DLRN Documentation

Release 1.0

RDO Community

May 24, 2023

Contents

Contents:

CHAPTER 1

Introduction

DLRN builds and maintains yum repositories following Openstack's upstream repositories.

CHAPTER 2

Installation

Installing prerequisites (CentOS 7):

```
$ sudo yum install git createrepo python-virtualenv mock gcc redhat-rpm-config␣
↪rpmdevtools httpd \
  libffi-devel openssl-devel
```

Installing prerequisites (CentOS 8 or Fedora):

```
$ sudo yum install git createrepo python3-virtualenv mock gcc redhat-rpm-config␣
↪rpmdevtools httpd \
  libffi-devel openssl-devel
```

Add the user you intend to run as to the mock group:

```
$ sudo usermod -a -G mock $USER
$ newgrp mock
$ newgrp $USER
```

If you want to serve the built packages and the status reports:

```
$ sudo systemctl start httpd
```

Install DLRN:

```
$ pip install dlrn
```

Or, if you have virtualenv installed:

```
$ virtualenv dlrn-venv
$ source dlrn-venv/bin/activate
$ pip install dlrn
```

The httpd module is not strictly required, DLRN does not use it. However, it will output it's results in a way that is suitable for a web-server to serve. This means you can easily set up a web-server to serve the finished `.rpm` and `.log` files.

2.1 Configuration

Configuration is done in an INI-file. An example file called `projects.ini` is included. The configuration file looks like this:

```
[DEFAULT]
datadir=./data
scriptsdir=./scripts
configdir=
baseurl=http://trunk.rdoproject.org/centos7/
distro=rpm-master
source=master
target=centos
project_name=RDO
smtpserver=
reponame=delorean
templatedir=./dlrn/templates
maxretries=3
pkginfo_driver=dlrn.drivers.rdoinfo.RdoInfoDriver
build_driver=dlrn.drivers.mockdriver.MockBuildDriver
tags=
rsyncdest=
rsyncport=22
workers=1
gerrit_topic=rdo-FTBFS
database_connection=sqlite:///commits.sqlite
fallback_to_master=1
nonfallback_branches=^master$,^rpm-master$,^rhos-
coprid=account/repo
release_numbering=0.date.hash
release_minor=0
custom_preprocess=
include_srpm_in_repo=true
keep_changelog=false
use_components=false
deps_url=
```

- `datadir` is the directory where the packages and repositories will be created. If not set, it will default to `./data` on the parent directory of where DLRN is installed.
- `scriptsdir` is the directory where scripts utilized during the build and test process are located. If not set, it will default to `./scripts` on the parent directory of where DLRN is installed.
- `configdir` is the directory where additional configuration files used by the build process are located, such as base mock configurations. If not set, it defaults to the value of `scriptsdir`.
- `baseurl` is the URL to the data-directory, as hosted by your web-server. Unless you are installing DLRN for local use only, this must be a publicly accessible URL.
- `distro` is the branch to use for building the packages.
- `source` is the branch to use from the upstream repository.
- `target` is the distribution to use for building the packages (`centos`, `fedora` or `redhat`, provided that you have the right content).
- `project_name` name of the project for which DLRN is building RPMs. This name is used to render various templates (emails, web pages).

- `smtpserver` is the address of the mail server for sending out notification emails. If this is empty no emails will be sent out. If you are running DLRN locally, then do not set an smtpserver.
- `reponame` name of the directory that contains the generated repository.
- `templatedir` path to the directory that contains the report templates and stylesheets. If not set, it will default to `./templates` under the directory where DLRN is installed.
- `maxretries` is the maximum number of retries on known errors before marking the build as failed. If a build fails, DLRN will check the log files for known, transient errors such as network issues. If the build fails for that reason more than maxretries times, it will be marked as failed.
- `gerrit` if set to anything, instructs dlrn to create a gerrit review when a build fails. See next section for details on how to configure gerrit to work.
- If `gerrit` is set, then `gerrit_topic` will define the Gerrit topic to use when a review is opened.
- `tags` is used to filter information received to decide what packages are built. Should be set to a release name (e.g. mitaka) to instruct the builder to only show packages with that release tag.
- `rsyncdest` if set, specifies a destination path where the hashed repository directories created by DLRN will be synchronized using `rsync`, after each commit build. An example would be `root@backupserver.example.com:/backupdir`. Make sure the user running DLRN has access to the destination server using passswordless SSH.
- `rsyncport` is the SSH port to be used when synchronizing the hashed repository. If `rsyncdest` is not defined, this option will be ignored.
- `workers` is the number of parallel build processes to launch. When using multiple workers, the mock build part will be handled by a pool of processes, while the repo creation and synchronization will still be sequential.
- The `database_connection` string defines a database connection string. By default, a local SQLite3 database is used, but it is also possible to set up an external database.
- `fallback_to_master` defines the fallback behavior when cloning Git repositories.
  - if dlrn fails to clone the branch defined in source dlrn parameter, it tries to clone tag <release>-eol if it exists. If it does not, it checks the fallback_to_master parameter. With the default value of 1, DLRN will fall back to the `master` or `main` branch.
  - if dlrn fails to clone a branch named `<name>-rdo`, it assumes it is a distgit and will try to fallback to `rpm-master` if parameter fallback_to_master is set to 1.
  - If dlrn fails to checkout a branch different to the one defined in source parameter and the name does not ends with `-rdo`, it will fail to build.
- `nonfallback_branches` defines a list of regular expressions of branches for source and distgit repositories that should never fall back to other branches, even if not present in the repository. This is used when we want to avoid certain type of fallback that could cause issues in our environment.

  The default value is `^master$,^rpm-master$`, which means that branches named `master` or `rpm-master` will never try to fall back.
- `pkginfo_driver` defines the driver to be used to manage the distgit repositories. Following drivers are available:
  - `dlrn.drivers.rdoinfo.RdoInfoDriver`, which uses information provided by rdoinfo to determine the distgit repo location and information.
  - `dlrn.drivers.downstream.DownstreamInfoDriver`, which uses information provided by a `distroinfo` repo such as rdoinfo while reusing `distro_hash` and `commit_hash` from a remote `versions.csv` file specified by `versions_url` config option in the `[downstream_driver]` section. It will also use a separate distgit to build the driver, as well as a downstream source git. The distgit

  URL will be defined by the `downstream_distgit_base` URL + the package name, and the distgit branch to use will be defined by the `downstream_distro_branch` variable.

  - `dlrn.drivers.gitrepo.GitRepoDriver`, which uses a single Git repository with per-distgit directories, following the same schema used by the RPM Packaging for OpenStack project. This driver requires setting some optional configuration options in the `[gitrepo_driver]` section.
  - `dlrn.drivers.local.LocalDriver`, which uses a current directory to discover a specfile. The current directory must be a git repository. The specfile is used as it is to build the rpm(s). This driver does not require specific configuration options.

- `build_driver` defines the driver used to build the packages. Source RPMs are always created using Mock, but the actual RPM build process can use the following drivers:
  - `dlrn.drivers.mockdriver.MockBuildDriver`, which uses Mock to build the package. There are some optional configuration options in the `[mockbuild_driver]` section.
  - `dlrn.drivers.kojidriver.KojiBuildDriver`, which uses koji to build the package. There are some mandatory configuration options in the `[kojibuild_driver]` section. To use this driver, you need to make sure the `koji` command (or any alternative if you use a different binary) is installed on the system.
  - `dlrn.drivers.coprdriver.CoprBuildDriver`, which uses copr to build the package. The mandatory configuration `coprid` option in the `[coprbuild_driver]` section must be set to use this driver. You need to make sure the `copr-cli` command is installed on the system. Configure only one target architecture per COPR builder else it would confuse DLRN.
- `release_numbering` defines the algorithm used by DLRN to assign release numbers to packages. The release number is created from the current date and the source repository git hash, and can use two algorithms:
  - `0.date.hash` if the old method is used: 0.<date>.<hash>
  - `0.1.date.hash` if the new method is used: 0.1.<date>.<hash>. This new method provides better compatibility with the Fedora packaging guidelines.
  - `minor.date.hash` allows you to specify the minor version to be used, which can be different from 0. If this release numbering schema is used, the value of `minor` will be determined by `release_minor`.
- `release_minor` only takes place when `release_numbering` is set to `minor.date.hash`. For example, if this value is set to `3`, the release number for all packages will be computed as 3.date.hash.
- `custom_preprocess`, if set, defines a comma-separated list of custom programs or scripts to be called as part of the pre-process step. The custom programs will be executed sequentially.

