Dissemin Documentation

Release 0.1

CAPSH

Mar 19, 2019

Contents

Dissemin is a web platform gathering academic publications and analyzing their online availability as well as publisher policy. This documentation explains how it works and how to use it.

This serves as the main location for the documentation. First, you might want to refer to *Installation*.

You will also need to follow *Getting API keys* to have access to the metadata sources.

CHAPTER 1

Dissemin API

Dissemin provides an API to query the availability of arbitrary papers. Please do not assume the interface will not change in the future as it is still being improved.

1.1 Querying by DOI

You can retrieve Dissemin's metadata for a specific paper by DOI:

https://dissem.in/api/10.1016/j.paid.2009.02.013.

1.2 Searching for papers

The search interface is also exposed as an API. The parameters it understands are the same as the human-readable version at https://dissem.in/search . Statistics about the results are also returned as well.

https://dissem.in/api/search/?q=pregroup

1.3 Understanding the results

```
{

    "status": "ok",
    "paper": {
        "classification": "UNK",
        "title": "Refining the Conceptualization of an Important
Future-Oriented Self-Regulatory Behavior: Proactive Coping",
        "pdf\_url": "http://www.ncbi.nlm.nih.gov/pubmed/19578529",
        "records": [
            {
```

(continues on next page)

(continued from previous page)


```
                "splash\_url":
"https://doi.org/10.1016/j.paid.2009.02.013",
                "doi": "10.1016/j.paid.2009.02.013",
                "publisher": "Elsevier BV",
                "issue": "2",
                "journal": "Personality and Individual Differences",
                "issn": "0191-8869",
                "volume": "47",
                "source": "crossref",
                "policy": {
                    "romeo\_id": "30",
                    "preprint": "can",
                    "postprint": "can",
                    "published": "cannot"
                },
                "identifier":
"oai:crossref.org:10.1016/j.paid.2009.02.013",
                "type": "journal-article",
                "pages": "139-144"
            },
            {
                "splash\_url":
"https://www.researchgate.net/publication/26648440\_Refining\_the\_Conceptualization\_
↪of\_an\_Important\_Future-Oriented\_Self-Regulatory\_Behavior\_Proactive\_Coping",
                "doi": "10.1016/j.paid.2009.02.013",
                "contributors": "",
                "abstract": "Proactive coping, directed at an upcoming as
opposed to an ongoing stressor, is a new focus in positive psychology
research. However, two differing conceptualizations of this construct
create confusion. This study compared how each operationalization of
proactive coping relates to well-being. Participants (N = 281) facing an
upcoming college examination completed the Proactive Coping Inventory
(PCI; consisting of two subscales that each assess one of the
conceptualizations), the Proactive Competence Scale (PCS; that assesses
the proactive coping process), and measures of well-being. The results
demonstrated that conceptualizing proactive coping as a
positively-focused striving for goals was predictive of well-being (the
shared variance from affect, subjective well-being and physical
symptoms), whereas conceptualizing proactive coping as focused on
preventing a negative future was not. The first conceptualization of
proactive coping's unique association with well-being was explained by
two of the proactive competencies, use of resources and realistic goal
setting, and the remaining variance in well-being was explained by the
first factor of optimism. These results demonstrated that aspiring for a
positive future is distinctly predictive of well-being and that research
should focus on accumulating resources and goal setting in designing
interventions to promote proactive coping.",
                "pdf\_url":
"https://www.researchgate.net/profile/Stephanie\_Sohl2/publication/26648440\_
↪Refining\_the\_Conceptualization\_of\_an\_Important\_Future-Oriented\_Self-
↪Regulatory\_Behavior\_Proactive\_Coping/links/55e463c008ae2fac47227a76.pdf",
                "source": "researchgate",
                "keywords": "",
                "identifier": "oai:researchgate.net:26648440",
                "type": "journal-article"
            },
            {
```


(continues on next page)

(continued from previous page)


```
                "splash\_url":
"http://www.ncbi.nlm.nih.gov/pubmed/19578529",
                "doi": "10.1016/j.paid.2009.02.013",
                "contributors": "",
                "abstract": "Proactive coping, directed at an upcoming as
opposed to an ongoing stressor, is a new focus in positive psychology
research. However, two differing conceptualizations of this construct
create confusion. This study compared how each operationalization of
proactive coping relates to well-being. Participants (N = 281) facing an
upcoming college examination completed the Proactive Coping Inventory
(PCI; consisting of two subscales that each assess one of the
conceptualizations), the Proactive Competence Scale (PCS; that assesses
the proactive coping process), and measures of well-being. The results
demonstrated that conceptualizing proactive coping as a
positively-focused striving for goals was predictive of well-being (the
shared variance from affect, subjective well-being and physical
symptoms), whereas conceptualizing proactive coping as focused on
preventing a negative future was not. The first conceptualization of
proactive coping's unique association with well-being was explained by
two of the proactive competencies, use of resources and realistic goal
setting, and the remaining variance in well-being was explained by the
first factor of optimism. These results demonstrated that aspiring for a
positive future is distinctly predictive of well-being and that research
should focus on accumulating resources and goal setting in designing
interventions to promote proactive coping.",
                "pdf\_url": "http://www.ncbi.nlm.nih.gov/pubmed/19578529",
                "source": "base",
                "keywords": "Article",
                "identifier":
"ftpubmed:oai:pubmedcentral.nih.gov:2705166",
                "type": "other"
            }
        ],
        "authors": [
            {
                "name": {
                    "last": "Sohl",
                    "first": "Stephanie Jean"
                }
            },
            {
                "name": {
                    "last": "Moyer",
                    "first": "Anne"
                }
            }
        ],
        "date": "2009-07-01",
        "type": "journal-article"
    }

}
```


Most fields are self-explanatory, here is a quick description of the other ones:

- ***classification*** is the code for the self-archiving policy of the publisher "OA" (available from the publisher), "OK" (some version can be shared), "UNK" (unknown/unclear sharing policy), "NOK" (restrictive sharing

policy).

