Pulp Documentation

Release 2.6.4

Pulp Team

May 02, 2016

Contents

CHAPTER 1

User Guide

Contents:

1.1 Introduction

Pulp is a platform for managing repositories of content, such as software packages, and pushing that content out to large numbers of consumers. If you want to locally mirror all or part of a repository, host your own content in a new repository, manage content from multiple sources in one place, and push content you choose out to large numbers of clients in one simple operation, Pulp is for you!

Pulp has a well-documented REST API and command line interface for management.

Pulp is completely free and open-source, and we invite you to join us on GitHub!

1.1.1 What Pulp Can Do

- Pull in content from existing repositories to the Pulp server. Do it manually or on a recurring schedule.
- Upload new content to the Pulp server.
- Mix and match uploaded and imported content to create new repositories, then publish and host them with Pulp.
- Publish your content as a web-based repository, to a series of ISOs, or in any other way that meets your needs.
- Push content out to consumer machines, and track what each consumer has installed.

1.1.2 Plugins

Pulp manages content, and its original focus was software packages such as RPMs and Puppet modules. The core features of Pulp, such as syncing and publishing repositories, have been implemented in a generic way that can be extended by plugins to support specific content types. We refer to the core implementation as the **Pulp Platform**.

With this flexible design, Pulp can be extended to manage nearly any type of digital content.

More importantly, Pulp makes it easy for third-party plugins to be written and deployed separately from the Pulp Platform. When new plugins are installed, Pulp detects and activates them automatically.

1.1.3 Goal of this Guide

Pulp can manage many types of content, but it is not tied to any one of them. As such, this guide will help you install and configure Pulp, and learn how to use Pulp's core features in a non-type-specific way. Once you are familiar with what Pulp can offer, visit the user guide that is specific to the content type in which you are interested. You can find all of our documentation at our docs page.

Many examples require the use of a type, and for those we will use "rpm". However, examples in this guide will only cover features that are common across content types.

1.2 Release Notes

Contents:

1.2.1 Pulp 2.6 Release Notes

Pulp 2.6.4

Pulp 2.6.4 is an important security update. It fixes one issue.

A security flaw (CVE-2015-5263) was discovered in Pulp's consumer management system. When the pulp-consumer CLI is used to register to the Pulp server, it downloads a public key from the Pulp server and stores it locally. Later when the Pulp server sends messages to the client via a message broker to instruct it to perform commands, it will use the corresponding private key to sign the messages. The client checks the signatures before executing the instructions to ensure that the messages came from the Pulp server and not from an attacker.

Versions of pulp-consumer-client between 2.4.0 and 2.6.3 do not check the server's TLS certificate signatures when retrieving the server's public key upon registration:

https://github.com/pulp/pulp/blob/aa432bf58497b5e3682333b1d5f5ae4f45788a61/client_consumer/pulp/client/consumer/cli.py#L103

This allows a man in the middle to inject their own message signing key and to then perform administrative actions on the machine, if they are able to send messages through the message broker.

Austin Macdonald fixed this issue in this commit by using our pulp.bindings library as the rest of our CLI does:

https://github.com/pulp/pulp/commit/b542d7465f7e6e02e1ea1aec059ac607a65cefe7#diff-17110211f89c042a9267e2167dedd754

Users who do not use pulp-consumer are not affected by this issue.

Thanks to Austin Macdonald for writing the fix, to Dennis Kliban for making our beta and release build, and to Preethi Thomas for testing our releases!

Pulp 2.6.3

Pulp 2.6.3 is released with packages for Fedora 22 and Fedora 21. Support for Fedora 20 has been dropped. Please see the Fedora lifecycle for more detail.

Bug Fixes

This is a minor release which contains bug fixes for these issues.

Pulp 2.6.2

Bug Fixes

This is a minor release which contains bug fixes for these issues.

Pulp 2.6.1

Bug Fixes

This is a minor release which contains bug fixes for these issues.

Improvements

- Pulp has been fully tested in a clustered configuration. A new section of documentation titled *Clustering Pulp* is available with more detail on configuring this type of Pulp deployment.
- One area of improvement relates to upgrades. Starting with 2.6.1, Pulp processes *pulp_workers*, *pulp_celerybeat*, and *pulp_resource_manager* are stopped on upgrade or removal of the *pulp-server* package. After upgrading, you must restart all Pulp related services.

Pulp 2.6.0

New Features

- Pulp now supports RabbitMQ as its task message broker. See the inline comments in `/etc/pulp/server.conf` for instruction on configuring Pulp to use RabbitMQ.
- Pulp now allows user credentials to be read from user's `~/.pulp/admin.conf`. This should allow pulp-admin to be automated more easily and more securely. Please see our Authentication documentation for details.
- Pulp no longer requires additional configuration of Qpid after installation. It now works with the ANONYMOUS authentication mechanism. Users can still use a username/password however if they set up a SASL database as described in the installation document.
- Additional status information is available via the status API. More information is available in the *status API document*.

Deprecation

- The `cancel_publish_repo` method provided by the `Distributor` base plugin class is deprecated and will be removed in a future release. Read more about the *plugin cancellation changes*.
- The `cancel_publish_group` method provided by the `GroupDistributor` base plugin class is deprecated and will be removed in a future release. Read more about the *plugin cancellation changes*.
- The `cancel_sync_repo` method provided by the `Importer` base plugin class is deprecated and will be removed in a future release. Read more about the *plugin cancellation changes*.
- The `api_version` field that is returned by the `/status` API is deprecated and will be removed in a future release.
- The python-gofer-amqplib package was discontinued in gofer 2.4. Installations must replace python-gofer-amqplib with python-gofer-amqp if installed.

Upgrade Instructions for 2.5.x –> 2.6.0

Prior to upgrading, all tasks must be stopped. One way to accomplish this is to stop all *pulp_workers*, *pulp_celerybeat*, and *pulp_resource_manager* processes and then list the current tasks using:

```
pulp-admin tasks list
```

Any task that is in the “Running” or “Waiting” state should be canceled by its <uuid> using:

```
pulp-admin tasks cancel --task-id <uuid>
```

After all tasks have been canceled upgrade the packages using:

```
sudo yum update
```

After yum completes you should migrate the database using:

```
sudo -u apache pulp-manage-db
```

After the database migrations finish, restart *httpd*, *pulp_workers*, *pulp_celerybeat*, and *pulp_resource_manager*.

Bugs

This release has fixes for these issues.

Known Issues

- An issue in the pulp (gofer) agent plugin *can* cause in-progress RMI requests to be discarded when *goferd* is restarted. Should this occur, an entry is written to the system log on the consumer. On the Pulp server, the associated task will appear to never complete. This has been fixed in Pulp 2.6.1.
- Version 2.5 of the python-gofer-amqp messaging adapter, which is used to support RabbitMQ, contains a regression. It pertains to the reconnect logic. Depending on how a connection error manifests itself, it *can* result in a traceback during reconnect. Should this occur, The logged traceback would contain: *RuntimeError: maximum recursion depth exceeded*. This issue has already been fixed in Gofer upstream and will be included with Pulp 2.6.1.

Client Changes

Agent Changes

Rest API Changes

- A new *Task Report* attribute named *worker_name* is introduced that holds the name of the worker a task is associated with. Previously the worker name was stored in a *Task Report* attribute named *queue*. The *queue* attribute now correctly records the queue a task is put in. The *queue* attribute is deprecated and will be removed from the *Task Report* in a future Pulp version.
- The URL for the content catalog entries `/v2/content/catalog/<source-id>` is missing the trailing ‘/’ and has been deprecated. Support for the URL `/v2/content/catalog/<source-id>/` has been added.
- A new API call is added to search profile attributes for all consumer profiles using the Search API. `/pulp/api/v2/consumers/profile/search/`. With this API call all the unit profiles can be retrieved at one time instead of querying each consumer through `/v2/consumers/<consumer_id>/profiles/`. It is also possible to query for a single package across all consumers.

Binding API Changes

Plugin API Changes

Plugin Cancellation Changes

Cancel now exits immediately by default. The `cancel_publish_repo`, `cancel_publish_group`, and `cancel_sync_repo` methods provided by the `Distributor`, `GroupDistributor`, and `Importer` base plugin classes now provide a behavior that exits immediately by default. Previously these methods raised a NotImplementedError() which required plugin authors to provide an implementation for these methods. These methods will be removed in a future version of Pulp, and all plugins will be required to adopt the exit-immediately behavior.

A cancel can occur at any time, which mean that in a future version of Pulp any part of plugin code can have its execution interrupted at any time. For this reason, the following recommendations should be adopted by plugin authors going forward in preparation for this future change:

- Group together multiple database calls that need to occur together for database consistency.
- Do not use subprocess. If your plugin code process gets cancelled it could leave orphaned processes.
- Assume that plugin code which is supposed to run later may not run.
- Assume that the previous executions of plugin code may not have run to completion.

Thank You

Thank you to all of Pulp's contributors, especially these new ones!

- Adam D.
- Andrea Giardini
- Andreas Schieb
- Ina Panova
- Michael Moll
- Patrick Creech
- Vijaykumar Jain

1.2.2 Pulp 2.5 Release Notes

Pulp 2.5.3

This release fixes #1185367

To upgrade, shut down all Pulp services:

```
$ for s in {goferd,pulp_celerybeat,pulp_resource_manager,pulp_workers,httpd}; do sudo systemctl stop
```

Next, update the packages:

```
$ sudo yum update
```

Run the pulp-manage-db script as the apache user:

```
$ sudo -u apache pulp-manage-db
```

Once pulp-manage-db is finished, start all Pulp services.

Pulp 2.5.2

This release fixes #1179463, #1178920, #1171278 and #1159040.

To upgrade, shut down all Pulp services:

```
$ for s in {goferd,pulp_celerybeat,pulp_resource_manager,pulp_workers,httpd}; do sudo systemctl stop
```

Next, update the packages:

```
$ sudo yum update
```

Run the pulp-manage-db script as the apache user:

```
$ sudo -u apache pulp-manage-db
```

Restart the pulp services and apache

Pulp 2.5.1

This is a an important bugfix release that contains the fix for #1165355 and #1171509. Additional bugs that were fixed in Pulp 2.5.1.

Upgrade Instructions for 2.5.0 –> 2.5.1

Perform the upgrade instructions for *upgrading from 2.4.x*

Additionally for systems that are being upgraded from 2.5.0 to 2.5.1 and are using pulp_rpm to manage yum repositories you will need to manually remove the published repositories from disk and republish due to 1171509 This is only necessary if you have been running Pulp 2.5.0. If you are upgrading from 2.4.x these steps are not required:

```
$ sudo rm -rf /var/lib/pulp/published/yum/*
```

For each of your rpm repositories:

```
$ pulp-admin rpm repo publish run --repo-id <repo-id>
```

Pulp 2.5.0

New Features

- pulp-admin now has a bash tab completion script.
- A new selinux policy is introduced which confines the *pulp_workers*, *pulp_celerybeat*, and *pulp_resource_manager* processes.
- Pulp 2.5.0 works with *pulp_docker*, an optional plugin to manage Docker repositories. In Pulp 2.5.0 this optional plugin is considered “tech preview” and did not undergo the same level of testing as other plugins. Please refer to the pulp_docker documentation for usage information.

- Pulp now supports SSL on its connection to MongoDB. It is strongly recommended that you configure MongoDB to perform SSL, and configure Pulp to require a validly signed certificate from Mongo. If you wish to do this, edit `/etc/pulp/server.conf` and configure `ssl` and `verify_ssl` to `true` in the `[database]` section.
- When the *pulp_workers* service is stopped, it will now cancel tasks that the workers are processing instead of waiting for those tasks to finish.

Deprecation

- The *task_type* attribute of a *Task Report* is deprecated with Pulp 2.5.0. This attribute will be removed in a future release.
- Many API calls return an attribute named *_ns*. This attribute will be removed in a future release and should not be used.

Bugs

You can see the complete list of bugs that were fixed in Pulp 2.5.0.

Upgrade Instructions for 2.4.x –> 2.5.x

To upgrade, shut down all Pulp services:

```
$ for s in {goferd,pulp_celerybeat,pulp_resource_manager,pulp_workers,httpd}; do sudo systemctl stop
```

Next, update the packages:

```
$ sudo yum update
```

Run the pulp-manage-db script as the apache user:

```
$ sudo -u apache pulp-manage-db
```

Restart the pulp services and apache

Next, run `pulp-manage-db`:

```
$ sudo -u apache pulp-manage-db
```

And lastly, restart the Pulp services:

```
$ for s in {goferd,pulp_celerybeat,pulp_resource_manager,pulp_workers,httpd}; do sudo systemctl start
```

Note: If you are using Upstart instead of systemd, you should use service $s {stop,start} in the lines above.

Thank You

Thank you to all of Pulp's contributors, especially these new ones!

- Irina Gulina
- Peter Gustafsson

- Petter Hassberg
- Dennis Kliban
- Christoffer Kylvåg
- Austin Macdonald

1.2.3 Pulp 2.4 Release Notes

Pulp 2.4.4

This is a minor bugfix release that contains a small patch to support the fix for #1165355.

To upgrade, shut down all Pulp services:

```
$ for s in {goferd,pulp_celerybeat,pulp_resource_manager,pulp_workers,httpd}; do sudo systemctl stop
```

Next, update the packages:

```
$ sudo yum update
```

Next, run `pulp-manage-db`:

```
$ sudo -u apache pulp-manage-db
```

And lastly, restart the Pulp services:

```
$ for s in {goferd,pulp_celerybeat,pulp_resource_manager,pulp_workers,httpd}; do sudo systemctl start
```

Note: If you are using Upstart instead of systemd, you should use service $s {stop,start} in the lines above.

Pulp 2.4.3

This release is a response to CVE-2014-3566, commonly known as the POODLE attack.

Bugs

You can see the complete list of bugs that were fixed in Pulp 2.4.3.

Upgrade Instructions for 2.4.2 –> 2.4.3

Upgrading from Pulp 2.4.2 to 2.4.3 is straightforward. All that is required is a yum update, and a restart:

```
$ sudo yum update
```

It is recommended that you configure your web server to refuse SSLv3.0. In Apache, you can do this by editing `/etc/httpd/conf.d/ssl.conf` and configuring the `SSLProtocol` directive like this:

```
SSLProtocol all -SSLv2 -SSLv3
```

Next, restart all of the Pulp services:

```
$ for s in {goferd,pulp_celerybeat,pulp_resource_manager,pulp_workers,httpd}; do sudo systemctl resta
```

Note: If you are using Upstart, substitute the above command with `s/systemctl restart $s/service $s restart/`.

Pulp 2.4.2

New Features

Pulp now supports running RHEL 5 consumers with SELinux in enforcing mode. It is recommended that all EL 5 users re-enable SELinux enforcing mode with this release of Pulp.

Rest API Changes

- Certain API calls under `/consumers/` related to repo binding would erroneously return full task information. This has been corrected; these calls now only return the task's ID.

Upgrade Instructions for 2.4.1 –> 2.4.2

For all systems that have Pulp software installed, perform a system update:

```
$ sudo yum update
```

For all EL 5 systems, please consider re-enabling SELinux's enforcing mode as Pulp is now capable of running this way.

Pulp 2.4.1

Bugs

You can see the complete list of bugs that were fixed in Pulp 2.4.1.

Backwards Incompatible Changes

This version of Pulp no longer performs CRL checks. Pulp used a custom version of M2Crypto to do this and it was decided that it was too risky to carry our own custom cryptography library in our repositories. As a result, users who were using that feature must configure their web server to perform CRL checks against client certificates. You can read more about *CRL Support* in our server configuration documentation.

Rest API Changes

- The timestamps returned for the following objects and fields have been converted from an ISO 8601 timestamp with a timezone offset to native ISO 8601 timestamps in UTC:
  1. Repository Distributor (last_published)
  2. Repository Group Distributor (last_published)

3. Repository Publish Results (started, completed)
4. Repository Group Publish Results (started, completed)
5. Repository Importer (last_sync)
6. Repository Importer Sync Results (started, completed)

Deprecation

- The 2.4.1 release deprecates the `operation_retries` setting from `/etc/pulp/server.conf`. All Pulp services now attempt to re-connect to the MongoDB indefinitely.

Upgrade Instructions for 2.4.0 –> 2.4.1

Upgrading from Pulp 2.4.0 to 2.4.1 is straightforward. Begin with a yum update:

```
$ sudo yum update
```

After you've performed the package updates, please remove the `operation_retries` setting from `/etc/pulp/server.conf`, as it's been deprecated. Once you have your configuration files in place, restart all Pulp services (`httpd`, `pulp_celerybeat`, `pulp_resource_manager`, `pulp_workers`, and `goferd`).

Pulp has stopped maintaining its own m2crypto package, and now relies on the package provided by the operating system. If you are using the Pulp provided package, you should remove it and use the one provided by your operating system instead. The Pulp provided m2crypto package had "pulp" in the package release, so you can use this command to detect if you have it or not:

```
$ rpm -q m2crypto | grep pulp
m2crypto-0.21.1.pulp-8.el6.x86_64
```

You can remove it by using yum. If a newer version is available for your operating system, you can use `yum update` to get it. If not, you will need to perform a downgrade to get to your operating system's supported m2crypto:

```
$ sudo yum downgrade m2crypto
```

Pulp 2.4.0

New Features

- An all-new distributed task system based on Celery.
- All of `/var/lib/pulp` can now be a shared filesystem.
- Username/password authentication for MongoDB. Requirement for python-pymongo was updated to version 2.5.2.
- Publishing Puppet repositories to flattened directories.
- All messaging between the Pulp server and agents is signed and authenticated using asymmetric keys. Public keys are exchanged during registration. Upgraded installations with existing consumers must run: `pulp-consumer update --keys` and restart the goferd service for messaging between the server and agent to continue working properly.
- Pulp now uses syslog for all log messages, rather than writing its own log files as in previous releases. Please see our *Logging* documentation for details, as Pulp does not write to `/var/log/pulp/` as it used to.

- Pulp also has eliminated the `/etc/pulp/logging/` folder, as well as the entire `[logs]` section of `/etc/pulp/server.conf`. All logging configuration has been replaced with a single `log_level` setting in the `[server]` section of `/etc/pulp/server.conf`, and it is also optional.
- Pulp's server.conf can now be completely empty, and Pulp will choose sane defaults for each setting. The server.conf that comes with a new Pulp server installation has all of the settings commented and set to the default values. Due to this, the OAuth key and secret fields are no longer automatically populated and users will need to provide these values when they wish to configure Pulp installations to use OAuth. OAuth is now disabled by default.
- Pulp has tightened the security in version 2.4.0 by adding functionality to validate the server SSL certificate against trusted CA certificates. This introduces two new settings (verify_ssl and ca_path) to three different files (/etc/pulp/admin/admin.conf, /etc/pulp/consumer/consumer.conf and /etc/pulp/nodes.conf). These settings are required as of now, so Pulp will not function properly until you add these settings to those files. It is strongly recommended that verify_ssl be set to 'True' for all production installations of Pulp. This is the default setting, and unless you have deployed Pulp with SSL certificates that have been signed by a trusted certificate authority, you will find that Pulp's clients give you error messages with this upgrade. You will need to install signed SSL certificates for the web server to use in order to operate Pulp's clients in a secure fashion. If you are unconcerned with security and wish to run Pulp for evaluation purposes (and won't be using real passwords with Pulp), you can set verify_ssl to false in these settings files and Pulp will revert to its former behavior.

Deprecation

Pulp uses the Python oauth2 library to perform OAuth. Unfortunately, that library seems to be unmaintained upstream and has at least one known security flaw, and so the Pulp team was faced with finding a replacement, writing a replacement, or removing OAuth as an authentication mechanism. The Pulp team does not believe there is a strong use case for OAuth in Pulp, and so Pulp 2.4.0 deprecates OAuth.

Client Changes

- The orphan remove command was converted to poll until the remove finishes. A background flag was added to match the pattern of other polling commands.
- The behavior of commands requiring agent participation have changed. The *Waiting to begin...* text displayed by the spinner now indicates that a task has been created and that a request has been sent to the agent, but that the agent has not yet accepted the request. Once the agent has accepted the request, the text displayed by the spinner will change to indicate this. The spinner will continue until the agent begins executing the request. Agent related tasks no longer have a timeout, so it's up to the caller to determine how long to wait for completion. It is the responsibility of the caller to cancel tasks not progressing as desired.

Agent Changes

- The pulp-agent service link is no longer installed. In previous versions, the pulp-agent service was just a symlink to goferd. Users should interact with the goferd service directly.
- goferd 1.3.0+ supports control by systemd.

Bugs

You can see the complete list of bugs that were fixed in Pulp 2.4.0.

Known Issues

- There was one regression discovered during the 2.4.0 QE cycle that has not been resolved as of the release. The 2.4.0 distributor publishes groups in a slightly different way than Anaconda expects during interactive kickstarting. This causes no groups to be chosen by default during the package group selection installation step. The Pulp team decided to release 2.4.0 anyway, as the workaround is for users to simply make sure to select at least one package group during the installation. Automated kickstarts are not affected by this issue.
- There is a configuration bug related to using MongoDB with authenticated database users. The error presents itself during syncs and other task-related operations. A workaround is documented in comment #1 of the bug.
- `/etc/pulp/admin/admin.conf` is owned by a different RPM than it was in 2.3.x. This means that when you upgrade Pulp, you will not get an admin.conf.rpmnew file. Instead, admin.conf will be overwritten with the new stock version.

Upgrade Instructions for 2.3.x –> 2.4.0

> **Warning:** Due to `/etc/pulp/admin/admin.conf` being owned by a different package in 2.4.0 than it was in 2.3.x releases, you will need to make a backup of admin.conf before performing the upgrade if you wish to keep any of your settings. No admin.conf.rpmnew file will be generated during the upgrade!

Begin by ensuring that you are using MongoDB version 2.4.0 or greater.

> **Warning:** Pulp 2.4.0 requires MongoDB version 2.4.0 or greater. You must upgrade your MongoDB installation before performing any further steps.

Upgrading from 2.3.x –> 2.4.0 requires all components to be upgraded together. Pulp 2.3.x servers and nodes are not compatible with Pulp 2.4.0 and vice versa. All consumers must be upgraded first, but will not be usable until they are re-registered with their new Pulp 2.4.0 server or node.

The 2.3.x –> 2.4.0 server or node upgrade process requires all associated consumers to either be upgraded or off. The upgrade process will not continue if there are active 2.3.x consumers still connected to the message bus. After the server and node installations are upgraded, the upgraded consumers need to be re-registered.

For Qpid environments, to upgrade a consumer from 2.3.x –> 2.4.0, run the command `sudo yum groupupdate pulp-consumer-qpid`.

> **Note:** For RabbitMQ installations, upgrade the Pulp consumer client and agent packages without any Qpid specific dependencies using `sudo yum groupinstall pulp-consumer`. You will need to upgrade or install additional RabbitMQ dependencies manually including the `python-gofer-amqplib` package.

The upgrade will create a file called `consumer.conf.rpmnew`, which contains the default `consumer.conf` for Pulp 2.4.0 consumers. The new `consumer.conf.rpmnew` file needs to be merged into your existing `consumer.conf` by hand as new, required configuration properties are introduced with 2.4.0, but portions of the old config will likely still be useful. For example, the newly required validate_ssl and ca_path settings must be included.

Once the `consumer.conf` file is setup to use the new configuration, restart the consumer. On Upstart systems the restart is done using:

```
$ sudo service goferd restart
```

For systemd systems:

```
$ sudo systemctl restart goferd
```

A message broker is required for Pulp 2.4.0. Pulp 2.3.x required Qpid specifically as the message broker, but Pulp 2.4 will work with either Qpid or RabbitMQ. If using Qpid, ensure that you are using Qpid 0.18 or later, and that the `qpid-cpp-server-store` package is also installed. It is recommended to upgrade the Qpid broker to the latest version available on your platform. You can do this by running the following commands on the broker machine:

```
$ sudo yum update qpid-cpp-server
$ sudo yum install qpid-cpp-server-store
```

Note: In environments that use Qpid, the `qpid-cpp-server-store` package provides durability, a feature that saves broker state if the broker is restarted. This is a required feature for the correct operation of Pulp. Qpid provides a higher performance durability package named `qpid-cpp-server-linearstore` which can be used instead of `qpid-cpp-server-store`, but may not be available on all versions of Qpid. If `qpid-cpp-server-linearstore` is available in your environment, consider uninstalling `qpid-cpp-server-store` and installing `qpid-cpp-server-linearstore` instead for improved broker performance. After installing this package, you will need to restart the Qpid broker to enable the durability feature.

To upgrade to the new Pulp release from version 2.3.x use yum to install the latest RPMs from the Pulp repository. To do this you can run:

```
$ sudo yum upgrade
```

After upgrading the packages on the system, you will need to upgrade the database schema by applying the database migrations. To apply migrations, your message broker needs to be configured and running. Run the database migrations as the `apache` user with the command:

```
$ sudo -u apache pulp-manage-db  # run this as the same user apache runs as
```

You can remove `/etc/pulp/logging/` if you like, as it is no longer used. Also, you can optionally edit the new `log_level` setting in the `[server]` section of `/etc/pulp/server.conf` to your preference:

```
$ sudo rm -rf /etc/pulp/logging/
$ sudo $EDITOR /etc/pulp/server.conf
```

Pulp 2.4.0 comes with some new services that perform distributed tasks using Celery. You can read about this more in the *Installation Guide*. You will need to enable Pulp's workers on at least one machine. Edit `/etc/default/pulp_workers` to your liking, and then enable and start the `pulp_workers` service. For Upstart systems:

```
$ sudo chkconfig pulp_workers on
$ sudo service pulp_workers start
```

For systemd systems:

```
$ sudo systemctl enable pulp_workers
$ sudo systemctl start pulp_workers
```

Warning: If you distribute Pulp across more than one server either through load balancing the HTTP requests, or through running pulp_workers on more than one machine, it is very important that you provide `/var/lib/pulp` as a shared filesystem to each host that is participating in the Pulp installation.

There are two more services that need to be running, but it is very important that only one instance of each of these runs across the entire Pulp installation.

Warning: `pulp_celerybeat` and `pulp_resource_manager` must both be singletons, so be sure that you only enable each of these on one host. They do not have to run on the same host, however. Note that each Pulp child node will also need its own instance of each of these services, as a Pulp child node is technically a separate distributed application from its parent.

On the host(s) that will run these two services (they do not have to run on the same host), edit `/etc/default/pulp_celerybeat` and `/etc/default/pulp_resource_manager` to your liking. Then enable and start the services. For Upstart:

```
$ sudo chkconfig pulp_celerybeat on
$ sudo service pulp_celerybeat start
$ sudo chkconfig pulp_resource_manager on
$ sudo service pulp_resource_manager start
```

For systemd:

```
$ sudo systemctl enable pulp_celerybeat
$ sudo systemctl start pulp_celerybeat
$ sudo systemctl enable pulp_resource_manager
$ sudo systemctl start pulp_resource_manager
```

After all Pulp servers and nodes have been upgraded, all consumers need to be re-registered. On each registered consumer, run `pulp-consumer update --keys` to exchange RSA keys needed for message authentication.

The Pulp 2.4.0 release includes an updated Admin Client which introduces new settings to the `/etc/pulp/admin/admin.conf` file. Install the updated Admin Client RPMs using the following command on any machine that already had the Admin Client installed:

```
$ sudo yum upgrade
```

If you made a backup of your admin.conf prior to this upgrade, you now need to manually merge your settings into `/etc/pulp/admin/admin.conf`. Do not overwrite this file, as there are some important new settings that must be present in `admin.conf`, for example the new `verify_ssl` and `ca_path` settings.

Lastly, merge the `/etc/pulp/nodes.conf.rpmnew` file which has also introduced new required settings. The Pulp team has plans to fix our configuration loaders to no longer require settings to be present to alleviate these issues.

Rest API Changes

Call Reports Every API that returns a Call Report with an HTTP 202 ACCEPTED response code has changed. For the sake of brevity, we will not list every API that returns 202 here. The structure of the Call Report has been changed significantly. The 2.3 Call Report had many more fields than the 2.4 Call Report does.

- The spawned_tasks list within the Call Report object does not contain the full list of all tasks that will be scheduled for a given call. Each spawned task is responsible for spawning whatever additional tasks are needed in order to complete processing. For example, the sync task with auto publishing enabled returns a Call Report that only lists the task_id for the sync portion of the work. When the sync task finishes it will have the task created for publishing listed in the spawned_tasks field.
- The exception and traceback fields have been deprecated from the Call Report and Task Report objects. In place of those fields a new "error" object has been created and will be returned.

Scheduled Calls

The Scheduled Call data structure has changed substantially.

- `last_run` is now `last_run_at`.

- `args` and `kwargs` are now top-level attributes of the object.
- `task` is a new attribute that is the python path to the task this schedule will execute.
- `resource` is a new attribute that is a globally-unique identifier for the object. this task will operate on. It is used internally to query schedules based on a given resource.

CRUD operations on schedules no longer depend on resource locking, so these API operations will never return a 202 or 409.

Schedule delete no longer returns a 404 when the schedule is not found. It will return a 200, because this is exactly the condition the user asked for.

Other Changes Here are other APIs that have changed, arranged by path:

`/v2/catalog/<source_id>/`

This is a new API. See the developer documentation for more detail.

`/v2/consumers/<consumer_id>/actions/content/regenerate_applicability/` The original applicability generation API did not allow a consumer to request regeneration of its own applicability. To allow this, we have introduced this new API which can be used by consumers and is documented on the same page as other applicability APIs.

`/v2/content/actions/delete_orphans/`

This has been deprecated in version 2.4, in favor of `/v2/content/orphans/`.

`/v2/queued_calls/`

This API has been removed in 2.4, as queued and running tasks are accessed through the same Tasks API.

`/v2/repositories/` Documentation for POST states that each distributor object should contain a key named `distributor_type_id`, but the API was actually requiring it to be named `distributor_type`. The API has been changed to match the documentation, so any code providing distributors to that API will need to be modified.

`/v2/repositories/<repo_id>/actions/unassociate/` Unassociating units is no longer blocked when the user performing the action is different than the user that created the unit. This most notably has the effect of eliminating the restriction that units could not be removed from repositories that are synced via a feed. However, if a unit is removed from a repo populated via a feed, syncing the repo again will recreate the unit.

`/v2/queued_calls/<call_request_id>/`

This API has been removed in 2.4, as queued and running tasks are accessed through the same Tasks API.

`/v2/task_groups/`

This API has been removed in 2.4, as there is no longer any concept of Task Groups.

`/v2/task_groups/<call_request_group_id>/`

This API has been removed in 2.4, as there is no longer any concept of Task Groups.

`/v2/tasks/<task_id>/`

Pulp 2.4 has replaced the tasking system with a new distributed task system. Due to this change, the data structure returned by the tasks API has changed. One notable change is that this API now returns something we call a Task Report, when it used to return a Call Report. The term Call Report is still used in Pulp 2.4 to refer to the returned data structure from all APIs that use the HTTP 202 code. That object has links to this API, which returns a Task Report. The notable difference is that the Task Report contains much greater detail. Some notable differences between the 2.3 Call Report and the 2.4 Task Report:

- The following attributes no longer exist: `response`, `reasons`, `task_group_id`, and `schedule_id`.
- The `traceback` and `exception` attributes have been deprecated in 2.4 and will always be null. See the new `error` attribute.
- The `progress` attribute has been renamed to `progress_report`.
- The following attributes are new in 2.4: `task_type`, `queue`, `error`, and `spawned_tasks`.

Feel free to compare the 2.3 Call Report API and the 2.4 Task Report API on your own.

`/v2/tasks/search/`

This is a new API to search tasks by criteria.

Task Behavior Changes

- When asynchronous tasks are created, they will be returned in the waiting state. The postponed or rejected states are no longer supported.
- Agent-related tasks no longer timeout, and it is now at the caller's discretion as to how long to wait for task completion. The task *state* now reflects the progression of the task on the agent.

Binding API Changes

- The pulp.bindings.responses.Task model has changed substantially to reflect changes in the REST API's task section.
  - The `call_request_group_id` attribute no longer exists.
  - The `call_request_id` attribute has been renamed to `task_id`.
  - The `call_request_tags` attribute has been renamed to `tags`.
  - The `reasons` attribute no longer exists, as Tasks cannot be postponed or rejected anymore.
  - The `progress` attribute has been renamed to `progress_report` to reflect the same name change in the API.
  - The `response` attribute no longer exists, as Tasks cannot be postponed or rejected anymore.
  - The `is_rejected()` and `is_postponed()` methods have been removed.
- The `pulp.bindings.repository.update_repo_and_plugins(...)` method has been deprecated in favor of `pulp.bindings.repository.update(...)`.

Plugin API Changes

If you are a plugin author, these changes are relevant to you:

- The Importer and Distributor cancellation method signatures have changed. `cancel_sync_repo()` and `cancel_publish_repo()` both used to take multiple arguments. With the conversion to Celery, we no longer had a need for those extra arguments, so each call now receives only the Importer or Distributor instance (self). If you have written an Importer or a Distributor, you will need to adjust your method signatures accordingly in order to work with this release of Pulp.

1.2.4 Pulp 2.3 Release Notes

Pulp 2.3.0

CVE-2013-7450

Versions of Pulp < 2.3.0 distributed the same certificate authority key and certificate to all Pulp users[0]. This CA is used by the /login API call (pulp-admin login uses this call) to generate and sign a client certificate. This client certificate is then used for subsequent API calls.

Due to this vulnerability, remote attackers are able to obtain the CA key from the Pulp git repository and use it to generate valid client certificates for any Pulp installations that use the default CA. The Pulp documentation did not emphasize the importance of replacing this CA for production deployments, so there may be users who use this common CA key in production environments.

