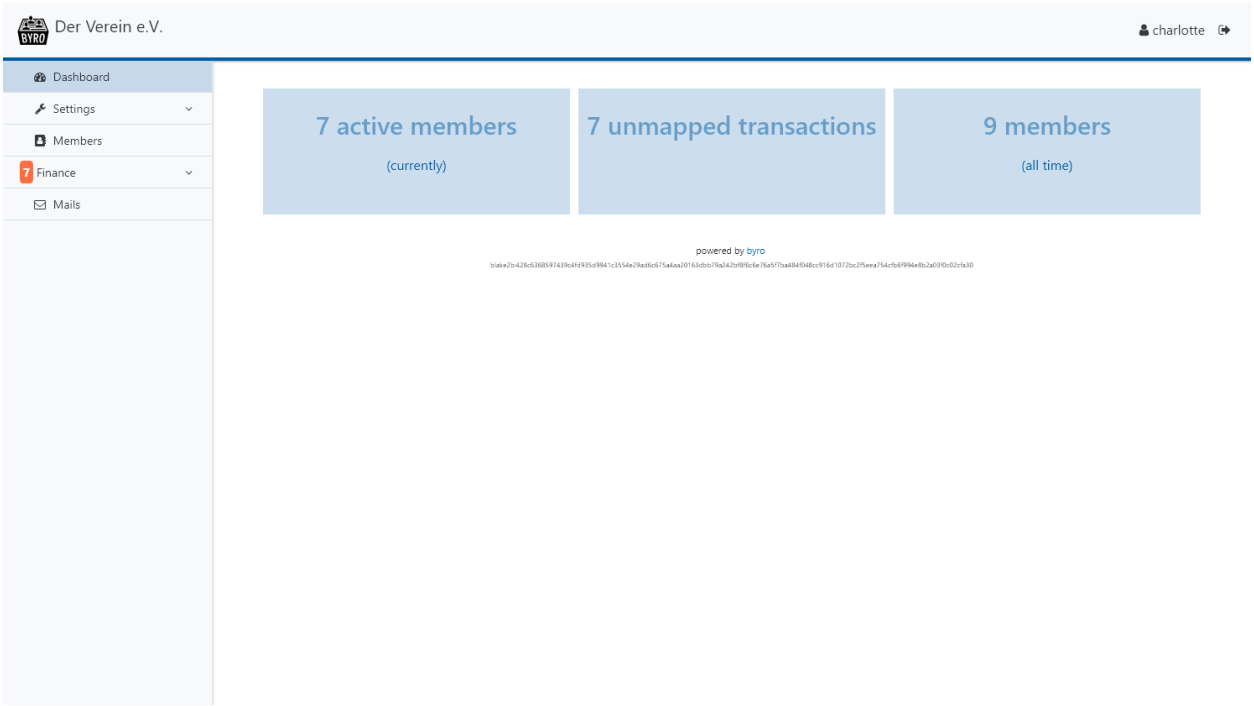
byro

Mar 07, 2019

Contents

byro is a membership administration tool. byro is best suited to small and medium sized clubs/NGOs/associations of all kinds, with a focus on the DACH region. byro is heavily plugin based to help fit it to different requirements in different situations and countries.



byro is stable and in active use in several communities. It is currently under active development.

CHAPTER 1

Developer documentation

This part of the byro documentation is for you if you want to improve byro or develop a plugin.

If you want to improve byro itself, please head to our *contributor documentation*.

If you want to work on byro's code (no matter which part), you'll want to start with the *Development setup*. It contains the contributing guidelines, an explanation on how to build a pull request, which style guidelines to follow, and an internal API documentation.

If you want to develop (or improve) a plugin, have a look at our *plugin development guides*: They show how to develop a byro plugin with some hopefully easy-to-follow example plugins.

If any part of the documentation is unclear, please head over to the *documentation setup* to learn how to improve it, and then follow the *contributor documentation* to create a pull request.

1.1 Development setup

To contribute to byro, it's useful to run byro locally on your device so you can test your changes. First of all, you need install some packages on your operating system:

If you want to install byro on a server for actual usage, go to the *administrator documentation* instead.

- git
- Python 3.x
- A recent version of pip (Ubuntu package: `python3-pip`)
- gettext (Debian package: `gettext`)
- libjpeg or any other library supported in pillow (for qrcode)
- libmagic
- A PostgreSQL server

Some Python dependencies might also need a compiler during installation, the Debian package `build-essential` or something similar should suffice.

