SQLAlchemy-ImageAttach Documentation

Release 0.8.1

Hong Minhee

August 25, 2013

CONTENTS

SQLAlchemy-ImageAttach is a SQLAlchemy extension for attaching images to entity objects. It provides the following features:

Storage backend interface You can use file system backend on your local development box, and switch it to AWS S3 when it's deployed to the production box. Or you can add a new backend implementation by yourself.

Maintaining multiple image sizes Any size of thumbnails can be generated from the original size without assuming the fixed set of sizes. You can generate a thumbnail of a particular size if it doesn't exist yet when the size is requested. Use RRS (Reduced Redundancy Storage) for reproducible thumbnails on S3.

Every image has its URL Attached images can be exposed as a URL.

SQLAlchemy transaction aware Saved file are removed when the ongoing transaction has been rolled back.

Tested on various environments

- Python versions: Python 2.6, 2.7, 3.2, 3.3, PyPy
- DBMS: PostgreSQL, MySQL, SQLite
- SQLAlchemy: 0.8 or higher

CHAPTER

ONE

INSTALLATION

It's already available on PyPI, so just use **pip**:

```
$ pip install SQLAlchemy-ImageAttach
```

CHAPTER

TWO

USER'S GUIDE

2.1 Declaring Image Entities

It's easy to use with `sqlalchemy.ext.declarative`:

```
from sqlalchemy import Column, ForeignKey, Integer, Unicode, relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy_imageattach.entity import Image, image_attachment


Base = declarative_base()


class User(Base):
    """User model."""

    id = Column(Integer, primary_key=True)
    name = Column(Unicode, nullable=False)
    picture = image_attachment('UserPicture')
    __tablename__ = 'user'


class UserPicture(Base, Image):
    """User picture model."""

    user_id = Column(Integer, ForeignKey('User.id'), primary_key=True)
    user = relationship('User')
    __tablename__ = 'user_picture'
```

In the above example, we declare two entity classes. `UserPicture` which inherits `Image` is an image entity, and `User` owns it. `image_attachment()` function is a specialized `relationship()` for image entities. You can understand it as one-to-many relationship.

2.1.1 Object type

Every image class has `object_type` string, which is used by the storage.

`UserPicture` in the above example omits `object_type` property, but it can be overridden if needed. Its default value is the table name (underscores will be replaced by hyphens).

When would you need to override `object_type`? The most common case is when you changed the table name. Identifiers like path names that are internally used by the stoage won't be automatically renamed even if you change the table name in the relational database. So you need to maintain the same `object_type` value.

2.1.2 Object identifier

Every image instance has `object_id` number, which is used by the storage. A pair of (`object_type`, `object_id` is an unique key for an image.

`UserPicture` in the above example omits `object_id` property, because it provides the default value when the primary key is integer. It has to be explicitly implemented when the primary key is not integer or composite key.

For example, the most simple and easiest (although naive) way to implement `object_id` for the string primary key is hashing it:

```
@property
def object_id(self):
    return int(hashlib.sha1(self.id).hexdigest(), 16)
```

If the primary key is a pair, encode a pair into an integer:

```
@property
def object_id(self):
    a = self.id_a
    b = selg.id_b
    return (a + b) * (a + b) + a
```

If the primary key is composite of three or more columns, encode a tuple into a linked list of pairs first, and then encode the pair into an integer. It's just a way to encode, and there are many other ways to do the same.

2.2 Storages

2.2.1 Choosing the right storage implementation

There are currently only two implementations:

- `sqlalchemy_imageattach.stores.fs`
- `sqlalchemy_imageattach.stores.s3`

We recommend you to use `fs` on your local development box, and switch it to `s3` when you deploy it to the production system.

If you need to use another storage backend, you can implement the interface by yourself: *Implementing your own storage*.

2.2.2 Using filesystem on the local development box

The most of computers have a filesystem, so using `fs` storage is suitable for development. It works even if you are offline.

Actually there are two kinds of filesystem storages:

`FileSystemStore` It just stores the images, and simply assumes that you have a separate web server for routing static files e.g. Lighttpd, Nginx. For example, if you have a sever configuration like this:

```
server {
    listen 80;
    server_name images.yourapp.com;
    root /var/local/yourapp/images;
}
```

`FileSystemStore` should be configured like this:

```
sqlalchemy_imageattach.stores.fs.FileSystemStore(
    path='/var/local/yourapp/images',
    base_url='http://images.yourapp.com/'
)
```

`HttpExposedFileSystemStore` In addition to `FileSystemStore`'s storing features, it does more for you: actually serving files through WSGI. It takes an optional `prefix` for url instead of `base_url`:

```
sqlalchemy_imageattach.stores.fs.HttpExposedFileSystemStore(
    path='/var/local/yourapp/images',
    prefix='static/images/'
)
```

The default `prefix` is simply `images/`.

It provides `wsgi_middleware()` method to inject its own server to your WSGI application. For example, if you are using Flask:

```
from yourapp import app
app.wsgi_app = store.wsgi_middleware(app.wsgi_app)
```

or if Pyramid:

```
app = config.make_wsgi_app()
app = store.wsgi_middleware(app)
```

or if Bottle:

```
app = bottle.app()
app = store.wsgi_middleware(app)
```

Note: The server provided by this isn't production-ready quality, so do not use this for your production service. We recommend you to use `FileSystemStore` with a separate web server like Nginx or Lighttpd instead.

2.2.3 Implementing your own storage

You can implement a new storage backend if you need. Every storage has to inherit `Store` and implement the following four methods:

`put_file()` The method puts a given image to the storage.

It takes a `file` that contains the image blob, four identifier values (`object_type`, `object_id`, `width`, `height`) for the image, a `mimetype` of the image, and a boolean value (`reproducible`) which determines whether it can be reproduced or not.

For example, if it's a filesystem storage, you can make directory/file names using `object_type`, `object_id`, and size values, and suffix using `mimetype`. If it's a S3 implementation, it can determine whether to use RRS (reduced redundancy storage) or standard storage using `reproducible` argument.

