File Depot Documentation

Release 0.7.1

Alessandro Molina

Jun 15, 2023

Contents

Welcome to the DEPOT Documentation. DEPOT is a framework for easily storing and serving files in web applications on Python2.6+ and Python3.2+.

Depot can be used *Standalone* or *with your ORM* to quickly provide attachment support to your model.

Modern web applications need to rely on a huge amount of stored images, generated files and other data which is usually best to keep outside of your database. DEPOT provides a simple and effective interface for storing your files on a storage backend at your choice (**Local**, **S3**, **GridFS**, **Google Cloud Storage**) and easily relate them to your application models (**SQLAlchemy**, **Ming**) like you would for plain data.

Depot is a swiss army knife for files that provides:

- Multiple backends: Store your data on all of them with a *single API*
- In Memory storage `depot.io.memory.MemoryFileStorage` provided for tests suite. Provides faster tests and no need to clean-up fixtures.
- Meant for *Evolution*: Change the backend anytime you want, old data will continue to work
- Integrates with your *ORM*: When using SQLAlchemy, attachments are handled like a plain model attribute. It's also *session ready*: Rollback causes the files to be deleted.
- Smart *File Serving*: When the backend already provides a public HTTP endpoint (like S3) the WSGI `depot.middleware.DepotMiddleware` will redirect to the public address instead of loading and serving the files by itself.
- Flexible: The `depot.manager.DepotManager` will handle configuration, middleware creation and files for your application, but if you want you can manually create multiple depots or middlewares without using the manager.

DEPOT was presented at PyConUK 2014 and PyConFR 2014, for a short presentation you can have a look at the PyConFR slides:

CHAPTER 1

Installing Depot

Installing DEPOT can be done from PyPi itself by installing the `filedepot` distribution:

```
$ pip install filedepot
```

Keep in mind that DEPOT itself has no dependencies, if you want to use GridFS storage, S3, GCS or any other storage that requires third party libraries, your own application is required to install the dependency. In this specific case `pymongo` and `boto` (or `boto3`) and `google-cloud-storage` are respectively needed for GridFS, S3 and GCS support.

CHAPTER 2

Depot Standalone

Depot can easily be used to save and retrieve files in any Python script, Just get a depot using the *depot.manager.DepotManager* and store the files. With each file, additional data commonly used in HTTP like *last_modified*, *content_type* and so on is stored. This data is available inside the *depot.io.interfaces.StoredFile* which is returned when getting the file back:

```
from depot.manager import DepotManager

# Configure a *default* depot to store files on MongoDB GridFS
DepotManager.configure('default', {
    'depot.backend': 'depot.io.gridfs.GridFSStorage',
    'depot.mongouri': 'mongodb://localhost/db'
})

depot = DepotManager.get()

# Save the file and get the fileid
fileid = depot.create(open('/tmp/file.png'))

# Get the file back
stored_file = depot.get(fileid)
print stored_file.filename
print stored_file.content_type
```

CHAPTER 3

Depot with your WebFramework

To start using DEPOT with your favourite Web Framework you can have a look at the web framework examples and read the *DEPOT for Web* guide.

CHAPTER 4

Attaching Files to Models

Depot also features simple integration with SQLAlchemy by providing customized model field types for storing files attached to your ORM document. Just declare columns with these types inside your models and assign the files:

```
from depot.fields.sqlalchemy import UploadedFileField
from depot.fields.specialized.image import UploadedImageWithThumb


class Document(Base):
    __tablename__ = 'document'

    uid = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(16), unique=True)
    content = Column('content_col', UploadedFileField)  # plain attached file

    # photo field will automatically generate thumbnail
    photo = Column(UploadedFileField(upload_type=UploadedImageWithThumb))


# Store documents with attached files, the source can be a file or bytes
doc = Document(name=u'Foo',
               content=b'TEXT CONTENT STORED AS FILE',
               photo=open('/tmp/file.png'))
DBSession.add(doc)
DBSession.flush()

# DEPOT is session aware, commit/rollback to keep or delete the stored files.
DBSession.commit()
```

Fetching back the attachments is as simple as fetching back the documents themselves:

```
d = DBSession.query(Document).filter_by(name=u'Foo').first()
print d.content.file.read()
print d.photo.url
print d.photo.thumb_url
```

In case the backend doesn't already provide HTTP serving of the files you can use the DepotMiddleware to serve them in your application:

```
wsgiapp = DepotManager.make_middleware(wsgiapp)
```

CHAPTER 5

User Guide

5.1 Getting Started with Depot

5.1.1 Configuring DepotManager

The DepotManager is the entity in charge of configuring and handling file storages inside your application. To start saving files the first required step is to configure a file storage through the DepotManager.

This can be done using *DepotManager.configure()* which accepts a storage name (used to identify the storage in case of multiple storages) and a set of configuration options:

```
DepotManager.configure('default', {
    'depot.storage_path': './files'
})
```

By default a *depot.io.local.LocalFileStorage* storage is configured, LocalFileStorage saves files on the disk at the `storage_path`. You can use one of the available storages through the `.backend` option. To store data on GridFS you would use:

```
DepotManager.configure('my_gridfs', {
    'depot.backend': 'depot.io.gridfs.GridFSStorage',
    'depot.mongouri': 'mongodb://localhost/db'
})
```

Every other option apart the `.backend` one will be passed to the storage as a constructor argument. You can even use your own storage by setting the full python path of the class you want to use.

