Paperless Documentation

Release 0.6.1

Daniel Quinn

Aug 12, 2017

Contents

Paperless is a simple Django application running in two parts: a *consumer* (the thing that does the indexing) and the *webserver* (the part that lets you search & download already-indexed documents). If you want to learn more about its functions keep on reading after the installation section.

CHAPTER 1

Why This Exists

Paper is a nightmare. Environmental issues aside, there's no excuse for it in the 21st century. It takes up space, collects dust, doesn't support any form of a search feature, indexing is tedious, it's heavy and prone to damage & loss.

I wrote this to make "going paperless" easier. I do not have to worry about finding stuff again. I feed documents right from the post box into the scanner and then shred them. Perhaps you might find it useful too.

CHAPTER 2

Contents

Requirements

You need a Linux machine or Unix-like setup (theoretically an Apple machine should work) that has the following software installed:

- Python3 (with development libraries, pip and virtualenv)
- GNU Privacy Guard
- Tesseract, plus its language files matching your document base.
- Imagemagick version 6.7.5 or higher
- unpaper

Notably, you should confirm how you access your Python3 installation. Many Linux distributions will install Python3 in parallel to Python2, using the names `python3` and `python` respectively. The same goes for `pip3` and `pip`. Running Paperless with Python2 will likely break things, so make sure that you're using the right version.

For the purposes of simplicity, `python` and `pip` is used everywhere to refer to their Python3 versions.

In addition to the above, there are a number of Python requirements, all of which are listed in a file called `requirements.txt` in the project root directory.

If you're not working on a virtual environment (like Vagrant or Docker), you should probably be using a virtualenv, but that's your call. The reasons why you might choose a virtualenv or not aren't really within the scope of this document. Needless to say if you don't know what a virtualenv is, you should probably figure that out before continuing.

Apple-tastic Complications

Some users have run into problems with installing ImageMagick on Apple systems using HomeBrew. The solution appears to be to install ghostscript as well as ImageMagick:

```
$ brew install ghostscript
$ brew install imagemagick
$ brew install libmagic
```

Python-specific Requirements: No Virtualenv

If you don't care to use a virtual env, then installation of the Python dependencies is easy:

```
$ pip install --user --requirement /path/to/paperless/requirements.txt
```

This will download and install all of the requirements into `${HOME}/.local`. Remember that your distribution may be using `pip3` as mentioned above.

Python-specific Requirements: Virtualenv

Using a virtualenv for this is pretty straightforward: create a virtualenv, enter it, and install the requirements using the `requirements.txt` file:

```
$ virtualenv --python=/path/to/python3 /path/to/arbitrary/directory
$ . /path/to/arbitrary/directory/bin/activate
$ pip install  --requirement /path/to/paperless/requirements.txt
```

Now you're ready to go. Just remember to enter (activate) your virtualenv whenever you want to use Paperless.

Documentation

As generation of the documentation is not required for the use of Paperless, dependencies for this process are not included in `requirements.txt`. If you'd like to generate your own docs locally, you'll need to:

```
$ pip install sphinx
```

and then cd into the `docs` directory and type `make html`.

If you are using Docker, you can use the following commands to build the documentation and run a webserver serving it on port 8001:

```
$ pwd
/path/to/paperless

$ docker build -t paperless:docs -f docs/Dockerfile .
$ docker run --rm -it -p "8001:8000" paperless:docs
```

Setup

Paperless isn't a very complicated app, but there are a few components, so some basic documentation is in order. If you follow along in this document and still have trouble, please open an issue on GitHub so I can fill in the gaps.

Download

The source is currently only available via GitHub, so grab it from there, either by using `git`:

```
$ git clone https://github.com/danielquinn/paperless.git
$ cd paperless
```

or just download the tarball and go that route:

```
$ cd to the directory where you want to run Paperless
$ wget https://github.com/danielquinn/paperless/archive/master.zip
$ unzip master.zip
$ cd paperless-master
```

Installation & Configuration

You can go multiple routes with setting up and running Paperless. The *Vagrant route* is quick & easy, but means you're running a VM which comes with memory consumption etc. We also *support Docker*, which you can use natively under Linux and in a VM with Docker Machine (this guide was written for native Docker usage under Linux, you might have to adapt it for Docker Machine.) Not to forget the virtualenv, this is similar to *bare metal* with the exception that you have to activate the virtualenv first. Last but not least, the standard *bare metal* approach is a little more complicated, but worth it because it makes it easier should you want to contribute some code back.

Standard (Bare Metal)

1. Install the requirements as per the *requirements* page.
2. Within the extract of master.zip go to the `src` directory.
3. Copy `paperless.conf.example` to `/etc/paperless.conf` also the virtual envrionment look there for it and open it in your favourite editor. Because this file contains passwords it should only be readable by user root and paperless ! Set the values for:
   - `PAPERLESS_CONSUMPTION_DIR`: this is where your documents will be dumped to be consumed by Paperless.
   - `PAPERLESS_PASSPHRASE`: this is the passphrase Paperless uses to encrypt/decrypt the original document.
   - `PAPERLESS_OCR_THREADS`: this is the number of threads the OCR process will spawn to process document pages in parallel.
4. Initialise the SQLite database with `./manage.py migrate`.
5. Create a user for your Paperless instance with `./manage.py createsuperuser`. Follow the prompts to create your user.
6. Start the webserver with `./manage.py runserver <IP>:<PORT>`. If no specifc IP or port are given, the default is `127.0.0.1:8000` also known as http://localhost:8000/. You should now be able to visit your (empty) at Paperless webserver or whatever you chose before. You can login with the user/pass you created in #5.
7. In a separate window, change to the `src` directory in this repo again, but this time, you should start the consumer script with `./manage.py document_consumer`.
8. Scan something or put a file into the `CONSUMPTION_DIR`.
9. Wait a few minutes

