WHY STORING FILES FOR THE WEB IS NOT AS STRAIGHTFORWARD AS YOU MIGHT THINK

Alessandro Molina
@__amol__
amol@turbogears.org
Who am I

- CTO @ AXANT.it, mostly Python company
- TurboGears2 core team member
- Contributions to web world python libraries
  - MING MongoDB ODM
  - Beaker
  - ToscaWidgets2
  - Formencode
Background

- Everything starts from a project which was just a POT with budget constraint.
- Obviously it became the final product.
- It saved and updated a lot of files, mostly images.
Technologies.

- **Short on budget**: cloud storage was not an available choice
- **Short on time**: developers choose to just store everything on disk and rely on nginx to serve them in a good enough manner
The Technical Consultant

- Customer had a technical leader that enforced deployment decisions.
- Customer decided production environment three days before the "go live"
- Due to limited budget he decided they were not going to rent a server.
The product owner choice
Murphy Law

- They went for Heroku free plan as PaaS
- Heroku *doesn’t support* storing files on disk
- The whole software did store files on disk
Ooops
Panic

- The day before launch, team *rewrote 30%* of the software to switch saving files from disk to *GridFS* (app was mongodb based)
- It was an *huge hack* based on monkeypatching the attachment classes
- It went online with practically no testing on the field.
The day after

- After emergency has been solved it was clear that we needed a better way to handle such issues.
- We decided to create a tool to solve the issue independently from the web development framework in use.
Lessons learnt by working on TurboGears2 for the past years:

- Web Apps are an unstable environment when designing a framework:
  - Their infrastructure might expand, downscale or change during their lifetime.
  - The technologies you relied on can change or even disappear during their lifetime.
  - Automatic testing should be easy to implement.
  - Easily usable wins over features, people will build features themselves over a solid foundation.
Allow for **Infrastructure changes**

- Permit to choose between multiple storage engines just by changing a configuration file
- Permit **switching** storage engine at runtime without breaking past files
- Permit to **concurrently** use multiple storages
Have your choice

I DON'T ALWAYS STORE FILES ON THIRD PARTY STORAGES

BUT WHEN I DO I USE AT LEAST 4 OF THEM
Multiple Storages

- One “default” storage, any other storage can be promoted to default, anytime.
- When uploading a file it goes to the default storage unless otherwise specified.
- Each storage has a name, files can be uniquely identified among storages by storage_name/fileid.
DepotManager

• The DepotManager is the single interface to DEPOT.

• It tracks the active storages, the default one, and the WSGI middleware.

• To work on a storage just get it from the DepotManager.
Easy to Use

- Simple things should be simple

```python
from depot.manager import DepotManager

# Configure a *default* depot to store files on MongoDB
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
```
With Batteries

• Complex things should be straightforward

```python
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)

    # 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'))
```
Allow for technology changes

- Attachment field for SQLAlchemy
- Attachment field for MongoDB
- Builtin support for S3, LocalFiles and GridFS
- Easily pluggable custom Backends
- Delivering files uses a WSGI middleware compatible with any web framework.
Empowers your loved queries!
Copes with Database

● Transactions rollback should delete newly uploaded files and recover the previous ones.

● Deleting an item deletes attached files (unless rollback happens)
Easy to Extend

● **Custom attachments** can be easily created

```
UploadedFileField(upload_type=UploadedImageWithMaxSize)
```

● **Filters** can be applied to attachments

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

● **Multiple filters** can be applied (rescale image and create thumbnails)
Custom Attachments

- Attachment Classes are in charge of storing the actually uploaded file.
- They can change the file before it’s uploaded.
- They can add additional data and even behaviours to the file.
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 filename 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)
Filters

- Each attachment can have **multiple** filters.
- They run after upload, so they can **add** metadata or generate **new files** but not replace the original one.
- They can store additional **metadata** with the file, but **not behaviours** (methods).
Writing a Filter

class WithThumbnailFilter(FileFilter):
    def __init__(self, size=(128,128), format='PNG'):
        self.thumbnail_size, self.thumbnail_format = (size, 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()
        thumb_path, thumb_id = uploaded_file.store_content(output, thumb_file_name)
        thumb_url = DepotManager.get_middleware().url_for(thumb_path)

        uploaded_file.update({"thumb_id": thumb_id, "thumb_path": thumb_path, "thumb_url": thumb_url})
Store what you **need** in metadata

```python
>>> d = DBSession.query(Document).filter_by(name='Foo').first()
>>> print d.photo.thumb_url
/depot/default/5b1a489e-0d33-11e4-8e2a-0800277ee230
```
And it’s WebScale™!
Made for the Web

- Storage backends can provide public url for any CDN

- File information common in HTTP are provided as properties out of the box
  - content_type
  - last_modified
  - content_length
  - filename
Web Application Friendly

- Need to serve stored files? Just mount `DepotManager.make.middleware` around your app and start serving them.

- If files are stored on a backend that supports HTTP, the user will be permanently redirected there by the middleware instead of serving files itself.
Feel **free to try it!**

- Python 2.6, 2.7, 3.2, 3.3 and 3.4
- pip install `filedepot`
- Fully Documented
  - https://depot.readthedocs.org
- Tested with 100% coverage
  - https://travis-ci.org/amol-/depot
Questions?

The End
is such a scary place to be.