`get_file()` The method finds a requested image in the storage.

It takes four identifier values (`object_type`, `object_id`, `width`, `height`) for the image, and a `mimetype` of the image. The return type must be file-like.

It should raise `IOError` or its subtype when there's no requested image in the storage.

get_url() The method is similar to `get_file()` except it returns a URL of the image instead of a file that contains the image blob.

It doesn't have to raise errors when there's no requested image in the storage. It's okay even if the returned URL is a broken link. Because we assume that it's called only when the requested image is sure to be there. It means you can quickly generate URLs by just calculation without any I/O.

Moreover, you can assume that these URLs are never cached, because SQLAlchemy-ImageAttach will automatically appends a query string that contains of its updated timestamp for you.

delete_file() The method deletes a requested image in the storage.

It takes the same arguments to `get_file()` and `get_url()` methods.

It must doesn't raise any exception even if there's no requested image.

The constructor of it can be anything. It's not part of the interface.

If you believe your storage implementation could be widely used as well as for others, please contribute your code by sending a pull request! We always welcome your contributions.

2.2.4 Migrating storage

SQLAlchemy-ImageAttach provides a simple basic utility to migrate image data in an old storage to a new storage (although it's not CLI but API). In order to migrate storage data you need used database as well, not only storage. Because some metadata are only saved to database.

The following code shows you how to migrate all image data in `old_store` to `new_store`:

```
plan = migrate(session, Base, old_store, new_store)
plan.execute()
```

In the above code, `Base` is declarative base class (which is created by `sqlalchemy.ext.declarative.declarative_base()`), and `session` is an instance of SQLAlchemy `Session`.

If you want to know progress of migration, iterating the result:

```
plan = migrate(session, Base, old_store, new_store)
for image in plan:
    print('Migrated ' + repr(image))
```

Or pass a `callback` function to `execute()` method:

```
def progress(image):
    print('Migrated ' + repr(image))

plan = migrate(session, Base, old_store, new_store)
plan.execute(progress)
```

2.3 Attaching Images

You've *declared entities* and *choose a storage*, so then the next step is to actually attach images to objects! In order to determine what storage to save images into, you can set the current *context*.

2.3.1 Context

A context knows what storage you are using now, and tell entities the storage to use. You can set a context using `store_context()` function in `with` block:

```
from sqlalchemy_imageattach.context import store_context

with store_context(store):
    with open('image_to_attach.jpg') as f:
        entity.picture.from_file(f)
```

You would face `ContextError` when you try attaching images without any context.

2.3.2 Attaching from file object

A way to attach an image to an object is loading it from a file object using `from_file()` method. The following example code shows how to attach a profile picture to an user:

```
from yourapp.config import session, store

def set_picture(request, user_id):
    try:
        user = session.query(User).get(int(user_id))
        with store_context(store):
            user.picture.from_file(request.files['picture'])
    except Exception:
        session.rollback()
        raise
    session.commit()
```

It takes any file-like objects as well e.g.:

```
from urllib2 import urlopen

def set_picture_url(request, user_id):
    try:
        user = session.query(User).get(int(user_id))
        picture_url = request.values['picture_url']
        with store_context(store):
            user.picture.from_file(urlopen(picture_url))
    except Exception:
        session.rollback()
        raise
    session.commit()
```

Note that the responsibility to close files is yours. Because some file-like objects can be reused several times, or don't have to be closed (or some of them even don't have any `close()` method).

2.3.3 Attaching from byte string

Of course you can load images from its byte strings. Use `from_blob()` method:

```
from requests import get

def set_picture_url(request, user_id):
    try:
        user = session.query(User).get(int(user_id))
```

```
        picture_url = request.values['picture_url']
        image_binary = get(picture_url).content
        with store_context(store):
            user.picture.from_blob(image_binary)
    except Exception:
        session.rollback()
        raise
    session.commit()
```

2.3.4 Getting image urls

In web environment, the most case you need just an url of an image, not its binary content. So `ImageSet` object provide `locate()` method:

```
def user_profile(request, user_id):
    user = session.query(User).get(int(user_id))
    with store_context(store):
        picture_url = user.picture.locate()
    return render_template('user_profile.html',
                           user=user, picture_url=picture_url)
```

It returns the url of the original image (which is not resized). Read about *Thumbnails* if you want a thumbnail url.

`ImageSet` also implements de facto standard `__html__()` special method, so it can be directly rendered in the most of template engines like Jinja2, Mako. It's expanded to `<img>` tag on templates:

```
<div class="user">
    <a href="{{ url_for('user_profile', user_id=user.id) }}"
       title="{{ user.name }}">{{ user.picture }}</a>
</div>

<div class="user">
    <a href="${url_for('user_profile', user_id=user.id)}"
       title="${user.name}">${user.picture}</a>
</div>
```

The above template codes are equivalent to:

```
<div class="user">
    <a href="{{ url_for('user_profile', user_id=user.id) }}"
       title="{{ user.name }}"><img src="{{ user.picture.locate() }}"
                                    width="{{ user.picture.width }}"
                                    height="{{ user.picture.height }}"></a>
</div>

<div class="user">
    <a href="${url_for('user_profile', user_id=user.id)}"
       title="${user.name}"><img src="${user.picture.locate()}"
                                 width="${user.picture.width}"
                                 height="${user.picture.height}"></a>
</div>
```

Note: Template expansion of `ImageSet` might raise `ContextError`. You should render the template in the context:

```
with store_context(store):
    return render_template('user_profile.html', user=user)
```

Or use *Implicit contexts*.

2.3.5 Getting image files

`ImageSet` provides `open_file()` method. It returns a file-like object:

```
from shutil import copyfileobj

with store_context(store):
    with user.picture.open_file() as f:
        copyfileobj(f, dst)
```

Note that the responsibility to close an opened file is yours. Recommend to open it in `with` block.