10. Visit the document list on your webserver, and it should be there, indexed and downloadable.

Vagrant Method

1. Install Vagrant. How you do that is really between you and your OS.
2. Run `vagrant up`. An instance will start up for you. When it's ready and provisioned...
3. Run `vagrant ssh` and once inside your new vagrant box, edit `/etc/paperless.conf` and set the values for:
   - `PAPERLESS_CONSUMPTION_DIR`: this is where your documents will be dumped to be consumed by Paperless.
   - `PAPERLESS_PASSPHRASE`: this is the passphrase Paperless uses to encrypt/decrypt the original document.
   - `PAPERLESS_SHARED_SECRET`: this is the "magic word" used when consuming documents from mail or via the API. If you don't use either, leaving it blank is just fine.
4. Exit the vagrant box and re-enter it with `vagrant ssh` again. This updates the environment to make use of the changes you made to the config file.
5. Initialise the database with `/opt/paperless/src/manage.py migrate`.
6. Still inside your vagrant box, create a user for your Paperless instance with `/opt/paperless/src/manage.py createsuperuser`. Follow the prompts to create your user.
7. Start the webserver with `/opt/paperless/src/manage.py runserver 0.0.0.0:8000`. You should now be able to visit your (empty) Paperless webserver at `172.28.128.4:8000`. You can login with the user/pass you created in #6.
8. In a separate window, run `vagrant ssh` again, but this time once inside your vagrant instance, you should start the consumer script with `/opt/paperless/src/manage.py document_consumer`.
9. Scan something. Put it in the `CONSUMPTION_DIR`.
10. Wait a few minutes
11. Visit the document list on your webserver, and it should be there, indexed and downloadable.

Docker Method

1. Install Docker.

   Caution: As mentioned earlier, this guide assumes that you use Docker natively under Linux. If you are using Docker Machine under Mac OS X or Windows, you will have to adapt IP addresses, volume-mounting, command execution and maybe more.

2. Install docker-compose.[1]

[1] You of course don't have to use docker-compose, but it simplifies deployment immensely. If you know your way around Docker, feel free to tinker around without using compose!

> **Caution:** If you want to use the included `docker-compose.yml.example` file, you need to have at least Docker version **1.10.0** and docker-compose version **1.6.0**.
>
> See the Docker installation guide on how to install the current version of Docker for your operating system or Linux distribution of choice. To get an up-to-date version of docker-compose, follow the docker-compose installation guide if your package repository doesn't include it.

3. Create a copy of `docker-compose.yml.example` as `docker-compose.yml` and a copy of `docker-compose.env.example` as `docker-compose.env`. You'll be editing both these files: taking a copy ensures that you can `git pull` to receive updates without risking merge conflicts with your modified versions of the configuration files.
4. Modify `docker-compose.yml` to your preferences, following the instructions in comments in the file. The only change that is a hard requirement is to specify where the consumption directory should mount.
5. Modify `docker-compose.env` and adapt the following environment variables:

   PAPERLESS_PASSPHRASE This is the passphrase Paperless uses to encrypt/decrypt the original document.

   PAPERLESS_OCR_THREADS This is the number of threads the OCR process will spawn to process document pages in parallel. If the variable is not set, Python determines the core-count of your CPU and uses that value.

   PAPERLESS_OCR_LANGUAGES If you want the OCR to recognize other languages in addition to the default English, set this parameter to a space separated list of three-letter language-codes after ISO 639-2/T. For a list of available languages – including their three letter codes – see the Debian packagelist.

   USERMAP_UID and USERMAP_GID If you want to mount the consumption volume (directory `/consume` within the containers) to a host-directory – which you probably want to do – access rights might be an issue. The default user and group `paperless` in the containers have an id of 1000. The containers will enforce that the owning group of the consumption directory will be `paperless` to be able to delete consumed documents. If your host-system has a group with an ID of 1000 and you don't want this group to have access rights to the consumption directory, you can use `USERMAP_GID` to change the id in the container and thus the one of the consumption directory. Furthermore, you can change the id of the default user as well using `USERMAP_UID`.
6. Run `docker-compose up -d`. This will create and start the necessary containers.
7. To be able to login, you will need a super user. To create it, execute the following command:

```
$ docker-compose run --rm webserver createsuperuser
```

   This will prompt you to set a username (default `paperless`), an optional e-mail address and finally a password.
8. The default `docker-compose.yml` exports the webserver on your local port 8000. If you haven't adapted this, you should now be able to visit your Paperless webserver at `http://127.0.0.1:8000`. You can login with the user and password you just created.
9. Add files to consumption directory the way you prefer to. Following are two possible options:

   (a) Mount the consumption directory to a local host path by modifying your `docker-compose.yml`:

```
diff --git a/docker-compose.yml b/docker-compose.yml
--- a/docker-compose.yml
+++ b/docker-compose.yml
@@ -17,9 +18,8 @@ services:
         volumes:
             - paperless-data:/usr/src/paperless/data
             - paperless-media:/usr/src/paperless/media
```

```
-             - /consume
+             - /local/path/you/choose:/consume
```

> **Danger:** While the consumption container will ensure at startup that it can **delete** a consumed file from a host-mounted directory, it might not be able to **read** the document in the first place if the access rights to the file are incorrect.
>
> Make sure that the documents you put into the consumption directory will either be readable by everyone (`chmod o+r file.pdf`) or readable by the default user or group id 1000 (or the one you have set with `USERMAP_UID` or `USERMAP_GID` respectively).

(b) Use `docker cp` to copy your files directly into the container:

```
$ # Identify your containers
$ docker-compose ps
        Name                       Command                State     Ports
-------------------------------------------------------------------------
paperless_consumer_1    /sbin/docker-entrypoint.sh ...   Exit 0
paperless_webserver_1   /sbin/docker-entrypoint.sh ...   Exit 0

$ docker cp /path/to/your/file.pdf paperless_consumer_1:/consume
```

`docker cp` is a one-shot-command, just like `cp`. This means that every time you want to consume a new document, you will have to execute `docker cp` again. You can of course automate this process, but option 1 is generally the preferred one.

> **Danger:** `docker cp` will change the owning user and group of a copied file to the acting user at the destination, which will be `root`.
>
> You therefore need to ensure that the documents you want to copy into the container are readable by everyone (`chmod o+r file.pdf`) before copying them.

