Patchwork Documentation

Release 1.0.0

Stephen Finucane, Jeremy Kerr, Damien Lespiau

Sep 04, 2017

Contents

Contents:

CHAPTER 1

patchwork

patchwork is a patch tracking system for community-based projects. It is intended to make the patch management process easier for both the project's contributors and maintainers, leaving time for the more important (and more interesting) stuff.

Patches that have been sent to a mailing list are 'caught' by the system, and appear on a web page. Any comments posted that reference the patch are appended to the patch page too. The project's maintainer can then scan through the list of patches, marking each with a certain state, such as Accepted, Rejected or Under Review. Old patches can be sent to the archive or deleted.

Currently, patchwork is being used for a number of open-source projects, mostly subsystems of the Linux kernel. Although Patchwork has been developed with the kernel workflow in mind, the aim is to be flexible enough to suit the majority of community projects.

Download

The latest version of Patchwork is available with git. To download:

```
$ git clone https://github.com/dlespiau/patchwork.git
```

Patchwork is distributed under the GNU General Public License.

Design

patchwork should supplement mailing lists, not replace them

Patchwork isn't intended to replace a community mailing list; that's why you can't comment on a patch in patchwork. If this were the case, then there would be two forums of discussion on patches, which fragments the patch review process. Developers who don't use patchwork would get left out of the discussion.

However, a future development item for patchwork is to facilitate on-list commenting, by providing a "send a reply to the list" feature for logged-in users.

Don't pollute the project's changelogs with patchwork poop

A project's changelogs are valuable - we don't want to add patchwork-specific metadata.

patchwork users shouldn't require a specific version control system

Not everyone uses git for kernel development, and not everyone uses git for patchwork-tracked projects.

It's still possible to hook other programs into patchwork, using the pwclient command-line client for patchwork, or directly to the XML RPC interface.

Getting Started

You should check out the *Deploying Patchwork* and *Developing patchwork* guides for information on how to get to work with patchwork.

Support

For questions and contributions, please use the GitHub project.

CHAPTER 2

Deploying Patchwork

Patchwork uses the Django framework - there is some background on deploying Django applications here:

```
http://www.djangobook.com/en/2.0/chapter12/
```

You'll need the following (applications used for patchwork development are in brackets):

- A Python interpreter
- Django >= 1.7. The latest version is recommended
- A web server and suitable WSGI plugin. Options include Apache with the mod_python plugin, or Gunicorn with nginx as the proxy server
- A database server (PostgreSQL, MySQL)
- Relevant Python modules for the database server (see the various requirements.txt files)

Database Configuration

Django's ORM support multiple database backends, though the majority of testing has been carried out with PostgreSQL and MySQL.

We need to create a database for the system, add accounts for two system users: the web user (the user that your web server runs as) and the mail user (the user that your mail server runs as). On Ubuntu these are `www-data` and `nobody`, respectively.

As an alternative, you can use password-based login and a single database account. This is described further down.

NOTE: For the following commands, a `$` prefix signifies that the command should be entered at your shell prompt, and a > prefix signifies the command-line client for your SQL server (`psql` or `mysql`).

Install Packages

If you don't already have MySQL installed, you'll need to do so now. For example, to install MySQL on RHEL:

```
$ sudo yum install mysql-server
```

Create Required Databases and Users

PostgreSQL (ident-based)

PostgreSQL support ident-based authentication, which uses the standard UNIX authentication method as a backend. This means no database-specific passwords need to be set/used. Assuming you are using this form of authentication, you just need to create the relevant UNIX users and database:

```
$ createdb patchwork
$ createuser www-data
$ createuser nobody
```

PostgreSQL (password-based)

If you are not using the ident-based authentication, you will need to create both a new database and a new database user:

```
$ createuser -PE patchwork
$ createdb -O patchwork patchwork
```

MySQL

```
$ mysql
> CREATE DATABASE patchwork CHARACTER SET utf8;
> CREATE USER 'www-data'@'localhost' IDENTIFIED BY '<password>';
> CREATE USER 'nobody'@'localhost' IDENTIFIED BY '<password>';
```

Configure Settings

Once that is done, you need to tell Django about the new database settings, by defining your own `production.py` settings file (see below). For PostgreSQL:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'HOST': 'localhost',
        'PORT': '',
        'USER': 'patchwork',
        'PASSWORD': 'my_secret_password',
        'NAME': 'patchwork',
        'TEST_CHARSET': 'utf8',
    },
}
```

If you're using MySQL, only the `ENGINE` changes:

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

NOTE: `TEST/CHARSET` is used when creating tables for the test suite. Without it, tests checking for the correct handling of non-ASCII characters fail.

Django Setup

Configure Directories

Set up some initial directories in the patchwork base directory:

```
mkdir -p lib/packages lib/python
```

`lib/packages` is for stuff we'll download, `lib/python` is to add to our Python path. We'll symlink Python modules into `lib/python`.

At the time of release, patchwork depends on Django version 1.7 or later. Where possible, try to use the latest stable version (currently 1.8). Your distro probably provides this. If not, install it manually:

```
cd lib/packages
git clone https://github.com/django/django.git -b stable/1.8.x
cd ../python
ln -s ../packages/django/django ./django
```

Configure Settings

You will also need to configure a settings file for Django. A [sample settings file] is provided, which defines default settings for patchwork. You'll need to configure settings for your own setup and save this as `production.py` (or override the `DJANGO_SETTINGS_MODULE` environment variable).