2.3.6 Getting image binary

There's a shortcut to read byte string from an opened file. Use `make_blob()` method. The following two ways are equivalent:

```
# make_blob()
with store_context(store):
    blob = user.picture.make_blob()

# open().read()
with store_context(store):
    with user.picture.open_file() as f:
        blob = f.read()
```

2.3.7 Thumbnails

You can make thumbnails and then store them into the store using `generate_thumbnail()` method. It takes one of three arguments: `width`, `height`, or `ratio`:

```
with store_context(store):
    # Make thumbnails
    width_150 = user.picture.generate_thumbnail(width=150)
    height_300 = user.picture.generate_thumbnail(height=300)
    half = user.picture.generate_thumbnail(ratio=0.5)
    # Get their urls
    width_150_url = width_150.locate()
    height_300_url = width_300.locate()
    half = half.locate()
```

It returns a made `Image` object, and it shares the most of the same methods to `ImageSet` like `locate()`, `open_file()`, `make_blob()`.

Once made thumbnails can be found using `find_thumbnail()`. It takes one of two arguments: `width` or `height` and returns a found `Image` object:

```
with store_context(store):
    # Find thumbnails
    width_150 = user.picture.find_thumbnail(width=150)
    height_300 = user.picture.find_thumbnail(height=300)
    # Get their urls
    width_150_url = width_150.locate()
    height_300_url = width_300.locate()
```

It raises `NoResultFound` exception when there's no such size.

You can implement find-or-create pattern using these two methods:

```
def find_or_create(imageset, width=None, height=None):
    assert width is not None or height is not None
    try:
        image = imageset.find_thumbnail(width=width, height=height)
    except NoResultFound:
        image = imageset.generate_thumbnail(width=width, height=height)
    return image
```

We recommend you to queue generating thumbnails and make it done by backend workers rather than web applications. There are several tools for that like Celery.

2.3.8 Expliciting storage

It's so ad-hoc, but there's a way to explicit storage to use without any context: passing the storage to operations as an argument. Every methods that need the context also optionally take `store` keyword:

```
user.picture.from_file(file_, store=store)
user.picture.from_blob(blob, store=store)
user.picture.locate(store=store)
user.picture.open_file(store=store)
user.picture.make_blob(store=store)
user.picture.generate_thumbnail(width=150, store=store)
user.picture.find_thumbnail(width=150, store=store)
```

The above calls are all equivalent to the following calls in `with` block:

```
with store_context(store):
    user.picture.from_file(file_)
    user.picture.from_blob(blob)
    user.picture.locate()
    user.picture.open_file()
    user.picture.make_blob()
    user.picture.generate_thumbnail(width=150)
    user.picture.find_thumbnail(width=150)
```

2.3.9 Implicit contexts

If your application already manage some context like request-response lifecycle, you can make context implicit by utilizing these hooks. SQLAlchemy-ImageAttach exposes underlayer functions like `push_store_context()` and `pop_store_context()` that are used for implementing `store_context()`.

For example, use `before_request()` and `teardown_request()` if you are using Flask:

```
from sqlalchemy_imageattach.context import (pop_store_context,
                                            push_store_context)
from yourapp import app
from yourapp.config import store

@app.before_request
def start_implicit_store_context():
    push_store_context(store)

@app.teardown_request
def stop_implicit_store_context(exception=None):
    pop_store_context()
```

2.4 SQLAlchemy-ImageAttach Changelog

2.4.1 Version 0.8.1

Released on August 26, 2013.

- Added `sqlalchemy_imageattach.migration` module for storage migration. See also *Migrating storage* guide.
- Added `public_base_url` option to `S3Store`. It's useful when used with CDN e.g. CloudFront.

2.4.2 Version 0.8.0

Released on June 20, 2013.

- Support Python 3.2 and 3.3. (Required minimum version of Wand also becomes 0.3.0 from 0.2.0.)
- Added manual `push_store_context()` and `pop_store_context()` API. It's useful when you can't use `with` keyword e.g. setup/teardown hooks.
- `Image.object_type` property now has the default value when the primary key is an integer.
- Columns of `Image` class become able to be used as SQL expressions.
- Added `block_size` option to `StaticServerMiddleware`.
- `StaticServerMiddleware` now supports `'wsgi.file_wrapper'`. See also optional platform-specific file handling.

2.4.3 Version 0.8.0.dev-20130531

Initially released on May 31, 2013.

CHAPTER

THREE

SQLALCHEMY_IMAGEATTACH — API

3.1 sqlalchemy_imageattach.context — Scoped context of image storage

Scoped context makes other modules able to vertically take an image store object without explicit parameter for it. It's similar to Flask's design decision and Werkzeug's context locals. Context locals are workaround to use dynamic scoping in programming languages that doesn't provide it (like Python).

For example, a function can take an image store to use as its parameter:

```
def func(store):
    url = store.locate(image)
    # ...

func(fs_store)
```

But, what if for various reasions it can't take an image store as parameter? You should vertically take it using scoped context:

```
def func():
    current_store.locate(image)

with store_context(fs_store):
    func()
```

What if you have to pass the another store to other subroutine?:

```
def func(store):
    decorated_store = DecoratedStore(store)
    func2(decorated_store)

def func2(store):
    url = store.locate(image)
    # ...

func(fs_store)
```

The above code can be rewritten using scoped context:

```
def func():
    decorated_store = DecoratedStore(current_store)
    with store_context(decorated_store):
        func2()
```

```
def func2():
    url = current_store.locate(image)
    # ...

with store_context(fs_store):
    func()
```

exception sqlalchemy_imageattach.context.**ContextError**
The exception which rises when the current_store is required but there's no currently set store context.

class sqlalchemy_imageattach.context.**LocalProxyStore**(*get_current_object*, *repr_string=None*)
Proxy of another image storage.