Making Things a Little more Permanent

Once you've tested things and are happy with the work flow, you can automate the process of starting the webserver and consumer automatically.

Standard (Bare Metal, Systemd)

If you're running on a bare metal system that's using Systemd, you can use the service unit files in the `scripts` directory to set this up. You'll need to create a user called `paperless` (without login (if not already done so #5)) and setup Paperless to be in a place that this new user can read and write to. Be sure to edit the service scripts to point to the proper location of your paperless install, referencing the appropriate Python binary. For example: `ExecStart=/path/to/python3 /path/to/paperless/src/manage.py document_consumer`. If you don't want to make a new user, you can change the `Group` and `User` variables accordingly.

Then, as `root` (or using `sudo`) you can just copy the `.service` files to the Systemd directory and tell it to enable the two services:

```
# cp /path/to/paperless/scripts/paperless-consumer.service /etc/systemd/system/
# cp /path/to/paperless/scripts/paperless-webserver.service /etc/systemd/system/
```

```
# systemctl enable paperless-consumer
# systemctl enable paperless-webserver
# systemctl start paperless-consumer
# systemctl start paperless-webserver
```

Ubuntu 14.04 (Bare Metal, Upstart)

Ubuntu 14.04 and earlier use the Upstart init system to start services during the boot process. To configure Upstart to run Paperless automatically after restarting your system:

1. Change to the directory where Upstart's configuration files are kept: `cd /etc/init`
2. Create a new file: `sudo nano paperless-server.conf`
3. In the newly-created file enter:

   ```
   start on (local-filesystems and net-device-up IFACE=eth0)
   stop on shutdown

   respawn
   respawn limit 10 5

   script
     exec /srv/paperless/src/manage.py runserver 0.0.0.0:80
   end script
   ```

   Note that you'll need to replace `/srv/paperless/src/manage.py` with the path to the `manage.py` script in your installation directory.

   If you are using a network interface other than `eth0`, you will have to change `IFACE=eth0`. For example, if you are connected via WiFi, you will likely need to replace `eth0` above with `wlan0`. To see all interfaces, run `ifconfig -a`.

   Save the file.

4. Create a new file: `sudo nano paperless-consumer.conf`
5. In the newly-created file enter:

   ```
   start on (local-filesystems and net-device-up IFACE=eth0)
   stop on shutdown

   respawn
   respawn limit 10 5

   script
     exec /srv/paperless/src/manage.py document_consumer
   end script
   ```

   Replace `/srv/paperless/src/manage.py` with the same values as in step 3 above and replace `eth0` with the appropriate value, if necessary. Save the file.

These two configuration files together will start both the Paperless webserver and document consumer processes when the file system and network interface specified is available after boot. Furthermore, if either process ever exits unexpectedly, Upstart will try to restart it a maximum of 10 times within a 5 second period.

Using a Real Webserver

The default is to use Django's development server, as that's easy and does the job well enough on a home network. However, if you want to do things right, it's probably a good idea to use a webserver capable of handling more than one thread. You will also have to let the webserver serve the static files (CSS, JavaScript) from the directory configured in `PAPERLESS_STATICDIR`. For that, you need to run `./manage.py collectstatic` in the `src` directory. The default static files directory is `../static`.

Apache

This is a configuration supplied by steckerhalter on GitHub. It uses Apache and mod_wsgi, with a Paperless installation in /home/paperless/:

```
<VirtualHost *:80>
    ServerName example.com

    Alias /static/ /home/paperless/paperless/static/
    <Directory /home/paperless/paperless/static>
        Require all granted
    </Directory>

    WSGIScriptAlias / /home/paperless/paperless/src/paperless/wsgi.py
    WSGIDaemonProcess example.com user=paperless group=paperless threads=5 python-
↪path=/home/paperless/paperless/src:/home/paperless/.env/lib/python3.4/site-packages
    WSGIProcessGroup example.com

    <Directory /home/paperless/paperless/src/paperless>
        <Files wsgi.py>
            Require all granted
        </Files>
    </Directory>
</VirtualHost>
```

Nginx + Gunicorn

If you're using Nginx, the most common setup is to combine it with a Python-based server like Gunicorn so that Nginx is acting as a proxy. Below is a copy of a simple Nginx configuration fragment making use of SSL and IPv6 to refer to a gunicorn instance listening on a local Unix socket:

```
upstream transfer_server {
  server unix:/run/example.com/gunicorn.sock fail_timeout=0;
}

# Redirect requests on port 80 to 443
server {
  listen 80;
  listen [::]:80;
  server_name example.com;
  rewrite ^ https://$server_name$request_uri? permanent;
}

server {

  listen 443 ssl;
  listen [::]:443;
```

```
  client_max_body_size 4G;
  server_name example.com;
  keepalive_timeout 5;
  root /var/www/example.com;

  ssl on;

  ssl_certificate         /etc/letsencrypt/live/example.com/fullchain.pem;
  ssl_certificate_key     /etc/letsencrypt/live/example.com/privkey.pem;
  ssl_trusted_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
  ssl_session_timeout 1d;
  ssl_session_cache shared:SSL:50m;

  # Diffie-Hellman parameter for DHE ciphersuites, recommended 2048 bits
  # Generate with:
  #   openssl dhparam -out /etc/nginx/dhparam.pem 2048
  ssl_dhparam /etc/nginx/dhparam.pem;

  # What Mozilla calls "Intermediate configuration"
  # Copied from https://mozilla.github.io/server-side-tls/ssl-config-generator/
  ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers 'ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-
↪AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-
↪RSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384:ECDHE-
↪ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-
↪SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA:ECDHE-
↪RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA256:DHE-
↪RSA-AES256-SHA:ECDHE-ECDSA-DES-CBC3-SHA:ECDHE-RSA-DES-CBC3-SHA:EDH-RSA-DES-CBC3-
↪SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-SHA256:AES128-
↪SHA:AES256-SHA:DES-CBC3-SHA:!DSS';
  ssl_prefer_server_ciphers on;

  add_header Strict-Transport-Security max-age=15768000;

  ssl_stapling on;
  ssl_stapling_verify on;

  access_log /var/log/nginx/example.com.log main;
  error_log /var/log/nginx/example.com.err info;

  location / {
    try_files $uri @proxy_to_app;
  }

  location @proxy_to_app {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto https;
    proxy_set_header Host $host;
    proxy_redirect off;
    proxy_pass http://transfer_server;
  }

}
```