Users are urged to replace the CA certificate and key on any Pulp installations that began their life with a version less than 2.3.0. Upgrading alone is not sufficient, as Pulp upgrades do not replace existing CA key pairs. Versions of Pulp >= 2.3.0 do ship a utility (pulp-gen-ca-certificate) that is capable of generating a new CA keypair for you, but it should be noted that there are some known local attacks that this script is vulnerable to as well[1][2]. The best option is to generate your own CA certificate if you are concerned about these local attacks.

Thanks to Sander Bos for notifying the Pulp team that we had neglected to acquire a CVE for this vulnerability at the time of its discovery.

[0] CVE-2013-7450: https://bugzilla.redhat.com/show_bug.cgi?id=1003326 [1] CVE-2016-3095 (fixed in Pulp >= 2.8.2):

http://www.openwall.com/lists/oss-security/2016/04/06/3

[2] CVE-2016-3106 (fixed in Pulp >= 2.8.3): https://pulp.plan.io/issues/1827

New Features

- Repository sync and publish history is now available. See `pulp-admin repo history --help` for details, or see the developer guide for how to retrieve these via the REST API.
- Qpid SSL Certificates generated by the pulp-qpid-ssl-cfg script are no longer world readable. It is recommended that existing installations are updated manually. If the default locations were used the following changes would be be needed.
  - chmod 640 /etc/pki/pulp/qpid/*.crt
  - chgrp apache /etc/pki/pulp/qpid/*.crt
  - chmod 640 /etc/pki/pulp/qpid/nss/*
  - chgrp qpidd /etc/pki/pulp/qpid/nss/*
- OAuth authentication is enabled by default using generated credentials.
- The out-of-the-box CA (Certificate Authority) used by Pulp to sign and validate user login is generated during installation. Previously, the SSL private key and certificate were installed from the git repository. This means that each installation initially had the same key and certificate installed, although any production deployments should have been configured to use a custom CA. In 2.3, the CA key stored at `/etc/pki/pulp/ca.key` and certificate stored at `/etc/pki/pulp/ca.crt` will be uniquely generated for each install. The key and certificate are **not** updated during RPM upgrade. Users upgrading to 2.3 who chose not to deploy their own CA are encouraged to generate a new (unique) CA key and certificate by running: *pulp-gen-ca-certificate* as root. Then, restart httpd. pulp-admin users will need to login again.

Note: It is strongly recommended that Pulp deployments use custom CA certificates.

New Node Features

- Users can now limit the bandwidth and number of connections used during a sync.
- The authentication method used by *Nodes* has been changed to OAuth. Users upgrading Pulp servers that are functioning as child *Nodes* will need to update a new *Nodes* configuration file as specified in Nodes section of this user guide.

Bugs

You can see the complete list of over 100 bugs that were fixed in Pulp 2.3.0.

REST API Changes

- The consumer applicability API is vastly different and performs much faster. Please see the developer guide for details on the new API.

Internal API Changes

- Importers no longer pass the related repositories to the validate_config(...) method.
- Distributors now pass a pulp.plugins.conduits.repo_config.RepoConfigConduit instead of the related repositories to the validate_config(...) method. The RepoConfigConduit is used to provide methods for performing the kind of cross repository searching & validation that formerly had to be done manually by comparing the configuration of each related repository.

Upgrade Instructions for 2.2.x –> 2.3.0

To upgrade to the new Pulp release from version 2.2.x, you should begin by using yum to install the latest RPMs from the Pulp repository and run the database migrations:

```
$ sudo yum upgrade
$ sudo pulp-manage-db
```

To address CVE-2013-7450, you will need to replace your CA certificate and key. As mentioned above, there are some known CVE's that the new `pulp-gen-ca-certificate` is vulnerable to. Thus, the recommended upgrade strategy is to generate a new CA certificate and key yourself, unless you upgrade all the way to Pulp 2.8.3 where CVE-2016-3095 and CVE-2016-3106 have been fixed. If you are not concerned about these local attacks, you can use `pulp-gen-ca-certificate` to regenerate the CA with the risk of a local user being able to read the private key. If you wish to use the script:

```
$ sudo pulp-gen-ca-certificate
# pulp-gen-ca-certificate in 2.3.0 does not install the files with the correct SELinux context
$ sudo restorecon -R /etc/pki/pulp
```

Pulp 2.3.1

Bugs Fixed

The `pulp-qpid-ssl-cfg` tool displayed an incorrect path to the qpid configuration file.

1.2.5 Pulp 2.2 Release Notes

Pulp 2.2.0

This release adds support for Fedora 19 and drops support for Fedora 17.

New Features

- Child node synchronization can now be scheduled with an optional recurrence. More information can be found in the Nodes section of the user guide.
- Some consumer operations can now be canceled.
- Orphan removal performs much better.

Noteworthy Bugs Fixed

- 906420 - Periods in consumer and repository IDs no longer cause problems.

All Bugs

You can see the complete list of bugs that were fixed in Pulp 2.2.0.

Upgrade Instructions for 2.1.x –> 2.2.0

To upgrade to the new Pulp release from version 2.1.x, you should begin by using yum to install the latest RPMs from the Pulp repository and run the database migrations:

```
$ sudo yum upgrade
$ sudo pulp-manage-db
```

Then restart apache.

Pulp 2.2.1

New Features

- MongoDB replica sets are now supported. See the comments in `/etc/pulp/server.conf` under the `[database]` section for information on how to configure Pulp to use a replica set.

Bug Fixes

Multiple proxy-related issues related to authentication and HTTPS to the proxy were fixed in RHBZ #1022662 and RHBZ #1014368.

See the RPM-specific user guide for highlights of the most important bug fixes there. All bug fixes for this release can be seen at this link:

All Bug Fixes

1.2.6 Pulp 2.1 Release Notes

Pulp 2.1.1

This release provides bugfixes and also includes some performance improvements.

Changes

When making a REST API call to copy units between repositories, there is now an opportunity to limit which fields of those units get loaded into RAM and handed to the importer. This can enable great reductions in RAM use when copying units that contain a lot of metadata.

Notable Bugs

RHBZ #928087 - pickling error in pulp.log when cancelling sync tasks

RHBZ #949186 - The pycurl downloader times out on active downloads, even during very good transfer rates

All Bugs

You can see the complete list of bugs that were fixed in Pulp 2.1.1.

Upgrade Instructions for 2.1.0 –> 2.1.1

Before beginning the upgrade, stop Apache so that it doesn't cause any concurrency problems during the database migration step.

To upgrade to the new Pulp release from version 2.1.0, use yum to install the latest RPMs from the Pulp repository and run the database migrations:

```
$ sudo yum upgrade
$ sudo pulp-manage-db
```

After upgrading, start Apache again and Pulp 2.1.1 should be functioning properly.

Pulp 2.1.0

New Features

1. Pulp now has support for hierarchical collections of Pulp Servers that are able to synchronize with each other. These are called Pulp Nodes, and you can read more about them in the nodes section.

2. Unit counts within each repository are now tracked by type.
3. We now support Fedora 18 and Apache 2.4.
4. It is now possible to upgrade from Pulp 1.1 to 2.1.

Client Changes

1. The `pulp-consumer [bind, unbind]` operations have been moved into `pulp-consumer rpm [bind, unbind]`. These operations have been moved out of this project into the pulp_rpm project. You will need to install pulp-rpm-consumer-extensions to get the pulp-consumer rpm section to find these commands.
2. The `pulp-admin rpm consumer [list, search, update, unregister, history]` commands from the pulp_rpm project have been moved into this project, and can now be found under `pulp-admin consumer *`.

Noteworthy Bugs Fixed

RHBZ #872724 - Requesting package profile on consumer without a package profile results in error

RHBZ #878234 - Consumer group package_install, update and uninstall not returning correct result

RHBZ #916794 - pulp-admin orphan list, Performance & Memory concerns (~17 minutes and consuming consuming ~1.65GB memory). The –summary flag was removed and summary behavior was made the default when listing orphans. A new –details flag has been added to get the previous behavior.

RHBZ #918160 - Orphan list –summary mode isn't a summary. Listing orphans now returns a much smaller set of related fields (namely only the unit keys).

RHBZ #920792 - High memory usage (growth of 2GB) from orphan remove –all. All server-side orphan operations now use generators instead of database batch queries.

RFE Bugs

RHBZ #876725 - RFE - consumer/agent - support option to perform 'best effort' install of content. We will now avoid aborting an install when one of the packages is not available for installation.

All Bugs

You can see the complete list of bugs that were fixed in Pulp 2.1.0.

API Changes

Applicability API Changes We have improved the Content Applicability API significantly in this release. A few major enhancements are:

1. Added an optional `repo_criteria` parameter that can restrict applicability searches by repository.
2. Changed units specification format to be a dictionary keyed by Content Type ID and a list of units of that type as a value. You can also pass in an empty list corresponding to a Content Type ID to check the applicability of all units of that specific type.
3. All 3 parameters are now optional. Check out updated API documentation to read more about the behavior of the API in case of missing parameters.

4. Return format is updated to a more compact format keyed by Consumer ID and Content Type ID and it now returns only applicable units.

The API is documented in detail in the applicability API documentation.

Distributor Plugin API Change The Distributor plugin method `create_consumer_payload` has changed to accept a new parameter, `binding_config`. Individual bindings can contain configuration options that may be necessary when providing the consumer with the information necessary to use the published repository. This field will contain those options if specified by the user.

Upgrade Instructions for 2.0 –> 2.1

To upgrade to the new Pulp release from version 2.0, you should begin by using yum to install the latest RPMs from the Pulp repository, run the database migrations, and cleanup orphaned packages:

```
$ sudo yum upgrade
$ sudo pulp-manage-db
$ sudo pulp-admin orphan remove --all
```

1.3 Installation

Pulp operates with three main components that get installed potentially on different machines.

Server This is the main application server that stores data and distributes content.

Agent This component runs on consumers and communicates with the server to provide remote content management.

Client This is a command line component that comes as two pieces: admin-client, which manages the server; and consumer-client, which manages a consumer's relationship to the server. admin-client can be run from any machine that can access the server's REST API, but the consumer-client must be run on a consumer.

Additional steps are needed for upgrading Pulp 1.1 installations. More information can be found in the `v1_upgrade` section of this guide.

1.3.1 Supported Operating Systems

Server

- RHEL Server 6 & 7
- Fedora 19 & 20
- CentOS Server 6 & 7

Consumer

- RHEL 5, 6, & 7
- Fedora 19 & 20
- CentOS 5, 6 & 7

Admin Client

- RHEL 6 & 7
- Fedora 19 & 20

- CentOS 6 & 7

1.3.2 Prerequisites

- The following ports must be open into the server:
- 80 for consumers to access repositories served over HTTP
- 443 for consumers to access repositories served over HTTPS
- 443 for clients (both admin and consumer) to access Pulp APIs
- 5672 for consumers to connect to the message bus if it is left unsecured
- 5671 for consumers to connect to the message bus if it is running over HTTPS
- The mod_python Apache module must be uninstalled or not loaded. Pulp uses mod_wsgi which conflicts with mod_python and will cause the server to fail.

Warning: The python-qpid package is not available from Pulp installation repositories on RHEL 5 or CentOS 5. This will prevent management of RHEL 5 or CentOS 5 clients with pulp-consumer using Qpid. Users who want to use Qpid instead of RabbitMQ and manage these RHEL 5 or CentOS 5 clients will need to build python-qpid from source.

Warning: MongoDB is known to have serious limitations on 32-bit architectures. It is strongly advised that you run MongoDB on a 64-bit architecture.

1.3.3 Storage Requirements

The MongoDB database can easily grow to 10GB or more in size, which vastly exceeds the amount of data actually stored in the database. This is normal (but admittedly surprising) behavior for MongoDB. As such, make sure you allocate plenty of storage within `/var/lib/mongodb`.

1.3.4 Repositories

1. Download the appropriate repo definition file from the Pulp repository:

- Fedora: https://repos.fedorapeople.org/repos/pulp/pulp/fedora-pulp.repo
- RHEL: https://repos.fedorapeople.org/repos/pulp/pulp/rhel-pulp.repo

Note: If you would like to install the RHEL 5 Client for Pulp, please use the RHEL 5 repository.

2. For RHEL and CentOS systems, the EPEL repositories are required. Following commands will add the appropriate repositories for RHEL6 and RHEL7 respectively:

   RHEL6:

```
$ sudo rpm -Uvh https://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
```

   RHEL7:

```
$ sudo rpm -Uvh https://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-5.noarch.rpm
```

Note: The above EPEL URLs change with each release of the epel-release package. If you receive a 404 error when attempting to install the above RPM, point your browser at the directory it is in and look for the updated name of epel-release package.

Note: EPEL requires users of RHEL 6.x to enable the `optional` repository, and users of RHEL 7.x to additionally enable the `extras` repository. Details are described here.

3. For RHEL 5 systems, subscribe to the following RHN channels:

- MRG Messaging v. 1
- MRG Messaging Base v. 1

Note: See the Qpid packaging docs or the RabbitMQ installation docs for information on where to get broker packages for your OS.

1.3.5 Server

1. You must provide a running MongoDB instance for Pulp to use. You can use the same host that you will run Pulp on, or you can give MongoDB its own separate host if you like. You can even use MongoDB replica sets if you'd like to have higher availability. For yum based systems, you can install MongoDB with this command:

```
$ sudo yum install mongodb-server
```

You need mongodb-server with version >= 2.4 installed for Pulp server. It is highly recommended that you configure MongoDB to use SSL. If you are using Mongo's authorization feature, you will need to grant the `readWrite` and `dbAdmin` roles to the user you provision for Pulp to use. The `dbAdmin` role allows Pulp to create collections and install indices on them.

After installing MongoDB, you should configure it to start at boot and start it. For Upstart based systems:

```
$ sudo service mongod start
$ sudo chkconfig mongod on
```

For systemd based systems:

```
$ sudo systemctl enable mongod
$ sudo systemctl start mongod
```

Warning: On new MongoDB installations, MongoDB takes some time to preallocate large files and will not accept connections until it finishes. When this happens, Pulp will wait for MongoDB to become available before starting.

1. You must also provide a message bus for Pulp to use. Pulp will work with Qpid or RabbitMQ, but is tested with Qpid, and uses Qpid by default. This can be on the same host that you will run Pulp on, or elsewhere as you please. To install Qpid on a yum based system, use this command:

```
$ sudo yum install qpid-cpp-server qpid-cpp-server-store
```

Note: In environments that use Qpid, the `qpid-cpp-server-store` package provides durability, a feature that saves broker state if the broker is restarted. This is a required feature for the correct operation of

Pulp. Qpid provides a higher performance durability package named `qpid-cpp-server-linearstore` which can be used instead of `qpid-cpp-server-store`, but may not be available on all versions of Qpid. If `qpid-cpp-server-linearstore` is available in your environment, consider uninstalling `qpid-cpp-server-store` and installing `qpid-cpp-server-linearstore` instead for improved broker performance. After installing this package, you will need to restart the Qpid broker to enable the durability feature.

Pulp uses the `ANONYMOUS` Qpid authentication mechanism by default. To enable username/password-based `PLAIN` broker authentication, you will need to configure SASL with a username/password, and then configure Pulp to use that username/password. Refer to the Qpid docs on how to configure username/password authentication using SASL. Once the broker is configured, configure Pulp according to the docs on using *Pulp with Qpid and username/password authentication*.

The server can be *optionally* configured so that it will connect to the broker using SSL by following the steps defined in the Qpid SSL Configuration Guide. By default, Pulp does not expect to use SSL and will connect to the broker using a plain TCP connection to localhost.

After installing and configuring Qpid, you should configure it to start at boot and start it. For Upstart based systems:

```
$ sudo service qpidd start
$ sudo chkconfig qpidd on
```

For systemd based systems:

```
$ sudo systemctl enable qpidd
$ sudo systemctl start qpidd
```

2. Install the Pulp server, task workers, and their dependencies. For Pulp installations that use Qpid, install Pulp server using:

```
$ sudo yum groupinstall pulp-server-qpid
```

Note: For RabbitMQ installations, install Pulp server without any Qpid specific libraries using `sudo yum groupinstall pulp-server`. You may need to install additional RabbitMQ dependencies manually.

3. Edit `/etc/pulp/server.conf`. Most defaults will work, but these are sections you might consider looking at before proceeding. Each section is documented in-line.
   - **email** if you intend to have the server send email (off by default)
   - **database** if your database resides on a different host or port. It is strongly recommended that you set ssl and verify_ssl to True.
   - **messaging** if your message broker for communication between Pulp components is on a different host or if you want to use SSL. For more information on this section refer to the *Pulp Broker Settings Guide*.
   - **tasks** if your message broker for asynchronous tasks is on a different host or if you want to use SSL. For more information on this section refer to the *Pulp Broker Settings Guide*.
   - **server** if you want to change the server's URL components, hostname, or default credentials
4. Initialize Pulp's database. It is important that the broker is running before initializing Pulp's database. It is also important to do this before starting Apache or any Pulp services. The database initialization needs to be run as the `apache` user, which can be done by running:

```
$ sudo -u apache pulp-manage-db
```

Note: If Apache or Pulp services are already running, restart them after running the `pulp-manage-db` command.

Warning: It is recommended that you configure your web server to refuse SSLv3.0. In Apache, you can do this by editing `/etc/httpd/conf.d/ssl.conf` and configuring the `SSLProtocol` directive like this:

```
`SSLProtocol all -SSLv2 -SSLv3`
```

1. Start Apache httpd and set it to start on boot. For Upstart based systems:

```
$ sudo service httpd start
$ sudo chkconfig httpd on
```

For systemd based systems:

```
$ sudo systemctl enable httpd
$ sudo systemctl start httpd
```

2. Pulp has a distributed task system that uses Celery. Begin by configuring, enabling and starting the Pulp workers. To configure the workers, edit `/etc/default/pulp_workers`. That file has inline comments that explain how to use each setting. After you've configured the workers, it's time to enable and start them. For Upstart systems:

```
$ sudo chkconfig pulp_workers on
$ sudo service pulp_workers start
```

For systemd systems:

```
$ sudo systemctl enable pulp_workers
$ sudo systemctl start pulp_workers
```

Note: The pulp_workers systemd unit does not actually correspond to the workers, but it runs a script that dynamically generates units for each worker, based on the configured concurrency level. You can check on the status of those generated workers by using the `systemctl status` command. The workers are named with the template `pulp_worker-<number>`, and they are numbered beginning with 0 and up to `PULP_CONCURRENCY - 1`. For example, you can use `sudo systemctl status pulp_worker-1` to see how the second worker is doing.

3. There are two more services that need to be running, but it is important that these two only run once each (i.e., do not enable either of these on any more than one Pulp server).

Warning: `pulp_celerybeat` and `pulp_resource_manager` must both be singletons, so be sure that you only enable each of these on one host if you are Pulp's clustered deployment.

On some Pulp system, configure, start and enable the Celerybeat process. This process performs a job similar to a cron daemon for Pulp. Edit `/etc/default/pulp_celerybeat` to your liking, and then enable and start it. Again, do not enable this on more than one host. For Upstart:

```
$ sudo chkconfig pulp_celerybeat on
$ sudo service pulp_celerybeat start
```

For systemd:

```
    $ sudo systemctl enable pulp_celerybeat
    $ sudo systemctl start pulp_celerybeat
```

Lastly, one `pulp_resource_manager` process must be running in the installation. This process acts as a task router, deciding which worker should perform certain types of tasks. Apologies for the repetitive message, but it is important that this process only be enabled on one host. Edit `/etc/default/pulp_resource_manager` to your liking. Then, for upstart:

```
    $ sudo chkconfig pulp_resource_manager on
    $ sudo service pulp_resource_manager start
```

For systemd:

```
    $ sudo systemctl enable pulp_resource_manager
    $ sudo systemctl start pulp_resource_manager
```

1.3.6 Admin Client

The Pulp Admin Client is used for administrative commands on the Pulp server, such as the manipulation of repositories and content. The Pulp Admin Client can be run on any machine that can access the Pulp server's REST API, including the server itself. It is not a requirement that the admin client be run on a machine that is configured as a Pulp consumer.

Pulp admin commands are accessed through the `pulp-admin` script.

1. Install the Pulp admin client packages:

```
$ sudo yum groupinstall pulp-admin
```

2. Update the admin client configuration to point to the Pulp server. Keep in mind that because of the SSL verification, this should be the fully qualified name of the server, even if it is the same machine (localhost will not work with the default apache generated SSL certificate). Regardless, the "host" setting below must match the "CN" value of the server's HTTP SSL certificate. This change is made globally to the `/etc/pulp/admin/admin.conf` file, or for one user in `~/.pulp/admin.conf`:

```
[server]
host = localhost.localdomain
```

1.3.7 Consumer Client And Agent

The Pulp Consumer Client is present on all systems that wish to act as a consumer of a Pulp server. The Pulp Consumer Client provides the means for a system to register and configure itself with a Pulp server. Additionally, the Pulp Consumer Client runs an agent that will receive messages and commands from the Pulp server.

Pulp consumer commands are accessed through the `pulp-consumer` script. This script must be run as root to permit access to add references to the Pulp server's repositories.

1. For environments that use Qpid, install the Pulp consumer client, agent packages, and Qpid specific consumer dependencies with one command by running:

```
$ sudo yum groupinstall pulp-consumer-qpid
```

Note: For RabbitMQ installations, install the Pulp consumer client and agent packages without any Qpid specific dependencies using `sudo yum groupinstall pulp-consumer`. You may need to install additional RabbitMQ dependencies manually including the `python-gofer-amqp` package.

2. Update the consumer client configuration to point to the Pulp server. Keep in mind that because of the SSL verification, this should be the fully qualified name of the server, even if it is the same machine (localhost will not work with the default Apache generated SSL certificate). Regardless, the "host" setting below must match the "CN" value of the server's HTTP SSL certificate. This change is made to the `/etc/pulp/consumer/consumer.conf` file:

```
[server]
host = localhost.localdomain
```

3. The agent may be configured so that it will connect to the Qpid broker using SSL by following the steps defined in the Qpid SSL Configuration Guide. By default, the agent will connect using a plain TCP connection.

4. Set the agent to start at boot. For upstart:

```
    $ sudo chkconfig goferd on
    $ sudo service goferd start
```

   For systemd:

```
    $sudo systemctl enable goferd
    $sudo systemctl start goferd
```

1.3.8 SSL Configuration

By default, all of the client components of Pulp will require validly signed SSL certificates from the servers on remote ends of its outbound connections. On a brand new httpd installation, a self-signed certificate will be generated for the server to use to serve Pulp. This means that a fresh installation will experience client errors similar to this:

```
(pulp)[rbarlow@coconut pulp]$ pulp-admin puppet repo list
+----------------------------------------------------------------------+
Puppet Repositories
+----------------------------------------------------------------------+

WARNING: The server's SSL certificate is untrusted!

The server's SSL certificate was not signed by a trusted authority. This could
be due to a man-in-the-middle attack, or it could be that the Pulp server needs
to have its certificate signed by a trusted authority. If you are willing to
accept the associated risks, you can set verify_ssl to False in the client
config's [server] section to disable this check.
```

You have two choices to solve this issue: You may make or acquire signed SSL certificates for httpd to use to serve Pulp, or you may configure Pulp's various clients not to perform SSL signature validation.

Signed Certificates

If you wish to use signed certificates, you must decide whether you will purchase signed certificates from a root certificate authority or use your own organization's certificate authority. How to make or buy signed certificates is outside the scope of this document. We will assume that you have these items:

1. A PEM-encoded X.509 certificate file, signed by a trusted certificate authority.
2. A PEM-encoded private key file that corresponds to your SSL certificate.
3. The CA certificate that signed your SSL certificate. This is only necessary if your Linux distribution does not already include the CA that signed your certificate in its system CA pack.

You must first configure httpd to use the SSL certificate and private key you have acquired. You must configure the SSLCertificateFile and SSLCertificateKeyFile mod_ssl directives to point at these files. On Red Hat based systems, these settings can be found in `/etc/httpd/conf.d/ssl.conf`.

If you are using a CA certificate that is not already trusted by your operating system's system CA pack, you may either configure Pulp to trust that CA, or you may configure your operating system to trust that CA.

Pulp has a setting called `ca_path` in these files: `/etc/pulp/admin/admin.conf`, `/etc/pulp/consumer/consumer.conf`, and `/etc/pulp/nodes.conf`. This setting indicates which CA pack each of these components should use when validating Pulp server certificates. By default, Pulp will use the operating system's CA pack. If you wish, you may adjust this setting to point to a different CA pack. The CA pack may be a single file that contains multiple concatenated certificates, or it may be a directory with OpenSSL style hashed symlinks pointing at CA certificate files, with one certificate per file. Of course, if you have exactly one CA certificate, you can configure this setting to point at it directly.

There are three settings in `/etc/pulp/server.conf` that you should be aware of, but probably should not alter. `capath` and `cakey` point to a CA certificate and key that Pulp uses to sign client authentication certificates. Note that this is not the CA that you signed your server certificate with earlier. It is used only internally by Pulp and Apache to create client certificates with login calls, and to validate those certificates when clients use the API. It is best to avoid altering these settings. The third setting is confusingly named `ssl_ca_certificate`. This setting should not be used, since it causes a chicken and egg situation that could cause the universe to experience a machine check exception. If it is configured, the yum consumer handlers will use this CA in their yum repository files for validating the Pulp server. The problem is that the consumer must have already trusted Pulp in order to have registered to Pulp to get this CA file, which helps the consumer to trust Pulp. It's best for users to configure CA trust themselves outside of Pulp, which is why this setting should not be used.

Warning: The Pulp team plans to deprecate the `cacert`, `cakey`, and `ssl_ca_certificate` settings. It is best to avoid altering these settings from their defaults, as described above. See 1123509 and 1165403.

If you want to use SSL with Qpid, see the Qpid SSL Configuration Guide.

Turning off Validation

Warning: It is strongly recommended that you make or acquire *Signed Certificates* to prevent man-in-the-middle attacks or other nefarious activities. It is very risky to assume that the other end of the connection is who they claim to be. SSL uses a combination of encryption and authentication to ensure private communication. Disabling these settings removes the authentication component from the SSL session, which removes the guarantee of private communication since you can't be sure who you are communicating with.

Pulp has a setting called `verify_ssl` in these files: `/etc/pulp/admin/admin.conf`, `/etc/pulp/consumer/consumer.conf`, `/etc/pulp/nodes.conf`, and `/etc/pulp/repo_auth.conf`. If you configure these settings to false, the respective Pulp components will no longer validate the Pulp server's certificate signature.

1.3.9 Pulp Broker Settings

To configure Pulp to work with a non-default broker configuration read the *Pulp Broker Settings Guide*.

1.3.10 MongoDB Authentication

To configure Pulp for connecting to the MongoDB with username/password authentication, use the following steps: 1. Configure MongoDB for username password authentication. See MongoDB - Enable Authentication for details. 2.

In `/etc/pulp/server.conf`, find the `[database]` section and edit the `username` and `password` values to match the user configured in step 1. 3. Restart the httpd service

```
$ sudo service httpd restart
```

1.4 Scaling Pulp

Great effort has been put into Pulp to make it scalable. A default Pulp install is an "all-in-one" style setup with everything running on one machine. However, Pulp supports a clustered deployment across multiple machines and/or containers to increase availability and performance.

1.4.1 Overview of Pulp Components

Pulp consists of several components:

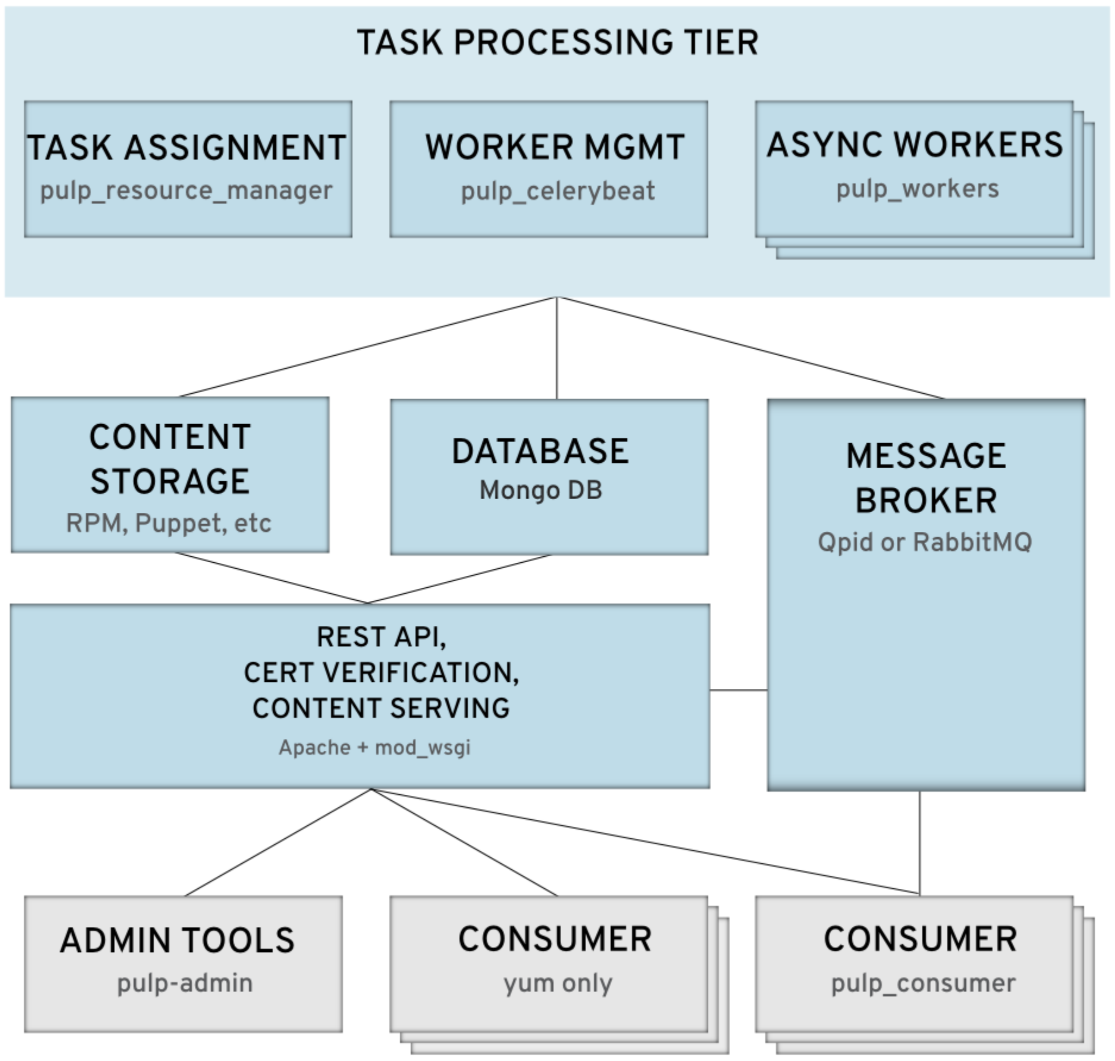
- `httpd` - The webserver process serves published repositories and handles Pulp REST API requests. Simple requests like repository creation are handled immediately whereas longer tasks are asynchronously processed by a worker.
- `pulp_workers` - Worker processes handle longer running tasks asynchronously, like repository publishes and syncs.
- `pulp_celerybeat` - The celerybeat process discovers and monitors workers. Additionally, it performs task cancellations in the event of a worker shutdown or failure. The celerybeat process also initiates scheduled tasks, and automatically cancels tasks that have failed more than *X* times. This process also initiates periodic jobs that Pulp runs internally. In a Pulp cluster, exactly one of these should be running!
- `pulp_resource_manager` - The resource manager assigns tasks to workers, and ensures multiple conflicting tasks on a repo are not executed at the same time. In a Pulp cluster, exactly one of these should be running!

Additionally, Pulp relies on other components:

- MongoDB - the database for Pulp
- Apache Qpid or RabbitMQ - the queuing system that Pulp uses to assign work to workers. Pulp can operate equally well with either Qpid or RabbitMQ.

Warning: It is critical to note that `pulp_celerybeat` and `pulp_resource_manager` should *never* have more than a single instance running under any circumstance!

The diagram below shows an example default deployment.



1.4.2 Choosing What to Scale

Not all Pulp installations are used in the same way. One installation may have hundreds of thousands of RPMs, another may have a smaller number of RPMs but with lots of consumers pulling content to their systems. Others may sync frequently from a number of upstream sources.

A good first step is to figure out how many systems will be pulling content from your Pulp installation at any given time. This includes RPMs, Puppet modules, Docker layers, OSTree layers, Python packages, etc. RPMs are usually pulled down on a regular basis as part of a system update schedule, but other types of content may be fetched in a more ad-hoc fashion.

If the number of concurrent downloads seems large, you may want to consider adding additional servers to service httpd requests. See the *Scaling httpd* section for more information.

If you expect to maintain a large set of repositories that get synced frequently, you may want to add additional servers for worker processes. Worker processes handle long-running tasks such as content downloads from external sources and also perform actions like repository metadata regeneration on publish. See the *Scaling workers* section for more information.

Another consideration for installations with a large number of repositories or repositories with a large numbers of RPMs is to have a dedicated server or set of servers for MongoDB. Pulp does not store actual content in the MongoDB database, but all metadata is stored there. More information on scaling MongoDB is available in the MongoDB Deployment docs.