- ***pdf_url*** is the URL where dissemin thinks the full text can be accessed for free. This is rarely a direct link to an actual PDF file. It is set to `null` if we could not find a free source for this paper.
- ***records*** gives a list of the places where the full text has been made available (so: repositories, homepages or social networks). Sometimes, these repositories only contain a bibliographical record and not the full text. The ***pdf_url*** field of each record indicates our assessment of the availability of that record. If the publisher has been found in RoMEO, it also indicates the summary of its policy, using the codes drawn from the RoMEO API. This list will remain empty if no DOI is provided.

1.4 License, usage

CAPSH claims no ownership of the metadata served via this API. It has been collected from various free sources.

The interface itself should not be abused: please do not use concurrent connections on it, and keep your requests to a slow rate (at most one per second). If you need a faster access to this data, please get in touch with us.

CHAPTER 2

Installation

There are two ways to install Dissemin. The automatic way uses Vagrant to install Dissemin to a container or VM, and only takes a few commands. The manual way is more complex and is described afterwards.

2.1 Automatic installation with Vagrant

First, install Vagrant and one of the supported providers: VirtualBox (should work fine), LXC (tested), libvirt (try it and tell us!). Then run the following commands:

- `git clone https://github.com/dissemin/dissemin` will clone the repository, i.e., download the source code of Dissemin. You should not reuse an existing copy of the repository, otherwise it may cause errors with Vagrant later.
- `cd dissemin` to go in the repository
- `vagrant up --provider=your_provider` will create the VM / container and provision the machine once
- `vagrant ssh` will let you poke into the machine and access its services (PostgreSQL, Redis, ElasticSearch)
- A tmux session is running so that you can check out the Celery and Django development server, attach it using: `tmux attach`. It contains a `bash` panel, two panels to check on Celery and Django development server and a panel to create a superuser (admin account) for Dissemin, which you can then use from *localhost:8080/admin*.

Dissemin will be available on your host machine at *localhost:8080*.

Note that, when rebooting the Vagrant VM / container, the Dissemin server will not be started automatically. To do it, once you have booted the machine, run `vagrant ssh` and then `cd /dissemin` and `./launch.sh` and wait for some time until it says that Dissemin has started. The same holds for other backend services, you can check the `Vagrantfile` and `provisioning/provision.sh` to find out how to start them.

2.2 Manual installation

This section describes manual installation, if you cannot or do not want to use Vagrant as indicated above. dissem.in is split in two parts:

- the web frontend, powered by Django;
- the tasks backend, powered by Celery.

Installing the tasks backend requires additional dependencies and is not necessary if you want to do light dev that does not require harvesting metadata or running author disambiguation. The next subsections describe how to install the frontend; the last one explains how to install the backend or how to bypass it in case you do not want to install it.

2.2.1 Installation instructions for the web frontend

First, install the following dependencies (debian packages) `postgresql postgresql-server-dev-all postgresql-client python3-venv build-essential libxml2-dev libxslt1-dev python3-dev gettext libjpeg-dev libffi-dev`

Then, build a virtual environment to isolate all the python dependencies:

```
python3 -m venv .virtualenv
source .virtualenv/bin/activate
pip install --upgrade setuptools
pip install --upgrade pip
pip install -r requirements.txt
```

2.2.2 Set up the database

Choose a unique database and user name (they can be identical), such as `dissemin_myuni`. Choose a strong password for your user:

```
sudo su postgres
psql
CREATE USER dissemin_myuni WITH PASSWORD 'b3a55787b3adc3913c2129205821765d';
ALTER USER dissemin_myuni CREATEDB;
CREATE DATABASE dissemin_myuni WITH OWNER dissemin_myuni;
```

2.2.3 Configure the secrets

Copy `dissemin/settings/secret_template.py` to `dissemin/settings/secret.py`. Edit this file to change the following settings:

- Create a fresh `SECRET_KEY` (any random-looking string that you can keep secret will do).
- Configure the `DATABASES` with the database you've set up earlier.
- (Optional) Set up the SMTP parameters to send emails to authors, in the `EMAIL` section.
- `ROMEO_API_KEY` and `PROAIXY_API_KEY` are required if you want to import papers automatically from these sources. See *Getting API keys* about how to get them.

2.2.4 Install and configure the search engine

dissem.in uses the Elasticsearch backend for Haystack. The current supported version is **2.x.x**.

Download Elasticsearch and unzip it:

```
cd elasticsearch-<version>
./bin/elasticsearch    # Add -d to start elasticsearch in the background
```

Alternatively you can install the .rpm or .deb package, see the documentation of Elasticsearch for further information.

Make sure to set the initial *heapsize <https://www.elastic.co/guide/en/elasticsearch/reference/2.4/setup-configuration.html#_environment_variables>* accordingly.

2.2.5 Configure the application for development or production

Finally, create a file `dissemin/settings/__init__.py` with this content:

```
# Development settings
from .dev import *
# Production settings.
from .prod import *
# Pick only one.
```

Depending on your environment, you might want to override `STATIC_ROOT` and `MEDIA_ROOT`, in your `__init__.py` file. Moreover, you have to create these locations.

Don't forget to edit `ALLOWED_HOSTS` for production or if your django server does not run on *localhost:8080*.

Common settings are available at `dissemin/settings/common.py`. Travis specific settings are available at `dissemin/settings/travis.py`.

2.2.6 Create the database structure

This is done by applying migrations:

```
python manage.py migrate
```

(this should be done every time the source code is updated). Then you can move on to *Deploying dissemin*.

2.2.7 Populate the database with test data

Dissemin comes with some sample data for development. You can use djangos *loaddata*:

```
django-admin loaddata
```

Note that this overwrites existing primary keys in your database.

2.2.8 Populate the search index

The search engine must be synchronized with the database manually using:

```
python manage.py update_index
```

That command should be run regularly to index new entries.

2.2.9 Social Authentication specific: Configuring sandbox ORCID

You are not forced to configure ORCID to work on Dissemin, just create a super user and use it!

Create an account on Sandbox ORCID.

Go to "Developer Tools", verify your mail using *Mailinator <mailinator.com>*.

