Pulp deb Support

Release 2.7.0.dev

Sep 08, 2020

Contents

The `pulp_deb` plugin extends the pulpcore python package with the ability to host deb packages within APT repositories. This plugin is a part of the Pulp Project, and assumes some familiarity with the pulpcore documentation.

If you are just getting started with the plugin, read the high level *feature overview* first. See *workflows* for detailed usage examples. See the *REST API* documentation for an exhaustive feature reference.

CHAPTER 1

Table of Contents

1.1 Installation

All REST API examples below use httpie to perform the requests. The `httpie` commands below assume that the user executing the commands has a `.netrc` file in the home directory. The `.netrc` should have the following configuration:

```
machine localhost
login admin
password admin
```

If you configured the `admin` user with a different password, adjust the configuration accordingly. If you prefer to specify the username and password with each request, please see `httpie` documentation on how to do that.

1.1.1 Install `pulpcore`

Please see the pulpcore installation instructions.

1.1.2 Install `pulp_deb` Plugin

This document assumes that you have used the pulpcore installation to install pulpcore into a the virtual environment `pulpvenv`.

Users should install from **either** PyPI or source.

From Source

```
sudo -u pulp -i
source ~/pulpvenv/bin/activate
cd pulp_deb
```

(continues on next page)

(continued from previous page)

```
pip install -e .
django-admin runserver 24817
```

1.1.3 Make and Run Migrations

```
pulp-manager makemigrations pulp_deb
pulp-manager migrate pulp_deb
```

1.1.4 Run Services

```
pulp-manager runserver
gunicorn pulpcore.content:server --bind 'localhost:24816' --worker-class 'aiohttp.
↪GunicornWebWorker' -w 2
sudo systemctl restart pulpcore-resource-manager
sudo systemctl restart pulpcore-worker@1
sudo systemctl restart pulpcore-worker@2
```

1.2 Feature Overview

This chapter aims to give a high level overview of what features the plugin currently supports, as well as any known limitations. The aim is to provide users with enough information to make informed decisions about how they may or may not want to use this plugin, as well as to set realistic expectations on what will and will not work.

For detailed usage examples, see *workflows* instead. See the *REST API* documentation for an exhaustive feature reference.

1.2.1 Content Types

Whether they are obtained via *synchronization* or *direct upload*, the `pulp_deb` plugin knows several different content types that it stores in the Pulp database.

Since each content type has its own REST API endpoint, you can find detailed descriptions in the *pulp_deb REST API documentation*. Content types may be associated with certain artifacts (aka files) or contain only metadata. For example, every content unit of the `Packages` content type is associated with exactly one `.deb` package file. In other words, each content unit of this type represents exactly one `.deb` package.

Currently, the plugin has dedicated content types for various types of metadata, as well as `.deb` (binary) packages, and `.udeb` installer packages. However, the latter can currently only be used in conjunction with the *verbatim publisher*.

1.2.2 Repository Synchronization

Synchronizing upstream repositories is one of two ways to obtain content for the `pulp_deb` plugin. See *package uploads* for the other. The aim is for the plugin to be able to synchronize (and publish) arbitrary (valid) APT repositories.

When synchronizing an upstream repository, only *content types* supported by the `pulp_deb` plugin are downloaded. Source packages, for example, are currently unsupported.

Even if a particular content type is downloaded during synchronization, it depends on the publisher that is used (*verbatim* or *standard APT* publisher), whether that content is actually served by the Pulp content app as part of the Pulp

distribution being created. For example, the plugin's APT publisher does not use the downloaded upstream metadata files, but rather generates its own. As another example, `.udeb` installer packages are currently only supported by the verbatim publisher.

Filtered Synchronization

Synchronization works via the use of so called *Pulp remotes*, which describe the upstream repository you intend to synchronize. It is possible to synchronize only a subset of a given upstream repository by specifying a set of "distributions", "components", and "architectures" in the remote. Specifying the desired distributions is mandatory, while not specifying any components or architectures is interpreted as: "synchronize all that are available".

Note: There will not be any errors if you specify components or architectures that do not exist for a given upstream distribution. This allows you to filter for components and architectures that may not be present in all of the upstream distributions, but it may also lead to unexpected results. For example, if you have made a typo, your desired component and/or architecture will simply be missing from your Pulp repository, without any failures or warnings.

Signature Verification

You may provide your remotes with the relevant (public) GPG key for `Release` file verification.

Note: For APT repositories, only the `Release` file of each distribution is signed. This file contains various checksums for all other metadata files contained within the distribution, which in turn contain the checksums of the packages themselves. As a result, signing the `Release` file is sufficient to guarantee the integrity of the entire distribution.

When synchronizing an upstream repository using a remote with GPG key, any `Release` (or `InRelease`) files that do not have a valid signature are discarded. If, for a given distribution, there is no `Release` file that can be successfully verified, a `NoReleaseFile` error is thrown and the sync fails.

1.2.3 Package Uploads

Rather than synchronizing upstream repositories, it is also possible to upload `.deb` package files to the `pulp_deb` plugin in a variety of ways. See the corresponding *workflow documentation* for more information. In general, uploading content works the same way as for any other Pulp plugin, so you may also wish to consult the pulpcore upload documentation.

Important: There is currently no way of associating an uploaded `.deb` package with an existing distribution or component. There is also no way of manually creating a distribution and component to associate it with in the first place. As a result, manually uploaded packages will only show up in your publications, if you are using the "simple" publisher. For more information, see *simple and structured publishing* below.