By default the first configured storage is the default one, which will be used whenever no explict storage is specified, to change the default storage you can use *DepotManager.set_default()* with the name of the storage you want to make the default one.

5.1.2 Getting a Storage

Once you have configured at least one storage, you can get it back using the *DepotManager.get()* method. If you pass a specific storage name it will retrieve the storage configured for that name:

```
depot = DepotManager.get('my_gridfs')
```

Otherwise the default storage can be retrieved by omitting the name argument:

```
depot = DepotManager.get()
```

5.2 Save and Manage Files

5.2.1 Saving and Retrieving Files

Once you have a working storage, saving files is as easy as calling the *FileStorage.create()* method passing the file (or the `bytes` object) you want to store:

```
depot = DepotManager.get()
fileid = depot.create(open('/tmp/file.png'))
```

The returned `fileid` will be necessary when you want to get back the stored file. By default the `name`, `content_type` and all the properties available through the *StoredFile* object are automatically detect from the argument file object.

If you want to explicitly set filename and content type they can be passed as arguments to the `create` method:

```
fileid = depot.create(open('/tmp/file.png'), 'thumbnail.png', 'image/png')
```

Getting the file back can be done using *FileStorage.get()* from the storage itself:

```
stored_file = depot.get(fileid)
```

Getting back the file will only retrieve the file metadata and will return a *StoredFile* object. This object can be used like a normal Python `file` object, so if you actually want to read the file content you should then call the `read` method:

```
stored_file.content_type  # This will be 'image/png'
image = stored_file.read()
```

If you don't have the depot instance available, you can use the *DepotManager.get_file()* method which takes the path of the stored file. Paths are in the form `depot_name/fileid`:

```
stored_file = DepotManager.get_file('my_gridfs/%s' % fileid)
```

5.2.2 Replacing and Deleting Files

If you don't need a file anymore it can easily be deleted using the *FileStorage.delete()* method with the file id:

```
depot.delete(fileid)
```

The `delete` method is guaranteed to be idempotent, so calling it multiple times will not lead to errors.

The storage can also be used to replace existing files, replacing the content of a file will actually also replace the file metadata:

```
depot.replace(fileid, open('/tmp/another_image.jpg'),
              'thumbnail.jpg', 'image/png')
```

This has the same behavior of deleting the old file and storing a new one, but instead of generating a new id it will reuse the existing one. As for the `create` call the filename and content type arguments can be omitted and will be detected from the file itself when available.

5.2.3 Storing data as files

Whenever you do not have a real file (often the case with web uploaded content), you might not be able to retrieve the name and the content type from the file itself, of those values might be wrong.

In such case *depot.io.utils.FileIntent* can be provided to DEPOT instead of the actual file, *depot.io.utils.FileIntent* can be used to explicitly tell DEPOT which filename and content_type to use to store the file. Also non files can be provided to FileIntent to store raw data:

```
# Works with file objects
file_id = self.fs.create(
    FileIntent(open('/tmp/file', 'rb'), 'file.txt', 'text/plain')
)

# Works also with bytes
file_id = self.fs.create(
    FileIntent(b'HELLO WORLD', 'file.txt', 'text/plain')
)

f = self.fs.get(file_id)
assert f.content_type == 'text/plain'
assert f.filename == 'file.txt'
assert f.read() == b'HELLO WORLD'
```

5.3 Depot for the Web

5.3.1 File Metadata

As Depot has been explicitly designed for web applications development, it will provide all the file metadata which is required for HTTP headers when serving files or which are common in the web world.

This is provided by the *StoredFile* you retrieve from the file storage and includes:

- `filename` -> Original name of the file, if you need to serve it to the user for download.
- `content_type` -> File content type, for the response content type when serving file back the file to the browser.
- `last_modified` -> Can be used to implement caching and last modified header in HTTP.
- `content_length` -> Size of the file, is usually the content length of the HTTP response when serving the file back.

5.3.2 Serving Files on HTTP

In case of storages that directly support serving files on HTTP (like *depot.io.awss3.S3Storage*, *depot.io.boto3.S3Storage* and depot.io.gcs.GCSStorage) the stored file itself can be retrieved at the url provided by StoredFile.public_url. In case the public_url is None it means that the storage doesn't provide direct HTTP access.

In such case files can be served using a *DepotMiddleware* WSGI middleware. The DepotMiddlware supports serving files from any backend, supports ETag caching and in case of storages directly supporting HTTP it will just redirect the user to the storage itself.

Unless you need to achieve maximum performances it is usually a good approach to just use the WSGI Middleware and let it serve all your files for you:

```
app = DepotManager.make_middleware(app)
```

By default the Depot middleware will serve the files at the /depot URL using their path (the same as passed to the *DepotManager.get_file()* method). So in case you need to retrieve a file with id **3774a1a0-0879-11e4-b658-0800277ee230** stored into **my_gridfs** depot the URL will be /depot/my_gridfs/3774a1a0-0879-11e4-b658-0800277ee230.

Changing the base URL and caching can be done through the *DepotManager.make_middleware()* options, any option passed to make_middleware will be forwarded to *DepotMiddleware*.

5.4 Handling Multiple Storages

5.4.1 Using Multiple Storages

Multiple storage can be used inside the same application, most common operations require the storage itself or the full file path, so you can use multiple storage without risk of collisions.

To start using multiple storage just call the *DepotManager.configure()* multiple times and give each storage a unique name. You will be able to retrieve the correct storage by name.

