GNU MediaGoblin Documentation

Release 0.5.1

Chris Webber, et al

September 27, 2013

CONTENTS

GNU MediaGoblin is a platform for sharing photos, video and other media in an environment that respects our freedom and independence.

This is a Free Software project. It is built by contributors for all to use and enjoy. If you're intrested in contributing, see the wiki which has pages that talk about the ways someone can contribute.

CHAPTER

ONE

PART 1: SITE ADMINISTRATOR'S GUIDE

This guide covers installing, configuring, deploying and running a GNU MediaGoblin website. It is written for site administrators.

1.1 Foreword

1.1.1 About the Site Administrator's Guide

This is the site administrator manual for GNU MediaGoblin. It covers how to set up and configure MediaGoblin and the kind of information that someone running MediaGoblin would need to know.

We have other documentation at:

- http://mediagoblin.org/join/ for general "join us" information
- http://wiki.mediagoblin.org/ for our contributor/developer-focused wiki

1.1.2 Improving the Site Administrator's Guide

There are a few ways—please pick whichever method is convenient for you!

1. Write up a bug report in the bug tracker
2. Tell someone on IRC `#mediagoblin` on Freenode.
3. Write an email to the devel mailing list.

Information about the bugtracker, IRC and the mailing list is all on the join page.

Patches are the most helpful, but even feedback on what you think could be improved and how to improve it is also helpful.

1.2 About GNU MediaGoblin

Sections

1.2.1 What is GNU MediaGoblin?

In 2008, a number of free software developers and activists gathered at the FSF to attempt to answer the question "What should software freedom look like on the participatory web?" Their answer, the Franklin Street Statement has lead to the development of autonomo.us community, and free software projects including Identi.ca and Libre.fm.

Identi.ca and Libre.fm address the need for micro-blogging and music sharing services and software that respect users' freedom and autonomy.

GNU MediaGoblin emerges from this milieu to create a platform for us to share photos, video and other media in an environment that respects our freedom and independence. In the future MediaGoblin will provide tools to facilitate collaboration on media projects.

1.2.2 Why Build GNU MediaGoblin?

The Internet is designed—and works best—as a complex and endlessly resilient network. When key services and media outlets are concentrated in centralized platforms, the network becomes less useful and increasingly fragile. As always, the proprietary nature of these systems, hinders users ability to develop, extend, and understand their software; however, in the case of network services it also means that users must forfeit control of their data to the service providers.

Therefore, we believe that network services must be federated to avoid centralization and that everyone ought to have control over their data. In support of this, we've decided to help build the tools to make these kinds of services possible. We hope you'll join us, both as users and as contributors.

1.2.3 Who Contributes to the Project?

You do!

We are free software activists and folks who have worked on a variety of other projects including: Libre.fm, GNU Social, Status.net, Miro, Miro Community, and OpenHatch among others. We're programmers, musicians, writers, and painters. We're friendly and dedicated to software and network freedom.

1.2.4 How Can I Participate?

See Get Involved on the website. We eagerly look forward to seeing you!

1.2.5 How is GNU MediaGoblin licensed?

GNU MediaGoblin software is released under an AGPLv3 license.

See the `COPYING` file in the root of the source for details.

1.2.6 Is MediaGoblin an official GNU project? What does that mean?

MediaGoblin is an official GNU project! This status means that we the meet the GNU Project's rigorous standards for free software. To find out more about what that means, check out the GNU website.

Please feel free to contact us with further questions!

1.3 Deploying MediaGoblin

GNU MediaGoblin is fairly new and so at the time of writing, there aren't easy package-manager-friendly methods to install MediaGoblin. However, doing a basic install isn't too complex in and of itself.

There's an almost infinite way to deploy things... for now, we'll keep it simple with some assumptions and use a setup that combines mediagoblin + virtualenv + fastcgi + nginx on a .deb or .rpm based GNU/Linux distro.

Note: These tools are for site administrators wanting to deploy a fresh install. If instead you want to join in as a contributor, see our Hacking HOWTO instead.

There are also many ways to install servers... for the sake of simplicity, our instructions below describe installing with nginx. For more recipes, including Apache, see our wiki.

1.3.1 Prepare System

Dependencies

MediaGoblin has the following core dependencies:

- Python 2.6 or 2.7
- python-lxml
- git
- SQLite/PostgreSQL
- Python Imaging Library (PIL)
- virtualenv

On a DEB-based system (e.g Debian, gNewSense, Trisquel, Ubuntu, and derivatives) issue the following command:

```
sudo apt-get install git-core python python-dev python-lxml \
    python-imaging python-virtualenv
```

On a RPM-based system (e.g. Fedora, RedHat, and derivatives) issue the following command:

```
yum install python-paste-deploy python-paste-script \
    git-core python python-devel python-lxml python-imaging \
    python-virtualenv
```

Configure PostgreSQL

Note: MediaGoblin currently supports PostgreSQL and SQLite. The default is a local SQLite database. This will "just work" for small deployments.

For medium to large deployments we recommend PostgreSQL.

If you don't want/need postgres, skip this section.

These are the packages needed for Debian Wheezy (stable):

```
sudo apt-get install postgresql postgresql-client python-psycopg2
```

The installation process will create a new *system* user named `postgres`, it will have privilegies sufficient to manage the database. We will create a new database user with restricted privilegies and a new database owned by our restricted database user for our MediaGoblin instance.

In this example, the database user will be `mediagoblin` and the database name will be `mediagoblin` too.

To create our new user, run:

```
sudo -u postgres createuser mediagoblin
```

then answer NO to *all* the questions:

```
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

then create the database all our MediaGoblin data should be stored in:

```
sudo -u postgres createdb -E UNICODE -O mediagoblin mediagoblin
```

where the first `mediagoblin` is the database owner and the second `mediagoblin` is the database name.

> **Caution:** Where is the password?
> These steps enable you to authenticate to the database in a password-less manner via local UNIX authentication provided you run the MediaGoblin application as a user with the same name as the user you created in PostgreSQL. More on this in *Drop Privileges for MediaGoblin*.

Drop Privileges for MediaGoblin

MediaGoblin does not require special permissions or elevated access to run. As such, the prefered way to run MediaGoblin is to create a dedicated, unpriviledged system user for sole the purpose of running MediaGoblin. Running MediaGoblin processes under an unpriviledged system user helps to keep it more secure.

The following command (entered as root or with sudo) will create a system account with a username of `mediagoblin`. You may choose a different username if you wish.:

```
adduser --system mediagoblin
```

No password will be assigned to this account, and you will not be able to log in as this user. To switch to this account, enter either:

```
sudo su - mediagoblin (if you have sudo permissions)
```

or:

```
su - mediagoblin (if you have to use root permissions)
```

You may get a warning similar to this when entering these commands:

```
warning: cannot change directory to /home/mediagoblin: No such file or directory
```

You can disregard this warning. To return to your regular user account after using the system account, just enter `exit`.

Note: Unless otherwise noted, the remainder of this document assumes that all operations are performed using this unpriviledged account.

Create a MediaGoblin Directory

You should create a working directory for MediaGoblin. This document assumes your local git repository will be located at `/srv/mediagoblin.example.org/mediagoblin/`. Substitute your prefered local deployment path as needed.

Setting up the working directory requires that we first create the directory with elevated priviledges, and then assign ownership of the directory to the unpriviledged system account.

To do this, enter either of the following commands, changing the defaults to suit your particular requirements:

```
sudo mkdir -p /srv/mediagoblin.example.org && sudo chown -hR mediagoblin:mediagoblin /srv/mediagobin
```

or (as the root user):

```
mkdir -p /srv/mediagoblin.example.org && chown -hR mediagoblin:mediagoblin /srv/mediagobin.example.or
```

1.3.2 Install MediaGoblin and Virtualenv

Note: MediaGoblin is still developing rapidly. As a result the following instructions recommend installing from the `master` branch of the git repository. Eventually production deployments will want to transition to running from more consistent releases.

We will now clone the MediaGoblin source code repository and setup and configure the necessary services. Modify these commands to suit your own environment. As a reminder, you should enter these commands using your unpriviledged system account.

Change to the MediaGoblin directory that you just created:

```
cd /srv/mediagoblin.example.org
```

Clone the MediaGoblin repository and set up the git submodules:

```
git clone git://gitorious.org/mediagoblin/mediagoblin.git
cd mediagoblin
git submodule init && git submodule update
```

And set up the in-package virtualenv:

```
(virtualenv --system-site-packages . || virtualenv .) && ./bin/python setup.py develop
```

Note: We presently have an experimental make-style deployment system. if you'd like to try it, instead of the above command, you can run:

```
./bootstrap.sh && ./configure && make
```

This also includes a number of nice features, such as keeping your viratualenv up to date by simply running *make update*.

The above provides an in-package install of `virtualenv`. While this is counter to the conventional `virtualenv` configuration, it is more reliable and considerably easier to configure and illustrate. If you're familiar with Python packaging you may consider deploying with your preferred method.

Assuming you are going to deploy with FastCGI, you should also install flup:

```
./bin/easy_install flup
```

(Sometimes this breaks because flup's site is flakey. If it does for you, try):

```
./bin/easy_install https://pypi.python.org/pypi/flup/1.0.3.dev-20110405
```

This concludes the initial configuration of the development environment. In the future, when you update your codebase, you should also run:

```
./bin/python setup.py develop --upgrade && ./bin/gmg dbupdate && git submodule fetch
```

Note: If you are running an active site, depending on your server configuration, you may need to stop it first or the dbupdate command may hang (and it's certainly a good idea to restart it after the update)

1.3.3 Deploy MediaGoblin Services

Edit site configuration

A few basic properties must be set before MediaGoblin will work. First make a copy of `mediagoblin.ini` for editing so the original config file isn't lost:

```
cp mediagoblin.ini mediagoblin_local.ini
```

Then:

- Set `email_sender_address` to the address you wish to be used as the sender for system-generated emails
- Edit `direct_remote_path`, `base_dir`, and `base_url` if your mediagoblin directory is not the root directory of your vhost.

Configure MediaGoblin to use the PostgreSQL database

If you are using postgres, edit the `[mediagoblin]` section in your `mediagoblin_local.ini` and put in:

```
sql_engine = postgresql:///mediagoblin
```

if you are running the MediaGoblin application as the same 'user' as the database owner.

Update database data structures

Before you start using the database, you need to run:

```
./bin/gmg dbupdate
```

to populate the database with the MediaGoblin data structures.

Test the Server

At this point MediaGoblin should be properly installed. You can test the deployment with the following command:

```
./lazyserver.sh --server-name=broadcast
```

You should be able to connect to the machine on port 6543 in your browser to confirm that the service is operable.

FastCGI and nginx

This configuration example will use nginx, however, you may use any webserver of your choice as long as it supports the FastCGI protocol. If you do not already have a web server, consider nginx, as the configuration files may be more clear than the alternatives.

Create a configuration file at `/srv/mediagoblin.example.org/nginx.conf` and create a symbolic link into a directory that will be included in your `nginx` configuration (e.g. "`/etc/nginx/sites-enabled` or `/etc/nginx/conf.d`) with one of the following commands (as the root user):

```
ln -s /srv/mediagoblin.example.org/nginx.conf /etc/nginx/conf.d/
ln -s /srv/mediagoblin.example.org/nginx.conf /etc/nginx/sites-enabled/
```

Modify these commands and locations depending on your preferences and the existing configuration of your nginx instance. The contents of this `nginx.conf` file should be modeled on the following:

```
server {
 ##################################################
 # Stock useful config options, but ignore them :)
 ##################################################
 include /etc/nginx/mime.types;

 autoindex off;
 default_type  application/octet-stream;
 sendfile on;

 # Gzip
 gzip on;
 gzip_min_length 1024;
 gzip_buffers 4 32k;
 gzip_types text/plain text/html application/x-javascript text/javascript text/xml text/css;

 ####################################
 # Mounting MediaGoblin stuff
 # This is the section you should read
 ####################################

 # Change this to update the upload size limit for your users
 client_max_body_size 8m;

 # prevent attacks (someone uploading a .txt file that the browser
 # interprets as an HTML file, etc.)
 add_header X-Content-Type-Options nosniff;

 server_name mediagoblin.example.org www.mediagoblin.example.org;
 access_log /var/log/nginx/mediagoblin.example.access.log;
 error_log /var/log/nginx/mediagoblin.example.error.log;

 # MediaGoblin's stock static files: CSS, JS, etc.
 location /mgoblin_static/ {
```

```
    alias /srv/mediagoblin.example.org/mediagoblin/mediagoblin/static/;
 }

 # Instance specific media:
 location /mgoblin_media/ {
    alias /srv/mediagoblin.example.org/mediagoblin/user_dev/media/public/;
 }