  After the distgit is cloned, and before the source RPM is built, the `pkginfo` drivers run a pre-process step where some actions are taken on the repository, such as Jinja2 template processing. In addition to this per-driver step, a custom pre-process step can be specified. The external program(s) will be executed with certain environment variables set:
  - `DLRN_PACKAGE_NAME`: name of the package being built.
  - `DLRN_DISTGIT`: path to the distgit in the local file system.
  - `DLRN_SOURCEDIR`: path to the source git in the local file system.
  - `DLRN_SOURCE_COMMIT`: commit hash of the source repository being built.
  - `DLRN_USER`: name of the user running DLRN.
  - `DLRN_UPSTREAM_DISTGIT`: for the `downstream` driver, path to the upstream distgit in the local file system.
  - `DLRN_DISTROINFO_REPO`: for the `rdoinfo` and `downstream` drivers, path to the local or remote distroinfo repository used by the instance.

Do not assume any other environment variable (such as PATH), since it may not be defined.

- `include_srpm_in_repo`, if set to true (default), includes source RPMs in the repositories generated by DLRN. If set to false, DLRN will exclude source RPMs from the repositories.
- `keep_changelog`, if set to true, will not clean the %changelog section from spec files when building the source RPM. When set to the default value of `false`, DLRN will remove all changelog content from specs.
- `use_components`, if set to true, will enable component support for DLRN. This is currently provided by the `dlrn.drivers.rdoinfo.RdoInfoDriver` driver only. Please refer to the internals page for details on component support.
- `deps_url` allows the user to specify a custom URL for the dependency repositories file. By default, if not set, DLRN will fetch a file from the URL formed by `baseurl` + `delorean-deps.repo`. Note it is possible to specify a URL in the traditional `http://example.com/path/to/file.repo` as well as a local file using `file:///path/to/file.repo`.

The optional `[gitrepo_driver]` section has the following configuration options:

```
[gitrepo_driver]
repo=http://github.com/openstack/rpm-packaging
directory=/openstack
skip=openstack-macros,keystoneauth1
use_version_from_spec=0
keep_tarball=0
```

- `repo` is the single Git repository where all distgits are located.
- `directory` is a directory or comma-separated list of the directories inside the repo. DLRN will expect each directory inside it to include the spec file for a single project, using a Jinja2 template like in the RPM Packaging for OpenStack project.
- `skip` is a comma-separated list of directories to skip from `directory` when creating the list of packages to build. This can be of use when the Git repo contains one or more directories without a spec file in it, or the package should not be built for any other reason.
- `use_version_from_spec` If set to 1 (or true), the driver will parse the template spec file and set the source branch to the Version: tag in the spec.
- `keep_tarball` If set to 1 (or true), and the spec template detects the package version automatically using a tarball (see[1]), DLRN will not replace the Source0 file with a tarball generated from the Git repo, but it will use the same tarball used to detect the package version. This defeats the purpose of following the commits from Git, but it is useful in certain scenarios, such as CI testing, when the tarball or its tags may not be in sync with the Git contents.

The optional `[rdoinfo_driver]` section has the following configuration options:

```
[rdoinfo_driver]
repo=http://github.com/org/rdoinfo-fork
info_files=file.yml
cache_dir=~/.distroinfo/cache
```

- `repo` defines the rdoinfo repository to use. This setting must be set if a fork of the rdoinfo repository must be used.
- `info_files` selects an info file (or a list of info files) to get package information from (within the distroinfo repo selected with `repo`). It defaults to `rdo.yml`.

[1] Version handling using renderspec templates https://github.com/openstack/renderspec/blob/master/doc/source/usage.rst#handling-the-package-version

- `cache_dir` defines the directory uses for caching to avoid downloading the same repo multiple times. By default, it uses None. A different base directory for the cache can be set for both `[rdoinfo_driver]` and `[downstream_driver]`

The optional `[downstream_driver]` section has the following configuration options:

```
[downstream_driver]
repo=http://github.com/org/fooinfo
info_files=foo.yml
versions_url=https://trunk.rdoproject.org/centos7-master/current/versions.csv
downstream_distro_branch=foo-rocky
downstream_tag=foo-
downstream_distgit_tag=foo-distgit
use_upstream_spec=False
downstream_spec_replace_list=^foo/bar,string1/string2
cache_dir=~/.distroinfo/cache
downstream_source_git_key=bar-distgit
downstream_source_git_branch=ds-master
```

- `repo` selects a distroinfo repository to get package information from.
- `info_files` selects an info file (or a list of info files) to get package information from (within the distroinfo repo selected with `repo`)
- `versions_url` must point to a `versions.csv` file generated by DLRN instance. Parameter `versions_url` can be a comma separated list of `versions.csv` URLs. In this case, the content of latest csv files overrides previous ones (last wins). This allows to override versions for packages in specific component by using component-specific versions.csv files provided by a different DLRN instance. `distro_hash` and `commit_hash` will be reused from supplied `versions.csv` URL(s) and only packages present in the file(s) are processed.
- `downstream_distro_branch` defines which branch to use when cloning the downstream distgit, since it may be different from the upstream distgit branch.
- `downstream_tag` if set, it will filter the `packages` section of packaging metadata (from `repo/info_files`) to only contain packages with the `downstream_tag` tag. This tag will be filtered in addition to the one set in the `DEFAULT/tags` section.
- `downstream_distgit_key` is the key used to find the downstream distgit in the `packages` section of packaging metadata (from `repo/info_files`).
- `use_upstream_spec` defines if the upstream distgit contents (spec file and additional files) should be copied over the downstream distgit after cloning.
- `downstream_spec_replace_list`, when `use_upstream_spec` is set to True, will perform some sed-like edits in the spec file after copying it from the upstream to the downstream distgit. This is specially useful when the downstream DLRN instance has special requirements, such as building without documentation. in that case, a regular expresion like the following could be used:
- `downstream_source_git_key` is the key used to find the downstream source git in the `packages` section of the packaging metadata (from `repo/info_files`).
- `downstream_source_git_branch` defines which branch to use when cloning the downstream source git.

```
 downstream_spec_replace_list=^%global with_doc.+/%global with_doc 0

Multiple regular expressions can be used, separated by commas.
```

- `cache_dir` defines the directory uses for caching to avoid downloading the same repo multiple times. By default, it uses None. A different base directory for the cache can be set for both `[rdoinfo_driver]` and `[downstream_driver]`

The optional `[mockbuild_driver]` section has the following configuration options:

```
[mockbuild_driver]
install_after_build=1
```

- The `install_after_build` boolean option defines whether mock should try to install the newly created package in the same buildroot or not. If not specified, the default is `True`.

The optional `[kojibuild_driver]` section is only taken into account if the build_driver option is set to `dlrn.drivers.kojidriver.KojiBuildDriver`. The following configuration options are included:

```
[kojibuild_driver]
koji_exe=koji
krb_principal=user@EXAMPLE.COM
krb_keytab=/home/user/user.keytab
scratch_build=True
build_target=koji-target-build
arch=aarch64
use_rhpkg=False
fetch_mock_config=False
mock_base_packages=basesystem rpm-build
mock_package_manager=yum
additional_koji_tags=tag1,tag2
```

- `koji_exe` defines the executable to use. Some Koji instances create their own client packages to add their default configuration, such as CBS or Brew. If not specified, it will default to `koji`.
- `krb_principal` defines the Kerberos principal to use for the Koji builds. If not specified, DLRN will assume that authentication is performed using SSL certificates.
- `krb_keytab` is the full path to a Kerberos keytab file, which contains the Kerberos credentials for the principal defined in the `krb_principal` option.
- `scratch_build` defines if a scratch build should be used. By default, it is set to `True`.
- `build_target` defines the build target to use. This defines the buildroot and base repositories to be used for the build.
- `arch` allows to override default architecture (x86_64) in some cases (e.g retrieving mock configuration from Koji instance).
- `use_rhpkg` allows us to use `rhpkg` as the build tool in combination with `koji_exe`. That involves some changes in the workflow:
  - Instead of using `koji_exe` to trigger the build, DLRN will generate the source RPM, and upload it to the distgit path using `rhpkg import`.
  - DLRN will run `rhpkg build` to actually trigger the build.

  Note that `rhpkg` requires a valid Kerberos ticket, so the `krb_principal` and `krb_keytab` options must be set.

  Also note that setting `rhpkg` only makes sense when using `dlrn.drivers.downstream.DownstreamInfoDriver` as the pkginfo driver.
- `koji_rhpkg_timeout`, indicates the timeout for rhpkg commands. Default 3600.
- `fetch_mock_config`, if set to `true`, will instruct DLRN to download the mock configuration for the build target from Koji, and use it when building the source RPM. If set to `false`, DLRN will use its internally defined mock configuration, based on the `DEFAULT/target` configuration option.