Parameters

- **get_current_object** (collections.Callable) – a function that returns "current" store
- **repr_string** (str) – an optional string for repr()

sqlalchemy_imageattach.context.**context_stacks** = {}
(dict) The dictionary of concurrent contexts to their stacks.

sqlalchemy_imageattach.context.**current_store** = sqlalchemy_imageattach.context.current_store
(LocalProxyStore) The currently set context of the image store backend. It can be set using store_context().

sqlalchemy_imageattach.context.**get_current_context_id**()
Identifis which context it is (greenlet, stackless, or thread).

Returns the identifier of the current context.

sqlalchemy_imageattach.context.**get_current_store**()
The lower-level function of current_store. It returns the **actual** store instance while current_store is a just proxy of it.

Returns the actual object of the currently set image store

Return type Store

sqlalchemy_imageattach.context.**pop_store_context**()
Manually pops the current store from the stack.

Although store_context() and with keyword are preferred than using it, it's useful when you have to push and pop the current stack on different hook functions like setup/teardown.

Returns the current image store

Return type Store

sqlalchemy_imageattach.context.**push_store_context**(*store*)
Manually pushes a store to the current stack.

Although store_context() and with keyword are preferred than using it, it's useful when you have to push and pop the current stack on different hook functions like setup/teardown.

Parameters **store** (Store) – the image store to set to the current_store

sqlalchemy_imageattach.context.**store_context**(**args*, ***kwds*)
Sets the new (nested) context of the current image storage:

```
with store_context(store):
    print current_store
```

It could be set nestedly as well:

```
with store_context(store1):
    print current_store  # store1
    with store_context(store2):
        print current_store  # store2
    print current_store  # store1 back
```

Parameters **store** (`Store`) – the image store to set to the `current_store`

3.2 `sqlalchemy_imageattach.entity` — Image entities

This module provides a short way to attach resizable images to other object-relationally mapped entity classes.

For example, imagine there's a fictional entity named `User` and it has its `picture` and `front_cover`. So there should be two image entities that subclass `Image` mixin:

```
class UserPicture(Base, Image):
    '''User's profile picture.'''

    user_id = Column(Integer, ForeignKey('User.id'), primary_key=True)
    user = relationship('User')

    __tablename__ = 'user_picture'
```

You have to also inherit your own `declarative_base()` class (`Base` in the example).

Assume there's also `UserFrontCover` in the same way.

Note that the class can override `object_id` property. Backend storages utilize this to identify images e.g. filename, S3 key. If the primary key of the image entity is integer, `object_id` automatically uses the primary key value by default, but it can be overridden if needed, and must be implemented if the primary key is not integer or composite key.

There's also `object_type` property. `Image` provides the default value for it as well. It uses the class name (underscores will be replaced by hyphens) by default, but you can override it.

These `Image` subclasses can be related to the their 'parent' entity using `image_attachment()` function. It's a specialized version of SQLAlchemy's built-in `relationship()` function, so you can pass the same options as `relationship()` takes:

```
class User(Base):
    '''Users have their profile picture and front cover.'''

    id = Column(Integer, primary_key=True)
    picture = image_attachment('UserPicture')
    front_cover = image_attachment('UserFrontCover')

    __tablename__ = 'user'
```

It's done, you can store the actual image files using `ImageSet.from_file()` or `ImageSet.from_blob()` method:

```
with store_context(store):
    user = User()
    with open('picture.jpg', 'rb') as f:
        user.picture.from_blob(f.read())
```

```
    with open('front_cover.jpg', 'rb') as f:
        user.front_cover.from_file(f)
    with session.begin():
        session.add(user)
```

Or you can resize the image to make thumbnails using `ImageSet.generate_thumbnail()` method:

```
with store_context(store):
    user.picture.generate_thumbnail(ratio=0.5)
    user.picture.generate_thumbnail(height=100)
    user.front_cover.generate_thumbnail(width=500)
```

class sqlalchemy_imageattach.entity.**Image**

The image of the particular size.

Note that it implements `__html__()` method, a de facto standard special method for HTML templating. So you can simply use it in HTML templates like:

```
{{ user.profile.find_thumbnail(120) }}
```

The above template is equivalent to:

```
{% with thumbnail = user.profile.find_thumbnail(120) %}
    <img src="{{ thumbnail.locate() }}"
         width="{{ thumbnail.width }}"
         height="{{ thumbnail.height }}">
{% endwith %}
```

object_type

(basestring) The identifier string of the image type. It uses `__tablename__` (which replaces underscores with hyphens) by default, but can be overridden.

created_at = Column('created_at', DateTime(timezone=True), table=None, nullable=False, default=ColumnDefault(<s

(datetime.datetime) The created time.

height = Column('height', Integer(), table=None, primary_key=True, nullable=False)

(numbers.Integral) The image's height."""

locate(*store=sqlalchemy_imageattach.context.current_store*)

Gets the URL of the image from the `store`.

Parameters **store** (`Store`) – the storage which contains the image. `current_store` by default

Returns the url of the image

Return type basestring

make_blob(*store=sqlalchemy_imageattach.context.current_store*)

Gets the byte string of the image from the `store`.

Parameters **store** (`Store`) – the storage which contains the image. `current_store` by default

Returns the binary data of the image

Return type str

mimetype = Column('mimetype', String(length=255), table=None, nullable=False)

(basestring) The mimetype of the image e.g. `'image/jpeg'`, `'image/png'`.

object_id
(numbers.Integral) The identifier number of the image. It uses the primary key if it's integer, but can be overridden, and must be implemented when the primary key is not integer or composite key.

open_file(*store=sqlalchemy_imageattach.context.current_store*, *use_seek=False*)
Opens the file-like object which is a context manager (that means it can used for `with` statement).

If `use_seek` is `True` (though `False` by default) it guarentees the returned file-like object is also seekable (provides `seek()` method).