Note: As a matter of best practice, the existence of multiple Debian packages with the same name, version, and architecture (but different content/checksum) should be avoided. Since the existence of such packages may be beyond the control of the `pulp_deb` user, the plugin takes a maximally permissive approach: Users can upload arbitrary (valid) packages to the Pulp database, but they cannot add multiple colliding packages of the same type (`.deb` or `.udeb`), to a single Pulp repository version. If users attempt to add one or more packages to a Pulp repository, and there are collisions with packages from the previous repository version, then the older packages will automatically be

removed. If there are still collisions in the new repository version, an error is thrown and the task will fail. (This latter case can only happen if users attempt to add several colliding packages in a single API call.)

1.2.4 Simple and Structured Publishing

You can create an APT publication from your synchronized repositories or your uploaded packages, using the `/pulp/api/v3/publications/deb/apt/` *REST API* endpoint. A publication must use `simple` or `structured` mode (or both).

The simple publisher will publish all packages associated with the pulp repository version you are using in a single APT distribution named `default`, which will contain a single component named `all`. That is, the simple publisher will add a single `Release` file at `dists/default/Release` to your published repository. There will be one package index for each architecture for which there are packages (in addition to `dists/default/all/binary-all/Packages`, which will always be created, but may be empty).

Important: The simple publisher is currently the only way to include manually uploaded packages in your distribution. Be sure to use `simple=true` if you have uploaded packages (as opposed to synchronized them) to your repository.

The structured publisher will publish all the distributions (aka releases), components, and architectures, that were synchronized to the Pulp repository being published. These various distribution, component, and architecture combinations, will contain the same packages as the upstream originals. However, unlike the *verbatim publisher*, the APT publisher will generate all new metadata files for the publication. It will also use a default `pool/` folder structure regardless of the package file locations used by the upstream repositories.

Important: Since synchronization is currently the only supported way to obtain the needed metadata content units, the structured publisher only makes sense if you have synchronized some upstream APT repository into your Pulp instance.

Finally, the APT publisher (both structured and simple mode) will never append `Architecture: all` type packages to any architecture specific package indices. It will always publish dedicated `binary-all` package indices. This behaviour is irrespective of how any upstream repositories might have handled such packages.

Metadata Signing

The `pulp_deb` plugin allows you to sign the `Release` files created by the simple or structured publishers by providing your publication with a signing service of type `AptReleaseSigningService` at the time of creation.

Important: We currently lack a workflow documentation for creating and using an `AptReleaseSigningService`. Until we get around to writing one, you can use the following resources to help you get started:

- The pulpcore metadata signing docs describe the process for creating an `AsciiArmoredDetachedSigningService`, which is largely analagous to creating an `AptReleaseSigningService`.
- The signing service setup script used by the `pulp_deb` test suite.
- The signing service script example used by the `pulp_deb` test suite.

1.2.5 Verbatim Publishing

In addition to the `pulp_deb` plugin's main *APT publisher*, there is also the "verbatim" publisher using a seperate *REST API* endpoint at `/pulp/api/v3/publications/deb/verbatim/`.

The verbatim publisher will recreate the synced subset of any upstream repositories exactly. It could also be referred to as "mirror mode". If you have used *filtered synchronization* to obtain your repository, this reduces the synced subset as one would expect. The synced subset currently includes `.deb` packages, `.udeb` installer packages, any upstream `Release` files, package indices, installer file indices, as well as installer and translation files.

The verbatim publisher (in combination with synchronization of a suitable upstream repository) is currently the only way to create a Pulp APT repository that can be used to install hosts with the Debian installer.

All files included in the verbatim publication will retain the exact same checksum that they had in the upstream repository. Any upstream `Release` file signatures are simply retained. As a result, hosts consuming the Pulp distribution can use the same GPG keys for repository verification as if they were attached directly to the upstream repository you synchronized. On the flip side, it is currently not possible to sign a verbatim publication with your own *signing services*.

1.3 Workflows

This chapter assumes you have a working pulp *installation* including the `pulp_deb` plugin.

All REST API examples below use httpie to perform the requests. The `httpie` commands below assume that the user executing the commands has a `.netrc` file in the home directory. The `.netrc` should have the following configuration:

```
machine localhost
login admin
password admin
```

If you configured the `admin` user with a different password, adjust the configuration accordingly. If you prefer to specify the username and password with each request, please see `httpie` documentation on how to do that.

In order to make the workflows usable via copy and paster, we make use of environmental variables in all examples:

```
export BASE_ADDR=http://<hostname>:24817
```

This chapter is structured into the following subsections:

1.3.1 Synchronize a Repository

Users can populate their repositories with content from an external source by syncing their repository.

This example syncs the nginx repository from nginx.org for Debian Buster.

