Patchwork

Release 2.0-alpha

Sep 05, 2017

User Documentation

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.

CHAPTER 1

Overview

The key concepts or models of Patchwork are outlined below.

Projects

Projects typically represent a software project or sub-project. A Patchwork server can host multiple projects. Each project can have multiple maintainers. Projects usually have a 1:1 mapping with a mailing list, though it's also possible to have multiple projects in the same list using the subject as filter. Patches, cover letters, and series are all associated with a single project.

People

People are anyone who has submitted a patch, cover letter, or comment to a Patchwork instance.

Users

Users are anyone who has created an account on a given Patchwork instance.

Standard Users

A standard user can associate multiple email addresses with their user account, create bundles and store TODO lists.

Maintainers

Maintainers are a special type of user that with permissions to do certain operations that regular Patchwork users can't. Patchwork maintainers usually have a 1:1 mapping with a project's code maintainers though this is not necessary.

The operations that a maintainer can invoke include:

- Change the state of a patch
- Archive a patch
- Delegate a patch, or be delegated a patch

Submissions

Patchwork captures three types of mail to mailing lists: patches, cover letters, and replies to either patches or cover letters, a.k.a. comments. Any mail that does not fit one of these categories is ignored.

Patches

Patches are the central object in Patchwork structure. A patch contains both a diff and some metadata, such as the name, the description, the author, the version of the patch etc. Patchwork stores not only the patch itself but also various metadata associated with the email that the patch was parsed from, such as the message headers or the date the message itself was received.

Cover Letters

Cover letters provide a way to offer a "big picture" overview of a series of patches. When using Git, these mails can be recognised by way of their *0/N* subject prefix, e.g. *[00/11] A sample series*. Like patches, Patchwork stores not only the various aspects of the cover letter itself, such as the name and body of the cover letter, but also various metadata associated with the email that the cover letter was parsed from.

Comments

Comments are replies to a submission - either a patch or a cover letter. Unlike a Mail User Agent (MUA) like Gmail, Patchwork does not thread comments. Instead, every comment is associated with either a patch or a cover letter, and organized by date.

Patch Metadata

Patchwork allows users to store various metadata against patches. This metadata is only configurable by a maintainer.

States

States track the state of patch in its lifecycle. States vary from project to project, but generally a minimum subset of "new", "rejected" and "accepted" will exist.

Delegates

Delegates are Patchwork users who are responsible for both reviewing a patch and setting its eventual state in Patchwork. This makes them akin to reviewers in other tools. Delegation works particularly well for larger projects where various subsystems, each with their own maintainer(s), can be identified. Only one delegate can be assigned to a patch.

Note: Patchwork supports automatic delegation of patches. Refer to *Autodelegation* for more information.

Tags

Tags are specially formatted metadata appended to the foot the body of a patch or a comment on a patch. Patchwork extracts these tags at parse time and associates them with the patch. You add extra tags to an email by replying to the email. The following tags are available on a standard Patchwork install:

Acked-by:

For example:

```
Acked-by: Stephen Finucane <stephen@that.guru>
```

Tested-by:

For example:

```
Tested-by: Stephen Finucane <stephen@that.guru>
```

Reviewed-by:

For example:

```
Tested-by: Stephen Finucane <stephen@that.guru>
```

The available tags, along with the significance of said tags, varies from project to project and Patchwork instance to Patchwork instance. The kernel project documentation provides an overview of the supported tags for the Linux kernel project.

Checks

Checks store the results of any tests executed (or executing) for a given patch. This is useful, for example, when using a continuous integration (CI) system to test patches. Checks have a number of fields associated with them:

Context

A label to discern check from the checks of other testing systems

Description

A brief, optional description of the check

Target URL

A target URL where a user can find information related to this check, such as test logs.

State

The state of the check. One of: pending, success, warning, fail

User

The user creating the check

Note: Checks can only be created through the Patchwork APIs. Refer to *../api* for more information.

Collections

Patchwork provides a number of ways to store groups of patches. Some of these are automatically generated, while others are user-defined.

Series

Series are groups of patches, along with an optional cover letter. Series are mostly dumb containers, though they also contain some metadata themselves such as a version (which is inherited by the patches and cover letter) and a count of the number of patches found in the series.

Bundles

Bundles are custom, user-defined groups of patches. Bundles can be used to keep patch lists, preserving order, for future inclusion in a tree. There's no restriction of number of patches and they don't even need to be in the same project. A single patch also can be part of multiple bundles at the same time. An example of Bundle usage would be keeping track of the Patches that are ready for merge to the tree.

To-do Lists

Patchwork users can store a to-do list of patches.

Events

Events are raised whenever patches are created or modified.

All events have a number of common properties, along with some event-specific properties:

category

The type of event

project

The project this event belongs to

date

When this event was created

Note: Checks can only be created and read through the Patchwork APIs. Refer to *../api/index* for more information.

Cover Letter Created

Sent when a cover letter is created.

category

`cover-created`

cover

Created cover letter

Patch Created

Sent when a patch is created.

category

`patch-created`

patch

Created patch

Patch Completed

Sent when a patch in a series has its dependencies met, or when a patch that is not in a series is created (since that patch has no dependencies).

category

`patch-completed`

patch

Completed patch

series

Series from which patch dependencies were extracted, if any

Patch Delegated

Sent when a patch's delegate is changed.

category

`patch-delegated`

patch

Updated patch

previous

Previous delegate, if any

current

Current delegate, if any

Patch State Changed

Sent when a patch's state is changed.

category

`patch-state-changed`

patch

Updated patch

previous

Previous state

current

Current state

Check Created

Sent when a patch check is created.

category

`check-created`

check

Created check

Series Created

Sent when a series is created.

category

`series-created`

series

Created series

Series Completed

Sent when a series is completed.

category

`series-completed`

series

Completed series

What's Not Exposed

- Bundles

  We don't expose an "added to bundle" event as it's unlikely that this will be useful to either users or CI setters.

- Comments

  Like Bundles, there likely isn't much value in exposing these via the API.

CHAPTER 2

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.

CHAPTER 3

Autodelegation

Autodelegation allows patches to be automatically delegated to a user based on the files modified by the patch. To do this, a number of rules can be configured in the project administration page. This can usually be found at:

/admin/patchwork/project/<project_id>/change

Note: Autodelegation can only be configured by Patchwork administrators, i.e. those that can access the 'admin' panel. If you require configuration of autodelegation rules on a local instance, contact your Patchwork administrator.

In this section there are the following fields:

User

The patchwork user that should be autodelegated to the patch

Priority

The priority of the rule relative to other patches. Higher values indicate higher priority. If two rules have the same priority, ordering will be based on the path.

Path

A path in fnmatch format. The fnmatch library allows for limited, Unix shell-style wildcarding. Filenames are extracted from patch lines beginning with `---` or `+++`. Note that for projects using Git or Mercurial, the tools these VCS provide for producing patches are prefixed with *a* or *b*. You should account for this in your path. For example, to match the path *patchwork/views* (relative to the top of a Git repo) your pattern should be:

```
?/patchwork/views/*
```

It is also possible to use relative paths, such as:

```
*/manage.py
```

For projects using other VCSs like Subversion can simply use a bare path:

```
patchwork/views/*
```

Rules are configured by setting the above fields and saving the rules. These rules will be applied at patch parse time.

CHAPTER 4

Hint Headers

Patchwork provides a number of special email headers to control how a patch is handled when it is received. The examples provided below use *git-send-email*, but custom headers can also be set when using tools like *mutt*.

X-Patchwork-Ignore

Valid values: *

When set, the mere presence of this header will ensure the provided email is not parsed by Patchwork. For example:

```
$ git send-email --add-header="X-Patchwork-Ignore: test" master
```

X-Patchwork-Delegate

Valid values: An email address associated with a Patchwork user

If set and valid, the user corresponding to the provided email address will be assigned as the delegate of any patch parsed. If invalid, it will be ignored. For example:

```
$ git send-email --add-header="X-Patchwork-Delegate: a@example.com" master
```

X-Patchwork-State

Valid values: Varies between deployments. This can usually be one of "Accepted", "Rejected", "RFC" or "Awaiting Upstream", among others.

If set and valid, the state provided will be assigned as the state of any patch parsed. If invalid, it will be ignored. For example:

```
$ git send-email --add-header="X-Patchwork-State: RFC" master
```

CHAPTER 5

Clients

A number of clients are available for interacting with Patchwork's various APIs.

pwclient

The *pwclient* application, provided with Patchwork, can be used to interact with Patchwork from the command line. Functionality provided by *pwclient* includes:

- Listing patches, projects, and checks
- Downloading and applying patches to a local code base
- Modifying the status of patches
- Creating new checks

pwclient can be downloaded from the Ozlabs Patchwork instance, or at the following path for most other Patchwork instances:

http://patchwork.example.com/pwclient/

where *patchwork.example.com* corresponds to the URL a Patchwork instance is hosted at.