Parameters **store** (`Store`) – the storage which contains image files. `current_store` by default

Returns the file-like object of the image, which is a context manager (plus, also seekable only if `use_seek` is `True`)

Return type `file`, `FileProxy`, file-like object

original = Column('original', Boolean(), table=None, nullable=False, default=ColumnDefault(False))
(`bool`) Whether it is original or resized.

width = Column('width', Integer(), table=None, primary_key=True, nullable=False)
(numbers.Integral) The image's width.

class sqlalchemy_imageattach.entity.**ImageSet**(*entities*, *session=None*)
The subtype of `Query` specialized for `Image`. It provides more methods and properties over `Query`.

Note that it implements `__html__()` method, a de facto standard special method for HTML templating. So you can simply use it in Jinja2 like:

```
{{ user.profile }}
```

instead of:

```
<img src="{{ user.profile|permalink }}"
     width="{{ user.profile.original.width }}"
     height="{{ user.profile.original.height }}">
```

find_thumbnail(*width=None*, *height=None*)
Finds the thumbnail of the image with the given `width` and/or `height`.

Parameters

- **width** (numbers.Integral) – the thumbnail width
- **height** (numbers.Integral) – the thumbnail height

Returns the thumbnail image

Return type `Image`

Raises **sqlalchemy.orm.exc.NoResultFound** when there's no image of such size

from_blob(*blob*, *store=sqlalchemy_imageattach.context.current_store*)
Stores the `blob` (byte string) for the image into the `store`.

Parameters

- **blob** (`str`) – the byte string for the image
- **store** (`Store`) – the storage to store the image data. `current_store` by default

Returns the created image instance

Return type `Image`

from_file(*file*, *store=sqlalchemy_imageattach.context.current_store*)
Stores the `file` for the image into the `store`.

Parameters

- **file** (file-like object, `file`) – the readable file of the image
- **store** (`Store`) – the storage to store the file. `current_store` by default

Returns the created image instance

Return type `Image`

from_raw_file(*raw_file*, *store=sqlalchemy_imageattach.context.current_store*, *size=None*, *mimetype=None*, *original=True*)
Similar to `from_file()` except it's lower than that. It assumes that `raw_file` is readable and seekable while `from_file()` only assumes the file is readable. Also it doesn't make any in-memory buffer while `from_file()` always makes an in-memory buffer and copy the file into the buffer.

If `size` and `mimetype` are passed, it won't try to read image and will use these values instead.

It's used for implementing `from_file()` and `from_blob()` methods that are higher than it.

Parameters

- **raw_file** (file-like object, `file`) – the seekable and readable file of the image
- **store** (`Store`) – the storage to store the file. `current_store` by default
- **size** (`tuple`) – an optional size of the image. automatically detected if it's omitted
- **mimetype** (`basestring`) – an optional mimetype of the image. automatically detected if it's omitted
- **original** (`bool`) – an optional flag which represents whether it is an original image or not. defualt is `True` (meaning original)

Returns the created image instance

Return type `Image`

generate_thumbnail(*ratio=None*, *width=None*, *height=None*, *filter='undefined'*, *store=sqlalchemy_imageattach.context.current_store*, *_preprocess_image=None*, *_postprocess_image=None*)
Resizes the `original` (scales up or down) and then store the resized thumbnail into the `store`.

Parameters

- **ratio** (`numbers.Real`) – resize by its ratio. if it's greater than 1 it scales up, and if it's less than 1 it scales down. exclusive for `width` and `height` parameters
- **width** (`numbers.Integral`) – resize by its width. exclusive for `ratio` and `height` parameters
- **height** (`numbers.Integral`) – resize by its height. exclusive for `ratio` and `width` parameters
- **filter** (`basestring`, `numbers.Integral`) – a filter type to use for resizing. choose one in `wand.image.FILTER_TYPES`. default is `'undefined'` which means ImageMagick will try to guess best one to use
- **store** (`Store`) – the storage to store the resized image file. `current_store` by default
- **_preprocess_image** (`collections.Callable`) – internal-use only option for preprocessing original image before resizing. it has to be callable which takes a `wand.image.Image` object and returns a new `wand.image.Image` object

- **_postprocess_image** (collections.Callable) – internal-use only option for pre-processing original image before resizing. it has to be callable which takes a wand.image.Image object and returns a new wand.image.Image object

Returns the resized thumbnail image. it might be an already existing image if the same size already exists

Return type Image

Raises exceptions.IOError when there's no original image yet

locate(*store=sqlalchemy_imageattach.context.current_store*)
The shorthand of locate() for the original.

Parameters **store** (Store) – the storage which contains the image files. current_store by default

Returns the url of the original image

Return type basestring

make_blob(*store=sqlalchemy_imageattach.context.current_store*)
The shorthand of make_blob() for the original.

Parameters **store** (Store) – the storage which contains the image files. current_store by default

Returns the byte string of the original image

Return type str

open_file(*store=sqlalchemy_imageattach.context.current_store*, *use_seek=False*)
The shorthand of open_file() for the original.

Parameters

- **store** (Store) – the storage which contains the image files current_store by default
- **use_seek** (bool) – whether the file should seekable. if True it maybe buffered in the memory. default is False

Returns the file-like object of the image, which is a context manager (plus, also seekable only if use_seek is True)

Return type file, FileProxy, file-like object

original
(Image) The original image. It could be None if there are no stored images yet.

require_original()
Returns the original image or just raise IOError (instead of returning None). That means it guarantees the return value is never None but always Image.

Returns the original image

Return type Image

Raises exceptions.IOError when there's no original image yet

sqlalchemy_imageattach.entity.**image_attachment**(**args*, ***kwargs*)
The helper function, decorates raw relationship() function, sepcialized for relationships between Image subtypes.

It takes the same parameters as relationship().

Parameters

- ***args** – the same arguments as `relationship()`
- ****kwargs** – the same keyword arguments as `relationship()`

Returns the relationship property

Return type `sqlalchemy.orm.properties.RelationshipProperty`

3.3 `sqlalchemy_imageattach.file` — File proxies

The file-like types which wraps/proxies an other file objects.

class `sqlalchemy_imageattach.file.`**`FileProxy`**(*wrapped*)

The complete proxy for `wrapped` file-like object.