```
cp patchwork/settings/production.example.py \
  patchwork/settings/production.py
```

At the very minimum, the following settings need to be configured:

```
SECRET_KEY
ADMINS
TIME_ZONE
LANGUAGE_CODE
DEFAULT_FROM_EMAIL
NOTIFICATION_FROM_EMAIL
```

You can generate the `SECRET_KEY` with the following python code:

```
import string, random
chars = string.letters + string.digits + string.punctuation
print repr("".join([random.choice(chars) for i in range(0,50)]))
```

If you wish to enable the XML-RPC interface, add the following to the file:

```
ENABLE_XMLRPC = True
```

Configure Database Tables

Then, get patchwork to create its tables in your configured database:

```
PYTHONPATH=../lib/python ./manage.py migrate
```

Add privileges for your mail and web users. This is only needed if you use the ident-based approach. If you use password-based database authentication, you can skip this step.

For Postgresql:

```
psql -f lib/sql/grant-all.postgres.sql patchwork
```

For MySQL:

```
mysql patchwork < lib/sql/grant-all.mysql.sql
```

Other Tasks

You will need to collect the static content into one location from which it can be served (by Apache or nginx, for example):

```
PYTHONPATH=lib/python ./manage.py collectstatic
```

You'll also need to load the initial tags, states and actions into the patchwork database:

```
PYTHONPATH=lib/python ./manage.py loaddata default_tags default_states default_events
```

Apache Setup

Example Apache configuration files are in `lib/apache2/`.

WSGI

django has built-in support for WSGI, which supersedes the fastcgi handler. It is thus the preferred method to run patchwork.

The necessary configuration for Apache2 may be found in:

```
lib/apache2/patchwork.wsgi.conf.
```

You will need to install/enable mod_wsgi for this to work:

```
a2enmod wsgi
apache2ctl restart
```

Configure patchwork

Now, you should be able to administer patchwork, by visiting the URL:

```
http://your-host/admin/
```

You'll probably want to do the following:

- Set up your projects
- Configure your website address (in the Sites section of the admin)

Subscribe a Local Address to the Mailing List

You will need an email address for patchwork to receive email on - for example - `patchwork@your-host`, and this address will need to be subscribed to the list. Depending on the mailing list, you will probably need to confirm the subscription - temporarily direct the alias to yourself to do this.

Setup your MTA to Deliver Mail to the parsemail Script

Your MTA will need to deliver mail to the parsemail script in the email/directory. (Note, do not use the `parsemail.py` script directly). Something like this in /etc/aliases is suitable for postfix:

```
patchwork: "|/srv/patchwork/patchwork/bin/parsemail.sh"
```

You may need to customise the `parsemail.sh` script if you haven't installed patchwork in `/srv/patchwork`.

Test that you can deliver a patch to this script:

```
sudo -u nobody /srv/patchwork/patchwork/bin/parsemail.sh < mail
```

Set up the patchwork cron script

Patchwork uses a cron script to clean up expired registrations, and send notifications of patch changes (for projects with this enabled). Something like this in your crontab should work:

```
# m h  dom mon dow   command
*/10 * * * * cd patchwork; ./manage.py cron
```

The frequency should be the same as the `NOTIFICATION_DELAY_MINUTES` setting, which defaults to 10 minutes.

(Optional) Configure your VCS to Automatically Update Patches

The tools directory of the patchwork distribution contains a file named `post-receive.hook` which is a sample git hook that can be used to automatically update patches to the `Accepted` state when corresponding commits are pushed via git.

To install this hook, simply copy it to the `.git/hooks` directory on your server, name it `post-receive`, and make it executable.

This sample hook has support to update patches to different states depending on which branch is being pushed to. See the `STATE_MAP` setting in that file.

If you are using a system other than git, you can likely write a similar hook using `pwclient` to update patch state. If you do write one, please contribute it.

Some errors:

- `ERROR: permission denied for relation patchwork_...` The user that patchwork is running as (i.e. the user of the web-server) doesn't have access to the patchwork tables in the database. Check that your web server user exists in the database, and that it has permissions to the tables.
- pwclient fails for actions that require authentication, but a username and password is given in ~/.pwclientrc. Server reports "No authentication credentials given". If you're using the FastCGI interface to Apache, you'll need the `-pass-header Authorization` option to the FastCGIExternalServer configuration directive.

CHAPTER 3

User Manual

Submitting patches

Initial Submission

Patches are normally submitted with **`git send-email`** to a mailing list. For instance, if we branched from `master`, have three patches to submit, we can use:

```
$ git send-email --to=<mailing-list> --cover-letter --annotate master
```

This command will produce the following email thread (providing you have the `chainreplyto` configuration option set to `false`):

```
+ [PATCH 0/3] Cover Letter Subject
+--> [PATCH 1/3] Patch 1
+--> [PATCH 2/3] Patch 2
+--> [PATCH 3/3] Patch 3
```

Patchwork receives those mails and construct Series and Patches objects to present a high level view of the mailing-list activity and a way to track what happens to that submission.

It's a good idea to include a cover letter to introduce the work. Patchwork will also pick up that cover letter and name the series with the subject of that email.

When sending only one patch, it's a bit much to send a cover letter along with it as the commit message should provide enough context. In that case, Patchwork will use the subject of the patch as the series title.

New Versions

Sometimes, maybe even more often than hoped, one needs to resend a few patches or even entire series to address review comments.

Patchwork supports:

- Re-sending a single patch as a reply to the reviewer email. This is usually only used when a few patches have to be resent.
- Re-sending a full series as a new thread.

A Series object in patchwork tracks all the changes on top of the initial submission.

New Patch

To send a v2 of a patch part of a bigger series, one would do something similar to:

```
$ git send-email --to=<mailing-list> --cc=<reviewer> \
                 --in-reply-to=<reviewer-mail-message-id> \
                 --reroll-count 2 -1 HEAD~2
```

And, continuing the previous example, this would result in the following email thread:

```
+ [PATCH 0/3] Cover Letter Subject
+--> [PATCH 1/3] Patch 1
+--> [PATCH 2/3] Patch 2
|  +--> Re: [PATCH 2/3] Patch 2                (reviewer comments)
|     +--> [PATCH v2 2/3] Patch 2              (v2 of patch 2/3)
+--> [PATCH 3/3] Patch 3
```

Patch work will create a new *revision* of the series, updating patch #2 to the new version of that patch.

New Series

When something is really wrong or when, to address the review, most patches of a series need to be revised, re-sending individual emails can be both annoying for the patch author but also hard to follow from the reviewer side. It's then better to re-send a full new thread and forget the previous one.

Patchwork will get that and create a new revision of the initial series with all patches updated to the latest and greatest.