Set up a redirection URI to be *localhost:8080* (supposed to be where your Dissemin instance server is running).

Take your client ID and your secret key, you'll use them later.

Ensure that in the settings, you have `BASE_DOMAIN` set up to `sandbox.orcid.org`.

Create a super user:

```
python manage.py createsuperuser
```

Browse to `localhost:8080/admin` and log in the administration interface. Go to "Social Application" and add a new one. Set the provider to `orcid.org`.

Here, you can use your app ID as your client ID and the secret key that you were given by ORCID earlier. You should also activate the default Site object for this provider.

Now, you can authenticate yourself using the ORCID sandbox!

2.2.10 Add deposit interfaces

If you want to enable deposit of papers to external repositories (such as Zenodo), you need to register them in the admin interface.

The page *localhost:8080/admin/deposit/repository/* lists the currently registered interfaces and allows you to add one.

To add a repository, you need the following settings:

- A name, description and logo. They will be shown to the user on the deposit page.
- A protocol: this is the internal name of the protocol Dissemin should use to perform the deposit. For now, only *ZenodoProtocol* is available: it can be used to deposit to Zenodo (both production and sandbox).
- Some other settings, such as the endpoint of the deposit interface, depending on what the protocol you have chosen requires. In the case of Zenodo, you need the endpoint (such as *https://zenodo.org/api/deposit/depositions* or *https://sandbox.zenodo.org/api/deposit/depositions*) and the API key (available from your account on Zenodo).

A checkbox allows you to enable or disable the repository without deleting its settings.

2.2.11 Installing or bypassing the tasks backend

Some features in Dissemin rely on an asynchronous tasks backend, celery. If you want to simplify your installation and ignore this asynchronous behaviour, you can add `CELERY_ALWAYS_EAGER = True` to your `dissemin/settings/__init__.py`. This way, all asynchronous tasks will be run from the main thread synchronously.

Otherwise, you need to run celery in a separate process. The rest of this subsection explains how.

The backend communicates with the frontend through a message passing infrastructure. We recommend redis for that (and the source code is configured for it). This serves also as a cache backend (to cache template fragments) and provides locks (to ensure that we do not fetch the publications of a given researcher twice, for instance).

First, install the redis server:

```
apt-get install redis-server
```

(this launches the redis server).:

To run the backend (still in the virtualenv):

```
celery --app=dissemin.celery:app worker -B -l INFO
```

The -B option starts the scheduler for periodic tasks, the -l option sets the debug level to INFO.

In production you want to run `celery` and `celerybeat` as a daemon and be controlled by `systemd`. `celery` and `celerybeat` are installed in the virtual environment of dissemin, so you have to take care, to use this environment. In particular you should use the same user for dissemin and celery.

You should use the following sample files that are similar to the official sample files. The main differences are a different `PYTHONPATH`, respect of the virtual environment and `stop` command for celerybeat. Put this into `/etc/default/celery` and change `CELERY_BIN` path.:

```
# See
# http://docs.celeryproject.org/en/latest/tutorials/daemonizing.html#available-options

CELERY_APP="dissemin.celery:app"
CELERYD_NODES="dissem"
CELERYD_OPTS=""
CELERY_BIN="/path/to/venv/env/bin/celery"
CELERYD_PID_FILE="/var/run/celery/%n.pid"
CELERYD_LOG_FILE="/var/log/celery/%n.log"
CELERYD_LOG_LEVEL="INFO"

CELERYBEAT_SCHEDULE_FILE="/var/run/celery/beat-schedule"
CELERYBEAT_PID_FILE="/var/run/celery/beat.pid"
CELERYBEAT_LOG_FILE="/var/log/celery/beat.log"
```

For `celeryd` systemd service put the following in `/etc/systemd/system/celery.service` and change `WorkingDirectory` to your dissemin root.:

```
[Unit]
Description=Celery Service
After=network.target

[Service]
Type=forking
User=dissemin
Group=dissemin
Restart=always
EnvironmentFile=-/etc/default/celery
WorkingDirectory=/path/to/dissemin/
ExecStart=/bin/sh -c '${CELERY_BIN} multi start ${CELERYD_NODES} -A ${CELERY_APP} --
↪pidfile=${CELERYD_PID_FILE} --logfile=${CELERYD_LOG_FILE} --loglevel=${CELERYD_LOG_
↪LEVEL} ${CELERYD_OPTS}'
ExecStop=/bin/sh -c '${CELERY_BIN} multi stopwait ${CELERYD_NODES} --pidfile=$
↪{CELERYD_PID_FILE}'
ExecReload=/bin/sh -c '${CELERY_BIN} multi restart ${CELERYD_NODES} -A ${CELERY_APP} -
↪-pidfile=${CELERYD_PID_FILE} --logfile=${CELERYD_LOG_FILE} --loglevel=${CELERYD_LOG_
↪LEVEL} ${CELERYD_OPTS}'

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

For `celeryd` systemd service put the following in `/etc/systemd/system/celerybeat.service` and change `WorkingDirectory` to your dissemin root.:

```
[Unit]
Description=Celerybeat Service
After=network.target

[Service]
Type=simple
User=dissemin
Group=dissemin
Restart=always
EnvironmentFile=-/etc/default/celery
WorkingDirectory=/path/to/dissemin/
ExecStart=/bin/sh -c 'PYTHONPATH=$(pwd) ${CELERY_BIN} beat -A ${CELERY_APP} --pidfile=
↪${CELERYBEAT_PID_FILE} --logfile=${CELERYBEAT_LOG_FILE} --loglevel=${CELERYD_LOG_
↪LEVEL} -s ${CELERYBEAT_SCHEDULE_FILE}'
ExecStop=/bin/kill -s TERM $MAINPID

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

Note that we use `/Bin/sh -c` to process the `PYTHONPATH` and `${CELERY_BIN}`.