Create a Repository

Start by creating a new repository named `nginx`:

```
http post $BASE_ADDR/pulp/api/v3/repositories/deb/apt/ name="nginx"
```

This will return a `201 Created` response:

```
{
    "description": null,
    "latest_version_href": "/pulp/api/v3/repositories/deb/apt/1c8734dd-d4cb-4c2f-811a-
↪87235f786444/versions/0/",
    "name": "nginx",
    "pulp_created": "2020-06-29T07:17:34.789398Z",
    "pulp_href": "/pulp/api/v3/repositories/deb/apt/1c8734dd-d4cb-4c2f-811a-
↪87235f786444/",
    "versions_href": "/pulp/api/v3/repositories/deb/apt/1c8734dd-d4cb-4c2f-811a-
↪87235f786444/versions/"
}
```


Create a Remote

Creating a remote object informs Pulp about an external content source:

```
http post $BASE_ADDR/pulp/api/v3/remotes/deb/apt/ name="nginx.org" url="http://nginx.
↪org/packages/debian" distributions="buster"
```

This will return a `201 Created` response:


```
{
    "architectures": null,
    "ca_cert": null,
    "client_cert": null,
    "client_key": null,
    "components": null,
    "distributions": "buster",
    "download_concurrency": 20,
    "gpgkey": null,
    "name": "nginx.org",
    "password": null,
    "policy": "immediate",
    "proxy_url": null,
    "pulp_created": "2020-06-29T07:18:44.906572Z",
    "pulp_href": "/pulp/api/v3/remotes/deb/apt/efd3848c-6d7b-4f25-baaf-ad05b52a5220/",
    "pulp_last_updated": "2020-06-29T07:18:44.906584Z",
    "sync_installer": false,
    "sync_sources": false,
    "sync_udebs": false,
    "tls_validation": true,
    "url": "http://nginx.org/packages/debian",
    "username": null
}
```


Sync Repository with Remote

Use a repository with a specified remote object to start the synchronization process. This is telling Pulp to fetch content from the remote and add it to the repository. The remote is the parameter to the sync endpoint on the repository.

```
http post $BASE_ADDR/pulp/api/v3/repositories/deb/apt/<uuid_repository>/sync/ remote=
↪$BASE_ADDR/pulp/api/v3/remotes/deb/apt/<uuid_remote>/
```

Replace the `uuid` of the repository as returned by step one and replace the `uuid` of the remote as returned by step two.

This will return a `202 Accepted` response:

```
{
    "task": "/pulp/api/v3/tasks/c3a75b6c-97c9-410b-b079-a7a70ebae0cb/"
}
```

Depending on the size of the repository, this might take a while.

You can follow the progress of the task with a `GET` request to the task:

```
http get $BASE_ADDR/pulp/api/v3/tasks/c3a75b6c-97c9-410b-b079-a7a70ebae0cb/
```

This will return a `200 OK` response:

```
{
      "child_tasks": [],
      "created_resources": [
          "/pulp/api/v3/repositories/deb/apt/1381e6e4-135f-49dd-8e1d-5336c475fe92/
↪versions/1/"
      ],
      "error": null,
      "finished_at": "2020-06-26T06:58:54.913964Z",
      "name": "pulp_deb.app.tasks.synchronizing.synchronize",
      "parent_task": null,
      "progress_reports": [
          {
              "code": "downloading.artifacts",
              "done": 125,
              "message": "Downloading Artifacts",
              "state": "completed",
              "suffix": null,
              "total": null
          },
          {
              "code": "update.release_file",
              "done": 1,
              "message": "Update ReleaseFile units",
              "state": "completed",
              "suffix": null,
              "total": null
          },
          {
              "code": "update.packageindex",
              "done": 2,
              "message": "Update PackageIndex units",
              "state": "completed",
              "suffix": null,
              "total": null
          },
          {
              "code": "associating.content",
              "done": 229,
              "message": "Associating Content",
              "state": "completed",
              "suffix": null,
              "total": null
          }
      ],
```

(continues on next page)

(continued from previous page)

```
        "pulp_created": "2020-06-26T06:58:35.505839Z",
        "pulp_href": "/pulp/api/v3/tasks/b280c885-d9f9-4155-b7e1-e7fe87649f10/",
        "reserved_resources_record": [
            "/pulp/api/v3/remotes/deb/apt/73492e70-c9a7-4a34-92ae-ce16bb69e060/",
            "/pulp/api/v3/repositories/deb/apt/1381e6e4-135f-49dd-8e1d-5336c475fe92/"
        ],
        "started_at": "2020-06-26T06:58:35.665451Z",
        "state": "completed",
        "task_group": null,
        "worker": "/pulp/api/v3/workers/e64b7bf5-90a0-439a-a4f2-bc1a5c0f6942/"
}
```

Notice that when the synchronize task completes, it creates a new version, which is specified in `created_resources`.

Continue with *Publish and Host* to make your synced repository consumable.

1.3.2 Upload and Manage Content

To manually upload a package to Pulp, this assumes you to already have a *repository created*.

List all existing repositories by running the following command:

```
http get $BASE_ADDR/pulp/api/v3/repositories/deb/apt/
```

This will return a `200 OK` response:

```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "description": null,
            "latest_version_href": "/pulp/api/v3/repositories/deb/apt/ff14907e-d1c5-
↪4bac-b8d6-4c534575ed41/versions/0/",
            "name": "vim",
            "pulp_created": "2020-06-29T13:41:29.281602Z",
            "pulp_href": "/pulp/api/v3/repositories/deb/apt/ff14907e-d1c5-4bac-b8d6-
↪4c534575ed41/",
            "versions_href": "/pulp/api/v3/repositories/deb/apt/ff14907e-d1c5-4bac-
↪b8d6-4c534575ed41/versions/"
        }
    ]
}
```

You will need the value of `pulp_href` of the repository you want to add a package to.

Create Content by Uploading a File

You can directly upload a file to the `content` API endpoint:

```
http --form post $BASE_ADDR/pulp/api/v3/content/deb/packages/ file@"./vim_8.2.0716-3_
↪arm64.deb"
```

Note: It is strongly recommended to omit the `relative_path` and have Pulp generate a common pool location. This will be `pool/v/vim/vim_2:8.2.0716-3_arm64.deb` as shown below.