Pulp uses either RabbitMQ or Apache Qpid as its messaging backend. Pulp does not generate many messages in comparison to other applications, so it is not expected that the messaging backend would need to be scaled for performance unless the number of concurrent consumer connections is large. However, additional configuration may be done to make the messaging backend more fault tolerant. Examples of this are available in the Apache Qpid HA docs and the RabbitMQ HA docs.

Warning: There is a bug in versions of Apache Qpid older than 0.30 that involves running out of file descriptors. This is an issue on deployments with large numbers of consumers. See RHBZ #1122987 for more information about this and for suggested workarounds.

1.4.3 Scaling httpd

Additional httpd servers can be added to Pulp to increase both throughput and redundancy.

In situations when there are more incoming HTTP or HTTPS requests than a single server can respond to, it may be time to add additional httpd servers. httpd serves both the Pulp API and content, so increasing capacity could improve both API and content delivery performance.

Consider using the Apache mod_status scoreboard to monitor how busy your httpd workers are.

Note: Pulp itself does not provide httpd load balancing capabilities. See the *Load Balancing Requirements* for more information.

To add additional httpd server capacity, configure the desired number of *Pulp clustered servers* and start `httpd` on them. Remember only one instance of `pulp_celerybeat` and `pulp_resource_manager` should be running across all *Pulp clustered servers*.

1.4.4 Scaling workers

Additional Pulp workers can be added to increase asynchronous work throughput and redundancy.

To add additional Pulp worker capacity, configure the desired number of *Pulp clustered servers* according to the the *clustering* docs and start `pulp_workers` on each of them. Remember only one instance of `pulp_celerybeat` and `pulp_resource_manager` should be running across all *Pulp clustered servers*.

1.4.5 Clustering Pulp

A clustered Pulp installation is comprised of two or more *Pulp clustered servers*. The term *Pulp clustered server* is used to distinguish it as a separate concept from *Nodes*. *Pulp clustered servers* share the following components:

Pulp Configuration	Pulp reads its configuration from conf files inside `/etc/pulp`.
Pulp Files	Pulp stores files on disk within `/var/lib/pulp`.
Certificates	By default, Pulp keeps certificates in `/etc/pki/pulp`.
MongoDB	All clustered Pulp servers must connect to the same MongoDB.
AMQP Bus	All consumers and servers must connect to the same AMQP bus.

Filesystem Requirements

Pulp requires a shared filesystem for *Pulp clustered servers* to run correctly. Sharing with NFS has been tested, but any shared filesystem will do. Pulp expects all shared filesystem directories to be mounted in their usual locations.

The following permissions are required for a *Pulp clustered server* to operate correctly.

User	Directory	Permission
apache	`/etc/pulp`	Read
apache	`/var/lib/pulp`	Read, Write
apache	`/etc/pki/pulp`	Read, Write
root	`/etc/pki/pulp`	Read

For more details on using NFS for sharing the filesystem with Pulp, see *Sharing with NFS*.

SELinux Requirements

Pulp clustered servers with SELinux in Enforcing mode need the following SELinux file contexts for correct operation:

Directory	SELinux Context
`/etc/pulp`	system_u:object_r:httpd_sys_rw_content_t:s0
`/var/lib/pulp`	system_u:object_r:httpd_sys_rw_content_t:s0
`/etc/pki/pulp`	system_u:object_r:pulp_cert_t:s0

For more details on using NFS with SELinux and Pulp, see *Sharing with NFS*.

Server Settings

Several Pulp settings default to `localhost`, which won't work in a clustered environment. In `/etc/pulp/server.conf` the following settings should be set, at a minimum, for correct Pulp clustering operation.

Section	Setting Name	Recommended Value
[server]	host	Update with the name used by your load balancer.
[database]	seeds	Update with the hostname and port of your network accessible MongoDB installation.
[messaging]	url	Update with the hostname and port of your network accessible AMQP bus installation.
[tasks]	broker_url	Update with the hostname and port of your network accessible AMQP bus installation.

Load Balancing Requirements

To effectively handle inbound HTTP/HTTPS requests to *Pulp clustered servers* running `httpd`, load balancing of some sort should be used. *Pulp clustered servers* not running `httpd` do not need to be involved in load balancing. Configuring load balancing is beyond the scope of Pulp documentation, but there are a few recommendations.

One option is to use a dedicated load balancer. Pulp defaults to using SSL for webserver traffic, so an easy thing is to use a TCP based load balancer. HAProxy has been tested with a clustered Pulp installation, but any TCP load balancer should work.

Another option is to use DNS based load balancing. Community users have reported this works, but it has not been explicitly tested by Pulp developers.

With either load balancing technique, all *Pulp clustered servers* running `httpd` need to be configured with SSL certificates which have the CN set to the hostname of the TCP load balancer or the DNS record providing load balancing. This ensures that as traffic arrives at Pulp webservers, clients will trust the certificate presented by the *Pulp clustered server*.

Clustered Logging

Pulp logs in the same way on a clustered server as it does for a single server. For more information on how Pulp logs, see *Logging*. To setup remote logging and aggregation, refer to the documentation for the log daemon running on your system.

Cluster Monitoring

A clustered deployment can be monitored with the techniques described in *Getting the Server Status*.

Warning: Information provided by the `/status/` API call does not include `httpd` status information. It is recommended that each *Pulp clustered server* acting as a webserver have its `/status/` API queried directly. If queried through the load balancer, the request may route to `httpd` servers in unexpected ways. See issue #915 for more information.

Consumer Settings

Consumers use a similar configuration as they would in a non-clustered environment. At a minimum there are two areas of `/etc/pulp/consumer/consumer.conf` which need updating.

- The `host` value in the `[server]` needs to be updated with the load balancer's hostname. This causes web requests from consumers to flow through the load balancer.
- The `[messaging]` section needs to be updated to use the same AMQP bus as the server.

Warning: Machines acting as a *Pulp clustered nodes* cannot be registered as a consumer until #859 is resolved.

Pulp Admin Settings

When using a clustered deployment, it is recommended to configure `pulp-admin` to connect to the load balancer hostname. To do this, add the following snippet to `~/.pulp/admin.conf`

```
[server]
host: example.com

# This example assumes example.com is your load balancer or DNS record
# providing load balancing
```

Sharing with NFS

NFS has been tested with Pulp to share the `/etc/pulp`, `/var/lib/pulp`, and `/etc/pki/pulp` sections of the filesystem, but any shared filesystem should work. Typically *Pulp clustered servers* will act as NFS clients, and a third party machine will act as the NFS server.

Warning: Exporting the same directory name (ie: pulp) multiple times can cause the NFS client to incorrectly believe it has already mounted the export. Use the NFS option `fsid` with integer numbers to uniquely identify NFS exports.

NFS expects user ids (UID) and group ids (GID) of a client to map directly with the UID and GID on the server. To keep your NFS export config simple, it is recommended that all NFS servers and clients have the same UID and GID for the user `apache`. If they differ throughout the cluster, use NFS options to map UIDs and GIDs accordingly.

Most NFS versions by default squash root which prevents `root` on NFS clients from automatically having root access on the NFS server. This typically prevents `root` on a *Pulp clustered server* from having the necessary Read access on `/etc/pki/pulp`. One secure way to workaround this without opening up root access on the NFS server is to use the `anonuid` and `anongid` NFS options to specify the UID and GID of `apache` on the NFS server. This will effectively provide `root` on the NFS client with read access to the necessary files in `/etc/pki/pulp`.

If using SELinux in Enforcing mode, specify the necessary *SELinux Requirements* with the NFS option `context`.

1.5 Tuning and Monitoring

1.5.1 Tuning

WSGI Processes

By default, each Apache server on which Pulp is deployed will start 3 WSGI processes to serve the REST API. The number of processes can be adjusted in `/etc/httpd/conf.d/pulp.conf` on the `WSGIDaemonProcess` statement, along with other items. See the Apache documentation of `mod_wsgi` for details.

For tuning purposes, consider Pulp's REST API to be a low-traffic web application that has occasional spikes in memory use when returning large data sets. Most of pulp's heavy-lifting has been offloaded to celery workers.

Pulp and Mongo Database

Pulp uses Mongo to manage repository information as well as content metadata. Mongo can be run on the same machine as Pulp, but we recommend that it run on dedicated hardware for larger production deployments. At this time, Pulp can be used with replication but does not support sharding.

1.5.2 Monitoring

Monitoring for outages

While Pulp has a number of processes, users will interact with Pulp via httpd. At a minimum, your monitoring system should alert for the following issues:

- *httpd* is not responsive on ports 80 or 443
- storage volumes associated with Pulp are about to run out of space
- Mongo is not responsive
- Apache Qpid or RabbitMQ is not responsive

You may also want to alert if no Pulp workers are available. This is optional since it affects long-running background tasks like syncing and publishing but would not affect content downloads for consumer systems.

Please consult the documentation of your monitoring software for information on how to check for these types of issues.

Monitoring for performance issues

Performance issues fall into a number of categories. However, here are some typical statistics that can be collected and reviewed periodically:

- work queue depth
- repository sync time
- repository publish time
- concurrent *httpd* connections to ports 80 and 443
- storage volume space usage

Many of these statistics can be collected and viewed using tools like Celery Flower or Munin.

1.6 Pulp Broker Settings

Pulp requires a message bus to run. Either Qpid or RabbitMQ can be used as that message bus. Pulp is developed and tested against the Qpid C++ server v0.22+ and is configured to expect Qpid on localhost without SSL or authentication by default. This documentation identifies changes necessary for the following configurations:

- Pulp Broker Settings Overview
- Configure Pulp to use Qpid on a different host
- Configure Pulp to use Qpid with SSL
- Configure Pulp to use RabbitMQ without SSL
- Configure Pulp to use RabbitMQ with SSL

1.6.1 Pulp Broker Settings Overview

Pulp uses the message broker in two ways:

- For Pulp Server <–> Pulp Consumer Agent communication such as a server initiated bind or update.
- For Pulp Server <–> Pulp Worker asynchronous, server-side tasks such as syncing, publishing, or deletion of content.

Pulp Server settings are contained in `/etc/pulp/server.conf` and are located in two sections corresponding with the two ways Pulp uses the message broker. The Pulp Server <–> Pulp Consumer Agent communication settings are contained in the `[messaging]` section. The asynchronous task settings are contained in the `[tasks]` section. Refer to the inline documentation of those sections for more information on the options and their usage.

All settings in `[tasks]` and `[messaging]` have a default. If a setting is not specified because it is either omitted or commented out, the default is used. The default values for each option are shown but commented out in `/etc/pulp/server.conf`.

Pulp Consumer Agent settings are contained in `/etc/pulp/consumer/consumer.conf` in the `[messaging]` section and define how the Consumer Agent connects to the broker to communicate with the Pulp Server. The `[messaging]` section of `/etc/pulp/consumer/consumer.conf` on each Pulp Consumer and the `[messaging]` section of `/etc/pulp/server.conf` on each Pulp Server need to connect to the same broker for correct operation. The values and settings in `/etc/pulp/consumer/consumer.conf` correspond with the settings in `/etc/pulp/server.conf`, but uses a slightly different setting names. Refer to the inline documentation in the `[messaging]` section of `/etc/pulp/consumer/consumer.conf` for more information on how to configure the settings of a consumer.

These two areas of Pulp can use the same message bus, or not. There is not a requirement that these use the same broker.

To apply your changes after making any adjustment to `/etc/pulp/server.conf`, you should restart all Pulp services on any Pulp Server using the `/etc/pulp/server.conf` file edited. To apply your changes made to a `/etc/pulp/consumer/consumer.conf` file, restart the Consumer Agent (`goferd`) on any Consumer that uses that file. Normally each configuration file is kept individually on each computer (Server or Consumer), and in those cases you only restart the corresponding service on that specific machine. For more custom environments where config files are shared between servers or consumers you may need to restart services on multiple computers.

1.6.2 Qpid on localhost (the default settings)

The default Pulp settings assume that both Pulp Server <–> Pulp Consumer Agent communication and Pulp Server <–> Pulp Worker communication use Qpid on localhost at the default port (5672) without SSL and without authentication. All settings in the `[messaging]` and `[tasks]` sections are commented out by default, so the default values are used. The defaults are included in the commented lines for clarity.

```
[messaging]
# url: tcp://localhost:5672
# transport: qpid
# auth_enabled: true
# cacert: /etc/pki/qpid/ca/ca.crt
# clientcert: /etc/pki/qpid/client/client.pem
# topic_exchange: 'amq.topic'

[tasks]
# broker_url: qpid://guest@localhost/
# celery_require_ssl: false
# cacert: /etc/pki/pulp/qpid/ca.crt
# keyfile: /etc/pki/pulp/qpid/client.crt
# certfile: /etc/pki/pulp/qpid/client.crt
```

The default settings of a Pulp Consumer Agent are found in `/etc/pulp/consumer/consumer.conf` and assume Qpid is running on localhost at the default port (5672) without SSL and without authentication. In almost all installations, at a minimum, the `host` attributed will need to be updated. The default configuration is shown below. If `host` in the `[messaging]` section is blank, the `host` attribute in the `[server]` section of `/etc/pulp/consumer/consumer.conf` is used, which defaults to `localhost.localdomain`.

```
[messaging]
scheme = tcp
host =
port = 5672
transport = qpid
cacert =
clientcert =
```

1.6.3 Qpid on a Different Host

To use Qpid on a different host for the Pulp Server <–> Pulp Consumer Agent communication, update the `url` parameter in the `[messaging]` section. For example, if the hostname to connect to is `someotherhost.com` uncomment `url` and set it as follows:

```
url: tcp://someotherhost.com:5672
```

The `/etc/pulp/consumer/consumer.conf` file on each Pulp Consumer also needs to be updated to correspond with this change. Refer to the inline documentation in `/etc/pulp/consumer/consumer.conf` to set the configuration correctly.

To use Qpid on a different host for Pulp Sever <–> Pulp Worker communication, update the `broker_url` parameter in the `[tasks]` section. For example, if the hostname to connect to is `someotherhost.com` uncomment `broker_url` and set it as follows:

```
broker_url: qpid://guest@someotherhost.com/
```

1.6.4 Qpid with Username and Password Authentication

The Pulp Server <–> Pulp Consumer Agent only support certificate based authentication, however the Pulp Server <–> Pulp Worker communication does allow for username and password based auth.

Pulp can authenticate using a username and password with Qpid using SASL. Refer to the Qpid docs on how to configure Qpid for SASL, but here are a few helpful pointers:

1. Ensure the Qpid machine has the `cyrus-sasl-plain` package installed. After installing it, restart Qpid to ensure it has taken effect.
2. Configure the username and password in the SASL database. Refer to Qpid docs for the specifics of this.
3. Ensure the qpidd user has read access to the SASL database.

After configuring the broker for SASL, then configure Pulp. This section explains how to configure Pulp to use a username and password configured in Qpid.

Assuming Qpid has the user `foo` and the password `bar` configured, enable Pulp to use them by uncommenting the `broker_url` setting in `[tasks]` and setting it as follows:

```
broker_url: qpid://foo:bar@localhost.com/
```

1.6.5 Qpid on a Non-Standard Port

To use Qpid with a non-standard port for Pulp Server <–> Pulp Consumer Agent communication, update the `url` parameter in the `[messaging]` section. For example, if Qpid is listening on port `9999`, uncomment `url` and set it as follows:

```
url: tcp://localhost:9999
```

The `/etc/pulp/consumer/consumer.conf` file on each Pulp Consumer also needs to be updated to correspond with this change. Refer to the inline documentation in `/etc/pulp/consumer/consumer.conf` to set the configuration correctly.

To use Qpid with a non-standard port for Pulp Sever <–> Pulp Worker communication, update the `broker_url` parameter in the `[tasks]` section. For example, if Qpid is listening on port `9999`, uncomment `broker_url` and set it as follows:

```
broker_url: qpid://guest@localhost:9999/
```

1.6.6 Qpid with SSL

SSL communication with Qpid is supported by both the Pulp Server <–> Pulp Consumer Agent and the Pulp Server <–> Pulp Worker components. To use Pulp with Qpid using SSL, you'll need to configure Qpid to accept SSL configuration. That configuration can be complex, so Pulp provides its own docs and utilities to make configuring the Qpid with SSL easier. You can find those items in the Qpid SSL Configuration Guide.

After configuring the broker with SSL and generating certificates, you should have a CA certificate, a client certificate, and a client certificate key. SSL with Qpid is by default on port 5671, and this example assumes that.

To configure Pulp Server <–> Pulp Consumer Agent communication to connect to Qpid using SSL, uncomment and set the following settings in the `[messaging]` section. The below configuration is an example; update `<host>` in the `url` setting and the absolute path of the `cacert` and `clientcert` settings for your environment accordingly.

```
[messaging]
url: ssl://<host>:5671
cacert: /etc/pki/pulp/qpid/ca.crt
clientcert: /etc/pki/pulp/qpid/client.crt
```

The Pulp Server <–> Pulp Consumer Agent SSL configuration requires the client keyfile and client certificate to be stored in the same file.

The `/etc/pulp/consumer/consumer.conf` file on each Pulp Consumer also needs to be updated to correspond with this change. Refer to the inline documentation in `/etc/pulp/consumer/consumer.conf` to set the configuration correctly.

To configure Pulp Server <–> Pulp Worker communication to connect to Qpid using SSL, uncomment and set the following settings in the `[messaging]` section. The below configuration is an example; update `<host>` in the `broker_url` setting and the absolute path of the `cacert`, `keyfile`, and `certfile` settings for your environment accordingly.

```
[tasks]
broker_url: qpid://<host>:5671/
celery_require_ssl: true
cacert: /etc/pki/pulp/qpid/ca.crt
keyfile: /etc/pki/pulp/qpid/client.crt
certfile: /etc/pki/pulp/qpid/client.crt
```

The Pulp Server <–> Pulp Worker communication allows the client key and client certificate to be stored in the same or different files. If the key and certificate are in the same file, set the same absolute path for both `keyfile` and `certfile`.

1.6.7 Using Pulp with RabbitMQ

Pulp Server <–> Pulp Consumer Agent and Pulp Server <–> Pulp Worker communication should both work with RabbitMQ, although it does not receive the same amount of testing by Pulp developers.

For a Pulp Server or Pulp Consumer Agent to use RabbitMQ, you'll need to install the `python-gofer-amqp` package on each Server or Consumer. This can be done by running:

```
sudo yum install python-gofer-amqp
```

Enable RabbitMQ support for Pulp Server <–> Pulp Consumer Agent communication by uncommenting and updating the `transport` setting in `[messaging]` to `rabbitmq`. Below is an example:

```
transport:  rabbitmq
```

The `/etc/pulp/consumer/consumer.conf` file on each Pulp Consumer also needs to be updated to correspond with this change. Refer to the inline documentation in `/etc/pulp/consumer/consumer.conf` to set the configuration correctly.

Enable RabbitMQ support for Pulp Server <–> Pulp Worker communication by uncommenting and updating the `broker_url` broker string to use the protocol handler `amqp://`. Below is an example:

```
broker_url:  amqp://guest:guest@localhost//
```

1.6.8 RabbitMQ with a Specific vhost

RabbitMQ supports an isolation feature called vhosts. These can be used by appending them to the broker string after the forward slash following the hostname. The default vhost in RabbitMQ is a forward slash, causing the broker string to sometimes be written with an additional slash. This form is for clarity as the the default vhost is assumed if none is specified.

Pulp Server <–> Pulp Consumer Agent communication through RabbitMQ on a vhost is not supported.

To enable Pulp Server <–> Pulp Worker communication through RabbitMQ on a vhost, uncomment and update the `broker_url` setting in `[tasks]` to include the vhost at the end. For example, if the vhost is 'foo' with the rest of the settings as defaults, the following example will work:

```
broker_url:  amqp://guest:guest@localhost/foo
```

1.6.9 RabbitMQ with SSL

RabbitMQ with SSL support is configured the same as it is with Qpid with the only difference being the adjustment to the `transport` setting in `[messaging]` and the protocol handler of `broker_url` in `[tasks]`. Both of these sections are contained on the Pulp Server in `/etc/pulp/server.conf`.

The `/etc/pulp/consumer/consumer.conf` file on each Pulp Consumer also needs to be updated to correspond with this change. Refer to the inline documentation in `/etc/pulp/consumer/consumer.conf` to set the configuration correctly.

1.7 Server

1.7.1 Conflicting Operations

Pulp, by its nature, is a highly concurrent application. Operations such as a repository sync or publish could conflict with each other if run against the same repository at the same time. For any such operation where it is important that a resource be effectively "locked", pulp will create a task object and put it into a queue. Pulp then guarantees that as workers take tasks off the queue, only one task will execute at a time for any given resource.

1.7.2 Failure and Recovery

For a recap of Pulp components and the work they are responsible for, read *components*.

- If a `pulp_worker` dies, the dispatched Pulp tasks destined for that worker (both the task currently being worked on and queued/related tasks) will not be processed. They will stall for at most six minutes before being cancelled. Status of the tasks is marked as cancelled after 5 minutes or in case the worker has been re-started, whichever action occurs first. Cancellation after 5 minutes is dependent on `pulp_celerybeat` service running. A monitoring component inside of `pulp_celerybeat` monitors all workers' heartbeats. If a worker does not heartbeat within five minutes, it is considered missing. This check occurs once a minute, causing a maximum delay of six minutes before a worker is considered missing and tasks cancelled by Pulp.

  A missing worker has all tasks destined for it cancelled, and no new work is assigned to the missing worker. This causes new Pulp operations dispatched to continue normally with the other available workers. If a worker with the same name is started again after being missing, it is added into the pool of workers as any worker starting up normally would.

- If `pulp_celerybeat` dies, and in case then new workers start, they won't be given work. If existing workers stop, Pulp will continue assigning them work. Once restarted, pulp_celerybeat will synchronize with the current state of all workers. Scheduled tasks will not run while pulp_celerybeat is down, but they will instead run when celerybeat is restarted.
- If `pulp_resource_manager` dies, the Pulp tasking system will halt. Once restarted it will resume.
- If the webserver dies the API will become unavailable until it is restored.

Note: From Pulp 2.6.0 and further, the /status/ url will show the currenct status of Pulp components. Read more about it here *status API*, which includes sample response output.

1.7.3 Backups

A complete backup of a pulp server includes:

- `/var/lib/pulp` a full copy of the filesystem
- `/etc/pulp` a full copy of the filesystem
- `/etc/pki/pulp` a full copy of the filesystem
- any custom Apache configuration
- *MongoDB*: a full backup of the database and configuration
- *Qpid* or *RabbitMQ*: a full backup of the durable queues and configuration

To do a complete restoration:

1. Install pulp and restore `/etc/pulp` and `/etc/pki/pulp`
2. Restore `/var/lib/pulp`
3. Restore the message broker service. If you cannot restore the state of the broker's durable queues, then first run `pulp-manage-db` against an empty database. Pulp will perform all initialization operations, including creation of required queues. Then drop the database before moving on.
4. Restore the database
5. Start all of the pulp services
6. Cancel any tasks that are not in a final state

1.7.4 Components

Pulp server has several components that can be restarted individually if the need arises. Each has a description below. See the *Services* section in this guide for more information on restarting services.

Apache

This component is responsible for the REST API.

The service name is `httpd`.

Workers

This component is responsible for performing asynchronous tasks, such as sync and publish.

The service name is `pulp_workers`.

Celery Beat

This is a singleton (there must only be one celery beat process per pulp deployment) that is responsible for queueing scheduled tasks. It also plays a role in monitoring the availability of workers.

The service name is `pulp_celerybeat`.

Resource Manager

This is a singleton (there must only be one of these worker processes per pulp deployment) celery worker that is responsible for assigning tasks to other workers based on which resource they need to reserve. When you see log messages about tasks that reserve and release resources, this is the worker that performs those tasks.

The service name is `pulp_resource_manager`.

1.7.5 Configuration

This section contains documentation on the configuration of the various Pulp Server components.

httpd

CRL Support

Pulp used to support Certificate Revocation Lists in versions up to and including 2.4.0. Starting with 2.4.1, the Pulp team decided not to carry their own M2Crypto build which had the patches necessary to perform CRL checks. Instead, users can configure httpd to do this using its SSLCARevocationFile and SSLCARevocationPath directives. See the mod-ssl documentation for more information.

Plugins

Many Pulp plugins support these settings in their config files. Rather than documenting these settings in each project repeatedly, the commonly accepted key-value pairs are documented below.

Importers

Most of Pulp's importers support these key-value settings in their config files:

`proxy_url`: A string in the form of scheme://host, where scheme is either `http` or `https`

`proxy_port`: An integer representing the port number to use when connecting to the proxy server

`proxy_username`: If provided, Pulp will attempt to use basic auth with the proxy server using this as the username

`proxy_password`: If provided, Pulp will attempt to use basic auth with the proxy server using this as the password

1.8 Authentication

1.8.1 Default

By default, pulp authenticates each request with a username and password against its own user database. Requests can also authenticate with a client-side SSL certificate that was provided by pulp's login feature.

1.8.2 Apache Preauthentication

If other forms of authentication are desired, authentication can be delegated to apache, which comes with a variety of authentication plugins that are well-documented and feature-rich. In order for users to then be authorized for any operation, they must have already been added to the Pulp user database using the `pulp-admin auth user` commands.

Once an apache authorization module is configured, pulp will read and trust the `REMOTE_USER` variable from apache.

Pulp's apache config file (`/etc/httpd/conf.d/pulp.conf`) contains an example of how to configure an apache auth module. The examples below demonstrate two different approaches.

LDAP Whole-API Example

To set up apache authentication for the entire REST API, modify the `<Files webservices.wsgi>` stanza in `/etc/httpd/conf.d/pulp.conf` to resemble the following:

```
<Files webservices.wsgi>
    # pass everything that isn't a Basic auth request through to Pulp
    SetEnvIfNoCase ^Authorization$ "Basic.*" USE_APACHE_AUTH=1
    Order allow,deny
    Allow from env=!USE_APACHE_AUTH
    Satisfy Any

    # configure basic auth
    AuthType basic
    AuthBasicProvider ldap
    AuthName "Pulp"
    AuthLDAPURL "ldaps://ad.example.com?sAMAccountName"
    AuthLDAPBindDN "cn=pulp,..."
    AuthLDAPBindPassword "adpassword"
    AuthLDAPRemoteUserAttribute sAMAccountName
    AuthzLDAPAuthoritative On
    Require valid-user

    # Standard Pulp REST API configuration goes here...
</Files>
```

Note that this *requires* LDAP authentication for the initial login, and *allows* either LDAP or Pulp certificate authentication on the entire API.

Basic Auth Login Example

Many deployments will only use a third-party authentication source for the login call, and then use pulp's certificate-based auth for successive calls.

You are responsible for ensuring that a user gets created in pulp prior to any login attempt. Pulp does not support auto-creation of users that exist in your external source.

Below is a "basic" example that works for demos, but a stronger mechanism is recommended.

```
<Location /pulp/api/v2/actions/login>
    AuthType Basic
    AuthName "Pulp Login"
    AuthUserFile /var/lib/pulp/.htaccess
    Require valid-user
</Location>
```

For this basic-auth example, the `.htaccess` file must then be created using the `htpasswd` command.

Note that this *requires* Apache authentication for the initial login, and also *requires* Pulp certificate authentication on the entire API.

1.8.3 LDAP

Deprecated since version 2.4: Please use apache's mod_authnz_ldap to provide preauthentication per instructions above.

Pulp supports LDAP authentication by configuring the `[ldap]` section in `server.conf`. An LDAP user who logs in for the first time will have a local account automatically created in the Pulp database.

The following options are supported:

- `enabled`: Boolean; controls whether or not LDAP authentication is enabled. Default: false.
- `uri`: URL of LDAP server. Default: `ldap://localhost`
- `base`: Location in the directory from which the LDAP search begins. Default: `dc=localhost`
- `tls`: Boolean; controls whether or not to use TLS security. Default: false.
- `default_role`: Role ID to assign LDAP users to by default. This role must first be created on the Pulp server. If `default_role` is not set or doesn't exist, LDAP users are given same default permissions as local users.
- `filter`: LDAP filter to limit the LDAP users who can authenticate to Pulp.

For example:

```
[ldap]
enabled = true
uri = ldap://ldap.example.com
base = ou=People,dc=example,dc=com
tls = true
default_role = ldap-users
filter = (gidNumber=200)
```

1.8.4 OAuth

Deprecated since version 2.4.0: OAuth support will be removed in a future release of Pulp. Please do not write new code that uses OAuth against Pulp, and please find a suitable replacement if you are already using it.

OAuth can be enabled by configuring the `[oauth]` section in `server.conf`. In order for a user or consumer to authenticate via OAuth, they must have already been added to the Pulp user database with the `pulp-admin auth user` commands. The following options are supported:

- `enabled`: Boolean; controls whether OAuth authentication is enabled. Default: false
- `oauth_key`: Key to enable OAuth style authentication. Required.

- `oauth_secret`: Shared secret that can be used for OAuth style authentication. Please be sure to choose a secret that is long enough for your desired level of security. Required.

For example:

```
[oauth]
enabled = true
oauth_key = ab3cd9j4ks73hf7g
oauth_secret = xyz4992k83j47x0bBoo8fue3yohneepo
```

Warning: Do not use the key or secret given in the above example. It is important that you use unique and secret values for these configuration items.

1.9 Admin Client

Contents:

1.9.1 Introduction

The admin client is used to remotely manage a Pulp server. This client is used to manage the operation of the server itself as well as trigger remote operations on registered consumers.

For more information on the client that runs on Pulp consumers, see the Consumer section of this guide.

The following sections describe some unique concepts that apply to the admin client.

Synchronous v. Asynchronous Commands

Commands run by the client execute in two different ways.

Many commands run synchronously. In these cases, the command is executed immediately on the server and the results are displayed before the client process exits. Examples of this behavior include logging in or displaying the list of repositories.

In certain cases, a request is sent to the server but the client does not wait for a response. There are several variations of this behavior:

- For long running operations, the request is sent to the server and the client immediately exits. The progress of the operation can be tracked using the commands in the `tasks` section of the client.
- Some operations cannot execute while a resource on the server is being used. If the resource is available, the operation will execute immediately and the result displayed in the client. If the operation is postponed because the resource is unavailable, the status of it can be tracked using the commands in the `tasks` section of the client.
- In certain circumstances, an operation may be outright rejected based on the state of a resource. For example, if a repository has a delete operation queued, any subsequent operation on the repository will be rejected.

For more information on the task commands, see the Tasks section of this guide.

Content Type Bundles

A portion of the admin client centers around standard Pulp functionality, such as user management or tasking related operations. However, Pulp's pluggable nature presents a challenge when it comes to type-specific operations. For example, the steps for synchronizing a repository will differ based on the type of content being synchronized.

To facilitate this, the client provides an *extension* mechanism. Extensions added by a content type *bundle* will customize the client with commands related to the types being supported. Typically, these commands will focus around managing repositories of a particular type, however there is no restriction to what commands an extension may add.

Type-specific sections will branch at the root of the client. For example, the following is a trimmed output of the client structure. Type-agnostic repository commands, such as the list of all repositories and group commands, are found under the `repo` section. Commands for managing RPM repositories and consumers are found under the `rpm` section and provided by the RPM extensions. Similarly, the commands for managing Puppet repositories are found under the `puppet` command:

```
$ pulp-admin --map
rpm: manage RPM-related content and features
  consumer: register, bind, and interact with rpm consumers
    bind:       binds a consumer to a repository
    history:    displays the history of operations on a consumer
    list:       lists summary of consumers registered to the Pulp
    ...
 repo: repository lifecycle commands
   create: creates a new repository
   delete: deletes a repository
   list:   lists repositories on the Pulp server
   ...
puppet: manage Puppet-related content and features
  repo: repository lifecycle commands
    create:  creates a new repository
    delete:  deletes a repository
    list:    lists repositories on the Pulp server
    ...
repo: list repositories and manage repo groups
  list: lists repositories on the Pulp server
  group: repository group commands
  ...
```

As new types are supported, additional root-level sections will be provided in their content type bundles.

1.9.2 Authentication

This guide covers basic authentication and authorization in the Pulp Platform.

Basic Authentication of Users

All pulp-admin commands accept username and password to capture authentication credentials.

```
$ pulp-admin --help
Usage: pulp-admin [options]

Options:
  -h, --help            show this help message and exit
  -u USERNAME, --username=USERNAME
                        username for the Pulp server; if used will bypass the
                        stored certificate and override a username specified
                        in ~/.pulp/admin.conf
  -p PASSWORD, --password=PASSWORD
                        password for the Pulp server; must be used with
                        --username. If used will bypass the stored certificate
                        and override a password specified in ~/.pulp/admin.conf
  --debug               enables debug logging
  --config=CONFIG       absolute path to the configuration file
  --map                 prints a map of the CLI sections and commands
```

Pulp Admin client allows the user to specify username and password credentials in the user's local admin.conf `~/.pulp/admin.conf`. Using the conf file avoids having to pass user credentials repeatedly using the command line. Also reading the password from a file that can only be read by certain users is more secure because it cannot be shown by listing the system processes.

```
# Add the following snippet to ``~/.pulp/admin.conf``

[auth]
username: admin
password: admin

# This enables the user to run pulp-admin commands without providing a username
# and password using the command line

$ pulp-admin repo list
+----------------------------------------------------------------------+
                              Repositories
+----------------------------------------------------------------------+
```

pulp-admin searches for a username and password to use in the following order:

- credentials specified from the command line.
- credentials set in the user's `~/.pulp/admin.conf`.

Pulp Server installation comes with one default user created with admin level privileges. Username and password for this user can be configured in `/etc/pulp/server.conf` at the time of installation.