To make systemd create the necessary directories with permissions put the follwing into `/etc/tmpfiles.d/celery.conf`:

```
d /var/run/celery 0755 dissemin dissemin
d /var/log/celery 0755 dissemin dissemin
```

After that run `systemctl daemon-reload` to reload systemd service files and you are ready to use `celery` and `celerybeat` with systemd by calling:

```
systemctl start celery.service
systemctl start celerybeat.service
```

To make them start on boot call:

```
systemctl enable celery.service
systemctl enable celerybeat.service
```

2.2.12 Importing papers

When running a test instance on Dissemin on your local machine, the database should be preconfigured to contain some papers. However, if you would like to test different papers, you can easily import more papers in the database of the test instance by visiting `localhost:8080/DOI` where `DOI` is the DOI of the paper that you would like to create.

CHAPTER 3

Getting API keys

Dissemin relies on various interfaces to fetch its metadata. Some of them require to register for an API key, that dissemin reads in `dissemin/settings.py`, the main configuration file.

Here is how to register for these interfaces.

3.1 SHERPA/RoMEO

SHERPA/RoMEO gives a machine-readable to publishers' self-archiving policies.

The API key is not required but encoraged as unauthenticated users can perform a limited number of queries daily.

To get an API key, visit this page. The key should then be written in `dissemin/settings/secrets.py`, as `ROMEO_API_KEY`.

3.2 Zenodo

Zenodo is a repository hosted by CERN, storing publications as well as research data. Dissemin uses it to upload papers on behalf of users.

To use Zenodo, you need an account. You can then generate an auth token from their web interface. Then, set up the repository via the Dissemin admin interface (available at /admin).

3.3 Proaixy

Proaixy is an OAI-PMH proxy where disemin discovers preprints. For now, no API key is required to use this service.

CHAPTER 4

Deploying dissemin

You have two options to run the web server: development or production settings.

4.1 Development settings

Simply run `./launch.sh`. This uses the default Django server (unsuitable for production) and serves the website locally on the port 8080. Note that the standard port for django-admins runserver-command is _8000_, but this ensures compatibility with the Vagrant installation.

This runs with `DEBUG = True`, which means that Django will report to the user any internal error in a transparent way. This is useful to debug your installation but should not be used for production as it exposes your internal settings.

4.2 Production settings

As any Django website, Dissemin can be served by various web servers. These settings are not specific to dissemin itself so you should refer to the relevant Django documentation.

No matter what web server you use, you need to run `python manage.py collectstatic` to copy the static files from the git repository to the desired location for your installation (in the example below, `/home/dissemin/www/static`), as well as `python manage.py compilemessages` to compile the translation files.

Make sure that your *media/* directory is writable by the user under which the application will run (*www-data* on Debian).

4.3 Self-hosting MathJax

Dissemin requires MathJax for rendering LaTeX formatting in the abstracts. Out of the box, Dissemin will use a CDN-hosted version of MathJax. This has the downside of preventing deposit when disabling third-party JS.

An easy solution to this is to self-host MathJax. You can follow the installation instructions from MathJax to get a local copy. Ideally, you should put it in the static directory (under `/home/dissemin/www/static/` in the example below).

Note that MathJax consists of many small files which can slow down a lot the built-in Django webserver. Hence, it is better to serve it directly by Apache and avoid having all these files in the `papers/static/libs` directory of Dissemin.

Once MathJax is downloaded and available by your webserver, you can use the setting `MATHJAX_SELFHOST_URL` (in `dissemin/settings`) to specify a location to load MathJax from. In the example below, this would be `//dissemin.myuni.edu/static/mathjax/MathJax.js?config=TeX-AMS-MML_HTMLorMML`.

4.4 Apache with WSGI

Here is a sample VirtualHost, assuming that the root of the Dissemin source code is at `/home/dissemin` and you use `python3.6`.:

```
<VirtualHost *:80>
        ServerAdmin webmaster@localhost
        ServerName dissemin.myuni.edu

        ### STATIC FILES ###

        # Instructions for robots
        Alias /robots.txt /home/dissemin/www/static/robots.txt
        <Location /robots.txt>
        Require all granted
        </Location>
        # Thumbnails of PDF files uploaded by users
        Alias /media/thumbnails/ /home/dissemin/media/thumbnails/
        <Directory /home/dissemin/media/thumbnails>
        Require all granted
        </Directory>
        # Logos of the repositories configured on your instance
        Alias /media/repository_logos/ /home/dissemin/media/repository_logos/
        <Directory /home/dissemin/media/repository_logos>
        Require all granted
        </Directory>
        # Generic static files (CSS, JS, images)
        Alias /static/ /home/dissemin/www/static/
        <Directory /home/dissemin/www/static>
        Require all granted
        </Directory>

        ### WSGI connection ###

        # Path to the WSGI application for the website
        WSGIScriptAlias / /home/dissemin/dissemin/wsgi.py
        # Python path for the application
        WSGIDaemonProcess dissemin.myuni.edu python-path=/home/dissemin:/home/
↪dissemin/.virtualenv/lib/python3.6/site-packages

        WSGIProcessGroup dissemin.myuni.edu

        <Directory /home/dissemin/dissemin>
        <Files wsgi.py>
```

(continues on next page)

(continued from previous page)

```
        Require all granted
        </Files>
        </Directory>

        ### Error handling ###
        ErrorDocument 500 /500-error
        ErrorDocument 404 /404-error

        ### Log settings ###
        ErrorLog ${APACHE_LOG_DIR}/django-dissemin-myuni.log

        # Possible values include: debug, info, notice, warn, error, crit,
        # alert, emerg.
        LogLevel debug

        CustomLog ${APACHE_LOG_DIR}/access-dissemin-myuni.log combined
</VirtualHost>
```

You should only have to change the path to the application and the domain name of the service.

4.5 lighttpd with FastCGI (deprecated)

We describe here how to set up the server with lighttpd, a lightweight web server, with FastCGI. This has been deprecated by Django, as support for FastCGI will be discontinued: use WSGI instead.