When specifying a `relative_path`, make sure to add the exact name of the package including its version as you'd get via `dpkg-deb -I vim_8.2.0716-3_arm64.deb`. It is composed of the *package name*, an *underscore*, and its *version*:

```
relative_path="any/arbitrary/location/vim_2:8.2.0716-3_arm64.deb"
```

This will return a `202 Accepted` response:

```
{
    "task": "/pulp/api/v3/tasks/5cc88067-f03c-4f7b-bda8-f193755a8aa5/"
}
```

Run the following command to view the status of the task:

```
http get $BASE_ADDR/pulp/api/v3/tasks/5cc88067-f03c-4f7b-bda8-f193755a8aa5/
```

This will return a `200 OK` response:

```
{
    "child_tasks": [],
    "created_resources": [
        "/pulp/api/v3/content/deb/packages/1eeabd4d-48b3-433e-9732-ce1b56cc9bb9/"
    ],
    "error": null,
    "finished_at": "2020-06-29T07:40:53.307389Z",
    "name": "pulpcore.app.tasks.base.general_create",
    "parent_task": null,
    "progress_reports": [],
    "pulp_created": "2020-06-29T07:40:53.113349Z",
    "pulp_href": "/pulp/api/v3/tasks/5cc88067-f03c-4f7b-bda8-f193755a8aa5/",
    "reserved_resources_record": [
        "/pulp/api/v3/artifacts/613e4817-6a3a-4f7f-8404-49ffe0085290/"
    ],
    "started_at": "2020-06-29T07:40:53.218540Z",
    "state": "completed",
    "task_group": null,
    "worker": "/pulp/api/v3/workers/50a13e76-fe27-4e3e-8cee-ae5ec41d272a/"
}
```

Note: Alternatively, you can upload an artifact to the `artifacts` API endpoint and then create a content unit of type `deb` from the exisiting artifact.

1. Upload a file to the `artifacts` endpoint

   ```
   http --form post $BASE_ADDR/pulp/api/v3/artifacts/ file@"./vim_8.2.0716-3_amd64.
   ↪deb"
   ```

2. Create Content from an exisiting artifact

   ```
   http post $BASE_ADDR/pulp/api/v3/content/deb/packages/ artifact=/pulp/api/v3/
   ↪artifacts/<uuid>/
   ```

Add Content to Repository

View the list of packages:

```
http get $BASE_ADDR/pulp/api/v3/content/deb/packages/
```

This will return the necessary `uuid` for the following step, which is identical to the `created_resources` from querying the task above.

Once there is a content unit, it can be added to and removed from repositories. This example adds the *arm* version of vim:

```
http post $BASE_ADDR/pulp/api/v3/repositories/deb/apt/250083a4-8eaa-42b6-a588-
↪c48c2a2935f0/modify/ add_content_units:="[\"http://localhost:24817/pulp/api/v3/
↪content/deb/packages/1eeabd4d-48b3-433e-9732-ce1b56cc9bb9/\"]"
```

This will return a `202 Accepted` response:

```
{
    "task": "/pulp/api/v3/tasks/ed0dfef8-7e5d-44a1-8f2b-7f7f29aae0dd/"
}
```

View the task by running the following command:

```
http get $BASE_ADDR/pulp/api/v3/tasks/ed0dfef8-7e5d-44a1-8f2b-7f7f29aae0dd/
```

This will return a `200 OK` response:

```
{
    "child_tasks": [],
    "created_resources": [
        "/pulp/api/v3/repositories/deb/apt/250083a4-8eaa-42b6-a588-c48c2a2935f0/
↪versions/1/"
    ],
    "error": null,
    "finished_at": "2020-06-29T07:47:50.816567Z",
    "name": "pulpcore.app.tasks.repository.add_and_remove",
    "parent_task": null,
    "progress_reports": [],
    "pulp_created": "2020-06-29T07:47:50.686844Z",
    "pulp_href": "/pulp/api/v3/tasks/ed0dfef8-7e5d-44a1-8f2b-7f7f29aae0dd/",
    "reserved_resources_record": [
        "/pulp/api/v3/repositories/deb/apt/250083a4-8eaa-42b6-a588-c48c2a2935f0/"
    ],
    "started_at": "2020-06-29T07:47:50.778375Z",
    "state": "completed",
    "task_group": null,
    "worker": "/pulp/api/v3/workers/50a13e76-fe27-4e3e-8cee-ae5ec41d272a/"
}
```

Go to *publish* to make your repository consumable.

1.3.3 Publish and Host

This section assumes that you have a repository with content in it. To do this, see the *Synchronize a Repository* or *Upload and Manage Content* documentation.

Create a Publication

Creating a publication is based on a repository:

```
http get $BASE_ADDR/pulp/api/v3/repositories/deb/apt/
```

This will return a `200 OK` response:


```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "description": null,
            "latest_version_href": "/pulp/api/v3/repositories/deb/apt/250083a4-8eaa-
↪42b6-a588-c48c2a2935f0/versions/1/",
            "name": "vim",
            "pulp_created": "2020-06-29T07:35:14.713025Z",
            "pulp_href": "/pulp/api/v3/repositories/deb/apt/250083a4-8eaa-42b6-a588-
↪c48c2a2935f0/",
            "versions_href": "/pulp/api/v3/repositories/deb/apt/250083a4-8eaa-42b6-
↪a588-c48c2a2935f0/versions/"
        }
    ]
}
```