Parameters **wrapped** (`file`, file-like object) – the file object to wrap

`close`()

Closes the file. It's a context manager as well, so prefer `with` statement than direct call of this:

```
with FileProxy(file_) as f:
    print f.read()
```

`next`()

Implementation of `collections.Iterator` protocol.

`read`(*size=-1*)

Reads at the most `size` bytes from the file. It maybe less if the read hits EOF before obtaining `size` bytes.

Parameters **size** – bytes to read. if it is negative or omitted, read all data until EOF is reached. default is -1

Returns read bytes. an empty string when EOF is encountered immediately

Return type `str`

`readline`(*size=None*)

Reads an entire line from the file. A trailing newline character is kept in the string (but maybe absent when a file ends with an incomplete line).

Parameters **size** (`numbers.Integral`) – if it's present and non-negative, it is maximum byte count (including trailing newline) and an incomplete line maybe returned

Returns read bytes

Return type `str`

Note: Unlike `stdio`'s `fgets()`, the returned string contains null characters (`'\0'`) if they occurred in the input.

`readlines`(*sizehint=None*)

Reads until EOF using `readline()`.

Parameters **sizehint** (`numbers.Integral`) – if it's present, instead of reading up to EOF, whole lines totalling approximately `sizehint` bytes (or more to accommodate a final whole line)

Returns a list containing the lines read

Return type `list`

xreadlines()
The same to `iter(file)`. Use that. Deprecated since version long: time ago Use `iter()` instead.

class sqlalchemy_imageattach.file.**ReusableFileProxy**(*wrapped*)
It memorizes the current position (`tell()`) when the context enters and then rewinds (`seek()`) back to the memorized `initial_offset` when the context exits.

class sqlalchemy_imageattach.file.**SeekableFileProxy**(*wrapped*)
The almost same to `FileProxy` except it has `seek()` and `tell()` methods in addition.

seek(*offset*, *whence=0*)
Sets the file's current position.

Parameters

- **offset** (`numbers.Integral`) – the offset to set
- **whence** – see the docs of `file.seek()`. default is `os.SEEK_SET`

tell()
Gets the file's current position.

Returns the file's current position

Return type `numbers.Integral`

3.4 sqlalchemy_imageattach.migration — Storage migration

class sqlalchemy_imageattach.migration.**MigrationPlan**(*function*)
Iterable object that yields migrated images.

execute(*callback=None*)
Execute the plan. If optional `callback` is present, it is invoked with an `Image` instance for every migrated image.

Parameters **callback** (`Callable`) – an optional callback that takes an `Image` instance. it's called zero or more times

sqlalchemy_imageattach.migration.**migrate**(*session*, *declarative_base*, *source*, *destination*)
Migrate all image data from `source` storage to `destination` storage. All data in `source` storage are *not* deleted.

It does not execute migration by itself alone. You need to `execute()` the plan it returns:

```
migrate(session, Base, source, destination).execute()
```

Or iterate it using `for` statement:

```
for i in migrate(session, Base, source, destination):
    # i is an image just done migration
    print(i)
```

Parameters

- **session** (`sqlalchemy.orm.session.Session`) – SQLAlchemy session
- **declarative_base** (`sqlalchemy.ext.declarative.api.DeclarativeMeta`) – declarative base class created by `sqlalchemy.ext.declarative.declarative_base()`
- **source** (`Store`) – the storage to copy image data from
- **destination** (`Store`) – the storage to copy image data to

Returns iterable migration plan which is not executed yet

Return type MigrationPlan

sqlalchemy_imageattach.migration.**migrate_class**(*session*, *cls*, *source*, *destination*)
Migrate all image data of `cls` from `source` storage to `destination` storage. All data in `source` storage are *not* deleted.

It does not execute migration by itself alone. You need to `execute()` the plan it returns:

```
migrate_class(session, UserPicture, source, destination).execute()
```

Or iterate it using `for` statement:

```
for i in migrate_class(session, UserPicture, source, destination):
    # i is an image just done migration
    print(i)
```

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy session
- **cls** (sqlalchemy.ext.declarative.api.DeclarativeMeta) – declarative mapper class
- **source** (Store) – the storage to copy image data from
- **destination** (Store) – the storage to copy image data to

Returns iterable migration plan which is not executed yet

Return type MigrationPlan

3.5 sqlalchemy_imageattach.store — Image storage backend interface

This module declares a common interface for physically agnostic storage backends. Whatever a way to implement a storage, it needs only common operations of the interface. This consists of some basic operations like writing, reading, deletion, and finding urls.

Modules that implement the storage interface inside `sqlalchemy_imageattach.storages` package might help to implement a new storage backend.

class sqlalchemy_imageattach.store.**Store**
The interface of image storage backends. Every image storage backend implementation has to implement this.

delete(*image*)
Delete the file of the given `image`.

Parameters **image** (sqlalchemy_imageattach.entity.Image) – the image to delete

delete_file(*object_type*, *object_id*, *width*, *height*, *mimetype*)
Deletes all reproducible files related to the image. It doesn't raise any exception even if there's no such file.

Parameters

- **object_type** (basestring) – the object type of the image to put e.g. `'comics.cover'`

- **object_id** (numbers.Integral) – the object identifier number of the image to put
- **width** (numbers.Integral) – the width of the image to delete
- **height** (numbers.Integral) – the height of the image to delete
- **mimetype** (basestring) – the mimetype of the image to delete e.g. 'image/jpeg'

get_file(*object_type*, *object_id*, *width*, *height*, *mimetype*)
Gets the file-like object of the given criteria.

Parameters

- **object_type** (basestring) – the object type of the image to find e.g. 'comics.cover'
- **object_id** (numbers.Integral) – the object identifier number of the image to find
- **width** (numbers.Integral) – the width of the image to find
- **height** (numbers.Integral) – the height of the image to find
- **mimetype** (basestring) – the mimetype of the image to find e.g. 'image/jpeg'

Returns the file of the image

Return type file-like object, file

Raises exceptions.IOError when such file doesn't exist

Note: This is an abstract method which has to be implemented (overridden) by subclasses.