Below is an example of basic authentication of users based on their username and password when running a pulp-admin command.

```
$ pulp-admin repo list
Enter password:
+----------------------------------------------------------------------+
                              Repositories
+----------------------------------------------------------------------+
```

Note that username and password are parameters to the `pulp-admin` command, not the sub-command, like `repo list` in this case. You can also pass the password parameter on the command line with `--password` argument, but this is not a recommended method. Users should use interactive password as a preferred method.

Rather than specifying the credentials on each call to pulp-admin, a user can log in to the Pulp server. Logging in stores a user credentials certificate at `~/.pulp/user-cert.pem`.

```
$ pulp-admin login -u admin
Enter password:
Successfully logged in. Session certificate will expire at Dec  6 21:47:33 2012
GMT.
```

Subsequent commands to pulp-admin will no longer require the username-password arguments and will instead use the user certificate. The user can be logged out by using the `pulp-admin logout` command.

```
$ pulp-admin logout
Session certificate successfully removed.
```

Layout of Auth Section

The root level `auth` section contains sub-sections to create and manage Pulp users, roles and their permissions for various resources.

```
$ pulp-admin auth
Usage: pulp-admin auth [SUB_SECTION, ..] COMMAND
Description: user, role and permission commands

Available Sections:
permission - manage granting, revoking and listing permissions for resources
role       - manage user roles
user       - manage users
```

Users

Users can be created to perform various administrative tasks on the Pulp Server. You can configure them with either admin level access or limited access to a few resources on the server.

```
$ pulp-admin auth user --help
Usage: pulp-admin user [SUB_SECTION, ..] COMMAND
Description: manage users

Available Commands:
create - creates a user
delete - deletes a user
list   - lists summary of users registered to the Pulp server
search - search items while optionally specifying sort, limit, skip, and requested fields
update - changes metadata of an existing user
```

Here is an example of creating and updating a user:

```
$ pulp-admin auth user create --login test-user
Enter password for user [test-user] :
Re-enter password for user [test-user]:
User [test-user] successfully created
```

If you intend to update the password for a user, you can use `-p` flag as shown in the example below to be prompted for a new password.

```
$ pulp-admin auth user update --login test-user --name "Test User" -p
Enter new password for user [test-user] :
Re-enter new password for user [test-user]:
User [test-user] successfully updated
```

You can also pass it on the command line with `--password` argument, but this method is just to provide a simpler way for scripting and is not recommended. Users should use interactive password update as a preferred method.

The `user list` command lists a summary of all users. It also accepts arguments to list all the details or specific fields for users.

```
$ pulp-admin auth user list --details
+----------------------------------------------------------------------+
                                 Users
+----------------------------------------------------------------------+

Login:  admin
Name:   admin
```

```
Roles:  super-users


Login:  test-user
Name:   test-user
Roles:
```

```
$ pulp-admin auth user list --fields roles
+----------------------------------------------------------------------+
                                 Users
+----------------------------------------------------------------------+

Login:  admin
Roles:  super-users


Login:  test-user
Roles:
```

Users can be removed from the Pulp server using the `user delete` command.

```
$ pulp-admin auth user delete --login test-user
User [test-user] successfully deleted
```

Users belonging to the `super-users` role can be deleted as well, as long as there is at least one such user remaining in the system.

```
$ pulp-admin auth user delete --login admin
The server indicated one or more values were incorrect. The server provided the
following error message:

The last superuser [admin] cannot be deleted

More information can be found in the client log file ~/.pulp/admin.log.
```

Permissions

Permissions to various resources can be accessed or manipulated using `pulp-admin auth permission` commands. There are 5 types of permissions - CREATE, READ, UPDATE, DELETE and EXECUTE. Permissions are granted and revoked from a resource which is essentially a REST API path.

Here are a few examples of accessing and manipulation permissions:

```
$ pulp-admin auth permission list --resource /
+----------------------------------------------------------------------+
                               Permissions for /
+----------------------------------------------------------------------+

Admin:  CREATE, READ, UPDATE, DELETE, EXECUTE
```

The following command will give permissions to create, read and update repositories to `test-user`.

```
$ pulp-admin auth permission grant --resource /v2/repositories/ --login test-user -o create -o updat
Permissions [/v2/repositories/ : ['CREATE', 'UPDATE', 'READ']] successfully granted
to user [test-user]
```

```
$ pulp-admin auth permission list --resource /v2/repositories/
+----------------------------------------------------------------------+
                 Permissions for /repositories
+----------------------------------------------------------------------+

Test-user:  CREATE, UPDATE, READ
```

The following command will revoke permissions to create and update repositories from `test-user`.

```
$ pulp-admin auth permission revoke --resource /v2/repositories/ --login test-user -o create -o updat
Permissions [/v2/repositories/ : ['CREATE', 'UPDATE']] successfully revoked from
user [test-user]
```

Note: The `/v2` prefix and the trailing `/` are always present in a resource name for permission commands.

Roles

In order to efficiently administer permissions, Pulp uses the notion of roles to enable an administrator to grant and revoke permission on a resource to a group of users instead of individually. The `pulp-admin auth role` command provides the ability to list the currently defined roles, create/delete roles, and manage user membership in a role. Pulp installation comes with a default `super-users` role with admin level privileges, and the default admin user belongs to this role.

The `role list` command is used to list the current roles.

```
$ pulp-admin auth role list
+----------------------------------------------------------------------+
                                Roles
+----------------------------------------------------------------------+

Id:     super-users
Users:  admin
```

A role can be created and deleted by specifying a role id.

```
$ pulp-admin auth role create --role-id consumer-admin
Role [consumer-admin] successfully created

$ pulp-admin auth role delete --role-id consumer-admin
Role [consumer-admin] successfully deleted
```

A user can be added and removed from a role using `role user add` and `role user remove` commands respectively. Note that both the user and the role should exist on the pulp server.

```
$ pulp-admin auth role user add --role-id super-users --login test-user
User [test-user] successfully added to role [super-users]

$ pulp-admin auth role user remove --role-id super-users --login test-user
User [test-user] successfully removed from role [super-users]
```

Permissions can be granted and revoked from roles just like users. In this case all the users belonging to the given role will inherit these permissions.

```
$ pulp-admin auth permission grant --resource /repositories --role-id test-role -o read
Permissions [/repositories : ['READ']] successfully granted to role [test-role]

$ pulp-admin auth permission revoke --resource /repositories --role-id test-role -o read
Permissions [/repositories : ['READ']] successfully revoked from role [test-role]
```

1.9.3 Consumer

There are several commands for managing consumers. Other type-specific commands, such as `bind`, are provided by type-specific extensions.

History

Pulp keeps a history of the operations that pertain to each consumer. The `history` command has several options for filtering and limiting its output.

```
$ pulp-admin consumer history --help
Command: history
Description: displays the history of operations on a consumer

Available Arguments:

  --consumer-id - (required) unique identifier; only alphanumeric, -, and _
                  allowed
  --event-type  - limits displayed history entries to the given type; supported
                  types: ("consumer_registered", "consumer_unregistered",
                  "repo_bound", "repo_unbound","content_unit_installed",
                  "content_unit_uninstalled", "unit_profile_changed",
                  "added_to_group","removed_from_group")
  --limit       - limits displayed history entries to the given amount (must be
                  greater than zero)
  --sort        - indicates the sort direction ("ascending" or "descending")
                  based on the entry's timestamp
  --start-date  - only return entries that occur on or after the given date in
                  iso8601 format (yyyy-mm-ddThh:mm:ssZ)
  --end-date    - only return entries that occur on or before the given date in
                  iso8601 format (yyyy-mm-ddThh:mm:ssZ)
```

The `history` command shows the most recent operations first.

```
$ pulp-admin consumer history --consumer-id=consumer1
+----------------------------------------------------------------------+
                        Consumer History [ consumer1 ]
+----------------------------------------------------------------------+

Consumer Id:  consumer1
Type:         repo_bound
Details:
  Distributor Id: puppet_distributor
  Repo Id:        repo1
Originator:   admin
Timestamp:    2013-01-22T16:07:52Z


Consumer Id:  consumer1
Type:         consumer_registered
Details:      None
Originator:   admin
Timestamp:    2013-01-22T15:09:58Z
```

List

This command retrieves a list of consumers. “Confirmed” bindings are those for which the agent on the remote consumer has performed a bind action. “Unconfirmed” bindings are waiting for that remote action to take place.

```
$ pulp-admin consumer list --help
Command: list
Description: lists a summary of consumers registered to the Pulp server

Available Arguments:

  --fields   - comma-separated list of consumer fields; Example:
               "id,display_name". If specified, only the given fields will be
               displayed.
  --bindings - if specified, the bindings information is displayed
  --details  - if specified, all of the consumer information is displayed

$ pulp-admin consumer list
+----------------------------------------------------------------------+
                               Consumers
+----------------------------------------------------------------------+

Id:             consumer1
Display Name:   Consumer 1
Description:    The first consumer.
Bindings:
  Confirmed:    repo1
  Unconfirmed:
Notes:
```

Search

For a more powerful way to find and list consumers, user the *Criteria* based `search` command.

```
$ pulp-admin consumer search --str-eq 'id=consumer1'
Capabilities:
Certificate:    -----BEGIN CERTIFICATE-----
                MIICETCB+gIBEDANBgkqhkiG9w0BAQUFADAUMRIwEAYDVQQDEwlsb2NhbGhvc3Qw
                HhcNMTMwMjA5MTQ1NzQ2WhcNMjMwMjA3MTQ1NzQ2WjAOMQwwCgYDVQQDEwNmb28w
                gZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAKvJ+5XzfArVxxrm4a16UoOA7F0x
                N++uip+GTqj/v9wG3ktHom+hlP0mlrzYOq731RS3zSBN8lkmCifRU+GKcyfG41/s
                k1LCGLR8N2AQin8XEeKjaloG4h9Q11ZLYWWklWSAbgL1HmzFg1FNiuEH7IPUR8MW
                PDExyOVOOHNjvhbTAgMBAAEwDQYJKoZIhvcNAQEFBQADggEBAIlpxab9wWOXczAZ
                bL+qdIf74bQ0yPug6wn1uWR6PamSYF6BuHzZIMHyq6n1ikx+RhBE2GGt0O01yR7Q
                Iq2zzOW80eJop5ct8pgoykVvMEG7xvF9qA2diJAi9npsA/dzvhaeyAFAcsCG60pU
                FKSOCjG8fXhyaU6o9oqX13dRo4ahW33ofYBnC/1Ck0L19ZDm5aA7zlu12j/ssMmI
                sDUZNzGg50lPvV58/1nalmxLWuNNScaWhOErPKowkfh8K7lcBfMVZs5H3VJQ6hW7
                iqjFyGBtASOdgw+Nc7yCkJSvUbkV+3uhKHNF+TG0uGGGPBcyOq+qkXEBeNwLKPbL
                taWnfe8= -----END CERTIFICATE-----
Description:    None
Display Name:   Consumer 1
Id:             consumer1
Notes:
```

Unregister

Registration must be initiated from `pulp-consumer`, but unregistering can be done from either end.

```
$ pulp-admin consumer unregister --help
Command: unregister
Description: unregisters a consumer

Available Arguments:

  --consumer-id - (required) unique identifier; only alphanumeric, -, and _
                  allowed

$ pulp-admin consumer unregister --consumer-id=consumer1
Consumer [ consumer1 ] successfully unregistered
```

Update

Basic attributes of consumers can be modified using the `update` command.

```
$ pulp-admin consumer update --help
Command: update
Description: changes metadata on an existing consumer

Available Arguments:

  --display-name - user-readable display name (may contain i18n characters)
  --description  - user-readable description (may contain i18n characters)
  --note         - adds/updates/deletes notes to programmatically identify the
                   resource; key-value pairs must be separated by an equal sign
                   (e.g. key=value); multiple notes can be changed by specifying
                   this option multiple times; notes are deleted by specifying
                   "" as the value
  --consumer-id  - (required) unique identifier; only alphanumeric, -, and _
                   allowed


$ pulp-admin consumer update --consumer-id=consumer1 --description='First consumer.'
Consumer [ consumer1 ] successfully updated
```

1.9.4 Events

Pulp has an event system that makes it easy for third party application to integrate and for users to stay informed via email about the Pulp server's activity. A specific set of operations inside the Pulp server produce reports about what they accomplished, and those reports are fed into an event framework. Users can then setup event listeners that listen for specific events, which then sends notifications to the user or an automated service. Notifier types include email, AMQP, and HTTP.

Listeners

Event listeners connect one or more event types with a notifier.

```
$ pulp-admin event listener
Usage: pulp-admin listener [SUB_SECTION, ..] COMMAND
```

```
Description: manage server-side event listeners

Available Sections:
  amqp  - manage amqp listeners
  email - manage email listeners
  http  - manage http listeners

Available Commands:
  delete - delete an event listener
  list   - list all of the event listeners in the system
```

From this section of the CLI, you can `list` and `delete` listeners, or drill down into type-specific sections to `create` and `update`. Examples for each type of notifier appear below.

Email

Event reports can be sent directly to an email address. The messages currently consists of a JSON-serialized representation of the actual event body. This meets a basic use case for having email notification with all of the available event data, but we intend to make the output more human-friendly in the future.

Note: Before attempting to setup email notifications, be sure to configure the "[email]" section of Pulp's settings file, `/etc/pulp/server.conf`

```
$ pulp-admin event listener email create --help
Command: create
Description: create a listener

Available Arguments:

  --event-type - (required) one of "repo.sync.start", "repo.sync.finish",
                 "repo.publish.start", "repo.publish.finish". May be specified
                 multiple times. To match all types, use value "*"
  --subject    - (required) text of the email's subject
  --addresses  - (required) this is a comma separated list of email addresses
                 that should receive these notifications. Do not include spaces.
```

To add an email notifier, you must specify what types of events to listen to, what the email subject should be, and who should receive the emails.

```
$ pulp-admin event listener email create --event-type="repo.sync.start" --subject="pulp notification"
Event listener successfully created

$ pulp-admin event listener list
Event Types:       repo.sync.start
Id:                5081a42ce19a00ea4300000e
Notifier Config:
  Addresses: someone@redhat.com, another@redhat.com
  Subject:   pulp notification
Notifier Type Id:  email
```

Using python's builtin testing MTA, the following message was captured after being sent by the above-configured listener.

```
$ python -m smtpd -n -c DebuggingServer localhost:1025
---------- MESSAGE FOLLOWS ----------
Content-Type: text/plain; charset="us-ascii"
```

```
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: pulp notification
From: no-repy@your.domain
To: someone@redhat.com
X-Peer: 127.0.0.1

{
  "call_report": {
    "task_group_id": "aaa8f2ec-964c-4d62-bba9-3191aad9c3ea",
    "exception": null,
    "task_id": "79be77a8-1a20-11e2-aeb9-1803731e94c4",
    "tags": [
      "pulp:repository:pulp2",
      "pulp:action:sync"
    ],
    "reasons": [],
    "start_time": "2012-10-19T19:09:16Z",
    "traceback": null,
    "schedule_id": null,
    "finish_time": null,
    "state": "running",
    "result": null,
    "progress": {},
    "principal_login": "admin",
    "response": "accepted"
  },
  "event_type": "repo.sync.start",
  "payload": {
    "repo_id": "pulp2"
  }
}
------------ END MESSAGE ------------
```

HTTP

Event reports can be sent via a POST call to any URL, and basic auth credentials may be supplied. The body of the HTTP request is a JSON-serialized version of the event report. Here is an example of creating an HTTP listener.

```
$ pulp-admin event listener http create --event-type=repo.sync.start --url=http://myserver.redhat.co
Event listener successfully created

[mhrivnak@redhrivnak pulp]$ pulp-admin event listener list
Event Types:       repo.sync.start
Id:                50bf51ffdd01fb5b9d000003
Notifier Config:
  URL: http://myserver.redhat.com
Notifier Type Id:  http
```

AMQP

AMQP is an industry standard for integrating separate systems, applications, or even components within an application through asynchronous messages. Pulp's event reports can be sent as the body of an AMQP message to a message broker, where it will be forwarded to any number of clients who subscribe to Pulp's topic exchange.

Pulp uses Apache Qpid as an AMQP broker and publishes its messages to a topic exchange. Even though AMQP is a widely-adopted standard protocol, there are several incompatible versions of it. For this reason, there is not another broker that can be used in place of Qpid.

Note: Before using an AMQP notifier, be sure to look in Pulp's server config file (`/etc/pulp/server.conf`) in the "[messaging]" section to configure your settings.

```
$ pulp-admin event listener amqp create --help
Command: create
Description: create a listener

Available Arguments:

  --event-type - (required) one of "repo.sync.start", "repo.sync.finish",
                 "repo.publish.start", "repo.publish.finish". May be specified
                 multiple times. To match all types, use value "*"
  --exchange   - optional name of an exchange that overrides the setting from
                 server.conf
```

Here you can also specify an exchange name. If you don't specify one, it will default to the value pulled from `/etc/pulp/server.conf` in the "[messaging]" section. If you don't set one there either, Pulp will default to "amq.topic", which is an exchange guaranteed to be available on any broker. Regardless of what name you choose (we suggest "pulp" as a reasonable choice), you do not need to create the exchange or take any action on the AMQP broker. Pulp will automatically create the exchange if it does not yet exist.

As for selecting event types, if you are unsure, we suggest going with "*" to select all of them. The client can choose which types of messages they want to subscribe to based on hierarchically matching against the event type (called a "subject" in AMQP). It is cheap and fast to send a message to a broker, making it convenient to fire and forget. Let the clients decide which subjects they care about. More about subject matching here.

This is an example of creating an AMQP event listener.

```
$ pulp-admin event listener amqp create --event-type='*' --exchange=pulp
Event listener successfully created

[mhrivnak@dhcp-230-147 pulp]$ pulp-admin event listener list
Event Types:       *
Id:                5092d9b3e19a00c58600000c
Notifier Config:
  Exchange: pulp
Notifier Type Id:  amqp
```

Event Types

These are the types of events that can be associated with listeners, and each description includes a partial list of the types of data that gets reported.

repo.publish.start

Fires when any repository starts a publish operation.

- start time
- repo_id
- user who initiated the sync
- task ID

repo.publish.finish

Fires when any repository finishes a publish operation.

- start time
- end time
- repo_id
- task ID
- success/failure
- number of items published
- errors

repo.sync.start

Fires when any repository starts a sync operation.

- start time
- repo_id
- user who initiated the sync
- task ID

repo.sync.finish

Fires when any repository finishes a sync operation.

- start time
- end time
- repo_id
- task ID
- success/failure
- number of items imported
- errors

1.9.5 Nodes

This guide covers admin client commands for managing *Pulp Nodes* in the Pulp Platform. For an overview, tips, and, troubleshooting, please visit the *Pulp Nodes Concepts Guide*.

Layout

The root level `node` section contains the following features.

```
$ pulp-admin node --help
Usage: pulp-admin [SUB_SECTION, ..] COMMAND
Description: pulp nodes related commands

Available Sections:
 repo - repository related commands
 sync - child node synchronization commands
```

```
Available Commands:
 activate   - activate a consumer as a child node
 bind       - bind a child node to a repository
 deactivate - deactivate a child node
 list       - list child nodes
 unbind     - removes the binding between a child node and a repository
```

Listing

The `node list` command may be used to list child *nodes*.

```
pulp-admin node list --help
Command: list
Description: list child nodes

Available Arguments:

 --fields   - comma-separated list of consumer fields; Example:
              "id,display_name". If specified, only the given fields will be
              displayed.
 --bindings - if specified, the bindings information is displayed
 --details  - if specified, all of the consumer information is displayed
```

Activation

A Pulp server that is registered as a consumer to another Pulp server can be designated as a *child node*. Once *activated* on the parent server, the consumer is recognized as a child node of the parent and can be managed using `node` commands.

To activate a consumer as a child node, use the `node activate` command. More information on *node-level* synchronization strategies can be found *here*.

```
$ pulp-admin node activate --help
Command: activate
Description: activate a consumer as a child node

Available Arguments:

 --consumer-id - (required) unique identifier; only alphanumeric, -, and _
                 allowed
 --strategy    - synchronization strategy (mirror|additive) default is additive
```

A child node may be deactivated using `node deactivate` command. Once deactivated, the node may no longer be managed using `node` commands.

```
$ pulp-admin node deactivate --help
Command: deactivate
Description: deactivate a child node

Available Arguments:

 --node-id - (required) unique identifier; only alphanumeric, -, and _ allowed
```

Note: Consumer (child node) un-registration will automatically deactivate the node. When a node is activated again, it will have the same repositories bound to it as it had before deactivation.

Repositories

The commands provided in the `node repo` section are used to perform *Nodes* specific management of existing repositories.

```
$ pulp-admin node repo --help
Usage: pulp-admin [SUB_SECTION, ..] COMMAND
Description: repository related commands

Available Commands:
 disable - disables binding to a repository by a child node
 enable  - enables binding to a repository by a child node
 list    - list node enabled repositories
 publish - publishing commands
```

Listing

A listing of *enabled repositories* may be obtained by using the `node repo list` command.

```
$ pulp-admin node repo list --help
Command: list
Description: list node enabled repositories

Available Arguments:

 --details - if specified, detailed configuration information is displayed for
             each repository
 --fields  - comma-separated list of repository fields; Example:
             "id,description,display_name,content_unit_counts". If
             specified, only the given fields will be displayed.
 --all, -a - if specified, information on all Pulp repositories, regardless of
             type, will be displayed
```

Enabling

A repository may be enabled using the `node repo enable` command. More information on *repository-level* synchronization strategies can be found *here*.

```
$ pulp-admin node repo enable --help
Command: enable
Description: enables binding to a repository by a child node

Available Arguments:

 --repo-id      - (required) unique identifier; only alphanumeric, -, and _
                  allowed
 --auto-publish - if "true", the nodes information will be automatically
                  published each time the repository is synchronized; defaults
                  to "true"
```

Warning: Using auto-publish causes the *Nodes* information to be published each time the repository is synchronized. This may increase the time it takes to perform the synchronization depending on the size of the repository.

Publishing

Manually publishing the *Nodes* data necessary for child node synchronization, can be triggered using the `node repo publish` command.

```
$ pulp-admin node repo publish --help
Command: publish
Description: publishing commands

Available Arguments:

 --repo-id - (required) unique identifier; only alphanumeric, -, and _ allowed
```

Note: Repositories MUST be published for child node synchronization to be successful.

Binding

The `node bind` command is used to associate a repository with a child node. This association determines which repositories may be synchronized to child nodes. The strategy specified here overrides the default strategy specified when the repository was enabled. More information on *repository-level* synchronization strategies can be found *here*.

```
$ pulp-admin node bind --help
Command: bind
Description: bind a child node to a repository

Available Arguments:

 --repo-id  - (required) unique identifier; only alphanumeric, -, and _ allowed
 --node-id  - (required) unique identifier; only alphanumeric, -, and _ allowed
 --strategy - synchronization strategy (mirror|additive) default is additive
```

The `node unbind` command may be used to remove the association between a child node and a repository. Once the association is removed, the specified repository can no longer be be synchronized to the child node.

```
$ pulp-admin node unbind --help
Command: unbind
Description: removes the binding between a child node and a repository

Available Arguments:

 --repo-id - (required) unique identifier; only alphanumeric, -, and _ allowed
 --node-id - (required) unique identifier; only alphanumeric, -, and _ allowed
```

Note: Only activated nodes and enabled repositories may be specified.

Synchronizing

The synchronization of child nodes may be triggered using the `node sync` commands. More information on node synchronization can be found *here*.

```
$ pulp-admin node sync --help
Usage: pulp-admin [SUB_SECTION, ..] COMMAND
Description: child node synchronization commands

Available Commands:
 run - triggers an immediate synchronization of a child node
```

An immediate synchronization can be triggered using the `node sync run` command.

```
$ pulp-admin node sync run --help
Command: run
Description: triggers an immediate synchronization of a child node

Available Arguments:

 --node-id       - (required) unique identifier; only alphanumeric, -, and _ allowed
 --max-downloads - maximum number of downloads permitted to run concurrently
 --max-speed     - maximum bandwidth used per download in bytes/sec
```

Warning: Make sure repositories have been published.

1.9.6 Repositories

This guide covers core features for managing repositories in the Pulp Platform. For more detail about how to work with repositories of a specific content type, please visit the user guide for that type. Examples will use "rpm" as the demo type when necessary, but they will be limited to generic features.

Layout

The root level `repo` section contains the following features. These features apply across all repositories, regardless of the specific types of content they will support.

```
$ pulp-admin repo --help
Usage: pulp-admin repo [SUB_SECTION, ..] COMMAND
Description: list repositories and manage repo groups

Available Sections:
  group - repository group commands
  tasks - list and cancel tasks related to a specific repository

Available Commands:
  list - lists repositories on the Pulp server
```

By comparison, many other features are implemented under a root-level section named for a content type. For example, the RPM repo section looks like this:

```
$ pulp-admin rpm repo --help
Usage: pulp-admin repo [SUB_SECTION, ..] COMMAND
Description: repository lifecycle commands
```

```
Available Sections:
  content - search the contents of a repository
  copy    - copies one or more content units between repositories
  export  - run or view the status of ISO export of a repository
  publish - run, schedule, or view the status of publish tasks
  remove  - remove copied or uploaded modules from a repository
  sync    - run, schedule, or view the status of sync tasks
  uploads - upload modules into a repository

Available Commands:
  create - creates a new repository
  delete - deletes a repository
  list   - lists repositories on the Pulp server
  search - searches for RPM repositories on the server
  update - changes metadata on an existing repository
```

The reason for putting repository commands in different places is that some features will be customized and augmented by plugins, such that their versions of common commands will only be applicable to their own content. Commands that can operate on all repositories go in the generic “repo” section.

Create, Update, Delete

To create a repository, the only required argument is a unique ID. Consult the help text for the create command of each particular repository type to see what other options are available.

```
$ pulp-admin rpm repo create --repo-id=foo
Successfully created repository [foo]
```

The `update` command takes similar arguments to the `create` command.

```
$ pulp-admin rpm repo update --repo-id=foo --display-name='Foo Repo'
Repository [foo] successfully updated
```

The new repository can be seen with its unique ID and display name.

```
$ pulp-admin rpm repo list
+----------------------------------------------------------------------+
                            RPM Repositories
+----------------------------------------------------------------------+

Id:                 foo
Display Name:       Foo Repo
Description:        None
Content Unit Count: 0
```

Deleting a repository is an asynchronous operation. In case other tasks are already in progress on this repository, the server will allow those tasks to complete before executing the deletion. The example below shows how to request deletion and then check the status of that task.

```
$ pulp-admin rpm repo delete --repo-id=foo
The request to delete repository [foo] has been received by the server. The
progress of the task can be viewed using the commands under "repo tasks"

$ pulp-admin repo tasks list --repo-id=foo
+----------------------------------------------------------------------+
                                 Tasks
+----------------------------------------------------------------------+

Operations:  delete
Resources:   foo (repository)
State:       Successful
Start Time:  2012-12-17T23:17:46Z
Finish Time: 2012-12-17T23:17:46Z
Result:      N/A
Task Id:     2d4fc3da-7ad7-448c-a9dd-78e79f71ef2f
```

List

This command lists all repositories in Pulp, regardless of their content type. To list and search repositories only of a particular type, go to that type's area of the CLI, such as `pulp-admin rpm repo list`.

```
$ pulp-admin repo list
+----------------------------------------------------------------------+
                              Repositories
+----------------------------------------------------------------------+

Id:                  pulp
Display Name:        Pulp
Description:         Pulp's stable repository
Content Unit Count: 39

Id:                  repo1
Display Name:        repo1
Description:         None
Content Unit Count: 0

Id:                  repo2
Display Name:        repo2
Description:         None
Content Unit Count: 0
```

Search

For more targeted results than the `list` command provides, you can use Pulp's *Criteria* search feature to search repositories. For example, to find a specific RPM repository that has id 'zoo':

```
pulp-admin rpm repo search --str-eq="id=zoo"
+----------------------------------------------------------------------+
                              Repositories
+----------------------------------------------------------------------+

Id:                   zoo
Display Name:         zoo-repo
Description:          None
Content Unit Counts:
  Erratum:           4
  Package Category: 1
  Package Group:     2
  Rpm:               32
Last Unit Added:      2014-11-14T13:02:47Z
Last Unit Removed:    None
Notes:
```

Content Search

Content units can be searched within a repository using Pulp's *Criteria* search feature. The layout of this command may vary based on the content type; for example, RPM support includes a separate command for each package type (rpm, srpm, etc.). Here is an example of searching for an rpm package by name. The `--repo-id` argument is required, and the `--match` argument applies a regular expression.

```
$ pulp-admin rpm repo content rpm --repo-id=pulp --match 'name=^python-w.+'
Arch:          noarch
Buildhost:     localhost
Checksum:      edfbe47f61a64c2196720e8ab1eb66c696303f89080fbe950444b9384bcfd2ee
Checksumtype:  sha256
Description:   web.py is a web framework for python that is as simple as it is
               powerful. web.py is in the public domain; you can use it for
               whatever purpose with absolutely no restrictions.
Epoch:         0
Filename:      python-webpy-0.32-9.fc17.noarch.rpm
License:       Public Domain and BSD
Name:          python-webpy
Provides:      [[u'python-webpy', u'EQ', [u'0', u'0.32', u'9.fc17']]]
Release:       9.fc17
Requires:      [[u'python(abi)', u'EQ', [u'0', u'2.7', None]]]
Vendor:
Version:       0.32
```

Copy Between Repositories

Content units can be copied from one repository to another using Pulp's *Criteria* search. For content units that involve an on-disk file (such as RPMs having a package stored on disk), the file is only stored once even if it is included in multiple Pulp repositories.

The following example assumes that the repository "foo" has some content units and that we want to copy all of them to the repository "bar".

```
$ pulp-admin rpm repo copy rpm --from-repo-id=foo --to-repo-id=bar
Progress on this task can be viewed using the commands under "repo tasks".

$ pulp-admin repo tasks list --repo-id=foo
+----------------------------------------------------------------------+
                                 Tasks
+----------------------------------------------------------------------+

Operations:  associate
Resources:   bar (repository), foo (repository)
State:       Successful
Start Time:  2012-12-17T23:27:12Z
Finish Time: 2012-12-17T23:27:13Z
Result:      N/A
Task Id:     8c3a6964-245f-4fe5-9d7c-8c6bac55cffb
```

The copy was successful. Here you can see that the repository "bar" now has the same number of content units as "foo".

```
$ pulp-admin rpm repo list
+----------------------------------------------------------------------+
                            RPM Repositories
+----------------------------------------------------------------------+

Id:                 foo
Display Name:       foo
Description:        None
Content Unit Count: 36

Id:                 bar
Display Name:       bar
Description:        None
Content Unit Count: 36
```

Groups

Repository Groups allow you to associate any number of repositories, even of varying content types, with a named group. Features that make use of repository groups are forthcoming in future releases of Pulp.

Here is an example of creating a repo group and adding members to it:

```
$ pulp-admin repo group create --group-id='group1' --description='misc. repos' --display-name='Group
Repository Group [group1] successfully created

$ pulp-admin repo group members add --group-id=group1 --str-eq='id=repo1'
Successfully added members to repository group [group1]
```

The `members add` command takes advantage of Pulp's *Criteria* search feature, so you can add many repositories at once. In this case, we provided a specific repository name. Let's look at the result of these two commands by listing the repository groups.

```
$ pulp-admin repo group list
+----------------------------------------------------------------------+
                          Repository Groups
+----------------------------------------------------------------------+

Id:           group1
Display Name: Group 1
Description:  misc. repos
Repo Ids:     repo1
Notes:
```

Notice that "repo1" shows up in the "Repo Ids" field.

Tasks

Some operations on repositories, such as `sync`, `publish`, and `delete`, may operate asynchronously. When you execute these operations, Pulp will give you a "task ID". You can use that task ID to check the status of the operation. From this section of the CLI, you can `cancel`, `list`, and get `details` about repository tasks.

```
$ pulp-admin repo tasks --help
Usage: pulp-admin tasks [SUB_SECTION, ..] COMMAND
Description: list and cancel tasks related to a specific repository

Available Commands:
  cancel  - cancel one or more tasks
  details - displays more detailed information about a specific task
  list    - lists tasks queued or running in the server
```

1.9.7 Orphaned Content Units

Introduction

Repositories are the container into which content is drawn into Pulp and by which Pulp serves content. However, under the hood, Pulp actually manages content separately from repositories. This allows Pulp to minimize disk space by never duplicating content that is shared between repositories (i.e. content units that appear in more than one repository).

The consequence of this approach is that when a repository is deleted from the Pulp server, the content associated with that repository is not. Content units that are no longer associated with any repositories are referred to as **orphaned content units** or simply **orphans**.

This page describes the management of orphaned content units.

Listing Orphaned Content Units

The **pulp-admin** command line client provides the `orphan` section and the `list` command to inspect the orphaned content units on your server:

```
$ pulp-admin orphan list --type=rpm --details
Arch:           noarch
Checksum:       7e9cad8b2cd436079fd524803ec7fa209a666ecdda05c6f9c8c5ee70cdea9ce6
Checksumtype:   sha256
Epoch:          0
Id:             a0079ca2-1d4f-4d01-8307-3f183f1843a6
Name:           tito
Release:        1.fc16
Version:        0.4.9

+----------------------------------------------------------------------+
                                Summary
+----------------------------------------------------------------------+

RPM:    1
Total:  1
```

You can filter the list by content type by using the `--type=<type>` flag.

You can use the `--details` flag to list the individual orphaned content units. Otherwise only the **Summary** section is displayed. This flag is not available when the content type is omitted, and will be ignored if specified.