Publications contain extra settings for how to publish. You can use the `apt` publisher on any repository of the apt type containing `deb` packages:

```
http post $BASE_ADDR/pulp/api/v3/publications/deb/apt/ repository=/pulp/api/v3/
↪repositories/deb/apt/250083a4-8eaa-42b6-a588-c48c2a2935f0/ simple=true
```

This will return a `202 Accepted` response:


```
{
    "task": "/pulp/api/v3/tasks/d49e056f-a637-454a-8797-67f81648b60f/"
}
```


Depending on the size of your repository, this might take a while. Check the status of the task by running the following command to see if the publication has been created:

```
http get $BASE_ADDR/pulp/api/v3/tasks/d49e056f-a637-454a-8797-67f81648b60f/
```

This will return a `200 OK` response:


```
{
    "child_tasks": [],
    "created_resources": [
        "/pulp/api/v3/publications/deb/apt/ecf87d05-978c-4327-8fe8-f50dc523b1a8/"
    ],
    "error": null,
    "finished_at": "2020-06-29T12:22:06.138655Z",
    "name": "pulp_deb.app.tasks.publishing.publish",
    "parent_task": null,
    "progress_reports": [],
    "pulp_created": "2020-06-29T12:22:05.892080Z",
```


(continues on next page)

(continued from previous page)


```
    "pulp_href": "/pulp/api/v3/tasks/d49e056f-a637-454a-8797-67f81648b60f/",
    "reserved_resources_record": [
        "/pulp/api/v3/repositories/deb/apt/250083a4-8eaa-42b6-a588-c48c2a2935f0/"
    ],
    "started_at": "2020-06-29T12:22:05.994098Z",
    "state": "completed",
    "task_group": null,
    "worker": "/pulp/api/v3/workers/6b8a7389-bafb-4d29-8e0b-184cd616ce10/"
}
```


`state` equaling `completed` indicates that your publication has been created successfully:

```
http get $BASE_ADDR/pulp/api/v3/tasks/d49e056f-a637-454a-8797-67f81648b60f/ | jq '.
↪state'
```

This returns the path of the created publication:

```
http get $BASE_ADDR/pulp/api/v3/tasks/d49e056f-a637-454a-8797-67f81648b60f/ | jq '.
↪created_resources[0]'
```

Create a Distribution

View a publication that you want to distribute and make consumable:

```
http get $BASE_ADDR/pulp/api/v3/publications/deb/apt/ecf87d05-978c-4327-8fe8-
↪f50dc523b1a8/
```

This will return a `200 OK` response:


```
{
    "pulp_created": "2020-06-29T12:22:06.006518Z",
    "pulp_href": "/pulp/api/v3/publications/deb/apt/ecf87d05-978c-4327-8fe8-
↪f50dc523b1a8/",
    "repository": "/pulp/api/v3/repositories/deb/apt/250083a4-8eaa-42b6-a588-
↪c48c2a2935f0/",
    "repository_version": "/pulp/api/v3/repositories/deb/apt/250083a4-8eaa-42b6-a588-
↪c48c2a2935f0/versions/1/",
    "signing_service": null,
    "simple": true,
    "structured": false
}
```


To host a publication which makes it consumable by a package manager, users create a distribution which will serve the associated publication at `/pulp/content/<distribution.base_path>`:

```
http post $BASE_ADDR/pulp/api/v3/distributions/deb/apt/ name="nginx" base_path="nginx
↪" publication=$BASE_ADDR/pulp/api/v3/publications/deb/apt/ecf87d05-978c-4327-8fe8-
↪f50dc523b1a8/
```

This will return a `202 Accepted` response:


```
{
    "task": "/pulp/api/v3/tasks/18159df8-b337-4ae8-b8cf-7ad0fba44bc7/"
}
```


Viewing the task will indicate if the distribution has been successful:

```
http get $BASE_ADDR/pulp/api/v3/tasks/18159df8-b337-4ae8-b8cf-7ad0fba44bc7/
```

This will return a `200 OK` response:


```
{
    "child_tasks": [],
    "created_resources": [
        "/pulp/api/v3/distributions/deb/apt/5cde2b30-7d35-4d64-a46b-0a4e5c984359/"
    ],
    "error": null,
    "finished_at": "2020-06-29T12:26:39.815218Z",
    "name": "pulpcore.app.tasks.base.general_create",
    "parent_task": null,
    "progress_reports": [],
    "pulp_created": "2020-06-29T12:26:39.575822Z",
    "pulp_href": "/pulp/api/v3/tasks/18159df8-b337-4ae8-b8cf-7ad0fba44bc7/",
    "reserved_resources_record": [
        "/api/v3/distributions/"
    ],
    "started_at": "2020-06-29T12:26:39.683538Z",
    "state": "completed",
    "task_group": null,
    "worker": "/pulp/api/v3/workers/50a13e76-fe27-4e3e-8cee-ae5ec41d272a/"
}
```


View the created resource (`created_resources`) to find the URL to the new repository hosted by Pulp:

```
http get $BASE_ADDR/pulp/api/v3/distributions/deb/apt/5cde2b30-7d35-4d64-a46b-
↪0a4e5c984359/
```

This will return a `200 OK` response:


```
{
    "base_path": "nginx",
    "base_url": "http://pulp3-source-debian10.hostname.example.com/pulp/content/nginx/
↪",
    "content_guard": null,
    "name": "nginx",
    "publication": "/pulp/api/v3/publications/deb/apt/ecf87d05-978c-4327-8fe8-
↪f50dc523b1a8/",
    "pulp_created": "2020-06-29T12:26:39.806283Z",
    "pulp_href": "/pulp/api/v3/distributions/deb/apt/5cde2b30-7d35-4d64-a46b-
↪0a4e5c984359/"
}
```


Running the following command will prove that Pulp exposes the repository as you'd expect:

```
http get http://pulp3-source-debian10.hostname.example.com/pulp/content/nginx/
```

This returns a `200 OK` response:

```
<!DOCTYPE html>
        <html>
            <body>
                <ul>
                    <li><a href="dists/">dists/</a></li>
                    <li><a href="pool/">pool/</a></li>
```

(continues on next page)

(continued from previous page)

```
                </ul>
            </body>
        </html>
```

You may use this url (`base_url`) to access Debian content from Pulp via a package manager like apt, i.e. in your `/etc/apt/sources.list` file.