It's not for consumers but implementations, so consumers should use open() method instead of this.

---

get_url(*object_type*, *object_id*, *width*, *height*, *mimetype*)
Gets the file-like object of the given criteria.

Parameters

- **object_type** (basestring) – the object type of the image to find e.g. 'comics.cover'
- **object_id** (numbers.Integral) – the object identifier number of the image to find
- **width** (numbers.Integral) – the width of the image to find
- **height** (numbers.Integral) – the height of the image to find
- **mimetype** (basestring) – the mimetype of the image to find e.g. 'image/jpeg'

Returns the url locating the image

Return type basestring

Note: This is an abstract method which has to be implemented (overridden) by subclasses.

It's not for consumers but implementations, so consumers should use locate() method instead of this.

---

locate(*image*)
Gets the URL of the given image.

Parameters **image** (sqlalchemy_imageattach.entity.Image) – the image to get its url

Returns the url of the image

Return type basestring

open(*image*, *use_seek=False*)

Opens the file-like object of the given `image`. Returned file-like object guarantees:

- context manager protocol
- `collections.Iterable` protocol
- `collections.Iterator` protocol
- `read()` method
- `readline()` method
- `readlines()` method

To sum up: you definitely can read the file, in `with` statement and `for` loop.

Plus, if `use_seek` option is `True`:

- `seek()` method
- `tell()` method

For example, if you want to make a local copy of the image:

```
import shutil

with store.open(image) as src:
    with open(filename, 'wb') as dst:
        shutil.copyfileobj(src, dst)
```

Parameters

- **image** (`sqlalchemy_imageattach.entity.Image`) – the image to get its file
- **use_seek** (`bool`) – whether the file should seekable. if `True` it maybe buffered in the memory. default is `False`

Returns the file-like object of the image, which is a context manager (plus, also seekable only if `use_seek` is `True`)

Return type `file`, `FileProxy`, file-like object

Raises **exceptions.IOError** when such file doesn't exist

put_file(*file*, *object_type*, *object_id*, *width*, *height*, *mimetype*, *reproducible*)

Puts the `file` of the image.

Parameters

- **file** (file-like object, `file`) – the image file to put
- **object_type** (`basestring`) – the object type of the image to put e.g. `'comics.cover'`
- **object_id** (`numbers.Integral`) – the object identifier number of the image to put
- **width** (`numbers.Integral`) – the width of the image to put
- **height** (`numbers.Integral`) – the height of the image to put
- **mimetype** (`basestring`) – the mimetype of the image to put e.g. `'image/jpeg'`
- **reproducible** (`bool`) – `True` only if it's reproducible by computing e.g. resized thumbnails. `False` if it cannot be reproduced e.g. original images

---

Note: This is an abstract method which has to be implemented (overridden) by subclasses.

It's not for consumers but implementations, so consumers should use `store()` method instead of this.

---

store(*image*, *file*)
Stores the actual data `file` of the given `image`.

```
with open(imagefile, 'rb') as f:
    store.store(image, f)
```

Parameters

- **image** (`sqlalchemy_imageattach.entity.Image`) – the image to store its actual data file
- **file** (file-like object, `file`) – the image file to put

3.6 `sqlalchemy_imageattach.util` — Utilities

This module provides some utility functions to manipulate docstrings at runtime. It's useful for adjusting the docs built by Sphinx without making the code ugly.

sqlalchemy_imageattach.util.**append_docstring**(*docstring*, **lines*)
Appends the `docstring` with given `lines`:

```
function.__doc__ = append_docstring(
    function.__doc__,
    '.. note::'
    '',
    '   Appended docstring!'
)
```

Parameters

- **docstring** – a docstring to be appended
- ***lines** – lines of trailing docstring

Returns new docstring which is appended

Return type `basestring`

sqlalchemy_imageattach.util.**append_docstring_attributes**(*docstring*, *locals*)
Manually appends class' `docstring` with its attribute docstrings. For example:

```
class Entity(object):
    # ...

    __doc__ = append_docstring_attributes(
        __doc__,
        dict((k, v) for k, v in locals()
                    if isinstance(v, MyDescriptor))
    )
```

Parameters

- **docstring** (basestring) – class docstring to be appended
- **locals** (collections.Mapping) – attributes dict

Returns appended docstring

Return type basestring

sqlalchemy_imageattach.util.**get_minimum_indent**(*docstring*, *ignore_before=1*)
Gets the minimum indent string from the docstring:

```
>>> get_minimum_indent('Hello')
''
>>> get_minimum_indent('Hello\n    world::\n        yeah')
'    '
```

Parameters

- **docstring** (basestring) – the docstring to find its minimum indent
- **ignore_before** (numbers.Integral) – ignore lines before this line. usually docstrings which follow **PEP 8** have no indent for the first line, so its default value is 1

Returns the minimum indent string which consists of only whitespaces (tabs and/or spaces)

Return type basestring

3.7 sqlalchemy_imageattach.version — Version data

sqlalchemy_imageattach.version.**VERSION = '0.8.1'**
(str) The version string e.g. '1.2.3'.

sqlalchemy_imageattach.version.**VERSION_INFO = (0, 8, 1)**
(tuple) The triple of version numbers e.g. (1, 2, 3).

3.8 sqlalchemy_imageattach.stores — Storage backend implementations

3.8.1 sqlalchemy_imageattach.stores.fs — Filesystem-backed image storage

It provides two filesystem-backed image storage implementations:

FileSystemStore It stores image files into the filesystem of the specified path, but locate() method returns URLs of the hard-coded base URL.