Once downloaded, view information about all the operations supported by *pwclient*, run:

```
$ pwclient --help
```

git-pw

The *git-pw* application can be used to integrate Git with Patchwork. The *git-pw* application relies on the REST API and can be used to interact to list, download and apply series, bundles and individual patches.

More information on *git-pw*, including installation and usage instructions, can be found in the documentation and the GitHub repo.

CHAPTER 6

Installation

This document describes the necessary steps to configure Patchwork in a production environment. This requires a significantly "harder" deployment than the one used for development. If you are interested in developing Patchwork, refer to the *development guide* instead.

This document describes a single-node installation of Patchwork, which will handle the database, server, and application. It is possible to split this into multiple servers, which would provide additional scalability and availability, but this is is out of scope for this document.

Deployment Guides, Provisioning Tools and Platform-as-a-Service

Before continuing, it's worth noting that Patchwork is a Django application. With the exception of the handling of incoming mail (described below), it can be deployed like any other Django application. This means there are tens, if not hundreds, of existing articles and blogs detailing how to deploy an application like this. As such, if any of the below information is unclear then we'd suggest you go search for "Django deployment guide" or similar, deploy your application, and submit a patch for this guide to clear up that confusion for others.

You'll also find that the same search reveals a significant number of existing deployment tools aimed at Django. These tools, be they written in Ansible, Puppet, Chef or something else entirely, can be used to avoid much of the manual configuration described below. If possible, embrace these tools to make your life easier.

Finally, many Platform-as-a-Service (PaaS) providers and tools support deployment of Django applications with minimal effort. Should you wish to avoid much of the manual configuration, we suggest you investigate the many options available to find one that best suits your requirements. The only issue here will likely be the handling of incoming mail - something which many of these providers don't support. We address this in the appropriate section below.

Requirements

For the purpose of this guide, we will assume an *Ubuntu 16.04 host*: commands, package names and/or package versions will likely change if using a different distro or release. Similarly, usage of different package versions to

the ones suggested may require slightly different configuration. For example, this guide describes configuration with *Python 3* and using Python 2 will require different packages and some minor changes to configuration files.

Before beginning, you should update this system:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

We also need to configure some environment variables to ease deployment:

`DATABASE_NAME=patchwork`

Name of the database. We'll name this after the application itself.

`DATABASE_USER=www-data`

Username that the Patchwork web application will access the database with. We will use `www-data`, for reasons described below.

`DATABASE_PASS=`

Password that the Patchwork web application will access the database with. As we're going to use ident authentication (more on this later), this will be unset.

`DATABASE_HOST=`

IP or hostname of the database host. As we're hosting the application on the same host as the database and hoping to use ident authentication, this will be unset.

`DATABASE_PORT=`

Port of the database host. As we're hosting the application on the same host as the database and using the default configuration, this will be unset.

The remainder of the requirements are listed as we install and configure the various components required.

Database

Install Requirements

We're going to rely on PostgreSQL, though MySQL is also supported:

```
$ sudo apt-get install -y postgresql postgresql-contrib
```

Configure Database

We need to create a database for the system using the database name above. In addition, we need to add accounts for two system users, the web user (the user that the web server runs as) and the mail user (the user that the mail server runs as). On Ubuntu these are `www-data` and `nobody`, respectively. PostgreSQL supports ident-based authentication, which uses the standard UNIX authentication method as a backend. This means no database-specific passwords need to be configured.

PostgreSQL created a user account called `postgres`; you will need to run commands as this user.

```
$ sudo -u postgres createdb $DATABASE_NAME
$ sudo -u postgres createuser $DATABASE_USER
$ sudo -u postgres createuser nobody
```

We will also need to apply permissions to the tables in this database but seeing as the tables haven't actually been created yet this will have to be done later.

Finally, we should enable `trust` authentication. This will allow us to use the local `www-data` user without having to set a password for a daemon account. Replace the following line in `/etc/postgresql/9.6/main/pg_hba.conf`:

```
local   all             all                                     ident
```

with:

```
local   all             all                                     trust
```

Patchwork

Install Requirements

The first requirement is Patchwork itself. It can be downloaded like so:

```
$ wget https://github.com/getpatchwork/patchwork/archive/v2.0.0.tar.gz
```

We will install this under */opt*, though this is only a suggestion:

```
$ tar -xvzf v2.0.0.tar.gz
$ sudo mv v2.0.0 /opt/patchwork
```

Important: Per the Django documentation, source code should not be placed in your web server's document root as this risks the possibility that people may be able to view your code over the Web. This is a security risk.

Next we require Python. If not already installed, then you should do so now. Patchwork supports both Python 2.7 and Python 3.3+, though we're going to use the latter to ease future upgrades. Python 3 is installed by default, but you should validate this now:

```
$ sudo apt-get install -y python3
```

We also need to install the various requirements. Let's use system packages for this also:

```
$ sudo apt-get install -y python3-django python3-psycopg2 \
    python3-djangorestframework python3-django-filters
```

Tip: The pkgs.org website provides a great reference for identifying the name of these dependencies.

You can also install requirements using *pip*. If using this method, you can install requirements like so:

```
$ sudo pip install -r /opt/patchwork/requirements-prod.txt
```

Configure Patchwork

You will also need to configure a settings file for Django. A sample settings file is provided that defines default settings for Patchwork. You'll need to configure settings for your own setup and save this as `production.py`.

```
$ cd /opt/patchwork
$ cp patchwork/settings/production{.example,}.py
```

Alternatively, you can override the `DJANGO_SETTINGS_MODULE` environment variable and provide a completely custom settings file.

The provided `production.example.py` settings file is configured to read configuration from environment variables. We're not actually going to use this here, preferring to hard code settings instead. If you wish to use environment variables, you should export each setting using the appropriate name, e.g. `DJANGO_SECRET_KEY`, `DATABASE_NAME`, `EMAIL_HOST`, etc.

Important: You should not include shell variables in settings but rather hardcoded values. These settings files are evaluated in Python - not a shell. Load any required environment variables using `os.environ`.

Databases

As described previously, we're going to modify the `production.py` settings file we created earlier to hard code our settings. Replace the `DATABASE` setting with the below to use the database configuration we're described in the introduction:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'patchwork',
        'USER': 'www-data',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
        'TEST': {
            'CHARSET': 'utf8',
        },
    },
}
```

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. It is not necessary if you don't plan to run tests, however.

Static Files

While we have not yet configured our proxy server, we need to configure the location that these files will be stored in. We will install these under `/var/www/patchwork`, though this is only a suggestion and can be changed.

```
$ sudo mkdir -p /var/www/patchwork
```

You can configure this by overriding the `STATIC_ROOT` variable with the below:

```
STATIC_ROOT = '/var/www/patchwork'
```

Other Options

Finally, the following settings need to be configured and the appropriate setting overridden. The purpose of many of these variables is described in *Configuration*.

- 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.ascii_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 API, you should add the following:

```
ENABLE_XMLRPC = True
```

Finally, should you wish to disable the REST API, you should add the following:

```
ENABLE_REST_API = False
```

Final Steps

Once done, we should be able to check that all requirements are met using the `check` command of the `manage.py` executable:

```
$ python3 manage.py check
```

We should also take this opportunity to both configure the database and static files:

```
$ python3 manage.py migrate
$ sudo python3 manage.py collectstatic
$ python3 manage.py loaddata default_tags default_states
```

Note: The above `default_tags` and `default_states` fixtures above are just that: defaults. You can modify these to fit your own requirements.

Finally, it may be helpful to start the development server quickly to ensure you can see *something*. For this to function, you will need to add the `ALLOWED_HOSTS` and `DEBUG` settings to your settings file.

```
ALLOWED_HOSTS = ['*']
DEBUG = True
```

Now, run the server.

```
$ python3 manage.py runserver 0.0.0.0:8000
```

Browse this instance at `http://[your_server_ip]:8000`. If everything is working, kill the development server using `Control-c` and remove `ALLOWED_HOSTS` and `DEBUG`.

Reverse Proxy and WSGI HTTP Servers

Install Packages

We will use *nginx* and *uWSGI* to deploy Patchwork, acting as reverse proxy server and WSGI HTTP server respectively. Other options are available, such as *Apache* with the *mod_wsgi* module, or *nginx* with the *Gunicorn* WSGI HTTP server. While we don't document these, sample configuration files for the former case are provided in *lib/apache2/*.