1.4 REST API

Pulp is strongly REST API driven. The REST API documentation for both pulpcore and the `pulp_deb` plugin are automatically generated. They provide a full documentation for every option of every API endpoint.

The latest REST API documentation for pulpcore and `pulp_deb` may be found here:

- pulpcore REST API documentation
- pulp_deb REST API documentation

1.5 Client Bindings

1.5.1 Python Client

The pulp-deb-client package on PyPI provides bindings for all `pulp_deb` *REST API* endpoints. It is currently published daily and with every RC.

The pulpcore-client package on PyPI provides bindings for all pulpcore *REST API* endpoints. It is currently published daily and with every RC.

1.5.2 Ruby Client

The pulp_deb_client Ruby Gem on rubygems.org provides bindings for all `pulp_deb` *REST API* endpoints. It is currently published daily and with every RC.

The pulpcore_client Ruby Gem on rubygems.org provides bindings for all pulpcore *REST API* endpoints. It is currently published daily and with every RC.

1.5.3 Client in a Language of Your Choice

A client can be generated using Pulp's OpenAPI schema and any of the available OpenAPI generators.

Generating a client is a two step process:

1) Download the OpenAPI schema for pulpcore and all installed plugins:

```
curl -o api.json http://<pulp-hostname>:24817/pulp/api/v3/docs/api.json
```

   The OpenAPI schema for a specific plugin can be downloaded by specifying the plugin's module name as a GET parameter. For example for `pulp_deb` only endpoints use a query like this:

```
curl -o api.json http://<pulp-hostname>:24817/pulp/api/v3/docs/api.json?
↪plugin=pulp_deb
```

2) Generate a client using OpenAPI generator:

   The schema can then be used as input to the `openapi-generator-cli`. See try OpenAPI generator for documentation on getting started.

1.5.4 Generating a Client on dev Environment

The easiest way to set up a Pulp development environment is to use pulplift and/or the pulp installer. Development boxes from `pulplift` provide the `pbindings` alias for generating bindings (aka clients):

For example, use the following for generating Python bindings for `pulp_deb`:

```
pbindings pulp_deb python
```

As another example, use the following for generating Ruby bindings for version `2.5.0b1` of `pulp_deb`:

```
pbindings pulp_deb ruby 2.5.0b1
```

1.6 Plugin Maintenance

This part of the documentation is intended as an aid to current and future plugin maintainers.

1.6.1 Plugin Version Semantics

Release version strings of the `pulp_deb` plugin use the following format: `X.Y.Z<beta_tag?>`

A `X` version release, signifies one or more of the following:

- There are major new features.
- There has been a major overhaul of existing features.
- The plugin has entered a new stage of its development.
- The plugin is compatible with a new pulpcore `X` version.

Note: A `X` version release, is more of a high level "marketing" communication, than something with a detailed technical definition. It is up to the judgement of plugin maintainers when a new `X` version is warranted.

A `Y` version release, signifies the following:

- The `Y` version is the same, as the pulpcore-`Y` version that the release is for.
- A `Y` version release is given its own release branch.
- A `Y` version release may contain new features or any other type of change.
- A `Y` version release is generally performed when a new pulpcore `Y` version has been released.

A `Z` version release, signifies the following:

- A `Z` version release may contain only bugfixes (semantic versioning).
- `Z` stream changes are cherry-picked to the relevant `Y` version release branch.
- `Z` stream changes may be released as soon as they are ready, and as needs arise.

1.6.2 Using the Plugin Template

The pulp plugin template is used to collect changes relevant to all Pulp plugins. When there are new changes, the plugin template can then be used to automatically apply those changes to plugins that do not yet include them.

To use the plugin template, make sure you have cloned the Git repository to the same folder as the `pulp_deb` repository. Then you can issue the following commands within the root of the plugin template repository.

To generate an up to date `template_config.yml` file in the base of the `pulp_deb` repository, use:

```
./plugin-template --generate-config pulp_deb
```

You can adjust the configuration in the `template_config.yml` file to affect the other plugin template commands.

In order to apply the latest Travis pipeline changes use:

```
./plugin-template --travis pulp_deb
```

In order to apply a full plugin skeleton from the plugin template use:

```
./plugin-template --bootstrap pulp_deb
```

Note: Bootstrapping the plugin will revert many files in the `pulp_deb` plugin to a skeletal version. When using the `--bootstrap` option one must carefully select and commit only those changes one really wants.

1.6.3 Plugin Release Process

This section is based on the pulpcore release guide. It may need to be amended to reflect changes in the release process for pulpcore.

Note: The plugin is released to pypi.org as the pulp_deb python package. In addition a new pulp-deb-client package and pulp_deb_client Ruby Gem will be released. Client packages based on the latest plugin master branch are also released daily. See the *plugin template* and the `template_config.yml` file for more information on those independent releases.