Add this to your lighttpd config:

```
$HTTP["host"] =~ "^myhostname.com$" {
    accesslog.filename   = "/var/log/lighttpd/dissemin-$INSTANCE.log"
    server.document-root = "$SOURCE_PATH/www/"
    $HTTP["url"] =~ "^(?!((/static/)|(/robots\.txt)))" {
        fastcgi.server = (
            "/" => (
                "/" => (
                    "socket" => "/tmp/django-dissemin-$INSTANCE.sock",
                    "check-local" => "disable",
                    "fix-root-scriptname" => "enable",
                )
            ),
        )
    }
    alias.url = (
        "/static/" => "$SOURCE_PATH/www/static/",
        "/robots.txt" => "$SOURCE_PATH/www/static/robots.txt",
    )
}
```

where `$INSTANCE` is the name of your instance and `$SOURCE_PATH` is the path to the root of the git repository of dissemin.

You can create the `.sock` file with `touch /tmp/django-dissemin-$INSTANCE.sock`.

CHAPTER 5

Data sources

Dissemin works with various data sources, providing bibliographic references, full texts and publisher's policies. Most of these sources only provide a search API to expose their data: we do not store a copy of their database but perform calls to their external API when required. This has the advantage of keeping the local data storage needs very modest, but fetching the appropriate data from the APIs takes some time.

5.1 CrossRef

CrossRef is an association of publishers, mainly in charge of issuing Digital Object Identifiers (DOIs) for academic publications. DOIs provide many useful features:

- Redirection to the publication's page on the publisher's website, with links of the form http://dx.doi.org/10.1103/physreve.89.033013. When a publisher changes the structure of its website, tells CrossRef where the resources have moved, updating the DOI proxy so that users are redirected to the new location.
- Associating metadata to DOIs, in a uniform format. The metadata associated with a given DOI can be retrieved using content negociation. This is useful to get the metadata associated with a DOI that we discover from other metadata sources. It works as follows:

```
$ curl -LH "Accept: application/citeproc+json" http://dx.doi.org/10.1103/physreve.
↪89.033013
{
    "indexed":
    {
        "date-parts": [[2015,6,10]],
        "timestamp": 1433897719282
    },
    "reference-count": 36,
    "publisher": "American Physical Society (APS)",
    "issue": "3",
    "license":[
```

(continues on next page)

(continued from previous page)


```
        {
            "content-version": "vor",
            "delay-in-days": 13,
            "start": {
                "date-parts": [[2014,3,14]],
                "timestamp": 1394755200000
            },
            "URL": "http://link.aps.org/licenses/aps-default-license"
        }

    ],
    "DOI": "10.1103/physreve.89.033013",
    "type": "journal-article",
    "source": "CrossRef",
    "title": "Small-scale anisotropic intermittency in magnetohydrodynamic␣
↪turbulence at low magnetic Reynolds numbers",
    "prefix": "http://id.crossref.org/prefix/10.1103",
    "volume": "89",
    "author": [
        {
            "affiliation": [ ],
            "family": "Okamoto",
            "given": "Naoya"
        },
        {
            "affiliation": [ ],
            "family": "Yoshimatsu",
            "given": "Katsunori"
        },
        {
            "affiliation": [ ],
            "family": "Schneider",
            "given": "Kai"
        },
        {
            "affiliation": [ ],
            "family": "Farge",
            "given": "Marie"
        }

    ],
    "member": "http://id.crossref.org/member/16",
    "container-title": "Physical Review E",
    "link": [
        {
            "intended-application": "syndication",
            "content-version": "vor",
            "content-type": "unspecified",
            "URL": "http://link.aps.org/article/10.1103/PhysRevE.89.033013"
        }
    ],
    "deposited":
    {
        "date-parts":[[2015,4,13]],
        "timestamp": 1428883200000
    },
    "score": 1,
```


(continues on next page)

(continued from previous page)


```
    "subtitle": [ ],
    "issued":
    {
        "date-parts":[[2014,3]]
    },
    "URL": "http://dx.doi.org/10.1103/physreve.89.033013",
    "ISSN":
    [
        "1539-3755",
        "1550-2376"
    ],
    "subject":
        [
            "Condensed Matter Physics",
            "Statistical and Nonlinear Physics",
            "Statistics and Probability"
        ]
}
```


- A search API, basically a machine-readable version of CrossRef Metadata Search. Similar metadata is returned for each search result. The documentation can be found here. By searching for an researcher's name and browsing through the few first results pages, we get the metadata for most papers written by that researcher and registered at CrossRef. When a researcher is associated with an ORCID id, we also search for papers using their id. This can return papers that do not appear in the ORCID profile (CrossRef has introduced an auto-update feature in 2015 to populate automatically the profiles, but it is an opt-in feature and only applies to subsequent papers). This service only returns DOIs issued by CrossRef, the two other services also work for other DOI registration agencies such as DataCite or MEDRA. DataCite offers a similar service but we do not use it as they cover mostly data.

5.2 SHERPA/RoMEO

SHERPA/RoMEO is a service run by JISC which provides a semi-structured representation of publisher's self-archiving policies. They offer an API, whose functionality is very similar to the search service they offer to their regular users. You can search for a policy by journal or by publisher. Since some publishers have multiple archiving policies, RoMEO recommends to search by journal because it ensures that you will get the policy in place for this specific journal. The `publishers` app replicates most of RoMEO's data model, by defining `Journal` and `Publisher` models with fields taken from SHERPA. We synchronize our model with SHERPA's data every two weeks using their dumps.

For many journal articles and all conference papers, RoMEO knows the publisher but not the journal, and the metadata returned by CrossRef contains both the journal (or the proceedings title) and the publisher. We use therefore a two-step approach:

- We search for the journal: if it succeeds, we assign the journal and the corresponding policy to the paper.
- If it fails, we search for the a default policy from the publisher. Default policies are those which have a null `romeo_parent_id`.

Because the publisher names are not always the same in Crossref and SHERPA, we add heuristics to disambiguate publishers. We use the papers for which a corresponding journal was found in SHERPA and collect their publisher names as seen in Crossref. If we see that a given Crossref publisher name is overwhelmingly associated to a given SHERPA Publisher which is a default publisher policy (`romeo_parent_id` is null), then we also link Crossref papers with this publisher name but no matching journal to this SHERPA Publisher.

