Miro Community Documentation

Release 1.9.1

Participatory Culture Foundation

January 22, 2013

CONTENTS

Miro Community is an open-source video curation platfrom running on Django. Rather than uploading videos to yet another place on the internet, you can leverage videos which have already been uploaded to other services, such as blip.tv, Vimeo, YouTube, either by importing the videos individually or by importing RSS/Atom feeds of videos.

Miro Community was originally developed by the Participatory Culture Foundation to allow *anyone* to build a community video site, amplifying the voices of grassroots and citizen media makers and connecting local producers with local audiences. Over time, the platform has been used by a variety of other groups, including universities, open-source communities, and more.

website http://mirocommunity.org/

docs http://readthedocs.org/docs/mirocommunity/

bugtracker http://bugzilla.pculture.org/

code https://github.com/pculture/mirocommunity

mailing list http://groups.google.com/group/miro-community-development

irc #miro-hackers on irc.freenode.net

CHAPTER

ONE

CONTENTS

1.1 Installation

Note: There are a couple of things that this installation guide assumes:

- That you have installed Mercurial and Git on your system.
- That you have installed Python and virtualenv on your system.

These are basic instructions for installing a copy of Miro Community for local development and testing. You will need to modify the installation for a production environment - for example, you will need to draw up a requirements file that describes your production environment, and you will need to use your own settings file.

1.1.1 Creating a virtualenv

First up, you'll want to create and activate a virtual environment somewhere on your computer:

```
virtualenv testenv
cd testenv
source bin/activate
```

1.1.2 Installing Miro Community

Run the following commands from the root of your (installed and activated) virtualenv:

```
pip install -e git+git://github.com/pculture/mirocommunity.git@1.9.1#egg=mirocommunity --no-deps
cd src/mirocommunity/test_mc_project
pip install -r requirements.txt
python manage.py syncdb # This will prompt you to create an admin user
python manage.py runserver
```

Congratulations! You're running a local testing instance of Miro Community! You can access it in your web browser by navigating to http://127.0.0.1:8000/, and you can get the admin by navigating to http://127.0.0.1:8000/admin/.

If this is your first time using a Django app, you should definitely check out the Django tutorial to get a better understanding of what's going on, how to change the project settings, etc. The testing project can be a helpful place to start, but it is not meant to be used in a production setting.

Warning: Using the test project unaltered for a production server would be *extremely insecure*, because its `SECRET_KEY` is not secret.

1.2 Contributing to Miro Community

1.2.1 Reporting bugs and requesting features

One of the easiest ways you can help Miro Community is to put tickets into the bug tracker, either reporting bugs or asking for new features.

Reporting bugs

Here are some guidelines for good bug reports:

- Check the bug tracker to make sure that the bug has not already been reported.
- Ask on the IRC channel (#miro-hackers on irc.freenode.net) or the mailing list to make sure that what you're seeing really is a bug.
- Make sure that the bug is reproducible. Include instructions for how to reproduce it.
- Be as specific as possible. If a video page looks strange, check whether other video pages also look strange, then report which is the case.
- Give as much information as possible. Include error text, screenshots, links, anything that you have.

The better your bug report, the more likely someone is to fix it!

Reporting security issues

Please report security issues *only* to dev@mirocommunity.org.

Requesting features

When requesting a new feature, please do the following:

- Ask on the IRC channel (#miro-hackers on FreeNode) or the mailing list to get a general feeling on the feature.
- In your ticket, give a clear use case for/reason behind the new feature.

Ticket life cycle

- New tickets can be claimed by any community member.
- New and assigned tickets may be `RESOLVED` by a core member at any time with the following resolutions:
  - `INVALID`: The ticket isn't applicable to Miro Community. For example, someone suggesting a change to Django.
  - `WONTFIX`: The ticket will not be accepted, probably because it is not a bug, because the payoff is not seen as worth the effort, or because the ticket is rendered obsolete by parallel work on another ticket.
  - `DUPLICATE`: The ticket is already in the tracker.
  - `WORKSFORME`: The ticket would be a valid bug, but it can't be reproduced.
  - `INCOMPLETE`: More information is required to confirm the bug or explain the feature.
- Once a ticket is claimed, it is up to the assignee to start a branch for that ticket and submit a pull request to the canonical repository. When a pull request is submitted, the assignee should set the ticket's `needs-peer-review` flag to `?` and link to the pull request.

- Once a pull request has been submitted and the ticket has been flagged, a core member will review the code. This should be someone other than the assignee. If there are problems with the branch, they should explain the problems by commenting on github, inline and on the pull request. Otherwise, they can merge it in and change the ticket status to `RESOLVED/FIXED` and the `needs-peer-review` flag to `+`.

1.2.2 Writing Documentation

Miro Community uses Sphinx for documentation, which translates reStructuredText files into HTML, PDF, or other formats. The canonical documentation is compiled and made available on readthedocs, but you can also use Sphinx to compile the documentation on your local development machine.

If you don't feel like you can write very well, don't sweat it. Write what you can, and pass it off to someone else to improve. The first draft of a piece of documentation is hard to write simply because it doesn't exist. Writing it can give someone else a great starting point.

Here are some good resources on writing documentation:

- Django's docs on writing documentation
- RTFM - wRite The Friendly Manual

1.2.3 Contributing Code

You want to contribute code? Great! Here's how you can do it.