5.4.2 Switching Default Storage

Once you started uploading files to a storage, it is best to avoid configuring another storage to the same name. Doing that will probably break all the previously uploaded files and will cause confusion.

If you want to switch to a different storage for saving your files just configure two storage giving the new storage an unique name and switch the default storage using the *DepotManager.set_default()* function.

5.4.3 Replacing a Storage through Aliases

Originally DEPOT only permitted switching the default storage, that way you could replace the storage in use whenever you needed and keep the old files around as the previous storage was still available. This was by the way only permitted for the default storage, since version 0.0.7 the DepotManager.alias() feature is provided which permits to assign alternative names for a storage.

If you only rely on the alternative name and never use the real storage name, you will be able to switch the alias to whatever new storage you want while the files previously uploaded to the old storage keep wFor example if you are storing all your user avatars locally you might have a configuration like:

```
DepotManager.configure('local_avatars', {
    'depot.storage_path': '/var/www/lfs'
})
DepotManager.alias('avatar', 'local_avatars')

storage = DepotManager.get('avatar')
fileid = storage.create(open('/tmp/file.png'), 'thumbnail.png', 'image/png')
```

Then when switching your avatars to GridFS you might switch your configuration to something like:

```
DepotManager.configure('local_avatars', {
    'depot.storage_path': '/var/www/lfs'
})
DepotManager.configure('gridfs_avatars', {
    'depot.backend': 'depot.io.gridfs.GridFSStorage',
    'depot.mongouri': 'mongodb://localhost/db'
})
DepotManager.alias('avatar', 'gridfs_avatars')

storage = DepotManager.get('avatar')
fileid = storage.create(open('/tmp/file.png'), 'thumbnail.png', 'image/png')
```

Note: While you can keep using the `avatar` name for the storage when saving files, it's important that the `local_avatars` storage continues to be configured as all the previously uploaded avatars will continue to be served from there.

5.4.4 Performing Backups between Storages

When in need to perform a backup between two storages, the best practice is to rely on the backend specific tools. Those are usually faster than trying to copy each file one by one in python.

In case you have the need to perform backups through the DEPOT apis themselves, you can configure a second `FileStorage` where you can copy all the files using the second storage `FileStorage.replace()` method:

```
DepotManager.configure('local_avatars', {
    'depot.storage_path': '/var/www/lfs'
})
DepotManager.configure('backup_avatars', {
    'depot.storage_path': '/var/www/lfs_backup'
})

storage = DepotManager.get('local_avatars')
backup = DepotManager.get('backup_avatars')

for fileid in storage.list():
    f = storage.get(fileid)
    backup.replace(f, f)
```

Note: This backup method will be very slow compared to native backup tools of the storage in use. As it has to download the file locally to reupload it to the backup storage.

5.5 Depot with Database

Depot provides built-in support for attachments to models, uploading a file and attaching it to a database entry is as simple as assigning the file itself to a model field.

5.5.1 Attaching to Models

Attaching files to models is as simple as declaring a field on the model itself, support is currently provided for **SQLAlchemy** through the *depot.fields.sqlalchemy.UploadedFileField* and for **Ming (MongoDB)** through the *depot.fields.ming.UploadedFileProperty*:

```
from depot.fields.sqlalchemy import UploadedFileField

class Document(Base):
    __tablename__ = 'document'

    uid = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(16), unique=True)

    content = Column(UploadedFileField)
```

To actually store the file into the Document, assigning it to the `content` property is usually enough, just like files uploaded using *FileStorage.create()* both file objects, `cgi.FieldStorage` and `bytes` can be used:

```
# Store documents with attached files, the source can be a file or bytes
doc = Document(name=u'Foo',
               content=open('/tmp/document.xls'))
DBSession.add(doc)
```

Note: In case of Python3 make sure the file is open in byte mode.

Depot will upload files to the default storage, to change where files are uploaded use *DepotManager.set_default()*.

5.5.2 Uploaded Files Information

Whenever a supported object is assigned to a *UploadedFileField* or *UploadedFileProperty* it will be converted to a *UploadedFile* object.

This is the same object you will get back when reloading the models from database and apart from the file itself which is accessible through the `.file` property, it provides additional attributes described into the *UploadedFile* documentation itself.

Most important property is probably the `.url` property which provides an URL where the file can be accessed in case the storage supports HTTP or the *DepotMiddleware* is available in your WSGI application.

5.5.3 Uploading on a Specific Storage

By default all the files are uploaded on the `default` storage (the one returned by *DepotManager.get_default()*. This can be changed by passing a `upload_storage` argument explicitly to the database field declaration:

```
from depot.fields.sqlalchemy import UploadedFileField

class Document(Base):
    __tablename__ = 'document'

    uid = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(16), unique=True)

    content = Column(UploadedFileField(upload_storage='another_storage'))
```

If the specified `upload_storage` is an *alias* to another storage, the file will actually keep track of the real storage, so that when the alias is switched to another storage, previously uploaded files continue to get served from the old storage.

5.5.4 Session Awareness

Whenever an object is *deleted* or a *rollback* is performed the files uploaded during the unit of work or attached to the deleted objects are automatically deleted.

This is performed out of the box for SQLAlchemy, but requires the *DepotExtension* to be registered as a session extension for Ming.

Note: Ming doesn't currently provide an entry point for session clear, so files uploaded without a session flush won't be deleted when the session is removed.