The task to update SHEPA policies is `fetch_updates_from_romeo`, from `publishers.tasks`. The first

time it is run, it fetches the entire dump. The next time, it will only fetch the publishers which were updated since the last update. To force a full ingestion, nullify all the update dates on the RoMEO publishers with `Publisher.objects.update(last_updated=None)`, and then run `fetch_updates_from_romeo`.

5.3 ORCID

ORCID has a public API that can be used to fetch the metadata of all papers ("works") made visible of any ORCID profile (unfortunately, very often, the profiles are empty). ORCID does not enforce any strict metadata format, which makes it hard to import papers in Dissemin. Specifically, works do not always have a list of authors (which is a shame given that this service is supposed to solve ambiguity of author names). Even worse, when an authors list is provided, the owner of the ORCID record is almost never identified in this list.

We try to make the most of the available metadata:

- If a DOI is present, fetch the metadata using content negociation ;
- If a Bibtex version of the metadata is available, parse the Bibtex record to extract the title and author names ;
- Otherwise, if no authors are given, skip the paper.

We then try to find which author is the owner of the ORCID record, using a dedicated name-matching heuristic (`papers.name.shallower_name_similarity()`). The name that matches the most the reference name of the ORCID record is assumed to refer to the record's owner.

5.4 Proaixy

Proaixy is our own OAI-PMH proxy. We use it to discover preprints. It harvests papers from various OAI-PMH sources (notably the sources handled by BASE, see complete list) and re-exposes the result in OAI-PMH, adding a few functionalities.

- Search by fingerprint: each paper in Dissemin has a fingerprint, a robust representation of the title, and sometimes the year of publication or the last names of the authors. Proaixy enables to fetch all records that match a given fingerprint.
- Search by author name or name signature: a similar feature to search for papers matching a given name.

5.5 BASE and CORE

Interfaces for the search APIs of BASE and CORE have been implemented but are not used anymore. They basically search for a researcher's name and go through the first few pages of results.

CHAPTER 6

The data model in Dissemin

This section explains how metadata is represented in Dissemin. There are two important models: `OaiRecord` and `Paper` (both defined in `papers/models.py`).

The `OaiRecord` model represents an occurence of a paper in some external repository (from the publisher or from an open repository). Each `OaiRecord` has at least a `splash_url` (the URL of the landing page of the paper in the repository) and sometimes a `pdf_url`. The `pdf_url` is present if and only if we think that the full text is available from this repository. This `pdf_url` should ideally be a direct link to the full text, but often it is actually equal to the `splash_url` (but its presence still indicates that the full text is available somehow).

These records are grouped into `Paper` objects (via a foreign key from `OaiRecord` to `Paper`). This deduplication process is done by two criteria:

- first, `OaiRecords` with the same DOI are merged into the same paper.
- second, we compute a fingerprint of the `OaiRecord` metadata, which consists of the title, author last names and publication year. Any two `OaiRecords` with identical fingerprints are also merged into the same `Paper`.

Dissemin *harvests four metadata sources*: ORCID, Crossref, BASE and Unpaywall (oadoi). Each of these implements the `PaperSource` interface, which provides mechanisms to push the papers to the database. The responsibility of each `PaperSource` is to provide `BarePaper` instances, which are Python objects representing papers which have not been saved to the database yet (and therefore not deduplicated). When doing this, each `PaperSource` determines from the medatada they have access to whether `pdf_url` should be filled or not (depending on whether we think the metadata indicates full text availability).

CHAPTER 7

Contributing to dissemin

This section explains how to do some development tasks on the source code.

7.1 Setting up Dissemin for development in an IDE

This page lists some possible ways to set up Dissemin locally for development, including setting up an IDE to edit Dissemin conveniently. First, you need to install Dissemin locally: see *Installation* for that. In particular, you will need to have postgres, redis and elasticsearch instanced running during development, as these services are required to run the tests.

7.1.1 Eclipse and PyDev

Although it is primarily designed to work on Java programs, Eclipse can be used to work on Python projects thanks to its PyDev extension. This includes a Python editor, debugger and importantly, the ability to run tests selectively from the editor. Moreover PyDev comes with a Django integration too.

To install Eclipse, simply download it and unzip it in your favourite location. Fire up Eclipse and click *Help*, *Eclipse Marketplace*. In the field to search for new software, type *PyDev* and install it.

You will then need point Eclipse to your copy of Dissemin, so that it can be opened as a project. To do so, click *File*, *Open Projects from File System*, and select the directory where you have cloned Dissemin. Click *Finish*: this will create your project. The project might not be recognized as a Python project, so enable PyDev on it with a right click on the project (in the Project Explorer), click *PyDev*, *Set as PyDev Project*. Do the same to enable it as a Django project too.

Assuming you have installed Dissemin's Python dependencies in a Virtualenv, you will need to configure that too (otherwise Eclipse will try to run Dissemin with the system's own Python installation). To do so, right click on the project, select *Properties* and go to the *PyDev - Interpreter/Grammar* pane. Then click the *Click here to configure an interpreter not listed*, choose *Open interpreter preference pages*. Then *Browse for python/pypy exe*, and select the Python executable in your virtualenv (it normally lives at *my_virtualenv/bin/python*). Give it a name such as *Dissemin virtualenv*. When prompted to add entries to PYTHONPATH, select them all and validate. Finally, click *Apply and*

close. Once you are back to the project's own interpreter configuration page, do not forget to select the newly-created interpreter configuration.

Finally, you will need to configure PyDev so that it uses *pytest* to run the tests. This will have the benefit of handling Django's initialization for you. This setting is not stored at project level, you need to go in PyDev's general preferences to change this. Click *Window*, *Preferences*, select the *PyUnit* pane in the *PyDev* group and choose the *Py.test runner*. Finally, click *Apply and Close*.

You can now easily run individual tests from the editor. Go to a test file, use *Ctrl-Shift* and the up and down arrows to navigate to the test method or test class that you want to run. Then press *Ctrl-F9* and validate with *Enter*. This will run your tests and display their results in the dedicated pane below. You can also set up breakpoints and run the tests in the debugger.