Finding tickets

Our tickets are all stored in our bugzilla installation; you can browse the tickets by component, or check out this summary of all open tickets.

Claiming tickets

If you don't see a ticket, feel free to open one! Any ticket that is assigned to admin@pculture.org is free to claim without asking, simply by assigning it to yourself. If a ticket is already claimed, but seems inactive, contact the assignee and see if it's all right for you to claim it. (Relatedly, if you don't actually have time to work on a ticket, don't claim it!)

Writing code

We use the same coding style as the Django project.

Submitting code

We accept code submissions as pull requests to our github repository, not as patch files, diff files, or anything else. Most of the time, when you submit code, it will be as a *ticket branch* - though if the change is extremely minor, you can submit a pull request without opening a ticket first. Pull requests must include any changes to unit tests and *documentation* that are needed.

Before we can accept your code submission, you will need to sign a `Contributor Assignment Agreement`, digitize it with a scanner or a good camera, and send it to legal@pculture.org. It's super easy, and it lets us keep the project open source.

Review process

After the pull request is made, a core contributor to the Miro Community project will review the code; they will either accept the pull request or point out where changes need to be made to get the branch ready for acceptance.

See Also:

Ticket life cycle

1.3 Git Branching Model

We use a variant of nvie's git branching model for our workflow.

The following main branches exist:

- `master` always points to the latest stable release. Any merge into master must represent a stable release as far as we can tell.
- `develop` always points to the latest development code. This should theoretically always be production-ready, but that is not guaranteed.

There are also some "supporting branch" types. Any supporting branch must be reviewed before being merged.

- Release branches. Naming convention: `release/<version_number>`. These branches originate from the development branch and may only receive bugfixes for release blockers. Essentially, they represent release candidates. The release branch should be merged back into the `develop` branch on a regular basis. When the release candidate is accepted, it will be merged into the master branch and the development branch. The master branch merge commit will be tagged with the new version number.
- Hotfix branches. Naming convention: `hotfix/<ticket_number>`. These branches represent *severe bugs* in the master branch. They originate from the master branch and fix a specific issue. Once checked and confirmed, they are merged into the master branch as a new point release, as well as being merged into the current release branch, or the development branch if no release is under way.
- Feature branches. Naming convention: `feature/<short_description>`. These branches represent large new features - for example, a large refactor or major UI change. Feature branches may only originate from the `develop` branch and may only be merged into the `develop` branch.
- Ticket branches. Naming convention: `ticket/<ticket_number>`. These branches represent tickets from the bug tracker which are not severe bugs in the master branch, and which have non-trivial solutions. Ticket branches may originate from `develop`, a feature branch (if the ticket is specific to that feature) or a release branch (if the ticket is a release blocker). They are merged into the branch they originated from.

  Trivial fixes to tickets can be made directly to the relevant branch, and should include the ticket number prominently in the commit message.

1.4 Release process

1.4.1 Version numbering

Miro Community numbers its versions as `A.B` or `A.B.C`.

- `A` is the *major* version number. This is incremented for broad, sweeping changes to Miro Community - for example, a refactor of the entire admin.
- `B` is the *minor* version number. This is incremented for new features which aren't broad, sweeping changes to Miro Community

- `C` is the *micro* version number. This is incremented for bug and security fixes.

Major releases

Very infrequent. May represent large changes.

Minor releases

The goal is to have minor releases on a fairly regular basis, every couple of months. Minor releases can add new features and remove features from previous releases.

Micro releases

Micro releases may not introduce new features; they may only fix critical issues:

- Security issues.
- Data loss.
- Crashing bugs/500 errors.
- Major bugs in new features from the latest minor release.

These bug fixes will be collected in a release branch; once all reported bugs have been resolved, the branch will be released. The one exception to this is security fixes, which cause the branch to be released immediately.

1.4.2 Release process

Alpha (development)

In this phase, features can be added to the upcoming release, with the approval of a core developer. Features with working patches are much more likely to be accepted than features with a thought-out design, which in turn are more likely to be accepted than off-hand suggestions.

Features will be marked on the following scale:

1. P1: Must have. The release can't happen without this.
2. P2: Should have. This would be good to have in the release.
3. P3: Maybe. This would be nice, but we don't need it.

Any features which are lower priority than that should be marked to the `future` milestone.

Bugs can also be added to the upcoming release, or marked `RESOLVED/WONTFIX` if an accepted P1 feature renders the bug irrelevant.

Micro releases do not have this phase.

Beta (bugfixes)

Features can no longer be added in this phase; this corresponds to the creation of a release branch. Any bugs which are rendered irrelevant by features which have made it in should be marked `RESOLVED/WONTFIX`.

Bugs can still be added to the upcoming release; bugs with patches are much more likely to be resolved.

Before this stage can end, all bugs which have been marked `RESOLVED/FIXED` must be verified and marked `VERIFIED/FIXED`.

Release candidate (blockers)

In this phase, the only bugfixes that will be addressed are critical issues:

- Security issues.
- Data loss.
- Crashing bugs/500 errors.
- Major bugs in new features introduced in this release.

Before this stage can end, all bugs which have been marked `RESOLVED/FIXED` must be verified and marked `VERIFIED/FIXED`.

Once this stage ends, the release will be merged into `master` and tagged with the new version number.

1.4.3 Support

Only the latest minor release will be supported with micro releases, all of which will be merged into the `develop` branch as well.

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*