- `mock_base_packages`, if `fetch_mock_config` is set to `true`, will define the set of base packages that will be installed in the mock configuration when creating the source RPM. This list of packages will override the one fetched in the mock configuration, if set. If not set, no overriding will be done.
- `mock_package_manager`, if `fetch_mock_config` is set to `true`, will override the `config_ops['package_manager']` option from the fetched mock configuration. This allows us to have different package managers if we are building for different operating system releases, such as CentOS 7 (yum) and CentOS 8 (dnf).
- `additional_koji_tags`, if set, will assign the build the additional tags defined in the list.

The optional `[coprbuild_driver]` section has the following configuration options:

```
[coprbuild_driver]
coprid=account/repo
```

- The `coprid` option defines Copr id to use to compile the packages.

2.1.1 Configuring for gerrit

You first need `git-review` installed. You can use a package or install it using pip.

Then the username for the user creating the gerrit reviews when a build will fail needs to be configured like this:

> $ git config –global gitreview.username dlrnbot $ git config –global user.email dlrn@dlrn.domain

and authorized to connect to Gerrit without password. Make sure the public SSH key of the user that run DLRN is defined in the Gerrit account linked to the DLRN user email.

2.2 Configuring your httpd

The output generated by DLRN is a file structure suitable for serving with a web-server. You can either add a section in the server configuration where you map a URL to the data directories, or just make a symbolic link:

```
$ cd /var/www/html
$ sudo ln -s <datadir>/repos .
```

2.3 Database support

DLRN supports different database engines through SQLAlchemy. SQLite3 and MariaDB have been tested so far. You can set the `database_connection` parameter in projects.ini with the required string, using the SQLAlchemy syntax.

For MariaDB, use a mysql+pymysql driver, with the following string:

```
database_connection=mysql+pymysql://user:password@serverIP/dlrn
```

That requires you to pre-create the `dlrn` database.

If your MariaDB database is placed on a publicly accessible server, you will want to secure it as a first step:

```
$ sudo mysql_secure_installation
```

You can use the following commands to create the database and grant the required permissions:

```
use mysql
create database dlrn;
grant all on dlrn.* to 'user'@'%' identified by 'password';
flush privileges;
```

You may also want to enable TLS support in your connections. In this case, follow the steps detailed in the MariaDB documentation to enable TLS support on your server. Generate the client key and certificates, and then set up your database connection string as follows:

```
database_connection=mysql+pymysql://user:password@serverIP/dlrn?ssl_cert=/dir/client-
↪cert.pem&ssl_key=/dir/client-key.pem
```

You can also force the MySQL user to connect using TLS if you create it as follows:

```
use mysql
create database dlrn;
grant all on dlrn.* to 'user'@'%' identified by 'password' REQUIRE SSL;
flush privileges;
```

2.3.1 Database migration

During DLRN upgrades, you may need to upgrade the database schemas, in order to keep your old history. To migrate database to the latest revision, you need the alembic command-line and to run the `alembic upgrade head` command.

```
$ sudo yum install -y python-alembic
$ alembic upgrade head
```

If the database doesn't exist, `alembic upgrade head` will create it from scratch.

If you are using a MariaDB database, the initial schema will not be valid. You should start by running DLRN a first time, so it creates the basic schema, then run the following command to stamp the database to the first version of the schema that supported MariaDB:

```
$ alembic stamp head
```

After that initial command, you will be able to run future migrations.

2.3.2 Adding a custom mock base configuration

The source RPM build operations, and the binary RPM build by default, are performed using `mock`. Mock uses a configuration file, and DLRN provides sample files for CentOS and Fedora in the `scripts/` directory.

You may want to use a different base mock configuration, if you need to specify a different base package set or an alternative yum repository. The procedure to do so is the following:

- Edit the `configdir` variable in your projects.ini file, and make it point to a configuration directory.
- In that new directory, create the configuration file. It should be named `<target>.cfg`, where `<target>` is the value of the target option in projects.ini.
- For the mock configuration file syntax, refer to the mock documentation.

References

CHAPTER 3

Repositories

DLRN doesn't stop at building packages, it also generates yum repositories you can install the packages from.

DLRN repositories are all hosted on http://trunk.rdoproject.org.

This documentation goes through the various repositories and what they are used for.

3.1 Building new packages and repositories

DLRN watches upstream git repositories for new commits. When there is one, DLRN builds a new version of the project's package with the new commit.

On a successful build, DLRN will generate a new repository with the latest version of every package that successfully built.

A package build can fail due to different reasons, for example when a new dependency was introduced that needs to be added to the RPM spec file. If there is a build failure, no repository is generated and the project's package is not updated.

The package will not be updated for as long as it fails to build. This means that newer repositories generated from other projects' commits would not contain all the latest commits of the project that failed to build.

DLRN does not delete any generated repositories. This means we can use any previously built repositories if necessary.

Generated repositories are unique and each have their own hash. For example, you might be using the DLRN `/centos7/current/delorean.repo` repository but in fact this corresponds to `/centos7/42/0c/420c638d6325d1ccf50eb5fe430c5d255dcbfb94_52cbbfe7`.

DLRN manages these references as simple symbolic links for the `current` and `consistent` repositories. The `current-passed-ci` repository is a symbolic link managed automatically by RDO's continuous integration pipeline and is not managed or known by DLRN itself.

3.2 DLRN repository: delorean-deps

OpenStack projects are typically built into the DLRN repositories. These projects require dependencies that DLRN does not build, for example python-requests, python-prettytable and so on.

The RDO project provides a mirror which contains all of these dependencies and the repository configuration is available at `/delorean-deps.repo` for each release.

For example:

- Trunk: http://trunk.rdoproject.org/centos7/delorean-deps.repo
- Liberty: http://trunk.rdoproject.org/centos7-liberty/delorean-deps.repo

3.3 DLRN repository: current

On a successful build, DLRN will generate a new repository with the latest version of every package that successfully built.

This new repository will be tagged as `current`. A `current` repo contains the last successfully built package from every project.

A DLRN current repository might not contain all the latest upstream commits, if any of them failed to build the package. For example, if we had 100 packages, 99 of them have been successfully built but `openstack-nova` failed, the `current` repository would contain the latest commits from 99 projects, and the last commit that could be built for openstack-nova, which is at least 1 commit behind the current master.

if there are any ongoing build failures that are unresolved.

This repository is available at `/current/delorean.repo` for each release.

For example:

- Trunk: http://trunk.rdoproject.org/centos7/current/delorean.repo
- Liberty: http://trunk.rdoproject.org/centos7-liberty/current/delorean.repo

3.4 DLRN repository: consistent

DLRN `consistent` repositories are generated for any given set of packages that have no current build failures.

These repositories have the latest and greatest of every package and all upstream commits have been successfully built up until that point. In the above example, if 99 packages are successfully built but `openstack-nova` fails to build, the `consistent` repository will not be updated until it is fixed.

The continuous integration done to test RDO packages target the DLRN consistent repositories.

This repository is available at `/consistent/delorean.repo` for each release.

For example:

- Trunk: http://trunk.rdoproject.org/centos7/consistent/delorean.repo
- Liberty: http://trunk.rdoproject.org/centos7-liberty/consistent/delorean.repo

3.5 DLRN repository: current-passed-ci

The RDO project has a continuous integration pipeline that consists of multiple jobs that deploy and test OpenStack as accomplished by different installers.

This vast test coverage attempts to ensure that there are no known issues either in packaging, in code or in the installers themselves.

Once a DLRN consistent repository has undergone these tests successfully, it will be promoted to `current-passed-ci`.

current-passed-ci represents the latest and greatest version of RDO trunk packages that were tested together successfully.

We encourage installer projects and users of RDO to use this repository to keep up with trunk while maintaining a certain level of stability provided by RDO's CI.

This repository is available at `/current-passed-ci/delorean.repo` for each release.