Once you've got Nginx configured, you'll want to have a configuration file for your gunicorn instance. This should do the trick:

```
import os

bind = 'unix:/run/example.com/gunicorn.sock'
backlog = 2048
workers = 6
worker_class = 'sync'
worker_connections = 1000
timeout = 30
keepalive = 2
debug = False
spew = False
daemon = False
pidfile = None
umask = 0
user = None
group = None
tmp_upload_dir = None
errorlog = '/var/log/example.com/gunicorn.err'
loglevel = 'warning'
accesslog = '/var/log/example.com/gunicorn.log'
proc_name = None

def post_fork(server, worker):
    server.log.info("Worker spawned (pid: %s)", worker.pid)

def pre_fork(server, worker):
    pass

def pre_exec(server):
    server.log.info("Forked child, re-executing.")

def when_ready(server):
    server.log.info("Server is ready. Spawning workers")

def worker_int(worker):
    worker.log.info("worker received INT or QUIT signal")

    ## get traceback info
    import threading, sys, traceback
    id2name = dict([(th.ident, th.name) for th in threading.enumerate()])
    code = []
    for threadId, stack in sys._current_frames().items():
        code.append("\n# Thread: %s(%d)" % (id2name.get(threadId,""),
            threadId))
        for filename, lineno, name, line in traceback.extract_stack(stack):
            code.append('File: "%s", line %d, in %s' % (filename,
                lineno, name))
            if line:
                code.append("  %s" % (line.strip()))
    worker.log.debug("\n".join(code))

def worker_abort(worker):
    worker.log.info("worker received SIGABRT signal")
```

Vagrant

You may use the Ubuntu explanation above. Replace `(local-filesystems and net-device-up IFACE=eth0)` with `vagrant-mounted`.

Docker

If you're using Docker, you can set a restart-policy in the `docker-compose.yml` to have the containers automatically start with the Docker daemon.

Consumption

Once you've got Paperless setup, you need to start feeding documents into it. Currently, there are three options: the consumption directory, IMAP (email), and HTTP POST.

The Consumption Directory

The primary method of getting documents into your database is by putting them in the consumption directory. The `document_consumer` script runs in an infinite loop looking for new additions to this directory and when it finds them, it goes about the process of parsing them with the OCR, indexing what it finds, and encrypting the PDF, storing it in the media directory.

Getting stuff into this directory is up to you. If you're running Paperless on your local computer, you might just want to drag and drop files there, but if you're running this on a server and want your scanner to automatically push files to this directory, you'll need to setup some sort of service to accept the files from the scanner. Typically, you're looking at an FTP server like Proftpd or Samba.

So where is this consumption directory? It's wherever you define it. Look for the `CONSUMPTION_DIR` value in `settings.py`. Set that to somewhere appropriate for your use and put some documents in there. When you're ready, follow the *consumer* instructions to get it running.

Hooking into the Consumption Process

Sometimes you may want to do something arbitrary whenever a document is consumed. Rather than try to predict what you may want to do, Paperless lets you execute scripts of your own choosing just before or after a document is consumed using a couple simple hooks.

Just write a script, put it somewhere that Paperless can read & execute, and then put the path to that script in `paperless.conf` with the variable name of either `PAPERLESS_PRE_CONSUME_SCRIPT` or `PAPERLESS_POST_CONSUME_SCRIPT`. The script will be executed before or or after the document is consumed respectively.

Important: These scripts are executed in a **blocking** process, which means that if a script takes a long time to run, it can significantly slow down your document consumption flow. If you want things to run asynchronously, you'll have to fork the process in your script and exit.

What Can These Scripts Do?

It's your script, so you're only limited by your imagination and the laws of physics. However, the following values are passed to the scripts in order:

Pre-consumption script

- Document file name

Post-consumption script

- Document id
- Generated file name
- Source path
- Thumbnail path
- Download URL
- Thumbnail URL
- Correspondent
- Tags

The script can be in any language you like, but for a simple shell script example, you can take a look at `post-consumption-example.sh` in the `scripts` directory in this project.

IMAP (Email)

Another handy way to get documents into your database is to email them to yourself. The typical use-case would be to be out for lunch and want to send a copy of the receipt back to your system at home. Paperless can be taught to pull emails down from an arbitrary account and dump them into the consumption directory where the process *above* will follow the usual pattern on consuming the document.

Some things you need to know about this feature:

- It's disabled by default. By setting the values below it will be enabled.
- It's been tested in a limited environment, so it may not work for you (please submit a pull request if you can!)
- It's designed to **delete mail from the server once consumed**. So don't go pointing this to your personal email account and wonder where all your stuff went.
- Currently, only one photo (attachment) per email will work.

So, with all that in mind, here's what you do to get it running:

1. Setup a new email account somewhere, or if you're feeling daring, create a folder in an existing email box and note the path to that folder.
2. In `/etc/paperless.conf` set all of the appropriate values in `PATHS AND FOLDERS` and `SECURITY`. If you decided to use a subfolder of an existing account, then make sure you set `PAPERLESS_CONSUME_MAIL_INBOX` accordingly here. You also have to set the `PAPERLESS_EMAIL_SECRET` to something you can remember 'cause you'll have to include that in every email you send.

3. Restart the *consumer*. The consumer will check the configured email account at startup and from then on every 10 minutes for something new and pulls down whatever it finds.
4. Send yourself an email! Note that the subject is treated as the file name, so if you set the subject to `Correspondent - Title - tag,tag,tag`, you'll get what you expect. Also, you must include the aforementioned secret string in every email so the fetcher knows that it's safe to import. Note that Paperless only allows the email title to consist of safe characters to be imported. These consist of alpha-numeric characters and `-_ ,.'`.
5. After a few minutes, the consumer will poll your mailbox, pull down the message, and place the attachment in the consumption directory with the appropriate name. A few minutes later, the consumer will import it like any other file.