Removing Orphaned Content Units

The **pulp-admin** command line client provides the `orphan` section and `remove` command to remove orphaned content units from your server.

It has three flags:

- `--type=<type>` to remove all the orphaned content units of a particular type
- `--id=<id>` to remove a particular orphaned content unit
- `--all` to remove all the orphaned content units on the server

```
$ pulp-admin orphan remove --all
Request accepted
```

```
check status of task e239ae4f-7fad-4004-bfb6-8e06f17d22ef with "pulp-admin tasks details"

$ pulp-admin tasks details --task-id e239ae4f-7fad-4004-bfb6-8e06f17d22ef
+----------------------------------------------------------------------+
                              Task Details
+----------------------------------------------------------------------+

Operations:
Resources:     orphans (content_unit)
State:         Successful
Start Time:    2012-12-09T03:26:51Z
Finish Time:   2012-12-09T03:26:51Z
Result:        N/A
Task Id:       e239ae4f-7fad-4004-bfb6-8e06f17d22ef
Progress:

$ pulp-admin orphan list
$
```

1.9.8 Inspecting the Server

Pulp as a Platform

The Pulp server has very little built in functionality. To provide the functionality that administrators rely on, Pulp loads a number of plugins that define `types` and `importers` and `distributors`. The former are definitions of categories of content units and the metadata that needs to be associated with them. The latter two are classes of plugins that allow Pulp to manage content units, of particular types, by bring them into the server and making them available from the server respectively.

This page shows how to query the server for each of these classes of plugins.

Content Unit Types

The **pulp-admin** command line client provides the `server` section and the `types` command to query the server about the content unit type definitions that have been loaded.

```
$ pulp-admin server types
+----------------------------------------------------------------------+
                        Supported Content Types
+----------------------------------------------------------------------+

Id:                distribution
Display Name:      Distribution
Description:       Kickstart trees and all accompanying files
Referenced Types:
Search Indexes:    id, family, variant, version, arch
Unit Key:          id, family, variant, version, arch

[snip]

Id:                rpm
Display Name:      RPM
Description:       RPM
Referenced Types: erratum
Search Indexes:    name, epoch, version, release, arch, filename, checksum,
                  checksumtype
Unit Key:         name, epoch, version, release, arch, checksumtype, checksum

[snip]

Id:               puppet_module
Display Name:     Puppet Module
Description:      Puppet Module
Referenced Types:
Search Indexes:   author, tag_list
Unit Key:         name, version, author
```

The output above shows a snippet of the types that are defined by the default plugins that have been developed with the Pulp server. Each type has the following fields:

- **Id**: this is a programmatic id that the server uses to identify the type
- **Display Name**: an optional, human-friendly, display name
- **Description**: an optional description of the type
- **Referenced Types**: a list of other types that may be referenced by a content unit of this type in some way
- **Search Indexes**: metadata fields that can potentially be used as search criteria
- **Unit Key**: a metadata field, or set of fields, that will uniquely identify a content unit of this type

Content Unit Importers

The **pulp-admin** command line client provides the `server` section and the `importers` command to query the server about the importer plugins that have been loaded.

```
$ pulp-admin server importers
+----------------------------------------------------------------------+
                          Supported Importers
+----------------------------------------------------------------------+

Id:           puppet_importer
Display Name: Puppet Importer
Types:        puppet_module

Id:           yum_importer
Display Name: Yum Importer
Types:        distribution, drpm, erratum, package_group, package_category, rpm,
              srpm
```

The output above shows the importers that have been developed along side the Pulp platform. Each importer has the following fields:

- **Id**: a programmatic id that the server uses to identify the importer
- **Display Name**: an optional, human-friendly, display name
- **Types**: a list of type ids that the importer can handle

Content Unit Distributors

The **pulp-admin** command line client provides the `server` section and the `distributors` command to query the server about the distributor plugins that have been loaded.

```
$ pulp-admin server distributors
+----------------------------------------------------------------------+
                       Supported Distributors
+----------------------------------------------------------------------+

Id:           yum_distributor
Display Name: Yum Distributor
Types:        rpm, srpm, drpm, erratum, distribution, package_category,
              package_group

Id:           export_distributor
Display Name: Export Distributor
Types:        rpm, srpm, drpm, erratum, distribution, package_category,
              package_group

Id:           puppet_distributor
Display Name: Puppet Distributor
Types:        puppet_module
```

The output above shows the distributors that have been developed along side the Pulp platform. Each distributor has the following fields:

- **Id**: a programmatic id that the server uses to identify the distributor
- **Display Name**: an optional, human-friendly, display name
- **Types**: a list of type ids that the distributor can handle

1.9.9 Tasks

Introduction

The Pulp server uses Celery to handle requests that may take longer than a HTTP request timeout to execute. Many of the commands from the **pulp-admin** command line client will return messages along the lines of

```
Request accepted

check status of task e239ae4f-7fad-4004-bfb6-8e06f17d22ef with "pulp-admin tasks details"
```

This means the server's REST API returned a 202 ACCEPTED response with the task information for the task that is handling the request.

This page details querying and managing these tasks.

Details

The **pulp-admin** command line client provides the `tasks` section and the `details` command to inspect the runtime details of a task, identified with the required `--task-id=<id>` flag.

```
$ pulp-admin tasks details --task-id e239ae4f-7fad-4004-bfb6-8e06f17d22ef
+----------------------------------------------------------------------+
                              Task Details
+----------------------------------------------------------------------+

Operations:
Resources:    orphans (content_unit)
State:        Successful
```

```
Start Time:   2012-12-09T03:26:51Z
Finish Time:  2012-12-09T03:26:51Z
Result:       N/A
Task Id:      e239ae4f-7fad-4004-bfb6-8e06f17d22ef
Progress:
```

In the output above there are several sections:

- *Operations*: a list of operations that are being performed by this task
- *Resources*: a list of resources that are being operated on
- *State*: the state of the task, such as: Waiting, Running, Successful or Error
- *Start Time*: the UTC time the task started
- *Finish Time*: the UTC time the task finished
- *Result*: the reported result of the task, if any
- *Task Id*: a unique identifier for the task (as a UUID)
- *Progress*: arbitrary progress information provided by the task, if any

Listing

To see all the tasks on the server at any given time, the **pulp-admin** command line client provides the `tasks` section and the `list` command.

It provides all the same information as the `details` command, but for every task that has been executed on the pulp server. The length of history depends on the settings described below.

In addition to tasks launched using pulp-admin or the API , `reaper` and `monthly` tasks appear in the list.

The `reaper` task is responsible for cleaning up the database on a regularly scheduled interval. The interval is configured with `reaper_interval` in `[data_reaping]` section of `/etc/pulp/server.conf`. The value can be whole or fraction of days. The length of time to keep documents for each collection is also configured in the same section. `archived_calls`, `task_status_history`, `consumer_history`, `repo_sync_history`, `repo_publish_history`, `repo_group_publish_history`, and `task_result_history` take values of whole or fraction of days to keep that type of history. This database cleanup is needed because these transactions can occur very frequently and as result the database can grow to an unreasonable size.

The `monthly` task is run every 30 days to clean up data referencing any repositories that no longer exist.

Canceling a Task

Tasks may be canceled before they are run (i.e. in the waiting state) or while they are running.

The **pulp-admin** command line client provides the `tasks` section and the `cancel` command to cancel a task identified by the required `--task-id` flag.

```
$ pulp-admin tasks cancel --task-id e0e0a250-eded-468f-9d97-0419a00b130f

$ pulp-admin tasks details --task-id e0e0a250-eded-468f-9d97-0419a00b130f
+----------------------------------------------------------------------+
                              Task Details
+----------------------------------------------------------------------+

Operations:   sync
Resources:    ff7-e6 (repository)
```

```
State:          Cancelled
Start Time:     2012-12-09T04:28:10Z
Finish Time:    2012-12-09T04:29:09Z
Result:         N/A
Task Id:        e0e0a250-eded-468f-9d97-0419a00b130f
Progress:
  Yum Importer:
    Comps:
      State: NOT_STARTED
    Content:
      Details:
        Delta Rpm:
          Items Left:  0
          Items Total: 0
          Num Error:   0
          Num Success: 0
          Size Left:   0
          Size Total:  0
        File:
          Items Left:  0
          Items Total: 0
          Num Error:   0
          Num Success: 0
          Size Left:   0
          Size Total:  0
        Rpm:
          Items Left:  6
          Items Total: 37
          Num Error:   0
          Num Success: 31
          Size Left:   112429996
          Size Total:  149958122
        Tree File:
          Items Left:  0
          Items Total: 0
          Num Error:   0
          Num Success: 0
          Size Left:   0
          Size Total:  0
      Error Details:
      Items Left:    0
      Items Total:   37
      Num Error:     0
      Num Success:   31
      Size Left:     112429996
      Size Total:    149958122
      State:         CANCELED
    Errata:
      State: NOT_STARTED
    Metadata:
      State: FINISHED
```

Note: It is possible for tasks to complete or experience an error before the task cancellation request is processed. In these instances, the task's final state might not be "canceled" even though a cancel was requested.

1.10 Consumer Client

Contents:

1.10.1 Introducing the Pulp Consumer Client

The Pulp consumer client, **pulp-consumer**, is a command line tool that allows administrators to register a consumer, bind that consumer to Pulp managed repositories, and install content onto the consumer from those bound repositories.

The client requires the Pulp Agent to be installed in order to communicate with the Pulp server. This is a daemon process that listens on an AMQP bus for messages from the server and uses the bus to send messages to the server.

This page gives a brief introduction to the consumer client and the agent.

Local v. Remote Commands

The **pulp-consumer** command line client is used to execute Pulp commands locally on the consumer machine. Pulp commands may be executed remotely, from the server, using the **pulp-admin** command line client. Details on the latter are covered in the **pulp-admin** documentation.

All commands are available locally. And some commands, namely `register` are only available locally.

Local Permissions

It is imperative that the **pulp-consumer** command line client be execute with enough permissions to write system-level configuration and credential files. In general, this will mean with `root` privileges. This is easiest with the `sudo` command.

```
$ sudo pulp-consumer ...
```

1.10.2 Registering

In order to register a consumer against the Pulp server, the **pulp-consumer** command line client provides the `register` command. A consumer must be registered against a server in order to benefit from the administrative functionality provided by Pulp.

```
$ sudo pulp-consumer register --consumer-id my-consumer
Enter password:
Consumer [my-consumer] successfully registered
```

Pre-Registration Authentication

The Pulp server's API is protected by basic authentication requirements. This means that the API is only accessible by defined users with the appropriate credentials.

Before a consumer is registered against a server, the server has no idea who (or what) the consumer is. In order to authenticate against the server's API to register the consumer, basic HTTP Auth credentials must be supplied along with the registration request.

Note: The **pulp-consumer** command must be executed with *root* permissions.

```
$ sudo pulp-consumer register --consumer-id my-consumer
Enter password:
Consumer [my-consumer] successfully registered
```

The `-u` and the `-p` flags supply the HTTP Basic Auth *username* and *password* respectively and must correspond to a user defined on the Pulp server. If the `-p` flag is not supplied, the command line client will ask for the password interactively.

Warning: Entering a password on the command line with the `-p` option is less secure than giving it interactively. The password will be visible to all users on the system for as long as the process is running by looking at the process list. It will also be stored in your bash history.

Post-Registration Authentication

Once a consumer is registered, a certificate is written into its PKI: `/etc/pki/pulp/consumer/consumer-cert.pem`

This certificate will automatically suffice for authentication against the server's API for all future operations until the consumer is unregistered.

It is worth noting that the **pulp-consumer** command line client should still be executed with *root* level permissions.

```
$ sudo pulp-consumer unregister
Consumer [my-consumer] successfully unregistered
```

1.10.3 Update

A consumer's attributes are stored on the server when it registers. Those attributes can be updated by using the `pulp-consumer update` command.

```
$ pulp-consumer update --help
Command: update
Description: changes metadata of this consumer

Available Arguments:

  --display-name - user-readable display name for the consumer
  --description  - user-readable description for the consumer
  --note         - adds/updates/deletes key-value pairs to programmatically
                   identify the repository; pairs must be separated by an equal
                   sign (e.g. key=value); multiple notes can be changed by
                   specifying this option multiple times; notes are deleted by
                   specifying "" as the value
```

Here is an example of updating the display name:

```
$ pulp-consumer update --display-name="Consumer 1"
Consumer [con1] successfully updated
```

1.10.4 Binding

Once a consumer is registered to a Pulp server, it can then *bind* to a repository. Binding allows content from a specific repository on the server to be installed on the consumer. Installation of content can be initiated either on the consumer or from the server. For example, in the case of RPM content, binding sets up a repository as a normal yum repository on the consumer. Packages can be installed with normal yum commands or by initiating an install through the Pulp server.

Bind

Binding to a server requires the consumer to first register. Then a bind command can be issued for a specific repository. Similar to the `pulp-admin` command, type-specific operations such as `bind` are located under a section bearing the type's name.

```
$ pulp-consumer rpm bind --repo-id=zoo
Bind tasks successfully created:

Task Id: 6e48ce85-60a0-4bf6-b2bb-9617eb7b3ef3

Task Id: 5bfe25f6-7325-4c29-b6e7-7e8df839570c
```

Looking at the consumer history, the first action in the list is a bind action to the specified repository.

```
$ pulp-consumer history --limit=1
+----------------------------------------------------------------------+
                        Consumer History [con1]
+----------------------------------------------------------------------+

Consumer Id:  con1
Type:         repo_bound
Details:
  Distributor Id: yum_distributor
  Repo Id:        zoo
Originator:   SYSTEM
Timestamp:    2013-01-02T22:01:17Z
```

Note: It may take a few moments for the bind to take effect. It happens asynchronously in the background, and we are working on a way to show positive confirmation of success from `pulp-consumer`.

Unbind

Unbinding is equally simple.

```
$ pulp-consumer rpm unbind --repo-id=zoo
Unbind tasks successfully created:

Task Id: 0c02d974-cf00-44b7-9b63-cdadfc9bfab7

Task Id: 947b03a6-6911-42c6-a8ce-3161bed08b15

Task Id: 358e9d1e-8531-4bbd-a8e8-ab64d596b345o
```

Looking at the history, it is clear that the consumer is no longer bound to the repository.

```
$ pulp-consumer history --limit=1
+----------------------------------------------------------------------+
                        Consumer History [con1]
+----------------------------------------------------------------------+

Consumer Id:  con1
```

```
Type:          repo_unbound
Details:
  Distributor Id: yum_distributor
  Repo Id:        zoo
Originator:    SYSTEM
Timestamp:     2013-01-02T22:09:47Z
```

In case a consumer is bound to a repository on a Pulp server that is no longer available, the `--force` option will make all of the local changes necessary to unbind from the remote repository without requiring the server to participate. When using this option, make sure a similar action is taken on the server so it does not continue to track a binding with the consumer.

1.10.5 History

Pulp Server keeps track of operations performed on its consumers in the consumer's history. The events tracked in the history and their respective event types are as follows:

- Consumer Registered - consumer_registered
- Consumer Unregistered - consumer_unregistered
- Repository Bound - repo_bound
- Repository Unbound - repo_unbound
- Content Unit Installed - content_unit_installed
- Content Unit Uninstalled - content_unit_uninstalled
- Unit Profile Changed - unit_profile_changed
- Added to a Consumer Group - added_to_group
- Removed from a Consumer Group - removed_from_group

Note that only operations that are triggered through Pulp are logged. If the consumer installs a content unit through another means (eg. rpm, yum etc.) an event will not be logged. The package profile, however, will eventually be sent to the server and will reflect any changes that have been made.

A consumer can view its own history using the *consumer history* command. A number of query arguments may be passed in to the *consumer history* command in order to refine the results. Here are a few examples of querying consumer history:

```
$ pulp-consumer history --limit 2 --sort ascending --event-type repo_bound
+----------------------------------------------------------------------+
                      Consumer History [consumer1]
+----------------------------------------------------------------------+

Consumer Id:  test-consumer
Type:         repo_bound
Details:
  Distributor Id: yum_distributor
  Repo Id:        test-repo1
Originator:   SYSTEM
Timestamp:    2013-01-17T05:43:36Z


Consumer Id:  test-consumer
Type:         repo_bound
Details:
  Distributor Id: yum_distributor
  Repo Id:        test-repo2
Originator:   SYSTEM
Timestamp:    2013-01-17T05:49:09Z
```

```
$ pulp-consumer history --start-date 2013-01-17T19:00:00Z --end-date 2013-01-17T21:00:00Z
+----------------------------------------------------------------------+
                      Consumer History [consumer1]
+----------------------------------------------------------------------+

Consumer Id:  consumer1
Type:         consumer_registered
Details:      None
Originator:   admin
Timestamp:    2013-01-17T19:14:49Z
```

1.10.6 Status

You can check registration status of a consumer using *pulp-consumer status* command. With this command you can get the information about which server the consumer is registered to and the consumer id used for the registration.

```
$ pulp-consumer status
This consumer is registered to the server [test-pulpserver.rdu] with the ID [f17-test-consumer].
```

When the consumer is not registered to a pulp server, it will simply display a message stating the same.

```
$ pulp-consumer status
This consumer is not currently registered.
```

1.10.7 Nodes

This guide covers consumer client commands for managing *Pulp Nodes* in the Pulp Platform. For an overview, tips, and, troubleshooting, please visit the *Pulp Nodes Concepts Guide*.

Layout

The root level `node` section contains the following features.

```
$ pulp-consumer node --help
Usage: pulp-consumer [SUB_SECTION, ..] COMMAND
Description: pulp nodes related commands

Available Commands:
 activate   - activate a consumer as a child node
 bind       - bind this node to a repository
 deactivate - deactivate a child node
 unbind     - remove the binding between this node and a repository
```

Activation

A Pulp server that is registered as a consumer to another Pulp server can be designated as a *child node*. Once *activated* on the parent server, the consumer is recognized as a child node of the parent and can be managed using `node` commands.

To activate a consumer as a child node, use the `node activate` command. More information on *node-level* synchronization strategies can be found *here*.

```
$ pulp-consumer node activate --help
Command: activate
Description: activate a consumer as a child node

Available Arguments:

 --strategy - synchronization strategy (mirror|additive) default is additive
```

A child node may be deactivated using the `node deactivate` command. Once deactivated, the node may no longer be managed using `node` commands.

```
$ pulp-consumer node deactivate --help
Command: deactivate
Description: deactivate a child node
```

Note: Consumer un-registration will automatically deactivate the node.

Binding

The `node bind` command is used to associate a child node with a repository on the parent. This association determines which repositories may be synchronized to child nodes. The strategy specified here overrides the default strategy specified when the repository was enabled. More information on *repository-level* synchronization strategies can be found *here*.

```
$ pulp-consumer node bind --help
Command: bind
Description: bind this node to a repository

Available Arguments:

 --repo-id  - (required) unique identifier; only alphanumeric, -, and _ allowed
 --strategy - synchronization strategy (mirror|additive) default is additive
```

The `node unbind` command may be used to remove the association between a child node and a repository. Once the association is removed, the specified repository can no longer be be synchronized to the child node.

```
$ pulp-consumer node unbind --help
Command: unbind
Description: remove the binding between this node and a repository

Available Arguments:

 --repo-id - (required) unique identifier; only alphanumeric, -, and _ allowed
```

Note: Only activated nodes and enabled repositories may be specified.

1.10.8 Repository Search

For each type of content supported, you can use Pulp's *Criteria* search feature to search repositories. For example, to find a specific repo by its id:

```
$ pulp-consumer rpm repos --str-eq="id=zoo"
+----------------------------------------------------------------------+
                              Repositories
+----------------------------------------------------------------------+

Id:                    zoo
Display Name:          zoo
Description:           None
Content Unit Counts:
Last Unit Added:       None
Last Unit Removed:     None
Notes:
```

1.11 Nodes

1.11.1 Overview

The *Pulp Nodes* concept describes the relationship between two Pulp servers for the purpose of sharing content. In this relationship, one is designated the *parent* and the other is designated the *child*. The *child* node consumes content that is provided by the *parent* node. It is important to understand that a child *node* is a complete and fully functional Pulp server capable of operating autonomously.

The following terms are used when discussing *Nodes*:

node A Pulp server that has the *Nodes* support installed and has a content sharing relationship to another Pulp server.

parent node A Pulp node that provides content to another Pulp server that has been registered as a *consumer* and activated as a node.

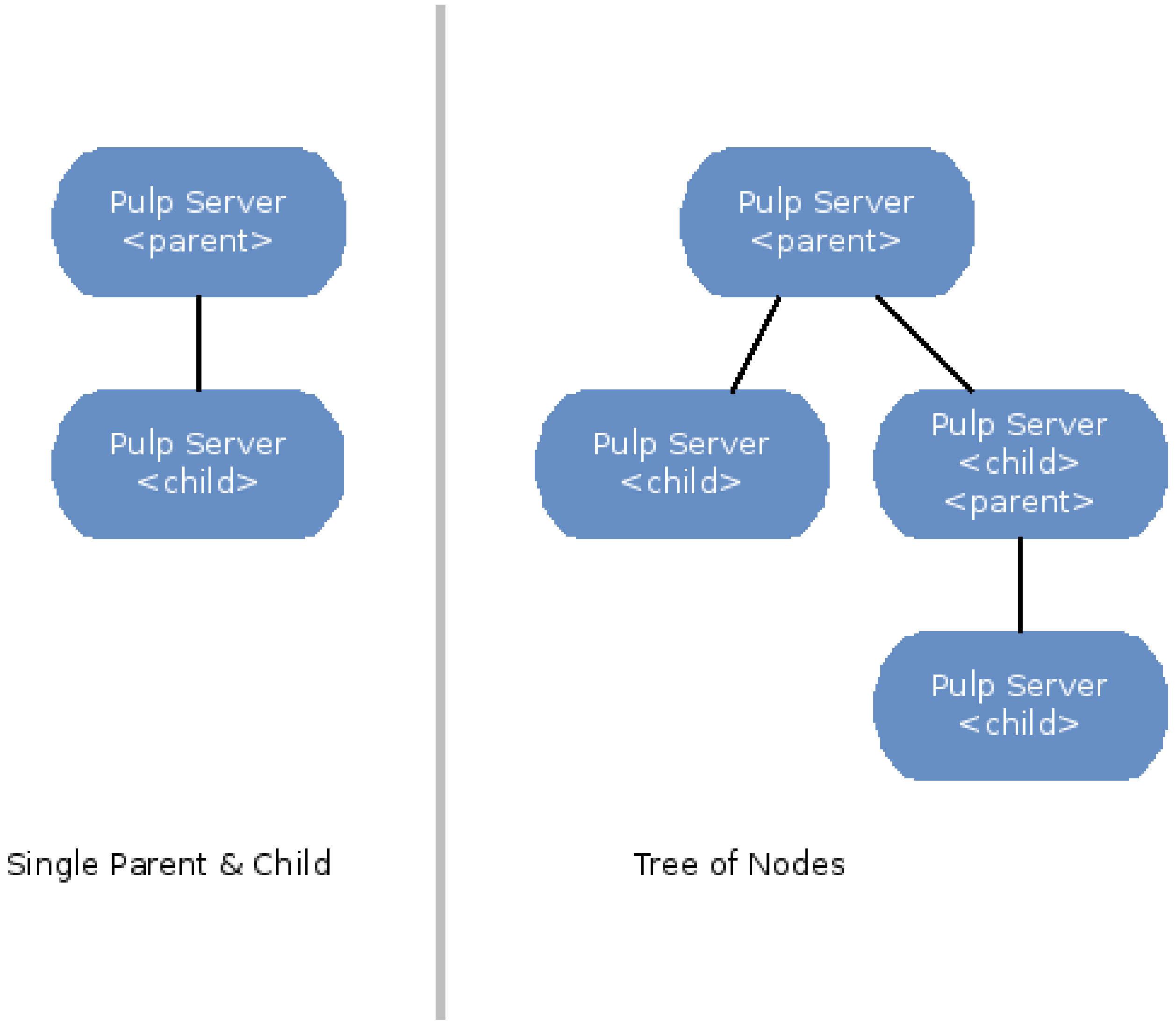
child node A Pulp node that consumes content from another Pulp server. The child node must be registered as a consumer to the parent and been activated as a child node.

node activation The designation of a registered consumer as a child node.

enabled repository A Pulp repository that has been *enabled* for *binding* by child nodes.

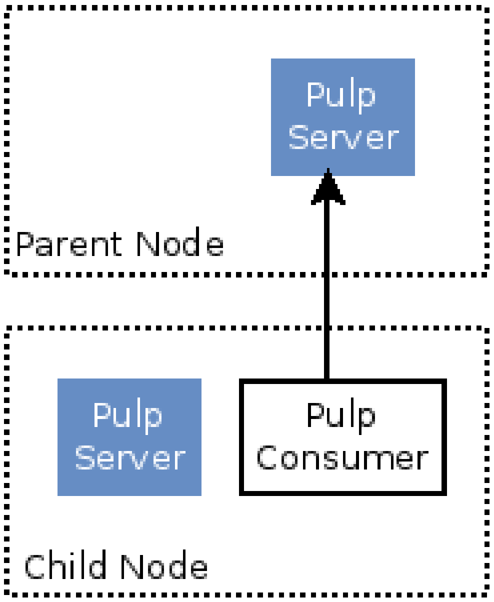
Node Topologies

Pulp nodes may be associated to form tree structures. Intermediate nodes may be designated as both a parent and a child node.



Node Anatomy

The anatomy of both parent and child nodes is simple. Parent nodes are Pulp servers that have the *Nodes* support installed. A Child node is a Pulp server with both the *Nodes* and *Consumer* support installed.



1.11.2 Authentication

The *child* node is authenticated to the *parent* node's REST API using Oauth. The connection is SSL but no client certificate is required. The *parent* node publishes content which is downloaded (as needed) by the *child* node using HTTPS requiring client certificate that has been signed by the Pulp CA.

During installation of the *Nodes* packages, an x.509 certificate is generated and signed using the Pulp CA (specified in server.conf) and stored in `/etc/pki/pulp/nodes/node.crt`. The certificate is generated using `/usr/bin/pulp-gen-nodes-certificate`, which is provided by the *Nodes* packages.

The node private key and certificate are sent to the *child* node at the beginning of each synchronization request to be used as the HTTPS credentials.

If the Pulp CA is changed after installation, administrators **must** regenerate the *Nodes* certificate using *pulp-gen-nodes-certificate*.

1.11.3 Installation

Since Pulp nodes *are* Pulp servers, the installation instructions for *Nodes* support assumes that the *server installation* has been completed. Next, follow the instructions below on each server depending on its intended role within the node topology.

Parent

To install *Nodes* parent support, follow the instructions below.

1. Install the node parent package group.

```
$ sudo yum install @pulp-nodes-parent
```

2. The communication between the child and parent nodes is secured using OAuth. The parent node must have OAuth enabled and configured. Please see *OAuth* for instructions on enabling and configuring OAuth.

3. Run `pulp-manage-db`.

4. Restart *server components*.

Child

To install *Nodes* child support, follow the instructions below.

1. Install the node child package group.

```
$ sudo yum install @pulp-nodes-child
```

2. The communication between the child and parent nodes is secured using OAuth. The child node must have OAuth enabled and configured. Please see *OAuth* for instructions on enabling and configuring OAuth.

```
[oauth]
enabled: true
oauth_key: Xohkaethaama5eki
oauth_secret: eePa7Bi3gohdir1pai2icohvaidai0io
```

Warning: Do not use the key or secret given in the above example. It is important that you use unique and secret values for these configuration items.

3. Edit `/etc/pulp/nodes.conf` and set the parent OAuth *key* and *secret* to match values found in `/etc/pulp/server.conf` on the parent node. The *user_id* must be updated as needed to match a user with administration privileges on the parent node.

Warning: The keys in `[parent_oauth]` are `key` and `secret`, whereas the values in the `[oauth]` section are `oauth_key` and `oauth_secret`. If you copy and paste from one to another without altering the names of the keys then the child node will not be able to communicate with the parent node.

```
[oauth]
user_id:  <EDIT: admin user on parent node>

[parent_oauth]
key:      <EDIT: matching value from parent /etc/pulp/server.conf>
secret:   <EDIT: matching value from parent /etc/pulp/server.conf>
user_id:  <admin user on parent node>
```

Example:

```
[oauth]
user_id: admin

[parent_oauth]
key: Xohkaethaama5eki
secret: eePa7Bi3gohdir1pai2icohvaidai0io
user_id: admin
```

4. Run `pulp-manage-db`.
5. Restart *server components*.
6. Restart `goferd`.

Admin Client Extensions

The admin extensions provide *Nodes* specific commands used to perform node administration which includes the following:

- Child node activation.
- Child node deactivation.
- List child nodes.
- Enable repositories for node binding.
- Disable repositories for node binding.
- List enabled repositories.
- Bind a child node to a repository.
- Unbind a child node from a repository.
- Initiate repository publishing of *Nodes* content.
- Initiate child node synchronization.

Install the *Nodes* admin client extensions.

```
$ sudo yum install pulp-nodes-admin-extensions
```

1.11.4 Enabling Repositories

In Pulp *Nodes*, there is a concept of enabling and disabling repositories for use with child nodes. Repositories must be *enabled* before being referenced in node bindings.

Repositories may be enabled using the admin client. See `node repo` commands for details.

```
$ pulp-admin node repo enable --repo-id <repo-id>
```

```
$ pulp-admin node repo disable --repo-id <repo-id>
```

Listing the enabled repositories can be done using the admin client. See: the `node repo list` for details.

```
$ pulp-admin node repo list
```

1.11.5 Repository Publishing

After a repository has been enabled, it MUST be published before synchronizing content to child nodes. Publishing a *Nodes* enabled repository generates the data necessary for repository content synchronization with child nodes. If auto-publishing is enabled, a normal repository synchronization will result in publishing this data as well.

The size of the published data varies based on the number of content units contained in the repository and the amount of metadata included in each unit. Each published unit consists of a copy of the metadata and a symlink to the actual file associated with the unit. The metadata is stored as gzip-compressed JSON.

The *Nodes* information can be manually published using the admin client. See: the `node repo publish` for details.

```
$ pulp-admin node repo publish --repo-id <repo-id>
```

1.11.6 Registration & Activation

Once the *Nodes* child support has been installed on a Pulp server, it can be registered to a parent server. This is accomplished using the Pulp consumer client. As mentioned, a child node is both a Pulp server and a consumer that is registered to the parent node.

On the child Pulp server:

1. Edit the `/etc/pulp/consumer/consumer.conf` file and set the `host` property the to the hostname or IP address of the Pulp server to be use as the child node's parent.

```
[server]
host = <parent hostname or IP>
```

2. Register to the parent server as a consumer. This command will prompt for a password.

```
$ sudo pulp-consumer -u <user> register --consumer-id <id>
```

3. Active the Pulp server as a child node. See: the `node activate` command for details.

```
$ sudo pulp-consumer node activate
```

1.11.7 Binding To Repositories

The selection of content to be replicated to child nodes is defined by repository bindings. Using the *Nodes* `bind` and `unbind` commands, users create an association between the child node and *Nodes* enabled repositories.

Examples:

```
$ pulp-admin node bind --node-id <node-id> --repo-id <repo-id>
```

```
$ pulp-consumer node bind --repo-id <repo-id>
```

1.11.8 Child Synchronization

A child node's repositories and their content can be synchronized with the parent. Technically, this action is seen by the parent as a content update on one of it's consumers. But, for most users, the term synchronization is easier to grasp. During this process, the following objects and properties are replicated to the child node:

- Repositories
- description
- notes
- Distributors
- configuration (includes certificates and other credentials)
- Content Units
- metadata
- associated files (bits)

Strategies

During child node synchronization, named strategies determine how the synchronization is performed and what the desired effect will be. Strategies are incorporated at two levels during node synchronization.

The first is the *node* level strategy which determines how the collection of repository objects are synchronized. Depending on the selected strategy, repositories are created, updated or deleted to match the set of repositories to which the node is associated through bindings.

The second is the *repository* level strategy which determines how each repository's content is synchronized. Depending on the selected strategy, content units are created, updated or deleted to match the content contained in the repository on the parent.

Current, there are two supported strategies.

additive Results in objects present in the parent but not in the child being created or updated as necessary. This strategy should be used when objects created locally in the child should be preserved.

mirror Results in objects present in the parent but not in the child being created or updated as necessary. Any objects present in the child that do not exist in the parent are removed. This strategy should be used when the desired effect of synchronization is for the child repositories to be an exact mirror of those on the parent.

The *node* level strategy is specified during node activation. Once activated, the strategy may be changed by performing a node deactivation followed by node activation specifying the desired strategy.

The *repository* level strategy is specified during node binding to a repository. Once bound, the strategy may be changed by performing an unbind followed by a bind to the repository specifying the desired strategy.

Note: The `additive` strategy is the default.

Running

The synchronization of a child node can be initiated using the admin client. This results in a request being sent to the agent on the child node which performs the update. A *partial* synchronization can be initiated by doing a regular repository synchronization on the child node. This will synchronize only the content of the repository.

The synchronization can be requested using the admin client. See: the `node sync` command.

```
$ pulp-admin node sync run --node-id <node-id>
```

Scheduling

Synchronization of a particular child can be scheduled with an optional recurrence. The format for describing the schedule follows the Pulp standard for *date and time intervals*. All commands related to the creation, removal, and listing of node sync schedules can be found under the `node sync schedules` command.

1.11.9 Quick Start

This assumes there are two Pulp servers up and running. The following steps could generally be followed to get a basic *Nodes* parent and child setup going. To simplify the writeup, it's assumed that the parent server's hostname is `parent.redhat.com` and it has a repository named `pulp-goodness` that we want to share with our child.