For example:

- Trunk: http://trunk.rdoproject.org/centos7/current-passed-ci/delorean.repo
- Liberty: http://trunk.rdoproject.org/centos7-liberty/current-passed-ci/delorean.repo

CHAPTER 4

Usage

4.1 Parameters

```
usage: dlrn [-h] [--config-file CONFIG_FILE]
            [--config-override CONFIG_OVERRIDE] [--info-repo INFO_REPO]
            [--build-env BUILD_ENV] [--local] [--head-only]
            [--project-name PROJECT_NAME | --package-name PACKAGE_NAME]
            [--dev] [--log-commands] [--use-public] [--order] [--sequential]
            [--status] [--recheck] [--force-recheck] [--version] [--run RUN]
            [--stop] [--verbose-build] [--no-repo] [--debug]

optional arguments:
  -h, --help            show this help message and exit
  --config-file CONFIG_FILE
                        Config file. Default: projects.ini
  --config-override CONFIG_OVERRIDE
                        Override a configuration option from the config file.
                        Specify it as: section.option=value. Can be used
                        multiple times if more than one override is needed.
  --info-repo INFO_REPO
                        use a local rdoinfo repo instead of fetching the
                        default one using rdopkg. Only applies when
                        pkginfo_driver is rdoinfo in projects.ini
  --build-env BUILD_ENV
                        Variables for the build environment.
  --local               Use local git repos if possible. Only commited changes
                        in the local repo will be used in the build.
  --head-only           Build from the most recent Git commit only.
  --project-name PROJECT_NAME
                        Build a specific project name only. Use multiple times
                        to build more than one project in a run.
  --package-name PACKAGE_NAME
                        Build a specific package name only. Use multiple times
                        to build more than one package in a run.
```

(continues on next page)

(continued from previous page)

```
  --dev                 Don't reset packaging git repo, force build and add
                        public master repo for dependencies (dev mode).
  --log-commands        Log the commands run by dlrn.
  --use-public          Use the public master repo for dependencies when doing
                        install verification.
  --order               Compute the build order according to the spec files
                        instead of the dates of the commits. Implies
                        --sequential.
  --sequential          Run all actions sequentially, regardless of the number
                        of workers specified in projects.ini.
  --status              Get the status of packages.
  --recheck             Force a rebuild for a particular package. Implies
                        --package-name
  --force-recheck       Force a rebuild for a particular package, even if its
                        last build was successful. Requires setting
                        allow_force_rechecks=True in projects.ini. Implies
                        --package-name and --recheck
  --version             show program's version number and exit
  --run RUN             Run a program instead of trying to build. Implies
                        --head-only
  --stop                Stop on error.
  --verbose-build       Show verbose output during the package build.
  --no-repo             Do not generate a repo with all the built packages.
  --debug               Print debug logs
```

4.2 Quickstart single package build

Run DLRN for the package you are trying to build.

```
$ dlrn --use-public --package-name openstack-cinder
```

By using the parameter `--use-public` DLRN will configure the build environment to use the public master repository.

In case of failure you might need to re-run a build by discarding the DLRN database content. To do so you need to run:

```
$ dlrn --recheck --package-name openstack-cinder
$ dlrn --use-public --package-name openstack-cinder
```

It is also possible to force the recheck of a successfully built commit. Please note that this is not advisable if you rely on the DLRN-generated repositories, since it will remove packages that other hashed repositories may have symlinked.

If you are sure you need it, set `allow_force_rechecks=true` in your projects.ini file, then run:

```
$ dlrn --recheck --force-recheck --package-name openstack-cinder
$ dlrn --use-public --package-name openstack-cinder
```

4.3 Full build

Some of the projects require others to build. As a result, use the special option `--order` to build in the order computed from the BuildRequires and Requires fields of the spec files. If this option is not specified, DLRN builds the packages in the order of the timestamps of the commits.

```
$ dlrn --order
```

4.4 Advanced single package build

Run DLRN for the package you are trying to build.

```
$ dlrn --local --package-name openstack-cinder
```

This will clone the packaging for the project you're interested in into `data/openstack-cinder_repo`, you can now change this packaging and rerun the DLRN command in test your changes.

This command expects build and runtime dependencies to be found in previously built repositories (during the initial full build).

If you have locally changed the packaging make sure to include `--dev` in the command line. This switches DLRN into **dev mode** which causes it to preserve local changes to your packaging between runs so you can iterate on spec changes. It will also cause the most current public master repository to be installed in your build image(as some of its contents will be needed for dependencies) so that the packager doesn't have to build the entire set of packages.

4.5 Output and log files

The output of DLRN is generated in the `<datadir>/repos` directory. It consists of the finished `.rpm` files for download, located in `/repos/current`, and reports of the failures in `/repos/status_report.html`, and a report of all builds in `/repos/report.html`.

4.6 Importing commits built by another DLRN instance

DLRN has the ability to import a commit built by another instance. This allows a master-worker architecture, where a central instance aggregates builds made by multiple, possibly short-lived instances.

The builder instance will be invoked as usual, and it will output a `commit.yaml` file in the generated repo. In general, we will want to use the `--use-public` command-line option to make sure all repos are available. Note it is very important to **not use** the `--dev` command-line option, as some of the commit metadata will be lost, specifically all data related to the distgit repository.

On the central instance side, the `dlrn-remote` has the following syntax:

```
usage: dlrn-remote [-h] [--config-file CONFIG_FILE] --repo-url REPO_URL [--info-repo␣
↪INFO_REPO]

arguments:
  -h, --help            show this help message and exit
  --config-file CONFIG_FILE
                        Config file. Default: projects.ini
  --repo-url REPO_URL   Base repository URL for remotely generated repo
                        (required)
  --info-repo INFO_REPO
                        use a local rdoinfo repo instead of fetching the
                        default one using rdopkg. Only applies when
                        pkginfo_driver is rdoinfo in projects.ini
```

An example command-line would be:

```
$ dlrn-remote --config-file projects.ini \
  --repo-url http://<builder IP>/repos/<hash>/
```

Where `http://192.168.122.164/repos/<hash>` is the URL where the builder instance exports its built repo. The `commit.yaml` file must be on the same hashed repo, as created by DLRN.

4.7 Purging old commits

Over time, the disk space consumed by DLRN will grow, as older commits and their repositories are never removed. It is possible to use the `dlrn-purge` command to purge commits built before a certain date.

```
usage: dlrn-purge [-h] --config-file CONFIG_FILE --older-than OLDER_THAN [-y] [--dry-
↪run]
arguments:
  -h, --help            show this help message and exit
  --config-file CONFIG_FILE
                        Config file (required)
  --older-than  OLDER_THAN
                        how old a build needs to be, in order to be considered
                        for removal (required). It is measured in days.
  -y                    Assume yes for all questions.
  --dry-run             If specified, do not apply any changes. Instead, show what␣
↪would
                        be removed from the filesystem.
```

Old commits will remain in the database, although their flag will be set to purged, and their associated repo directory will be removed. There is one exception to this rule, when an old commit is the newest one that was successfully built. In that case, it will be preserved.

4.8 Building only the last commit

You can use the `--head-only` option to build only the last commit of the branch for all the projects or a particular project using `--project-name` or `--package-name`.

Doing so you skip commits and if you find a problem in the last commit, you can use the `./scripts/bisect.sh` helper to drive a `git bisect` session to find which commit has caused the problem:

```
Usage: ./scripts/bisect.sh <dlrn config file> <project name> <good sha1> <bad sha1> [
↪<dlrn extra args>]
```

CHAPTER 5

Troubleshooting

If you interrupt dlrn during mock build you might get an error

```
OSError: [Errno 16] Device or resource busy: '/var/lib/mock/dlrn-centos-x86_64/root/
↪var/cache/yum'
```

Solution is to clear left-over bind mount as root:

```
# umount /var/lib/mock/dlrn-centos-x86_64/root/var/cache/yum
```

5.1 Other requirements

If the git clone operation fails for a package, DLRN will try to remove the source directory using sudo. Please make sure the user running DLRN can run `rm -rf /path/to/dlrn/data/*` without being asked for a password, otherwise DLRN will fail to process new commits.

5.2 API issues

If you want to quickly check the API status, you can use the /api/health endpoint. It will allow you to test API connectivity, database access and authentication:

```
# curl http://localhost:5000/api/health
# curl -d test=test --user user:password http://localhost:5000/api/health
```

CHAPTER 6

API definition

6.1 General information

`GET` operations will be non-authenticated. `POST` operations will require authentication using username+password.

Password information is stored in the database using the SHA512 hash.

For POST operations, all data will be sent/received using JSON objects, unless stated otherwise. For GET operations, the recommended method is to send data using in-query parameters. JSON in-body objects still work, but are deprecated and expected to be removed in a future version.

6.2 API calls

6.2.1 GET /api/health

Check the API server health. This will trigger a database connection to ensure all components are in working condition.

Normal response codes: 200

Error response codes: 401

Response:

Parameter	Type	Description
result	string	A simple success string

6.2.2 POST /api/health

Check the API server health. This will trigger a database connection to ensure all components are in working condition. In addition to this, the POST call will check authentication.