HTTP POST

You can also submit a document via HTTP POST, so long as you do so after authenticating. To push your document to Paperless, send an HTTP POST to the server with the following name/value pairs:

- `correspondent`: The name of the document's correspondent. Note that there are restrictions on what characters you can use here. Specifically, alphanumeric characters, -, ,, ., and ' are ok, everything else is out. You also can't use the sequence ' - ' (space, dash, space).
- `title`: The title of the document. The rules for characters is the same here as the correspondent.
- `document`: The file you're uploading

Specify `enctype="multipart/form-data"`, and then POST your file with:

```
Content-Disposition: form-data; name="document"; filename="whatever.pdf"
```

An example of this in HTML is a typical form:

```
<form method="post" enctype="multipart/form-data">
    <input type="text" name="correspondent" value="My Correspondent" />
    <input type="text" name="title" value="My Title" />
    <input type="file" name="document" />
    <input type="submit" name="go" value="Do the thing" />
</form>
```

But a potentially more useful way to do this would be in Python. Here we use the requests library to handle basic authentication and to send the POST data to the URL.

```
import os

from hashlib import sha256

import requests
from requests.auth import HTTPBasicAuth

# You authenticate via BasicAuth or with a session id.
# We use BasicAuth here
username = "my-username"
password = "my-super-secret-password"

# Where you have Paperless installed and listening
url = "http://localhost:8000/push"

# Document metadata
```

```
correspondent = "Test Correspondent"
title = "Test Title"

# The local file you want to push
path = "/path/to/some/directory/my-document.pdf"


with open(path, "rb") as f:

    response = requests.post(
        url=url,
        data={"title": title,  "correspondent": correspondent},
        files={"document": (os.path.basename(path), f, "application/pdf")},
        auth=HTTPBasicAuth(username, password),
        allow_redirects=False
    )

    if response.status_code == 202:

        # Everything worked out ok
        print("Upload successful")

    else:

        # If you don't get a 202, it's probably because your credentials
        # are wrong or something.  This will give you a rough idea of what
        # happened.

        print("We got HTTP status code: {}".format(response.status_code))
        for k, v in response.headers.items():
            print("{}: {}".format(k, v))
```

The REST API

Paperless makes use of the Django REST Framework standard API interface because of its inherent awesomeness. Conveniently, the system is also self-documenting, so to learn more about the access points, schema, what's accepted and what isn't, you need only visit `/api` on your local Paperless installation.

Uploading

File uploads in an API are hard and so far as I've been able to tell, there's no standard way of accepting them, so rather than crowbar file uploads into the REST API and endure that headache, I've left that process to a simple HTTP POST, documented on the *consumption page*.

Utilities

There's basically three utilities to Paperless: the webserver, consumer, and if needed, the exporter. They're all detailed here.

The Webserver

At the heart of it, Paperless is a simple Django webservice, and the entire interface is based on Django's standard admin interface. Once running, visiting the URL for your service delivers the admin, through which you can get a detailed listing of all available documents, search for specific files, and download whatever it is you're looking for.

How to Use It

The webserver is started via the `manage.py` script:

```
$ /path/to/paperless/src/manage.py runserver
```

By default, the server runs on localhost, port 8000, but you can change this with a few arguments, run `manage.py --help` for more information.

Note that this command runs continuously, so exiting it will mean your webserver disappears. If you want to run this full-time (which is kind of the point) you'll need to have it start in the background – something you'll need to figure out for your own system. To get you started though, there are Systemd service files in the `scripts` directory.

The Consumer

The consumer script runs in an infinite loop, constantly looking at a directory for PDF files to parse and index. The process is pretty straightforward:

1. Look in `CONSUMPTION_DIR` for a PDF. If one is found, go to #2. If not, wait 10 seconds and try again.
2. Parse the PDF with Tesseract
3. Create a new record in the database with the OCR'd text
4. Attempt to automatically assign document attributes by doing some guesswork. Read up on the *guesswork documentation* for more information about this process.
5. Encrypt the PDF and store it in the `media` directory under `documents/pdf`.
6. Go to #1.

How to Use It

The consumer is started via the `manage.py` script:

```
$ /path/to/paperless/src/manage.py document_consumer
```

This starts the service that will run in a loop, consuming PDF files as they appear in `CONSUMPTION_DIR`.

Note that this command runs continuously, so exiting it will mean your webserver disappears. If you want to run this full-time (which is kind of the point) you'll need to have it start in the background – something you'll need to figure out for your own system. To get you started though, there are Systemd service files in the `scripts` directory.

The Exporter

Tired of fiddling with Paperless, or just want to do something stupid and are afraid of accidentally damaging your files? You can export all of your PDFs into neatly named, dated, and unencrypted.

How to Use It

This too is done via the `manage.py` script:

```
$ /path/to/paperless/src/manage.py document_exporter /path/to/somewhere/
```

This will dump all of your unencrypted PDFs into `/path/to/somewhere` for you to do with as you please. The files are accompanied with a special file, `manifest.json` which can be used to *import the files* at a later date if you wish.

Docker

If you are *using Docker*, running the expoorter is almost as easy. To mount a volume for exports, follow the instructions in the `docker-compose.yml.example` file for the `/export` volume (making the changes in your own `docker-compose.yml` file, of course). Once you have the volume mounted, the command to run an export is:

```
$ docker-compose run --rm consumer document_exporter /export
```

If you prefer to use `docker run` directly, supplying the necessary commandline options:

```
$ # Identify your containers
$ docker-compose ps
        Name                        Command                State     Ports
-------------------------------------------------------------------------------
paperless_consumer_1    /sbin/docker-entrypoint.sh ...   Exit 0
paperless_webserver_1   /sbin/docker-entrypoint.sh ...   Exit 0

$ # Make sure to replace your passphrase and remove or adapt the id mapping
$ docker run --rm \
    --volumes-from paperless_data_1 \
    --volume /path/to/arbitrary/place:/export \
    -e PAPERLESS_PASSPHRASE=YOUR_PASSPHRASE \
    -e USERMAP_UID=1000 -e USERMAP_GID=1000 \
    paperless document_exporter /export
```

The Importer

Looking to transfer Paperless data from one instance to another, or just want to restore from a backup? This is your go-to toy.