 # Theme static files (usually symlinked in)
 location /theme_static/ {
    alias /srv/mediagoblin.example.org/mediagoblin/user_dev/theme_static/;
 }

 # Plugin static files (usually symlinked in)
 location /plugin_static/ {
    alias /srv/mediagoblin.example.org/mediagoblin/user_dev/plugin_static/;
 }

 # Mounting MediaGoblin itself via FastCGI.
 location / {
    fastcgi_pass 127.0.0.1:26543;
    include /etc/nginx/fastcgi_params;

    # our understanding vs nginx's handling of script_name vs
    # path_info don't match :)
    fastcgi_param PATH_INFO $fastcgi_script_name;
    fastcgi_param SCRIPT_NAME "";
 }
}
```

Now, nginx instance is configured to serve the MediaGoblin application. Perform a quick test to ensure that this configuration works. Restart nginx so it picks up your changes, with a command that resembles one of the following (as the root user):

```
sudo /etc/init.d/nginx restart
sudo /etc/rc.d/nginx restart
```

Now start MediaGoblin. Use the following command sequence as an example:

```
cd /srv/mediagoblin.example.org/mediagoblin/
./lazyserver.sh --server-name=fcgi fcgi_host=127.0.0.1 fcgi_port=26543
```

Visit the site you've set up in your browser by visiting <http://mediagoblin.example.org>. You should see MediaGoblin!

Note: The configuration described above is sufficient for development and smaller deployments. However, for larger production deployments with larger processing requirements, see the "*Considerations for Production Deployments*" documentation.

Apache

Instructions and scripts for running MediaGoblin on an Apache server can be found on the MediaGoblin wiki.

Security Considerations

Warning: The directory `user_dev/crypto/` contains some very sensitive files. Especially the `itsdangeroussecret.bin` is very important for session security. Make sure not to leak its contents anywhere. If the contents gets leaked nevertheless, delete your file and restart the server, so that it creates a new secret key. All previous sessions will be invalidated.

1.4 Considerations for Production Deployments

This document contains a number of suggestions for deploying MediaGoblin in actual production environments. Consider "*Deploying MediaGoblin*" for a basic overview of how to deploy MediaGoblin.

1.4.1 Deploy with Paste

The MediaGoblin WSGI application instance you get with `./lazyserver.sh` is not ideal for a production MediaGoblin deployment. Ideally, you should be able to use an "init" or "control" script to launch and restart the MediaGoblin process.

Use the following command as the basis for such a script:

```
CELERY_ALWAYS_EAGER=true \
 /srv/mediagoblin.example.org/mediagoblin/bin/paster serve \
 /srv/mediagoblin.example.org/mediagoblin/paste.ini \
 --pid-file=/var/run/mediagoblin.pid \
 --server-name=fcgi fcgi_host=127.0.0.1 fcgi_port=26543
```

The above configuration places MediaGoblin in "always eager" mode with Celery, this means that submissions of content will be processed synchronously, and the user will advance to the next page only after processing is complete. If we take Celery out of "always eager mode," the user will be able to immediately return to the MediaGoblin site while processing is ongoing. In these cases, use the following command as the basis for your script:

```
CELERY_ALWAYS_EAGER=false \
 /srv/mediagoblin.example.org/mediagoblin/bin/paster serve \
 /srv/mediagoblin.example.org/mediagoblin/paste.ini \
 --pid-file=/var/run/mediagoblin.pid \
 --server-name=fcgi fcgi_host=127.0.0.1 fcgi_port=26543
```

1.4.2 Separate Celery

MediaGoblin uses Celery to handle heavy and long-running tasks. Celery can be launched in two ways:

1. Embedded in the MediaGoblin WSGI application [1]. This is the way `./lazyserver.sh` does it for you. It's simple as you only have to run one process. The only bad thing with this is that the heavy and long-running tasks will run *in* the webserver, keeping the user waiting each time some heavy lifting is needed as in for example processing a video. This could lead to problems as an aborted connection will halt any processing and since most front-end web servers *will* terminate your connection if it doesn't get any response from the MediaGoblin WSGI application in a while.
2. As a separate process communicating with the MediaGoblin WSGI application via a broker. This offloads the heavy lifting from the MediaGoblin WSGI application and users will be able to continue to browse the site while the media is being processed in the background.

[1] The MediaGoblin WSGI application is the part that of MediaGoblin that processes HTTP requests.

To launch Celery separately from the MediaGoblin WSGI application:

1. Make sure that the `CELERY_ALWAYS_EAGER` environment variable is unset or set to `false` when launching the MediaGoblin WSGI application.
2. Start the `celeryd` main process with

```
CELERY_CONFIG_MODULE=mediagoblin.init.celery.from_celery ./bin/celeryd
```

1.4.3 Set up sentry to monitor exceptions

We have a plugin for raven integration, see the "*raven plugin*" documentation.

1.4.4 Use an Init Script

Look in your system's `/etc/init.d/` or `/etc/rc.d/` directory for examples of how to build scripts that will start, stop, and restart MediaGoblin and Celery. These scripts will vary by distribution/operating system.

These are scripts provided by the MediaGoblin community:

Debian

- GNU MediaGoblin init scripts by Joar Wandborg

Arch Linux

- MediaGoblin - ArchLinux rc.d scripts by Jeremy Pope
- Mediagoblin init script on Archlinux by Chimo

1.5 Configuring MediaGoblin

So! You've got MediaGoblin up and running, but you need to adjust some configuration parameters. Well you've come to the right place!

1.5.1 MediaGoblin's config files

When configuring MediaGoblin, there are two files you might want to make local modified versions of, and one extra file that might be helpful to look at. Let's examine these.

mediagoblin.ini This is the config file for MediaGoblin, the application. If you want to tweak settings for MediaGoblin, you'll usually tweak them here.

paste.ini This is primarily a server configuration file, on the Python side (specifically, on the WSGI side, via paste deploy / paste script). It also sets up some middleware that you can mostly ignore, except to configure sessions... more on that later. If you are adding a different Python server other than fastcgi / plain HTTP, you might configure it here. You probably won't need to change this file very much.

There's one more file that you certainly won't change unless you're making coding contributions to mediagoblin, but which can be useful to read and reference:

mediagoblin/config_spec.ini This file is actually a specification for mediagoblin.ini itself, as a config file! It defines types and defaults. Sometimes it's a good place to look for documentation... or to find that hidden option that we didn't tell you about. :)

1.5.2 Making local copies

Let's assume you're doing the virtualenv setup described elsewhere in this manual, and you need to make local tweaks to the config files. How do you do that? Let's see.

To make changes to mediagoblin.ini

```
cp mediagoblin.ini mediagoblin_local.ini
```

To make changes to paste.ini

```
cp paste.ini paste_local.ini
```

From here you should be able to make direct adjustments to the files, and most of the commands described elsewhere in this manual will "notice" your local config files and use those instead of the non-local version.

Note: Note that all commands provide a way to pass in a specific config file also, usually by a `-cf` flag.

1.5.3 Common changes

Enabling email notifications

You'll almost certainly want to enable sending email. By default, MediaGoblin doesn't really do this... for the sake of developer convenience, it runs in "email debug mode".

To make MediaGoblin send email, you need a mailer daemon.

Change this in your `mediagoblin.ini` file:

```
email_debug_mode = false
```

You should also change the "from" email address by setting `email_sender_address`. For example:

```
email_sender_address = "foo@example.com"
```

If you have more custom SMTP settings, you also have the following options at your disposal, which are all optional, and do exactly what they sound like.

- email_smtp_host
- email_smtp_port
- email_smtp_user
- email_smtp_pass

All other configuration changes

To be perfectly honest, there are quite a few options and we haven't had time to document them all.

So here's a cop-out section saying that if you get into trouble, hop onto IRC and we'll help you out. Details for the IRC channel is on the join page of the website.

1.5.4 Celery

FIXME: List Celery configuration here.

1.6 Media Types

In the future, there will be all sorts of media types you can enable, but in the meanwhile there are five additional media types: video, audio, ascii art, STL/3d models, PDF and Document.

First, you should probably read "*Configuring MediaGoblin*" to make sure you know how to modify the mediagoblin config file.

1.6.1 Enabling Media Types

Note: Media types are now plugins

Media types are enabled in your mediagoblin configuration file, typically it is created by copying `mediagoblin.ini` to `mediagoblin_local.ini` and then applying your changes to `mediagoblin_local.ini`. If you don't already have a `mediagoblin_local.ini`, create one in the way described.

Most media types have additional dependencies that you will have to install. You will find descriptions on how to satisfy the requirements of each media type on this page.

To enable a media type, add the the media type under the `[plugins]` section in you `mediagoblin_local.ini`. For example, if your system supported image and video media types, then it would look like this:

```
[plugins]
[[mediagoblin.media_types.image]]
[[mediagoblin.media_types.video]]
```

Note that after enabling new media types, you must run dbupdate like so:

```
./bin/gmg dbupdate
```

If you are running an active site, depending on your server configuration, you may need to stop it first (and it's certainly a good idea to restart it after the update).

1.6.2 How does MediaGoblin decide which media type to use for a file?

MediaGoblin has two methods for finding the right media type for an uploaded file. One is based on the file extension of the uploaded file; every media type maintains a list of supported file extensions. The second is based on a sniffing handler, where every media type may inspect the uploaded file and tell if it will accept it.

The file-extension-based approach is used before the sniffing-based approach, if the file-extension-based approach finds a match, the sniffing-based approach will be skipped as it uses far more processing power.

1.6.3 Video

To enable video, first install gstreamer and the python-gstreamer bindings (as well as whatever gstremaer extensions you want, good/bad/ugly). On Debianoid systems

```
sudo apt-get install python-gst0.10 \
    gstreamer0.10-plugins-base \
    gstreamer0.10-plugins-bad \
    gstreamer0.10-plugins-good \
    gstreamer0.10-plugins-ugly \
    gstreamer0.10-ffmpeg
```

Add `[[mediagoblin.media_types.video]]` under the `[plugins]` section in your `mediagoblin_local.ini` and restart MediaGoblin.

Run

```
./bin/gmg dbupdate
```

Now you should be able to submit videos, and mediagoblin should transcode them.

Note: You almost certainly want to separate Celery from the normal paste process or your users will probably find that their connections time out as the video transcodes. To set that up, check out the "*Considerations for Production Deployments*" section of this manual.

1.6.4 Audio

To enable audio, install the gstreamer and python-gstreamer bindings (as well as whatever gstreamer plugins you want, good/bad/ugly), scipy and numpy are also needed for the audio spectrograms. To install these on Debianoid systems, run:

```
sudo apt-get install python-gst0.10 gstreamer0.10-plugins-{base,bad,good,ugly} \
    gstreamer0.10-ffmpeg python-numpy python-scipy
```

The `scikits.audiolab` package you will install in the next step depends on the `libsndfile1-dev` package, so we should install it. On Debianoid systems, run

```
sudo apt-get install libsndfile1-dev
```

Note: scikits.audiolab will display a warning every time it's imported if you do not compile it with alsa support. Alsa support is not necessary for the GNU MediaGoblin application but if you do not wish the alsa warnings from audiolab you should also install `libasound2-dev` before installing scikits.audiolab.

Then install `scikits.audiolab` for the spectrograms:

```
./bin/pip install scikits.audiolab
```

Add `[[mediagoblin.media_types.audio]]` under the `[plugins]` section in your `mediagoblin_local.ini` and restart MediaGoblin.

Run

```
./bin/gmg dbupdate
```

You should now be able to upload and listen to audio files!

1.6.5 Ascii art

To enable ascii art support, first install the chardet library, which is necessary for creating thumbnails of ascii art

```
./bin/easy_install chardet
```

Next, modify (and possibly copy over from `mediagoblin.ini`) your `mediagoblin_local.ini`. In the `[plugins]` section, add `[[mediagoblin.media_types.ascii]]`.

Run

```
./bin/gmg dbupdate
```

Now any .txt file you uploaded will be processed as ascii art!

1.6.6 STL / 3d model support

To enable the "STL" 3d model support plugin, first make sure you have a recentish Blender installed and available on your execution path. This feature has been tested with Blender 2.63. It may work on some earlier versions, but that is not guaranteed (and is surely not to work prior to Blender 2.5X).

Add `[[mediagoblin.media_types.stl]]` under the `[plugins]` section in your `mediagoblin_local.ini` and restart MediaGoblin.

Run