Normal response codes: 200

Error response codes: 401

Response:

Parameter	Type	Description
result	string	A simple success string

6.2.3 GET /api/last_tested_repo

Get the last tested repo since a specific time.

If a `job_id` is specified, the order of precedence for the repo returned is:

- The last tested repo within that timeframe for that CI job.
- The last tested repo within that timeframe for any CI job, so we can have several CIs converge on a single repo.
- The last "consistent" repo, if no repo has been tested in the timeframe.

If `sequential_mode` is set to true, a different algorithm is used. Another parameter `previous_job_id` needs to be specified, and the order of precedence for the repo returned is:

- The last tested repo within that timeframe for the CI job described by `previous_job_id`.
- If no repo for `previous_job_id` is found, an error will be returned

The sequential mode is meant to be used by CI pipelines, where a CI (n) job needs to use the same repo tested by CI (n-1).

Normal response codes: 200

Error response codes: 400

Request:

Parameter	Type	Description
max_age	integer	Maximum age (in hours) for the repo to be considered. Any repo tested or being tested after "now - max_age" will be taken into account. If set to 0, all repos will be considered.
success	boolean (optional)	If set to a value, find repos with a successful/unsuccessful vote (as specified). If not set, any tested repo will be considered.
job_id	string (optional)	Name of the CI that sent the vote. If not set, no filter will be set on CI.
sequential_mode	boolean (optional)	Use the sequential mode algorithm. In this case, return the last tested repo within that timeframe for the CI job described by previous_job_id. Defaults to false.
previous_job_id	string (optional)	If sequential_mode is set to true, look for jobs tested by the CI identified by previous_job_id.
component	string (optional)	Only report votes associated to this component

Response:

Parameter	Type	Description
commit_hash	string	commit_hash of tested repo
distro_hash	string	distro_hash of tested repo
extended_hash	string	extended_hash of tested repo
success	boolean	whether the test was successful or not
job_id	string	name of the CI sending the vote
in_progress	boolean	is this CI job still in-progress?
timestamp	integer	timestamp for the repo
user	string	user who created the CI vote
component	string	Component associated to the commit/distro hash

6.2.4 GET /api/repo_status

Get all the CI reports for a specific repository.

Normal response codes: 200

Error response codes: 400

Request:

Parameter	Type	Description
commit_hash	string	commit_hash of the repo to fetch information for
distro_hash	string	distro_hash of the repo to fetch information for
extended_hash	string (optional)	If set, extended_hash of the repo to fetch information for. If not set, the latest commit with the commit/distro hash combination will be reported.
success	boolean (optional)	If set to a value, only return the CI reports with the specified vote. If not set, return all CI reports.

Response:

The JSON output will contain an array where each item contains:

Parameter	Type	Description
job_id	string	name of the CI sending the vote
commit_hash	string	commit_hash of tested repo
distro_hash	string	distro_hash of tested repo
extended_hash	string	extended_hash of tested repo
url	string	URL where to find additional information from the CI execution
timestamp	integer	Timestamp (in seconds since the epoch)
in_progress	boolean	False -> is this CI job still in-progress?
success	boolean	Was the CI execution successful?
notes	Text	Additional notes
user	string	user who created the CI vote
component	string	Component associated to the commit/distro hash

6.2.5 GET /api/agg_status

Get all the CI reports for a specific aggregated repository.

Normal response codes: 200

Error response codes: 400

Request:

Parameter	Type	Description
aggregate_hash	string	hash of the aggregated repo to fetch information for
success	boolean (optional)	If set to a value, only return the CI reports with the specified vote. If not set, return all CI reports.

Response:

The JSON output will contain an array where each item contains:

Parameter	Type	Description
job_id	string	name of the CI sending the vote
aggregate_hash	string	hash of tested aggregated repo
url	string	URL where to find additional information from the CI execution
timestamp	integer	Timestamp (in seconds since the epoch)
in_progress	boolean	False -> is this CI job still in-progress?
success	boolean	Was the CI execution successful?
notes	Text	Additional notes
user	string	user who created the CI vote

6.2.6 GET /api/promotions

Get all the promotions, optionally for a specific repository or promotion name. The output will be sorted by the promotion timestamp, with the newest first, and limited to 100 results per query.

Normal response codes: 200

Error response codes: 400

Request:

Parameter	Type	Description
commit_hash	string (optional)	If set, commit_hash of the repo to use as filter key. Requires distro_hash.
distro_hash	string (optional)	If set, commit_hash of the repo to use as filter key. Requires commit_hash.
extended_hash	string (optional)	If set, extended_hash of the repo to use as filter key. Requires commit_hash and distro_hash.
aggregate_hash	string (optional)	If set, use the generated aggregate_hash as filter key. Only makes sense when components are enabled.
promote_name	string (optional)	If set to a value, filter results by the specified promotion name.
offset	integer (optional)	If set to a value, skip the initial <offset> promotions.
limit	integer (optional)	If set to a value, limit the returned promotions amount to <limit>.
component	string (optional)	If set to a value, only report promotions for this component.

The JSON output will contain an array where each item contains:

Parameter	Type	Description
commit_hash	string	commit_hash of the promoted repo
distro_hash	string	distro_hash of the promoted repo
extended_hash	string	extended_hash of the promoted repo
agggregate_hash	string	Hash of the aggregated repo file, when using components
repo_hash	string	Repository hash, composed of the commit_hash and short distro_hash
repo_url	string	Full URL of the promoted repository
promote_name	string	name used for the promotion
component	string	Component associated to the commit/distro hash
timestamp	integer	Timestamp (in seconds since the epoch)
user	string	user who created the promotion

The array will be sorted by the promotion timestamp, with the newest first.

6.2.7 GET /api/metrics/builds

Retrieve statistics on the number of builds during a certain period, optionally filtered by package name.

Normal response codes: 200

Error response codes: 400

Parameter	Type	Description
start_date	string	Start date for the period, in YYYY-mm-dd format. The start date is included in the reference period.
end_date	string	End date for the period, in YYYY-mm-dd format. The end date is not included in the period, so it is start_date <= date < end_date.
package_name	string (optional)	If set to a value, report metrics only for the specified package name.

Response:

Parameter	Type	Description
succeeded	integer	Number of commits that were built successfully in the period
failed	integer	Number of commits that failed to build in the period
total	integer	Total number of commits processed in the period

6.2.8 GET /metrics

Retrieve statistics on the absolute number of builds for the builder, in Prometheus format.

Normal response codes: 200

Error response codes: 400

No parameters.

Response:

In text/plain format:

```
# HELP dlrn_builds_succeeded_total Total number of successful builds
# TYPE dlrn_builds_succeeded_total counter
dlrn_builds_succeeded_total{baseurl="http://trunk.rdoproject.org/centos8/"} 9296.0
# HELP dlrn_builds_failed_total Total number of failed builds
# TYPE dlrn_builds_failed_total counter
dlrn_builds_failed_total{baseurl="http://trunk.rdoproject.org/centos8/"} 244.0
# HELP dlrn_builds_retry_total Total number of builds in retry state
# TYPE dlrn_builds_retry_total counter
dlrn_builds_retry_total{baseurl="http://trunk.rdoproject.org/centos8/"} 119.0
# HELP dlrn_builds_total Total number of builds
# TYPE dlrn_builds_total counter
dlrn_builds_total{baseurl="http://trunk.rdoproject.org/centos8/"} 9659.0
```

6.2.9 GET /api/graphql

Query the GraphQL interface. The available GraphQL schema is described in detail in its own documentation.

6.2.10 POST /api/last_tested_repo

Get the last tested repo since a specific time (optionally for a CI job), and add an "in progress" entry in the CI job table for this.

If a job_id is specified, the order of precedence for the repo returned is:

- The last tested repo within that timeframe for that CI job.
- The last tested repo within that timeframe for any CI job, so we can have several CIs converge on a single repo.
- The last "consistent" repo, if no repo has been tested in the timeframe.

If `sequential_mode` is set to true, a different algorithm is used. Another parameter `previous_job_id` needs to be specified, and the order of precedence for the repo returned is:

- The last tested repo within that timeframe for the CI job described by `previous_job_id`.
- If no repo for `previous_job_id` is found, an error will be returned

The sequential mode is meant to be used by CI pipelines, where a CI (n) job needs to use the same repo tested by CI (n-1).