5.6 Custom Behaviour in Attachments

Often attaching a file to the model is not enough, if a video is uploaded you probably want to convert it to a supported format. Or if a big image is uploaded you might want to scale it down.

Most simple changes can be achieved using **Filters**, filters can create thumbnails of an image or trigger actions when the file gets uploaded, multiple filters can be specified as a list inside the `filters` parameter of the column. More complex actions like editing the content before it gets uploaded can be achieved subclassing *UploadedFile* and passing it as column `upload_type`.

5.6.1 Attachment Filters

File filters are created by subclassing *FileFilter* class, the only required method to implement is *FileFilter.on_save()* which you are required implement with the actions you want to perform. The method will receive the uploaded file (after it already got uploaded) and can add properties to it.

Inside filters the original content is available as a property of the uploaded file, by accessing `original_content` you can read the original content but not modify it, as the file already got uploaded changing the original content has no effect.

If you need to store additional files, only use the `UploadedFile.store_content()` method so that they are correctly tracked by the unit of work and deleted when the associated document is deleted.

A filter that creates a thumbnail for an image would look like:

```
from depot.io import utils
from PIL import Image
from io import BytesIO


class WithThumbnailFilter(FileFilter):
    def __init__(self, size=(128,128), format='PNG'):
        self.thumbnail_size = size
        self.thumbnail_format = format

    def on_save(self, uploaded_file):
        content = utils.file_from_content(uploaded_file.original_content)

        thumbnail = Image.open(content)
        thumbnail.thumbnail(self.thumbnail_size, Image.BILINEAR)
        thumbnail = thumbnail.convert('RGBA')
        thumbnail.format = self.thumbnail_format

        output = BytesIO()
        thumbnail.save(output, self.thumbnail_format)
        output.seek(0)

        thumb_file_name = 'thumb.%s' % self.thumbnail_format.lower()

        # If you upload additional files do it with store_content
        # to ensure they are correctly tracked by unit of work and
        # removed on model deletion.
        thumb_path, thumb_id = uploaded_file.store_content(output,
                                                           thumb_file_name)
        thumb_url = DepotManager.get_middleware().url_for(thumb_path)

        uploaded_file['thumb_id'] = thumb_id
        uploaded_file['thumb_path'] = thumb_path
        uploaded_file['thumb_url'] = thumb_url
```

To use it, just provide the `filters` parameter in your *UploadedFileField* or *UploadedFileProperty*:

```
class Document(DeclarativeBase):
    __tablename__ = 'docu'

    uid = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(16), unique=True)

    photo = Column(UploadedFileField(filters=[WithThumbnailFilter()]))
```

As *UploadedFile* remembers every value/attribute stored before saving it on the database, all the *thumb_id*, *thumb_path* and *thumb_url* values will be available when loading back the document:

```
>>> d = DBSession.query(Document).filter_by(name='Foo').first()
>>> print d.photo.thumb_url
/depot/default/5b1a489e-0d33-11e4-8e2a-0800277ee230
```

5.6.2 Custom Attachments

Filters are convenient and can be mixed together to enable multiple behaviours when a file is uploaded, but they have a limit: They cannot modify the uploaded file or the features provided when the file is retrieved from the database.

To avoid this limit users can specify their own upload type by subclassing *UploadedFile*. By specializing the *UploadedFile.process_content()* method it is possible to change the content before it's stored and provide additional attributes.

Whenever the stored document is retrieved from the database, the file will be recovered with the same type specified as the `upload_type`, so any property or method provided by the specialized type will be available also when the file is loaded back.

A possible use case for custom attachments is ensure an image is uploaded at a maximum resolution:

```
from depot.io import utils
from depot.fields.upload import UploadedFile
from depot.io.interfaces import FileStorage
from PIL import Image
from depot.io.utils import INMEMORY_FILESIZE
from tempfile import SpooledTemporaryFile


class UploadedImageWithMaxSize(UploadedFile):
    max_size = 1024

    def process_content(self, content, filename=None, content_type=None):
        # As we are replacing the main file, we need to explicitly pass
        # the filanem and content_type, so get them from the old content.
        __, filename, content_type = FileStorage.fileinfo(content)

        # Get a file object even if content was bytes
        content = utils.file_from_content(content)

        uploaded_image = Image.open(content)
        if max(uploaded_image.size) >= self.max_size:
            uploaded_image.thumbnail((self.max_size, self.max_size),
                                     Image.BILINEAR)
            content = SpooledTemporaryFile(INMEMORY_FILESIZE)
            uploaded_image.save(content, uploaded_image.format)

        content.seek(0)
        super(UploadedImageWithMaxSize, self).process_content(content,
                                                              filename,
                                                              content_type)
```

Using it to ensure every uploaded image has a maximum resolution of 1024x1024 is as simple as passing it to the column:

```
class Document(DeclarativeBase):
    __tablename__ = 'docu'

    uid = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(Unicode(16), unique=True)

    photo = Column(UploadedFileField(upload_type=UploadedImageWithMaxSize))
```

When saved the image will be automatically resized to 1024 when bigger than the maximum allowed size.

CHAPTER 6

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

6.1 API Reference

This part of the documentation covers all the public classes in Depot.

6.1.1 Application Support

class depot.manager.**DepotManager**

Takes care of managing the whole Depot environment for the application.

DepotManager tracks the created depots, the current default depot, and the WSGI middleware in charge of serving files for local depots.

While this is used to create the default depot used by the application it can also create additional depots using the `new()` method.