```
./bin/gmg dbupdate
```

You should now be able to upload .obj and .stl files and MediaGoblin will be able to present them to your wide audience of admirers!

1.6.7 PDF and Document

To enable the "PDF and Document" support plugin, you need:

1. pdftocairo and pdfinfo for pdf only support.
2. unoconv with headless support to support converting libreoffice supported documents as well, such as doc/ppt/xls/odf/odg/odp and more. For the full list see mediagoblin/media_types/pdf/processing.py, unoconv_supported.

All executables must be on your execution path.

To install this on Fedora:

```
sudo yum install -y poppler-utils unoconv libreoffice-headless
```

Note: You can leave out unoconv and libreoffice-headless if you want only pdf support. This will result in a much smaller list of dependencies.

pdf.js relies on git submodules, so be sure you have fetched them:

```
git submodule init
git submodule update
```

This feature has been tested on Fedora with: poppler-utils-0.20.2-9.fc18.x86_64 unoconv-0.5-2.fc18.noarch libreoffice-headless-3.6.5.2-8.fc18.x86_64

It may work on some earlier versions, but that is not guaranteed.

Add `[[mediagoblin.media_types.pdf]]` under the `[plugins]` section in your `mediagoblin_local.ini` and restart MediaGoblin.

Run

```
./bin/gmg dbupdate
```

1.7 How to Get Help with MediaGoblin

There are a couple of ways to get help with problems with MediaGoblin:

1. ask for help on IRC
2. ask for help on the devel mailing list

Details for both IRC and the mailing list are on the join page of the website.

1.8 Release Notes

This chapter has important information for releases in it. If you're upgrading from a previous release, please read it carefully, or at least skim over it.

1.8.1 0.5.1

v0.5.1 is a bugfix release... the steps are the same as for 0.5.1.

Bugfixes:

- python 2.6 compatibility restored
- Fixed last release's release notes ;)

1.8.2 0.5.0

NOTE: If using the API is important to you, we're in a state of ransition towards a new API via the Pump API. As such, though the old API still probably works, some changes have happened to the way oauth works to make it more Pump-compatible. If you're heavily using clients using the old API, you may wish to hold off on upgrading for now. Otherwise, jump in and have fun! :)

Do this to upgrade

1. Make sure to run `./bin/python setup.py develop --upgrade && ./bin/gmg dbupdate` after upgrading.
2. Note that a couple of things have changed with `mediagoblin.ini`. First we have a new Authentication System. You need to add `[[mediagoblin.plugins.basic_auth]]` under the `[plugins]` section of your config file. Second, media types are now plugins, so you need to add each media type under the `[plugins]` section of your config file.
3. We have made a script to transition your `mediagoblin_local.ini` file for you. This script can be found at:

   http://mediagoblin.org/download/0.5.0_config_converter.py

If you run into problems, don't hesitate to contact us (IRC is often best).

New features

- As mentioned above, we now have a plugable Authentication system. You can use any combination of the multiple authentication systems (*basic_auth plugin*, *persona plugin*, *openid plugin*) or write your own!
- Media types are now plugins! This means that new media types will be able to do new, fancy things they couldn't in the future.

- We now have notification support! This allows you to subscribe to media comments and to be notified when someone comments on your media.
- New reprocessing framework! You can now reprocess failed uploads, and send already processed media back to processing to re-transcode or resize media.
- Comment preview!
- Users now have the ability to change their email associated with their account.
- New oauth code as we move closer to federation support.
- Experimental pyconfigure support for GNU-style configue and makefile deployment.
- Database foundations! You can now pre-populate the database models.
- Way faster unit test run-time via in-memory database.
- All mongokit stuff has been cleaned up.
- Fixes for non-ascii filenames.
- The option to stay logged in.
- Mediagoblin has been upgraded to use the latest celery version.
- You can now add jinja2 extensions to your config file to use in custom templates.
- Fixed video permission issues.
- Mediagoblin docs are now hosted with multiple versions.
- We removed redundent tooltips from the STL media display.
- We are now using itsdangerous for verification tokens.

1.8.3 0.4.1

This is a bugfix release for 0.4.0. This only implements one major fix in the newly released document support which prevented the "conversion via libreoffice" feature.

If you were running 0.4.0 you can upgrade to v0.4.1 via a simple switch and restarting mediagoblin/celery with no other actions.

Otherwise, follow 0.4.0 instructions.

1.8.4 0.4.0

Do this to upgrade

1. Make sure to run `./bin/python setup.py develop --upgrade && ./bin/gmg dbupdate` after upgrading.
2. See "For Theme authors" if you have a custom theme.
3. Note that `./bin/gmg theme assetlink` is now just `./bin/gmg assetlink` and covers both plugins and assets. Keep on reading to hear more about new plugin features.
4. If you want to take advantage of new plugins that have statically served assets, you are going to need to add the new "plugin_static" section to your nginx config. Basically the following for nginx:

```
# Plugin static files (usually symlinked in)
location /plugin_static/ {
   alias /srv/mediagoblin.example.org/mediagoblin/user_dev/plugin_static/;
}
```

Similarly, if you've got a modified paste config, you may want to borrow the app:plugin_static section from the default paste.ini file.

5. We now use itsdangerous for sessions; if you had any references to beaker in your paste config you can remove them. Again, see the default paste.ini config
6. We also now use git submodules. Please do: `git submodule init && git submodule update` You will need to do this to use the new PDF support.

For theme authors

If you have your own theme or you have any "user modified templates", please note the following:

- mediagoblin/bits/ files above-content.html, body-end.html, body-start.html now are renamed... they have underscores instead of dashes in the filenames now :)
- There's a new file: `mediagoblin/bits/frontpage_welcome.html`. You can easily customize this to give a welcome page appropriate to your site.

New features

- PDF media type!
- Improved plugin system. More flexible, better documented, with a new plugin authoring section of the docs.
- itsdangerous based sessions. No more beaker!
- New, experimental Piwigo-based API. This means you should be able to use MediaGoblin with something like Shotwell. (Again, a word of caution: this is *very experimental*!)
- Human readable timestamps, and the option to display the original date of an image when available (available as the "original_date_visible" variable)
- Moved unit testing system from nosetests to py.test so we can better handle issues with sqlalchemy exploding with different database configurations. Long story :)
- You can now disable the ability to post comments.
- Tags now can be up to length 255 characters by default.

1.8.5 0.3.3

Do this to upgrade

1. Make sure to run `bin/gmg dbupdate` after upgrading.
2. OpenStreetMap is now a plugin, so if you want to use it, add the following to your config file:

```
[plugins]
[[mediagoblin.plugins.geolocation]]
```

If you have your own theme, you may need to make some adjustments to it as some theme related things may have changed in this release. If you run into problems, don't hesitate to contact us (IRC is often best).

New features

- New dropdown menu for accessing various features.

- Significantly improved URL generation. Now mediagoblin won't give up on making a slug if it looks like there will be a duplicate; it'll try extra hard to generate a meaningful one instead.

  Similarly, linking to an id no longer can possibly conflict with linking to a slug; /u/username/m/id:35/ is the kind of reference we now use to linking to entries with ids. However, old links with entries that linked to ids should work just fine with our migration. The only urls that might break in this release are ones using colons or equal signs.

- New template hooks for plugin authoring.
- As a demonstration of new template hooks for plugin authoring, openstreetmap support now moved to a plugin!
- Method to add media to collections switched from icon of paperclip to button with "add to collection" text.
- Bug where videos often failed to produce a proper thumbnail fixed!
- Copying around files in MediaGoblin now much more efficient, doesn't waste gobs of memory.
- Video transcoding now optional for videos that meet certain criteria. By default, MediaGoblin will not transcode webm videos that are smaller in resolution than the MediaGoblin defaults, and MediaGoblin can also be configured to allow theora files to not be transcoded as well.
- Per-user license preference option; always want your uploads to be BY-SA and tired of changing that field? You can now set your license preference in your user settings.
- Video player now responsive; better for mobile!
- You can now delete your account from the user preferences page if you so wish.

Other changes

- Plugin writers: Internal restructuring led to mediagoblin.db.sql* be mediagoblin.db.* starting from 0.3.3
- Dependency list has been reduced not requiring the "webob" package anymore.
- And many small fixes/improvements, too numerous to list!

1.8.6 0.3.2

This will be the last release that is capable of converting from an earlier MongoDB-based MediaGoblin instance to the newer SQL-based system.

Do this to upgrade

> # directory of your mediagoblin install cd /srv/mediagoblin.example.org
>
> # copy source for this release git fetch git checkout tags/v0.3.2
>
> # perform any needed database updates bin/gmg dbupdate
>
> # restart your servers however you do that, e.g., sudo service mediagoblin-paster restart sudo service mediagoblin-celeryd restart

New features

- **3d model support!**

  You can now upload STL and OBJ files and display them in MediaGoblin. Requires a recent-ish Blender; for details see: *Deploying MediaGoblin*

- **trim_whitespace**

  We bundle the optional plugin trim_whitespace which reduces the size of the delivered html output by reducing redundant whitespace.

  See *Part 2: Core plugin documentation* for plugin documentation

- **A new API!**

  It isn't well documented yet but we do have an API. There is an android application in progress which makes use of it, and there are some demo applications between automgtic, an automatic media uploader for your desktop and OMGMG, an example of a web application hooking up to the API.

  This is a plugin, so you have to enable it in your mediagoblin config file by adding a section under [plugins] like:

  ```
  [plugins]
  [[mediagoblin.plugins.api]]
  ```

  Note that the API works but is not nailed down... the way it is called may change in future releases.

- **OAuth login support**

  For applications that use OAuth to connect to the API.

  This is a plugin, so you have to enable it in your mediagoblin config file by adding a section under [plugins] like:

  ```
  [plugins]
  [[mediagoblin.plugins.oauth]]
  ```

- **Collections**

  We now have user-curated collections support. These are arbitrary galleries that are customizable by users. You can add media to these by clicking on the paperclip icon when logged in and looking at a media entry.

- **OpenStreetMap licensing display improvements**

  More accurate display of OSM licensing, and less disruptive: you click to "expand" the display of said licensing.

  Geolocation is also now on by default.

- **Miscelaneous visual improvements**

  We've made a number of small visual improvements including newer and nicer looking thumbnails and improved checkbox placement.

1.8.7 0.3.1

Do this to upgrade

1. Make sure to run `bin/gmg dbuptdate` after upgrading.
2. If you set up your server config with an older version of mediagoblin and the mediagoblin docs, it's possible you don't have the "theme static files" alias, so double check to make sure that section is there if you are having problems.

New features

- **theming support**

  MediaGoblin now also includes theming support, which you can read about in the section *Theming MediaGoblin*.

- **flatpages**

  MediaGoblin has a flatpages plugin allowing you to add pages that are aren't media-related like "About this site...", "Terms of service...", etc.

  See *Part 2: Core plugin documentation* for plugin documentation

1.8.8 0.3.0

This release has one important change. You need to act when upgrading from a previous version!

This release changes the database system from MongoDB to SQL(alchemy). If you want to setup a fresh instance, just follow the instructions in the deployment chapter. If on the other hand you want to continue to use one instance, read on.

To convert your data from MongoDB to SQL(alchemy), you need to follow these steps:

1. Make sure your MongoDB is still running and has your data, it's needed for the conversion.
2. Configure the `sql_engine` URI in the config to represent your target database (see: *Deploying MediaGoblin*)
3. You need an empty database.
4. Then run the following command:

   ```
   bin/gmg [-cf mediagoblin_config.ini] convert_mongo_to_sql
   ```

5. Start your server and investigate.
6. That's it.

1.9 Theming MediaGoblin

We try to provide a nice theme for MediaGoblin by default, but of course, you might want something different! Maybe you want something more light and colorful, or maybe you want something specifically tailored to your organization. Have no fear—MediaGoblin has theming support! This guide should walk you through installing and making themes.

1.9.1 Installing a theme

Installing the archive

Say you have a theme archive such as `goblincities.tar.gz` and you want to install this theme! Don't worry, it's fairly painless.

1. `cd ./user_dev/themes/`
2. Move the theme archive into this directory
3. `tar -xzvf <tar-archive>`
4. Open your configuration file (probably named `mediagoblin_local.ini`) and set the theme name:

   ```
   [mediagoblin]
   # ...
   theme = goblincities
   ```

5. Link the assets so that they can be served by your web server:

   ```
   $ ./bin/gmg assetlink
   ```

Note: If you ever change the current theme in your config file, you should re-run the above command!

(In the near future this should be even easier ;))

Set up your webserver to serve theme assets

If you followed the nginx setup example in *FastCGI and nginx* you should already have theme asset setup. However, if you set up your server config with an older version of mediagoblin and the mediagoblin docs, it's possible you don't have the "theme static files" alias, so double check to make sure that section is there if you are having problems.

If you are simply using this for local development and serving the whole thing via paste/lazyserver, assuming you don't have a paste_local.ini, the asset serving should be done for you.

Configuring where things go

By default, MediaGoblin's install directory for themes is `./user_dev/themes/` (relative to the MediaGoblin checkout or base config file.) However, you can change this location easily with the `theme_install_dir` setting in the `[mediagoblin]` section.

For example:

