in section A1 of the tutorial) in your terminal, install `boto3` as follows:

```
$ pip install boto3
```

The *django-storages* documentation for the Amazon S3 backend details the various configuration settings that are available.

django-drf-filepond requires that, for a remote storage backend, you set the `DJANGO_DRF_FILEPOND_STORAGES_BACKEND` parameter in your `settings.py` file. The value to use for this parameter is the same as value shown in the *django-storages* documentation for the *DEFAULT_FILE_STORAGE* setting for a given storage backend. For example, for the Amazon S3 backend, this would be `'storages.backends.s3boto3.S3Boto3Storage'`. For Azure Storage, the value would be `'storages.backends.azure_storage.AzureStorage'`. Set the parameter in your *drf-filepond-tutorial* `${TUTORIAL_DIR}/drf_filepond_tutorial/settings.py` file as follows:

```
DJANGO_DRF_FILEPOND_STORAGES_BACKEND = 'storages.backends.s3boto3.S3Boto3Storage'
```

You now need to add a number of *django-storages*-specific parameters to configure the S3 backend. For targeting a local MinIO deployment, running over SSL with a valid SSL server certificate, we use the following parameters (note that you'll need to modify some of the values to match your own MinIO or S3 settings):

```
AWS_ACCESS_KEY_ID = '<Your MinIO access key>'
AWS_SECRET_ACCESS_KEY = '<Your MinIO secret key>'
AWS_STORAGE_BUCKET_NAME = 'drf-filepond-tutorial'
AWS_AUTO_CREATE_BUCKET = True
AWS_S3_ENDPOINT_URL = 'https://myminio.local:9000'
```

With this configuration, when you first attempt to store a temporary upload, a bucket named *drf-filepond-tutorial* will be created in MinIO, if it is not already present, and your stored upload will be placed in that bucket, prefixed with any relative path location provided in the code that stores the upload.

If you wish to target Amazon S3 directly, a couple of changes to the above settings will be required, the following set of settings will allow you to store uploads to S3:

```
AWS_ACCESS_KEY_ID = '<Your AWS access key>'
AWS_SECRET_ACCESS_KEY = '<Your AWS secret key>'
AWS_STORAGE_BUCKET_NAME = 'drf-filepond-tutorial'
AWS_AUTO_CREATE_BUCKET = True
AWS_S3_REGION_NAME = 'eu-west-1' # Set to your chosen storage region
```

As mentioned above, you can find the full set of available S3 configuration options in the django-storages S3 documentation.

Warning: Avoid storing your AWS/MinIO credentials directly in your configuration file. Be very careful to ensure that your settings file containing private credentials is not unintentionally committed to a code repository, especially a public repository!

There are various options for avoiding placing credentials directly in configuration files and many discussions online of methods. This blog post provides some useful examples and ideas.