On The Parent

On the Pulp server to be used as the parent node:

1. Install the pulp-nodes-parent package group.

```
$ sudo yum install @pulp-nodes-parent
$ sudo service httpd restart
```

2. Enable the `pulp-goodness` repository.

```
$ pulp-admin node repo enable --repo-id pulp-goodness
```

3. Publish the `pulp-goodness` repository.

```
$ pulp-admin node repo publish --repo-id pulp-goodness
```

On The Child

On the Pulp server to be used as the child node:

1. Install the pulp-nodes-child package group.

```
$ sudo yum install @pulp-nodes-child
```

2. Edit `/etc/pulp/nodes.conf` and set the parent OAuth *key* and *secret* to match values found in `/etc/pulp/server.conf` on the parent node.

```
[parent_oauth]
key:    <matching value from parent /etc/pulp/server.conf>
secret: <matching value from parent /etc/pulp/server.conf>
```

3. Edit `/etc/pulp/consumer/consumer.conf` and change:

```
[server]
host = parent.redhat.com
```

4. Restart Apache. For upstart:

```
    $ sudo service httpd restart
```

   For systemd:

```
    $sudo systemctl restart httpd
```

5. Restart the Pulp agent. For upstart:

```
    $ sudo service goferd restart
```

   For systemd:

```
    $ sudo systemctl restart goferd
```

6. Register as a consumer. This command will prompt for a password.

```
$ pulp-consumer register --consumer-id child-1
```

7. Activate the node.

```
$ pulp-consumer node activate
```

8. Bind to the `pulp-goodness` repository.

```
$ pulp-consumer node bind --repo-id pulp-goodness
```

Anywhere Using Admin Client

1. Synchronize the child.

```
$ pulp-admin node sync run --node-id child-1
```

1.11.10 Tips & Troubleshooting

1. Make sure httpd was restarted after installing *Nodes* packages on both the parent and child.
2. Make sure goferd was restarted after installing *Nodes* packages on the child.
3. Make sure that *Nodes* enabled repositories have been published.
4. Make sure that ALL plugins installed on the parent are installed on the child.

1.12 Content Sources

Pulp supports a generic concept of *Alternate Content Sources* that is independent of Importers and Repositories. Each content source is a potential alternate provider of files that are associated with content units in Pulp. Pulp maintains a catalog of the content provided by each source which is periodically refreshed using a *Cataloger* server-side plugin. During a refresh, the cataloger queries the content source using this information to update the catalog. The next time Pulp needs to download a file associated with a content unit, it searches the catalog for alternate sources based on the source's *priority*. Each alternate source is tried in *priority* order. If the file cannot be successfully downloaded from one of the alternate sources, it is finally downloaded from the original (primary) source.

1.12.1 Defining A Content Source

Content sources are defined in `/etc/pulp/content/sources/conf.d`. Each file with a .conf suffix may contain one or more sections. Each section defines a content source.

The [section] defines the content source ID. The following properties are supported:

- **enabled <bool>** The content source is enabled. Disabled sources are ignored.
- **name <str>** The content source display name.
- **type <str>** The type of content source. Must correspond to the ID of a cataloger plugin ID.
- **priority <int>** The *optional* source priority used when downloading content. (0 is highest and the default).
- **expires <str>** How long until cataloged information expires. The default unit is seconds but and optional suffix can (and should) be used. Supported suffixes: (s=seconds, m=minutes, h=hours, d=days)
- **base_url <str>** The URL used to fetch info used to refresh the catalog.
- **paths <str>** An *optional* list of URL relative paths. Delimited by space or newline.
- **max_concurrent <int>** An *optional* limit to the number of concurrent downloads.
- **max_speed <int>** An *optional* limit to the bandwidth used during downloads.
- **ssl_ca_cert <str>** An *optional* SSL CA certificate (absolute path).
- **ssl_validation <bool>** An *optional* flag to validate the server SSL certificate using the CA.
- **ssl_client_cert <str>** An *optional* SSL client certificate (absolute path).
- **ssl_client_key <str>** An *optional* SSL client key (absolute path).
- **proxy_url <str>** An *optional* URL for a proxy.
- **proxy_port <short>** An *optional* proxy port#.

- **proxy_username <str>** An *optional* proxy userid.
- **proxy_password <str>** An *optional* proxy password.

Example:

```
[content-world]
enabled: 1
priority: 0
expires: 3d
name: Content World
type: yum
base_url: http://content-world/content/
paths: f18/x86_64/os/ \
       f18/i386/os/ \
       f19/x86_64/os \
       f19/i386/os
max_concurrent: 10
max_speed: 1000
ssl_ca_cert: /etc/pki/tls/certs/content-world.ca
ssl_client_key: /etc/pki/tls/private/content-world.key
ssl_client_cert: /etc/pki/tls/certs/content-world.crt
```

1.12.2 Recipes

The pulp-admin client can be use to list all defined content sources as follows:

```
$ pulp-admin content sources list

+----------------------------------------------------------------------+
                            Content Sources
+----------------------------------------------------------------------+

Base URL:       http://content-world/content/
Enabled:        1
Expires:        3d
Max Concurrent: 2
Name:           Content World
Paths:          f18/x86_64/os/ f18/i386/os/ f19/x86_64/os f19/i386/os
Priority:       0
Source Id:      content-world
SSL Validation: true
Type:           yum
```

The pulp-admin client can be used to delete entries contributed by specific content sources as follows:

```
$ pulp-admin content catalog delete -s content-world
Successfully deleted [10] catalog entries.
```

The pulp-admin client can be used to refresh content catalog using all content sources:

```
$ pulp-admin content sources refresh
+----------------------------------------------------------------------+
                        Refresh Content Sources
+----------------------------------------------------------------------+

This command may be exited via ctrl+c without affecting the request.
```

```
Refreshing content sources
[==================================================] 100%
2 of 2 items
... completed


Task Succeeded
```

The pulp-admin client can be used to refresh content catalog using a specific content source:

```
$ pulp-admin content sources refresh --source-id content-zoo
+----------------------------------------------------------------------+
                        Refresh Content Sources
+----------------------------------------------------------------------+

This command may be exited via ctrl+c without affecting the request.


Refreshing content sources
[|]
... completed


Task Succeeded
```

1.13 General Reference

1.13.1 Resource IDs

All resource ID values must contain only letters, numbers, underscores, periods, and hyphens.

1.13.2 Date and Time Units

Dates and times, including intervals, are specified using the ISO8601 format. While it is useful to be familiar with the full specification, a summary of the common usage patterns can be found below.

Dates are written in the format YYYY-MM-DD. For example, May 28th, 2005 is represented as `2005-05-28`.

Times are specified as HH:MM and should always be expressed in UTC. To mark the time as UTC, a `Z` is appended to the end of the time designation. For example, 1:45 is represented as `01:45Z`.

These two pieces can be combined with a capital T as the delimiter. Using the above two examples, the full date expression is `2005-05-28T01:45Z`.

Intervals

Some situations, such as scheduling a recurring operation, call for an interval to be specified. The general syntax for an interval is to begin with a capital P (used to designate the start of the interval, historically called a "period") followed by the quantity of the interval and the units. For example, an interval of three days is expressed as `P3D`.

The following are the commonly used interval units (more are supported, these are just a subset):

- `D` - Days
- `W` - Weeks

- `M` - Months
- `Y` - Years

Additionally, the following "time"-based intervals are supported:

- `S` - Seconds (likely too frequent to use in most cases)
- `M` - Minutes
- `H` - Hours

Time based intervals require a capital T prior to their definition. For example, an interval of every 6 hours is expressed as `PT6H`.

In many cases, Pulp allows schedules to be created with a start time in the past. The server will apply the interval until it determines the next valid timeframe in the future. Thus an interval defined as starting on January 1st and executing every month, if added in mid-April, will execute for its first time on May 1st.

The interval is appended after the start time, separated by a front slash. For example, an interval of one day starting on October 10th is represented as `2011-10-10/P1D`.

Recurrence

The ISO8601 format also includes the ability to specify the number of times an interval based operation should perform. The recurrence is defined as a capital R and the number of times it should execute. This value is prefixed in front of the rest of the expression and separated by a front slash. For example, running an operation every hour for 5 runs is expressed as `R5/PT1H`.

A recurrence expression is only valid when an interval is included as well.

Examples

Putting it all together, below are some examples and their real world explanations:

`PT1H` Every hour; in most cases Pulp will default the start time if unspecified to the time when the server received the request.

`P2W` Every other week starting immediately.

`2012-01-01T00:00Z/P1M` The first of every month at midnight, starting at January 1st.

`R7/P1D` Every day for one week (techincally, for 7 days).

`R5/2007-07-05T23:16Z/P1D` Starting on July 5th at 11:16pm UTC, run at that time every day for the next 5 days.

1.13.3 Criteria

Pulp offers a standard search interface across all resource types. This interface is used in two different ways:

- As a query syntax to scope the resources returned, data retrieved for each resource, and pagination constructs such as limits and skips.
- As a matching syntax, used when indicating resources that should be included in an operation.

In other words, the same parameters used to search for specific resources can then be fed into an operation that affects matching resources. For example, a query can be passed to the repository search to determine which repositories match. The same query can then be passed into the repository group membership command to add all matching repositories to a particular group.

Where applicable, the client supports a number of arguments for describing the desired query. More information on each argument can be found using the `--help` argument on the command in question.

An example of this functionality is the `pulp-admin rpm repo search` command. The output of the usage text for that command is as follows:

```
Command: search
Description: searches for RPM repositories on the server

Available Arguments:

 --filters - filters provided as JSON in mongo syntax. This will override any
             options specified from the 'Filters' section below.
 --limit   - max number of items to return
 --skip    - number of items to skip
 --sort    - field name, a comma, and either the word "ascending" or
             "descending". The comma and direction are optional, and the
             direction defaults to ascending. Do not put a space before or
             after the comma. For multiple fields, use this option multiple
             times. Each one will be applied in the order supplied.
 --fields  - comma-separated list of resource fields. Example:
             "id,display_name". Do not include spaces. Default is all fields.

Filters
 These are basic filtering options that will be AND'd together. These will be
 ignored if --filters= is specified. Any option may be specified multiple
 times. The value for each option should be a field name and value to match
 against, specified as "name=value". Example: $ pulp-admin repo search
 --str-eq="id=<repo_id>"

 --str-eq - match where a named attribute equals a string value exactly.
 --int-eq - match where a named attribute equals an int value exactly.
 --match  - for a named attribute, match a regular expression using the mongo
            regex engine.
 --in     - for a named attribute, match where value is in the provided list of
            values, expressed as one row of CSV
 --not    - field and expression to omit when determining units for inclusion
 --gt     - matches resources whose value for the specified field is greater
            than the given value
 --gte    - matches resources whose value for the specified field is greater
            than or equal to the given value
 --lt     - matches resources whose value for the specified field is less than
            the given value
 --lte    - matches resources whose value for the specified field is less than
            or equal to the given value
```

Unit Association Criteria

The criteria when dealing with units in a repository is slightly different from the standard model. The metadata about the unit itself is split apart from the metadata about when and how it was associated to the repository. This split occurs in the filters, sort, and fields sections.

The primary differences are as follows:

- There are two added search criteria, `--after` and `--before`. These fields apply to the point at which the unit was first added to the repository. The values for these fields are expressed as an *iso8601* timestamp.
- A `--details` flag is provided when searching for units within a repository. If specified, information about the association between the unit and the repository will be displayed in addition to the metadata about the unit

itself.

1.13.4 Client Argument Boolean Values

Depending on the situation, booleans are expressed in one of two ways in the client:

Flags are used to indicate the behavior of the immediate command:

```
$ pulp-admin repo list --details
```

Boolean values are specified for cases where the value is saved:

```
$ pulp-admin rpm repo create --repo-id foo --verify-feed-ssl true
$ pulp-admin rpm repo create --repo-id foo --verify-feed-ssl false
```

1.13.5 Services

The platform includes several services which can be managed using standard system tools such as *upstart* and *systemd*.

For further information:

- For upstart: `$ man service`. Pulp init.d scripts support the following actions:
- start
- restart
- status
- stop
- For systemd: `$ man systemctl`

1.14 Glossary

agent A daemon running on a consumer. The agent provides a command & control API which is used by the Pulp server to initiate content changes on the consumer. It also sends scheduled reports concerning consumer status and installed content profiles to the Pulp server.

binding An association between a *consumer* and a *repository distributor* for the purpose of installing *content units* on the specified consumer.

bundle Term used to denote the collection of server, client, and agent components to provide support for a particular set of content types. For example, support for handling RPMs and errata is provided by the RPM bundle whereas support for Puppet modules is provided by the Puppet bundle.

CLI Command Line Interface

consumer A managed system that is the consumer of content. Consumption refers to the installation of software contained within a *repository* and published by an associated *distributor*.

content unit An individual piece of content managed by the Pulp server. A unit does not necessarily correspond to a file. It is possible that a content unit is defined as the aggregation of other content units as a grouping mechanism.

distributor Server-side plugin that takes content from a repository and publishes it for consumption. The process by which a distributor publishes content varies based on the desired approach of the distributor. A repository may have more than one distributor associated with it at a given time.

extension Client-side component that augments the CLI with new functionality. While all functionality in the client is provided through extensions, this term is typically used to refer to content type specific extensions provided by a content type bundle.

importer Server-side plugin that provides support for synchronizing content from an external source and importing that content into the Pulp server. Importers are added to repositories to define the supported functionality of that repository.

iso8601 ISO Date format that is able to specify an optional number of recurrences, an optional start time, and a time interval. More information can be found *in the conventions section of this guide*.

node A Pulp node is a Pulp server that has either a parent or child relationship to another Pulp server. Parent nodes provide content to child nodes. Child nodes consume content from a parent node as registered *consumers*.

registration The association of a *consumer* to a Pulp server. Once registered, a consumer is added to Pulp's inventory and may be *bound* to Pulp provided *repositories*. *Content* installs, updates and uninstalls may be initiated from the Pulp server on consumers running the Pulp *agent*.

repository A collection of content units. A repository's supported types is dictated by the configured *importer*. A repository may have multiple *distributors* associated which are used to publish its content to multiple destinations, formats, or protocols.

unit profile A list of *content unit* installed on a *consumer*. The structure and content of each item in the profile varies based on the unit type.

1.15 Troubleshooting

1.15.1 Logging

Starting with 2.4.0, Pulp uses syslog for its log messages. How to read Pulp's log messages therefore depends on which log handler your operating system uses. Two different log handlers that are commonly used will be documented here, journald and rsyslogd. If you happen to use a different syslog handler on your operating system, please refer to its documentation to learn how to access Pulp's log messages.

Log Level

Pulp's log level can be adjusted with the `log_level` setting in the `[server]` section of `/etc/pulp/server.conf`. This setting is optional and defaults to INFO. Valid choices are CRITICAL, ERROR, WARNING, INFO, DEBUG, and NOTSET.

Note: This setting will only adjust the verbosity of the messages that Pulp emits. If you wish to see all of these messages, you may also need to set the log level on your syslog handler. For example, rsyslog typically only displays INFO and higher, so if you set Pulp to DEBUG it will still be filtered by rsyslog. See the *rsyslogd* section for more information.

journald

journald is the logging daemon that is distributed as part of systemd. If you are using Fedora this is your primary logging daemon, though it's possible that you also have rsyslogd installed. journald is a very nice logging daemon that provides a very useful interface to the logs, journalctl. If your system uses journald, you might not have any logs written to /var/log depending on how your system is configured. For Pulp's purposes, you should use `journalctl` to access Pulp's various logs. Most of the log messages that you will wish to see will have the "pulp" tag on them, so this command will display most of Pulp's log messages:

```
$ sudo journalctl SYSLOG_IDENTIFIER=pulp
```

We'll leave it to the systemd team to thoroughly document `journalctl`, but it's worth mentioning that it can be used to aggregate the logs from Pulp's various processes together into one handy view using it's + operator. Pulp server runs in a variety of units, and if there are problems starting Pulp, you may wish to see log messages from httpd or celery. If you wanted to see the log messages from all server processes together you could use this command:

```
$ sudo journalctl SYSLOG_IDENTIFIER=pulp + SYSLOG_IDENTIFIER=celery + SYSLOG_IDENTIFIER=httpd
```

A `journalctl` flag to know about is `-f`, which performs a similar function as `tail`'s `-f` flag.

rsyslogd

rsyslogd is another popular logging daemon. If you are using RHEL 6, this is your logging daemon. On many distributions, it is configured to log most messages to `/var/log/messages`. If this is your logging daemon, it is likely that all of Pulp's logs will go to this file by default. If you wish to filter Pulp's log messages out and place them into a separate file, you will need to configure rsyslogd to match Pulp's messages. Pulp prefixes all of its log messages with "pulp" to aid in matching its messages in the logging daemon.

If you wish to match Pulp messages and have them logged to a different file than `/var/log/messages`, you may adjust your `/etc/rsyslog.conf`. See RSyslog for details. An alternative could be to use `tail` and `grep` to view Pulp messages logged to `/var/log/messages`.

Why Syslog?

Pulp's use of syslog is a departure from previous Pulp releases which used to write their own log files to /var/log/pulp/. This was problematic for Pulp's 2.4.0 release as Pulp evolved to use a multi-process distributed architecture. Python's file-based log handler cannot be used by multiple processes to write to the same file path, and so Pulp had to do something different. Syslog is a widely used logging protocol, and given the distributed nature of Pulp it was the most appropriate logging solution available.

Other logs

Some of Pulp's other processes still log to files. Those file locations are documented here.

/var/log/pulp/celerybeat.log, /var/log/pulp/reserved_resource_worker-*.log, /var/log/pulp/resource_manager.log
All of these files will only be present if your operating system uses Upstart for init. If you use systemd, these log messages will all be sent to the syslog by the Celery units.

These files will contain messages from Celery's early startup, before it initializes the Pulp application. If there are problems loading Pulp, Celery will log those problems here. Once Pulp initializes, it begins capturing all of the Celery logs and writing them to syslog.

/var/log/httpd/error_log This is where Apache will log errors that the Pulp server itself did not handle. Bootstrap errors often get logged here.

/var/log/httpd/ssl_error_log This is where Apache will log errors that the Pulp server itself did not handle. 5xx level HTTP response codes generally get logged here, often with a stack trace or other information that can help a developer determine what went wrong.

~/.pulp/admin.log pulp-admin logs its activity here.

~/.pulp/consumer.log pulp-consumer logs its activity here.

~/.pulp/server_calls.log HTTP requests and responses get logged by the admin client in this file. To enable/disable this, consult the `[logging]` section of `/etc/pulp/admin/admin.conf`.

~/.pulp/consumer_server_calls.log HTTP requests and responses get logged by the consumer client in this file. To enable/disable this, consult the `[logging]` section of `/etc/pulp/consumer/consumer.conf`.

1.15.2 Common Issues

The server hostname configured on the client did not match the name found in the server's SSL certificate

In some distributions, such as RHEL 6.3 and Fedora 17, the default SSL certificate used by Apache is created with its Common Name set to the hostname of the machine. This can cause Pulp to return an error similar to `The server hostname configured on the client did not match the name found in the server's SSL certificate.`

If you want to connect to localhost, you need to regenerate this certificate, which is stored in /etc/pki/tls/certs/localhost.crt. For testing purposes, delete it, then run `make testcert`. Be sure to answer "localhost" for the "Common Name". Other responses do not matter.

For production installations of Pulp, it is up to the installer to provide appropriate SSL certificates and configure Apache to use them.

Sync from within /tmp fails to find files

If you experience a problem where Pulp cannot find content that is in /tmp, please move that content outside of /tmp and try again.

A sync operation can use a local filesystem path on the server by specifying the feed URL starting with `file:///`. If the content is within /tmp, Apache may fail to read that content on distributions such as Fedora that use private /tmp directories. Since /tmp is temporary and may not persist through a system reboot, it is not generally the best place to put important content anyway.

apr_sockaddr_info_get() failed error when starting apache on F18

You may run into apr_sockaddr_info_get() failed error when starting apache on F18. This is because of incorrect hostname configuration. Make sure your /etc/hosts file contains the hostname of your machine as returned by the 'hostname' command. If not, update /etc/hosts and run 'apachectl restart'.

Qpid connection issues when starting services or executing tasks

When setting up Pulp, or adjusting its configuration, you may encounter connection issues between Pulp and Qpid. If Pulp services cannot connect to the Qpid broker then Pulp cannot continue. The most common root cause of this issue is the Qpid broker not being configured as expected due to changes being put into a `qpidd.conf` that the Qpid broker is not reading from. For Qpid 0.24+ the qpidd.conf file should be located at `/etc/qpid/qpidd.conf` and for earlier Qpid versions, it should be located at `/etc/qpidd.conf`. The user who you run qpidd as must be able to read the `qpidd.conf` file.

I see 'NotFound: no such queue: pulp.task' in the logs

This is experienced on a Pulp installation that uses Qpid 0.18 or earlier, and does not have the qpid-cpp-server-store package installed with their broker. Later version of Qpid do not require this package to be installed. This exception may not occur until the Qpid broker is restarted unexpectedly with other Pulp services running. The exception is shown as Pulp recovers from a Qpid availability issue.

Tasks are accepted but never run

Starting with Pulp 2.6.0, any pulp-admin or API action that creates a Pulp Task will be accepted without error as long as the webserver is running. Once those tasks are accepted, they wait to be executed through a coordination between the non-webserver components: `pulp_celerybeat`, `pulp_resource_manager`, and `pulp_workers`. If your tasks are being accepted but not running, ensure that you have `pulp_celerybeat`, `pulp_resource_manager`, and `pulp_workers` configured and running correctly. If you are using systemd, please see the special note about `pulp_workers` below.

Note: If you are using systemd, the pulp_workers service is really a proxy that starts pulp_worker-0, pulp_worker-1, pulp_worker-2... and so forth, depending on the number of workers you have configured. `systemctl status pulp_workers` will not report status on the real workers, but rather will report status on itself. Therefore if you see a successful status from pulp_workers it only means that it was able to start pulp_worker-0, pulp_worker-1, etc. It does not mean that those services are still running. It is possible to ask for pulp_worker statuses using wildcards, such as `systemctl status pulp_worker-\* -a`, for example.

Warning: Remember that `pulp_celerybeat` and `pulp_resource_manager` must be singletons across the entire Pulp distributed installation. Please be sure to only start one instance of each of these. `pulp_workers` is safe to start on as many machines as you like.

qpid.messaging is not installed

If you are using Qpid as your message broker, you will need the Python package `qpid.messaging`. On Red Hat operating systems, this is provided by the `python-qpid` package.

qpidtoollibs is not installed

If you are using Qpid as your message broker, you will also need the Python package `qpidtoollibs`. On Red Hat operating systems this is provided by either the qpid-tools package or the python-qpid-qmf package, depending on the versions of qpid you are using (newer qpid versions provide it with qpid-tools.)

pulp-manage-db gives an error "Cannot delete queue"

While running pulp-manage-db, you may see "Cannot delete queue xxxxxxxxxxxxxx; queue in use".

You will encounter this while upgrading to Pulp 2.4.0 if there are still 2.3.x or earlier consumers running. All consumers must be upgraded first, or turned off, prior to running the pulp-manage-db that is part of the Pulp 2.3.x –> 2.4.0 upgrade. For more information see the *Pulp 2.3.x –> 2.4.0 upgrade docs*.

Cannot start/stop Qpid – Not enough file descriptors or AIO contexts

In environments with a very large number of Consumers, Pulp relies on the broker to manage a large number of persistent queues. Pulp installations that have a very large number of consumers and are using Qpid may experience issues when starting or stopping qpidd.

If you experience an issue starting or stopping qpidd that complains about file descriptors or AIO contexts, you probably have encountered a scalability limit within Qpid. If you experience this issue you can:

1. Ensure you are running the latest version of Qpid that is available to you. An improvement was made in Qpid 0.30 that improves its scalability of Qpid in this area.
2. Follow the Qpid scalability guide for configuring Qpid to handle a large number of persistent queues.
3. Consider spreading your consumers over multiple Pulp installations, each with its own Qpid broker to reduce the number of Pulp Consumers per broker. The Pulp nodes feature should make this architecture manageable.

Pickle Security Warning

In the Pulp logs you may see a Celery warning similar to the following:

```
CDeprecationWarning:
Starting from version 3.2 Celery will refuse to accept pickle by default.

The pickle serializer is a security concern as it may give attackers
the ability to execute any command.  It's important to secure
your broker from unauthorized access when using pickle, so we think
that enabling pickle should require a deliberate action and not be
the default choice.

If you depend on pickle then you should set a setting to disable this
warning and to be sure that everything will continue working
when you upgrade to Celery 3.2::

   CELERY_ACCEPT_CONTENT = ['pickle', 'json', 'msgpack', 'yaml']

You must only enable the serializers that you will actually use.


 warnings.warn(CDeprecationWarning(W_PICKLE_DEPRECATED))
```

This is related to how data is passed around internally inside of Pulp, and this warning is displayed as part of normal Pulp operation.

User permissions not behaving as expected

Resource names should always start with `/v2` and end with a trailing `/`. For example, the following command will add a permission to `test-user` to create repositories:

```
pulp-admin auth permission grant --resource /v2/repositories/ --login test-user -o create
```

CHAPTER 2

Developer Guide

2.1 How To Contribute

There are three distinct types of contributions that this guide attempts to support. As such, there may be portions of this document that do not apply to your particular area of interest.

Existing Code Add new features or fix bugs in the platform or existing type support projects.

New Type Support Create a new project that adds support for a new content type to Pulp.

Integration Integrate some other project with Pulp, especially by using the event system and REST API.

2.1.1 For New Contributors

If you are interested in contributing to Pulp, the best way is to select a bug that interests you and submit a patch for it. We use Redmine for tracking bugs. Of course, you may have your own idea for a feature or bugfix as well! You may want to send an email to pulp-list@redhat.com or ask in the #pulp channel on freenode before working on your feature or bugfix. The other contributors will be able to give you pointers and advice on how to proceed. Additionally, if you are looking for a bug that is suitable for a first-time contributor, feel free to ask.

Pulp is written in Python. While you do not need to be a Python expert to contribute, it would be advantageous to run through the Python tutorial if you are new to Python or programming in general. Contributing to an open source project like Pulp is a great way to become proficient in a programming language since you will get helpful feedback on your code.

Some knowledge of git and GitHub is useful as well. Documentation on both is available on the GitHub help page.

2.1.2 Contribution Checklist

1. Make sure that you choose the appropriate upstream branch.

   Branching

2. Test your code. We ask that all new code has 100% coverage.

   Testing

3. Please ensure that your code follows our style guide.

   Style Guide

4. Please make sure that any new features are documented and that changes are reflected in existing docs.

   Documentation

5. Please squash your commits and use our commit message guidelines.

   Rebasing and Squashing

6. Make sure your name is in our AUTHORS file found at the root of each of our repositories. That way you can prove to all your friends that you contributed to Pulp!

2.1.3 Developer Guide

Developer Setup

For the Impatient

These instructions will create a developer install of Pulp on a dedicated development instance.

- Start a RHEL 7 or current Fedora x86_64 instance that will be dedicated for development with at least 2GB of memory and 10GB of disk space. More disk space is needed if you plan on syncing larger repos for test purposes.
- If one does not already exist, create a non-root user on that instance with sudo access. If you are using a Fedora cloud image, the "fedora" user is sufficient.
- As that user, `curl https://raw.githubusercontent.com/pulp/pulp/master/playpen/dev-setup.sh | bash`. Note that this installs RPMs and makes system modifications that you wouldn't want to apply on a VM that was not dedicated to Pulp development.
- While it runs, read the rest of this document! It details what the quickstart script does and gives background information on the development process.

Source Code

Pulp's code is stored on GitHub. The repositories should be forked into your personal GitHub account where all work will be done. Changes are submitted to the Pulp team through the pull request process outlined in Merging.

Follow the instructions on that site for checking out each repository with the appropriate level of access (Read+Write v. Read-Only). In most cases, Read-Only will be sufficient; contributions will be done through pull requests into the Pulp repositories as described in Merging.

Dependencies

The easiest way to download the other dependencies is to install Pulp through yum or dnf, which pulls in the latest dependencies according to the spec file.

1. Download the appropriate repository from https://repos.fedorapeople.org/repos/pulp/pulp/

   Example for Fedora:

```
$ cd /etc/yum.repos.d/
$ sudo wget https://repos.fedorapeople.org/repos/pulp/pulp/fedora-pulp.repo
```

2. Edit the repo and enable the most recent testing repository.
3. When using dnf, install the dependencies with this command. `$ sudo dnf install -y $(rpmspec -q --queryformat '[%{REQUIRENAME}\n]' *.spec | grep -v "/.*" | grep -v "python-pulp.* " | grep -v "pulp.*" | uniq)`

4. When using yum, install the main Pulp groups to get all of the dependencies. `$ sudo yum install @pulp-server-qpid @pulp-admin @pulp-consumer`
5. When using yum, remove the installed Pulp RPMs; these will be replaced with running directly from the checked out code. `$ sudo yum remove pulp-\* python-pulp\*`
6. Install some additional dependencies for development:

```
    $ sudo yum install python-setuptools redhat-lsb mongodb mongodb-server \
```

    qpid-cpp-server qpid-cpp-server-store python-qpid-qmf python-nose python-mock python-paste python-pip python-flake8

The only caveat to this approach is that these dependencies will need to be maintained after this initial setup. Leaving the testing builds repository enabled will cause them to be automatically updated on subsequent `yum update` calls. Messages are sent to the Pulp mailing list when these dependencies are updated as well to serve as a reminder to update before the next code update.

Installation

Pulp can be installed to run directly from the checked out code base through `setup.py` scripts. Running these scripts requires the `python-setuptools` package to be installed. Additionally, it is also recommended to install `python-pip` for access to additional setup-related features.

This method of installation links the git repositories as the locally deployed libraries and scripts. Any changes made in the working copy will be immediately deployed in the site-packages libraries and installed scripts. Setup scripts are automatically run for you by `pulp-dev.py`.

Note: Not all Pulp projects need to be installed in this fashion. When working on a new plugin, the Pulp platform can continue to be run from the RPM installation and the pulp_rpm and pulp_puppet plugins would not be required.

Additionally, Pulp specific files such as configuration and package directories must be linked to the checked out code base. These additions are performed by the `pulp-dev.py` script located in the root of each git repository. The full command is:

```
$ sudo python ./pulp-dev.py -I
```

Uninstallation

The `pulp-dev.py` script has an uninstall option that will remove the symlinks from the system into the local source directory, as well as the Python packages. It is run using the `-U` flag:

```
$ sudo python ./pulp-dev.py -U
```

Permissions

The `pulp-dev.py` script links Pulp's WSGI application into the checked out code base. In many cases, Apache will not have the required permissions to serve the applications (for instance, if the code is checked out into a user's home directory).

One solution, if your system supports it, is to use ACLs to grant Apache the required permissions.

For example, assuming the Pulp source was checked out to `~/code/pulp`, the following series of commands would grant Apache the required access:

```
$ cd $HOME
$ setfacl -m user:apache:rwx .
$ cd code
$ setfacl -m user:apache:rwx .
$ cd pulp
$ setfacl -m user:apache:rwx .
```

SELinux

Unfortunately, when developing Pulp SELinux needs to be disabled or run in Permissive mode. Most development environments will be created with `pulp-dev.py`, which deploys Pulp onto the system differently than a rpm based install. The SELinux policy of Pulp expects an RPM layout, and if SELinux is run in Enforcing mode your development to not function correctly.

To turn off SELinux, you can use `sudo setenforce 0` which will set SELinux to permissive. By default, SELinux will be enabled on the next restart so make the change persistent by editing `/etc/sysconfig/selinux`.

```
SELINUX=permissive
```

mod_python

Pulp is a mod_wsgi application. The mod_wsgi and mod_python modules can not both be loaded into Apache at the same time as they conflict in odd ways. Either uninstall mod_python before starting Pulp or make sure the mod_python module is not loaded in the Apache config.

Start Pulp and Related Services

The instructions below are written to be a simple process to start pulp. You should read the user docs for more information on each of these services. Systemd shown below,see user docs for upstart commands.

Start the broker (Though qpid shown here, it is not your only option):

```
sudo systemctl start qpidd
```

Start the agent:

```
sudo systemctl start goferd
```

Install a plugin (the server requires at least one to start):

```
git clone https://github.com/pulp/pulp_rpm.git
cd pulp_rpm
sudo ./manage_setup_pys.sh develop
sudo python ./pulp-dev.py -I
```

Initialize the database:

```
sudo systemctl start mongod
sudo -u apache pulp-manage-db
```

Start the server:

```
sudo systemctl start httpd
```

Start pulp services:

```
sudo systemctl start pulp_workers
sudo systemctl start pulp_celerybeat
sudo systemctl start pulp_resource_manager
```

Login:

```
pulp-admin login -u admin
```

The default password is `admin`

Uninstallation

The `pulp-dev.py` script has an uninstall option that will remove the symlinks from the system into the local source directory. It is run using the `-U` flag:

```
$ sudo python ./pulp-dev.py -U
```

Each python package installed above must be removed by its package name.:

```
$ sudo pip uninstall <package name>
```

Branching Model

Pulp lives on GitHub. The “pulp” repository is for the platform, and then each supported content family (like “rpm” and “puppet”) has its own repository.