Preparing the Release Environment

Important: The release process uses the release script at `.travis/release.py`. Before performing a major release, it may be worth checking if the *plugin template* has new changes for this script.

Creating a release uses the `.travis/release.py` python script. Running this script locally, requires the python dependencies from `.travis/release_requirements.txt`.

Since the script will be creating commits, you should run it somewhere with a configured Git identity (i.e. not from within a `pulplift` development box). As a result, a local python virtual environment is recommended. This can be created as follows:

```
mkvirtualenv -r .travis/release_requirements.txt pulp_release
```

You can then reenter this venv later using `workon pulp_release`.

Release Steps

Note: As one might expect, various release steps require merge/push rights on the `pulp_deb` repository. However, a lot of the preparation can be performed by opening the relevant PRs.

For a `Y` release, perform the following steps (the example assumes we are releasing version `2.6.0`):

1. Ensure the `2.6.0` pulp_deb milestone exists and contains the correct issues.

   Note: You can double check this one more time after step 2. The `Releasing 2.6.0` commit from step 2 provides a redmine query for all issues that should be in the milestone.

2. Run the following commands to generate the `release_2.6.0` branch (with commits):

   ```
   workon pulp_release
   python .travis/release.py minor --lower 3.6 --upper 3.7
   ```

   The `--lower` and `--upper` parameters give the pulpcore version range that the release should be compatible with. Currently, each release is pegged to exactly one pulpcore-`Y` release. This is due to change with the pulpcore `3.7` release.

3. Create a PR for the `release_2.6.0` branch generated in step 2.
4. Review and merge the PR to `master`.
5. Create and push the `2.6` release branch (with the `Releasing 2.6.0` commit checked out).
6. On the `2.6` release branch, manually bump the version from `2.6.0` to `2.6.1.dev` in `.bumpversion.cfg`, `pulp_deb/__init__.py`, and `setup.py`. The commit message should be `Bump to 2.6.1.dev`. This is for the benefit of any future `Z` releases on this branch. Don't forget to push.
7. Trigger the release by creating and pushing the `2.6.0` release tag at the `Releasing 2.6.0` commit.
8. Check the pulp_deb travis build page, the pulp_deb python package, the pulp-deb-client package, and the pulp_deb_client Ruby Gem to see if everything has released correctly.
9. Finally, send a release announcement to the `pulp-list` mailing list. See *release announcements* for more information.

For a `Z` release, perform the following steps (the example assumes we are releasing version `2.6.1`):

1. Ensure the `2.6.1` pulp_deb milestone exists and contains the correct issues.

   Note: You can double check this one more time after step 2. The `Releasing 2.6.1` commit from step 2 provides a redmine query for all issues that should be in the milestone.

2. Run the following commands to generate the `release_2.6.1` branch (with commits):

   ```
   workon pulp_release
   python .travis/release.py patch --lower 3.6 --upper 3.7
   ```

   The `--lower` and `--upper` parameters give the pulpcore version range that the release should be compatible with. For the release script to do the right thing, they need to be provided, even if they should not change for the `Z` release.

3. Create a PR for the `release_2.6.1` branch generated in step 2. It needs to go into the `2.6` branch, not `master`!

4. Review and merge the PR.
5. Switch back to `master`, and `git cherry-pick -x` the `Building changelog for 2.6.1` commit from the `2.6` release branch. Push directly to master or create and merge a PR for it.
6. Trigger the release by creating and pushing the `2.6.1` release tag at the `Releasing 2.6.1` commit (on the `2.6` release branch).
7. Check the pulp_deb travis build page, the pulp_deb python package, the pulp-deb-client package, and the pulp_deb_client Ruby Gem to see if everything has released correctly.
8. Finally, send a release announcement to the `pulp-list` mailing list. See *release announcements* for more information.

Release Announcements

`pulp_deb` releases are announced on the `pulp-list` mailing list.

Example announcement email:

```
To: pulp-list@redhat.com
Subject: pulp_deb 2.6.1 released

pulp_deb version 2.6.1 [0] has been released.

Have a look at the release notes [1] for changes.
This version of the plugin is compatible with pulpcore version 3.6 [2].
You can check the known issues [3] (and open new ones).

[0] https://pypi.org/project/pulp-deb/2.6.0/
[1] https://pulp-deb.readthedocs.io/en/latest/changes.html#b1-2020-09-01
[2] https://www.redhat.com/archives/pulp-list/2020-August/msg00008.html
[3] https://pulp.plan.io/projects/pulp_deb/issues

kind regards,
Quirin Pamp (quba42)
```

Feel free to add additional highlights from the release.

1.7 Contributing

To contribute to the `pulp_deb` plugin follow this process:

1. Clone the GitHub repo
2. Make a change
3. Make sure all tests passed
4. Add a file into CHANGES folder (Changelog update).
5. Commit changes to own `pulp_deb` clone
6. Make pull request from github page for your clone against master branch

1.7.1 Changelog Update

The `CHANGES.rst` file is managed using the towncrier tool and all non trivial changes must be accompanied by a news entry.