1.1.1 Get a copy of the source code

You can clone our git repository:

```
git clone https://github.com/byro/byro.git
cd byro/
```

1.1.2 Database setup

Having the database server installed, we still need a database and a database user:

```
sudo -u postgres -i
postgres $ createuser <yourusername>
postgres $ createdb byro -O <yourusername>
```

Substitute your system username for `<yourusername>`.

1.1.3 Your local python environment

Please execute `python -V` or `python3 -V` to make sure you have Python 3.x installed. Also make sure you have pip for Python 3 installed, you can execute `pip3 -V` to check. Then use Python's internal tools (Ubuntu package: `python3-venv`) to create a virtual environment and activate it for your current session:

```
python3 -m venv env  # or virtualenv -p /usr/bin/python3 env, or ...
source env/bin/activate
```

You should now see a `(env)` prepended to your shell prompt. You have to do this in every shell you use to work with byro (or configure your shell to do so automatically). If you are working on Ubuntu or Debian, we strongly recommend upgrading your pip and setuptools installation inside the virtual environment, otherwise some of the dependencies might fail:

```
(env)$ pip3 install -U pip setuptools wheel
```

1.1.4 Working with the code

The first thing you need are all the main application's dependencies:

```
(env)$ cd src/
(env)$ pip3 install -r requirements/production.txt -r requirements/development.txt
```

Note: (Windows only) If you get the error message `failed to find libmagic. Check your installation` error, do `pip install python-magic-bin` in the virtual environment to install the necessary magic library for Windows.

Next, if you have custom database settings or other settings you need, make a new file `byro/local_settings.py` with contents like these:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
```

(continues on next page)

(continued from previous page)

```
        'NAME': 'byro',
        'USER': 'byro',
        'PASSWORD': 'byro',
        'HOST': 'localhost',
    }
}
```

(The default – and recommended – installation uses PostgreSQL "Peer Authentication", in which the Unix user is mapped to the Postgres database user. This works only for local connections, and only on Linux, most BSDs, OS X, and Solaris, but provides the highest level of security and the least amount of configuration. In this mode the keys `USER`, `PASSWORD`, and `HOST` MUST NOT be set.)

Then, create the local database:

```
(env)$ python manage.py migrate
```

To be able to log in, you should also create an admin user:

```
(env)$ python manage.py createsuperuser
```

If you want to see byro in a different language than English, you have to compile our language files:

```
(env)$ python manage.py compilemessages
```

Run the development server

To run the local development server, execute:

```
(env)$ python manage.py runserver
```

Now point your browser to http://localhost:8000/ – You should be able to log in and play around! You can use the following command to generate example data:

```
(env)$ python manage.py make_testdata
```

Code checks and unit tests

Before you check in your code into git, always run the static checkers and unit tests:

```
(env)$ pylama
(env)$ isort -c -rc .
(env)$ python manage.py check
(env)$ py.test tests
```

Note: If you have more than one CPU core and want to speed up the test suite, you can run `py.test -n NUM` with `NUM` being the number of threads you want to use.

It's a good idea to put the style checks into your git hook `.git/hooks/pre-commit`, for example:

```
#!/bin/sh
set -e
cd $GIT_DIR/../src
source ../env/bin/activate
pylama
isort -c -rc .
```

Working with translations

If you want to translate new strings that are not yet known to the translation system, you can use the following command to scan the source code for strings we want to translate and update the `*.po` files accordingly:

```
(env)$ python manage.py makemessages
```

To actually see byro in your language, you have to compile the `*.po` files to their optimized binary `*.mo` counterparts:

```
(env)$ python manage.py compilemessages
```

Next steps

To contribute to byro, please read the *contributing documentation*.

Head over to the *documentation patching section* if you want to improve the documentation.

If you want to work on plugins, please go to the *plugin guides*.

1.2 Contributing

We're always interested in improvements in byro, and **your** help is welcome! We'll review your contributions and give feedback on your changes, and help you if you're not sure how to solve a problem.

You'll need to have a GitHub account to contribute to byro, and should know how to pull, push, and commit using git.

If you have some improvement already in mind, please open an issue for it. Otherwise, look at our open issues and choose one you want to resolve. Don't hesitate to comment on the issue if anything is unclear.

First off, *fork byro*, and then clone your repository (GitHub will provide instructions). Once you have cloned your repository and opened it, create a new feature branch including the issue ID:

```
git checkout -b issue/123
```

If your issue requires code changes, complete the *development setup*, then continue here. If you want to change the documentation, please read up on the *documentation setup*.

We have a couple of style checkers both for code and for documentation, as documented in the setup docs. We check them in our Continuous Integration for every commit and pull request, but you should run the tests and checks locally as well, and consider your pull request ready once those tests pass.

Please write helpful, well-formatted commit messages – you can find a guide here. Once you have committed your work, add yourself to the `CONTRIBUTORS` file, and push your branch:

```
git push -u origin issue/123
```

and open a pull request. This will cause our Continuous Integration to check your changes for any issues (breaking tests, code style issues, documentation style issues, ...). Please give us five to seven days to get back to you with a review or a direct merge.

1.3 Patching documentation

You have found something to improve in our documentation? Great! We'll assume that you have already forked and cloned byro as detailed in the *contributing documentation*. For the following steps, you'll need to have Python 3 installed on your system.

Start out in a shell in the repository. We'll start by generating a virtualenv and installing the required Python packages:

```
python3 -m venv .venv
pip install -Ur src/requirements/documentation.txt
```

1.3.1 Writing documentation

Now go to the `docs` directory, find the file you want to adjust (or create), and make your changes. You can look at the files by running `make html` in the `docs` directory and then browsing the `_build/html` directory. For more convenience, you can run

```
sphinx-autobuild . ./_build/html
```

Which starts an HTTP server and rebuilds the documentation upon any changes.

1.3.2 Translating documentation

Our documentation is multilingual. To update the translation files, run

```
make gettext
sphinx-intl update -p _build/gettext -l de
```

and then edit the generated `.po` files with an appropriate editor such as poedit.

1.3.3 Checking documentation

In the future we want to use spell checking and style checking on our documentation.

1.4 Plugin Development

byro will not be usable for you without any plugins. We recommend that you have a look at our GitHub organisation to see what kind of plugins are already available and you can adapt it to your use cases.

byro plugin capabilities should be endless, so if you run into something you'd like to do with a plugin, and are missing a plugin hook, please tell us about it, and we'll be happy to add a hook to help you out.

The following pages will discuss plugin workings in general, list the plugin hooks provided so far, and show how to save member data that is not included by default, or add a custom payment provider.

If you run into trouble using this documentation, please tell us about it, too.

1.4.1 Creating a plugin

You can, and probably need to, extend byro with custom Python code using the official plugin API. You'll have to think of every plugin as an independent Django 'app' living in its own python package installed like any other python module.

The communication between byro and the plugins happens primarily using Django's signal dispatcher feature. The core modules of byro expose signals for different purposes. You can find their documentation on the next pages. We also provide guides for common plugin use cases, such as tracking custom member data, or importing and matching payments.

To create a new plugin, create a new python package which must be a valid Django app and must contain plugin metadata, as described below. You will need some boilerplate code for every plugin to get started. To save your time, we created a cookiecutter template that you can use like this:

```
(env)$ pip install cookiecutter
(env)$ cookiecutter https://github.com/byro/byro-plugin-cookiecutter
```

This will ask you some questions and then create a project folder for your plugin.

The following pages go into detail about the different types of plugins byro supports. While these instructions don't assume that you know a lot about byro, they do assume that you have prior knowledge about Django (e.g. its view layer, how its ORM works, etc.).

Plugin metadata

The plugin metadata lives inside a `ByroPluginMeta` class inside your app's configuration class. The metadata class must define the following attributes:

Attribute	Type	Description
name	string	The human-readable name of your plugin
author	string	Your name
version	string	A human-readable version code of your plugin
description	string	A more verbose description of what your plugin does.

A working example would be:

```
from django.apps import AppConfig
from django.utils.translation import ugettext_lazy as _


class IRCApp(AppConfig):
    name = 'byro_irc'
    verbose_name = _("IRC")

    class ByroPluginMeta:
        name = _("IRC")
        author = _("irclover")
        version = '1.0.0'
        visible = True
        description = _("This plugin sends notifications via IRC.")


default_app_config = 'byro_irc.IRCApp'
```

Plugin registration

Somehow, byro needs to know that your plugin exists at all. For this purpose, we make use of the entry point feature of setuptools. To register a plugin that lives in a separate python package, your `setup.py` should contain something like this:

```
setup(
    args...,
    entry_points="""
[byro.plugin]
byro_irc=byro_irc:ByroPluginMeta
"""
)
```

This will automatically make byro discover this plugin as soon as you have installed it, e.g. through `pip`. During development, you can run `python setup.py develop` inside your plugin source directory to make it discoverable.

Signals

byro defines different signals which your plugin can listen for. We will go into the details of the different signals in the following pages. We suggest that you put your signal receivers into a `signals` submodule of your plugin. You should extend your `AppConfig` (see above) by the following method to make your receivers available:

```
class IRCApp(AppConfig):
    ...

    def ready(self):
        from . import signals  # noqa
```

Views

Your plugin may define custom views. If you put an `urls` submodule into your plugin module, byro will automatically import it and include it into the root URL configuration with the namespace `plugins:<label>:`, where `<label>` is your Django app label.

Warning: If you define custom URLs and views, you are on your own with checking that the user has appropriate permissions. byro ensures that you are dealing with an authenticated user, but nothing else.

1.4.2 Signal list

This page lists the signals and hooks that are available in byro. The following guides will give examples on how to use these signals.

Member management

`byro.members.signals.`**`new_member = <django.dispatch.dispatcher.Signal object>`**

Receives the new member as signal. If an exception is raised, the error message will be displayed in the frontend as a warning.

byro.members.signals.**new_member_mail_information = <django.dispatch.dispatcher.Signal objec**
Receives the new member as signal. Response will be added to the email welcoming the new member.

byro.members.signals.**new_member_office_mail_information = <django.dispatch.dispatcher.Signa**
Receives the new member as signal. Response will be added to the email notifying the office about the new member.

byro.members.signals.**leave_member = <django.dispatch.dispatcher.Signal object>**
Receives the new member as signal. If an exception is raised, the error message will be displayed in the frontend as a warning.

byro.members.signals.**leave_member_mail_information = <django.dispatch.dispatcher.Signal obj**
Receives the leaving member as signal. Response will be added to the email confirming termination to the member.

byro.members.signals.**leave_member_office_mail_information = <django.dispatch.dispatcher.Sig**
Receives the leaving member as signal. Response will be added to the email notifying the office about the termination of the member.

byro.members.signals.**update_member = <django.dispatch.dispatcher.Signal object>**
If a member is updated via the office form collection at members/view/{id}/data. The signal receives the request, and the form_list as parameters. The changes will already have been saved at this point.

Payment

byro.bookkeeping.signals.**process_transaction = <django.dispatch.dispatcher.Signal object>**
This signal provides a Transaction as sender and expects the receiver to augment the Transaction with auto-detected information as appropriate.

The common case is a Transaction that is unbalanced and can be augmented to be a balanced Transaction by adding one or more Bookings.

Recipients MUST NOT change any data in the Transaction or its Bookings if Transaction.is_read_only is True.

byro.bookkeeping.signals.**process_csv_upload = <django.dispatch.dispatcher.Signal object>**
This signal provides a RealTransactionSource as sender and expects a list of one or more Transactions in response.

If the RealTransactionSource has already been processed, no Transactions should be created, unless you are very sure what you are doing.

Display

byro.office.signals.**nav_event = <django.dispatch.dispatcher.Signal object>**
This signal allows you to add additional views to the sidebar. Receives the request as sender. Must return a dictionary containing at least the keys `label` and `url`. You can also return a ForkAwesome icon name iwth the key `icon`. You should also return an `active` key with a boolean set to `True` if this item should be marked as active.

If you want your Plugin to appear in the "Finance" or "Settings" submenu in the side bar, please set `section` in your return dict to either `finance` or `settings`, and don't set an `icon`.

May return an iterable of multiple dictionaries as described above.

byro.office.signals.**member_view = <django.dispatch.dispatcher.Signal object>**
This signal allows you to add a tab to the member detail view tab list. Receives the member as sender, and additionally the request Must return a dict:

```
{
    "label": _("Fancy Member View"),
    "url": "/member/123/foo/",
    "url_name": "plugins:myplugin:foo_view",
}
```

Please use byro.office.views.members.MemberView as base class for these views.

byro.office.signals.**member_dashboard_tile = <django.dispatch.dispatcher.Signal object>**
This signal allows you to add tiles to the member's dashboard. Receives None as argument, must return either None or a dict:

```
{
    "title": _("Dash!"),
    "lines": [_('Line 1'), _('Line 2')]
    "url": "/member/123/foo/",
    "public": False,  # False is the default
}
```

All of the parts of this dict are optional. You cannot include HTML in the response, all strings will be escaped at render time. If "public" is set to True, the dasboard tile will also be shown on the member's personal page.

byro.common.signals.**unauthenticated_urls = <django.dispatch.dispatcher.Signal object>**
This signal is used to compile a list of URLs that should bypass the normal authentication middleware. The return value must be an iterable where each item is either a) a local view name, or b) a callable with signature (request, resolver_match) that should return True if authentication should be bypassed. Note: in case a) only the local name must be provided, not with the "plugins:$plugin_name:" namespace prefix, in case b) the callable will see the request and resolver_match with the full name, including namespace prefix.

byro.common.signals.**log_formatters = <django.dispatch.dispatcher.Signal object>**
This signal is used to compile a list of log entry formatters. The return value must be a mapping of action_type: callable(LogEntry) -> html_fragment: str

Import

byro.office.signals.**member_list_importers = <django.dispatch.dispatcher.Signal object>**
This signal allows you to add additional member list importers. Receives None as argument, must return a dict:

```
{
    "id": "dot.scoped.importer.id",
    "label": _("My super importer"),
    "form_valid": form_valid_callback,
}
```

where *form_valid_callback* should accept two arguments: view (the View object handling the request), and form (the form object that was submitted, the file to import is in the *upload_file* form field) and should return a Response object.

General

byro.common.signals.**periodic_task = <django.dispatch.dispatcher.Signal object>**
This is a signal that we send out every time the periodic task cronjob runs. This interval is not sharply defined, it can be everything between a minute and a day. The actions you perform should be idempotent, i.e. it should not make a difference if this is sent out more often than expected.

1.4.3 Plugin: Custom member data

Most groups will need to save more data about their members than byro does by default.

General

You can save custom member data via a special model class, which **must** reference `byro.members.Member` in a OneToOne relation, and the related name **must** start with "profile".

Once you have generated this plugin (and have added the migrations, and run them), byro will discover the profile on its own, generate the fitting forms for members' profile pages, and offer you to include it when configuring your registration form.

The Profile class

If you want to track, for example, if a member has agreed to receive your newsletter, you'd add a `models.py` file to your plugin, and put this inside:

```
from annoying.fields import AutoOneToOneField
from django.db import models

class NewsletterProfile(models.Model):
    member = AutoOneToOneField(
        to='members.Member',
        on_delete=models.CASCADE,
        related_name='profile_shack',
    )
    receives_newsletter = models.BooleanField(default=True)

    def get_member_data(self):
         return [
            "You have opted in to receive our newsletter." if self.receives_
↪newsletter else "",
         ]
```

Members will receive occasional emails with all data that is saved about them – you can either return a list of strings, or a list of tuples (of keys and values, such as `("Has agreed to receive the newsletter", "True")`). If you do not implement this method, byro will display all relevant data from this profile directly.

Custom views

If you want to add an custom tab to a member's view related to your new content, you'll have to write a simple view, add its url in your `urls.py`, and register it in your `signals.py`:

```
from django.dispatch import receiver
from django.urls import reverse
from django.utils.translation import ugettext_lazy as _

from byro.office.signals import member_view


@receiver(member_view)
def newsletter_member_view(sender, signal, **kwargs):
    member = sender
```

(continues on next page)

(continued from previous page)

```
    return {
        'label': _('Newsletter'),
        'url': reverse('plugins:byro_newsletter:members.newsletter', kwargs={'pk': member.pk}),
        'url_name': 'plugins:byro_newsletter',
    }
```

Every member will now have a tab with the label "Newsletter". You could also add a general newsletter view to the sidebar:

```
from django.dispatch import receiver
from django.urls import reverse
from django.utils.translation import ugettext_lazy as _

from byro.office.signals import nav_event

@receiver(nav_event)
def newsletter_sidebar(sender, **kwargs):
    request = sender
    return {
        'icon': 'envelope-o',
        'label': _('Newsletter'),
        'url': reverse('plugins:byro_newsletter:dashboard'),
        'active': 'byro_newsletter' in request.resolver_match.namespace and 'member' not in request.resolver_match.url_name,
    }
```

Configuring your plugin

If you'd like to provide custom configuration options (for example, the name or latest issue of your newsletter), you can add a special configuration related model. If the model class inherits from `ByroConfiguration` and ends in `Configuration`, it will be automatically added to the settings page:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

from byro.common.models.configuration import ByroConfiguration


class NewsletterConfiguration(ByroConfiguration):

    url = models.CharField(
        null=True, blank=True,
        max_length=300,
        verbose_name=_('Newsletter information URL'),
        help_text=_('e.g. https://foo.bar.de/news')
    )
```

CHAPTER 2

Administrator documentation

byro is free software, which means you can run it yourself on your own server (or your own Raspberry Pi, ...). But while this offers you great freedom, it also comes with great responsibility:

> **Warning:** Hosting byro means taking responsibility for your members' personal and financial data. Please make sure that your installation and servers are secure and will be maintained in the future. If you don't feel comfortable with this, consider contacting us for information, or choosing an offline installation.