How to Use It

The importer works just like the exporter. You point it at a directory, and the script does the rest of the work:

```
$ /path/to/paperless/src/manage.py document_importer /path/to/somewhere/
```

Docker

Assuming that you've already gone through the steps above in the *export* section, then the easiest thing to do is just re-use the `/export` path you already setup:

```
$ docker-compose run --rm consumer document_importer /export
```

Similarly, if you're not using docker-compose, you can adjust the export instructions above to do the import.

The Re-tagger

Say you've imported a few hundred documents and now want to introduce a tag and apply its matching to all of the currently-imported docs. This problem is common enough that there's a tool for it.

How to Use It

This too is done via the `manage.py` script:

```
$ /path/to/paperless/src/manage.py document_retagger
```

That's it. It'll loop over all of the documents in your database and attempt to match all of your tags to them. If one matches, it'll be applied. And don't worry, you can run this as often as you like, it' won't double-tag a document.

Guesswork

During the consumption process, Paperless tries to guess some of the attributes of the document it's looking at. To do this it uses two approaches:

File Naming

Any document you put into the consumption directory will be consumed, but if you name the file right, it'll automatically set some values in the database for you. This is is the logic the consumer follows:

1. Try to find the correspondent, title, and tags in the file name following the pattern: `Date - Correspondent - Title - tag,tag,tag.pdf`. Note that the format of the date is **rigidly defined** as `YYYYMMDDHHMMSSZ` or `YYYYMMDDZ`. The `Z` refers "Zulu time" AKA "UTC".
2. If that doesn't work, we skip the date and try this pattern: `Correspondent - Title - tag,tag,tag.pdf`.
3. If that doesn't work, we try to find the correspondent and title in the file name following the pattern: `Correspondent - Title.pdf`.
4. If that doesn't work, just assume that the name of the file is the title.

So given the above, the following examples would work as you'd expect:

- `20150314000700Z - Some Company Name - Invoice 2016-01-01 - money,invoices.pdf`
- `20150314Z - Some Company Name - Invoice 2016-01-01 - money,invoices.pdf`
- `Some Company Name - Invoice 2016-01-01 - money,invoices.pdf`
- `Another Company - Letter of Reference.jpg`
- `Dad's Recipe for Pancakes.png`

These however wouldn't work:

- `2015-03-14 00:07:00 UTC - Some Company Name, Invoice 2016-01-01, money, invoices.pdf`
- `2015-03-14 - Some Company Name, Invoice 2016-01-01, money, invoices.pdf`
- `Some Company Name, Invoice 2016-01-01, money, invoices.pdf`
- `Another Company- Letter of Reference.jpg`

Reading the Document Contents

After the consumer has tried to figure out what it could from the file name, it starts looking at the content of the document itself. It will compare the matching algorithms defined by every tag and correspondent already set in your database to see if they apply to the text in that document. In other words, if you defined a tag called `Home Utility` that had a `match` property of `bc hydro` and a `matching_algorithm` of `literal`, Paperless will automatically tag your newly-consumed document with your `Home Utility` tag so long as the text `bc hydro` appears in the body of the document somewhere.

The matching logic is quite powerful, and supports searching the text of your document with different algorithms, and as such, some experimentation may be necessary to get things Just Right.

How Do I Set Up These Matching Algorithms?

Setting up of the algorithms is easily done through the admin interface. When you create a new correspondent or tag, there are optional fields for matching text and matching algorithm. From the help info there:

Note: Which algorithm you want to use when matching text to the OCR'd PDF. Here, "any" looks for any occurrence of any word provided in the PDF, while "all" requires that every word provided appear in the PDF, albeit not in the order provided. A "literal" match means that the text you enter must appear in the PDF exactly as you've entered it, and "regular expression" uses a regex to match the PDF. If you don't know what a regex is, you probably don't want this option.

Then just save your tag/correspondent and run another document through the consumer. Once complete, you should see the newly-created document, automatically tagged with the appropriate data.

Migrating, Updates, and Backups

As Paperless is still under active development, there's a lot that can change as software updates roll out. You should backup often, so if anything goes wrong during an update, you at least have a means of restoring to something usable. Thankfully, there are automated ways of backing up, restoring, and updating the software.

Backing Up

So you're bored of this whole project, or you want to make a remote backup of the unencrypted files for whatever reason. This is easy to do, simply use the *exporter* to dump your documents and database out into an arbitrary directory.

Restoring

Restoring your data is just as easy, since nearly all of your data exists either in the file names, or in the contents of the files themselves. You just need to create an empty database (just follow the *installation instructions* again) and then import the `tags.json` file you created as part of your backup. Lastly, copy your exported documents into the consumption directory and start up the consumer.

```
$ cd /path/to/project
$ rm data/db.sqlite3  # Delete the database
$ cd src
$ ./manage.py migrate  # Create the database
$ ./manage.py createsuperuser
$ ./manage.py loaddata /path/to/arbitrary/place/tags.json
$ cp /path/to/exported/docs/* /path/to/consumption/dir/
$ ./manage.py document_consumer
```

Importing your data if you are *using Docker* is almost as simple:

```
# Stop and remove your current containers
$ docker-compose stop
$ docker-compose rm -f

# Recreate them, add the superuser
$ docker-compose up -d
$ docker-compose run --rm webserver createsuperuser

# Load the tags
$ cat /path/to/arbitrary/place/tags.json | docker-compose run --rm webserver loaddata_
↪stdin -

# Load your exported documents into the consumption directory
# (How you do this highly depends on how you have set this up)
$ cp /path/to/exported/docs/* /path/to/mounted/consumption/dir/
```

After loading the documents into the consumption directory the consumer will immediately start consuming the documents.