In case you need to migrate your application to a different storage while keeping compatibility with previously stored file simply change the default depot through *set_default()* all previously stored file will continue to work on the old depot while new files will be uploaded to the new default one.

classmethod configure(*name*, *config*, *prefix='depot.'*)

Configures an application depot.

This configures the application wide depot from a settings dictionary. The settings dictionary is usually loaded from an application configuration file where all the depot options are specified with a given `prefix`.

The default `prefix` is *depot.*, the minimum required setting is `depot.backend` which specified the required backend for files storage. Additional options depend on the choosen backend.

classmethod from_config(*config*, *prefix='depot.'*)

Creates a new depot from a settings dictionary.

Behaves like the *configure()* method but instead of configuring the application depot it creates a new one each time.

classmethod get(*name=None*)
Gets the application wide depot instance.

Might return None if *configure()* has not been called yet.

classmethod get_default()
Retrieves the current application default depot

classmethod get_file(*path*)
Retrieves a file by storage name and fileid in the form of a path

Path is expected to be storage_name/fileid.

classmethod make_middleware(*app*, ***options*)
Creates the application WSGI middleware in charge of serving local files.

A Depot middleware is required if your application wants to serve files from storages that don't directly provide and HTTP interface like *depot.io.local.LocalFileStorage* and *depot.io.gridfs.GridFSStorage*

classmethod set_default(*name*)
Replaces the current application default depot

classmethod url_for(*path*)
Given path of a file uploaded on depot returns the url that serves it

Path is expected to be storage_name/fileid.

class depot.middleware.**DepotMiddleware**(*app*, *mountpoint='/depot'*, *cache_max_age=604800*, *replace_wsgi_filewrapper=False*)
WSGI Middleware in charge of serving Depot files.

Usually created using *depot.manager.DepotManager.make_middleware()*, it's a WSGI middleware that serves files stored inside depots that do not provide a public HTTP url. For depot that provide a public url the request is redirected to the public url.

In case you have issues serving files with your WSGI server your can try to set replace_wsgi_filewrapper=True which forces DEPOT to use its own internal FileWrapper instead of the one provided by your WSGI server.

6.1.2 Database Support

class depot.fields.sqlalchemy.**UploadedFileField**(*filters=()*, *upload_type=<class 'depot.fields.upload.UploadedFile'>*, *upload_storage=None*, **args*, ***kw*)
Provides support for storing attachments to **SQLAlchemy** models.

UploadedFileField can be used as a Column type to store files into the model. The actual file itself will be uploaded to the default Storage, and only the *depot.fields.upload.UploadedFile* information will be stored on the database.

The UploadedFileField is transaction aware, so it will delete every uploaded file whenever the transaction is rolled back and will delete any old file whenever the transaction is committed. This is the reason you should never associate the same *depot.fields.upload.UploadedFile* to two different UploadedFileField, otherwise you might delete a file already used by another model. It is usually best to just set the file or bytes as content of the column and let the UploadedFileField create the *depot.fields.upload.UploadedFile* by itself whenever it's content is set.

impl
alias of sqlalchemy.sql.sqltypes.Unicode

load_dialect_impl(*dialect*)
Return a TypeEngine object corresponding to a dialect.

This is an end-user override hook that can be used to provide differing types depending on the given dialect. It is used by the TypeDecorator implementation of type_engine() to help determine what type should ultimately be returned for a given TypeDecorator.

By default returns self.impl.

process_bind_param(*value*, *dialect*)
Receive a bound parameter value to be converted.

Custom subclasses of _types.TypeDecorator should override this method to provide custom behaviors for incoming data values. This method is called at **statement execution time** and is passed the literal Python data value which is to be associated with a bound parameter in the statement.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

See also:

types_typedecorator

_types.TypeDecorator.process_result_value()

process_result_value(*value*, *dialect*)
Receive a result-row column value to be converted.

Custom subclasses of _types.TypeDecorator should override this method to provide custom behaviors for data values being received in result rows coming from the database. This method is called at **result fetching time** and is passed the literal Python data value that's extracted from a database result row.

The operation could be anything desired to perform custom behavior, such as transforming or deserializing data.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

See also:

types_typedecorator

_types.TypeDecorator.process_bind_param()

class depot.fields.ming.**UploadedFileProperty**(*filters=()*, *upload_type=<class 'depot.fields.upload.UploadedFile'>*, *upload_storage=None*)
Provides support for storing attachments to **Ming** MongoDB models.

UploadedFileProperty can be used as a field type to store files into the model. The actual file itself will be uploaded to the default Storage, and only the *depot.fields.upload.UploadedFile* information will be stored on the database.

The UploadedFileProperty is UnitOfWork aware, so it will delete every uploaded file whenever unit of work is flushed and deletes a Document that stored files or changes the field of a document storing files. This is the reason you should never associate the same *depot.fields.upload.UploadedFile* to two different UploadedFileProperty, otherwise you might delete a file already used by another document. It is usually best to just set the file or bytes as content of the column and let the UploadedFileProperty create the *depot.fields.upload.UploadedFile* by itself whenever it's content is set.

Warning: As the Ming UnitOfWork does not notify any event in case it gets cleared instead of being flushed all the files uploaded before clearing the unit of work will be already uploaded but won't have a document referencing them anymore, so DEPOT will be unable to delete them for you.

class depot.fields.ming.**DepotExtension**(*session*)
Extends the Ming Session to track files.