```
[mediagoblin]
# ... other parameters go here ...
theme_install_dir = /path/to/themes/
```

Other variables you may consider setting:

theme_web_path When theme-specific assets are specified, this is where MediaGoblin will set the urls. By default this is `"/theme_static/"` so in the case that your theme was trying to access its file `"images/shiny_button.png"` MediaGoblin would link to `/theme_static/images/shiny_button.png`.

theme_linked_assets_dir Your web server needs to serve the theme files out of some directory, and MediaGoblin will symlink the current theme's assets here. See the "Link the assets" step in *Installing the archive*.

1.9.2 Making a theme

Okay, so a theme layout is pretty simple. Let's assume we're making a theme for an instance about hedgehogs! We'll call this the "hedgehogified" theme.

Change to where your `theme_install_dir` is set to (by default, `./user_dev/themes/` ... make those directories or otherwise adjust if necessary):

```
hedgehogified/
|- theme.cfg                # configuration file for this theme
|- templates/               # override templates
|  '- mediagoblin/
|     |- base.html          # overriding mediagoblin/base.html
|     '- root.html          # overriding mediagoblin/root.html
'- assets/
|  '- images/
|  |  |- im_a_hedgehog.png  # hedgehog-containing image used by theme
|  |  '- custom_logo.png    # your theme's custom logo
|  '- css/
|     '- hedgehog.css       # your site's hedgehog-specific css
|- README.txt               # Optionally, a readme file (not required)
|- AGPLv3.txt               # AGPL license file for your theme. (good practice)
'- CC0_1.0.txt              # CC0 1.0 legalcode for the assets [if appropriate!]
```

The top level directory of your theme should be the symbolic name for your theme. This is the name that users will use to refer to activate your theme.

Note: It's important to note that templates based on MediaGoblin's code should be released as AGPLv3 (or later), like MediaGoblin's code itself. However, all the rest of your assets are up to you. In this case, we are waiving our copyright for images and CSS into the public domain via CC0 (as MediaGoblin does) but do what's appropriate to you.

1.9.3 The config file

The config file is not presently strictly required, though it is nice to have. Only a few things need to go in here:

```
[theme]
name = Hedgehog-ification
description = For hedgehog lovers ONLY
licensing = AGPLv3 or later templates; assets (images/css) waived under CC0 1.0
```

The name and description fields here are to give users an idea of what your theme is about. For the moment, we don't have any listing directories or admin interface, so this probably isn't useful, but feel free to set it in anticipation of a more glorious future.

Licensing field is likewise a textual description of the stuff here; it's recommended that you preserve the "AGPLv3 or later templates" and specify whatever is appropriate to your assets.

Templates

Your template directory is where you can put any override and custom templates for MediaGoblin.

These follow the general MediaGoblin theming layout, which means that the MediaGoblin core templates are all kept under the `./mediagoblin/` prefix directory.

You can copy files right out of MediaGoblin core and modify them in this matter if you wish.

To fit with best licensing form, you should either preserve the MediaGoblin copyright header borrowing from a MediaGoblin template, or you may include one like the following:

```
{#
# [YOUR THEME], a MediaGoblin theme
# Copyright (C) [YEAR] [YOUR NAME]
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU Affero General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU Affero General Public License for more details.
#
# You should have received a copy of the GNU Affero General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#}
```

Assets

Put any files, such as images, CSS, etc, that are specific to your theme in here.

You can reference these in your templates like so:

```
<img src="{{ request.staticdirect('/images/im_a_shark.png', 'theme') }}" />
```

This will tell MediaGoblin to reference this image from the current theme.

Licensing file(s)

You should include AGPLv3.txt with your theme as this is required for the assets. You can copy this from `mediagoblin/licenses/`.

In the above example, we also use CC0 to waive our copyrights to images and css, so we also included CC0_1.0.txt

A README.txt file

A README file is not strictly required, but probably a good idea. You can put whatever in here, but restating the license choice clearly is probably a good idea.

Simple theming by adding CSS

Many themes won't require anything other than the ability to override some of MediaGoblin's core css. Thankfully, doing so is easy if you combine the above steps!

In your theme, do the following (make sure you make the necessary directories and cd to your theme's directory first):

```
$ cp /path/to/mediagoblin/mediagoblin/templates/mediagoblin/extra_head.html templates/mediagoblin/
```

Great, now open that file and add something like this at the end:

```
<link rel="stylesheet" type="text/css"
      href="{{ request.staticdirect('/css/theme.css', 'theme') }}"/>
```

You can name the css file whatever you like. Now make the directory for `assets/css/` and add the file `assets/css/theme.css`.

You can now put custom CSS files in here and any CSS you add will override default MediaGoblin CSS.

Packaging it up!

Packaging a theme is really easy. It's just a matter of making an archive!

Change to the installed themes directory and run the following:

```
tar -cvfz yourtheme.tar.gz yourtheme
```

Where "yourtheme" is replaced with your theme name.

That's it!

1.10 Plugins

GNU MediaGoblin supports plugins that allow you to augment MediaGoblin's behavior.

This chapter covers discovering, installing, configuring and removing plugins.

1.10.1 Discovering plugins

MediaGoblin comes with core plugins. Core plugins are located in the `mediagoblin.plugins` module of the MediaGoblin code. Because they come with MediaGoblin, you don't have to install them, but you do have to add them to your config file if you're interested in using them.

You can also write your own plugins and additionally find plugins elsewhere on the Internet. Once you find a plugin you like, you need to first install it, then add it to your configuration.

1.10.2 Installing plugins

Core plugins

MediaGoblin core plugins don't need to be installed because they come with MediaGoblin. Further, when you upgrade MediaGoblin, you will also get updates to the core plugins.

Other plugins

If the plugin is available on the Python Package Index, then you can install the plugin with pip:

```
pip install <plugin-name>
```

For example, if we wanted to install the plugin named "mediagoblin-licenses" (which allows you to customize the licenses you offer for your media), we would do:

```
pip install mediagoblin-licenses
```

Note: If you're using a virtual environment, make sure to activate the virtual environment before installing with pip. Otherwise the plugin may get installed in a different environment than the one MediaGoblin is installed in. Also make sure, you use e.g. pip-2.7 if your default python (and thus pip) is python 3 (e.g. in Ubuntu).

Once you've installed the plugin software, you need to tell MediaGoblin that this is a plugin you want MediaGoblin to use. To do that, you edit the `mediagoblin.ini` file and add the plugin as a subsection of the plugin section.

For example, say the "mediagoblin-licenses" plugin has the Python package path `mediagoblin_licenses`, then you would add `mediagoblin_licenses` to the `plugins` section as a subsection:

```
[plugins]

[[mediagoblin_licenses]]
license_01=abbrev1, name1, http://url1
license_02=abbrev2, name1, http://url2
```

1.10.3 Configuring plugins

Configuration for a plugin goes in the subsection for that plugin. Core plugins are documented in the administration guide. Other plugins should come with documentation that tells you how to configure them.

Example 1: Core MediaGoblin plugin

If you wanted to use the core MediaGoblin flatpages plugin, the module for that is `mediagoblin.plugins.flatpagesfile` and you would add that to your `.ini` file like this:

```
[plugins]

[[mediagoblin.plugins.flatpagesfile]]
# configuration for flatpagesfile plugin here!
about-view = '/about', about.html
terms-view = '/terms', terms.html
```

(Want to know more about the flatpagesfile plugin? See *flatpagesfile plugin*)

Example 2: Plugin that is not a core MediaGoblin plugin

If you installed a hypothetical restrictfive plugin which is in the module `restrictfive`, your `.ini` file might look like this (with comments making the bits clearer):

```
[plugins]

[[restrictfive]]
# configuration for restrictfive here!
```

Check the plugin's documentation for what configuration options are available.

1.10.4 Removing plugins

To remove a plugin, use `pip uninstall`. For example:

```
pip uninstall mediagoblin-licenses
```

Note: If you're using a virtual environment, make sure to activate the virtual environment before uninstalling with pip. Otherwise the plugin may get installed in a different environment.

1.10.5 Upgrading plugins

Core plugins

Core plugins get upgraded automatically when you upgrade MediaGoblin because they come with MediaGoblin.

Other plugins

For plugins that you install with pip, you can upgrade them with pip:

```
pip install -U <plugin-name>
```

The `-U` tells pip to upgrade the package.

1.10.6 Troubleshooting plugins

Sometimes plugins just don't work right. When you're having problems with plugins, think about the following:

1. Check the log files.

   Some plugins will log errors to the log files and you can use that to diagnose the problem.

2. Try running MediaGoblin without that plugin.

   It's easy to disable a plugin from MediaGoblin. Add a − to the name in your config file.

   For example, change:

   ```
   [[mediagoblin.plugins.flatpagesfile]]
   ```

   to:

   ```
   [[-mediagoblin.plugins.flatpagesfile]]
   ```

   That'll prevent the `mediagoblin.plugins.flatpagesfile` plugin from loading.

3. If it's a core plugin that comes with MediaGoblin, ask us for help!

   If it's a plugin you got from somewhere else, ask them for help!

CHAPTER

TWO

PART 2: CORE PLUGIN DOCUMENTATION

2.1 flatpagesfile plugin

This is the flatpages file plugin. It allows you to add pages to your MediaGoblin instance which are not generated from user content. For example, this is useful for these pages:

- About this site
- Terms of service
- Privacy policy
- How to get an account here
- ...

2.1.1 How to configure

Add the following to your MediaGoblin .ini file in the `[plugins]` section:

```
[[mediagoblin.plugins.flatpagesfile]]
```

This tells MediaGoblin to load the flatpagesfile plugin. This is the subsection that you'll do all flatpagesfile plugin configuration in.

2.1.2 How to add pages

To add a new page to your site, you need to do two things:

1. add a route to the MediaGoblin .ini file in the flatpagesfile subsection
2. write a template that will get served when that route is requested

Routes

First, let's talk about the route.

A route is a key/value in your configuration file.

The key for the route is the route name You can use this with *url()* in templates to have MediaGoblin automatically build the urls for you. It's very handy.

It should be "unique" and it should be alphanumeric characters and hyphens. I wouldn't put spaces in there.

Examples: `flatpages-about`, `about-view`, `contact-view`, ...

The value has two parts separated by commas:

1. **route path**: This is the url that this route matches.

   Examples: `/about`, `/contact`, `/pages/about`, ...

   You can do anything with this that you can do with the routepath parameter of *routes.Route*. For more details, see the routes documentation.

   Example: `/siteadmin/{adminname:\w+}`

   Note: If you're doing something fancy, enclose the route in single quotes.

   For example: `'/siteadmin/{adminname:\w+}'`

   ---

2. **template**: The template to use for this url. The template is in the flatpagesfile template directory, so you just need to specify the file name.

   Like with other templates, if it's an HTML file, it's good to use the `.html` extensions.

   Examples: `index.html`, `about.html`, `contact.html`, ...

Here's an example configuration that adds two flat pages: one for an "About this site" page and one for a "Terms of service" page:

```
[[mediagoblin.plugins.flatpagesfile]]
about-view = '/about', about.html
terms-view = '/terms', terms.html
```

Note: The order in which you define the routes in the config file is the order in which they're checked for incoming requests.

Templates

To add pages, you must edit template files on the file system in your *local_templates* directory.

The directory structure looks kind of like this:

```
local_templates
|- flatpagesfile
   |- flatpage1.html
   |- flatpage2.html
   |- ...
```

The `.html` file contains the content of your page. It's just a template like all the other templates you have.

Here's an example that extends the *flatpagesfile/base.html* template:

```
{% extends "flatpagesfile/base.html" %}
{% block mediagoblin_content %}
<h1>About this site</h1>
<p>
  This site is a MediaGoblin instance set up to host media for
```

```
  me, my family and my friends.
</p>
{% endblock %}
```

Note: If you have a bunch of flatpages that kind of look like one another, take advantage of Jinja2 template extending and create a base template that the others extend.

2.1.3 Recipes

Url variables

You can handle urls like `/about/{name}` and access the name that's passed in in the template.

Sample route:

```
about-page = '/about/{name}', about.html
```

Sample template:

```
{% extends "flatpagesfile/base.html" %}
{% block mediagoblin_content %}

<h1>About page for {{ request.matchdict['name'] }}</h1>

{% endblock %}
```

See the the routes documentation for syntax details for the route. Values will end up in the `request.matchdict` dict.

2.2 sampleplugin

This is a sample plugin. It does nothing interesting other than show one way to structure a MediaGoblin plugin.

The code for this plugin is in `mediagoblin/plugins/sampleplugin/`.

2.3 Trim whitespace plugin

Mediagoblin templates are written with 80 char limit for better readability. However that means that the html output is very verbose containing LOTS of whitespace. This plugin inserts a Middleware that filters out whitespace from the returned HTML in the Response() objects.

Simply enable this plugin by putting it somewhere where python can reach it and put it's path into the [plugins] section of your mediagoblin.ini or mediagoblin_local.ini like for example this:

    [plugins] [[mediagoblin.plugins.trim_whitespace]]

There is no further configuration required. If this plugin is enabled, all text/html documents should not have lots of whitespace in between elements, although it does a very naive filtering right now (just keep the first whitespace and delete all subsequent ones).

Nonetheless, it is a useful plugin that might serve as inspiration for other plugin writers.

It was originally conceived by Sebastian Spaeth. It is licensed under the GNU AGPL v3 (or any later version) license.

2.4 raven plugin

Warning: this plugin is somewhat experimental.

2.4.1 Set up the raven plugin

1. Add the following to your MediaGoblin .ini file in the `[plugins]` section:

```
[[mediagoblin.plugins.raven]]
sentry_dsn = <YOUR SENTRY DSN>
# Logging is very high-volume, set to 0 if you want to turn off logging
setup_logging = 1
```

2.5 basic_auth plugin

The basic_auth plugin is enabled by default in mediagoblin.ini. This plugin provides basic username and password authentication for GNU Mediagoblin.

This plugin can be enabled alongside *openid plugin* and *persona plugin*.

2.5.1 Set up the basic_auth plugin

1. Add the following to your MediaGoblin .ini file in the `[plugins]` section:

```
[[mediagoblin.plugins.basic_auth]]
```

2. Run:

```
gmg assetlink
```

   in order to link basic_auth's static assets

2.6 openid plugin

The openid plugin allows user to login to your GNU Mediagoblin instance using their openid url.

This plugin can be enabled alongside *basic_auth plugin* and *persona plugin*.

Note: When *basic_auth plugin* is enabled alongside this openid plugin, and a user creates an account using their openid. If they would like to add a password to their account, they can use the forgot password feature to do so.

2.6.1 Set up the openid plugin

1. Install the `python-openid` package.
2. Add the following to your MediaGoblin .ini file in the `[plugins]` section:

```
[[mediagoblin.plugins.openid]]
```

3. Run:

```
gmg dbupdate
```

in order to create and apply migrations to any database tables that the plugin requires.

2.7 persona plugin

The persona plugin allows users to login to you GNU MediaGoblin instance using Mozilla Persona.

This plugin can be enabled alongside *openid plugin* and *basic_auth plugin*.

Note: When *basic_auth plugin* is enabled alongside this persona plugin, and a user creates an account using their persona. If they would like to add a password to their account, they can use the forgot password feature to do so.

2.7.1 Set up the persona plugin

1. Install the `requests` package.
2. Add the following to your MediaGoblin .ini file in the `[plugins]` section:

   ```
   [[mediagoblin.plugins.persona]]
   ```

3. Run:

   ```
   gmg dbupdate
   ```

   in order to create and apply migrations to any database tables that the plugin requires.

4. Run:

   ```
   gmg assetlink
   ```

   in order to persona's static assets.

CHAPTER

THREE

PART 3: PLUGIN WRITER'S GUIDE

This guide covers writing new GNU MediaGoblin plugins.

3.1 Foreword

3.1.1 About the Plugin Writer's Guide

This guide covers writing plugins for GNU MediaGoblin. It's very much a work in progress partially because we just started writing it and partially because the plugin API is currently in flux.

3.1.2 Improving the Plugin Writer's Guide

There are a few ways—please pick whichever method is convenient for you!

1. Write up a bug report in the bug tracker
2. Tell someone on IRC `#mediagoblin` on Freenode.
3. Write an email to the devel mailing list.

Information about the bugtracker, IRC and the mailing list is all on the join page.

Patches are the most helpful, but even feedback on what you think could be improved and how to improve it is also helpful.

3.2 Quick Start

This is a quick start. It's not comprehensive, but it walks through writing a basic plugin called "sampleplugin" which logs "I've been started!" when `setup_plugin()` has been called.

3.2.1 Step 1: Files and directories

GNU MediaGoblin plugins are Python projects at heart. As such, you should use a standard Python project directory tree:

```
sampleplugin/
 |- README
 |- LICENSE
 |- setup.py
 |- sampleplugin/
    |- __init__.py
```

The outer `sampleplugin` directory holds all the project files.

The `README` should cover what your plugin does, how to install it, how to configure it, and all the sorts of things a README should cover.

The `LICENSE` should have the license under which you're distributing your plugin.

The inner `sampleplugin` directory is the Python package that holds your plugin's code.

The `__init__.py` denotes that this is a Python package. It also holds the plugin code and the `hooks` dict that specifies which hooks the sampleplugin uses.

3.2.2 Step 2: README

Here's a rough `README`. Generally, you want more information because this is the file that most people open when they want to learn more about your project.

```
README
======

This is a sample plugin. It logs a line when ``setup_plugin()`` is
run.
```

3.2.3 Step 3: LICENSE

GNU MediaGoblin plugins must be licensed under the AGPLv3 or later. So the LICENSE file should be the AGPLv3 text which you can find at http://www.gnu.org/licenses/agpl-3.0.html

3.2.4 Step 4: setup.py

This file is used for packaging and distributing your plugin.

We'll use a basic one:

```
from setuptools import setup, find_packages

setup(
    name='sampleplugin',
    version='1.0',
    packages=find_packages(),
    include_package_data=True,
    install_requires=[],
    license='AGPLv3',
    )
```

See http://docs.python.org/distutils/index.html#distutils-index for more details.

3.2.5 Step 5: the code

The code for `__init__.py` looks like this:

```
1  import logging
2  from mediagoblin.tools.pluginapi import Plugin, get_config
3
4
5  # This creates a logger that you can use to log information to
6  # the console or a log file.
7  _log = logging.getLogger(__name__)
8
9
10 # This is the function that gets called when the setup
11 # hook fires.
12 def setup_plugin():
13     _log.info("I've been started!")
14     config = get_config('sampleplugin')
15     if config:
16         _log.info('%r' % config)
17     else:
18         _log.info('There is no configuration set.')
19
20
21 # This is a dict that specifies which hooks this plugin uses.
22 # This one only uses one hook: setup.
23 hooks = {
24     'setup': setup_plugin
25     }
```

Line 12 defines the `setup_plugin` function.

Line 23 defines `hooks`. When MediaGoblin loads this file, it sees `hooks` and registers all the callables with their respective hooks.

3.2.6 Step 6: Installation and configuration

To install the plugin for development, you need to make sure it's available to the Python interpreter that's running MediaGoblin.

There are a couple of ways to do this, but we're going to pick the easy one.

Use `python` from your MediaGoblin virtual environment and do:

```
python setup.py develop
```

Any changes you make to your plugin will be available in your MediaGoblin virtual environment.

Then adjust your `mediagoblin.ini` file to load the plugin:

```
[plugins]

[[sampleplugin]]
```

3.2.7 Step 7: That's it!

When you launch MediaGoblin, it'll load the plugin and you'll see evidence of that in the log file.

That's it for the quick start!

3.2.8 Where to go from here

See the documentation on the *Plugin API* for code samples and other things you can use when building your plugin. If your plugin needs its own database models, see *Database models for plugins*.

See Hitchhiker's Guide to Packaging for more information on packaging your plugin.

3.3 Database models for plugins

3.3.1 Accessing Existing Data

If your plugin wants to access existing data, this is quite straight forward. Just import the appropiate models and use the full power of SQLAlchemy. Take a look at the (upcoming) database section in the Developer's Chapter.

3.3.2 Creating new Tables

If your plugin needs some new space to store data, you should create a new table. Please do not modify core tables. Not doing so might seem inefficient and possibly is. It will help keep things sane and easier to upgrade versions later.

So if you create a new plugin and need new tables, create a file named `models.py` in your plugin directory. You might take a look at the core's db.models for some ideas. Here's a simple one:

```
from mediagoblin.db.base import Base
from sqlalchemy import Column, Integer, Unicode, ForeignKey

class MediaSecurity(Base):
    __tablename__ = "yourplugin__media_security"

    # The primary key *and* reference to the main media_entry
    media_entry = Column(Integer, ForeignKey('core__media_entries.id'),
        primary_key=True)
    get_media_entry = relationship("MediaEntry",
        backref=backref("security_rating", cascade="all, delete-orphan"))

    rating = Column(Unicode)

MODELS = [MediaSecurity]
```

That's it.

Some notes:

- Make sure all your `__tablename__` start with your plugin's name so the tables of various plugins can't conflict in the database. (Conflicts in python naming are much easier to fix later).
- Try to get your database design as good as possible in the first attempt. Changing the database design later, when people already have data using the old design, is possible (see next chapter), but it's not easy.

3.3.3 Changing the Database Schema Later

If your plugin is in use and instances use it to store some data, changing the database design is a tricky thing.

1. Make up your mind how the new schema should look like.
2. Change `models.py` to contain the new schema. Keep a copy of the old version around for your personal reference later.
3. Now make up your mind (possibly using your old and new `models.py`) what steps in SQL are needed to convert the old schema to the new one. This is called a "migration".
4. Create a file `migrations.py` that will contain all your migrations and add your new migration.

Take a look at the core's `db/migrations.py` for some good examples on what you might be able to do. Here's a simple one to add one column:

```
from mediagoblin.db.migration_tools import RegisterMigration, inspect_table
from sqlalchemy import MetaData, Column, Integer

MIGRATIONS = {}

@RegisterMigration(1, MIGRATIONS)
def add_license_preference(db):
    metadata = MetaData(bind=db.bind)

    security_table = inspect_table(metadata, 'yourplugin__media_security')

    col = Column('security_level', Integer)
    col.create(security_table)
    db.commit()
```

3.4 Plugin API

This documents the general plugin API.

Please note, at this point OUR PLUGIN HOOKS MAY AND WILL CHANGE. Authors are encouraged to develop plugins and work with the MediaGoblin community to keep them up to date, but this API will be a moving target for a few releases.

Please check the *Release Notes* for updates!

3.4.1 How are hooks added? Where do I find them?

Much of this document talks about hooks, both as in terms of regular hooks and template hooks. But where do they come from, and how can you find a list of them?

For the moment, the best way to find available hooks is to check the source code itself. (Yes, we should start a more official hook listing with descriptions soon.) But many hooks you may need do not exist yet: what to do then?

The plan at present is that we are adding hooks as people need them, with community discussion. If you find that you need a hook and MediaGoblin at present doesn't provide it at present, please http://mediagoblin.org/pages/join.html! We'll evaluate what to do from there.

3.4.2 `pluginapi` Module

This module implements the plugin api bits.

Two things about things in this module:

1. they should be excessively well documented because we should pull from this file for the docs

2. they should be well tested

How do plugins work?

Plugins are structured like any Python project. You create a Python package. In that package, you define a high-level `__init__.py` module that has a `hooks` dict that maps hooks to callables that implement those hooks.

Additionally, you want a LICENSE file that specifies the license and a `setup.py` that specifies the metadata for packaging your plugin. A rough file structure could look like this:

```
myplugin/
 |- setup.py          # plugin project packaging metadata
 |- README            # holds plugin project information
 |- LICENSE           # holds license information
 |- myplugin/         # plugin package directory
    |- __init__.py    # has hooks dict and code
```

Lifecycle

1. All the modules listed as subsections of the `plugins` section in the config file are imported. MediaGoblin registers any hooks in the `hooks` dict of those modules.
2. After all plugin modules are imported, the `setup` hook is called allowing plugins to do any set up they need to do.

mediagoblin.tools.pluginapi.**get_config**(*key*)
Retrieves the configuration for a specified plugin by key

Example:

```
>>> get_config('mediagoblin.plugins.sampleplugin')
{'foo': 'bar'}
>>> get_config('myplugin')
{}
>>> get_config('flatpages')
{'directory': '/srv/mediagoblin/pages', 'nesting': 1}}
```

mediagoblin.tools.pluginapi.**register_routes**(*routes*)
Registers one or more routes

If your plugin handles requests, then you need to call this with the routes your plugin handles.

A "route" is a *routes.Route* object. See the routes.Route documentation for more details.

Example passing in a single route:

```
>>> register_routes(('about-view', '/about',
...     'mediagoblin.views:about_view_handler'))
```

Example passing in a list of routes:

```
>>> register_routes([
...     ('contact-view', '/contact', 'mediagoblin.views:contact_handler'),
...     ('about-view', '/about', 'mediagoblin.views:about_handler')
... ])
```

Note: Be careful when designing your route urls. If they clash with core urls, then it could result in DISASTER!

---

mediagoblin.tools.pluginapi.**register_template_path**(*path*)
Registers a path for template loading

If your plugin has templates, then you need to call this with the absolute path of the root of templates directory.

Example:

```
>>> my_plugin_dir = os.path.dirname(__file__)
>>> template_dir = os.path.join(my_plugin_dir, 'templates')
>>> register_template_path(template_dir)
```

Note: You can only do this in *setup_plugins()*. Doing this after that will have no effect on template loading.

---

mediagoblin.tools.pluginapi.**register_template_hooks**(*template_hooks*)
Register a dict of template hooks.

Takes template_hooks as an argument, which is a dictionary of template hook names/keys to the templates they should provide. (The value can either be a single template path or an iterable of paths.)

Example:

```
{"media_sidebar": "/plugin/sidemess/mess_up_the_side.html",
 "media_descriptionbox": ["/plugin/sidemess/even_more_mess.html",
                          "/plugin/sidemess/so_much_mess.html"]}
```

mediagoblin.tools.pluginapi.**get_hook_templates**(*hook_name*)
Get a list of hook templates for this hook_name.

Note: for the most part, you access this via a template tag, not this method directly, like so:

```
{% template_hook("media_sidebar") %}
```

... which will include all templates for you, partly using this method.

However, this method is exposed to templates, and if you wish, you can iterate over templates in a template hook manually like so:

```
{% for template_path in get_hook_templates("media_sidebar") %}
  <div class="extra_structure">
    {% include template_path %}
  </div>
{% endfor %}
```

Returns: A list of strings representing template paths.

mediagoblin.tools.pluginapi.**hook_handle**(*hook_name*, **args*, ***kwargs*)
Run through hooks attempting to find one that handle this hook.

All callables called with the same arguments until one handles things and returns a non-None value.

(If you are writing a handler and you don't have a particularly useful value to return even though you've handled this, returning True is a good solution.)

Note that there is a special keyword argument: if "default_handler" is passed in as a keyword argument, this will be used if no handler is found.

Some examples of using this:

- You need an interface implemented, but only one fit for it
- You need to *do* something, but only one thing needs to do it.

mediagoblin.tools.pluginapi.**hook_runall**(*hook_name*, **args*, ***kwargs*)
Run through all callable hooks and pass in arguments.

All non-None results are accrued in a list and returned from this. (Other "false-like" values like False and friends are still accrued, however.)

Some examples of using this:

- You have an interface call where actually multiple things can and should implement it
- You need to get a list of things from various plugins that handle them and do something with them
- You need to *do* something, and actually multiple plugins need to do it separately

mediagoblin.tools.pluginapi.**hook_transform**(*hook_name*, *arg*)
Run through a bunch of hook callables and transform some input.

Note that unlike the other hook tools, this one only takes ONE argument. This argument is passed to each function, which in turn returns something that becomes the input of the next callable.

Some examples of using this:

- You have an object, say a form, but you want plugins to each be able to modify it.

3.4.3 Configuration

Your plugin may define its own configuration defaults.

Simply add to the directory of your plugin a config_spec.ini file. An example might look like:

```
[plugin_spec]
some_string = string(default="blork")
some_int = integer(default=50)
```

This means that when people enable your plugin in their config you'll be able to provide defaults as well as type validation.

You can access this via the app_config variables in mg_globals, or you can use a shortcut to get your plugin's config section:

```
>>> from mediagoblin.tools import pluginapi
# Replace with the path to your plugin.
# (If an external package,  it won't be part of mediagoblin.plugins)
>>> floobie_config = pluginapi.get_config('mediagoblin.plugins.floobifier')
>>> floobie_dir = floobie_config['floobie_dir']
# This is the same as the above
>>> from mediagoblin import mg_globals
>>> config = mg_globals.global_config['plugins']['mediagoblin.plugins.floobifier']
>>> floobie_dir = floobie_config['floobie_dir']
```

A tip: you have access to the *%(here)s* variable in your config, which is the directory that the user's mediagoblin config is running out of. So for example, your plugin may need a "floobie" directory to store floobs in. You could give them a reasonable default that makes use of the default *user_dev* location, but allow users to override it, like so:

```
[plugin_spec]
floobie_dir = string(default="%(here)s/user_dev/floobs/")
```

Note, this is relative to the user's mediagoblin config directory, *not* your plugin directory!

3.4.4 Context Hooks

View specific hooks

You can hook up to almost any template called by any specific view fairly easily. As long as the view directly or indirectly uses the method `render_to_response` you can access the context via a hook that has a key in the format of the tuple:

```
(view_symbolic_name, view_template_path)
```

Where the "view symbolic name" is the same parameter used in `request.urlgen()` to look up the view. So say we're wanting to add something to the context of the user's homepage. We look in mediagoblin/user_pages/routing.py and see:

```
add_route('mediagoblin.user_pages.user_home',
          '/u/<string:user>/',
          'mediagoblin.user_pages.views:user_home')
```

Aha! That means that the name is `mediagoblin.user_pages.user_home`. Okay, so then we look at the view at the `mediagoblin.user_pages.user_home` method:

```
@uses_pagination
def user_home(request, page):
    # [...] whole bunch of stuff here
    return render_to_response(
        request,
        'mediagoblin/user_pages/user.html',
        {'user': user,
         'user_gallery_url': user_gallery_url,
         'media_entries': media_entries,
         'pagination': pagination})
```

Nice! So the template appears to be `mediagoblin/user_pages/user.html`. Cool, that means that the key is:

```
("mediagoblin.user_pages.user_home",
 "mediagoblin/user_pages/user.html")
```

The context hook uses `hook_transform()` so that means that if we're hooking into it, our hook will both accept one argument, `context`, and should return that modified object, like so:

```
def add_to_user_home_context(context):
    context['foo'] = 'bar'
    return context

hooks = {
    ("mediagoblin.user_pages.user_home",
     "mediagoblin/user_pages/user.html"): add_to_user_home_context}
```

Global context hooks

If you need to add something to the context of *every* view, it is not hard; there are two hooks hook that also uses hook_transform (like the above) but make available what you are providing to *every* view.

Note that there is a slight, but critical, difference between the two.

The most general one is the `'template_global_context'` hook. This one is run only once, and is read into the global context... all views will get access to what are in this dict.

The slightly more expensive but more powerful one is `'template_context_prerender'`. This one is not added to the global context... it is added to the actual context of each individual template render right before it is run! Because of this you also can do some powerful and crazy things, such as checking the request object or other parts of the context before passing them on.

3.4.5 Adding static resources

It's possible to add static resources for your plugin. Say your plugin needs some special javascript and images... how to provide them? Then how to access them? MediaGoblin has a way!

Attaching to the hook

First, you need to register your plugin's resources with the hook. This is pretty easy actually: you just need to provide a function that passes back a PluginStatic object.

class `mediagoblin.tools.staticdirect.`**`PluginStatic`**(*name*, *file_path*)

Pass this into the `'static_setup'` hook to register your plugin's static directory.

This has two mandatory attributes that you must pass in on class init:

- *name:* this name will be both used for lookup in "urlgen" for your plugin's static resources and for the subdirectory that it'll be "mounted" to for serving via your web browser. It *MUST* be unique. If writing a plugin bundled with MediaGoblin please use the pattern 'coreplugin__foo' where 'foo' is your plugin name. All external plugins should use their modulename, so if your plugin is 'mg_bettertags' you should also call this name 'mg_bettertags'.
- *file_path:* the directory your plugin's static resources are located in. It's recommended that you use pkg_resources.resource_filename() for this.

An example of using this:

```
from pkg_resources import resource_filename
from mediagoblin.tools.staticdirect import PluginStatic

hooks = {
    'static_setup': lambda: PluginStatic(
        'mg_bettertags',
        resource_filename('mg_bettertags', 'static'))
}
```

Running plugin assetlink

In order for your plugin assets to be properly served by MediaGoblin, your plugin's asset directory needs to be symlinked into the directory that plugin assets are served from. To set this up, run:

```
./bin/gmg assetlink
```

Using staticdirect

Once you have this, you will want to be able to of course link to your assets! MediaGoblin has a "staticdirect" tool; you want to use this like so in your templates:

```
staticdirect("css/monkeys.css", "mystaticname")
```

Replace "mystaticname" with the name you passed to PluginStatic. The staticdirect method is, for convenience, attached to the request object, so you can access this in your templates like:

```
<img alt="A funny bunny"
     src="{{ request.staticdirect('images/funnybunny.png', 'mystaticname') }}" />
```

3.4.6 Additional hook tips

This section aims to explain some tips in regards to adding hooks to the MediaGoblin repository.

WTForms hooks

We haven't totally settled on a way to tranform wtforms form objects, but here's one way. In your view:

```
from mediagoblin.foo.forms import SomeForm

def some_view(request)
    form_class = hook_transform('some_form_transform', SomeForm)
    form = form_class(request.form)
```

Then to hook into this form, do something in your plugin like:

```
import wtforms

class SomeFormAdditions(wtforms.Form):
    new_datefield = wtforms.DateField()

def transform_some_form(orig_form):
    class ModifiedForm(orig_form, SomeFormAdditions)
    return ModifiedForm

hooks = {
    'some_form_transform': transform_some_form}
```

Interfaces

If you want to add a pseudo-interface, it's not difficult to do so. Just write the interface like so:

```
class FrobInterface(object):
    """
    Interface for Frobbing.

    Classes implementing this interface should provide defrob and frob.
    They may also implement double_frob, but it is not required; if
    not provided, we will use a general technique.
    """

    def defrob(self, frobbed_obj):
        """
        Take a frobbed_obj and defrob it.  Returns the defrobbed object.
        """
        raise NotImplementedError()

    def frob(self, normal_obj):
        """
```

```
        Take a normal object and frob it.  Returns the frobbed object.
        """
        raise NotImplementedError()

    def double_frob(self, normal_obj):
        """
        Frob this object and return it multiplied by two.
        """
        return self.frob(normal_obj) * 2


def some_frob_using_method():
    # something something something
    frobber = hook_handle(FrobInterface)
    frobber.frob(blah)

    # alternately you could have a default
    frobber = hook_handle(FrobInterface) or DefaultFrobber
    frobber.defrob(foo)
```

It's fine to use your interface as the key instead of a string if you like. (Usually this is messy, but since interfaces are public and since you need to import them into your plugin anyway, interfaces might as well be keys.)

Then a plugin providing your interface can be like:

```
from mediagoblin.foo.frobfrogs import FrobInterface
from frogfrobber import utils

class FrogFrobber(FrobInterface):
    """
    Takes a frogputer science approach to frobbing.
    """
    def defrob(self, frobbed_obj):
        return utils.frog_defrob(frobbed_obj)

    def frob(self, normal_obj):
        return utils.frog_frob(normal_obj)

 hooks = {
     FrobInterface: lambda: return FrogFrobber}
```

3.5 Writing unit tests for plugins

Here's a brief guide to writing unit tests for plugins. However, it isn't really ideal. It also hasn't been well tested... yes, there's some irony there :)

Some notes: we're using py.test and webtest for unit testing stuff. Keep that in mind.

My suggestion is to mime the behavior of *mediagoblin/tests/* and put that in your own plugin, like *myplugin/tests/*. Copy over *conftest.py* and *pytest.ini* to your tests directory, but possibly change the *test_app* fixture to match your own tests' config needs. For example:

```
import pkg_resources
# [...]