7.1.2 PyCharm

PyCharm's Django integration is not available in the Community edition. However, because we use Pytest to run tests, it might be possible to use the Community edition anyway. If you try, please let us know how it goes so that we can update this documentation.

7.1.3 Using a standard text editor

That works too, of course. In that case you might want to make sure you run *./pyflakes.sh* to check for import errors and other syntactical issues before you commit or push. This can easily done by adding a git hook:

```
ln -s pyflakes.sh .git/hooks/pre-commit
```

7.2 Localization

Translations are hosted at TranslateWiki, for an easy-to-use interface for translations and statistics.

We use Django's standard localization system, based on i18n. This lets us translate strings in various places:

- in Python code, use `_("some translatable text")`, where `_` is imported by `from django.utils.translation import ugettext_lazy as _`
- in Javascript code, use `gettext("some translatable text")`
- in HTML templates, use either `{% trans "some translatable text" %}` for inline translations, or, for longer blocks of text:

  ```
  {% blocktrans trimmed %}
  My longer translatable text.
  {% endblocktrans %}
  ```

  The `trimmed` argument is important as it ensures leading and trailing whitespace and newlines are not included in the translation strings (they mess up the translation system).

To generate the PO files, run:

```
python manage.py makemessages --keep-pot --ignore doc
```

This will generate a PO template *locale/django.pot* that can be used to update the translated files for each language, such as `locale/fr/LC_MESSAGES/django.po`. Note that in some circumstances Django can generate new translation files for languages not yet covered. In this case these new files should be deleted, as they will break Translatewiki. It is also necessary to generate separate PO files for JavaScript translations:

```
python manage.py makemessages -d djangojs --keep-pot --ignore doc
```

You can then compile all the PO files into MO files so that they can be displayed on the website:

```
python manage.py compilemessages
```

That's it! The translations should show up in your browser if it indicates the target language as your preferred one.

7.3 Available Languages

You can change the set of available languages for your installation in `dissemin/settings/common.py` by changing the `LANGUAGES` list, e.g. by commenting or uncommenting the corresponding lines.

7.4 Writing an interface for a new repository

Writing an interface for a new repository is very easy! Here is a quick tutorial, whose only requirements are some familiarity with Python, and have a running instance of Dissemin.

First, check that the repository you want to create an interface for has an API that allows that.

In this tutorial we will write an interface for a DSpace repository, using the DSpace Demo as a test instance. This repository supports the SWORD protocol, so let's write a SWORD adapter for Dissemin.

7.4.1 Implementing the protocol

Protocol implementations are stored as submodules of the `deposit` module. We start by creating a *sword* subdirectory within *deposit*, where we will write all our Python code. Create an empty *__init__.py* file in our *deposit/sword* directory to make sure it is a valid Python module.

To tell Dissemin how to interact with DSpace, we need to write an implementation of that protocol. This has to be done in a dedicated file, *deposit/sword/protocol.py*, by creating a subclass of `RepositoryProtocol`:

```
# this imports generic tools for our protocol implementation
from deposit import protocol
# this imports django's localization tools
# it should sometimes be replaced by ugettext_lazy
from django.utils.translation import ugettext as _

class SwordProtocol(protocol.RepositoryProtocol):

    def submit_deposit(self, pdf, form):
        result = DepositResult()

        ########################################
        # Your paper-depositing code goes here! #
        ########################################


        # If the deposit succeeds, our deposit function should return
        # a DepositResult, with the following fields filled correctly:

        # A unique id provided by the repository (useful to modify
```

(continues on next page)

(continued from previous page)

```
        # the deposit afterwards). The form is free, it only has to be a string.
        result.identifier = 'myrepo/deposit/12345'

        # The URL of the page of the paper on the repository,
        # and the URL of the full text
        result.splash_url = 'http://arxiv.org/abs/0809.3425'
        result.pdf_url = 'http://arxiv.org/pdf/0809.3425'

        return result
```

Let us see how we can access the data provided by Dissemin to perform the upload. The paper to be deposited is available in *self.paper*, as a `Paper` instance. This gives you access to all we know about the paper: title, authors, sources, bibliographic information, identifiers, publisher's policy, and so on. You can either access it directly from the attributes of the paper, for instance with *self.paper.title*, or use the JSON representation that we generate for the API, which can be generated using *self.paper.json()*. For instance, *self.paper.json()['title']* gives you the title.

The PDF file is passed as an argument to the *submit_deposit* method. It is a path to the PDF file, which you can open with *open(pdf, 'r')* for instance.

You also have access to the settings for the target repository, as a `Repository` object, in *self.repository*. This should give you all the information you need about how to connect to the repository: *self.repository.endpoint*, *self.repository.username*, *self.repository.password*, and so on (see the documentation of `Repository` for more details).

If the deposit fails for any reason, you should raise `protocol.DepositError` with an helpful error message, like this:

```
raise protocol.DepositError(_('The repository refused your paper.'))
```

Note that we localize the error (with the underscore function).

It is generally a good idea to log messages to keep track of how the deposit does. You can use the embedded logger, so that your log messages will be saved by Dissemin in the relevant DepositRecords, like this:

```
self.log("Do not forget to log the responses you get from the server.")
```

7.4.2 Testing the protocol

So now, how do you test this protocol implementation? Instead of testing it manually by yourself, you are encouraged to take advantage of the testing framework available in Dissemin. You will write test cases, that check the behaviour of your implementation for particular PDF files and paper metadata.

To do so, we will create a file at *deposit/sword/tests.py* with the following code:

```
from deposit.tests import ProtocolTest, lorem_ipsum
# lorem_ipsum contains a sample abstract you can reuse in your test case

class SwordProtocolTest(ProtocolTest):
    @classmethod
    def setUpClass(self):
        super(SwordProtocolTest, self).setUpClass()

        # Fill here the details of your test repository
        self.repo.username = 'dspacedemo+submit@gmail.com'
        self.repo.password = 'dspace'
```

(continues on next page)

(continued from previous page)

```
        self.repo.endpoint = 'http://demo.dspace.org/swordv2/servicedocument'

        # Now we set up the protocol for the tests
        self.proto = SwordProtocol(self.repo)

        # Fill here the details of the metadata form for your repository
        data = {'onbehalfof':'dspacedemo+colladmin@gmail.com'}
        self.form = self.proto.get_bound_form(data)
        self.form.is_valid() # this validates our sample data
```

So, once you have done that, you migth think that you have not written any test. In fact, as your test case subclasses `ProtocolTest`, it inherits various test cases, including one that will try to submit a PDF to the repository you have defined, with the contents of the form as above.