Normal response codes: 201

Error response codes: 400, 415

Request:

Parameter	Type	Description
max_age	integer	Maximum age (in hours) for the repo to be considered. Any repo tested or being tested after "now - max_age" will be taken into account. If set to 0, all repos will be considered.
reporting_job_id	string	Name of the CI that will add the "in progress" entry in the CI job table
success	boolean (optional)	If set to a value, find repos with a successful/unsuccessful vote (as specified). If not set, any tested repo will be considered.
job_id	string (optional)	name of the CI that sent the vote. If not set, no filter will be set on CI.
sequential_mode	boolean (optional)	Use the sequential mode algorithm. In this case, return the last tested repo within that timeframe for the CI job described by previous_job_id. Defaults to false.
previous_job_id	string (optional)	If sequential_mode is set to true, look for jobs tested by the CI identified by previous_job_id.
component	string (optional)	Only report votes associated to this component

Response:

Parameter	Type	Description
commit_hash	string	commit_hash of tested repo
distro_hash	string	distro_hash of tested repo
extended_hash	string	extended_hash of tested repo
success	boolean	whether the test was successful or not
job_id	string	name of the CI sending the vote
in_progress	boolean	True -> is this CI job still in-progress?
timestamp	integer	Timestamp for this CI Vote (taken from the DLRN system time)
user	string	user who created the CI vote
component	string	Component associated to the commit/distro hash

6.2.11 POST /api/report_result

Report the result of a CI job.

It is possible to report results on two sets of objets:

- A commit, represented by a `commit_hash` and a `distro_hash`.
- An aggregated repo, represented by an `aggregate_hash`.

One of those two parameters needs to be specified, otherwise the call will return an error.

Normal response codes: 201

Error response codes: 400, 415, 500

Request:

Parameter	Type	Description
job_id	string	name of the CI sending the vote
commit_hash	string	commit_hash of tested repo
distro_hash	string	distro_hash of tested repo
extended_hash	string (optional)	extended_hash of the tested repo. If not set, the latest commit with the commit_hash/distro_hash combination will be used
aggregate_hash	string	hash of the aggregated repo that was tested
url	string	URL where to find additional information from the CI execution
timestamp	integer	Timestamp (in seconds since the epoch)
success	boolean	Was the CI execution successful?
notes	Text	Additional notes (optional)

Response:

Parameter	Type	Description
job_id	string	name of the CI sending the vote
commit_hash	string	commit_hash of tested repo
distro_hash	string	distro_hash of tested repo
extended_hash	string	extended_hash of tested repo
url	string	URL where to find additional information from the CI execution
timestamp	integer	Timestamp (in seconds since the epoch)
in_progress	boolean	False -> is this CI job still in-progress?
success	boolean	Was the CI execution successful?
notes	Text	Additional notes
user	string	user who created the CI vote
component	string	Component associated to the commit/distro hash

6.2.12 POST /api/promote

Promote a repository. This can be implemented as a local symlink creation in the DLRN worker, or any other form in the future.

Note the API will refuse to promote using promote_name=”consistent” or “current”, since those are reserved keywords for DLRN. Also, a commit that has been purged from the database cannot be promoted.

When the projects.ini `use_components` option is set to `true`, an aggregated repo file will be created, including the repo files of all components that were promoted with the same promotion name. The hash of that file will be returned as `aggregated_hash`. If the option is set to `false`, a null value will be returned.

Normal response codes: 201

Error response codes: 400, 403, 410, 415, 500

Request:

Parameter	Type	Description
commit_hash	string	commit_hash of the repo to be promoted
distro_hash	string	distro_hash of the repo to be promoted
extended_hash	string	extended_hash of the repo to be promoted (optional). If not specified, the API will take the last commit built with the commit and distro hash.
promote_name	string	name to be used for the promotion. In the current implementation, this is the name of the symlink to be created

Response:

Parameter	Type	Description
commit_hash	string	commit_hash of the promoted repo
distro_hash	string	distro_hash of the promoted repo
extended_hash	string	extended_hash of the promoted repo
repo_hash	string	Repository hash, composed of the commit_hash and short distro_hash
repo_url	string	Full URL of the promoted repository
promote_name	string	name used for the promotion
component	string	Component associated to the commit/distro hash
timestamp	integer	Timestamp (in seconds since the epoch)
user	string	user who created the promotion
agggregate_hash	string	Hash of the aggregated repo file, when using components

6.2.13 POST /api/promote-batch

Promote a list of commits. This is the equivalent of calling /api/promote multiple times, one with each commit/distro_hash combination. The only difference is that the call is atomic, and when components are enabled, the aggregated repo files are only updated once.

If any of the individual promotions fail, the API call will try its best to undo all the changes to the file system (e.g. symlinks).

Note the API will refuse to promote using promote_name="consistent" or "current", since those are reserved keywords for DLRN. Also, a commit that has been purged from the database cannot be promoted.

Normal response codes: 201

Error response codes: 400, 403, 410, 415, 500

Request:

The JSON input will contain an array where each item contains:

Parameter	Type	Description
commit_hash	string	commit_hash of the repo to be promoted
distro_hash	string	distro_hash of the repo to be promoted
extended_hash	string	extended_hash of the repo to be promoted (optional). If not specified, the API will take the last commit built with the commit and distro hash.
promote_name	string	name to be used for the promotion. In the current implementation, this is the name of the symlink to be created

Response:

Parameter	Type	Description
commit_hash	string	commit_hash of the promoted repo
distro_hash	string	distro_hash of the promoted repo
extended_hash	string	extended_hash of the promoted repo
repo_hash	string	Repository hash, composed of the commit_hash and short distro_hash
repo_url	string	Full URL of the promoted repository
promote_name	string	name used for the promotion
component	string	Component associated to the commit/distro hash
timestamp	integer	Timestamp (in seconds since the epoch)
user	string	user who created the promotion
agggregate_hash	string	Hash of the aggregated repo file, when using components

This is the last promoted commit.

6.2.14 POST /api/remote/import

Import a commit built by another instance. This API call mimics the behavior of the `dlrn-remote` command, with the only exception of not being able to specify a custom rdoinfo location.

Normal response codes: 201

Error response codes: 400, 415, 500

Request:

Parameter	Type	Description
repo_url	string	Base repository URL for remotely generated repo

Response:

Parameter	Type	Description
repo_url	string	Base repository URL for imported remote repo

6.3 Running the API server using WSGI

6.3.1 Requirements

It is possible to run the DLRN API server as a WSGI process in Apache. To do this, you need to install the following packages:

```
$ sudo yum -y install httpd mod_wsgi
```

6.3.2 WSGI file and httpd configuration

To run the application, you need to create a WSGI file. For example, create `/var/www/dlrn/dlrn-api.wsgi` with the following contents:

```python
import os
import sys
sys.path.append('/home/centos-master/.venv/lib/python2.7/site-packages/')

def application(environ, start_response):
    os.environ['CONFIG_FILE'] = environ['CONFIG_FILE']
    from dlrn.api import app
    return app(environ, start_response)
```

You need to change the path appended using `sys.path.append` to be the path to the virtualenv where you have installed DLRN.

Then, create an httpd configuration file to load the WSGI application. The following is an example file, named `/etc/httpd/conf.d/wsgi-dlrn.conf`:

```
<VirtualHost *>
    ServerName example.com

    WSGIDaemonProcess dlrn  user=centos-master group=centos-master threads=5
    WSGIScriptAlias / /var/www/dlrn/dlrn-api-centos-master.wsgi
    SetEnv CONFIG_FILE /etc/dlrn/dlrn-api.cfg

    <Directory /var/www/dlrn>
        WSGIProcessGroup dlrn
        WSGIApplicationGroup %{GLOBAL}
        WSGIScriptReloading On
        WSGIPassAuthorization On
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

Set `CONFIG_FILE` to the path of the DLRN configuration file, and make sure you specify the right user and group for the `WSGIDaemonProcess` line.

Set `DLRN_DEBUG` to enable debug logs and set `DLRN_LOG_FILE` to the path of a logfile (default false). If `DLRN_LOG_FILE` is not set, then the logs are redirected to logs set by ErrorLog and CustomLog in the apache conf file.

Set `API_AUTH_DEBUG` to enable debug logs for API authentication (default false) and set `API_AUTH_LOG_FILE` to the path of an API authentication logfile. If `API_AUTH_LOG_FILE` is not set, then the logs are redirected to logs set by ErrorLog and CustomLog in the apache conf file.

DLRN_DEBUG also specifies if debug when logs are redirected to logs set by ErrorLog and CustomLog in the apache conf file.

Those variables are also applied within the CONFIG_FILE with higher precedence.

6.3.3 DLRN API configuration

The DLRN API take a default configuration from file dlrn/api/config.py. Since it may not match your actual configuration when deployed as an WSGI application, you can create a configuration file, /etc/dlrn/dlrn-api.cfg in the above example, with the following syntax:

```
DB_PATH = 'sqlite:////home/centos-master/DLRN/commits.sqlite'
REPO_PATH = '/home/centos-master/DLRN/data/repos'
CONFIG_FILE = 'projects.ini'
```

Where DB_PATH is the path to the SQLite database for your environment, REPO_PATH will point to the base directory for the generated repositories, and CONFIG_FILE will point to the projects.ini file used when running DLRN.

6.4 User management

There is a command-line tool to manage DLRN API users:

```
usage: dlrn-user [-h] [--config-file CONFIG_FILE] {create,delete,update} ...

arguments:
  -h, --help            show this help message and exit
  --config-file CONFIG_FILE
                        Config file. Default: projects.ini

subcommands:
  available subcommands

  {create,delete,update}
    create              Create a user
    delete              Delete a user
    update              Update a user
```

6.4.1 User creation

Use the create subcommand to create a new user.