Deletes old files when an entry gets removed or replaced, apply this as a Ming SessionExtension according to Ming documentation.

after_flush(*obj=None*)
After the session is flushed for obj

If obj is None it means all the objects in the UnitOfWork which can be retrieved by iterating over ODMSession.uow

before_flush(*obj=None*)
Before the session is flushed for obj

If obj is None it means all the objects in the UnitOfWork which can be retrieved by iterating over ODMSession.uow

class depot.fields.interfaces.**DepotFileInfo**(*content*, *depot_name=None*)
Keeps information on a content related to a specific depot.

By itself the DepotFileInfo does nothing, it is required to implement a *process_content()* method that actually saves inside the file info the information related to the content. The only information which is saved by default is the depot name itself.

It is a specialized dictionary that provides also attribute style access, the dictionary parent permits easy encoding/decoding to most marshalling systems.

process_content(*content*, *filename=None*, *content_type=None*)
Process content in the given depot.

This is implemented by subclasses to provide some kind of behaviour on the content in the related Depot. The default implementation is provided by *depot.fields.upload.UploadedFile* which stores the content into the depot.

class depot.fields.interfaces.**FileFilter**
Interface that must be implemented by file filters.

File filters get executed whenever a file is stored on the database using one of the supported fields. Can be used to add additional data to the stored file or change it. When file filters are run the file has already been stored.

on_save(*uploaded_file*)
Filters are required to provide their own implementation

class depot.fields.upload.**UploadedFile**(*content*, *depot_name=None*)
Simple *depot.fields.interfaces.DepotFileInfo* implementation that stores files.

Takes a file as content and uploads it to the depot while saving around most file information. Pay attention that if the file gets replaced through depot manually the UploadedFile will continue to have the old data.

Also provides support for encoding/decoding using JSON for storage inside databases as a plain string.

Default attributes provided for all `UploadedFile` include:

- filename - This is the name of the uploaded file
- file_id - This is the ID of the uploaded file
- **path - This is a depot_name/file_id path which can** be used with `DepotManager.get_file()` to retrieve the file
- content_type - This is the content type of the uploaded file
- uploaded_at - This is the upload date in YYYY-MM-DD HH:MM:SS format
- url - Public url of the uploaded file
- file - The *depot.io.interfaces.StoredFile* instance of the stored file

process_content(*content*, *filename=None*, *content_type=None*)

Standard implementation of *DepotFileInfo.process_content()*

This is the standard depot implementation of files upload, it will store the file on the default depot and will provide the standard attributes.

Subclasses will need to call this method to ensure the standard set of attributes is provided.

Filters

class depot.fields.filters.thumbnails.**WithThumbnailFilter**(*size=(128, 128)*, *format='PNG'*)

Uploads a thumbnail together with the file.

Takes for granted that the file is an image. The resulting uploaded file will provide three additional properties named:

- `thumb_X_id` -> The depot file id
- `thumb_X_path` -> Where the file is available in depot
- `thumb_X_url` -> Where the file is served.

Where `X` is the resolution specified as `size` in the filter initialization. By default this is `(128, 128)` ?so you will get `thumb_128x128_id`, `thumb_128x128_url` and so on.

Warning: Requires Pillow library

Specialized FileTypes

class depot.fields.specialized.image.**UploadedImageWithThumb**(*content*, *depot_name=None*)

Uploads an Image with thumbnail.

The default thumbnail format and size are PNG@128x128, those can be changed by inheriting the `UploadedImageWithThumb` and replacing the `thumbnail_format` and `thumbnail_size` class properties.

The Thumbnail file is accessible as `.thumb_file` while the thumbnail url is `.thumb_url`.

> **Warning:** Requires Pillow library

6.1.3 Storing Files

class depot.io.interfaces.**StoredFile**(*file_id*, *filename=None*, *content_type=None*, *last_modified=None*, *content_length=None*)

Interface for already saved files.

It provides metadata on the stored file through following properties:

- file_id
- filename
- content_type
- last_modified
- content_length

Already stored files can only be read back, so they are required to only provide `read(self, n=-1)`, `close()` methods and `closed` property so that they can be read.

To replace/overwrite a file content do not try to call the `write` method, instead use the storage backend to replace the file content.

close(**args*, ***kwargs*)
Closes the file.

After closing the file it won't be possible to read from it anymore. Some implementation might not do anything when closing the file, but they still are required to prevent further reads from a closed file.

closed
Returns if the file has been closed.

When `closed` return `True` it won't be possible to read anoymore from this file.

fileno()
Returns underlying file descriptor if one exists.

An IOError is raised if the IO object does not use a file descriptor.

flush()
Flush write buffers, if applicable.

This is not implemented for read-only and non-blocking streams.

isatty()
Return whether this is an 'interactive' stream.

Return False if it can't be determined.

name
This is the filename of the saved file

If a filename was not available when the file was created this will return "unnamed" as filename.

next

public_url
The public HTTP url from which file can be accessed.

When supported by the storage this will provide the public url to which the file content can be accessed. In case this returns `None` it means that the file can only be served by the `DepotMiddleware` itself.

read(*n=-1*)
Reads `n` bytes from the file.

If `n` is not specified or is `-1` the whole file content is read in memory and returned

readable()
Returns if the stored file is readable or not

Usually all stored files are readable

readline()
Read and return a line from the stream.

If limit is specified, at most limit bytes will be read.

The line terminator is always b'n' for binary files; for text files, the newlines argument to open can be used to select the line terminator(s) recognized.

readlines()
Return a list of lines from the stream.

hint can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds hint.

seek()
Change stream position.