```
+ [PATCH 0/3] Cover Letter Subject
+--> [PATCH 1/3] Patch 1
+--> [PATCH 2/3] Patch 2
+--> [PATCH 3/3] Patch 3

+ [PATCH v2 0/3] Cover Letter Subject
+--> [PATCH v2 1/3] Patch 1                   (v2 of patch 1/3)
+--> [PATCH v2 2/3] Patch 2                   (v2 of patch 2/3)
+--> [PATCH v2 3/3] Patch 3                   (v2 of patch 3/3)
```

Patchwork uses the cover letter subject to detect that intent. So one doesn't need to use the `reroll-count` like above, the following would work as well:

```
+ [PATCH 0/3] Cover Letter Subject
+--> [PATCH 1/3] Patch 1
+--> [PATCH 2/3] Patch 2
+--> [PATCH 3/3] Patch 3

+ [PATCH 0/3] Cover Letter Subject (v2)
+--> [PATCH 1/3] Patch 1                   (v2 of patch 1/3)
+--> [PATCH 2/3] Patch 2                   (v2 of patch 2/3)
+--> [PATCH 3/3] Patch 3                   (v2 of patch 3/3)
```

Of course, we've now entered a dangerous territory. Trying to parse some human-generated text. The regular expression used accepts several ways of saying that the series is a new version of a previous one. If your favourite way isn't among what's supported, consider contributing (like filing an issue)!

Considering an initial series with `Awesome feature` as the cover letter subject, Patchwork will considering series with the following cover letter subjects as new revisions:

Regular Expression	Cover Letter
	• Awesome feature • awesome feature
`[, \(]*(v|take)[\) 0-9]+$')`	• Awesome feature v2 • awesome feature V2 • Awesome feature, v3 • Awesome feature (v4) • Awesome feature (take 5) • Awesome feature, take 6

git-pw

git-pw (or **git pw**) is a command line tool that bridges git and patchwork.

Installation

Requirements

git-pw uses GitPython and requests, so those dependencies need to be installed. Using the distribution packages should work.

On Fedora:

```
$ sudo dnf install GitPython python-requests
```

On Debian/Ubuntu:

```
$ sudo apt-get install python-git python-requests
```

Alternatively it's possible to use **pip**. `git-pw/requirements.txt` in the patchwork git repository has the list of required packages:

```
$ cat git-pw/requirements.txt
GitPython
requests
$ pip install -r requirements.txt
```

Getting git-pw

git-pw can be directly downloaded from patchwork's git repository, given execution permission (**chmod +x**) and put it anywhere in your `PATH`.

Because this tool is still very young and to easily get the latest version I would suggest cloning patchwork's repository and use a symlink. This way, **git-pw** can be updated with a single **git pull** command. From patchwork's checkout:

```
$ ln -s $PWD/git-pw/git-pw ~/.local/bin/
```

Setup

git-pw configuration is stored in git config files and so can be set per git repository. Two pieces of information are needed to get started: the URL of the patchwork instance and the project this git repository maps to.

For example, the following sets **git-pw** up for the intel-gfx project on freedesktop.org:

```
$ git config patchwork.default.url https://patchwork.freedesktop.org
$ git config patchwork.default.project intel-gfx
```

git-pw is ready to go! Applying a series known to patchwork to the current git tree is now a single command away:

```
$ git pw apply -s 122
Applying series: DP refactoring v2 (rev 1)
Applying: drm/i915: Don't pass *DP around to link training functions
Applying: drm/i915: Split write of pattern to DP reg from intel_dp_set_link_train
Applying: drm/i915 Call get_adjust_train() from clock recovery and channel eq
Applying: drm/i915: Move register write into intel_dp_set_signal_levels()
Applying: drm/i915: Move generic link training code to a separate file
...
```

CHAPTER 4

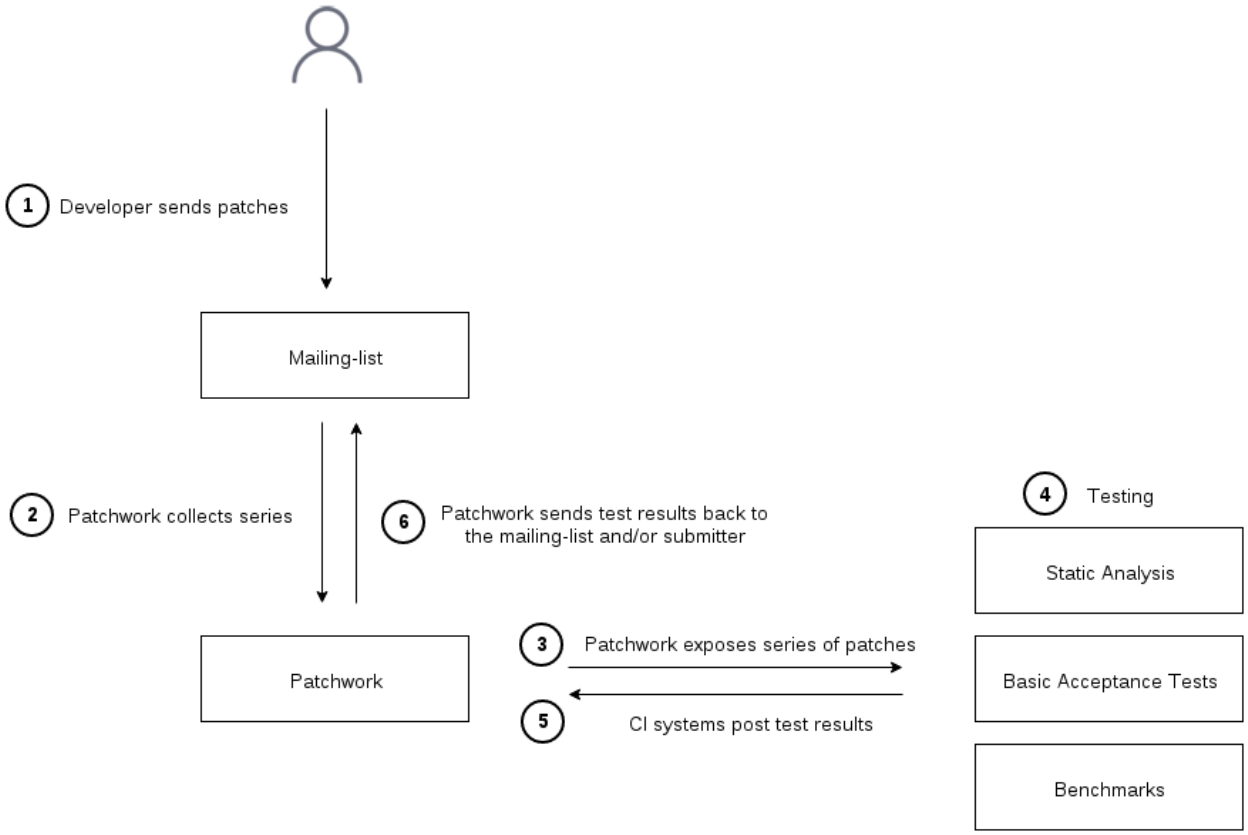
Testing with Patchwork

Patchwork can be used in conjunction with tests suites to build a CI (Continuous Integration) system.

Flow

Patches sent to the mailing list are grouped into *series* by Patchwork which, then, exposes *events* corresponding to the mailing-list activity. Listening to those *events*, one or more testing infrastructure(s) can retrieve the patches, test them and post test results back to Patchwork to display them in the web UI (User Interface). Optionally, Patchwork can send those test results back to the user and/or mailing-list.

The following diagram describes that flow:



Series and Revisions

Details about steps **1** and **2** can be found in *Submitting patches*.

Polling for events

Step **3** is Patchwork exposing new series and new series revisions appearing on the mailing list through the `series-new-revision` *event*. This event is created when Patchwork has seen all patches that are part of a new series/revision, so the API user can safely start processing the new series as soon as they notice the new event, which, for now, is done polling the `/events/` entry point.

To poll for new events, the user can use the `since` GET parameter to ask for events since the last query. The time stamp to give to that `since` parameter is the `event_time` of the last event seen.

Testing

Step **4** is, of course, where the testing happens. A mbox file of the series to test can be retrieved with the `/mbox/` revision method.

Test results

Once done, results are posted back to Patchwork in step **5** with the `/test-results/` entry point.

One note on the intention behind the `pending` state: if running the test(s) takes a long time, it's a good idea to mark the test results as `pending` as soon as the `series-new-revision` event has been detected to indicate to the user their patches have been picked up for testing.

Email reports

Finally, step **6**, the test results can be communicated back by mail to the submitter. By default, Patchwork will not send any email, that's to allow test scripts authors to develop without the risk of sending confusing emails to people.

The test result emailing is configurable per test, identified by a unique tuple (`project`, `test_name`). That configuration is done using the Django administration interface. The first parameter to configure is the email recipient(s):

none No test result email should be sent out (default).

submitter Test result emails are sent to the patch submitter.

mailing list Test result emails are sent to the patch submitter with the project mailing-list in Cc.

recipient list Test result emails are sent to the list of email addresses specified in the `Mail To list` field.

The `Mail To list` and `Mail Cc list` are list of addresses that will be appended to the `To:` and `Cc:` fields.

When the test is configured to send emails, the *when to send* can be tweaked as well:

always Always send an email, disregarding the status of the test result.

on warning/failure Only send an email when the test has some warnings or errors.

on failure Only send an email when the test has some errors.

`git-pw` helper commands

To interact with Patchwork, the REST API can be used directly with any language and an HTTP library. For python, requests is a winning choice and I'd have a look at the git-pw source code.

git-pw also provides a couple of commands that can help with writing test scripts without resorting to using the REST API.

`git pw poll-events`

`git pw poll-events` will print events since the last invocation of this command. The output is one event per line, oldest event first. **`poll-events`** stores the time stamp of the last event seen in a file called `.git-pw.$project.poll.timestamp`.

`--since` can be used to override the last seen time stamp and ask for all the events since a specific date:

```
$ git pw poll-events --since=2016-02-12
{"series": 3324, "parameters": {"revision": 1}, "name": "series-new-revision", ... }
{"series": 3304, "parameters": {"revision": 3}, "name": "series-new-revision", ... }
{"series": 3072, "parameters": {"revision": 2}, "name": "series-new-revision", ... }
{"series": 3344, "parameters": {"revision": 1}, "name": "series-new-revision", ... }
```

As shown, **`git pw poll-events`** prints JSON objects on stdout. Its intended usage is as input to a filter that would take each event one at a time and do something with it, test a new revision for instance.

As a quick example of the above, to print the list of series created or updated since a specific date (we need to use `--name` to select that type of event only), a simple filter can be written:

```
#!/bin/python
import fileinput
import json

for line in fileinput.input():
    event = json.loads(line)
    series = event['series']
    revision = event['parameters']['revision']
    print("series %d (rev %d)" % (series, revision))
```

Which gives:

```
$ git pw poll-events --name=series-new-revision --since=2016-02-12 | ./show-series
series 3324 (rev 1)
series 3304 (rev 3)
series 3072 (rev 2)
series 3344 (rev 1)
```

git pw post-result

The other side of the patchwork interaction with testing is sending test results back. Here as well **git-pw** provides a command to simplify the process. Remember it's always possible to directly use the REST API.

No need to repeat what's written in the */test-results/* documentation here. Just a couple of examples, setting a test result as pending:

```
$ git pw post-result 3324 checkpatch.pl pending
```

And posting the final results:

```
$ git pw post-result 3324 checkpatch.pl failure --summary-from-file results.txt
```

Example: running checkpatch.pl on incoming series

A slightly longer example can be found in the Patchwork repository, in docs/examples/testing-checkpatch.py. This script will take `series-new-revision` events as input, as offered by **git pw poll-events** and run checkpatch.pl on the incoming series. The main complexity beyond what has been explained in this document is that checkpatch.pl is run on each patch of the series individually (by looping on all mails of the series mbox) and the checkpatch.pl output is aggregated to be sent in the `summary` field of the test result.

Putting the following line in a cron entry should be enough to run checkpatch.pl on each new series:

```
git pw poll-events | testing-checkpatch.pl
```

There are a few improvements to make to have a nicer solution: for instance, one could make sure that the checkpatch.pl script is up-to-date by updating the Linux checkout before running the test.

CHAPTER 5

REST API

Patchwork exposes a REST API to allow other systems and scripts to interact with it. The basic service it offers is exposing a mailing-list used for sending patches and review comments as high level objects: series, revisions and patches.

series A collection of revisions. Series objects are created, along with an initial revision, when a set of patches are sent to a mailing-list, usually through **`git send-email`**. Series can evolve over time and gain new revisions as the work matures through reviews, testing and new iterations.

More about series and revisions can be found in *Submitting patches*.

revision A collection of patches.

patch The usual collection of changes expressed as a diff. With **`git`**, a patch also contains full commit metadata.

API Patterns

All the API entry points share common patterns to offer a coherent whole and limit surprises when using the API.

Lists

Various entry points expose lists of objects. They all follow the same structure:

```
{
    "count": 25,
    "next": "http://patchwork.example.com/api/1.0/series/?page=2",
    "previous": null,
    "results": [
        {
            "object": 0
        },
        {
            "object": 1
```

```
        },
        {
        },
        {
            "object": 19
        },
    ]
}
```

Lists are paginated with 20 elements per page by default. `count` is the total number of objects while `next` and `previous` will hold URLs to the next and previous pages. It's possible to change the number of elements per page with the `perpage` GET parameter, with a limit of 100 elements per page.

API Reference

API Metadata

GET /api/1.0/
Metadata about the API itself.

```
GET /api/1.0/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
    "revision": 0
}
```

Response JSON Object

- **revision** (*int*) – API revision. This can be used to ensure the server supports a feature introduced from a specific revision. The list of API revisions and the changes introduced by each of them is documented in *API Revisions*.

Projects

A project is merely one of the projects defined for this patchwork instance.

GET /api/1.0/projects/
List of all projects.

```
GET /api/1.0/projects/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
```

```
    "count": 2,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 2,
            "name": "beignet",
            "linkname": "beignet",
            "listemail": "beignet@lists.freedesktop.org",
            "web_url": "http://www.freedesktop.org/wiki/Software/Beignet/",
            "scm_url": "git://anongit.freedesktop.org/git/beignet",
            "webscm_url": "http://cgit.freedesktop.org/beignet/"
        },
        {
            "id": 1,
            "name": "Cairo",
            "linkname": "cairo",
            "listemail": "cairo@cairographics.org",
            "web_url": "http://www.cairographics.org/",
            "scm_url": "git://anongit.freedesktop.org/git/cairo",
            "webscm_url": "http://cgit.freedesktop.org/cairo/"
        }
    ]
}
```


GET /api/1.0/projects/(string: linkname)/

GET /api/1.0/projects/(int: project_id)/

```
GET /api/1.0/projects/intel-gfx/ HTTP/1.1
Accept: application/json
```


```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "id": 1,
    "name": "intel-gfx",
    "linkname": "intel-gfx",
    "listemail": "intel-gfx@lists.freedesktop.org",
    "web_url": "",
    "scm_url": "",
    "webscm_url": ""
}
```


Events

GET /api/1.0/projects/(string: linkname)/events/

GET /api/1.0/projects/(int: project_id)/events/

List of events for this project.

```
GET /api/1.0/projects/intel-gfx/events/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "count": 23,
    "next": "http://patchwork.example.com/api/1.0/events/?page=2",
    "previous": null,
    "results": [
        {
            "name": "series-new-revision",
            "event_time": "2015-10-20T19:49:49.494183",
            "series": 23,
            "patch": null,
            "user": null,
            "parameters": {
                "revision": 2
            }
        },
        {
        },
        {
            "name": "patch-state-change",
            "event_time": "2016-02-18T09:30:33.853206",
            "series": 285,
            "patch": 685
            "user": 1,
            "parameters": {
                "new_state": "Under Review",
                "previous_state": "New"
            }
        }
    ]
}
```

Query Parameters

- **since** – Retrieve only events newer than a specific time. Format is the same as `event_time` in response, an ISO 8601 date. That means that the `event_time` from the last seen event can be used in the next query with a `since` parameter to only retrieve events that haven't been seen yet.
- **name** – Filter the events by name. This field is a comma separated list of events names.
- **series** – Filter the events by series id.
- **patch** – Filter the events by patch id.

Each event type has some `parameters` specific to that event. At the moment, two events are possible:

- **series-new-revision**: This event corresponds to patchwork receiving a new revision of a series, should it be the initial submission or subsequent updates. The difference can be made by looking at the version of the series.

  This event only appears when patchwork has received the full set of mails belonging to the same series, so the revision object is guaranteed to contain all patches.

revision: The version of the new revision that has been created. `series` and `revision` can be used to retrieve the corresponding patches.

- **patch-state-change**: This event corresponds to patchwork receiving a patch state change, either automatic or manually performed by an authorized user, who will be identified by its patchwork-user id.

Series

A series object represents a lists of patches sent to the mailing-list through **`git send-email`**. It also includes all subsequent patches that are sent to address review comments, both single patch and full new series.

A series has then `n` revisions, `n` going from `1` to `version`.

`GET /api/1.0/projects/(string: linkname)/series/`

`GET /api/1.0/projects/(int: project_id)/series/`
List of all Series belonging to a specific project. The project can be specified using either its `linkname` or `id`.