To add an entry to the news file, you first need an issue on the pulp_deb issue tracker describing the change you want to make. Once you have an issue, take its number and create a file inside of the `CHANGES/` directory named as the issue number with an extension of `.feature`, `.bugfix`, `.doc`, `.removal`, or `.misc`. So if your issue is 3543 and it fixes a bug, you would create the file `CHANGES/3543.bugfix`. The content of your new file should be a short sentence describing your change in a single line of reStructuredText formatted text. You do not need to reference any issue numbers since they are already referenced via the filename. The sentence should be in past tense. An example might be:

```
Fixed synchronization of Release files without a Suite field.
```

PRs can span multiple categories by creating multiple files (for instance, if you added a feature and deprecated an old feature at the same time, you would create `CHANGES/NNNN.feature` and `CHANGES/NNNN.removal`). Likewise if a PR touches multiple issues you may create a file for each of them with the exact same contents and Towncrier will deduplicate them.

1.8 Changelog

1.8.1 2.6.1 (2020-09-03)

Misc

- Dropped the beta status of the plugin. The plugin is now GA! #6999

1.8.2 2.6.0b1 (2020-09-01)

Features

- Added handling of packages with the same name, version, and architecture, when saving a new repository version. #6429
- Both simple and structured publish now use separate `Architecture: all` package indecies only. #6991

Bugfixes

- Optional version strings are now stripped from the sourcename before using it for package file paths. #7153
- Fixed several field names in the to deb822 translation dict. #7190
- `Section` and `Priority` are no longer required for package indecies. #7236
- Fixed content creation for fields containing more than 255 characters by using `TextField` instead of `CharField` for all package model fields. #7257
- Fixed a bug where component path prefixes were added to package index paths twice instead of once when using structured publish. #7295

Improved Documentation

- Added a note on per repository package uniqueness constraints to the feature overview documentation. #6429
- Fixed several URLs pointing at various API documentation. #6506
- Reworked the workflow documentation and added flow charts. #7148
- Completely refactored the documentation source files. #7211
- Added a high level "feature overview" documentation. #7318
- Added meaningful endpoint descriptions to the REST API documentation. #7355

Misc

- Added tests for repos with distribution paths that are not equal to the codename. #6051
- Added a long_description to the python package. #6882
- Added test to publish repository with package index files but no packages. #7344

1.8.3 2.5.0b1 (2020-07-15)

Features

- Added additional metadata fields to published Release files. #6907

Bugfixes

- Fixed a bug where some nullable fields for remotes could not be set to null via the API. #6908
- Fixed a bug where APT client was installing same patches again and again. #6982

Misc

- Renamed some internal models to Apt.. to keep API consistent with other plugins. #6897

1.8.4 2.4.0b1 (2020-06-17)

Features

- The "Date" field is now added to Release files during publish. #6869

Bugfixes

- Fixed structured publishing of architecture 'all' type packages. #6787
- Fixed a bug where published Release files were using paths relative to the repo root, instead of relative to the release file. #6876

1.8.5 2.3.0b1 (2020-04-29)

Features

- Added Release file signing using signing services. #6171

Bugfixes

- Fixed synchronization of Release files without a Suite field. #6050
- Fixed publication creation with packages referenced from multiple package inecies. #6383

Improved Documentation

- Documented bindings installation for the dev environment. #6396

Misc

- Added tests for invalid Debian repositories (bad signature, missing package indecies). #6052
- Made tests use the bindings config from pulp-smash. #6393

1.8.6 2.2.0b1 (2020-03-03)

Features

- Structured publishing (with releases and components) has been implemented. #6029
- Verification of upstream signed metadata has been implemented. #6170

1.8.7 2.0.0b3 (2019-11-14)

Features

- Change *relative_path* from *CharField* to *TextField* #4544
- Add more validation for uploading packages and installer packages. #5377

Deprecations and Removals

- Change *_id*, *_created*, *_last_updated*, *_href* to *pulp_id*, *pulp_created*, *pulp_last_updated*, *pulp_href* #5457
- Remove "_" from *_versions_href*, *_latest_version_href* #5548
- Removing base field: *_type* . #5550
- Sync is no longer available at the {remote_href}/sync/ repository={repo_href} endpoint. Instead, use POST {repo_href}/sync/ remote={remote_href}.

  Creating / listing / editing / deleting deb repositories is now performed on /pulp/api/v3/repositories/deb/apt/ instead of /pulp/api/v3/repositories/. #5698

Bugfixes

- Fix *fields* filter. #5543

Misc

- Depend on pulpcore, directly, instead of pulpcore-plugin. #5580

1.8.8 2.0.0b2 (2019-10-02)

Features

- Rework Package and InstallerPackage serializers to allow creation from artifact or uploaded file with specifying any metadata. #5379
- Change generic content serializer to create content units by either specifying an artifact or uploading a file. #5403, #5487

Deprecations and Removals

- Remove one shot uploader in favor of the combined create endpoint. #5403

Bugfixes

- Change content serializers to use relative_path instead of _relative_path. #5376

Improved Documentation

- Change the prefix of Pulp services from pulp-* to pulpcore-* #4554
- Reflect artifact and upload functionality in the content create endpoint documentation. #5371

Misc

- PublishedMetadata is now a type of Content. #5304
- Replace *ProgressBar* with *ProgressReport*. #5477

1.8.9 2.0.0b1 (2019-09-06)

Features

- Add oneshot upload functionality for deb type packages. #5391

Bugfixes

- Add relative_path to package units natural key to fix uniqueness constraints. #5377
- Fix publishing of lazy content and add download_policy tests. #5405

Improved Documentation

- Reference the fact you must have both _relative_path and relative_path. #5376
- Fix various documentation issues from API changes, plus other misc fixes. #5380

Misc

- Adopting related names on models. #4681
- Generate and commit initial migrations. #5401