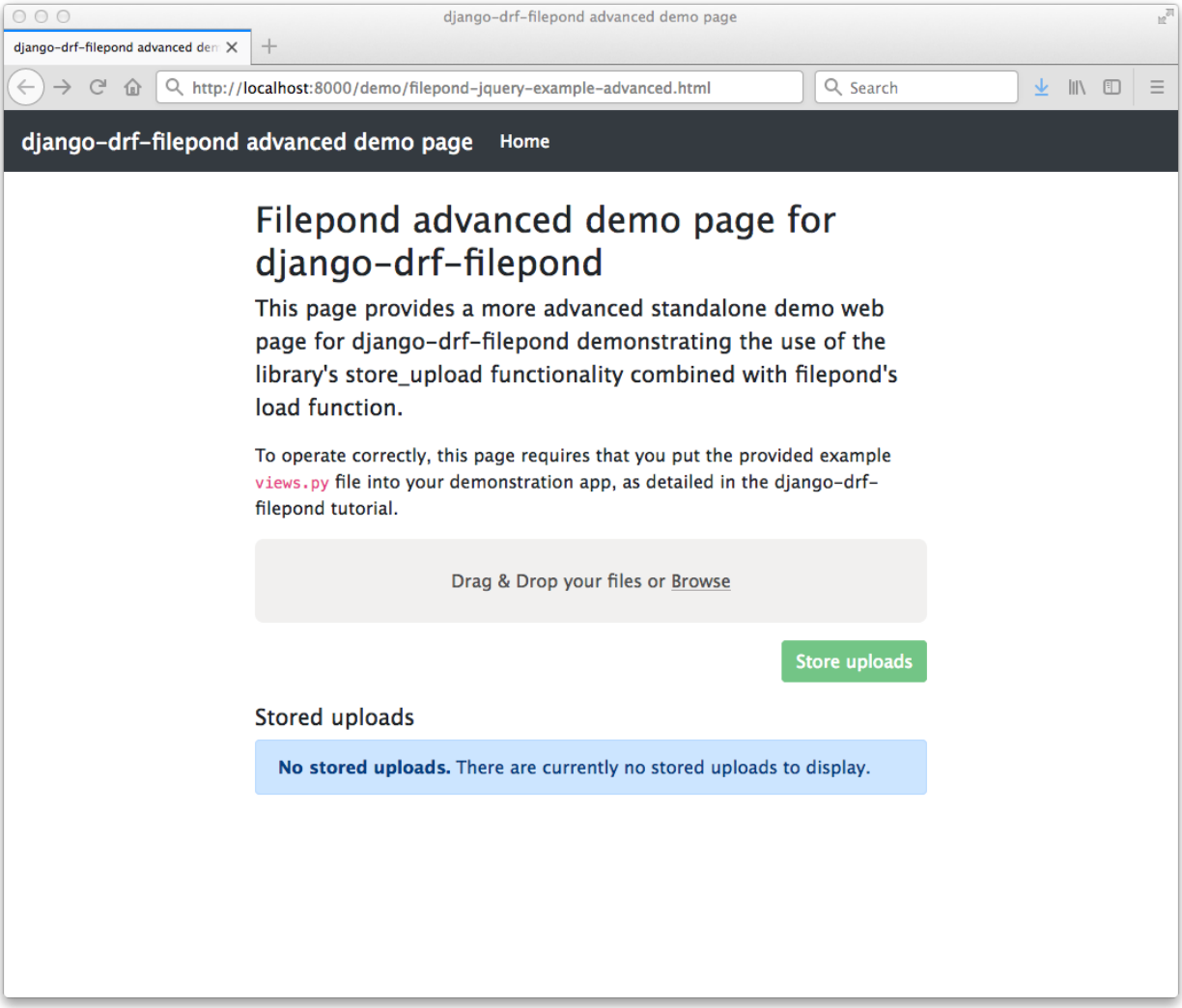
4.2.3 B3: Testing the updated service

Ensure that you have an open terminal in which you have activated the Python virtual environment that you created in tutorial section A1.

If you have stopped the Django development server that was started in part A of the tutorial, you should restart it now by running the following in a shell in the `${TUTORIAL_DIR}` directory:

```
$ python manage.py runserver
```

Assuming that the server starts successfully and there are no errors, you should now be able to open the advanced demo page in your browser. Point the browser to http://localhost:8000/demo/filepond-jquery-example-advanced.html and you should see the advanced demo page shown in the figure below:



Drag and drop a file onto the grey filepond component panel or click *Browse* to select a file to upload. It is recommended that you add a png or jpeg image file with the extension `.png` or `.jpg` for the purpose of the example in this tutorial. The file should upload successfully and the *Store uploads* button should become active.

If you now click the *Store uploads* button, this will make a request (containing one or more unique IDs representing the filepond temporary upload(s)) to the web application's */submitForm* URL and this will then be handled by the relevant function in the view class defined in the *views.py* file that you added to the application in section B1.

Assuming that your configuration is correct and the request is successful, you should then see the file you uploaded appear in the *Stored uploads* section of the page. If you uploaded a png or jpeg image file, the demo page will make a request to the *django-drf-filepond load* endpoint to retrieve the file from the remote storage platform and display it as a preview. If you see the image displayed, then your link to the remote storage platform is fully operational.

You can verify this by using MinIO or S3's web-based console to check that the file you stored has been correctly uploaded to the remote platform.

If you now click the *Delete stored upload* button, this will DELETE THE FILE from the remote storage platform. You should now be able to verify that the file has been removed from the remote storage platform.

Note: There is a known issue with file deletion on storage platforms that are based on a standard filesystem, for example *django-storages* SFTP backend.

When a file is deleted, using the API, the file itself is removed but any directories created to store the file at the full path specified when storing the file are left in place. This was a design decision since there is currently no way to know exactly which directories were created when the upload was stored so removing an arbitrary set of directories on a remote filesystem was not considered a reasonable approach.

If there is demand for use of the SFTP backend, there is scope to store in the database details of created directories and then remove these if they're empty when a file is removed.

This completes the advanced section of the *django-drf-filepond* tutorial. If you require assistance with using the *django-drf-filepond* API to store files to a remote storage backend, take a look at the code in the example views.py file provided with the tutorial.