```
GET /api/1.0/projects/intel-gfx/series/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "count": 59,
    "next": "http://patchwork.example.com/api/1.0/projects/intel-gfx/series/?
↪page=2",
    "previous": null,
    "results": [
        {
            "id": 3,
            "project": 1,
            "name": "drm/i915: Unwind partial VMA rebinding after failure in set-
↪cache-level",
            "n_patches": 1,
            "submitter": 77,
            "submitted": "2015-10-09T11:51:38",
            "last_updated": "2015-10-09T11:51:59.013345",
            "version": 1,
            "reviewer": null,
            "test_state": null,
            "state": "initial",
            "state_summary": [
                {
                    "count": 1,
                    "name": "New",
                    "final": false
                }
            ]
        },
        {
            "id": 5,
            "project": 1,
            "name": "RFC drm/i915: Stop the machine whilst capturing the GPU␣
↪crash dump",
```

```
            "n_patches": 1,
            "submitter": 77,
            "submitted": "2015-10-09T12:21:45",
            "last_updated": "2015-10-09T12:21:58.657976",
            "version": 1,
            "reviewer": null,
            "test_state": null,
            "state": "initial",
            "state_summary": [
                {
                    "count": 1,
                    "name": "New",
                    "final": false
                }
            ]
        }
    ]
}
```


Response JSON Object

- **state** – The state of the series. One of `initial`, `in progress`, `done` or `incomplete`.
- **state_summary** – A summary of the patch status in the more recent revision of the series. This a a list of objects containing the number of patches (`count`) in a given state (`name`). `final` is whether the state is final or not, if a final decision (ie. merged or rejected) has been made about a patch.

Query Parameters

- **project** – Filter series by project `id`.
- **name** – Filter series by name.
- **submitter** – Filter series by submitter `id`. `self` can be used as a special value meaning the current logged in user.
- **reviewer** – Filter series by reviewer `id` or `null` for no reviewer assigned.
- **submitted_since** – Retrieve only submitted series newer than a specified time. Format is the same as `submitted` in response, an ISO 8601 date.
- **updated_since** – Retrieve only updated series newer than a specified time. Format is the same as `last_updated` in response, an ISO 8601 date.
- **submitted_before** – Retrieve only submitted series older than the specified time. Format is the same as `submitted` in response, an ISO 8601 date.
- **updated_before** – Retrieve only updated series older than a specified time. Format is the same as `last_updated` in response, an ISO 8601 date.
- **test_state** – Filter series by test state. Possible values are `pending`, `info`, `success`, `warning`, `failure` or `null` series that don't have any test result. It's also possible to give a comma separated list of states.

GET /api/1.0/series/

List of all Series known to patchwork.

```
GET /api/1.0/series/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: application/json

{
    "count": 344,
    "next": "http://patchwork.example.com/api/1.0/series/?page=2",
    "previous": null,
    "results": [
        {
            "id": 10,
            "project": 1,
            "name": "intel: New libdrm interface to create unbound wc user␣
↪mappings for objects",
            "n_patches": 1,
            "submitter": 10,
            "submitted": "2015-01-02T11:06:40",
            "last_updated": "2015-10-09T07:55:18.608251",
            "version": 1,
            "reviewer": null,
            "test_state": null,
            "state": "initial",
            "state_summary": [
                {
                    "count": 1,
                    "name": "New",
                    "final": false
                }
            ]
        },
        {
            "id": 1,
            "project": 1,
            "name": "PMIC based Panel and Backlight Control",
            "n_patches": 4,
            "submitter": 1,
            "submitted": "2014-12-26T10:23:26",
            "last_updated": "2015-10-09T07:55:01.558523",
            "version": 1,
            "reviewer": null,
            "state": "initial",
            "state_summary": [
                {
                    "count": 4,
                    "name": "New",
                    "final": false
                }
            ]
        },
    ]
}
```

GET /api/1.0/series/(int: series_id)/

A series (*series_id*). A Series object contains metadata about the series.

```
GET /api/1.0/series/47/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, PUT, PATCH, HEAD, OPTIONS

{
    "id": 47,
    "name": "Series without cover letter",
    "n_patches": 2,
    "submitter": 21,
    "submitted": "2015-01-13T09:32:24",
    "last_updated": "2015-10-09T07:57:23.541373",
    "version": 1,
    "reviewer": null,
    "state": "initial",
    "state_summary": [
        {
            "count": 2,
            "name": "New",
            "final": false
        }
    ]
}
```

GET /api/1.0/series/(int: series_id)/revisions/

The list of revisions of the series *series_id*.

```
GET /api/1.0/series/47/revisions/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "version": 1,
            "cover_letter": null,
            "patches": [
                120,
                121
            ]
        }
    ]
}
```

GET /api/1.0/series/(int: series_id)/revisions/(int: version)/

The specific `version` of the series *series_id*.