```
$ dlrn-user create --username foo --password bar
```

If you do not specify a password in the command-line, you will be prompted to enter one interactively.

6.4.2 User update

You can use the update subcommand to change user data. Currently, only the password can be changed.

```
$ dlrn-user update --username foo --password new
```

6.4.3 User deletion

Use the `delete` subcommand to delete a user.

```
$ dlrn-user delete --username foo
```

The command will ask for confirmation, and you have to type "YES" (without the quotes) in uppercase to delete the user. You can also avoid the confirmation request by adding the `--force` parameter.

```
$ dlrn-user delete --username foo --force
```

CHAPTER 7

GraphQL information

7.1 GraphQL schema

This page describes the schema definition used for the GraphQL types and queries available through the DLRN API.

You can generate a human-readable version of the schema by running the following script:

```
from dlrn.api.graphql import schema
from graphql.utils import schema_printer

schema_str = schema_printer.print_schema(schema)
print(schema_str)
```

7.1.1 Types

The Commit type is converted from its schema in the database.

.. code-block:

```
type CIVote {
     id: ID!
     commitId: Int!
     ciName: String
     ciUrl: String
     ciVote: Boolean
     ciInProgress: Boolean
     timestamp: Int
     notes: String
     user: String
     component: String
     commit: Commit
}
```

Like Commit type, CIVote is converted from its schema to database.

The CIVoteAgg is converted from CIVote_Aggregate DB schema.

The PackageStatus type is generated directly in Graphene.

7.1.2 Queries

All queries should conform to the GraphQL language. When more than one item is returned, they will be sorted by descending id order, which means newer commits or CI Votes are displayed first.

Note that you will need to specify which fields from the return type you want to get. See the GraphQL tutorial for additional details.

Available queries:

- commits

Arguments:

- projectName: limit the results to the commits belonging to the specified project name.
- component: limit the results to the commits belonging to the specified component.
- status: limit the results to the commits with the specified status.
- offset: return the results after the specified entry.
- limit: return a maximum amount of commits (100 by default, cannot be higher than 100).
- commitHash: limit the results to the commits containing the specified commit hash.
- distroHash: limit the results to the commits containing the specified distro hash.
- extendedHash: limit the results to the commits containing the specified extended hash. In this case, extendedHash can contain wildcards in SQL format, so setting extendedHash to “foo%” in the query will return all commits with an extended hash that starts by “foo”.
- civote

Arguments: - commitId: limit the results to the civote belonging to the commit id. - ciName: limit the results to the civote belonging to the CI name. - ciVote: limit the results to the civote belonging to the voting CI. - ciInProgress: limit the results to the civote belonging to “In Progress” state. - timestamp: limit the results to the civote belonging to the specified timestamp. - user: limit the results to the civote belonging to the specified user. - component: limit the results to the civote belonging to the specified component. - offset: return the results after the specified entry. - limit: return a maximum amount of commits (100 by default, cannot be higher than 100).

- civoteAgg

Arguments: - refHash: limit the results to the civote_aggregation belonging to the specified reference hash. - ciName: limit the results to the civote_aggregation belonging to the specified CI name. - ciVote: limit the results to the civote_aggregation belonging to the specified CI vote. - ciInProgress: limit the results to the civote_aggregation belonging to the specified CI in progress state. - timestamp: limit the results to the civote_aggregation belonging to the specified timestamp. - user: limit the results to the civote_aggregation belonging to the specified user. - offset: return the results after the specified entry. - limit: return a maximum amount of commits (100 by default, cannot be higher than 100).

- packageStatus

Arguments: - projectName: limit the results to the status of the specified project name. - status: limit the results to the packages with the specified status.

7.2 Querying the GraphQL endpoint

As described in the GraphQL website, when GraphQL is served over HTTP it is possible to run queries using both GET and POST methods.

7.2.1 GET example

```
$ curl 'http://localhost:5000/api/graphql?query=\{commits\{component%20projectName\}\}
↪'
```

Note that in the curl command line we are escaping braces and replacing blank spaces with %20. The equivalent query when run from a broswer would be `http://localhost:5000/api/graphql?query={ commits { component projectName } }`.

7.2.2 POST example

```
$ curl http://localhost:5000/api/graphql -H POST -d 'query={ commits { component␣
↪projectName } }'
```

In this case, we are using a POST method, and the query is JSON-encoded. Note that it is also possible to use a GET method with a JSON-encoded payload.

CHAPTER 8

Contributing

8.1 Setting up a development environment in an OpenStack VM using cloud-init

The following cloud-config script can be passed as a –user-data argument to *nova boot*. This will result in a fully operational DLRN environment to hack on.

```
#cloud-config
disable_root: 0

users:
  - default

package_upgrade: true

packages:
  - vim
  - git
  - policycoreutils-python-utils

runcmd:
  - yum -y install epel-release
  - yum -y install puppet
  - git clone https://github.com/rdo-infra/puppet-dlrn /root/puppet-dlrn
  - cd /root/puppet-dlrn
  - puppet module build
  - puppet module install pkg/jpena-dlrn-*.tar.gz
  - cp /root/puppet-dlrn/examples/common.yaml /var/lib/hiera
  - puppet apply --debug /root/puppet-dlrn/examples/site.pp 2>&1 | tee /root/puppet.
↪log

final_message: "DLRN installed, after $UPTIME seconds."
```

8.2 Setting up a development environment manually

Follow the instructions from the Setup section of README.rst to manually setup a development environment.

8.3 Submitting pull requests

Pull requests submitted through GitHub will be ignored. They should be sent to SoftwareFactory's Gerrit instead, using git-review. The usual workflow is:

```
$ sudo yum install git-review  (you can also use pip install if needed)
$ git clone https://github.com/softwarefactory-project/DLRN
<edit your files here>
$ git add <your edited files>
$ git commit
$ git review
```

Once submitted, your change request will show up here:

https://softwarefactory-project.io/r/#/q/project:DLRN+status:open

8.4 Generating the documentation

Please note that the RDO Packaging Documentation also contains instructions for DLRN.

The documentation is generated with Sphinx. To generate the documentation, go to the documentation directory and run the make file:

```
$ cd DLRN/doc/source
$ make html
```

The output will be in DLRN/doc/build/html

CHAPTER 9

DLRN internals

This document aims at describing the inner workings of DLRN, so a new contributor can get up to speed as quickly as possible.

9.1 Main concepts

The following basic concepts are used in DLRN. You'll need to get used to them if you want to understand the code:

- **Source Git**: DLRN will always take the source code to build the package from a Git repository, regardless of the `Source0` entry in the spec file.
- **Distgit**: spec files are assumed to be present in a Git repository. DLRN has a driver-based mechanism to allow different options for the distgit location, see the *Package Info Drivers* section.
- **Project**: each project corresponds with a package to be built. A package may define a number of subpackages in the spec file, but a single source RPM file is always created. The DLRN driver mechanism allows us to have different sources of information for the project list, such as rdoinfo or a single git repo.
- **Commit**: a commit is the main abstraction used by DLRN. It aggregates all information related to each package built, such as:
  - Project name
  - Hash of the commit from the source git
  - Hash of the commit from the distgit
  - Build status (successful or not)
  - Name of the rpms

9.2 Directory structure

The DLRN codebase is structured as follows:

- **doc/**: project documentation
- **scripts/**: several useful scripts, used by CI and DLRN itself, plus some other miscellaneous files.
  - *build_rpm.sh*: script that calls mock to build rpm files (see the *Building packages* section).
  - *submit_review.sh*: script to open a Gerrit review after a build failure (see the *Error reporting* section).
  - *centos.cfg*, *centos8.cfg*, *fedora.cfg* and *redhat.cfg*: base mock configurations for CentOS 7, CentOS 8, Fedora and RHEL 8 builders. For a RHEL 8 builder, you will have to make sure the appropriate base repos are configured, since those are not publicly available. These base configurations can be located in a separate directory, defined by the `configdir` option in projects.ini.
- **dlrn/**: main DLRN code
  - *build.py*: build functions, described in detail in the *Building packages* section.
  - *config.py*: general configuration management.
  - *db.py*: database-related code.
  - *notifications.py*: error reporting functions.
  - *purge.py*: dlrn-purge command, used to reduce the disk usage on long-running instances.
  - *reporting.py*: reporting functions, create simple HTML reports of the repository status.
  - *repositories.py*: functions required to clone a git repo and get information from it.
  - *rpmspecfile.py*: basic rpm spec file parsing to be able to get package names and dependencies.
  - *rsync.py*: synchronizes yum repositories between servers, used to have a multi-node architecture.
  - *shell.py*: main file, reads command-line arguments and launches the build process.
  - *utils.py*: miscellaneous utilities.
  - **api/**: DLRN API code, described in detail in its own page.
  - **drivers/**: modular drivers for project listing and distgit location, described in the *Package Info Drivers* section. We are also including modular drivers for different build methods.
  - **migrations/**: Alembic scripts for database maintenance.
  - **stylesheets/**: contains a CSS file used by the reporting module.
  - **templates/**: contains Jinja2 templates, also used by the reporting module.
  - **tests/**: unit tests.

9.3 High level algorithm

When DLRN is run, the following (simplified) sequence of events is executed:

```
fetch information for available projects
for each project
    find last processed commit
    refresh source git
    find last commit in source git and distgit
    if any commit is later than last_processed_commit
        add commit to list of commits to be processed
for each commit_to_be_processed
    build package
```

(continues on next page)

(continued from previous page)

```
    create yum repo with built package and the latest versions of every other package
    if errors
        report via e-mail or Gerrit review if configured
    store commit in DB
    generate HTML report
```

9.4 Configuration

DLRN uses a simple INI file for its configuration. Most config options are located in the `[DEFAULT]` section and read during startup. Only driver-specific options have their own section.

The dlrn/config.py file defines a `ConfigOptions` class, that will create an object including all parsed options.

9.5 Package Info Drivers

Package info drivers are derived from the `PkgInfoDriver` class (see dlrn/drivers/pkginfo.py), and are used to:

- Define a list of projects (packages) to be built
- Define the source and distgit repos for each project
- Fetch the new commits for each project's source and distgit repos
- Pre-process spec files, if needed

Each driver must provide the following methods:

- **getpackages()**. This method will return a list of dictionaries. Each individual dict must contain the following mandatory keys (others are optional):
  - 'name': package name
  - 'upstream': URL for source repo
  - 'master-distgit': URL for distgit repo
  - 'maintainers': list of e-mail addresses for package maintainers
- **getinfo()**. This method will return a list of commits to be processed for a specific package.
- **preprocess()**. This method will run any required pre-processing for the spec files. If the `custom_preprocess` variable is defined in `projects.ini`, the external program(s) or script(s) defined in the variable will be executed as the last step of the pre-processing.
- **distgit_dir()**. This method will return the distgit repo directory for a given package name.

You can check the code of the existing rdoinfo driver and gitrepo driver to see their implementation specifics. If you create a new driver, you need to add the project name to the `projects.ini` configuration file, and if you need any new options, be sure to add them to a driver-specific section (see the *Configuration* section for details).

9.6 Package Build Drivers

Package build drivers are derived from the `BuildRPMDriver` class (see dlrn/drivers/buildrpm.py), and are used to perform the actual package build from an SRPM file.

Each driver must provide the following method:

- **build_package** This method will take an output directory, where the SRPM is located, and build it using the driver-specific method.

You can check the code of the existing mock driver to see its implementation specifics. If you create a new driver, you need to add the project name to the `projects.ini` configuration file, and if you need any new options, be sure to add them to a driver-specific section (see the *Configuration* section for details).

9.7 Building packages

The package build logic is included in build.py. There we have several functions:

- **build()**. This is the function called externally. It gathers some configuration options and parameters, then calls `build_rpm_wrapper` to launch the build process and returns a list with the built rpms.
- **build_rpm_wrapper()**. This wrapper function prepares the mock configuration file to be used during the build using the configuration. It will also add the most current repository to the mock configuration, so we can use packages in the current repository as dependencies during the build. Then, it will spawn a Bash script, `build_srpm.sh` to build the source RPM, and call the appropriate build driver to generate the binary RPM.

The `build_srpm.sh` script takes care of creating the source RPM. Some magic is required to build it, specifically:

- The script tries to determine a version and release number for the package. This version number should be compatible with the Fedora guidelines, and allow upgrades **from** and **to** packages from stable releases, which is not always easy. We use the following algorithm:
  - For Python projects, take the output from `python setup.py --version`. Most OpenStack projects use PBR, which gives us proper pre-versioning after a tagged release.
  - For Puppet projects, we take the version from the `metadata.json` or `Modulefile` files, if available, and increase the .Z version if there are any commits after the tagged release.
  - For other projects, we take the version number from the latest git tag.
  - If everything fails, default to version 0.0.1.
  - The release number is always 0.<date>.<upstream source commit short hash>.
- A tarball is generated using `python setup.py sdist` for Python projects, `gem build` for Ruby gems, and tar for any other project. Then, the spec file is updated to use this tarball as `Source0`, and a source RPM is created.

The binary RPM is built from the SRPM using a the build driver specified in `projects.ini`. This can be done using Mock, Copr, Brew, or any other tool, provided that the required driver is available.

9.8 Hashed yum repositories

Each build is stored on a separate directory. A hashed structure is used for the directories, such as `cd/af/cdaf2c77d974d5e794909313dceb3554be69a42e_4b1619fe`. In this structure, `cdaf2c77d974d5e794909313dceb3554be69a42e` is the commit hash for the source git repo, and `4b1619fe` is the short hash for the distgit commit. The first two directory levels (`cd/af`) are taken from the commit hash.

9.9 Component support

DLRN now supports the concept of *components* inside a repository. We can use components to divide the packages in a repo into logical aggregations. For example, in the OpenStack use case, we could have separate components for those packages related to networking, compute, storage, etc.

Currently, only the `RdoInfoDriver` and `DownstreamInfoDriver` package info drivers supports this. When components are defined, and enabled with the `use_components=True` option in `projects.ini`, DLRN will change its behavior in the following ways:

- Hashed yum repositories will change their paths, including a component part. For example, a commit for a package in the compute component will use hash `component/compute/cd/af/cdaf2c77d974d5e794909313dceb3554be69a42e_4b1619fe`.
- Each component will have a separate repository (`component/compute`, `component/network``and so on), and the ``current` and `consistent` symlinks will also be relative to each component.
- To preserve compatibility with instances without component support, the top-level `current` and `consistent` symlinks will be replaced by a `current` and `consistent` directory. Each directory will contain a single .repo file, and that file will aggregate the .repo files for the current/consistent repositories of all components.

9.10 Post-build actions

After a package is built, we need to create a package repository with the latest version for every package in the project list. The `post_build()` function in `shell.py` takes care of that. The idea behind this is that the repo for each build will contain the most current version of each package to date. This behavior can be skipped if the `--no-repo` command-line option is provided, so only the build package and logs will be stored.

To minimize the amount of storage used for each repo, DLRN does not copy the packages to the current hashed directory. Instead, `post_build()` iterates through the list of packages, finding the RPMs for their latest successful builds, and symlinks them in the current hashed directory.

It is probably easier to understand with an example:

- Initially, we only have source commit 010b0a and distgit commit 020202 for project foo, then its hashed repo will look like:

  ```
  01/0b/010b0a_020202/foo-<version>.el7.centos.noarch.rpm
  ```

- Then, we build project bar, with source commit 030303 and distgit commit 040404. Its hashed repo will be:

  ```
  03/03/030303_040404/bar-<version>.el7.centos.noarch.rpm
  03/03/030303_040404/foo-<version>.el7.centos.noarch.rpm -> ../../../01/0b/010b0a_
  ↪020202/foo-<version>.el7.centos.noarch.rpm
  ```

  And the same process will be followed for every new package.

9.11 Error reporting

DLRN allows two different ways to notify build errors, both included in notifications.py:

- A notification e-mail, sent using the `sendnotifymail()` function. The mail recipient list is taken from the `maintainers` project property.

- A Gerrit review. This option makes use of a utility script `submit_review.sh` and the configured options in options.ini to create the review. It also adds the project maintainers to the generated review.

9.12 API internals

The API is described in detail in its own documentation.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search