Updates

For the most part, all you have to do to update Paperless is run `git pull` on the directory containing the project files, and then use Django's `migrate` command to execute any database schema updates that might have been rolled in as part of the update:

```
$ cd /path/to/project
$ git pull
$ cd src
$ ./manage.py migrate
```

Note that it's possible (even likely) that while `git pull` may update some files, the `migrate` step may not update anything. This is totally normal.

Additionally, as new features are added, the ability to control those features is typically added by way of an environment variable set in `paperless.conf`. You may want to take a look at the `paperless.conf.example` file to see if there's anything new in there compared to what you've got int `/etc`.

If you are *using Docker* the update process requires only one additional step:

```
$ cd /path/to/project
$ git pull
$ docker build -t paperless .
$ docker-compose up -d
$ docker-compose run --rm webserver migrate
```

If `git pull` doesn't report any changes, there is no need to continue with the remaining steps.

Troubleshooting

Consumer warns `OCR for XX failed`

If you find the OCR accuracy to be too low, and/or the document consumer warns that `OCR for XX failed, but we're going to stick with what we've got since FORGIVING_OCR is enabled`, then you might need to install the Tesseract language files marching your document's languages.

As an example, if you are running Paperless from the Vagrant setup provided (or from any Ubuntu or Debian box), and your documents are written in Spanish you may need to run:

```
apt-get install -y tesseract-ocr-spa
```

Consumer dies with `convert: unable to extent pixel cache`

During the consumption process, Paperless invokes ImageMagick's `convert` program to translate the source document into something that the OCR engine can understand and this can burn a Very Large amount of memory if the original document is rather long. Similarly, if your system doesn't have a lot of memory to begin with (ie. a Raspberry Pi), then this can happen for even medium-sized documents.

The solution is to tell ImageMagick *not* to Use All The RAM, as is its default, and instead tell it to used a fixed amount. `convert` will then break up the job into hundreds of individual files and use them to slowly compile the finished image. Simply set `PAPERLESS_CONVERT_MEMORY_LIMIT` in `/etc/paperless.conf` to something like `32000000` and you'll limit `convert` to 32MB. Fiddle with this value as you like.

HOWEVER: Simply setting this value may not be enough on system where `/tmp` is mounted as tmpfs, as this is where `convert` will write its temporary files. In these cases (most Systemd machines), you need to tell ImageMagick to use a different space for its scratch work. You do this by setting `PAPERLESS_CONVERT_TMPDIR` in `/etc/paperless.conf` to somewhere that's actually on a physical disk (and writable by the user running Paperless), like `/var/tmp/paperless` or `/home/my_user/tmp` in a pinch.

DecompressionBombWarning and/or no text in the OCR output

Some users have had issues using Paperless to consume PDFs that were created by merging Very Large Scanned Images into one PDF. If this happens to you, it's likely because the PDF you've created contains some very large pages (millions of pixels) and the process of converting the PDF to a OCR-friendly image is exploding.

Typically, this happens because the scanned images are created with a high DPI and then rolled into the PDF with an assumed DPI of 72 (the default). The best solution then is to specify the DPI used in the scan in the conversion-to-PDF step. So for example, if you scanned the original image with a DPI of 300, then merging the images into the single PDF with `convert` should look like this:

```
$ convert -density 300 *.jpg finished.pdf
```

For more information on this and situations like it, you should take a look at Issue #118 as that's where this tip originated.

Changelog

- 0.7.0 * **Potentially breaking change**: As per #235, Paperless will no longer

  automatically delete documents attached to correspondents when those correspondents are themselves deleted. This was Django's default behaviour, but didn't make much sense in Paperless' case. Thanks to Thomas Brueggemann and David Martin for their input on this one.

  – Fix for #232 wherein Paperless wasn't recognising `.tif` files properly. Thanks to ayounggun for reporting this one and to Kusti Skytén for posting the correct solution in the Github issue.

- 0.6.0 * Abandon the shared-secret trick we were using for the POST API in favour

  of BasicAuth or Django session.

  – Fix the POST API so it actually works. #236

  – **Breaking change**: We've dropped the use of `PAPERLESS_SHARED_SECRET` as it was being used both for the API (now replaced with a normal auth) and form email polling. Now that we're only using it for email, this variable has been renamed to `PAPERLESS_EMAIL_SECRET`. The old value will still work for a while, but you should change your config if you've been using the email polling feature. Thanks to Joshua Gilman for all the help with this feature.

- 0.5.0 * Support for fuzzy matching in the auto-tagger & auto-correspondent systems

  thanks to Jake Gysland's patch #220.

  – Modified the Dockerfile to prepare an export directory (#212). Thanks to combined efforts from Pit and Strubbl in working out the kinks on this one.

  – Updated the import/export scripts to include support for thumbnails. Big thanks to CkuT for finding this shortcoming and doing the work to get it fixed in #224.

  – All of the following changes are thanks to David Martin: * Bumped the dependency on pyocr to 0.4.7 so new users can make use of Tesseract 4 if they so prefer (#226). * Fixed a number of issues with the automated mail handler (#227, #228) * Amended the documentation for better handling of systemd service files (#229) * Amended the Django Admin configuration to have nice headers (#230)

- 0.4.1 * Fix for #206 wherein the pluggable parser didn't recognise files with

  all-caps suffixes like `.PDF`

- 0.4.0 * Introducing reminders. See #199 for more information, but the short

  explanation is that you can now attach simple notes & times to documents which are made available via the API. Currently, the default API (basically just the Django admin) doesn't really make use of this, but Thomas Brueggemann over at Paperless Desktop has said that he would like to make use of this feature in his project.

- 0.3.6 * Fix for #200 (!!) where the API wasn't configured to allow updating the

  correspondent or the tags for a document.

  – The `content` field is now optional, to allow for the edge case of a purely graphical document.

  – You can no longer add documents via the admin. This never worked in the first place, so all I've done here is remove the link to the broken form.