The following pages document a uncomplicated setup without going into details on administrative basics like securing your server, or performing backups.

2.1 Installation

This guide will help you to install byro on a Linux distribution, as long as the prerequisites are present.

2.1.1 Step 0: Prerequisites

Please set up the following systems beforehand, we'll not explain them here (but see these links for external installation guides):

- **Python 3.5+** and `pip` for Python 3. You can use `python -V` and `pip3 -V` to check.
- An SMTP server to send out mails
- An HTTP reverse proxy, e.g. nginx or Apache to allow HTTPS connections
- A PostgreSQL (9.4 or higher) database server.

We also recommend that you use a firewall, although this is not a byro-specific recommendation. If you're new to Linux and firewalls, we recommend that you start with ufw.

Note: Please do not run byro without HTTPS encryption. You'll handle sensitive data and thanks to Let's Encrypt, SSL certificates are free these days. We also *do not* provide support for HTTP-exclusive installations except for evaluation purposes.

2.1.2 Step 1: Unix user

As we do not want to run byro as root, we first create a new unprivileged user:

```
# adduser byro --disabled-password --home /var/byro
```

In this guide, all code lines prepended with a `#` symbol are commands that you need to execute on your server as `root` user (e.g. using `sudo`); you should run all lines prepended with a `$` symbol as the unprivileged user.

2.1.3 Step 2: Database setup

Having the database server installed, we still need a database and a database user. As the `postgres` user, execute:

```
postgres $ createuser byro -P
Enter password for new role:
Enter it again:
postgres $ createdb byro
postgres $ psql
postgres=# GRANT ALL PRIVILEGES ON DATABASE byro to byro;
```

Replace the asterisks with a password of your own.

2.1.4 Step 3: Package dependencies

To build and run byro, you will need the following Debian packages beyond the dependencies mentioned above:

```
# apt-get install git build-essential libssl-dev gettext
```

Replace all further "pip" commands with "pip3" if your system does not have Python 3 as default Python version.

2.1.5 Step 4: Configuration

We now create a configuration directory and configuration file for byro:

```
# mkdir /etc/byro
# touch /etc/byro/byro.cfg
# chown -R byro:byro /etc/byro/
# chmod 0600 /etc/byro/byro.cfg
```

Fill the configuration file `/etc/byro/byro.cfg` with the following content (adjusted to your environment):

```
[filesystem]
data = /var/byro/data
media = /var/byro/data/media
logs = /var/byro/data/logs
```

(continues on next page)

(continued from previous page)

```
[site]
debug = False
url = https://byro.mydomain.com

[database]
name = byro
user = byro
password = byro
host = localhost
port = 5432

[mail]
from = admin@localhost
host = localhost
port = 25
user = admin
password = something
tls = False
ssl = True
```

2.1.6 Step 5: Installation

Now we will install byro itself. Please execute the following steps as the `byro` user. We will install all Python packages, including byro, in the user's Python environment, so that your global Python installation will not know of them:

```
$ pip install --user -U pip setuptools wheel byro gunicorn psycopg2-binary
```

We also need to create a data directory:

```
$ mkdir -p /var/byro/data/media
```

We compile static files and translation data and create the database structure:

```
$ python -m byro migrate
$ python -m byro rebuild
```

Now, create an administrator user by running:

```
$ python -m byro createsuperuser
```

If you just want to play around with byro, you can load test data:

```
$ python -m byro make_testdata
```

2.1.7 Step 6: Starting byro as a service

We recommend starting byro using systemd to make sure it starts up after a reboot. Create a file named `/etc/systemd/system/byro-web.service` with the following content:

```
[Unit]
Description=byro web service
After=network.target
```

(continues on next page)