Change the stream position to the given byte offset. The offset is interpreted relative to the position indicated by whence. Values for whence are:

- 0 – start of stream (the default); offset should be zero or positive
- 1 – current stream position; offset may be negative
- 2 – end of stream; offset is usually negative

Return the new absolute position.

seekable()
Returns if the stored file is seekable or not

By default stored files are not seekable

tell()
Return current stream position.

truncate()
Truncate file to size bytes.

File pointer is left unchanged. Size defaults to the current IO position as reported by tell(). Returns the new size.

writable()
Returns if the stored file is writable or not

Stored files are not writable, you should rely on the relative *FileStorage* to overwrite their content

class depot.io.interfaces.**FileStorage**
Interface for storage providers.

The FileStorage base class declares a standard interface for storing and retrieving files in an underlying storage system.

Each storage system implementation is required to provide this interface to correctly work with filedepot.

create(*content*, *filename=None*, *content_type=None*)
Saves a new file and returns the ID of the newly created file.

`content` parameter can either be `bytes`, another `file object` or a `cgi.FieldStorage`. When `filename` and `content_type` parameters are not provided they are deducted from the content itself.

delete(*file_or_id*)
Deletes a file. If the file didn't exist it will just do nothing.

exists(*file_or_id*)
Returns if a file or its ID still exist.

static fileid(*file_or_id*)
Gets the ID of a given *StoredFile*

If the given parameter is already a StoredFile id it will directly return it.

static fileinfo(*fileobj*, *filename=None*, *content_type=None*, *existing=None*)
Tries to extract from the given input the actual file object, filename and content_type

This is used by the create and replace methods to correctly deduce their parameters from the available information when possible.

get(*file_or_id*)
Opens the file given by its unique id.

This operation is guaranteed to return a *StoredFile* instance or should raise `IOError` if the file is not found.

list()
Returns a list of file IDs that exist in the Storage.

Depending on the implementation there is the possibility that this returns more IDs than there have been created. Therefore this method is NOT guaranteed to be RELIABLE.

replace(*file_or_id*, *content*, *filename=None*, *content_type=None*)
Replaces an existing file, an `IOError` is raised if the file didn't already exist.

Given a *StoredFile* or its ID it will replace the current content with the provided `content` value. If `filename` and `content_type` are provided or can be deducted by the `content` itself they will also replace the previous values, otherwise the current values are kept.

class depot.io.local.**LocalFileStorage**(*storage_path*)
depot.io.interfaces.FileStorage implementation that stores files locally.

All the files are stored inside a directory specified by the `storage_path` parameter.

create(*content*, *filename=None*, *content_type=None*)
Saves a new file and returns the ID of the newly created file.

`content` parameter can either be `bytes`, another `file object` or a `cgi.FieldStorage`. When `filename` and `content_type` parameters are not provided they are deducted from the content itself.

delete(*file_or_id*)
Deletes a file. If the file didn't exist it will just do nothing.

exists(*file_or_id*)
Returns if a file or its ID still exist.

get(*file_or_id*)
Opens the file given by its unique id.

This operation is guaranteed to return a `StoredFile` instance or should raise `IOError` if the file is not found.

list()
Returns a list of file IDs that exist in the Storage.

Depending on the implementation there is the possibility that this returns more IDs than there have been created. Therefore this method is NOT guaranteed to be RELIABLE.

replace(*file_or_id*, *content*, *filename=None*, *content_type=None*)
Replaces an existing file, an `IOError` is raised if the file didn't already exist.

Given a `StoredFile` or its ID it will replace the current content with the provided `content` value. If `filename` and `content_type` are provided or can be deducted by the `content` itself they will also replace the previous values, otherwise the current values are kept.

class depot.io.gridfs.**GridFSStorage**(*mongouri*, *collection='filedepot'*)
depot.io.interfaces.FileStorage implementation that stores files on MongoDB.

All the files are stored using GridFS to the database pointed by the `mongouri` parameter into the collection named `collection`.

create(*content*, *filename=None*, *content_type=None*)
Saves a new file and returns the ID of the newly created file.

`content` parameter can either be `bytes`, another `file object` or a `cgi.FieldStorage`. When `filename` and `content_type` parameters are not provided they are deducted from the content itself.

delete(*file_or_id*)
Deletes a file. If the file didn't exist it will just do nothing.

exists(*file_or_id*)
Returns if a file or its ID still exist.

get(*file_or_id*)
Opens the file given by its unique id.

This operation is guaranteed to return a `StoredFile` instance or should raise `IOError` if the file is not found.

list()
Returns a list of file IDs that exist in the Storage.

Depending on the implementation there is the possibility that this returns more IDs than there have been created. Therefore this method is NOT guaranteed to be RELIABLE.

replace(*file_or_id*, *content*, *filename=None*, *content_type=None*)
Replaces an existing file, an `IOError` is raised if the file didn't already exist.

Given a `StoredFile` or its ID it will replace the current content with the provided `content` value. If `filename` and `content_type` are provided or can be deducted by the `content` itself they will also replace the previous values, otherwise the current values are kept.

class depot.io.boto3.**S3Storage**(*access_key_id*, *secret_access_key*, *bucket=None*, *region_name=None*, *policy=None*, *storage_class=None*, *endpoint_url=None*, *prefix=''*)
depot.io.interfaces.FileStorage implementation that stores files on S3.