@pytest.fixture()
def test_app(request):
```

```
    return get_app(
        request,
        mgoblin_config=pkg_resources.resource_filename(
            'myplugin.tests', 'myplugin_mediagoblin.ini'))
```

In any test module in your tests directory you can then do:

```
def test_somethingorother(test_app):
    # real code goes here
    pass
```

And you'll get a mediagoblin application wrapped in webtest passed in to your environment.

If your plugin needs to define multiple configuration setups, you can actually set up multiple fixtures very easily for this. You can just set up multiple fixtures with different names that point to different configs and pass them in as that named argument.

To run the tests, from mediagoblin's directory (make sure that your plugin has been added to your mediagoblin checkout's virtualenv!) do:

```
./runtests.sh /path/to/myplugin/tests/
```

replacing */path/to/myplugin/* with the actual path to your plugin.

NOTE: again, the above is untested, but it should probably work. If you run into trouble, contact us, preferably on IRC!

3.6 Media Type hooks

This documents the hooks that are currently available for `media_type` plugins.

3.6.1 What hooks are available?

'sniff_handler'

This hook is used by `sniff_media` in `mediagoblin.media_types.__init__`. Your media type should return its `sniff_media` method when this hook is called.

Note: Your `sniff_media` method should return either the `media_type` or `None`.

'get_media_type_and_manager'

This hook is used by `get_media_type_and_manager` in `mediagoblin.media_types.__init__`. When this hook is called, your media type plugin should check if it can handle the given extension. If so, your media type plugin should return the media type and media manager.

('media_manager', MEDIA_TYPE)

If you already know the string representing the media type of a type of media, you can pull down the manager specifically. Note that this hook is not a string but a tuple of two strings, the latter being the name of the media type.

This is used by media entries to pull down their media managers, and so on.

3.7 Authentication Hooks

This documents the hooks that are currently available for authentication plugins. If you need new hooks for your plugin, go ahead a submit a patch.

3.7.1 What hooks are available?

'authentication'

This hook just needs to return `True` as this is how the MediaGoblin app knows that an authentication plugin is enabled.

'auth_extra_validation'

This hook is used to provide any additional validation of the registration form when using `mediagoblin.auth.tools.register_user()`. This hook runs through all enabled auth plugins.

'auth_create_user'

This hook is used by `mediagoblin.auth.tools.register_user()` so plugins can store the necessary information when creating a user. This hook runs through all enabled auth plugins.

'auth_get_user'

This hook is used by `mediagoblin.auth.tools.check_login_simple()`. Your plugin should return a `User` object given a username.

'auth_no_pass_redirect'

This hook is called in `mediagoblin.auth.views` in both the `login` and `register` views. This hook should return the name of your plugin, so that if *basic_auth plugin* is not enabled, the user will be redirected to the correct login and registration views for your plugin.

The code assumes that it can generate a valid url given `mediagoblin.plugins.{{ your_plugin_here }}.login` and `mediagoblin.plugins.{{ your_plugin_here }}.register`. This is only needed if you will not be using the `login` and `register` views in `mediagoblin.auth.views`.

'auth_get_login_form'

This hook is called in `mediagoblin.auth.views.login()`. If you are not using that view, then you do not need this hook. This hook should take a `request` object and return the `LoginForm` for your plugin.

'auth_get_registration_form'

This hook is called in `mediagoblin.auth.views.register()`. If you are not using that view, then you do not need this hook. This hook should take a `request` object and return the `RegisterForm` for your plugin.

'auth_gen_password_hash'

This hook should accept a `raw_pass` and an `extra_salt` and return a hashed password to be stored in `User.pw_hash`.

'auth_check_password'

This hook should accept a `raw_pass`, a `stored_hash`, and an `extra_salt`. Your plugin should then check that the `raw_pass` hashes to the same thing as the `stored_hash` and return either `True` or `False`.

'auth_fake_login_attempt'

This hook is called in `mediagoblin.auth.tools.check_login_simple`. It is called if a user is not found and should do something that takes the same amount of time as your `check_password` function. This is to help prevent timining attacks.

CHAPTER

FOUR

PART 4: DEVELOPER'S ZONE

This chapter contains various information for developers.

4.1 Codebase Documentation

Sections

- What's where
- Software Stack

This chapter covers the libraries that GNU MediaGoblin uses as well as various recipes for getting things done.

Note: This chapter is in flux. Clearly there are things here that aren't documented. If there's something you have questions about, please ask!

See the join page on the website for where we hang out.

For more information on how to get started hacking on GNU MediaGoblin, see the wiki, and specifically, go through the Hacking HOWTO which explains generally how to get going with running an instance for development.

4.1.1 What's where

After you've run checked out mediagoblin and followed the virtualenv instantiation instructions, you're faced with the following directory tree:

```
mediagoblin/
|- mediagoblin/              # source code
|  |- db/                    # database setup
|  |- tools/                 # various utilities
|  |- init/                  # "initialization" tools (arguably should be in tools/)
|  |- tests/                 # unit tests
|  |- templates/             # templates for this application
|  |- media_types/           # code for processing, displaying different media
|  |- storage/               # different storage backends
|  |- gmg_commands/          # command line tools (./bin/gmg)
|  |- themes/                # pre-bundled themes
|  |
|  |  # ... some submodules here as well for different sections
|  |  # of the application... here's just a few
```

```
|  |- auth/                  # authentication (login/registration) code
|  |- user_dev/              # user pages (under /u/), including media pages
|  \- submit/                # submitting media for processing
|
|- docs/                     # documentation
|- devtools/                 # some scripts for developer convenience
|
|- user_dev/                 # local instance sessions, media, etc
|
|  # the below directories are installed into your virtualenv checkout
|
|- bin/                      # scripts
|- develop-eggs/
|- lib/                      # python libraries installed into your virtualenv
|- include/
|- mediagoblin.egg-info/
\- parts/
```

As you can see, all the code for GNU MediaGoblin is in the `mediagoblin` directory.

Here are some interesting files and what they do:

routing.py maps url paths to views

views.py views handle http requests

forms.py wtforms stuff for this submodule

You'll notice that there are several sub-directories: tests, templates, auth, submit, ...

`tests` holds the unit test code.

`templates` holds all the templates for the output.

`auth` and `submit` are modules that enacpsulate authentication and media item submission. If you look in these directories, you'll see they have their own `routing.py`, `view.py`, and forms.py in addition to some other code.

You'll also notice that mediagoblin/db/ contains quite a few things, including the following:

models.py This is where the database is set up

mixin.py Certain functions appended to models from here

migrations.py When creating a new migration (a change to the database structure), we put it here

4.1.2 Software Stack

- Project infrastructure
  - Python: the language we're using to write this
  - Py.Test: for unit tests
  - virtualenv: for setting up an isolated environment to keep mediagoblin and related packages (potentially not required if MediaGoblin is packaged for your distro)
- Data storage
  - SQLAlchemy: SQL ORM and database interaction library for Python. Currently we support sqlite and postgress as backends.
- Web application
  - Paste Deploy and Paste Script: we'll use this for configuring and launching the application

  - werkzeug: nice abstraction layer from HTTP requests, responses and WSGI bits
  - itsdangerous: for handling sessions
  - Jinja2: the templating engine
  - WTForms: for handling, validation, and abstraction from HTML forms
  - Celery: for task queuing (resizing images, encoding video, ...)
  - Babel: Used to extract and compile translations.
  - Markdown (for python): implementation of Markdown text-to-html tool to make it easy for people to write richtext comments, descriptions, and etc.
  - lxml: nice xml and html processing for python.
- Media processing libraries
  - Python Imaging Library: used to resize and otherwise convert images for display.
  - GStreamer: (Optional, for video hosting sites only) Used to transcode video, and in the future, probably audio too.
  - chardet: (Optional, for ascii art hosting sites only) Used to make ascii art thumbnails.
- Front end
  - JQuery: for groovy JavaScript things

4.2 Storage

4.2.1 The storage systems attached to your app

Dynamic content: queue_store and public_store

Two instances of the StorageInterface come attached to your app. These are:

- **queue_store:** When a user submits a fresh piece of media for their gallery, before the Processing stage, that piece of media sits here in the queue_store. (It's possible that we'll rename this to "private_store" and start storing more non-publicly-stored stuff in the future...). This is a StorageInterface implementation instance. Visitors to your site probably cannot see it... it isn't designed to be seen, anyway.
- **public_store:** After your media goes through processing it gets moved to the public store. This is also a StorageInterface implelementation, and is for stuff that's intended to be seen by site visitors.

The workbench

In addition, there's a "workbench" used during processing... it's just for temporary files during processing, and also for making local copies of stuff that might be on remote storage interfaces while transitionally moving/converting from the queue_store to the public store. See the workbench module documentation for more.

class mediagoblin.tools.workbench.**Workbench**(*dir*)
Bases: object

Represent the directory for the workbench

WARNING: DO NOT create Workbench objects on your own, let the WorkbenchManager do that for you!

destroy()
Destroy this workbench! Deletes the directory and all its contents!

WARNING: Does no checks for a sane value in self.dir!

localized_file(*storage*, *filepath*, *filename_if_copying=None*, *keep_extension_if_copying=True*)
Possibly localize the file from this storage system (for read-only purposes, modifications should be written to a new file.).

If the file is already local, just return the absolute filename of that local file. Otherwise, copy the file locally to the workbench, and return the absolute path of the new file.

If it is copying locally, we might want to require a filename like "source.jpg" to ensure that we won't conflict with other filenames in our workbench... if that's the case, make sure filename_if_copying is set to something like 'source.jpg'. Relatedly, if you set keep_extension_if_copying, you don't have to set an extension on filename_if_copying yourself, it'll be set for you (assuming such an extension can be extacted from the filename in the filepath).

Returns: localized_filename

Examples:

```
>>> wb_manager.localized_file(
...     '/our/workbench/subdir', local_storage,
...     ['path', 'to', 'foobar.jpg'])
u'/local/storage/path/to/foobar.jpg'


>>> wb_manager.localized_file(
...     '/our/workbench/subdir', remote_storage,
...     ['path', 'to', 'foobar.jpg'])
'/our/workbench/subdir/foobar.jpg'


>>> wb_manager.localized_file(
...     '/our/workbench/subdir', remote_storage,
...     ['path', 'to', 'foobar.jpg'], 'source.jpeg', False)
'/our/workbench/subdir/foobar.jpeg'