  - The consumer code has been heavily refactored to support a pluggable interface. Install a paperless consumer via pip and tell paperless about it with an environment variable, and you're good to go. Proper documentation is on its way.

- 0.3.5 * A serious facelift for the documents listing page wherein we drop the

  tabular layout in favour of a tiled interface.

  - Users can now configure the number of items per page.
  - Fix for #171: Allow users to specify their own `SECRET_KEY` value.
  - Moved the dotenv loading to the top of settings.py
  - Fix for #112: Added checks for binaries required for document consumption.

- 0.3.4 * Removal of django-suit due to a licensing conflict I bumped into in 0.3.3.

  Note that you *can* use Django Suit with Paperless, but only in a non-profit situation as their free license prohibits for-profit use. As a result, I can't bundle Suit with Paperless without conflicting with the GPL. Further development will be done against the stock Django admin.

  - I shrunk the thumbnails a little 'cause they were too big for me, even on my high-DPI monitor.
  - BasicAuth support for document and thumbnail downloads, as well as the Push API thanks to @thomas-brueggemann. See #179.

- 0.3.3 * Thumbnails in the UI and a Django-suit -based face-lift courtesy of @ekw! * Timezone, items per page, and default language are now all configurable,

  also thanks to @ekw.

- 0.3.2 * Fix for #172: defaulting ALLOWED_HOSTS to `["*"]` and allowing the

  user to set her own value via `PAPERLESS_ALLOWED_HOSTS` should the need arise.

- 0.3.1 * Added a default value for `CONVERT_BINARY`

- 0.3.0 * Updated to using django-filter 1.x * Added some system checks so new users aren't confused by misconfigurations. * Consumer loop time is now configurable for systems with slow writes. Just

  set `PAPERLESS_CONSUMER_LOOP_TIME` to a number of seconds. The default is 10.

  - As per #44, we've removed support for `PAPERLESS_CONVERT`, `PAPERLESS_CONSUME`, and `PAPERLESS_SECRET`. Please use `PAPERLESS_CONVERT_BINARY`, `PAPERLESS_CONSUMPTION_DIR`, and `PAPERLESS_SHARED_SECRET` respectively instead.

- 0.2.0
  - #150: The media root is now a variable you can set in `paperless.conf`.
  - #148: The database location (sqlite) is now a variable you can set in `paperless.conf`.
  - #146: Fixed a bug that allowed unauthorised access to the `/fetch` URL.
  - #131: Document files are now automatically removed from disk when they're deleted in Paperless.
  - #121: Fixed a bug where Paperless wasn't setting document creation time based on the file naming scheme.
  - #81: Added a hook to run an arbitrary script after every document is consumed.
  - #98: Added optional environment variables for ImageMagick so that it doesn't explode when handling Very Large Documents or when it's just running on a low-memory system. Thanks to Florian Harr for his help on this one.
  - #89 Ported the auto-tagging code to correspondents as well. Thanks to Justin Snyman for the pointers in the issue queue.

  - Added support for guessing the date from the file name along with the correspondent, title, and tags. Thanks to Tikitu de Jager for his pull request that I took forever to merge and to Pit for his efforts on the regex front.
  - #94: Restored support for changing the created date in the UI. Thanks to Martin Honermeyer and Tim White for working with me on this.

- 0.1.1
  - Potentially **Breaking Change**: All references to "sender" in the code have been renamed to "correspondent" to better reflect the nature of the property (one could quite reasonably scan a document before sending it to someone.)
  - #67: Rewrote the document exporter and added a new importer that allows for full metadata retention without depending on the file name and modification time. A big thanks to Tikitu de Jager, Pit, Florian Jung, and Christopher Luu for their code snippets and contributing conversation that lead to this change.
  - #20: Added *unpaper* support to help in cleaning up the scanned image before it's OCR'd. Thanks to Pit for this one.
  - #71 Added (encrypted) thumbnails in anticipation of a proper UI.
  - #68: Added support for using a proper config file at `/etc/paperless.conf` and modified the systemd unit files to use it.
  - Refactored the Vagrant installation process to use environment variables rather than asking the user to modify `settings.py`.
  - #44: Harmonise environment variable names with constant names.
  - #60: Setup logging to actually use the Python native logging framework.
  - #53: Fixed an annoying bug that caused `.jpeg` and `.JPG` images to be imported but made unavailable.

- 0.1.0
  - Docker support! Big thanks to Wayne Werner, Brian Conn, and Tikitu de Jager for this one, and especially to Pit who spearheadded this effort.
  - A simple REST API is in place, but it should be considered unstable.
  - Cleaned up the consumer to use temporary directories instead of a single scratch space. (Thanks Pit)
  - Improved the efficiency of the consumer by parsing pages more intelligently and introducing a threaded OCR process (thanks again Pit).
  - #45: Cleaned up the logic for tag matching. Reported by darkmatter.
  - #47: Auto-rotate landscape documents. Reported by Paul and fixed by Pit.
  - #48: Matching algorithms should do so on a word boundary (darkmatter)
  - #54: Documented the re-tagger (zedster)
  - #57: Make sure file is preserved on import failure (darkmatter)
  - Added tox with pep8 checking

- 0.0.6
  - Added support for parallel OCR (significant work from Pit)
  - Sped up the language detection (significant work from Pit)
  - Added simple logging

- 0.0.5

  - Added support for image files as documents (png, jpg, gif, tiff)
  - Added a crude means of HTTP POST for document imports
  - Added IMAP mail support
  - Added a re-tagging utility
  - Documentation for the above as well as data migration
- 0.0.4
  - Added automated tagging basted on keyword matching
  - Cleaned up the document listing page
  - Removed `User` and `Group` from the admin
  - Added `pytz` to the list of requirements
- 0.0.3
  - Added basic tagging
- 0.0.2
  - Added language detection
  - Added datestamps to `document_exporter`.
  - Changed `settings.TESSERACT_LANGUAGE` to `settings.OCR_LANGUAGE`.
- 0.0.1
  - Initial release