(continued from previous page)

```
[Service]
User=byro
Group=byro
WorkingDirectory=/var/byro/.local/lib/python3.5/site-packages/byro
ExecStart=/var/byro/.local/bin/gunicorn byro.wsgi \
                      --name byro --workers 4 \
                      --max-requests 1200  --max-requests-jitter 50 \
                      --log-level=info --bind=127.0.0.1:8345
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

You can now run the following commands to enable and start the services:

```
# systemctl daemon-reload
# systemctl enable byro-web
# systemctl start byro-web
```

2.1.8 Step 7: SSL

The following snippet is an example on how to configure a nginx proxy for byro:

```
server {
    listen 80 default_server;
    listen [::]:80 ipv6only=on default_server;
    server_name byro.mydomain.com;
}
server {
    listen 443 default_server;
    listen [::]:443 ipv6only=on default_server;
    server_name byro.mydomain.com;

    ssl on;
    ssl_certificate /path/to/cert.chain.pem;
    ssl_certificate_key /path/to/key.pem;

    add_header Referrer-Options same-origin;
    add_header X-Content-Type-Options nosniff;

    location / {
        proxy_pass http://localhost:8345/;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header Host $http_host;
    }

    location /media/ {
        alias /var/byro/data/media/;
        add_header Content-Disposition 'attachment; filename="$1"';
        expires 7d;
        access_log off;
    }
```

(continues on next page)

(continued from previous page)

```
    location /static/ {
        alias /path/to/static.dist/;
        access_log off;
        expires 365d;
        add_header Cache-Control "public";
    }
}
```

Note: Remember to replace the `python3.5` in the `/static/` path in the config above with your python version.

We recommend reading about setting strong encryption settings for your web server.

You've made it! You should now be able to reach byro at https://byro.yourdomain.com/ and log in as the administrator you configured above. byro will take you through the remaining configuration steps.

2.1.9 Step 8: Check the installation

You can make sure the web interface is up and look for any issues with:

```
# journalctl -u byro-web
```

In the start-up output, byro also lists its logging directory, which is also a good place to look for the reason for issues.

2.1.10 Next Steps: Updates

Warning: While we try hard not to issue breaking updates, **please perform a backup before every upgrade**.

To upgrade byro, please first read through our changelog and if available our release blog post to check for relevant update notes. Also, make sure you have a current backup.

Next, please execute the following commands in the same environment (probably your virtualenv) to first update the byro source, then update the database if necessary, then rebuild changed static files, and then restart the byro service. Please note that you will run into an entertaining amount of errors if you forget to restart the services.

If you want to upgrade byro to a specific release, you can substitute `byro` with `byro==1.2.3` in the first line:

```
$ pip3 install -U byro gunicorn
$ python -m byro migrate
# systemctl restart byro-web
```

2.2 Configuration

You can configure byro in two different ways: using configuration files or environment variables. You can combine those two options, and their precedence is in this order:

1. Environment variables
2. **Configuration files**
    - The configuration file in the environment variable `BYRO_CONFIG_FILE` if present, **or**:

- **The following three configuration files in this order:**
  - The configuration file `byro.cfg` in the `src` directory, next to the `byro.example.cfg` file.
  - The configuration file `~/.byro.cfg` in the home of the executing user.
  - The configuration file `/etc/byro/byro.cfg`

3. Sensible defaults

This page explains the options by configuration file section and notes the corresponding environment variable next to it. A configuration file looks like this:

```
[filesystem]
data = /var/byro/data
media = /var/byro/data/media
logs = /var/byro/data/logs

