Uploaded Files and Upload Handlers

Uploaded files

class **`UploadedFile`**[source]

During file uploads, the actual file data is stored in **`request.FILES`**. Each entry in this dictionary is an **`UploadedFile`** object (or a subclass) – a wrapper around an uploaded file. You'll usually use one of these methods to access the uploaded content:

`UploadedFile.read`()

Read the entire uploaded data from the file. Be careful with this method: if the uploaded file is huge it can overwhelm your system if you try to read it into memory. You'll probably want to use **`chunks()`** instead; see below.

`UploadedFile.multiple_chunks`(*chunk_size=None*)

Returns **`True`** if the uploaded file is big enough to require reading in multiple chunks. By default this will be any file larger than 2.5 megabytes, but that's configurable; see below.

`UploadedFile.chunks`(*chunk_size=None*)

A generator returning chunks of the file. If **`multiple_chunks()`** is **`True`**, you should use this method in a loop instead of **`read()`**.

In practice, it's often easiest to use **`chunks()`** all the time. Looping over **`chunks()`** instead of using **`read()`** ensures that large files don't overwhelm your system's memory.

Here are some useful attributes of **`UploadedFile`**:

`UploadedFile.name`

The name of the uploaded file (e.g. **`my_file.txt`**).

`UploadedFile.size`

The size, in bytes, of the uploaded file.

`UploadedFile.content_type`

The content-type header uploaded with the file (e.g. *text/plain* or *application/pdf*). Like any data supplied by the user, you shouldn't trust that the uploaded file is actually this type. You'll still need to validate that the file contains the content that the content-type header claims – "trust but verify."

`UploadedFile.content_type_extra`

A dictionary containing extra parameters passed to the **`content-type`** header. This is typically provided by services, such as Google App Engine, that intercept and handle file uploads on your behalf. As a result your handler may not receive the uploaded file content, but instead a URL or other pointer to the file (see **RFC 2388**).

`UploadedFile.charset`

For *text/** content-types, the character set (i.e. **`utf8`**) supplied by the browser. Again, "trust but verify" is the best policy here.

Note

Like regular Python files, you can read the file line-by-line by iterating over the uploaded file:

```
for line in uploadedfile:
    do_something_with(line)
```

Getting Help

Language: **en**

Lines are split using universal newlines. The following are recognized as ending a line: the Unix end-of-line convention **`'\n'`**, the Windows convention **`'\r\n'`**, and the old Macintosh convention **`'\r'`**.

Documentation version: **3.0**

Subclasses of **`UploadedFile`** include:

class **`TemporaryUploadedFile`**[source]

A file uploaded to a temporary location (i.e. stream-to-disk). This class is used by the **`TemporaryFileUploadHandler`**. In addition to the methods from **`UploadedFile`**, it has one additional method:

`TemporaryUploadedFile.temporary_file_path`()[source]

Returns the full path to the temporary uploaded file.

class **`InMemoryUploadedFile`**[source]

A file uploaded into memory (i.e. stream-to-memory). This class is used by the **`MemoryFileUploadHandler`**.

Built-in upload handlers

Together the **`MemoryFileUploadHandler`** and **`TemporaryFileUploadHandler`** provide Django's default file upload behavior of reading small files into memory and large ones onto disk. They are located in **`django.core.files.uploadhandler`**.

class **`MemoryFileUploadHandler`**[source]

File upload handler to stream uploads into memory (used for small files).

class **`TemporaryFileUploadHandler`**[source]

Upload handler that streams data into a temporary file using **`TemporaryUploadedFile`**.

Writing custom upload handlers

class **`FileUploadHandler`**[source]

All file upload handlers should be subclasses of **`django.core.files.uploadhandler.FileUploadHandler`**. You can define upload handlers wherever you wish.

Required methods

Custom file upload handlers **must** define the following methods:

`FileUploadHandler.receive_data_chunk`(*raw_data*, *start*)[source]

Receives a "chunk" of data from the file upload.

`raw_data` is a bytestring containing the uploaded data.

`start` is the position in the file where this **`raw_data`** chunk begins.

The data you return will get fed into the subsequent upload handlers' **`receive_data_chunk`** methods. In this way, one handler can be a "filter" for other handlers.

Return **`None`** from **`receive_data_chunk`** to short-circuit remaining upload handlers from getting this chunk. This is useful if you're storing the uploaded data yourself and don't want future handlers to store a copy of the data.

If you raise a **`StopUpload`** or a **`SkipFile`** exception, the upload will abort or the file will be completely skipped.

`FileUploadHandler.file_complete`(*file_size*)[source]

Called when a file has finished uploading.

The handler should return an **`UploadedFile`** object that will be stored in **`request.FILES`**. Handlers may also return **`None`** to indicate that the **`UploadedFile`** object should come from subsequent upload handlers.

Optional methods

Custom upload handlers may also define any of the following optional methods or attributes:

`FileUploadHandler.chunk_size`

Size, in bytes, of the "chunks" Django should store into memory and feed into the handler. That is, this attribute controls the size of chunks fed into **`FileUploadHandler.receive_data_chunk`**.

For maximum performance the chunk sizes should be divisible by `4` and should not exceed 2 GB (2^{31} bytes) in size. When there are multiple chunk sizes provided by multiple handlers, Django will use the smallest chunk size defined by any handler.

The default is $64*2^{10}$ bytes, or 64 KB.

`FileUploadHandler.new_file`(*field_name*, *file_name*, *content_type*, *content_length*, *charset*, *content_type_extra*)[source]

Callback signaling that a new file upload is starting. This is called before any data has been fed to any upload handlers.

`field_name` is a string name of the file **`<input>`** field.

`file_name` is the filename provided by the browser.

`content_type` is the MIME type provided by the browser – E.g. **`'image/jpeg'`**.

`content_length` is the length of the image given by the browser. Sometimes this won't be provided and will be **`None`**.

`charset` is the character set (i.e. **`utf8`**) given by the browser. Like **`content_length`**, this sometimes won't be provided.

`content_type_extra` is extra information about the file from the **`content-type`** header. See **`UploadedFile.content_type_extra`**.

This method may raise a **`StopFutureHandlers`** exception to prevent future handlers from handling this file.

`FileUploadHandler.upload_complete`()[source]

Callback signaling that the entire upload (all files) has completed.

`FileUploadHandler.handle_raw_input`(*input_data*, *META*, *content_length*, *boundary*, *encoding*)[source]

Allows the handler to completely override the parsing of the raw HTTP input.

`input_data` is a file-like object that supports **`read()`**-ing.

`META` is the same object as **`request.META`**.

`content_length` is the length of the data in **`input_data`**. Don't read more than **`content_length`** bytes from **`input_data`**.

`boundary` is the MIME boundary for this request.

`encoding` is the encoding of the request.

Return **`None`** if you want upload handling to continue, or a tuple of **`(POST, FILES)`** if you want to return the new data structures suitable for the request directly.

‹ **File storage API** **Forms** ›