```
GET /api/1.0/series/47/revisions/1/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "version": 1,
    "cover_letter": null,
    "patches": [
        120,
        121
    ]
}
```

GET /api/1.0/series/(int: series_id)/revisions/(int: version)/mbox/

Retrieve an mbox file that will contain all patches of this revision, in order in which to apply them. This mbox file can be directly piped into `git am`.

Query Parameters

- **link** – Add an HTTP link to the Patchwork patch page in each commit message. This link is preceded by a tag which name is given as argument of this parameter, eg. `?link=Patchwork`.

```
$ curl -s http://patchwork.example.com/api/1.0/series/42/revisions/2/mbox/ | git am -3
```

POST /api/1.0/series/(int: series_id)/revisions/(int: version)/test-results/

Post test results for this revision.

```
POST /api/1.0/series/47/revisions/1/test-results/ HTTP/1.1

{
    "test_name": "checkpatch.pl",
    "state": "success",
    "url": "http://jenkins.example.com/logs/47/checkpatch.log",
    "summary": "total: 0 errors, 0 warnings, 10 lines checked"
}
```

Request JSON Object

- **test_name** – Required. The name of the test we're reporting results for. This uniquely identifies the test. Any subsequent data sent through this entry point with the same `test_name` will be conflated into the same object. It's thus possible to create a test result with a `pending` state when a CI system picks up patches to indicate testing has started and then update the result with the final (`state`, `url`, `summary`) when finished.
- **state** – Required. State of the test results. One of `pending`, `success`, `warning` or `failure`
- **url** – Optional. A URL where to find the detailed logs of the test run.
- **summary** – Optional. A summary with some details about the results. If set, this will be displayed along with the test result to provide some detailed about the failure. It's suggested to use `summary` for something short while `url` can be used for full logs, which can be rather large.

GET /api/1.0/series/(int: series_id)/revisions/(int: version)/test-results/

Get test results for this revision.

```
GET /api/1.0/series/47/revisions/1/test-results/ HTTP/1.1

[
 {
     "date": "2017-08-09T23:00:03.529",
     "state": "pending",
     "summary": "total: 0 errors, 0 warnings, 10 lines checked"
     "test_name": "checkpatch.pl",
     "url": "http://jenkins.example.com/logs/47/checkpatch.log"
 },
 {
     "date": "2017-08-09T23:00:05.551",
     "state": "warning",
     "summary": "total: 0 errors, 2 warnings, 20 passes"
     "test_name": "BAT",
     "url": "http://jenkins.example.com/logs/47/BAT.log"
 }
]
```

Request JSON Object

- **date** – Date when the results were posted to the patchwork (ISO 8061).

Patches

GET /api/1.0/projects/(string: linkname)/patches/

GET /api/1.0/projects/(int: project_id)/patches/

List of all patches belonging to a specific project. The project can be specified using either its `linkname` or `id`.

```
GET /api/1.0/projects/intel-gfx/patches/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "count": 1392,
    "next": "http://patchwork.example.com/api/1.0/projects/intel-gfx/patches/?
↪page=2",
    "previous": null,
    "results": [
        {
            "id": 1,
            "project": 1,
            "name": "[RFC,1/4] drm/i915: Define a common data structure for Panel␣
↪Info",
            "date": "2014-12-26T10:23:27",
            "last_updated": "2014-12-26T10:23:27",
            "submitter": 1,
```

```
            "state": 1,
            "content": "<diff content>"
        },
        {
            "id": 4,
            "project": 1,
            "name": "[RFC,2/4] drm/i915: Add a drm_panel over INTEL_SOC_PMIC",
            "date": "2014-12-26T10:23:28",
            "last_updated": "2014-12-26T10:23:28",
            "submitter": 1,
            "state": 1,
            "content": "<diff content>"
        }
    ]
}
```


Query Parameters

- **project** – Filter patches by project `id`.
- **name** – Filter patches by name.
- **submitter** – Filter patches by submitter `id`. `self` can be used as a special value meaning the current logged in user.
- **submitted_since** – Retrieve only submitted patches newer than a specified time. Format is the same as `date` in response, an ISO 8601 date.
- **updated_since** – Retrieve only updated patches newer than a specified time. Format is the same as `last_updated` in response, an ISO 8601 date.
- **submitted_before** – Retrieve only submitted patches older than the specified time. Format is the same as `date` in response, an ISO 8601 date.
- **updated_before** – Retrieve only updated patches older than a specified time. Format is the same as `last_updated` in response, an ISO 8601 date.

GET /api/1.0/patches/

List of all patches.

```
GET /api/1.0/patches/ HTTP/1.1
Accept: application/json
```


```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "count": 1392,
    "next": "http://patchwork.example.com/api/1.0/patches/?page=2",
    "previous": null,
    "results": [
        {
            "id": 1,
            "project": 1,
            "name": "[RFC,1/4] drm/i915: Define a common data structure for Panel␣
↪Info",
            "date": "2014-12-26T10:23:27",
```

```
            "last_updated": "2014-12-26T10:23:27",
            "submitter": 1,
            "state": 1,
            "content": "<diff content>"
        },
        {
            "id": 4,
            "project": 1,
            "name": "[RFC,2/4] drm/i915: Add a drm_panel over INTEL_SOC_PMIC",
            "date": "2014-12-26T10:23:28",
            "last_updated": "2014-12-26T10:23:28",
            "submitter": 1,
            "state": 1,
            "content": "<diff content>"
        }
    ]
}
```

GET /api/1.0/patches/(int: patch_id)/
A specific patch.

```
GET /api/1.0/patches/120/ HTTP/1.1
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "id": 120,
    "name": "[1/2] drm/i915: Balance context pinning on reset cleanup",
    "date": "2015-01-13T09:32:24",
    "last_updated": "2015-01-13T09:32:24",
    "submitter": 21,
    "state": 1,
    "content": "<diff content>"
}
```

GET /api/1.0/patches/(int: patch_id)/mbox/
Retrieve an mbox file. This mbox file can be directly piped into `git am`.

Query Parameters

- **link** – Add an HTTP link to the Patchwork patch page in the commit message. This link is preceded by a tag which name is given as argument of this parameter, eg. `?link=Patchwork`.