Pulp uses a version scheme x.y.z. Pulp’s branching strategy is designed for bugfix development for older “x.y” release streams without interfering with development or contribution of new features to a future, unreleased “x” or “x.y” release. This strategy encourages a clear separation of bugfixes and features as encouraged by *Semantic Versioning* <http://semver.org/>‘_.

master

This is the latest bleeding-edge code. All new feature work should be done out of this branch. Typically this is the development branch for future, unreleased “x” or “x.y” release.

Release Branches

A branch will be made for each x.y release stream named “x.y-release”. For example, the 2.4 release lives in a branch named “2.4-release”. Increments of “z” releases occur within the same release branch and are identified by tags.

The HEAD of each release branch points to a tagged release version. When a new “z” increment version of Pulp is released, the testing branch is merged into the release branch and the new HEAD of the release branch is tagged. Development occurs on a separate development branch.

Release branches are where Read The Docs builds from, so in some situations documentation commits may be merged into a release branch after a release has occurred. For example if a known issue is discovered in a release after it is released, it may be added to the release notes. In those situations the release tag will stay the same and diverge from HEAD.

Testing Branches

Each x.y release will also have a branch for testing builds named “x.y-testing”. For example, the 2.4 stream has a “2.4-testing” branch. This branch is made when we are ready to begin regression testing 2.4.1. After 2.4.0 has been released, the 2.4-dev branch will be merged into 2.4-testing, and this branch will be used to make beta builds. Release candidates will also be built out of this branch. Once we believe the 2.4-testing branch has code that is ready to be release, it will be merged into 2.4-release.

Development Branches

Development for future “z” releases are done in a corresponding branch named “x.y-dev”. For example, assuming Pulp 2.4.0 is released on the branch “2.4-release” and 2.4.1 is being tested in “2.4-testing”, 2.4.2 work will be developed in 2.4-dev. When 2.4.2 is ready to be beta tested, 2.4-dev will be merged into the “2.4-testing” branch at which point “2.4-dev” will be used for 2.4.3 development.

Bug Fix Branches

When creating a Pull Request (PR) that fixes a specific bug, title the PR as you would the *git commit message*.

Feature Branches

Similar to bug fix branches, when creating a pull request that holds features until they are merged into a development branch, the pull request branch should be a brief name relevant to the feature. For example, a branch to add persistent named searches might be named “feature/named-searches”.

Choosing an Upstream Branch

When creating a bug fix or feature branch, it is very important to choose the right upstream branch. The general rule is to always choose the oldest upstream branch that will need to contain your work.

After choosing your upstream branch to merge your changes into and performing that merge, you additionally need to merge forward your commit to all “newer” branches. See *Merging to Multiple Releases* for more information on merging forward from an older branch.

Commit Messages

The primary commit in a bug fix should have a log message that starts with ‘<bug_id> - ‘, for example `123456 - fixes a silly bug`.

Cherry-picking and Rebasing

Don’t do it! Seriously though, this should not happen between release branches. It is a good idea (but not required) for a developer to rebase his or her development branch *before* merging a pull request. Cherry-picking may also be valuable among development branches. However, master and release branches should not be involved in either.

The reason is that both of these operations generate new and unique commits from the same changes. We do not want pulp-x.y and master to have the same bug fix applied by two different commits. By merging the same commit into both, we can easily verify months later that a critical bug fix is present in every appropriate release branch and build tag.

Note: If you are not sure what "rebasing" and "cherry-picking" mean, Pro Git by Scott Chacon is an excellent resource for learning about git, including advanced topics such as these.

Merging

Pull Requests

You have some commits in a branch, and you're ready to merge. The Pulp Team makes use of pull requests for all but the most trivial contributions. Please have a look at our Contribution checklist.

On the GitHub page for the repo where your development branch lives, there will be a "Pull Request" button. Click it. From there you will choose the source and destination branches.

If there is a bug for this issue, please title the pull request "<bug_id> - Short Message". In the comment section below, please include a link to the issue. Use of GitHub's markdown for the link is prefered. Example: `[Issue 123456](https://link.tobug)` Additionally, please also include a link to the pull request in the bugs notes.

For details about using pull requests, see GitHub's official documentation.

Review

Once a pull request has been submitted, a member of the team will review it. That person can indicate their intent to review a particular pull request by assigning it to themself.

Comments on a pull request are meant to be helpful for the patch author. They may point out critical flaws, suggest more efficient approaches, express admiration for your work, ask questions, make jokes, etc. Once review is done, the reviewer assigns the pull request back to the author. The next step for the author will go in one of two directions:

1. If you have commit access and can merge the pull request yourself, you can take the comments for whatever you think they are worth. Use your own judgement, make any revisions you see fit, and merge when you are satisfied. Think of the review like having someone proof-read your paper in college.
2. If you are a community member and do not have commit access, we ask that you take the review more literally. Since the Pulp Team is accepting responsibility for maintaining your code into perpetuity, please address all concerns expressed by the reviewer, and assign it back to them when you are done. The reviewer will strive to make it clear which issues are blocking your pull request from being merged.

Note: *To the community:* The Pulp Team is very grateful for your contribution and values your involvement tremendously! There are few things in an OSS project as satisfying as receiving a pull request from the community.

We are very open and honest when we review each other's work. We will do our best to review your contribution with respect and professionalism. In return, we hope you will accept our review process as an opportunity for everyone to learn something, and to make Pulp the best product it can be. If you are uncertain about comments or instructions, please let us know!

Rebasing and Squashing

Before you submit a pull request, consider an interactive rebase with some squashing. We prefer each PR to contain a single commit. This offers some significant advantages:

- Squashing makes it more likely that your merge will be fast-forward only, which helps avoid conflicts.

- Nobody wants to see a your typo fixes in the commit log. Consider squashing trivial commits so that each commit you merge is as story-focused as possible.
- The `git commit --amend` command is very useful, but be sure that you understand what it does before you use it! GitHub will update the PR and keep the comments when you force push an amended commit.
- Rebasing makes cherry picking features and bug fixes much simpler.

If this is not something that you are comfortable with, an excellent resource can be found here:

How to Rebase a Pull Request

Warning: Keep in mind that rebasing creates new commits that are unique from your original commits. Thus, if you have three commits and rebase them, you must make sure that all copies of those original commits get deleted. Did you push your branch to origin? Delete it and re-push after the rebase.

Merging to Multiple Releases

The most important aspect of merging a change into multiple release branches is *choosing the right branch to start from*.

Once your work is complete, submit a pull request from your GitHub fork into the branch for the oldest release you intend to merge into. Once review and revision is complete, merge your branch from the pull request web page. Do not delete the branch yet.

For cases where there are few merge conflicts, merge your working branch manually into each successively newer release branch, and finally into master. Generally, unless you are resolving conflicts or otherwise modifying your initial fix to accommodate the newer branches, no additional pull requests or review are needed.

For cases where there are substantial merge conflicts whose resolution merits review, create a new branch from your working branch and merge the release branch into it. For example, assume you have branch “username-foo” from the “pulp-2.0” branch.

```
$ git checkout username-foo
$ git checkout -b username-foo-merge-2.1
$ git merge pulp-2.1
```

At this point you can resolve conflicts, then create a pull request from username-foo-merge-2.1 into pulp-2.1.

Merging to Old Releases Only

Infrequently, there may be a need to apply a change to an old release but not newer releases. This should only be a last resort.

One way or another, it is important to merge this change into newer release branches, even if the actual changes don’t get applied. When fixing code that no longer exists in newer branches, simply do the merge and resolve any conflicts that arise.

Otherwise, to merge the work but not apply any of its code changes, use merge strategy “ours”.

```
$ git merge -s ours username-bugfix
```

In either case, git’s history records that your fix has been applied to each release branch. Make sure the human-readable description of your fix accurately describes its scope. For example, a good commit message would be “Fixed memory use issue in ABC system, which was removed in pulp 2.1”, or “Fixed a python 2.4 compatibility issue that is no longer applicable as of pulp 2.2”.

Creating Documentation

Platform vs. Type-Specific User Guides

The platform user guide should cover all generic features of Pulp. When examples are appropriate, it should use the "rpm" content type, but only utilize features that are generic across content types.

User guides for content types should avoid repeating what is already in the platform guide and instead focus on these two topics:

1. What new features does this content type provide? For example, RPM support includes protected repos.
2. Create a quick-start guide that shows examples of how to do the most basic and interesting operations. For example, create a repository, sync it, and publish it. Show more advanced stories as "recipes".

Command Line User Guide

Our command line tools such as `pulp-admin` and `pulp-consumer` do a very good job of self-documenting with help text. Pulp's user guides should not duplicate this information. Enumerating every flag and option of the CLI tools would leave us with two places to maintain the same documentation, which would inevitably go out of sync.

The user guides should instead add value beyond what the CLI tools can self-document. Focus on how to use specific features, show lots of examples, and keep in mind what use cases users are likely to be interested in.

Examples should not include long lines that will require horizontal scrolling.

All example commands should begin with only `$ `` as a prompt.  Commands that must be run as root should be shown using ``sudo`.

Docs Layout

Relative to the root of pulp, the user guide is stored at `dev/sphinx/user-guide/` and the dev guide is stored at `docs/sphinx/dev-guide/`.

Read the Docs

Pulp's documentation is hosted on Read the Docs. Links to all current documentation can be found at http://www.pulpproject.org/docs.

RTD Versions

When viewing docs on Read the Docs, there are multiple versions linked in the bottom-left corner of the page. Past releases each have a link named "pulp-x.y" and are built from the most recent commit on the corresponding "pulp-x.y" release branch. Documentation shown on Read the Docs must be merged onto the appropriate branch for it to be displayed. The "latest" version corresponds to the most recently released version of Pulp.

Docs automatically get built when a commit happens to a corresponding branch. However, it seems that builds may not happen automatically when only a merge takes place.

Note: You can manually start a build on Read the Docs for a specific version using the user guide build page or the dev guide build page.

There may be a "staging" version at times. This build is used by the team to share documentation that has not yet been reviewed.

Editing the Docs

The Pulp docs support intersphinx and extlinks.

To refer to a document in a plugin or platform, you can do something like so:

```
:ref:`installation <platform:server_installation>`
```

This will create a link to the correct reference in the platform docs.

Use the `:redmine:` directive to easily create links to Pulp issues. For example:

```
:redmine:`123`
```

Creates a link to issue 123 like this: #123. You can also set the text for the link using this syntax:

```
:redmine:`my new link text <123>`
```

Which creates this link: my new link text There is also a `:fixedbugs:` directive to link to all bugs for a specific version of Pulp. This is useful in release notes. For example:

```
:fixedbugs:`2.6.0`
```

Create a link to bugs fixed in 2.6.0 like this: bugs fixed in 2.6.0. This can have its link text set using the syntax:

```
:fixedbugs:`these great bugs were fixed <2.6.0>`
```

Which creates this link: these great bugs were fixed

Building the Docs

Anyone can build the docs in their own dev environment, which is useful for proofing changes to the docs before committing them. For either the user guide or the dev guide, navigate to the base docs folder and run `make html`. Once run, the html is available in `_build/html`.

The html is built with the vanilla sphinx theme, so the look and feel is different than Read the Docs look and feel.

You do not need to clean the docs before rebuilding. If you do need to clean the docs, you should run `make clean` from the documentation root.

Bugs

All bugs and feature requests (stories) are tracked in Pulp's Redmine instance. You can view existing bugs or existing stories.

How to file a bug

Bugs are one of the main ways that the Pulp team interacts with users (pulp-list@redhat.com and the #pulp IRC channel being the other methods).

You can file a new bug or feature request.

Warning: Security related bugs need to be marked as private when reported. Use the private checkbox (top right) for this.

If you are filing an issue or defect, select `Issue` as the *Tracker*. If you are filing a feature request, select `Story`.

Fill in the *Subject* and *Description*. Leave the status at `NEW`. Please select the closest corresponding *Category*, if any. Select the *Severity* field and any *Tags* based on your best judgement.

Use the *Version* field to indicate which Pulp version you are using. It has an entry for each Pulp release (2.0.6, 2.0.7, 2.1.0, etc.). If a bug is found when running from source instead of a released version, the value `master` should be selected.

Use the *OS* field to indicate which Operating System the bug was discovered on.

You can also upload attachments, but please only upload relevant data. For example, if you have an entire log which contains some errors, please trim it to just the relevant portions and upload those.

Once a week, the Pulp team triages all new bugs, at which point its *Severity* rating and other aspects of the report will be evaluated. If necessary, the bug may be commented on requesting more information or clarification from the reporter. When a bug has enough information, its *Priority* rating set and is marked as triaged using the *Triaged* boolean.

Fixing

When fixing a bug, all bugs will follow this process, regardless of how trivial.

Developer

1. Once the bug has been triaged it waits for a developer to pick it up. Generally developers should pick bugs from the top of the Prioritized Bugs query.
2. Once a bug is selected, the developer sets themselves as the assignee and also sets the bug state to `ASSIGNED`.
3. The developer creates a new remote branch for the bug on their GitHub fork.
4. When the fix is complete, the developer submits a pull request for the bug into the appropriate branch (master, release branch, etc.). It's appreciated by the reviewer if a link to the bug is included in the merge request, as well as a brief description of what the change is. It is not required to find and assign someone to do the review.
5. When the pull request is submitted, the developer changes the status of the bug to `POST`.
6. Wait for someone to review the pull request. The reviewer will assign the pull request back to the developer when done and should also ping them through other means. The developer may take the reviewer's comments as they see fit and merge the pull request when satisfied. Once merged, set bug status to `MODIFIED`. It is also helpful to include a link to the pull request in a comment on the bug.
7. Delete both local and remote branches for the bug.

Note: See Branching Model for more information on how to create branches for bug fixes.

Reviewer

1. When reviewing a pull request, all feedback is appreciated, including compliments, questions, and general Python knowledge. It is up to the developer to decide what (if any) changes will be made based on each comment.
2. When done reviewing, assign the pull request back to the developer and ping them through other means.

Triaging Bugs

Once a week, a rotating subset of the Pulp team meets to sort through that week's new bugs. Bugs that need additional information will have notes put onto the issue asking for input. Unless a Redmine user specifically disabled e-mail support, adding a note will e-mail the reporter. Bugs with enough information will have their severity and priority set, as well as a component if appropriate. Once triaged, the bug is included in the Prioritized Bugs query and awaits a developer to pick it up.

The Pulp team uses some additional Tags to help keep track of bugs.

Tag Name	Usage
Documentation	The bug/story itself is documentation related.
EasyFix	A bug that is simple to fix, at least in theory.

You may occasionally see discussion in #pulp or on the mailing list about "bug grooming". This simply means that someone is applying the rules above to existing bugs that are not new. This is needed from time to time to keep the bug list up to date.

Building Instructions

Getting Started

Concepts There are some concepts you should internalize before you begin making builds. Koji has a concept called tags. A tag is essentially a grouping of package builds. Pulp uses one Koji tag per Pulp X.Y release stream, per distribution, per architecture. For example, the 2.6 releases of pulp will build into the pulp-2.6-<distribution> tags in koji. You can see the full list of Pulp's Koji tags here.

Another thing to know about Koji is that once a particular NEVRA (Name, Epoch, Version, Release, Architecture) is built in Koji, it cannot be built again. However, it can be tagged into multiple Koji tags. For example, if `python-celery-3.1.11-1.el7.x86_64` is built into the `pulp-2.4-rhel7` tag and you wish to add that exact package in the `pulp-2.5-rhel7` tag, you cannot build it again. Instead, you must tag that package for the new tag. You will see later on in this document that Pulp has a tool to help you do this.

Pulp release and testing builds are collections of components that are versioned independently. For example, the core Pulp server may be at version 2.6 while pulp_docker may be at version 1.0. This assembly is accomplished using release definitions specified in the `pulp_packaging/ci/config/releases/<build-name>.yaml` files. Each file specifies the detail of a build that the Pulp build scripts can later assemble. The components within that file specify the target koji tag as well as the individual git repositories and branches that will be assembled as part of a build. In addition it specifies the directory within https://repos.fedorapeople.org/repos/pulp/pulp/testing/automation/ where the build results will be published.

Because there is no way to automatically determine when a particular component needs a new version, or what that version should be, the build-infrastructure assumes that whatever version is specified in the rpm spec file is the final version that is required. If a release build of that version has already been built in koji then those RPMs will be used. Ideally, whenever a branch is forked, or the first commit is pushed into a branch after a release build, the version should be incremented.

Tools used when building Test or release builds (with the exclusion of the signing step) may be performed using Jenkins. There are automated jobs that will run nightly which build repositories that can be used for validation. When those jobs are initiated manually there is a parameter to enable the release build process in koji. If a release build is performed with Jenkins you will still need to sign the rpms and manually push them to the final location on fedorapeople.

Pulp has some helper scripts in the pulp_packaging/ci directory to assist with builds. These wrapper scripts call tito and koji to do the actual tagging and build work.

Both packages are in Fedora and EPEL so you should not need to install from source. Technically you do not need to ever call these scripts directly. However, some familiarity with both tito and koji is good, especially when debugging build issues.

What you will need In order to build Pulp, you will need the following from the Foreman team:

1. An account on Foreman's Koji instance
2. A client certificate for your account
3. The Katello CA certificate

See the Foreman Wiki to get these items.

In order to publish builds to the Pulp repository, you will need the SSH keypair used to upload packages to the fedorapeople.org repository. You can get this from members of the Pulp team.

Configuring your build environment If you are interested in building Pulp, it is strongly recommended that you use a separate checkout from your normal development environment to avoid any potential errors such as building in local changes, or building the wrong branches. It is also a good idea to use a build host in a location with good outbound bandwidth, as the repository publish can be at or over 250 MB. Thus, the first step is to make a clean checkout of the pulp_packging somewhere away from your other checkouts:

```
$ mkdir ~/pulp_build
$ cd ~/pulp_build
$ git clone git@github.com:pulp/pulp_packaging.git; done;
```

The next step is to install and configure the Koji client on your machine. You will need to put the Katello CA certificate and your client certificate in your home folder:

```
$ sudo yum install koji
```

Here is an example $HOME/.koji/config file you can use:

```
``
[koji]

;configuration for koji cli tool

;url of XMLRPC server
server = http://koji.katello.org/kojihub

;url of web interface
weburl = http://koji.katello.org/koji

;url of package download site
topurl = http://koji.katello.org/

;path to the koji top directory
;topdir = /mnt/koji

;configuration for SSL athentication

;client certificate
cert = ~/.katello.cert

;certificate of the CA that issued the client certificate
ca = ~/.katello-ca.cert
```

```
;certificate of the CA that issued the HTTP server certificate
serverca = ~/.katello-ca.cert
``
```

Make sure you install your Katello CA certificate and client certificate to the paths listed in the example above:

```
$ cp <katello CA> ~/.katello-ca.cert
$ cp <katello client cert> ~/.katello.cert
```

If all went well, you should be able to say hello to Koji:

```
$ [rbarlow@notepad]~% koji moshimoshi
olá, rbarlow!

You are using the hub at http://koji.katello.org/kojihub
```

Next, you should install Tito:

```
$ sudo yum install tito
```

Now you are ready to begin building.

Dependencies

Building Dependencies If you wish to add or update the version or release of one of our dependencies, you should begin by adding/updating the dependency's tarball, patches, and spec file in the Pulp git repository as appropriate for the task at hand. **Don't forget to set the version/release in the spec file.** Once you have finished that work, you are ready to test the changes. In the directory that contains the dependency, use tito to build a test RPM. For example, for python-celery:

```
$ cd deps/python-celery
$ tito build --test --rpm
```

Pay attention to the output from tito. There may be errors you will need to respond to. If all goes well, it should tell you the location that it placed some RPMs. You should install these RPMs and test them to make sure they work with Pulp and that you want to introduce this change to the repository.

If you are confident in your changes, submit a pull request with the changes you have made so far. Once someone approves the changes, merge the pull request. Once you have done this, you are ready to tag the git repository with your changes:

```
$ tito tag --keep-version
```

Pay attention to the output of tito here as well. It will instruct you to push your branch and the new tag to GitHub.

Warning: It is very important that you perform the steps that tito instructs you to do. If you do not, others will not be able to reproduce the changes you have made!

At this point the dependency will automatically be built during all test builds of Pulp and will automatically have a release build performed when the next release build containing this dependency is performed.

Test Building Pulp and the plugins

Are you ready to build something? If so, you should *cd* to the `pulp_packaging/ci` directory. The next step is to perform the build. There is a helper script that will perform the following actions:

1. Load the specified configuration from `pulp_packaging/ci/config/releases`.
2. Clone all the required git repositories to the `working/<repo_name>` directory.
3. Check out the appropriate branch for each git repos.
4. Find all the spec files in the repositories.
5. Check koji to determine if the version in the spec already exists in koji.
6. Test build all the packages that do not already exist in koji.
7. Optionally release build all the packages that do not already exist in koji.
8. Download the already existing packages from koji.
9. Download the scratch built packages from koji.
10. Assemble the repositories for all the associated distributions.
11. Optionally push the repositories to fedorapeople.

The `build-all.py` script can be used to do all of this. For example, to perform a test build of the 2.6-dev release as specified in `pulp_packaging/ci/config/releases/2.6-dev.py` where the results are not pushed to fedorapeople:

```
$ build-all.py 2.6-dev --disable-push
```

At this point, you may wish to ensure that the branches are all merged forward to master. This step is not strictly required at this point, as we will have to do it again later. However, sometimes developers forget to do this, and it may be advantageous to resolve potential merge conflicts before tagging.

Here is a quick way to see if everything's been merged forward through to master. You'll likely want to edit the BRANCHES list so the branch you are releasing from is the first in the list:

```
$ BRANCHES="2.4-release 2.4-testing 2.4-dev 2.5-testing 2.5-dev"; git log origin/master | fgrep -f <
```

Next it is time to raise the version of the branches. This process is different depending on the stream you are building.

Note: Pulp uses the release field in pre-release builds as a build number. The first pre-release build will always be 0.1, and every build thereafter prior to the release will be the last release plus 0.1, even when switching from alpha to beta. For example, if we have build 7 2.5.0 alphas and it is time for the first beta, we would be going from 2.5.0-0.7.alpha to 2.5.0-0.8.beta. We loosely follow the Fedora Package Versioning Scheme.

Checking the versions that will be built You can use the `build-all.py` script to validate versions that will be built/downloaded. This can be used to double check to ensure that the correct versions have been set in the spec files. This command will exit immediately after displaying the versions.:

```
$ ./build-all.py 2.6-testing --show-versions
```

If one of the versions is not what you expect updated it using the `update-version.py` script and push the change to GitHub.

Updating Versions The `update-version.py` script will scan a directory and set the versions inside according to your specifications. For example, to if you have been working in the `/tmp/pulp_build` directory the following command can be used to update pulp_docker from the release candidate to the release build version.:

```
$ cd /tmp/pulp_build/pulp_packaging/ci/
$ ./build-all.py 2.6-testing --show-versions
.. Versions displayed ..
$ ./update-version.py --update-type stage /tmp/pulp_build/pulp_packaging
```

At this point you can inspect the files to ensure the versions are as you expect. The changes will still need to be committed and pushed to GitHub before they can be used to build. You can rerun the `build-all.py` script to check the versions again after your changes have been pushed to GitHub.

Submit to Koji We are now prepared to submit the build to Koji. This task is simple:

```
$ cd pulp_packaging/ci
$ ./build-all.py 2.6-testing --release
```

This command will build SRPMs, upload them to Koji, and monitor the resulting builds. If any of them fail, you can view the failed builds to see what went wrong. If the build was successful, it will automatically download the results into a new folder called mash that will be a peer to the `pulp_packaging` directory.

At the end it will automatically upload the resulting build to fedorapeople in the directory specified in the release config file.

Now is a good time to start our Jenkins builder to run the unit tests in all the supported operating systems. You can configure it to run the tests in the git branch that you are building. Make sure these pass before publishing the build.

After the repositories are built, the next step is to merge the tag changes you have made all the way forward to master. You may experience merge conflicts with this step. Be sure to merge forward on all of the repositories.

Warning: Do not use the ours strategy, as that will drop the changelog entries. You must manually resolve the conflicts!

You may experience conflicts when you push these changes. If you do, merge your checkout with upstream. Then you can `git push <branch>:<branch>` after you check the diff to make sure it is correct. Lastly, do a new git checkout elsewhere and check that `tito build --srpm` is tagged correctly and builds.

Updating Docs

The docs for Pulp platform and each plugin use intersphinx to facilitiate linking between documents. It is important that each branch of Pulp and Pulp plugins link to the correct versions of their sister documents. This is accomplished by editing the URLs in the `intersphinx_mapping` variable, which is set in `docs/conf.py` for both Pulp platform and all plugins.

Here are some guidelines for what to set the URL to:

- The master branch of Pulp or any plugins should always point to "latest".
- Plugins should point to the latest stable version of Pulp that they are known to support.
- Pulp platform's intersphinx URLs should point back to whatever the plugin is set to. For example, if the "pulp_foo" plugin's docs for version 1.0 point to the "2.8-release" version of the Pulp platform docs, then platform version 2.8 should point back to "1.0-release" for pulp_foo's docs. This ensures a consistent experience when users click back and forth between docs.

Testing the Build

In order to test the build you have just made, you can publish it to the Pulp testing repositories. Be sure to add the shared SSH keypair to your ssh-agent, and cd into the mash directory:

```
$ ssh-add /path/to/key
$ cd mash/
$ rsync -avz --delete * pulpadmin@repos.fedorapeople.org:/srv/repos/pulp/pulp/testing/<X.Y>/
```

For our 2.4 beta example, the rsync command would be:

> $ rsync -avz –delete * pulpadmin@repos.fedorapeople.org:/srv/repos/pulp/pulp/testing/2.4/

You can now run the automated QE suite against the testing repository to ensure that the build is stable and has no known issues. We have a Jenkins server for this purpose, and you can configure it to test the repository you just published.

Signing the RPMS

Before signing RPMs, you will need access to the Pulp signing key. Someone on the Pulp team can provide you with this. Additionally you should be familiar with the concepts in the Creating GPG Keys guide.

All alpha, beta and GA RPMs should be signed with the Pulp team's GPG key. A new key is created for each X release (3.0.0, 4.0.0, etc). If you are doing a new X release, a new key needs to be created. To create a new key, run `gpg --gen-key` and follow the prompts. We usually set "Real Name" to "Pulp (3)" and "Email address" to "pulp-list@redhat.com". Key expiriation should occur five years after the key's creation date. After creating the key, export both the private and public keys. The public key should be saved as `GPG-RPM-KEY-pulp-3` and the private as `pulp-3.private.asc`. The password can go into `pulp-3-password.txt`. Please update `encrypt.sh` and `decrypt.sh` as well to include the new private key and password file. Run `encrypt.sh` to encrypt the new keys.

Warning: If you are making an update to the key repo, be sure to always verify that you are not committing the unencrypted private key or password file!

Note: If you are adding a new team member, just add their key to `encrypt.sh` and `decrypt.sh`, then re-encrypt the keys and commit. The new team member will also need to obtain the "sign" permission in koji.

The `GPG-RPM-KEY-pulp-3` file should be made available under https://repos.fedorapeople.org/repos/pulp/pulp/.

If you are simply creating a new build in an existing X stream release, you need to perform some one-time setup steps in your local environment. First, create or update your `~/.rpmmacros` file to include content like so, substituting X with your intended release:

```
%_gpg_name Pulp (X)
```

Next, run the following from your mash directory:

```
$ find -name "*.rpm" | xargs rpm --addsign
```

This will sign all of the RPMs in the mash. You then need to import signatures into koji:

```
$ find -name "*.rpm" | xargs koji import-sig
```

Note: Koji does not store the entire signed RPM. It merely stores the additional signature metadata, and then re-creates a signed RPM in a different directory when the `write-signed-rpm` command is issued. The original unsigned RPM will remain untouched.

As `list-signed` does not seem to work, do a random check in http://koji.katello.org/packages/ that http://koji.katello.org/packages/<name>/<version>/<release>/data/sigcache/<sig-hash>/ exists and has some content in it. Once this is complete, you will need to tell koji to write out the signed RPMs (both commands are run from your mash dir):

```
$ for r in `find -name "*src.rpm"`; do basename $r; done | sort | uniq | sed s/\.src\.rpm//g > /tmp/b
$ for x in `cat /tmp/builds`; do koji write-signed-rpm <SIGNATURE-HASH> $x; done
```

Sync down your mash one more time (run from the `pulp_packaging/ci` dir):

```
$ ./build-all.py <release_config> --disable-push --rpmsig <SIGNATURE-HASH>
```

Note: This command does not download signed RPMs for RHEL 5, due to bugs in RHEL 5 related to signature verification. While we sign all RPMs including RHEL 5, we do not publish the signed RPMs for this particular platform.

After it is synced down, you can publish the build.

Publishing the Build

Alpha builds should only be published to the testing repository. If you have a beta or stable build that has passed tests in the testing repository, and you wish to promote it to the appropriate place, you can use a similar rsync command to do so:

```
$ rsync -avz --delete * pulpadmin@repos.fedorapeople.org:/srv/repos/pulp/pulp/<stream>/<X.Y>/ --dry-r
```

Replace stream with "beta" or "stable", and substitute the correct version. For our 2.4 beta example:

```
$ rsync -avz --delete * pulpadmin@repos.fedorapeople.org:/srv/repos/pulp/pulp/beta/2.4/ --dry-run
```

Note the `--dry-run` argument. This causes rsync to print out what it *would* do. Review its output to ensure that it is correct. If it is, run the command again while omitting that flag.

Warning: Be sure to check that you are publishing the build to the correct repository. It's important to never publish an alpha build to anything other than a testing repository. A beta build can go to testing or the beta repository (but never the stable repository), and a stable build can go to a testing or a stable repository.

If you have published a beta build, you must move all issues and stories for the target release from `MODIFIED` to `ON_QA`.

After publishing a beta build, email pulp-list@redhat.com to announce the beta. Here is a typical email you can use:

```
Subject: [devel] Pulp beta <version> is available

Pulp <version> has been published to the beta repositories. This fixes <add some text here>.
```

If you have published a stable build, there are a few more items to take care of:

1. Update the "latest release" text on http://www.pulpproject.org/.
2. Verify that the new documentation was published. You may need to explicitly build them if they were not automatically build.
3. Update the channel topic in #pulp on Freenode with the new release.
4. Move all bugs that were in the `VERIFIED` state for this target release to `CLOSED CURRENT RELEASE`.

After publishing a stable build, email pulp-list@redhat.com to announce the new release. Here is a typical email you can use:

```
Subject: Pulp <version> is available!

The Pulp team is pleased to announce that we have released <version>
to our stable repositories. <Say if it's just bugfixes or bugs and features>.

Please see the release notes[0][1][2] if you are interested in reading about
the fixes that are included. Happy upgrading!

[0] link to pulp release notes (if updated)
[0] link to pulp-rpm release notes (if updated)
[0] link to pulp-puppet release notes (if updated)
```

Please ensure that the release notes have in fact been updated before sending the email out.

New Stable Major/Minor Versions If you are publishing a new stable <X.Y> build that hasn't been published before (i.e., X.Y.0-1), you must also update the symlinks in the repository. There is no automated tool to perform this step. ssh into repos.fedorapeople.org using the SSH keypair, and perform the task manually. Ensure that the "X" symlink points at the latest X.Y release, and ensure that the "latest" symlink points at that largest "X" symlink. For example, if you just published 3.1.0, and the latest 2.Y version was 2.5, the stable folder should look similar to this:

```
[pulpadmin@people03 pulp]$ ls -lah stable/
total 24K
drwxrwxr-x. 6 pulpadmin pulpadmin 4.0K Sep 17 18:26 .
drwxrwxr-x. 7 jdob      gitpulp   4.0K Sep  8 22:40 ..
lrwxrwxrwx. 1 pulpadmin pulpadmin    3 Aug  9 06:35 2 -> 2.5
drwxrwxr-x. 7 pulpadmin pulpadmin 4.0K Aug 15  2013 2.1
drwxrwxr-x. 7 pulpadmin pulpadmin 4.0K Sep  6  2013 2.2
drwxrwxr-x. 7 pulpadmin pulpadmin 4.0K Dec  5  2013 2.3
drwxrwxr-x. 7 pulpadmin pulpadmin 4.0K Aug  9 06:32 2.4
drwxrwxr-x. 7 pulpadmin pulpadmin 4.0K Aug 19 06:32 2.5
drwxrwxr-x. 7 pulpadmin pulpadmin 4.0K Aug 20 06:32 3.0
drwxrwxr-x. 7 pulpadmin pulpadmin 4.0K Aug 24 06:32 3.1
lrwxrwxrwx. 1 pulpadmin pulpadmin    3 Aug 24 06:35 3 -> 3.1
lrwxrwxrwx. 1 pulpadmin pulpadmin   29 Aug 20 06:32 latest -> /srv/repos/pulp/pulp/stable/3
```

The `rhel-pulp.repo` and `fedora-pulp.repo` files also need to be updated for the new GPG public key location if you are creating a new X release.

2.2 Architecture

Pulp can be viewed as consisting of two parts, the platform (which includes both the server and client applications) and plugins (which provide support for a particular set of content types).

2.2.1 Platform

The Pulp application is a *platform* into which support for particular types of content is installed. The Pulp Platform refers to the following major components:

- **Server** - The server-side application includes all of the infrastructure for handling repositories, consumers, and users. This includes the plumbing for handling functionality related to these pieces, such as synchronizing a repository or sending an install request to a consumer. The actual behavior of how those methods function, however, is dependent on the plugin fielding the request.

- **Client** - The platform includes both the admin and consumer clients. These clients contain type-agnostic commands, such as logging in or handling events. Each client can be enhanced with type-specific commands through the use of *extensions*.
- **Agent** - The agent runs on each consumer and is used to field requests sent from the Pulp server to that consumer. Similar to the client, the platform provides the agent and relies on the use of *handlers* to provide support for a particular content type.