```
$ sudo apt-get install -y nginx-full uwsgi uwsgi-plugin-python3
```

Configure nginx and uWSGI

Configuration files for *nginx* and *uWSGI* are provided in the *lib* subdirectory of the Patchwork source code. These can be modified as necessary, but for now we will simply copy them.

First, let's load the provided configuration for *nginx*:

```
$ sudo cp /opt/patchwork/lib/nginx/patchwork.conf \
    /etc/nginx/sites-available/
```

If you wish to modify this configuration, now is the time to do so. Once done, validate and enable your configuration:

```
$ sudo ln -s /etc/nginx/sites-available/patchwork.conf \
    /etc/nginx/sites-enabled/patchwork.conf
$ sudo nginx -t
```

If you see a "duplicate default server" error message, You may need to disable the `default` application at this point:

```
$ sudo unlink /etc/nginx/sites-enabled/default
$ sudo nginx -t
```

Now, use the provided configuration for *uWSGI*:

```
$ sudo mkdir -p /etc/uwsgi/sites
$ sudo cp /opt/patchwork/lib/uwsgi/patchwork.ini \
    /etc/uwsgi/sites/patchwork.ini
```

Note: We created the `/etc/uwsgi` directory above because we're going to run *uWSGI* in emperor mode. This has benefits for multi-app deployments.

Create systemd Unit File

As things stand, *uWSGI* will need to be started manually every time the system boots, in addition to any time it may fail. We can automate this process using *systemd*. To this end a systemd unit file should be created to start *uWSGI* at boot:

```
$ sudo cat << EOF > /etc/systemd/system/uwsgi.service
[Unit]
Description=uWSGI Emperor service

[Service]
ExecStartPre=/bin/bash -c 'mkdir -p /run/uwsgi; chown www-data:www-data /run/uwsgi'
ExecStart=/usr/bin/uwsgi --emperor /etc/uwsgi/sites
Restart=always
KillSignal=SIGQUIT
Type=notify
NotifyAccess=all

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

You should also delete the default service file found in `/etc/init.d` to ensure the unit file defined above is used.

```
sudo rm /etc/init.d/uwsgi
sudo systemctl daemon-reload
```

Final Steps

Start the *uWSGI* service we created above:

```
$ sudo systemctl restart uwsgi
$ sudo systemctl status uwsgi
$ sudo systemctl enable uwsgi
```

Next up, restart the *nginx* service:

```
$ sudo systemctl restart nginx
$ sudo systemctl status nginx
$ sudo systemctl enable nginx
```

Finally, browse to the instance using your browser of choice. You may wish to take this opportunity to setup your projects and configure your website address (in the Sites section of the admin console, found at */admin*).

If there are issues with the instance, you can check the logs for *nginx* and *uWSGI*. There are a couple of commands listed below which can help:

- `sudo systemctl status uwsgi`, `sudo systemctl status nginx`

  To ensure the services have correctly started

- `sudo cat /var/log/nginx/error.log`

  To check for issues with *nginx*

- `sudo cat /var/log/patchwork.log`

  To check for issues with *uWSGI*. This is the default log location set by the `daemonize` setting in the *uWSGI* configuration file.

Django administrative console

In order to access the administrative console at */admin*, you need at least one user account to be registered and configured as a super user or staff account to access the Django administrative console. This can be achieved by doing the following:

```
$ python3 manage.py createsuperuser
```

Once the administrative console is accessible, you would want to configure your different sites and their corresponding domain names, which is required for the different emails sent by Patchwork (registration, password recovery) as well as the sample *pwclientrc* files provided by your project's page.

Incoming Email

Patchwork is designed to parse incoming mails which means you need an address to receive email at. This is a problem that has been solved for many web apps, thus there are many ways to go about this. Some of these ways are discussed below.

IMAP/POP3

The easiest option for getting mail into Patchwork is to use an existing email address in combination with a mail retriever like getmail, which will download mails from your inbox and pass them to Patchwork for processing. getmail is easy to set up and configure: to begin, you need to install it:

```
$ sudo apt-get install -y getmail4
```

Once installed, you should configure it, substituting your own configuration details where required below:

```
$ sudo cat << EOF > /etc/getmail/user@example.com/getmailrc
[retriever]
type = SimpleIMAPSSLRetriever
server = imap.example.com
port = 993
username = XXX
password = XXX
mailboxes = ALL

[destination]
# we configure Patchwork as a "mail delivery agent", in that it will
# handle our mails
type = MDA_external
path = /opt/patchwork/patchwork/bin/parsemail.sh

[options]
# retrieve only new emails
read_all = false
# do not add a Delivered-To: header field
delivered_to = false
# do not add a Received: header field
received = false
EOF
```

Validate that this works as expected by starting *getmail*:

```
$ getmail --getmaildir=/etc/getmail/user@example.com --idle INBOX
```

If everything works as expected, you can create a *systemd* script to ensure this starts on boot:

```
$ sudo cat << EOF > /etc/systemd/system/getmail.service
[Unit]
Description=Getmail for user@example.com

[Service]
User=nobody
ExecStart=/usr/bin/getmail --getmaildir=/etc/getmail/user@example.com --idle INBOX
Restart=always

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

And start the service:

```
$ sudo systemctl start getmail
$ sudo systemctl status getmail
$ sudo systemctl enable getmail
```

Mail Transfer Agent (MTA)

The most flexible option is to configure our own mail transfer agent (MTA) or "email server". There are many options, of which Postfix is one. While we don't cover setting up Postfix here (it's complicated and there are many guides already available), Patchwork does include a script to take received mails and create the relevant entries in Patchwork for you. To use this, you should configure your system to forward all emails to a given localpart (the bit before the @) to this script. Using the *patchwork* localpart (e.g. *patchwork@example.com*) you can do this like so:

```
$ sudo cat << EOF > /etc/aliases
patchwork: "|/opt/patchwork/patchwork/bin/parsemail.sh"
EOF
```

You should ensure the appropriate user is created in PostgreSQL and that it has (minimal) access to the database. Patchwork provides scripts for the latter and they can be loaded as seen below:

```
$ sudo -u postgres createuser nobody
$ sudo -u postgre psql -f \
    /opt/patchwork/lib/sql/grant-all.postgres.sql patchwork
```

Note: This assumes your Postfix process is running as the *nobody* user. If this is not correct (use of *postfix* user is also common), you should change both the username in the *createuser* command above and substitute the username in the *grant-all-postgres.sql* script with the appropriate alternative.

Use a Email-as-a-Service Provider

Setting up an email server can be a difficult task and, in the case of deployment on PaaS provider, may not even be an option. In this case, there are a variety of web services available that offer "Email-as-as-Service". These services typically convert received emails into HTTP POST requests to your endpoint of choice, allowing you to sidestep

configuration issues. We don't cover this here, but a simple wrapper script coupled with one of these services can be more than to get email into Patchwork.

You can also create such as service yourself using a PaaS provider that supports incoming mail and writing a little web app.

(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.

(Optional) Configure the Patchwork Cron Job

Patchwork can send notifications of patch changes. Patchwork uses a cron management command - `manage.py cron` - to send these notifications and to clean up expired registrations. To enable this functionality, add the following to your crontab:

```
# m h  dom mon dow   command
*/10 * * * * cd patchwork; python3 ./manage.py cron
```

Note: The frequency should be the same as the `NOTIFICATION_DELAY_MINUTES` setting, which defaults to 10 minutes. Refer to the *configuration guide* for mor information.

CHAPTER 7

Configuration

This document describes the various configuration options available in Patchwork. These options can be used for both *development* and *deployment* installations.

The `settings.py` File

Patchwork is a Django application and, as such, relies on Python-based settings files. Refer to the Django documentation for more information on the general format.

Patchwork provides three settings files:

`base.py`

A base settings file that should not be used directly.

`dev.py`

A settings file for development use. **This file is horribly insecure and must not be used in production**.

`production.example.py`

A sample settings file for production use. This will likely require some heavy customization. The *deployment guide* provides more information.

Patchwork-specific Settings

Patchwork utilizes a number of Patchwork-only settings in addition to the Django and Django REST Framework settings.

`DEFAULT_ITEMS_PER_PAGE`

The default number of items to display in the list pages for a project (`/project/{projectID}/list`) or bundle (`/bundle/{userID}/{bundleName}`).

This is customizable on a per-user basis from the user configuration page.

Changed in version 2.0: This option was previously named `DEFAULT_PATCHES_PER_PAGE`. It was renamed as cover letters are now supported also.

CONFIRMATION_VALIDITY_DAYS

The number of days to consider an account confirmation request valid. After this interval, the *cron management command* will delete the request.

NOTIFICATION_DELAY_MINUTES

The number of minutes to wait before sending any notifications to a user. An notification generated during this time are gathered into a single digest email, ensuring users are not spammed with emails from Patchwork.

NOTIFICATION_FROM_EMAIL

The email address that notification emails should be sent from.

ENABLE_XMLRPC

Enable the *XML-RPC API*.

ENABLE_REST_API

Enable the *REST API*.

New in version 2.0.

REST_RESULTS_PER_PAGE

The number of items to include in REST API responses by default. This can be overridden by the `per_page` parameter for some endpoints.

New in version 2.0.

COMPAT_REDIR

Enable redirections of URLs from previous versions of Patchwork.

FORCE_HTTPS_LINKS

Force use of `https://` links instead of guessing the scheme based on current access. This is useful if SSL protocol is terminated upstream of the server (e.g. at the load balancer)

CHAPTER 8

Management

This document describes the myriad administrative commands available with Patchwork. Many of these commands are referenced in the *development* and *deployment* installation guides.

The `manage.py` Script

Django provides the `django-admin` command-line utility for interacting with Django applications and projects, as described in the Django documentation. Patchwork, being a Django application, provides a wrapper for this command - `manage.py` - that exposes not only the management commands of Django and its default applications, but also a number of custom, Patchwork-only management commands.

An overview of the Patchwork-specific commands is provided below. For information on the commands provided by Django itself, refer to the Django documentation. Information on any command can also be found by passing the `--help` parameter:

```
./manage.py cron --help
```

Available Commands

cron

```
./manage.py cron
```

Run periodic Patchwork functions: send notifications and expire unused users.

This is required to ensure notifications emails are actually sent to users that request them and is helpful to expire unused users created by spambots. For more information on integration of this script, refer to the *deployment installation guide*.

parsearchive

Parse an mbox archive file and store any patches/comments found.

```
./manage.py parsearchive [--list-id <list-id>] <infile>
```

This is mostly useful for development or for adding message that were missed due to, for example, an outage.

--list-id <list-id>
: mailing list ID. If not supplied, this will be extracted from the mail headers.

infile
: input mbox filename

parsemail

Parse an mbox file and store any patch/comment found.

```
./manage.py parsemail [--list-id <list-id>] <infile>
```

This is the main script used to get mails (and therefore patches) into Patchwork. It is generally used by the `parsemail.sh` script in combination with a mail transfer agent (MTA) like Postfix. For more information, refer to the *deployment installation guide*.

--list-id <list-id>
: mailing list ID. If not supplied, this will be extracted from the mail headers.

infile
: input mbox filename. If not supplied, a patch will be read from `stdin`.

rehash

Update the hashes on existing patches.

```
./manage.py rehash [<patch_id>, ...]
```

Patchwork stores hashes for each patch it receives. These hashes can be used to uniquely identify a patch for things like *automatically changing the state of the patch in Patchwork when it merges*. If you change your hashing algorithm, you may wish to rehash the patches.

patch_id
: a patch ID number. If not supplied, all patches will be updated.

retag

Update the tag (Ack/Review/Test) counts on existing patches.

```
./manage.py retag [<patch_id>...]
```

Patchwork extracts *tags* from each patch it receives. By default, three tags are extracted, but it's possible to change this on a per-instance basis. Should you add additional tags, you may wish to scan older patches for these new tags.

patch_id
: a patch ID number. If not supplied, all patches will be updated.

CHAPTER 9

Upgrading

This document provides some general tips and tricks that one can use when upgrading an existing, production installation of Patchwork. If you are interested in the specific changes between each release, refer to the *UPGRADING* document instead. If this is your first time installing Patchwork, refer to the *Installation* instead.

Before You Start

Before doing anything, always **backup your data**. This generally means backing up your database, but it might also be a good idea to backup your environment in case you encounter issues during the upgrade process.

While Patchwork won't explicitly prevent it, it's generally wise to avoid upgrades spanning multiple releases in one go. An iterative upgrade approach will provide an easier, if slower, upgrade process.

Identify Changed Scripts, Requirements, etc.

The *CHANGELOG* document provides a comprehensive listing of all backwards-incompatible changes that occur between releases of Patchwork. Examples of such changes include:

- Moved/removed scripts and files
- Changes to the requirements, e.g. supported Django versions
- Changes to API that may affect, for example, third-party tools

It is important that you understand these changes and ensure any scripts you may have, such as systemd scripts, are modified accordingly.

Understand What Requirements Have Changed

New versions of Patchwork can often require additional or updated version of dependencies, e.g. newer versions of Django. It is important that you understand these requirements and can fulfil them. This is particularly true for users

relying on distro-provided packages, who may have to deal with older versions of a package or may be missing a package altogether (though we try to avoid this). Such changes are usually listed in the *UPGRADING* document, but you can also diff the *requirements.txt* files in each release for comparison.

Collect Static Files

New versions of Patchwork generally contain changes to the additional files like images, CSS and JavaScript. To do this, run the *collectstatic* management commands:

```
$ ./manage.py collectstatic
```

Upgrade Your Database

Migrations of the database can be tricky. Prior to v1.0.0, database migrations were provided by way of manual, SQL migration scripts. After this release, Patchwork moved to support Django migrations. If you are upgrading from *v1.0.0* or later, it is likely that you can rely entirely on the later to bring your database up-to-date. This can be done like so:

```
$ ./manage.py migrate
```

However, there are a number of scenarios in which you may need to fall back to the provided SQL migrations or provide your own:

- You are using Django < 1.6

  Patchwork supports Django 1.6. However, Django Migrations was added in 1.7 and is not available for previous versions. As such, you must continue to use manual migrations or upgrade your version of Django. For many of the migrations, this can be done automatically:

  ```
  $ ./manage.py sqlmigrate patchwork 0003_add_check_model
  ```

  However, this only works for schema migrations. For data migrations, however, this will fail. In this cases, these migrations will need to be handwritten.

- You are using Django > 1.6, but upgrading from Patchwork < 1.0.0

  Patchwork only started providing migrations in *v1.0.0*. SQL migrations are provided for versions prior to this and must be applied to get the database to the "initial" state that Django migrations expects.

- You have diverged from upstream Patchwork

  If you have applied custom patches that change the database models, the database in an "inconsistent state" and the provided migrations will likely fail to apply.

Steps to handle the latter two of these are described below.

Upgrading a pre-v1.0.0 Patchwork instance

The process for this type of upgrade is quite simple: upgrade using manual SQL upgrades until better options become available. As such, you should apply all unapplied SQL migrations that are not duplicated by Django migrations. Once such duplication occurs, rely on the Django migrations only and continue to do so going forward.

Upgrading a "diverged" Patchwork instance

This type of upgrade is a little trickier. There are two options you can take:

1. Bring your Patchwork instance back in sync with upstream
2. Provide your own migrations

The former option is particularly suitable if you decide to upstream your change or decide it's not valuable enough to retain. This will require either reworking any migrations that exist prior to your feature being upstreamed, or deleting any added database fields and tables, respectively. In both cases, manually, hand-written SQL migrations will be required to get the databse into a consistent state (remember: **backup**!). Once this is done, you can resume using the upstream-provided migrations, ensuring any Django migrations that you may have skipped are not applied again:

```
$ ./manage.py migrate 000x-abc --fake  # when 000x-abc is last "skippable"
```

It's worth adding that with the databases now back in sync it should be possible to return to using upstream code rather than maintaining a fork.

The latter option is best chosen if you wish to retain the aforementioned fork. How you do this depends on the extensiveness of your changes, but getting the latest version of Patchwork, deleting the provided migrations, applying any patches you may have and regenerating the migrations seems like the best option.

Note: To prevent the latter case above from occurring, we'd ask that you submit any patches you may have to the upstream Patchwork so that the wider community can benefit from this new functionality.

CHAPTER 10

Contributing

Coding Standards

Follow PEP8. All code is currently PEP8 compliant and it should stay this way.

Changes that fix semantic issues will be generally be happily received, but please keep such changes separate from functional changes.

pep8 targets are provided via tox. Refer to the *Testing* section below for more information on usage of this tool.

Testing

Patchwork includes a tox script to automate testing. This requires a functional database and some Python requirements like *tox*. Refer to *Installation* for information on how to configure these.

You may also need to install *tox*. If so, do this now:

```
$ sudo pip install tox
```

Tip: If you're using Docker or Vagrant-based installs, you may not need to install *tox* locally. Instead, it will already be installed inside the container/VM. For Docker, you can run *tox* like so:

```
$ docker-compose run web tox [ARGS...]
```

For Vagrant, SSH into the container and run *tox* as below.

Assuming these requirements are met, actually testing Patchwork is quite easy to do. To start, you can show the default 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 one of 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 support multiple versions of Django, it's very important that you test against all supported versions. When run without argument, tox will do this:

```
$ tox
```

Release Notes

Patchwork uses reno for release note management. To use *reno*, you must first install it:

```
$ sudo pip install tox
```

Once installed, a new release note can be created using the `reno new` command:

```
$ reno new <slugified-summary-of-change>
```

Modify the created file, removing any irrelevant sections, and include the modified file in your change.

Submitting Changes

All patches should be sent to the mailing list. When doing so, please abide by the QEMU guidelines on contributing or submitting patches. This covers both the initial submission and any follow up to the patches. In particular, ensure:

- *All tests pass*
- Documentation has been updated with new requirements, new script names etc.
- *A release note is included*

CHAPTER 11

Installation

This document describes the necessary steps to configure Patchwork in a development environment. If you are interested in deploying Patchwork in a production environment, refer to the deployment guide instead.

To begin, you should clone Patchwork:

```
$ git clone git://github.com/getpatchwork/patchwork.git
```

Docker-Based Installation

Patchwork provides a Docker-based environment for quick configuration of a development environment. This is the preferred installation method. To configure Patchwork using Docker:

1. Install docker and docker-compose.
2. Build the images. This will download over 200MB from the internet:

   ```
   $ docker-compose build
   ```

3. Run *docker-compose up*:

   ```
   $ docker-compose up
   ```

   This will be visible at http://localhost:8000/.

To run a shell within this environment, run:

```
$ docker-compose run --rm web --shell
```

To run *django-manage* commands, such as *createsuperuser* or *migrate*, run:

```
$ docker-compose run --rm web python manage.py createsuperuser
```

To access the SQL command-line client, run:

```
$ docker-compose run --rm web python manage.py dbshell
```

To run unit tests, excluding Selenium UI interaction tests, using only the package versions installed during container initialization, run:

```
$ docker-compose run --rm web --quick-test
```

To run the same against all supported versions of Django (via tox), run:

```
$ docker-compose run --rm web --quick-tox
```

To run specific tox targets or tests, pass arguments to the above:

```
$ docker-compose run --rm web --quick-tox -e py27-django17 \
    patchwork.tests.test_bundles
```

To run all tests, including Selenium UI interaction tests, using only the package versions installed container initialization, run:

```
$ docker-compose run --rm web --test
```

To run the same against all supported versions of Django (via tox), run:

```
$ docker-compose run --rm web --tox
```

To run all tests, including Selenium UI interaction tests in non-headless mode, run:

```
$ docker run -it --rm -v (pwd):/home/patchwork/patchwork/ \
    --link patchwork_db_1:db -p 8000:8000 \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    -e PW_TEST_DB_HOST=db -e DISPLAY patchwork_web bash
```

To reset the database before any of these commands, add *–reset* to the command line after *web* and before any other arguments.

Any local edits to the project files made locally are immediately visible to the Docker container, and so should be picked up by the Django auto-reloader.

For more information on Docker itself, please refer to the docker and docker-compose documentation.

Note: If using SELinux, you will need to create a custom SELinux rule to allow the Docker process to access your working directory. Run:

```
$ chcon -RT svirt_sandbox_file_t $PATCHWORK_DIR
```

where *$PATCHWORK_DIR* is the absolute patch to the *patchwork* folder created when you cloned the repo. For more information, see *man docker run*.

Note: If you see an error like the below:

```
ERROR: Couldn't connect to the Docker daemon at http+docker://localunixsocket - is it␣
↪running?
```

ensure you have correctly installed Docker, added your user to the *docker* group, and started the daemon, per the docker documentation.

Note: If you see an error like the below:

```
py.error.EACCES: [Permission denied]: open('/home/patchwork/patchwork/.tox/py27-
↪django18/.tox-config1', 'w')
```

your host user account is likely using a different UID to the one hardcoded in the Dockerfile. You can confirm this like so:

```
$ echo $UID
1234
```

If this is anything other than *1000*, you must must modify the *Dockerfile* found in *tools/docker* to use your UID and then rebuild:

```
$ sed -i "/ARG UID=/c\ARG UID=$(echo $UID)" tools/docker/Dockerfile
$ docker-compose build web
```

This change must be retained in the event that you rebuild the container. You can "hide" the change from Git like so:

```
$ git update-index --assume-unchanged tools/docker/Dockerfile
$ git update-index --skip-worktree tools/docker/Dockerfile
```

This should be resolved in a future release when we support docker-compose 2.1 syntax in *docker-compose.yml*.

Vagrant-Based Installation

Patchwork provides a Vagrant-based environment as an alternative to Docker. Like Docker, Vagrant can be used to quickly configure Patchwork in a development environment. To configure Patchwork using Vagrant:

1. Install **Vagrant**
2. Run *vagrant up* from the project directory:

   ```
   $ cd patchwork
   $ vagrant up
   ```

Once stacked, follow the on-screen instructions. For more information on Vagrant itself, refer to the Vagrant documentation.

Manual Installation

Manual installation can be used where use of Docker or Vagrant is not possible or desired.

Install Required Packages

There are a number of different requirements for developing Patchwork:

- Python and libraries
- A supported database (RDBMS)

These are detailed below.

Python Requirements

To develop Python-based software you first need Python. Patchwork supports both Python 2.7 and Python 3.3+. One of these will be installed by default on many installations, though they can also be installed manually using the *python* or *python3* packages.

It's a good idea to use virtual environments 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 possibly deploy Patchwork against a "well known" set of dependencies, but they can also be used to test Patchwork against several versions of Django.

If you do not have *virtualenv* installed then you should install it now. This can be installed using the *python-virtualenv* or *python3-virtualenv* packages. Alternatively you can install these using *pip*.

It is also helpful to install *tox* which is used for running tests in Patchwork. This can be installed using the *python-tox* or *python3-tox* packages, or via *pip*.

Database Requirements

If not already installed, you may need to install an RDBMS. You can use either MariaDB/MySQL or PostgreSQL for this purpose. You should also install the development headers, known as *libmysqlclient-dev* or *libpq-dev* respectively on Debian-based Debian-based distros like Ubuntu and *mysql-devel* or *postgresql-devel* on RHEL-based distros.

Note: While Django provides support for multiple database backends, Patchwork itself is only tested against MySQL/MariaDB and PostgreSQL. Should you wish to use a different backend, ensure you validate this first (and perhaps upstream any changes you may find necessary).

Note: You may be tempted to use SQLite to develop Patchwork. We'd advise against doing this. SQLite supports a subset of the functionality of "full" RDBMS like MySQL: for example, case-sensitive matching of Unicode is not supported. You will find some tests provided by Patchwork fail and some patches you develop may fail in production due to these differences.

Example Installation

An example for installing all these packages and the MySQL RDBMS on Ubuntu 15.04 is given below:

```
$ sudo apt-get install python python-pip python-dev python-virtualenv \
    python-tox mysql-server libmysqlclient-dev
```

If you have an existing MariaDB/MySQL installation and have installed *pip* already/are using Python 3.4+ then you can install all packages using *pip*:

```
$ sudo pip install virtualenv tox
```

If you wish to use Python 3 then simply replace *python* with *python3* in the above command.

Configure Virtual Environment

Note: If you are interested in simply *testing Patchwork*, many of the below steps are not required. tox will automatically install dependencies and use virtual environments when testing.

Once these requirements are installed, you should create and activate a new virtual environment. This can be done like so:

```
$ virtualenv .venv
$ source .venv/bin/activate
(.venv)$
```

Note: If you installed a Python 3.x-based virtual environment package, adjust the executable indicated above as necessary, e.g. *virtualenv-3.4*.

Now install the packages. Patchwork provides three requirements files.

requirements-dev.txt

Packages required to configure a development environment

requirements-prod.txt

Packages required for deploying Patchwork in production

requirements-test.txt

Packages required to run tests

We're going to install the first of these, which can be done like so:

```
(.venv)$ cd patchwork
(.venv)$ pip install -r requirements-dev.txt
```

Note: Once configured this does not need to be done again *unless* the requirements change, e.g. Patchwork requires an updated version of Django.

Initialize the Database

One installed, the database must be configured. We will assume you have root access to the database for these steps.

To begin, export your database credentials as follows:

```
(.venv)$ db_user=root
(.venv)$ db_pass=password
```

Now, create the database. If this is your first time configuring the database, you must create a *patchwork* user (or similar) along with the database instance itself. The commands below will do this, dropping existing databases if necessary:

```
(.venv)$ mysql -u$db_user -p$db_pass << EOF
DROP DATABASE IF EXISTS patchwork;
CREATE DATABASE patchwork CHARACTER SET utf8;
GRANT ALL PRIVILEGES ON patchwork.* TO 'patchwork'@'localhost'
    IDENTIFIED BY 'password';
EOF
```

Note: The *patchwork* username and *password* password are the defaults expected by the provided *dev* settings files. If using something different, export the *PW_TEST_DB_USER* and *PW_TEST_DB_PASS* variables described in the *Environment Variables* section below. Alternatively, you can create your own settings file with these variables hardcoded and change the value of *DJANGO_SETTINGS_MODULE* as described below.

Load Initial Data

Before continuing, we need to tell Django where it can find our configuration. Patchwork provides a default development *settings.py* file for this purpose. To use this, export the *DJANGO_SETTINGS_MODULE* environment variable as described below:

```
(.venv)$ export DJANGO_SETTINGS_MODULE=patchwork.settings.dev
```

Alternatively you can provide your own *settings.py* file and provide the path to that instead.

Once done, we need to create the tables in the database. This can be done using the *migrate* command of the *manage.py* executable:

```
(.venv)$ ./manage.py migrate
```

Next, you should load the initial fixtures into Patchwork. These initial fixtures provide.

default_tags.xml

The tags that Patchwork will extract from mails. For example: *Acked-By*, *Reviewed-By*

default_states.xml

The states that a patch can be in. For example: *Accepted*, *Rejected*

default_projects.xml

A default project that you can then upload patches for

These can be loaded using the *loaddata* command:

```
(.venv)$ ./manage.py loaddata default_tags default_states default_projects
```

You should also take the opportunity to create a "superuser". You can do this using the aptly-named *createsuperuser* command:

```
(.venv)$ ./manage.py createsuperuser
```

Import Mailing List Archives

Regardless of your installation method of choice, you will probably want to load some real emails into the system. This can be done manually, however it's generally much easier to download an archive from a Mailman instance and load these using the *parsearchive* command. You can do this like so:

```
(.venv)$ mm_user=<myusername>
(.venv)$ mm_pass=<mypassword>
(.venv)$ mm_host=https://lists.ozlabs.org
(.venv)$ mm_url=$mm_host/private/patchwork.mbox/patchwork.mbox
(.venv)$ curl -F username=$mm_user -F password=$mm_pass -k -O $mm_url
```

where *mm_user* and *mm_pass* are the username and password you have registered with on the Mailman instance found at *mm_host*.

Note: We provide instructions for downloading archives from the Patchwork mailing list, but almost any instance of Mailman will allow downloading of archives as seen above; simply change the *pw_url* variable defined. You can find more informations about this here.

Load these archives into Patchwork. Depending on the size of the downloaded archives this may take some time:

```
(.venv)$ ./manage.py parsearchive --list-id=patchwork.ozlabs.org \
   patchwork.mbox
```

Finally, run the server and browse to the IP address of your board using your browser of choice:

```
(.venv)$ ./manage.py runserver 0.0.0.0:8000
```

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

```
(.venv)$ deactivate
$
```

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

Django Debug Toolbar

Patchwork installs and enables the 'Django Debug Toolbar' by default. However, by default this is only displayed if you are developing on localhost. If developing on a different machine, you should configure an SSH tunnel such that, for example, *localhost:8000* points to *[DEV_MACHINE_IP]:8000*.

Environment Variables

The following environment variables are available to configure settings when using the provided *dev* settings file.

PW_TEST_DB_NAME=patchwork

Name of the database

PW_TEST_DB_USER=patchwork

Username to access the database with

PW_TEST_DB_PASS=password

Password to access the database with<

PW_TEST_DB_TYPE=mysql

Type of database to use. Options: 'mysql', 'postgres'

CHAPTER 12

Release Process

Versioning

Since version 1.0, Patchwork has implemented a version of Semantic Versioning . To summarise, releases take the format **MAJOR.MINOR.PATCH** (or just **MAJOR.MINOR**). We increment:

1. **MAJOR** version when we make major UI changes or functionality updates
2. **MINOR** version when we make minor UI changes or functionality updates
3. **PATCH** version when we make make bug fixes, dependency updates etc.

In Git, each release will have a tag indicating the version number. In addition, each release series has it's own branch called *stable/MAJOR.MINOR* to allow backporting of bugfixes or security updates to older versions.

Release Cycle

There is no cadence for releases: they are made available as necessary.

Supported Versions

Typically all development should occur on *master*. While we will backport bugfixes and security updates, we will not backport any new features. This is to ensure stability for users of these versions of Patchwork.

Release Checklist

- Documentation has been updated with latest release version
- Documentation references latest supported version of Django

- 'alpha' tag has been removed from `__version__` in `patchwork/__init__.py`
- Commit has been tagged with an annotated tag. The tag should take the form *v[MAJOR].[MINOR].[PATCH]*, e.g. *v2.0.1*. The message should read:

```
Version [MAJOR].[MINOR].[PATCH]
```

- A GitHub Release, with text corresponding to an abbreviated form of the release notes for that cycle, has been created

The following only apply to full releases, or those where the *MAJOR* or *MINOR* number is incremented:

- A new branch called *stable/MAJOR.MINOR* has been created from the tagged commit

Once released, bump the version found in `patchwork/__init__.py` once again.

Backporting

We will occasionally backport bugfixes and security updates. When backporting a patch, said patch should first be merged into *master*. Once merged, you can backport by cherry-picking commits, using the `-x` flag for posterity:

```
$ git cherry-pick -x <master_commit>
```

There may be some conflicts; resolve these, uncommenting the *Conflicts* line when committing:

```
Conflicts
        patchwork/bin/pwclient
```

When enough patches have been backported, you should release a new *PATCH* release.

CHAPTER 13

Using the APIs

Patchwork provides two APIs: the legacy *XML-RPC API* and the *REST API*. You can use these APIs to interact with Patchwork programmatically and to develop your own clients.

For quick usage examples of the APIs, refer to the *documentation*. For examples of existing clients, refer to *Clients*.

CHAPTER 14

Static Assets

Patchwork relies on a number of third-party JavaScript libraries. These, along with their supporting assets and the Patchwork-only libraries and assets, are described below.

css

`bootstrap.min.css`

CSS for the *Bootstrap* library.

Refer to the *js* section below for more information on *Bootstrap*.

`selectize.bootstrap3.css`

CSS for the *Selectize* library.

Refer to the *js* section below for more information on *Selectize*.

`style.css`

Custom, Patchwork styling. Mostly a collection of overrides for default Bootstrap styles.

Part of Patchwork.

fonts

`glyphicons-halflings-regular.*`

Library of precisely prepared monochromatic icons and symbols, created with an emphasis to simplicity and easy orientation. Provided as part of the Bootstrap library.

These are in multiple formats to support different browsers/environments. Refer to the *js* section below for more information on Bootstrap.

js

`bootstrap.js`

The most popular HTML, CSS, and JavaScript framework for developing responsive, mobile first projects on the web.

This is used for the main UI of Patchwork.

Website https://getbootstrap.com/

GitHub https://github.com/twbs/bootstrap/

Version 3.2.0

`bundle.js`

Utility functions for bundle patch list manipulation (re-ordering patches, etc.)

Part of Patchwork.

`clipboard.min.js`

Modern copy to clipboard. No Flash. Just 3kb gzipped

This is used to allow us to "click to copy" various elements in the UI.

Website https://clipboardjs.com/

GitHub https://github.com/zenorocha/clipboard.js/

Version 1.7.1

`jquery.js`

jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

This is used across Patchwork, including by the likes of `bundle.js`, as well as by the various plugins below.

Website https://jquery.com/

GitHub https://github.com/jquery/jquery

Version 1.10.1

`jquery.checkboxes.js`

A jQuery plugin that gives you nice powers over your checkboxes.

This is used to allow shift-select of checkboxes on the patch list page.

Website http://rmariuzzo.github.io/checkboxes.js

GitHub https://github.com/rmariuzzo/checkboxes.js

Version 1.0.6

`jquery.stickytableheaders.js`

A jQuery plugin that makes large tables more usable by having the table header stick to the top of the screen when scrolling.

This is used to ensure the heads on the patch list page stay at the top as we scroll.

GitHub https://github.com/jmosbech/StickyTableHeaders

Version 0.1.19

`jquery.tablednd.js`

jQuery plug-in to drag and drop rows in HTML tables.

This is used by the bundle patch list to allow us to control the order of the patches in said bundle.

Website http://www.isocra.com/2008/02/table-drag-and-drop-jquery-plugin/

GitHub jQuery plug-in to drag and drop rows in HTML tables

Version ???

`selectize.min.js`

Selectize is the hybrid of a `textbox` and `<select>` box. It's jQuery based and it has autocomplete and native-feeling keyboard navigation; useful for tagging, contact lists, etc.

Website https://selectize.github.io/selectize.js/

GitHub https://github.com/selectize/selectize.js

Version 0.11.2

CHAPTER 15

The REST API

Patchwork provides a REST API. This API can be used to retrieve and modify information about patches, projects and more.

This guide provides an overview of how one can interact with the REST API. For detailed information on type and response format of the various resources exposed by the API, refer to the web browsable API. This can be found at:

> https://patchwork.example.com/api/1.0/

where *patchwork.example.com* refers to the URL of your Patchwork instance.

Important: The REST API can be enabled/disabled by the administrator: it may not be available in every instance. Refer to `/about` on your given instance for the status of the API, e.g.

> https://patchwork.ozlabs.org/about

New in version 2.0: The REST API was introduced in Patchwork v2.0. Users of earlier Patchwork versions should instead refer to *XML-RPC API* documentation.

Getting Started

The easiest way to start experimenting with the API is to use the web browsable API, as described above.

REST APIs run over plain HTTP(S), thus, the API can be interfaced using applications or libraries that support this widespread protocol. One such application is curl, which can be used to both retrieve and send information to the REST API. For example, to get the version of the REST API for a Patchwork instance hosted at *patchwork.example.com*, run:

```
$ curl -s 'https://patchwork.example.com/api/1.0/' | python -m json.tool
{
    "bundles": "https://patchwork.example.com/api/1.0/bundles/",
    "covers": "https://patchwork.example.com/api/1.0/covers/",
    "events": "https://patchwork.example.com/api/1.0/events/",
```

```
    "patches": "https://patchwork.example.com/api/1.0/patches/",
    "people": "https://patchwork.example.com/api/1.0/people/",
    "projects": "https://patchwork.example.com/api/1.0/projects/",
    "series": "https://patchwork.example.com/api/1.0/series/",
    "users": "https://patchwork.example.com/api/1.0/users/"
}
```

In addition, a huge variety of libraries are available for interacting with and parsing the output of REST APIs. The requests library is wide-spread and well-supported. To repeat the above example using *requests*:, run

```
$ python
>>> import json
>>> import requests
>>> r = requests.get('https://patchwork.example.com/api/1.0/')
>>> print(json.dumps(r.json(), indent=2))
{
    "bundles": "https://patchwork.example.com/api/1.0/bundles/",
    "covers": "https://patchwork.example.com/api/1.0/covers/",
    "events": "https://patchwork.example.com/api/1.0/events/",
    "patches": "https://patchwork.example.com/api/1.0/patches/",
    "people": "https://patchwork.example.com/api/1.0/people/",
    "projects": "https://patchwork.example.com/api/1.0/projects/",
    "series": "https://patchwork.example.com/api/1.0/series/",
    "users": "https://patchwork.example.com/api/1.0/users/"
}
```

Tools like *curl* and libraries like *requests* can be used to build anything from small utilities to full-fledged clients targeting the REST API. For an overview of existing API clients, refer to *Clients*.

Tip: While you can do a lot with existing installations, it's possible that you might not have access to all resources or may not wish to modify any existing resources. In this case, it might be better to *deploy your own instance of Patchwork locally* and experiment with that instead.

Versioning

By default, all requests will receive the latest version of the API: currently `1.0`:

```
GET /api HTTP/1.1
```

You should explicitly request this version through the URL to prevent API changes breaking your application:

```
GET /api/1.0 HTTP/1.1
```

Schema

Responses are returned as JSON. Blank fields are returned as `null`, rather than being omitted. Timestamps use the ISO 8601 format:

```
YYYY-MM-DDTHH:MM:SSZ
```

Requests should use either query parameters or form-data, depending on the method. Further information is provided below.

Summary Representations

Some resources are particularly large or expensive to compute. When listing these resources, a summary representation is returned that omits certain fields. To get all fields, fetch the detailed representation. For example, listing patches will return summary representations for each patch:

```
GET /patches HTTP/1.1
```

Detailed Representations

When fetching an individual resource, all fields will be returned. For example, fetching a patch with an ID of `123` will return all available fields for that particular resource:

```
GET /patches/123 HTTP/1.1
```

Parameters

Most API methods take optional parameters. For `GET` requests, these parameters are mostly used for filtering and should be passed as a HTTP query string parameters:

```
$ curl 'https://patchwork.example.com/api/patches?state=under-review'
```

For all other types of requests, including `POST` and `PATCH`, these parameters should be passed as form-encoded data:

```
$ curl -X PATCH -F 'state=under-review' \
  'https://patchwork.example.com/api/patches/123'
```

Authentication

Patchwork supports authentication using your username and password (basic authentication) or with a token (token authentication). The latter is recommended.

To authenticate with token authentication, you must first obtain a token. This can be done from your profile, e.g. https://patchwork.example.com/profile. Once you have a token, run:

```
$ curl -H "Authorization: Token ${token}" \
    'https://patchwork.example.com/api/'
```

To authenticate using basic auth, you should use your Patchwork username and password. To do this, run:

```
$ curl -u ${username}:${password} \
    'https://patchwork.example.com/api/'
```

Not all resources require authentication. Those that do will return `404 (Not Found)` if authentication is not provided to avoid leaking information.

Pagination

Requests that return multiple items will be paginated by 30 items by default, though this can vary from instance to instance. You can change page using the `?page` parameter. You can also set custom page sizes up to 100 on most endpoints using the `?per_page` parameter.

```
$ curl 'https://patchwork.example.com/api/patches?page=2&per_page=100'
```

Link Header

The Link header includes pagination information:

```
Link: <https://patchwork.example.com/api/patches?page=3&per_page=100>; rel="next",
  <https://patchwork.example.com/api/patches?page=50&per_page=100>; rel="last"
```

The possible `rel` values are:

Name	Description
`next`	The link relation for the immediate next page of results.
`last`	The link relation for the last page of results.
`first`	The link relation for the first page of results.
`prev`	The link relation for the immediate previous page of results.

CHAPTER 16

The XML-RPC API

Patchwork provides an XML-RPC API. This API can be used to be used to retrieve and modify information about patches, projects and more.

Important: The XML-RPC API can be enabled/disabled by the administrator: it may not be available in every instance. Refer to `/about` on your given instance for the status of the API, e.g.

https://patchwork.ozlabs.org/about

Alternatively, simply attempt to make a request to the API.

Deprecated since version 2.0: The XML-RPC API is a legacy API and has been deprecated in favour of the *REST API*. It will be removed in Patchwork 3.0.

Getting Started

The Patchwork XML-RPC API provides a number of "methods". Some methods require authentication (via HTTP Basic Auth) while others do not. Authentication uses your Patchwork account and the on-server documentation will indicate where it is necessary. We will only cover the unauthenticated method here for brevity - consult the xmlrpclib documentation for more detailed examples:

To interact with the Patchwork XML-RPC API, a XML-RPC library should be used. Python provides such a library - xmlrpclib - in its standard library. For example, to get the version of the XML-RPC API for a Patchwork instance hosted at *patchwork.example.com*, run:

```
$ python
>>> import xmlrpclib  # or 'xmlrpc.client' for Python 3
>>> rpc = xmlrpclib.ServerProxy('http://patchwork.example.com/xmlrpc/')
>>> rpc.pw_rpc_version()
1.1
```

Once connected, the `rpc` object will be populated with a list of available functions (or procedures, in RPC terminology). In the above example, we used the `pw_rpc_version` method, however, it should be possible to use all the methods listed in the server documentation.

Further Information

Patchwork provides automatically generated documentation for the XML-RPC API. You can find this at the following URL:

> https://patchwork.example.com/xmlrpc/

where *patchwork.example.com* refers to the URL of your Patchwork instance.

Changed in version 1.1: Automatic documentation generation for the Patchwork API was introduced in Patchwork v1.1. Prior versions of Patchwork do not offer this functionality.

CHAPTER 17

Unreleased

CHAPTER 18

v2.0 Series (“Dazzle”)

v2.0.0

Prelude

The v2.0.0 release includes many new features and bug fixes. For full information on the options avaiable, you should look at the full release notes in detail. However, there are two key features that make v2.0.0 a worthwhile upgrade:

- A REST API is now provided, which will eventually replace the legacy XML-RPC API
- Patch series and series cover letters are now supported

For further information on these features and the other changes in this release, review the full release notes.

New Features

- REST API.

  Previous versions of Patchwork provided an XML-RPC API. This was functional but there were a couple of issues around usability and general design. This API also provided basic versioning information but the existing clients, mostly *pwclient* variants, did not validate this version. Together, this left us with an API that needed work but no way to fix it without breaking every client out there.

  Rather than breaking all those users, make a clean break and provide another API method. REST APIs are the API method de jour providing a number of advantages over XML-RPC APIs, thus, a REST API is chosen. The following resources are exposed over this new API:

  – Bundles
  – Checks
  – Projects
  – People
  – Users

  - Patches
  - Series
  - Cover letters

  For information on the usage of the API, refer to the documentation.

- Cover letters are now supported.

  Cover letters are often sent in addition to a series of patches. They do not contain a diff and can generally be identified as number 0 of a series. For example:

```
[PATCH 0/3] A cover letter
```

  Cover letters contain useful information that should not be discarded. Both cover letters and replies to these mails are now stored for use with series.

- Series are now supported.

  Series are groups of patches sent as one bundle. For example:

```
[PATCH 0/3] A cover letter
  [PATCH 1/3] The first patch
  [PATCH 2/3] The second patch
  [PATCH 3/3] The third patch
```

  While Patchwork already supports bundles, these must be created manually, defeating the purpose of using series in the first place. Series make use of the information provided in the emails themselves, avoiding this manual step. The series support implemented is basic and does not support versioning. This will be added in a future release.

- All comments now have a permalink which can be used to reference individual replies to patches and cover letters.
- Django Debug Toolbar is now enabled by defaut when using development settings.
- Django 1.9 and 1.10 are now supported.
- Python 3.5 is now supported.
- Docker support is now integrated for development usage. To use this, refer to the documentation.
- Series markers are now parsed from patches generated by the Mercurial Patchbomb extension.

Upgrade Notes

- The REST API is enabled by default.

  The REST API is enabled by default. It is possible to disable this API, though this functionality may be removed in a future release. Should you wish to disable this feature, configure the `ENABLE_REST_API` setting to `False`.

- The `parsemail.py` and `parsearchive.py` scripts have been replaced by the `parsemail` and `parsearchive` management commands. These can be called like any other management commands. For example:

```
$ ./manage.py parsemail [args...]
```

- The `DEFAULT_PATCHES_PER_PAGE` has been renamed as `DEFAULT_ITEMS_PER_PAGE` as it is now possible to list cover letters in addition to patches.

- The `context` field for patch checks must now be slug, or a string consisting of only ASCII letters, numbers, underscores or hyphens. While older, non-slugified strings won't cause issues, any scripts creating contexts must be updated where necessary.

Bug Fixes

- When downloading an mbox, a user's name will now be set to the name used in the last email recieved from them. Previously, the name used in the first email received from a user was used.
- *user at domain*-style email addresses, commonly found in Mailman archives, are now handled correctly.
- Unicode characters transmitted over the XML-RPC API are now handled correctly under Python 3
- The *pwclient* tool will no longer attempt to re-encode unicode to ascii bytes, which was a frequent cause of `UnicodeEncodeError` exceptions. Instead, a warning is produced if your environement is not configured for unicode.

Other Notes

- reno is now used for release note management.
- Patch diffs now download with a `diff` extension.

CHAPTER 19

v1.1 Series (“Cashmere”)

1.1.3

This release fixes a number of issues with the 1.1.2 release.

Bug Fixes

- Some Python 3 issues are resolved in *pwclient*
- *pwclient* now functions as expected behind a proxy

1.1.2

This release fixed a number of issues with the 1.1.1 release.

Bug Fixes

- Headers containing invalid characters or codings are now parsed correctly
- Patches can no longer be delegated to any user

  This had significant performance impacts and has been reverted.

1.1.1

This release fixed a number of issues with the 1.1.0 release.

Bug Fixes

- Numerous issues in the *parsemail.py*, *parsearchive.py* and *parsemail.sh* scripts are resolved
- Permissions of database tables, as set by *grant-all* SQL scripts, are now set for tables added in Patchwork 1.1.0
- Some performance and usability regressions in the UI are resolved

1.1.0

This release focuses on usability and maintainability, and sets us up nicely for a v2.0.0 release in the near future. Feature highlights of v1.1.0 include:

- Automated delegation of patches, based on the files modified in said patches.
- Storing of test results, a.k.a. "checks", on a patch-by-patch basis.
- Delegation of patches to any registered Patchwork user (previously one had to be a registered maintainer).
- Overhaul of the web UI, which is now based on Bootstrap.
- Python 3 support.

New Features

- The web UI is updated to reflect modern web standards. Bootstrap 3.x is used.
- Python 3.4 is now supported
- Checks, which can be used to report the status of tests, have been added
- Automatic delegation of patches based on file path
- Automated documentation for the XML-RPC API. This can be found at the '/xmlrpc' in most Patchwork deployments
- Vagrant is now integrated for use during development

Upgrade Notes

- Patches can now be delegated to any Patchwork user.

CHAPTER 20

v1.0 Series ("Burlap")

1.0.0

This release changes a few admin-visible components of Patchwork, so upgrading involves a few steps.

New Features

- Patch tags are now supported

  Patch "tags", such as *Acked-by*, *Reviewed-by*, are typically included in patches and replies. They provide important information as to the activity and "mergability" of a patch. These tags are now extracted from patches and included in the patch list.

- Django 1.7 and Django 1.8 are now supported
- tox support is integrated for use by developers

Upgrade Notes

- Migrations are now executed using the Django migrations framework.

  Future database migrations will be implemented using Django Migrations, rather than raw SQL scripts. Before switching to Django migrations, first apply any unapplied migrations in the *lib/sql/migration* folder. For example, on postgres:

  ```
  $ psql -f lib/sql/migration/015-add-patch-tags.sql patchwork
  $ psql -f lib/sql/grant-all.postgres.sql patchwork
  ```

  Once applied, configure the required Django Migration tables using the *migrate* managment command:

  ```
  $ ./manage.py migrate --fake-initial
  ```

- Moved Patchwork source from the *apps* directory to the top level directory.

  Any scripts or tools that call Patchwork applications, such as *parsemail.sh*, must be updated to reference the new location of these scripts. To do this, simply remove *apps/* from the path, i.e. *apps/patchwork/* becomes *patchwork*.

- The *patchwork-cron.py* script has been replaced by the *cron* management command.

  Any references to the former should be updated to the latter. The *cron* management command can be called like so:

```
$ ./manage.py cron
```

- The *settings.py* file has been updated to reflect modern Django practices.

  You may need to manually migrate your existing configuration to the new settings file(s). By default, settings are read from *patchwork/settings/production.py*. To migrate, use the provided template:

```
$ cp patchwork/settings/production{.example,}.py
```

  Merge your previous settings, usually located in *apps/local_settings.py*, to this file.

  In addition, any scripts that set the *DJANGO_SETTINGS_MODULE* environment variable will need to be updated to reflect the new location, typically:

```
DJANGO_SETTINGS_MODULE=patchwork.settings.production
```

- Django *staticfiles* is now used to to gather static files for for serving via a web server

  Static content should now be located in the folder indicated by *STATIC_ROOT*. This should point to somewhere sensible, such as the absolute path of *htdocs/static* in the Patchwork tree. Configure the *STATIC_ROOT* setting in your settings file, then run the *collectstatic* management command:

```
$ ./manage.py collectstatic
```

  Finally, update your webserver configuration to serve the static content from this new location. Refer to the sample web configuration files provided in *lib* for more information.

- Django 1.5 is no longer supported
- Python 2.5 support was broken and is officially no longer supported

Deprecation Notes

- Django 1.6 support will be removed in a future release
- Raw SQL migration scripts, currently found at *lib/sql/migration*, will no longer be updated and will be removed in a future release. The Django Migration framework, found in Django 1.7 and above, should be used instead.

CHAPTER 21

v0.9 Series (“Alpaca”)

This represents the state of the Patchwork code before adopting semantic versioning, along with fabric-inspired release names. For information on the features available in this release, refer to the git logs.

Index

Symbols

I

M

P