[site]
debug = False
url = https://byro.mydomain.com

[database]
name = byro
user = byro
password = byro
host = localhost
port = 5432

[mail]
from = admin@localhost
host = localhost
port = 25
user = admin
password = something
tls = False
ssl = True
```

2.2.1 The filesystem section

data

- The `data` option describes the path that is the base for the media files directory, and where byro will save log files. Unless you have a compelling reason to keep those files apart, setting the `data` option is the easiest way to configure byro.
- **Environment variable:** `BYRO_DATA_DIR`
- **Default:** A directory called `data` next to byro's `manage.py`.

media

- The `media` option sets the media directory that contains user generated files. It needs to be writeable by the byro process.
- **Environment variable:** `BYRO_FILESYSTEM_MEDIA`

- **Default:** A directory called `media` in the `data` directory (see above).

logs

- The `logs` option sets the log directory that contains logged data. It needs to be writeable by the byro process.
- **Environment variable:** `BYRO_FILESYSTEM_LOGS`
- **Default:** A directory called `logs` in the `data` directory (see above).

static

- The `statics` option sets the directory that contains static files. It needs to be writeable by the byro process. byro will put files there during the `collectstatic` command.
- **Environment variable:** `BYRO_FILESYSTEM_STATIC`
- **Default:** A directory called `static.dist` next to byro's `manage.py`.

2.2.2 The site section

debug

- Decides if byro runs in debug mode. Please use this mode for development and debugging, not for live usage.
- **Environment variable:** `BYRO_DEBUG`
- **Default:** `True` if you're executing `runserver`, `False` otherwise. **Never run a production server in debug mode.**

url

- This value will appear wherever byro needs to render full URLs (for example in emails and), and set the appropriate allowed hosts variables.
- **Environment variable:** `BYRO_SITE_URL`
- **Default:** `http://localhost`

secret

- Every Django application has a secret that Django uses for cryptographic signing. You do not need to set this variable – byro will generate a secret key and save it in a local file if you do not set it manually.
- **Default:** None

2.2.3 The database section

name

- The database's name.
- **Environment variable:** `BYRO_DB_NAME`
- **Default:** `''`

user

- The database user.
- **Environment variable:** `BYRO_DB_USER`
- **Default:** `''`

password

- The database password.
- **Environment variable:** `BYRO_DB_PASS`
- **Default:** `''`

host

- The database host, or the socket location, as needed.
- **Environment variable:** `BYRO_DB_HOST`
- **Default:** `''`

port

- The database port.
- **Environment variable:** `BYRO_DB_PORT`
- **Default:** `''`

2.2.4 The mail section

from

- The fall-back sender address, e.g. for when byro sends event independent emails.
- **Environment variable:** `BYRO_MAIL_FROM`
- **Default:** `admin@localhost`

host

- The email server host address.
- **Environment variable:** `BYRO_MAIL_HOST`
- **Default:** `localhost`

port

- The email server port.
- **Environment variable:** `BYRO_MAIL_PORT`
- **Default:** `25`

user

- The user account for mail server authentication, if needed.
- **Environment variable:** `BYRO_MAIL_USER`
- **Default:** `''`

password

- The password for mail server authentication, if needed.
- **Environment variable:** `BYRO_MAIL_PASSWORD`
- **Default:** `''`

tls

- Should byro use TLS when sending mail? Please choose either TLS or SSL.
- **Environment variable:** `BYRO_MAIL_TLS`
- **Default:** `False`

ssl

- Should byro use SSL when sending mail? Please choose either TLS or SSL.
- **Environment variable:** `BYRO_MAIL_SSL`
- **Default:** `False`

2.2.5 The logging section

email

- The email address (or addresses, comma separated) to send system logs to.
- **Environment variable:** `BYRO_LOGGING_EMAIL`
- **Default:** `''`

email_level

- The log level to start sending emails at. Any of `[DEBUG, INFO, WARNING, ERROR, CRITICAL]`.
- **Environment variable:** `BYRO_LOGGING_EMAIL_LEVEL`
- **Default:** `'ERROR'`

2.2.6 The locale section

language_code

- The system's default locale.
- **Environment variable:** `BYRO_LANGUAGE_CODE`
- **Default:** `'de'`

time_zone

- The system's default time zone as a `pytz` name.
- **Environment variable:** `BYRO_TIME_ZONE`
- **Default:** `'UTC'`

CHAPTER 3

Features

As byro is under active development, this feature list can become outdated. Please open issues for features you are missing!

- **Member management:** Add, and edit members and their data.
- **Membership management:** Add and change the membership fees a member should pay.
- **Add custom member data:** Track non-standard member data by adding a plugin to byro. There are plenty of example plugins and developer documentation to help you.
- **Import payment data:** Inbuilt support for CSV imports.
- **Import and match payment data** to members via custom methods, added by plugins.
- **Send mails:** All mails can be reviewed before they are sent out. You can also edit the default mail templates and add new ones.
- **See member balances**. You can also check every single transaction at any time.
- **Upload member specific documents:** (either for or by them); optionally send them per mail automatically.
- **Let members interact:** Members can choose to make their data (which parts is their decision) visible to other members. Having a look at the member directory helps them interact directly with other members.

Please note that byro is a tool for tracking member data and payments, and the administrative acts around it. byro does support bookkeeping and transactions, but it is not a complete bookkeeping tool (yet).

Python Module Index

b

Index