To try it out, run the following command at the root of your Dissemin instance:

```
python manage.py test deposit.sword.tests
```

It is a very good idea to add more test cases, for instance by creating multiple subclasses of `ProtocolTest` as above, or by adding other tests methods to the same subclass (they have to

7.4.3 Using the protocol

So now you have your shiny new protocol implementation and you want to use it.

First, we need to register the protocol in Dissemin. To do so, add the following lines at the end of *deposit/sword/protocol.py*:

```
from deposit.registry import *
protocol_registry.register(SwordProtocol)
```

Next, add your protocol to the enabled apps, by adding *deposit.sword* in the *INSTALLED_APPS* list of *dissemin/settings/common.py*:

```
...
'deposit',
'deposit.zenodo',
'deposit.sword',
...
```

Now the final step is to configure a repository using that protocol. Launch Dissemin, go to Django's web admin, click *Repositories* and add a new repository, filling in all the configuration details of that repository. The *Protocol* field should be filled by the name of your protocol, *SwordProtocol* in our case.

Now, when you go to a paper page and try to deposit it, your repository should show up, and if everything went well you should be able to deposit papers.

Each deposit (successful or not) creates a `DepositRecord` object that you can see from the web admin interface. If you have used the provided log function, the logs of your deposits are available there.

To debug the protocol directly from the site, you can enable Django's *settings.DEBUG* (in *dissemin/settings.py*) so that exceptions raised by your code are popped up to the user.

7.4.4 Adding extra metadata with forms

What if the repository you submit to requires additional metadata, that Dissemin does not always provide? We need to add a field in the deposit form to let the user fill this gap.

Fortunately, Django has a very convenient interface to deal with forms, so it should be quite straightforward to add the fields you need.

Let's say that the repository we want to deposit into takes two additional pieces of information: the topic of the paper (in a set of predefined categories) and an optional comment for the moderators.

All we need to do is to define a form with these two fields:

```
# import the forms API
from django import forms
# import localization
from django.utils.translation import ugettext_lazy as _

# First, we define the possible topics for a submission
MYREPO_TOPIC_CHOICES = [
    ('quantum epistemology',_('Quantum Epistemology')),
    ('neural petrochemistry',_('Neural Petrochemistry')),
    ('ethnography of predicative turbulence',_('Ethnography of Predicative Turbulence
↪')),
    ('other',_('Other')),
    ]

# Then, we define our metadata form
class MyRepoForm(forms.Form):

    # Fields are declared as class arguments
    topic = forms.ChoiceField(
        label=_('Topic'), # the label that will be displayed on the field
        choices=MYREPO_TOPIC_CHOICES, # the possible choices for the user
        required=True, # is this field mandatory?
        # other arguments are possible, see https://docs.djangoproject.com/en/2.2/ref/
↪forms/fields/
        )

    comment = forms.CharField(
         label=_('Comment for the moderators'),
         required=False)
```

Then, we need to bind this form to our protocol. TODO

7.5 FAQ for contributing to Dissemin

Here are some frequently asked questions and tips for getting started to work and contribute to Dissemin. The best idea to start hacking on Dissemin is probably to use the VM (Vagrant method from *Installation*).

7.5.1 Fetching a specific paper by DOI

The Dissemin VM is quite empty by default. If you want to inspect particular paper, it is possible to fetch it by DOI by visiting `http://localhost:8080/<DOI>`.

7.5.2 Fetching a specific ORCID profile

The Dissemin VM uses the sandbox ORCID API out of the box. Therefore, you cannot fetch a specific profile from ORCID. Here is how to locally fetch a specific profile from ORCID, in order to reproduce and debug bugs in fetching the list of papers for instance.

First, edit the `dissemin/settings/__init__.py` file to set the ORCID API to use to the real ORCID API, putting the line `ORCID_BASE_DOMAIN = 'orcid.org'`.

Then, you should find the ORCID ID you want to fetch locally. You can use the official instance, https://dissem.in/, to search for a given author and get the ORCID ID.

Finally, restart both the Django process as well as the Celery process in the VM and head to `http://localhost:8080/<ORCID_ID>`, replacing `<ORCID_ID>` by the full ORCID ID.

CHAPTER 8

Testing dissemin

Dissemin's test suite is run using `pytest` rather than using Django's `./manage.py test`. Pytest offers many additional features compared to Python's standard `unittest` which is used in Django. To run the test suite, you need to install pytest and other packages, mentioned in `requirements-dev.txt`.

The test suite is configured in `pytest.ini`, which determines which files are scanned for tests, and where Django's settings are located.

Some tests rely on remote services. Some of them require API keys, they will fetch them from the following environment variables (or be skipped if these environment variables are not defined): * `ROMEO_API_KEY` * `ZENODO_SANDBOX_API_KEY` required for tests of the Zenodo interface. This can be obtained

by creating an account on sandbox.zenodo.org and creating a "Personal Access Token" from there.

CHAPTER 9

Building the docs

This documentation is generated by sphinx, both from the source code and with some additional documentation files. To build it, you need a working dissemin install where you have installed the packages in `requirements-dev.txt`.

There are two steps to generate the docs: first, auto-generate the reStructuredText sources of the docs with sphinx-apidoc:

```
# first, make sure you are in an environment where
# the requirements are available
source .virtualenv/bin/activate
# then, invoke sphinx-apidoc via make
make -B doc
```

Then, compile these RST sources to HTML:

```
cd doc/sphinx ; make html
```

The HTML output is then available in `doc/sphinx/_build/html/`.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search