>>> wb_manager.localized_file(
...     '/our/workbench/subdir', remote_storage,
...     ['path', 'to', 'foobar.jpg'], 'source', True)
'/our/workbench/subdir/foobar.jpg'
```

class mediagoblin.tools.workbench.**WorkbenchManager**(*base_workbench_dir*)
Bases: object

A system for generating and destroying workbenches.

Workbenches are actually just subdirectories of a (local) temporary storage space for during the processing stage. The preferred way to create them is to use:

with workbenchmger.create() as workbench: do stuff...

This will automatically clean up all temporary directories even in case of an exceptions. Also check the @mediagoblin.decorators.get_workbench decorator for a convenient wrapper.

create()
Create and return the path to a new workbench (directory).

Static assets / staticdirect

On top of all that, there is some static media that comes bundled with your application. This stuff is kept in:

mediagoblin/static/

These files are for mediagoblin base assets. Things like the CSS files, logos, etc. You can mount these at whatever location is appropriate to you (see the direct_remote_path option in the config file) so if your users are keeping their static assets at http://static.mgoblin.example.org/ but their actual site is at http://mgoblin.example.org/, you need to be able to get your static files in a where-it's-mounted agnostic way. There's a "staticdirector" attached to the request object. It's pretty easy to use; just look at this bit taken from the mediagoblin/templates/mediagoblin/base.html main template:

<link rel="stylesheet" type="text/css" href="Template:Request.staticdirect('/css/extlib/text.css')"/>

see? Not too hard. As expected, if you configured direct_remote_path to be http://static.mgoblin.example.org/ you'll get back http://static.mgoblin.example.org/css/extlib/text.css just as you'd probably expect.

4.2.2 StorageInterface and implementations

The guts of StorageInterface and friends

So, the StorageInterface!

So, the public and queue stores both use StorageInterface implementations ... but what does that mean? It's not too hard.

Open up:

mediagoblin/storage.py

In here you'll see a couple of things. First of all, there's the StorageInterface class. What you'll see is that this is just a very simple python class. A few of the methods actually implement things, but for the most part, they don't. What really matters about this class is the docstrings. Each expected method is documented as to how it should be constructed. Want to make a new StorageInterface? Simply subclass it. Want to know how to use the methods of your storage system? Read these docs, they span all implementations.

There are a couple of implementations of these classes bundled in storage.py as well. The most simple of these is BasicFileStorage, which is also the default storage system used. As expected, this stores files locally on your machine.

There's also a CloudFileStorage system. This provides a mapping to [OpenStack's swift http://swift.openstack.org/] storage system (used by RackSpace Cloud files and etc).

Between these two examples you should be able to get a pretty good idea of how to write your own storage systems, for storing data across your beowulf cluster of radioactive monkey brains, whatever.

Writing code to store stuff

So what does coding for StorageInterface implementations actually look like? It's pretty simple, really. For one thing, the design is fairly inspired by [Django's file storage API https://docs.djangoproject.com/en/dev/ref/files/storage/]... with some differences.

Basically, you access files on "file paths", which aren't exactly like unix file paths, but are close. If you wanted to store a file on a path like dir1/dir2/filename.jpg you'd actually write that file path like:

['dir1', 'dir2', 'filename.jpg']

This way we can be *sure* that each component is actually a component of the path that's expected... we do some filename cleaning on each component.

Your StorageInterface should pass in and out "file like objects". In other words, they should provide .read() and .write() at minimum, and probably also .seek() and .close().

4.3 Original Design Decisions

Sections

This chapter talks a bit about design decisions.

Note: This is an outdated document. It's more or less the historical reasons for a lot of things. That doesn't mean these decisions have stayed the same or we haven't changed our minds on some things!

4.3.1 Why GNU MediaGoblin?

Chris and Will on "Why GNU MediaGoblin":

> Chris came up with the name MediaGoblin. The name is pretty fun. It merges the idea that this is a Media hosting project with Goblin which sort of sounds like gobbling. Here's a piece of software that gobbles up your media for all to see.
>
> According to Wikipedia, a goblin is:
>
> > a legendary evil or mischievous illiterate creature, described as grotesquely evil or evil-like phantom
>
> So are we evil? No. Are we mischievous or illiterate? Not really. So what kind of goblin are we thinking about? We're thinking about these goblins:
>
> Those are pretty cute goblins. Those are the kinds of goblins we're thinking about.
>
> Chris started doing work on the project after thinking about it for a year. Then, after talking with Matt and Rob, it became an official GNU project. Thus we now call it GNU MediaGoblin.
>
> That's a lot of letters, though, so in the interest of brevity and facilitating easier casual conversation and balancing that with what's important to us, we have the following rules:
>
> 1. "GNU MediaGoblin" is the name we're going to use in all official capacities: web site, documentation, press releases, ...
> 2. In casual conversation, it's ok to use more casual names.
> 3. If you're writing about the project, we ask that you call it GNU MediaGoblin.
> 4. If you don't like the name, we kindly ask you to take a deep breath, think a happy thought about cute little goblins playing on a playground and taking cute pictures of themselves, and let it go. (Will added this one.)

Figure 4.1: *Figure 1: Cute goblin with a beret. llustrated by Chris Webber*

Figure 4.2: *Figure 2: Snuggly goblin. Illustrated by Karen Rustad*

4.3.2 Why Python

Chris Webber on "Why Python":

> Because I know Python, love Python, am capable of actually making this thing happen in Python (I've worked on a lot of large free software web applications before in Python, including Miro Community, the Miro Guide, a large portion of Creative Commons, and a whole bunch of things while working at Imaginary Landscape). Me starting a project like this makes sense if it's done in Python.
>
> You might say that PHP is way more deployable, that Rails has way more cool developers riding around on fixie bikes—and all of those things are true. But I know Python, like Python, and think that Python is pretty great. I do think that deployment in Python is not as good as with PHP, but I think the days of shared hosting are (thankfully) coming to an end, and will probably be replaced by cheap virtual machines spun up on the fly for people who want that sort of stuff, and Python will be a huge part of that future, maybe even more than PHP will. The deployment tools are getting better. Maybe we can use something like Silver Lining. Maybe we can just distribute as `.debs` or `.rpms`. We'll figure it out when we get there.
>
> Regardless, if I'm starting this project, which I am, it's gonna be in Python.

4.3.3 Why WSGI Minimalism

Chris Webber on "Why WSGI Minimalism":

> If you notice in the technology list I list a lot of components that are very "django-like", but not actually Django components. What can I say, I really like a lot of the ideas in Django! Which leads to the question: why not just use Django?
>
> While I really like Django's ideas and a lot of its components, I also feel that most of the best ideas in Django I want have been implemented as good or even better outside of Django. I could just use Django and replace the templating system with Jinja2, and the form system with wtforms, and the database with MongoDB and MongoKit, but at that point, how much of Django is really left?
>
> I also am sometimes saddened and irritated by how coupled all of Django's components are. Loosely coupled yes, but still coupled. WSGI has done a good job of providing a base layer for running applications on and if you know how to do it yourself [1], it's not hard or many lines of code at all to bind them together without any framework at all (not even say Pylons, Pyramid or Flask which I think are still great projects, especially for people who want this sort of thing but have no idea how to get started). And even at this already really early stage of writing MediaGoblin, that glue work is mostly done.
>
> Not to say I don't think Django isn't great for a lot of things. For a lot of stuff, it's still the best, but not for MediaGoblin, I think.
>
> One thing that Django does super well though is documentation. It still has some faults, but even with those considered I can hardly think of any other project in Python that has as nice of documentation as Django. It may be worth learning some lessons on documentation from Django [2], on that note.
>
> I'd really like to have a good, thorough hacking-howto and deployment-howto, especially in the former making some notes on how to make it easier for Django hackers to get started.

4.3.4 Why MongoDB

(Note: We don't use MongoDB anymore. This is the original rationale, however.)

Chris Webber on "Why MongoDB":

[1] http://pythonpaste.org/webob/do-it-yourself.html
[2] http://pycon.blip.tv/file/4881071/

In case you were wondering, I am not a NOSQL fanboy, I do not go around telling people that MongoDB is web scale. Actually my choice for MongoDB isn't scalability, though scaling up really nicely is a pretty good feature and sets us up well in case large volume sites eventually do use MediaGoblin. But there's another side of scalability, and that's scaling down, which is important for federation, maybe even more important than scaling up in an ideal universe where everyone ran servers out of their own housing. As a memory-mapped database, MongoDB is pretty hungry, so actually I spent a lot of time debating whether the inability to scale down as nicely as something like SQL has with sqlite meant that it was out.

But I decided in the end that I really want MongoDB, not for scalability, but for flexibility. Schema evolution pains in SQL are almost enough reason for me to want MongoDB, but not quite. The real reason is because I want the ability to eventually handle multiple media types through MediaGoblin, and also allow for plugins, without the rigidity of tables making that difficult. In other words, something like:

```
{"title": "Me talking until you are bored",
 "description": "blah blah blah",
 "media_type": "audio",
 "media_data": {
     "length": "2:30",
     "codec": "OGG Vorbis"},
 "plugin_data": {
     "licensing": {
         "license": "http://creativecommons.org/licenses/by-sa/3.0/"}}}
```

Being able to just dump media-specific information in a media_data hashtable is pretty great, and even better is having a plugin system where you can just let plugins have their own entire key-value space cleanly inside the document that doesn't interfere with anyone else's stuff. If we were to let plugins to deposit their own information inside the database, either we'd let plugins create their own tables which makes SQL migrations even harder than they already are, or we'd probably end up creating a table with a column for key, a column for value, and a column for type in one huge table called "plugin_data" or something similar. (Yo dawg, I heard you liked plugins, so I put a database in your database so you can query while you query.) Gross.

I also don't want things to be too loose so that we forget or lose the structure of things, and that's one reason why I want to use MongoKit, because we can cleanly define a much structure as we want and verify that documents match that structure generally without adding too much bloat or overhead (MongoKit is a pretty lightweight wrapper and doesn't inject extra MongoKit-specific stuff into the database, which is nice and nicer than many other ORMs in that way).

4.3.5 Why Sphinx for documentation

Will Kahn-Greene on "Why Sphinx":

> Sphinx is a fantastic tool for organizing documentation for a Python-based project that makes it pretty easy to write docs that are readable in source form and can be "compiled" into HTML, LaTeX and other formats.
>
> There are other doc systems out there, but given that GNU MediaGoblin is being written in Python and I've done a ton of documentation using Sphinx, it makes sense to use Sphinx for now.

4.3.6 Why AGPLv3 and CC0?

Chris, Brett, Will, Rob, Matt, et al curated into a story where everyone is the hero by Will on "Why AGPLv3 and CC0":

> The AGPL v3 preserves the freedoms guaranteed by the GPL v3 in the context of software as a service. Using this license ensures that users of the service have the ability to examine the source, deploy their own

instance, and implement their own version. This is really important to us and a core mission component of this project. Thus we decided that the software parts should be under this license.

However, the project is made up of more than just software: there's CSS, images, and other output-related things. We wanted the templates/images/css side of the project all permissive and permissive in the same absolutely permissive way. We're waiving our copyrights to non-software things under the CC0 waiver.

That brings us to the templates where there's some code and some output. The template engine we're using is called Jinja2. It mixes HTML markup with Python code to render the output of the software. We decided the templates are part of the output of the software and not the software itself. We wanted the output of the software to be licensed in a hassle-free way so that when someone deploys their own GNU MediaGoblin instance with their own templates, they don't have to deal with the copyleft aspects of the AGPLv3 and we'd be fine with that because the changes they're making are identity-related. So at first we decided to waive our copyrights to the templates with a CC0 waiver and then add an exception to the AGPLv3 for the software such that the templates can make calls into the software and yet be a separately licensed work. However, Brett brought up the question of whether this allows some unscrupulous person to make changes to the software through the templates in such a way that they're not bound by the AGPLv3: i.e. a loophole. We thought about this loophole and between this and the extra legalese involved in the exception to the AGPLv3, we decided that it's just way simpler if the templates were also licensed under the AGPLv3.

Then we have the licensing for the documentation. Given that the documentation is tied to the software content-wise, we don't feel like we have to worry about ensuring freedom of the documentation or worry about attribution concerns. Thus we're waiving our copyrights to the documentation under CC0 as well.

Lastly, we have branding. This covers logos and other things that are distinctive to GNU MediaGoblin that we feel represents this project. Since we don't currently have any branding, this is an open issue, but we're thinking we'll go with a CC BY-SA license.

By licensing in this way, we make sure that users of the software receive the freedoms that the AGPLv3 ensures regardless of what fate befalls this project.

So to summarize:

- software (Python, JavaScript, HTML templates): licensed under AGPLv3
- non-software things (CSS, images, video): copyrights waived under CC0 because this is output of the software
- documentation: copyrights waived under CC0 because it's not part of the software
- branding assets: we're kicking this can down the road, but probably CC BY-SA

This is all codified in the `COPYING` file.

4.3.7 Why (non-mandatory) copyright assignment?

Chris Webber on "Why copyright assignment?":

> GNU MediaGoblin is a GNU project with non-mandatory but heavily encouraged copyright assignment to the FSF. Most, if not all, of the core contributors to GNU MediaGoblin will have done a copyright assignment, but unlike some other GNU projects, it isn't required here. We think this is the best choice for GNU MediaGoblin: it ensures that the Free Software Foundation may protect the software by enforcing the AGPL if the FSF sees fit, but it also means that we can immediately merge in changes from a new contributor. It also means that some significant non-FSF contributors might also be able to enforce the AGPL if seen fit.
>
> Again, assignment is not mandatory, but it is heavily encouraged, even incentivized: significant contributors who do a copyright assignment to the FSF are eligible to have a unique goblin drawing produced for them by the project's main founder, Christopher Allan Webber. See the wiki for details.

4.4 Migrations

So, about migrations. Every time we change the way the database structure works, we need to add a migration so that people running older codebases can have their databases updated to the new structure when they run *./bin/gmg dbupdate*.

The first time *./bin/gmg dbupdate* is run by a user, it creates the tables at the current state that they're defined in models.py and sets the migration number to the current migration... after all, migrations only exist to get things to the current state of the db. After that, every migration is run with dbupdate.

There's a few things you need to know:

- We use sqlalchemy-migrate. See their docs.
- Alembic might be a better choice than sqlalchemy-migrate now or in the future, but we originally decided not to use it because it didn't have sqlite support. It's not clear if that's changed.
- SQLAlchemy has two parts to it, the ORM and the "core" interface. We DO NOT use the ORM when running migrations. Think about it: the ORM is set up with an expectation that the models already reflect a certain pattern. But if a person is moving from their old patern and are running tools to *get to* the current pattern, of course their current database structure doesn't match the state of the ORM!
- How to write migrations? Maybe there will be a tutorial here in the future... in the meanwhile, look at existing migrations in *mediagoblin/db/migrations.py* and look in *mediagoblin/tests/test_sql_migrations.py* for examples.
- Common pattern: use *inspect_table* to get the current state of the table before we run alterations on it.
- Make sure you set the RegisterMigration to be the next migration in order.
- What happens if you're adding a *totally new* table? In this case, you should copy the table in entirety as it exists into migrations.py then create the tables based off of that... see add_collection_tables. This is easier than reproducing the SQL by hand.
- If you're writing a feature branch, you don't need to keep adding migrations every time you change things around if your database structure is in flux. Just alter your migrations so that they're correct for the merge into master.

That's it for now! Good luck!

CHAPTER

FIVE

INDICES AND TABLES

- *search*
- *genindex*

This guide was built on September 27, 2013.

PYTHON MODULE INDEX

m