HttpExposedFileSystemStore The mostly same to FileSystemStore except it provides WSGI middleware (wsgi_middleware()) which actually serves image files and its locate() method returns URLs based on the actual requested URL.

class sqlalchemy_imageattach.stores.fs.**BaseFileSystemStore**(*path*)
Abstract base class of FileSystemStore and HttpExposedFileSystemStore.

class sqlalchemy_imageattach.stores.fs.**FileSystemStore**(*path*, *base_url*)
Filesystem-backed storage implementation with hard-coded URL routing.

class sqlalchemy_imageattach.stores.fs.**HttpExposedFileSystemStore**(*path*, *prefix='__images__'*)

Filesystem-backed storage implementation with WSGI middleware which serves actual image files.

```
from flask import Flask
from sqlalchemy_imageattach.stores.fs import HttpExposedFileSystemStore

app = Flask(__name__)
fs_store = HttpExposedFileSystemStore('userimages', 'images/')
app.wsgi_app = fs_store.wsgi_middleware(app.wsgi_app)
```

wsgi_middleware(*app*)

WSGI middlewares that wraps the given `app` and serves actual image files.

```
fs_store = HttpExposedFileSystemStore('userimages', 'images/')
app = fs_store.wsgi_middleware(app)
```

Parameters **app** (`collections.Callable`) – the wsgi app to wrap

Returns the another wsgi app that wraps `app`

Return type `StaticServerMiddleware`

class sqlalchemy_imageattach.stores.fs.**StaticServerMiddleware**(*app*, *url_path*, *dir_path*, *block_size=8192*)

Simple static server WSGI middleware.

Parameters

- **app** (`collections.Callable`) – the fallback app when the path is not scoped in `url_path`
- **url_path** (`basestring`) – the exposed path to url
- **dir_path** (`basestring`) – the filesystem directory path to serve
- **block_size** (`numbers.Integral`) – the block size in bytes

sqlalchemy_imageattach.stores.fs.**guess_extension**(*mimetype*)

Finds the right filename extension (e.g. `'.png'`) for the given `mimetype` (e.g. *image/png*).

Parameters **mimetype** (`basestring`) – mimetype string e.g. `'image/jpeg'`

Returns filename extension for the mimetype

Return type `basestring`

3.8.2 sqlalchemy_imageattach.stores.s3 — AWS S3 backend storage

The backend storage implementation for Simple Storage Service provided by Amazon Web Services.

sqlalchemy_imageattach.stores.s3.**BASE_URL_FORMAT = 'https://{0}.s3.amazonaws.com'**

(`str`) The format string of base url of AWS S3. Contains no trailing slash. Default is `'https://{0}.s3.amazonaws.com'`.

sqlalchemy_imageattach.stores.s3.**DEFAULT_MAX_AGE = 31536000**

(`numbers.Integral`) The default `max-age` seconds of *Cache-Control*. It's the default value of `S3Store.max_age` attribute.

class sqlalchemy_imageattach.stores.s3.**S3Request**(*url*, *bucket*, *access_key*, *secret_key*, *data=None*, *headers={}*, *method=None*, *content_type=None*)

HTTP request for S3 REST API which does authentication.

class sqlalchemy_imageattach.stores.s3.**S3SandboxStore**(*underlying*, *overriding*, *access_key=None*, *secret_key=None*, *max_age=31536000*, *underlying_prefix=''*, *overriding_prefix=''*)

It stores images into physically two separated S3 buckets while these look like logically exist in the same store. It takes two buckets for *read-only* and *overwrite*: `underlying` and `overriding`.

It's useful for development/testing purpose, because you can use the production store in sandbox.

Parameters

- **underlying** (`basestring`) – the name of *underlying* bucket for read-only
- **overriding** (`basestring`) – the name of *overriding* bucket to record overriding modifications
- **max_age** (`numbers.Integral`) – the `max-age` seconds of *`Cache-Control`*. default is `DEFAULT_MAX_AGE`
- **overriding_prefix** (`basestring`) – means the same to `S3Store.prefix` but it's only applied for `overriding`
- **underlying_prefix** (`basestring`) – means the same to `S3Store.prefix` but it's only applied for `underlying`

DELETED_MARK_MIMETYPE = 'application/x-sqlalchemy-imageattach-sandbox-deleted'

All keys marked as "deleted" have this mimetype as its *`Content-Type`* header.

overriding = None

(`S3Store`) The *overriding* store to record overriding modification.

underlying = None

(`S3Store`) The *underlying* store for read-only.

class sqlalchemy_imageattach.stores.s3.**S3Store**(*bucket*, *access_key=None*, *secret_key=None*, *max_age=31536000*, *prefix=''*, *public_base_url=None*)

Image storage backend implementation using S3. It implements `Store` interface.

If you'd like to use it with Amazon CloudFront, pass the base url of the distribution to `public_base_url`. Note that you should configure *Forward Query Strings* to *Yes* when you create the distribution. Because SQLAlchemy-ImageAttach will add query strings to public URLs to invalidate cache when the image is updated.

Parameters

- **bucket** (`basestring`) – the buckect name
- **max_age** (`numbers.Integral`) – the `max-age` seconds of *`Cache-Control`*. default is `DEFAULT_MAX_AGE`
- **prefix** (`basestring`) – the optional key prefix to logically separate stores with the same bucket. not used by default
- **public_base_url** (`basestring`) – an optional url base for public urls. useful when used with cdn

Changed in version 0.8.1: Added `public_base_url` parameter.

bucket = None
(`basestring`) The S3 bucket name.

max_age = None
(`numbers.Integral`) The `max-age` seconds of *Cache-Control*.

prefix = None
(`basestring`) The optional key prefix to logically separate stores with the same bucket.

public_base_url = None
(`basestring`) The optional url base for public urls.

CHAPTER

FOUR

OPEN SOURCE

SQLAlchemy-ImageAttach is an open source software written by Hong Minhee (initially written for Crosspop). The source code is distributed under MIT license and you can find it at GitHub repository. Check out now:

```
$ git clone git://github.com/crosspop/sqlalchemy-imageattach.git
```

If you find any bug, please create an issue to the issue tracker. Pull requests are also always welcome!

Check out *SQLAlchemy-ImageAttach Changelog* as well.

PYTHON MODULE INDEX

S