This is a version implemented on top of boto3. Installing `boto3` as a dependency is required to use this.

All the files are stored inside a bucket named `bucket` on `host` which Depot connects to using `access_key_id` and `secret_access_key`.

Additional options include:

- `region` which can be used to specify the AWS region.
- `endpoint_url` which can be used to specify an host different from Amazon AWS S3 Storage
- `policy` which can be used to specify a canned ACL policy of either `private` or `public-read`.
- `storage_class` which can be used to specify a class of storage.
- `prefix` parameter can be used to store all files under specified prefix. Use a prefix like **dirname/** (*see trailing slash*) to store in a subdirectory.

create(*content*, *filename=None*, *content_type=None*)
Saves a new file and returns the ID of the newly created file.

`content` parameter can either be `bytes`, another `file object` or a `cgi.FieldStorage`. When `filename` and `content_type` parameters are not provided they are deducted from the content itself.

delete(*file_or_id*)
Deletes a file. If the file didn't exist it will just do nothing.

exists(*file_or_id*)
Returns if a file or its ID still exist.

get(*file_or_id*)
Opens the file given by its unique id.

This operation is guaranteed to return a `StoredFile` instance or should raise `IOError` if the file is not found.

list()
Returns a list of file IDs that exist in the Storage.

Depending on the implementation there is the possibility that this returns more IDs than there have been created. Therefore this method is NOT guaranteed to be RELIABLE.

replace(*file_or_id*, *content*, *filename=None*, *content_type=None*)
Replaces an existing file, an `IOError` is raised if the file didn't already exist.

Given a `StoredFile` or its ID it will replace the current content with the provided `content` value. If `filename` and `content_type` are provided or can be deducted by the `content` itself they will also replace the previous values, otherwise the current values are kept.

class depot.io.awss3.**S3Storage**(*access_key_id*, *secret_access_key*, *bucket=None*, *host=None*, *policy=None*, *encrypt_key=False*, *prefix=''*)
depot.io.interfaces.FileStorage implementation that stores files on S3.

This is a version implemented on boto Installing `boto` as a dependency is required to use this.

All the files are stored inside a bucket named `bucket` on `host` which Depot connects to using `access_key_id` and `secret_access_key`.

Additional options include:

- `host` which can be used to specify an host different from Amazon AWS S3 Storage
- `policy` which can be used to specify a canned ACL policy of either `private` or `public-read`.
- `encrypt_key` which can be specified to use the server side encryption feature.
- `prefix` parameter can be used to store all files under specified prefix. Use a prefix like **dirname/** (*see trailing slash*) to store in a subdirectory.

create(*content*, *filename=None*, *content_type=None*)
Saves a new file and returns the ID of the newly created file.

`content` parameter can either be `bytes`, another `file object` or a `cgi.FieldStorage`. When `filename` and `content_type` parameters are not provided they are deducted from the content itself.

delete(*file_or_id*)
Deletes a file. If the file didn't exist it will just do nothing.

exists(*file_or_id*)
Returns if a file or its ID still exist.

get(*file_or_id*)
Opens the file given by its unique id.

This operation is guaranteed to return a `StoredFile` instance or should raise `IOError` if the file is not found.

list()
Returns a list of file IDs that exist in the Storage.

Depending on the implementation there is the possibility that this returns more IDs than there have been created. Therefore this method is NOT guaranteed to be RELIABLE.

replace(*file_or_id*, *content*, *filename=None*, *content_type=None*)
Replaces an existing file, an `IOError` is raised if the file didn't already exist.

Given a `StoredFile` or its ID it will replace the current content with the provided `content` value. If `filename` and `content_type` are provided or can be deducted by the `content` itself they will also replace the previous values, otherwise the current values are kept.

class depot.io.memory.**MemoryFileStorage**(***kwargs*)
depot.io.interfaces.FileStorage implementation that keeps files in memory.

This is generally useful for caches and tests.

create(*content*, *filename=None*, *content_type=None*)
Saves a new file and returns the ID of the newly created file.

`content` parameter can either be `bytes`, another `file object` or a `cgi.FieldStorage`. When `filename` and `content_type` parameters are not provided they are deducted from the content itself.

delete(*file_or_id*)
Deletes a file. If the file didn't exist it will just do nothing.

exists(*file_or_id*)
Returns if a file or its ID still exist.

get(*file_or_id*)
Opens the file given by its unique id.

This operation is guaranteed to return a `StoredFile` instance or should raise `IOError` if the file is not found.

list()
Returns a list of file IDs that exist in the Storage.

Depending on the implementation there is the possibility that this returns more IDs than there have been created. Therefore this method is NOT guaranteed to be RELIABLE.

replace(*file_or_id*, *content*, *filename=None*, *content_type=None*)
Replaces an existing file, an `IOError` is raised if the file didn't already exist.

Given a `StoredFile` or its ID it will replace the current content with the provided `content` value. If `filename` and `content_type` are provided or can be deducted by the `content` itself they will also replace the previous values, otherwise the current values are kept.

6.1.4 Utilities

depot.io.utils.**file_from_content**(*content*)
Provides a real file object from file content

Converts `FileStorage`, `FileIntent` and `bytes` to an actual file.

class depot.io.utils.**FileIntent**(*fileobj*, *filename*, *content_type*)
Represents the intention to upload a file

Whenever a file can be stored by depot, a FileIntent can be passed instead of the file itself. This permits to easily upload objects that are not files or to add missing information to the uploaded files.

Python Module Index

d

Index