```
$ curl -s http://patchwork.example.com/api/1.0/patches/42/mbox/ | git am -3
```

API Revisions

Revision 3

- Add test results entry points:

  - /series/${id}/revisions/${version}/test-results/
  - /series/${id}/revisions/${version}/newrevision/
- Add the *project*, *name*, *submitter*, *reviewer*, *submitted_since*, *updated_since*, *submitted_before*, *updated_before* and *test_state* query parameters to the list of series entry points.
- Add the *test_state*, *state* and *test_summary* series fields.
- Add the patch-state-change event.
- Add the *name* query parameter to the /events/ entry point.

Revision 2

- Add mbox entry points for both patches and series:
  - /patches/${id}/mbox/
  - /series/${id}/revisions/${version}/mbox/
- Add a `parameters` field to events and include the revision number to the `series-new-revision` event.
- Change /series/${id}/revisions/ to follow the same list system as other entry points. This is technically an API change, but the impact is limited at this early point. Hopefully no one will ever find out.
- Document how lists of objects work.
- Make all DateTime field serialize to ISO 8061 format and not the ECMA 262 subset.
- Add `since`, `name`, `series` and `patch` GET parameters to /projects/${id,linkname}/events/

Revision 1

- Add /projects/${linkname}/events/ entry point.

Revision 0

- Initial revision. Basic objects exposed: api root, projects, series, revisions and patches.

CHAPTER 6

Developing patchwork

Quick Start

We have scripts that will get developers started in no time:

```
$ git clone https://github.com/dlespiau/patchwork/
$ cd patchwork
$ ./tools/setup-devel.sh
$ ./tools/run-devel.sh
```

`setup-devel.sh` will:

- Create a virtual environment in the `venv` directory,
- Install all the required dependencies in that virtual environment,
- Populate a SQLite database with a few patches,
- Create an `admin` account with `pass` as password.

`run-devel.sh` will run the web server serve the patchwork application. Pointing your browser to http://127.0.0.1:8000/ should bring up patchwork.

Using virtualenv

It's a good idea to use virtualenv to develop Python software. Virtual environments are "instances" of your system Python, without any of the additional Python packages installed. They are useful to develop and deploy patchwork against a "well known" set of dependencies, but they can also be used to test patchwork against several versions of Django.

1. Install pip, virtualenv (python-pip, python-virtualenv packages)

Because we're going to recompile our dependencies, we'll also need development headers. For the MySQL/MariaDB setups these are `mariadb-devel` (Fedora), `libmysqlclient-dev` (Debian)

2. Create a new virtual environment.

Inside a virtual env, we'll just install the dependencies needed for patchwork and run it from there.

```
$ virtualenv django-1.8
```

This will create a virtual env called 'django-1.8' in the eponymous directory.

3. Activate a virtual environment

```
$ source django-1.8/bin/activate
(django-1.8)$
```

The shell prompt is prepended with the virtual env name.

4. Install the required dependencies

To ease this task, it's customary to maintain a list of dependencies in a text file and install them in one go. Patchwork can work with multiple databases so we keep the requirements for each supported db:

```
(django-1.8)$ pip install -r docs/requirements-dev-mysql.txt
```

or:

```
(django-1.8)$ pip install -r docs/requirements-dev-postgresql.txt
```

5. Export the `DJANGO_SETTINGS_MODULE` variable

Django needs to be told which settings to use. By default it will try to load settings from the `patchwork/settings/production.py` file. This can be overridden with the `DJANGO_SETTINGS_MODULE` environment variable.

Patchwork provides a convenience settings template suitable for development in `patchwork/settings/dev.py`. To use it, you can simply export the path to this file (in Python module format) like so:

```
(django-1.8)$ export DJANGO_SETTINGS_MODULE=patchwork.settings.dev
```

And adjust your database settings through environment variables. See the *Environment Variables* section below for details. For example:

```
(django-1.8)$ export PW_TEST_DB_USER=root
(django-1.8)$ export PW_TEST_DB_PASS=password
```

You may also provide your own settings file and have `DJANGO_SETTINGS_MODULE` point to that file.

6. Run the development server

```
(django-1.8)$ ./manage.py runserver
```

Once finished, you can kill the server (`Ctrl` + `C`) and exit the virtual environment:

```
(django-1.8)$ deactivate
$
```

Should you wish to re-enter this environment, simply source the `activate` script again.

Environment Variables

The following environment variables are available to configure various settings if `dev.py` is used:

PW_TEST_DB_NAME Name of the database. Defaults to `patchwork`.

PW_TEST_DB_USER User name to access the database with. Defaults to `patchwork`.

PW_TEST_DB_PASS Password to access the database with. Defaults to `password`.

PW_TEST_DB_TYPE Type of database to use. Either `mysql` (default) or `postgres`.

Running Tests

patchwork includes a tox script to automate testing. Before running this, you should probably install tox:

```
$ pip install tox
```

You can show available targets like so:

```
$ tox --list
```

You'll see that this includes a number of targets to run unit tests against the different versions of Django supported, along with some other targets related to code coverage and code quality. To run these, use the `-e` parameter:

```
$ tox -e py27-django18
```

In the case of the unit tests targets, you can also run specific tests by passing the fully qualified test name as an additional argument to this command:

```
$ tox -e py27-django18 patchwork.tests.SubjectCleanUpTest
```

Because patchwork supports multiple versions of Django, it's very important that you test against all supported versions. When run without argument, tox will do this:

```
$ tox
```

HTTP Routing Table

/api