2.2.2 Plugins

Support for handling specific content types, such as RPMs or Puppet Modules, is provided through plugins into each of the three major platform components.

Note: The term plugin is typically used to refer just to the server-side plugins. Collectively, the set of server, client, and agent plugins for a particular set of content types are usually referred to as a "support bundle."

- **Plugin** - The server-side components that handle either downloading and inventorying content in Pulp (called an *importer*) or publishing content in a repository (referred to as a *distributor*).
- **Extension** - The components plugged into either the admin or consumer command line clients to provide new commands in the client for type-specific operations.
- **Handler** - The agent-side components that are used to field invocations from the server to a consumer, such as binding to a repository or installing content.

2.2.3 Git Repositories

Pulp's code is stored on GitHub. The Pulp organization's GitHub repositories are divided into two types:

- The Pulp repository is used for all platform code, including the server, clients, and agent. There should be no code in here that caters to a specific content type. Put another way, all plugins to the platform components are located outside of this repository.
- Each type support bundle (RPM, Puppet, etc.) is in its own repository. Each of these repositories contains their own *setup.py* and RPM spec files, as well as any other configuration files needed to support the plugins (for example, httpd configuration files).

2.3 Conventions

2.3.1 Searching

Search Criteria

Pulp offers a standard set of criteria for searching through collections as well as for specifying resources in a collection to act upon.

Any API that supports this criteria will accept a JSON document with a **criteria** field. The criteria field will be a sub-document with the following fields:

- **filters**
- **sort**
- **limit**

- **skip**
- **fields**

The **filters** field is itself a document that specifies, using the pymongo find specification syntax, the resource fields and values to match. For more information on the syntax, see: http://www.mongodb.org/display/DOCS/Querying

The **sort** field is an array of arrays. Each specifying a field and a direction.

The **limit** field is a number that gives the maximum amount of resources to select. Useful for pagination.

The **skip** field is a number that gives the index of the first resource to select. Useful for pagination.

The **fields** field is an array of resource field names to return in the results.

Example search criteria:

```
{
 "criteria": {
   "filters": {"id": {"$in": ["fee", "fie", "foe", "foo"]}, "group": {"$regex": ".*-dev"}},
   "sort": [["id", "ascending"], ["timestamp", "descending"]],
   "limit": 100,
   "skip": 0,
   "fields": ["id", "group", "description", "timestamp"]}
}
```

Unit Association Criteria

The criteria when dealing with units in a repository is slightly different from the standard model. The metadata about the unit itself is split apart from the metadata about when and how it was associated to the repository. This split occurs in the filters, sort, and fields sections.

The valid fields that may be used in the association sections are as follows:

- `created` - Timestamp in iso8601 format indicating when the unit was *first* associated with the repository.
- `updated` - Timestamp in iso8601 format indicating when the unit was most recently confirmed to be in the repository.
- `owner_type` - Indicates where the association between the unit and the repository was created. Valid values are `importer` and `user`.
- `owner_id` - Indicates specifically who created the association. This will be the importer ID if added by an importer or the user login if added by a user.

Example unit association criteria:

```
{
  'type_ids' : ['rpm'],
  'filters' : {
    'unit' : <mongo spec syntax>,
    'association' : <mongo spec syntax>
  },
  'sort' : {
    'unit' : [ ['name', 'ascending'], ['version', 'descending'] ],
    'association' : [ ['created', 'descending'] ]
  },
  'limit' : 100,
  'skip' : 200,
  'fields' : {
    'unit' : ['name', 'version', 'arch'],
    'association' : ['created']
  },
  'remove_duplicates' : True
}
```

Search API

The search API is consistent for various data types. Please see the documentation for each individual resource type for information about extra parameters and behaviors specific to that type.

Searching can be done via a GET or POST request, and both utilize the concepts of *Search Criteria*.

The most robust way to do a search is through a POST request, including a JSON- serialized representation of a Critera in the body under the attribute name "criteria". This is useful because some filter syntax is difficult to serialize for use in a URL.

Returns items from the Pulp server that match your search parameters. It is worth noting that this call will never return a 404; an empty array is returned in the case where there are no items in the database.

Method: POST
Path: `/pulp/api/v2/<resource type>/search/`
Permission: read
Request Body Contents: include the key "criteria" whose value is a mapping structure as defined in *Search Criteria*
Response Codes:

- **200** - containing the list of items

Return: the same format as retrieving a single item, except the base of the return value is a list of them

The GET method is slightly more limiting than the POST alternative because some filter expressions may be difficult to serialize as query parameters.

Method: GET
Path: `/pulp/api/v2/<resource type>/search/`
Permission: read
Query Parameters: query params should match the attributes of a Criteria object as defined in *Search Criteria*. The exception is that field names should be specified in singular form with as many 'field=foo' pairs as may be required.

For example:

```
/pulp/api/v2/<resource type>/search/?field=id&field=display_name&limit=20
```

Response Codes:

- **200** - containing the array of items

Return: the same format as retrieving a single item, except the base of the return value is an array of them

2.3.2 Exception Handling

In the event of a failure (non-200 status code), the returned body will be a JSON document describing the error. This applies to all method calls; for simplicity, the individual method documentation will not repeat this information. The document will contain the following:

- **http_status** *(number)* - HTTP status code describing the error.
- **href** *(string)* - Currently unused.
- **error_message** *(string)* - Description of what caused the error; may be empty but will be included in the document.
- **exception** *(string)* - Message extracted from the exception if one occurred on the server; may be empty if the error was due to a data validation instead of an exception.
- **traceback** *(string)* - Traceback of the exception if one occurred; may be empty for the same reasons as exception.
- **error** *(object)* - error details and nested errors. *Error Details*

All methods have the potential to raise a 500 response code in the event of an unexpected server-side error. Again, for simplicity that has not been listed on a per method basis but applies across all calls.

Example serialized exception:

```
{
 "exception": null,
 "traceback": null,
 "_href": "/pulp/api/v2/repositories/missing-repo/",
 "resource_id": "missing-repo",
 "error_message": "Missing resource: missing-repo",
 "http_request_method": "DELETE",
 "http_status": 404,
 "error": {
           "code": "PLP0009",
           "description": "Missing resource(s): foo",
           "data": {"resource": "foo"},
           "sub_errors": []
           }
}
```

2.3.3 Error Details

Pulp is moving to provide more programmatically useful results when errors occur. One of the primary ways we are doing this is through the new "error" object. This object will be included in the body for all JSON calls that have errors. The error object will contain the following fields.

- **code** ***(string)*** **- A 7 digit string uniquely identifying this error. The first 3 characters** are [A-Z] and identify the project in which the error occurred. Today the possible values are "PLP", "PPT", and "RPM" for the pulp, pulp_puppet and pulp_rpm projects. The last 4 digits are numeric to identify the error.
- **description** *(string)* - A user readable message describing the error that occurred
- **data** ***(object)*** **- The data specific to this error. Each error code specifies the fields that** will be included in this object.
- **sub_errors** *(array)* - An array of error details objects that contributed to this error.

Example serialized error details:

```
{
 "code": "PLP0018",
 "description": "Duplicate resource: foo",
 "data": {"resource_id": "foo"},
 "sub_errors": []
}
```

2.3.4 Error Codes

Pulp Error codes should be segmented. The following segments have been established * PLP0000-PLP0999 - General Server Errors (and legacy PulpException errors) * PLP1000-PLP2999 - Validation errors

2.3.5 Synchronous and Asynchronous Calls

Overview

Pulp uses an advanced task queueing system that detects and avoids concurrent operations that conflict. All REST API calls are managed through this system. Any REST API will return one of three responses:

- Success Response - 200 OK or a 201 CREATED
- Postponed Response - 202 ACCEPTED
- Conflict Response - 409 CONFLICT

A success response indicates no conflicts were detected and the REST call executed. This is what is typically expected from a REST API call.

A postponed response indicates that some portion of the command has been queued to execute asynchronously. In this case a *Call Report* will be returned with the results of the synchronously executed portion of the command, if there are any, and a list of the tasks that have been spawned to complete the work in the future.

More information on retrieving and displaying task information can be found *in the Task Management API documentation*.

A conflict response indicates that a conflict was detected that causes the call to be unserviceable now or at any point in the future. An example of such a situation is the case where an update operation is requested after a delete operation has been queued for the resource. The body of this response is Pulp's standard exception format including the reasons for the response.

Call Report

A 202 ACCEPTED response returns a **Call Report** JSON object as the response body that has the following fields:

- **result** *(Object)* - the return value of the call, if any
- **error** *(Object)* - error details if an error occurred. See *Error Details*.
- **spawned_tasks** *(array)* - list of references to tasks that were spawned. Each task object contains the relative url to retrieve the task and the unique ID of the task.

Example Call Report:

```
{
 "result": {},
 "error": {},
 "spawned_tasks": [{"_href": "/pulp/api/v2/tasks/7744e2df-39b9-46f0-bb10-feffa2f7014b/",
                    "task_id": "7744e2df-39b9-46f0-bb10-feffa2f7014b" }]
}
```

2.3.6 Scheduled Tasks

Pulp can schedule a number of tasks to be performed at a later time, on a recurring interval, or both.

Pulp utilizes the ISO8601 interval format for specifying these schedules. It supports recurrences, start times, and durations for any scheduled task. The recurrence and start time is optional on any schedule. When the recurrence is omitted, it is assumed to recur indefinitely. When the start time is omitted, the start time is assumed to be now.

More information on ISO8601 interval formats can be found here: http://en.wikipedia.org/wiki/ISO_8601#Time_intervals

Scheduled tasks are generally treated as sub-collections and corresponding resources in Pulp's REST API. All scheduled tasks will have the following fields:

- `_id` The schedule id
- `_href` The uri path of the schedule resource
- `schedule` The schedule as specified as an ISO8601 interval
- `failure_threshold` The number of consecutive failures to allow before the scheduled task is automatically disabled
- `enabled` Whether or not the scheduled task is enabled
- `consecutive_failures` The number of consecutive failures the scheduled tasks has experienced
- `remaining_runs` The number of runs remaining
- `first_run` The date and time of the first run as an ISO8601 datetime
- `last_run_at` The date and time of the last run as an ISO8601 datetime (changed in 2.4 from `last_run`)
- `next_run` The date and time of the next run as an ISO8601 datetime
- `task` The name (also the python path) of the task that will be executed

Scheduled tasks may have additional fields that are specific to that particular task.

Sample scheduled task resource

```
{
  "next_run": "2014-01-28T16:33:26Z",
  "task": "pulp.server.tasks.consumer.update_content",
  "last_updated": 1390926003.828128,
  "first_run": "2014-01-28T10:35:08Z",
  "schedule": "2014-01-28T10:35:08Z/P1D",
  "args": [
    "me"
  ],
  "enabled": true,
  "last_run_at": null,
  "_id": "52e7d8b3dd01fb0c8428b8c2",
  "total_run_count": 0,
  "failure_threshold": null,
  "kwargs": {
    "units": [
      {
        "unit_key": {
          "name": "pulp-server"
        },
        "type_id": "rpm"
      }
    ],
    "options": {}
  },
  "resource": "pulp:consumer:me",
  "remaining_runs": null,
  "consecutive_failures": 0,
  "options": {},
  "_href": "/pulp/api/v2/consumers/me/schedules/content/update/52e7d8b3dd01fb0c8428b8c2/"
}
```

2.4 Policies

2.4.1 Compatibility

Python Version

All server components must support Python 2.6. This includes importers, distributors, managers, middleware, and related code.

All client-side components must support python 2.4. This includes extensions and handlers.

2.4.2 Style

PEP-8

New Pulp code should adhere as closely as is reasonable to PEP-8. One modification is that our line length limit is 100 characters, which we chose to prevent line wrapping on GitHub.

In-code Documentation

Document your functions using the markup described here. If it's worth having a function, it's worth taking 1 minute to describe what it does, define each parameter, and define its return value. This saves a tremendous amount of time for the next person who looks at your code.

Include reasonable in-line comments where they might be helpful.

Naming

Use meaningful names.

Bad:

```
def update(t, n, p):
```

Good:

```
def update(title, name, path):
```

Be mindful of the global namespace, and don't collide with builtins and standard library components. For example, don't name anything "id", "file", "copy" etc.

Indentation

4 spaces, never tabs

Encoding

Specify UTF-8 encoding in each file:

```
# -*- coding: utf-8 -*-
```

2.4.3 Testing

Conventions

Unit tests are found under the `test/unit` directory in each subproject. Within the `test/unit` directory are two subdirectories:

- **data** - Holds any data files used by the unit tests
- **unit** - Holds the unit tests themselves.

Within the **unit** directory the tests are to be organized in a directory structure that matches the directory structure of the module they are testing. The test module itself shall be named test_<module_name>.py where <module_name> matches the module that is being tested.

For example: If the module being tested is `/common/pulp/common/plugins/progress.py` the corresponding unit test module would be `/common/test/unit/common/plugins/test_progress.py`

Unit tests may use the database but should not make any other external connections, such as to an external repository or the message bus.

Test cases are required for all submitted pull requests.

Python Libraries

The Pulp project uses the `mock` library as its mocking framework. More information on the framework can be found here: http://pypi.python.org/pypi/mock

Tests should not introduce any extra libraries in order to keep the test environment light-weight.

Testing Utilities

Each project contains a number of testing utility modules under the `test/unit` directory. Of particular interest is the module named `base.py`. This module provides a number of test case base classes that are used to simulate the necessary state of the different Pulp components. For example, in the platform, base classes are provided that set up the state for client tests (`PulpClientTests`) or REST API tests (`PulpWebserviceTests`). Applicability and usage information can be found in the docstrings for each class.

The above base classes should only be used in the event they are needed. The added setup and tear down time should be avoided in the event that those features are not needed, in which case simply subclassing `unittest.TestCase` is preferred.

If a there are any utilities that are used by many test modules they should be placed in the `devel` subproject. This subproject contains a number of mock objects and utilities that are used by unit tests in all of the pulp projects.

Compatibility

Each unit test directory contains a subdirectory called `server`. This distinction is due to the fact that the Pulp client must be Python 2.4 compatible whereas the server need only be Python 2.6 compatible. To keep the project's continuous integration tests against 2.4 from failing, the server tests are not run in those environments. More information on supported versions can be found on our Compatibility page.

This structure is only present in git repositories that have not yet been migrated into a multiple Python package format. In the latter case, the division between server and client code is expressed by the packages themselves and thus this construct is unnecessary.

Coverage

Each Pulp git repository contains a script named `run-tests.py`. This script will run all of the unit tests for that repository and generate coverage reports. The python `coverage` library is used to produce the reports and must be installed before running that script. An HTML version of the coverage report is created in the git repository root under `coverage/index.html`.

2.4.4 Versioning

This version policy closely follows the Semantic Versioning model. Formatting of pre-release designations and build information does not exactly follow the Semantic Versioning model, but instead follows established standards for Python and RPM packaging.

Python Package Version

The version of Pulp's Python packages should follow the PEP-386 scheme for setuptools utilizing a "X.Y.Z" pattern as major, minor and patch versions respectively.

Alpha, beta, and release candidate versions will be designated by a single character "a", "b" or "c" after the patch number, for example "2.1.0a".

Pulp is not currently distributed as Python packages, but instead as RPM packages. Thus, there may not be an effort to distinguish between different versions of an alpha, beta or release candidate. Instead, the version is likely to stay at something like "2.1.0a" for some period of time before moving to "2.1.0b", etc. Put another way, pre-release version numbers may not be incremented regularly.

RPM Package Version

Pulp's RPM packages should follow the version scheme prescribed by Fedora.

This scheme is very similar to the Python version scheme, except the pre-release designations go in the release field.

For all pre-release versions, the first item of the "release" field will be "0". The second item will increment with each build. The third item will be one of "alpha", "beta", and "rc".

For each released version, the "release" field will begin at "1" and increment with new builds of the same version.

Lifecycle Example

Stage	Python	RPM
Release	2.0.0	2.0.0-1
New Build	2.0.0	2.0.0-2
Bug Fix Beta	2.0.1b	2.0.1-0.1.beta
Bug Fix Release	2.0.1	2.0.1-1
Begin Minor Release	2.1.0a	2.1.0-0.1.alpha
More Work	2.1.0a	2.1.0-0.2.alpha
Feature Complete	2.1.0b	2.1.0-0.3.beta
More Work	2.1.0b	2.1.0-0.4.beta
Release Candidate	2.1.0c	2.1.0-0.5.rc
Release Candidate	2.1.0c	2.1.0-0.6.rc
Release	2.1.0	2.1.0-1

2.4.5 Release Management

This page explains processes that should be carried out by a Release Manager only.

Branch Management

To create a new release branch, follow these steps.

1. Create a new branch from master called “pulp-x.y”.
2. In master, bump all versions to x.(y+1).

Work can now continue as normal. Bug and feature branches can branch from the “pulp-x.y” branch and be merged into both “pulp-x.y” and master.

When it is time to release version x.y.(z+1), follow these steps. There should be a branch freeze during this operation.

1. make sure pulp-x.y is fully merged into master.
2. Follow the release steps, which will bump version numbers and grow the change log.
3. Merge pulp-x.y into master (and every release branch greater than x.y) using the “ours” strategy. This ignores the file changes from step 2, but makes sure that pulp-x.y is still fully merged into master. To protect from a race condition where someone might commit or merge to pulp-x.y between steps 1 and 3, merge a specific commit instead of a branch name. For example, `git show pulp-2.0` will show a commit ID on the first line, and then you can merge that commit as shown below.

```
$ git checkout master
$ git merge -s ours b41f5b49
```

Note: The “ours” strategy merges from the specified branch, but it ignores all changes from that branch. Thus, each change set from pulp-x.y will become a part of the master branch’s history, but the actual code changes that took place will not be applied to the master branch.

If step 3 were not performed, every future branch of pulp-x.y would have a conflict with master over the version and changelog. Merging immediately after the release with the “ours” strategy effectively resolves that conflict.

2.4.6 Building RPMs

Pulp repositories are configured to be built with the tito tool. Each build is done against a git tag. The steps are done per git repository and are as follows:

```
$ cd <git repo root>
$ tito tag
$ git push && git push --tags
$ tito build --rpm
```

The tito output will indicate the directory the RPMs are built into. The `--srpm` flag can be passed to tito to request SRPMs be built as well.

Note: When testing spec file changes, it is suggested to fork the Pulp repositories into your personal GitHub account. That way, any tags created will not affect the Pulp repositories themselves.

These policies apply to code that is supported by the Pulp Team. Third-party code can of course do as it may, but you are encouraged to follow these policies so that:

- Other Pulp developers will have an easy time working with your code.
- Your code is compatible with existing Pulp deployments.
- There is the option for your code to one day become an official part of the Pulp Project, and thus supported by the Pulp Team.

2.5 Implementing Support for New Types

2.5.1 Server Plugins

Introduction

There are three components that can be used in a server side plugin:

- *Type definitions* are used to tell Pulp about the types of content that will be stored. These definitions contain metadata about the type itself (e.g. name, description) and clues about the structure of the unit, such as uniqueness information and search indexes. Pulp uses that information to properly configure its internal storage of the unit to be optimized for the expected usage of the type. Type definitions are defined in a JSON document.
- *Importers* are used to handle the addition of content to a repository. This includes both synchronizing content from an external source or handling user-uploaded content units. An importer is linked to one or more type definitions to describe the types of units it will handle. At runtime, an importer is attached to a repository to provide the behavior of that repository's sync call. Importers are Python code that is run by the server.
- *Distributors* are added to a repository to publish its content. The definition of publish varies depending on the distributor and can include anything from serving the repository's content over HTTP to generating an RSS feed with information about the repositories contents. One or more distributors may be attached to a repository, allowing the repository to be exposed over a number of different mechanisms. Like importers, distributors are Python code executed on the server.

Documentation

Details on each server-side component can be found in the pages below:

Type Definitions

Overview A type definition is used to configure Pulp to support inventorying a type of content unit, such as an RPM or a Puppet module. Pulp uses the data in the definition to configure the database storage for those units with uniqueness constraints and to optimize queries relevant to that type.

Attributes Each type definition contains the following attributes.

`id` Programmatic identifier for the content type. The ID must be unique across all type definitions installed on the Pulp server.

`display_name` User-friendly name of the type.

`description` User-friendly description of the type.

`unit_key` List of all attributes that will be in units of this type that, when combined, represent the unique key for a unit. Pulp will enforce the uniqueness for units of this type based on this attribute.

`search_indexes` List of added non-unique indexes to add for storing units of the type. Each entry in the list may itself be another list to represent a compound index.

`referenced_types` List of type IDs for other types that are related to the type being defined. This nature of relationship is not explicitly defined; depending on the types of units involved it may be parent/child, dependent units, or something else. Pulp uses this information when the importer indicates to link a unit with another.

The `unit_key` attribute creates one or more indexes in the database as well. Given a value of [”a”, “b”, “c”], the following indexes are automatically created and need not be specified in the `search_indexes` field:

- a
- a, b
- a, b, c

Note that neither an index on just “b” nor the index “b, c” are automatically created.

Format Type definitions are defined in a JSON file. Multiple types may be defined in a single file. The file must be placed in the `/usr/lib/pulp/plugins/types` directory and has no restrictions on its name.

Installation Once the type definition file is in the appropriate directory, the `pulp-manage-db` script must be run to install the type. This script should also be run after making any changes to the type definition.

Sample Below is a sample type definition file, taken from the Puppet support bundle.

```
{"types": [
    {
        "id" : "puppet_module",
        "display_name" : "Puppet Module",
        "description" : "Puppet Module",
        "unit_key" : ["name", "version", "author"],
        "search_indexes" : ["author", "tag_list"]
    }
]}
```

This file installs a single type that is referenced by the id “puppet_module”. Each inventoried module will have a unique tuple of name, version, and author in its metadata. In addition to the indexes created by the unit key, indexes will be created on the “author” and “tag_list” attributes in each unit.

Migrations

From time to time, you might wish to adjust the schema of your database objects. The Pulp platform provides a migration system to assist you with this process. In this section of the guide, we will discuss how to configure your project's migrations.

Registration In order to write migrations for your Pulp plugin, you will need to register your plugin's migrations package with the Pulp server.

How to Register There are a few steps you will need to perform in order to configure your project to advertise itself to Pulp's migration system. First you will need to create a migrations Python package in your project's plugin space. For example, the Pulp RPM project has its migrations at `pulp_rpm.migrations`. You don't have to call it "migrations", but that's a reasonable choice of name.

Second, you will need to use the Python entry points system to advertise your migration package to Pulp. To do that, add an entry_points argument to in your *setup()* function in your setup.py file, like this:

```
    setup(<other_arguments>, entry_points = {
    <other_entry_points>,
    'pulp.server.db.migrations': [
        '<your_project_name> = <path.to.migrations.package>'
    ]
})
```

It's important that the entry point name "pulp.server.db.migrations" be used here. To clarify this with an example, the Pulp RPM project's setup.py has this as it's entry_points setup argument:

```
    entry_points = {
    'pulp.distributors': [
        'distributor = pulp_rpm.plugins.distributors.iso_distributor.distributor:entry_point',
    ],
    'pulp.importers': [
        'importer = pulp_rpm.plugins.importers.iso_importer.importer:entry_point',
    ],
    'pulp.server.db.migrations': [
        'pulp_rpm = pulp_rpm.migrations'
    ]
}
```

Once you have that in your *setup()* function, you will need to install your package using your setup.py file. This will advertise your package's migrations to Pulp, and you will be registered with Pulp's migration system. Once you have installed your package, you should run `pulp-manage-db` as the same user that apache runs as, and you should see some output that mentions your migration package:

```
$ sudo -u apache pulp-manage-db
Beginning database migrations.
Migration package pulp.server.db.migrations is up to date at version 2
Migration package pulp_rpm.migrations is up to date at version 4
Migration package <path.to.migrations.package> is up to date at version 0
Database migrations complete.
Loading content types.
Content types loaded.
```

It should say that your package is at version 0, because you haven't written any migrations yet. We'll talk about that next.

Creating Migrations In the event that you need to make an adjustment to your data in Pulp, you should write a migration script. There are a few rules to follow for migration scripts, and if you follow them carefully, nobody gets hurt. Here are the rules:

1. Migration scripts should be modules in your migrations package.
2. Each migration module should be named starting with a version number.
3. Your migration version numbers are significant. Pulp tracks which version each install has been migrated to. It requires your migration versions to start with 1, and to be sequential with no gaps in version numbers. For example, 0001_my_first_migration.py, 0002_my_second_migration.py, 0003_add_email_addresses_to_users.py, etc. You don't have to use leading zeros in the names, as the number is processed with a regular expression that interprets it as an integer. However, the advantage to using leading zeros is that programs like `ls` will display your migrations in order when you inspect the contents of your migration package.
4. Each migration module should have a function called "migrate" with this signature: `def migrate(*args, **kwargs)`.
5. Inside your `migrate()` function, you can perform the necessary work to change the data in the Pulp install.
6. Your `migrate()` function must be written in such a way that it will not fail for a new installation. New installations will start at migration version 0, and all migrations up to the most recent migration will be applied by the system. Therefore, you must not assume that there is data in the database, or on the filesystem. Your migration script should detect what work, if any, needs to be done before performing any operations.

For example, your migrations package might look like this:

```
migrations
|
|-- __init__.py
|-- 0001_rename_user_to_username.py
|-- 0002_remove_spaces_from_username.py
|-- 0003_recalculate_unit_hashes.py
```

Here's what the first migration, 0001_rename_user_to_username.py, might look like:

```
# Getting the db handle is left as an exercise for the reader
from somewhere import initialize_db

def migrate(*args, **kwargs):
        """
        We want to rename the 'user' attribute in our users collection to 'username' for clarity.
        """
        db = initialize_db()
        db.users.update({}, {'$rename': {'user': 'username'}})
```

Importers

Overview The fundamental role of an importer is to bring new units into a Pulp repository. The typical method is the sync process through which the contents of an external source (yum repository, Puppet Forge, etc.) are downloaded and inventoried on the Pulp server. Additionally, the importer is also responsible for handling uploaded units (inventorying and persistence on disk) and any logic involved with copying units between repositories.

Operations cannot be performed on an importer until it is attached to a repository. When adding an importer to a repository, the importer's configuration will be stored in the Pulp server and provided to the importer in each operation. More information on how this configuration functions can be found in the *configuration section* of this guide.

Only one importer may be attached to a repository at a time.

The Plugin Conventions page describes behavior and APIs common to both importers and distributors.

Note: Currently, the API for the base class is not published. The code can be found at `Importer` in `platform/src/pulp/plugins/importer.py`.

Implementation Each importer must subclass the `pulp.plugins.importer.Importer` class. That class defines the operations an importer may be requested to perform on a repository. Not every method must be overridden in the subclass. Some, such as the *lifecycle methods* will have no effect. Others, such as `upload_unit`, will raise an exception indicating the operation is not supported by the importer if not overridden.

Warning: The importer instance is not reused between invocations. Any state maintained in the importer is only valid during the current operation's execution. If state is required across multiple operations, the *plugin's scratchpad* should be used to store the necessary information.

There are two methods in the `Importer` class that must be overridden in order for the importer to work:

Metadata The importer implementation must implement the `metadata` method as *described here*.

Configuration Validation The importer implementation must implement the `validate_config` method as *described here*.

Functionality There are a number of abilities an importer implementation can support. All of these are optional; it is possible to have an importer that handles uploaded units but has no support for synchronizing against an external repository.

The sections below will cover an overview of each feature. More information on the specifics of how to implement them are found in the docstrings for each method.

Warning: Importers that implement a sync method must also implement support for cancelling the sync.

Synchronize an External Respository Methods: `sync_repo`, `cancel_sync_repo`

One of the most common uses of an importer is to download content from an external source and inventory it in the Pulp server. The importer serves as an adapter between the Pulp server and the external repository, using whatever protocols are necessary.

While the importer is responsible for downloading the unit, it is up to Pulp to determine the absolute path on disk to store it. The importer provides a relative path for where it would like to store the unit, taking into account enough information to create a unique path. This is passed to the conduit's `init_unit` call which allows Pulp to derive the absolute path on the server to store it. The path will be in the returned `pulp.plugins.model.Unit` object in the `storage_path` attribute.

Plugin implementations for repository sync will obviously vary wildly. Below is a short outline of a common sync process.

1. Call the conduit's `get_units` method to understand what units are already associated with the repository being synchronized.
2. For each new unit to add to the Pulp server and associate with the repository, the plugin takes the following steps:
   (a) Calls the conduit's `init_unit` which takes unit specific metadata and allows Pulp to populate any calculated/derived values for the unit. The result of this call is an object representation of the unit.

   (b) Uses the `storage_path` field in the returned unit to save the bits for the unit to disk.
   (c) Calls the conduit's `save_unit` which creates/updates Pulp's knowledge of the content unit and creates an association between the unit and the repository
   (d) If necessary, calls the conduit's `link_unit` to establish any relationships between units.

3. For units previously associated with the repository (known from `get_units`) that should no longer be, calls the conduit's `remove_unit` to remove that association.

Note: It is valid for a unit to be purely metadata and not have a corresponding file. In these cases, simply specify a relative path of `None` to the `init_unit` call and ignore the step about using the `storage_path`.

The conduit defines a `set_progress` call that should be used throughout the process to update the Pulp server with details on what has been accomplished and what remains to be done. The Pulp server does not require these calls. The progress message must be JSON-serializable (primitives, lists, dictionaries) but is otherwise entirely at the discretion of the plugin writer. The most recent progress report is saved in the database and made available to users as a means to track the progress of the sync.

When implementing the sync functionality, the importer's `cancel_sync_repo` method must be implemented as well. This call will be made on the same instance performing the sync, therefore it is valid to use an instance variable as a flag the sync process uses to determine if it should continue to proceed.

Upload Units Method: `upload_unit`

The Pulp server provides the infrastructure for users to upload units into a repository. It is the job of the importer to take the steps necessary to:

- Generate and save the inventoried representation of the unit.
- Determine the appropriate relative path at which to store the unit.
- Move the unit from the provided temporary location to the final storage path as provided by Pulp.

The conduit provides the `init_unit` and `save_unit` calls as described in *Synchronize an External Respository*. Refer to that section for more information on usage.

Import Units Method: `import_units`

The Pulp server provides an API for selecting units to copy from one repository to another. The importer's `import_units` method is called on the **destination repository** to handle the copy.

There are two approaches to handling this method:

- In most cases, the unit can be shared between the two repositories. A new association is created between the destination repository and the original database representation of the unit. This approach is accomplished by simply calling the conduit's `save_unit` method for each unit to be copied.
- In certain cases, the same unit cannot be safely referenced by both repositories. A new unit must be created using the `init_unit` method and then saved to the repository with `save_unit` in the same way as in *Synchronize an External Respository*.

Note: Take note if which attributes on the unit are required for use when importing. It is then possible to specify in the associate request's *unit association criteria* which fields should be loaded, which result in reduced RAM use during the import process, especially for units with a lot of metadata.

Remove Units Method: `remove_units`

When a user unassociates units from a repository, the Pulp server will make the necessary database changes to reflect the change. The `remove_units` method is called on the repository's importer to allow the importer to perform any clean up steps is may need to make, such as removing any data it may have been storing about the unit from the working directory. In most cases, this method does not need to be overridden.

Warning: This call should not remove the unit from its final location specified by Pulp. Pulp will handle the deletion of the file itself during its orphan clean up process.

Distributors

Overview While an importer is responsible for bringing content into a repository, a distributor is used to expose that content from the Pulp server. The specifics for what it means to expose the repository, performed through an operation referred to as *publishing*, is dependent on the distributor's goals. Publishing examples include serving the repository over HTTP/HTTPS, packaging it as an ISO, or using rsync to transfer it into a legacy system.

Operations cannot be performed on a distributor until it is attached to a repository. When adding a distributor to a repository, the distributor's configuration will be stored in the Pulp server and provided to the distributor in each operation. More information on how this configuration functions can be found in the *configuration section* of this guide.

Multiple distributors may be associated with a single repository at one time. When publishing the repository, the user selects which distributor to use.

The Plugin Conventions page describes behavior and APIs common to both importers and distributors.

Note: Currently, the API for the base class is not published. The code can be found at `Distributor` in `platform/src/pulp/plugins/distributor.py`.

Implementation Each distributor must subclass the `pulp.plugins.distributor.Distributor` class. That class defines the operations a distributor may be requested to perform on a repository.

Warning: The distributor instance is not reused between invocations. Any state maintained in the distributor is only valid during the current operation's execution. If state is required across multiple operations, the *plugin's scratchpad* should be used to store the necessary information.

There are two methods in the `Distributor` class that must be overridden in order for the distributor to work:

Metadata The distributor implementation must implement the `metadata()` method as *described here*.

Configuration Validation The distributor implementation must implement the `validate_config` method as *described here*.

Functionality The primary role of a distributor is to publish a repository. Optionally, the distributor can provide information to be automatically sent to consumers when they are bound to it.

The sections below will cover a high-level overview of the distributor's functionality. More information on the specifics of how to implement them are found in the docstrings for each method.

Warning: Both the `publish_repo` and `cancel_publish_repo` methods must be implemented together.

Publish a Repository Methods: `publish_repo`, `cancel_publish_repo`

The distributor's role in publishing a repository is to take the units currently in the repository and make them available outside of the Pulp server. The approach for how that is done will vary based on needs. The typical approach is to serve the repository over HTTP/HTTPS. However, it is also possible to use a variety of other protocols depending on the nature of the content being served or the specific needs of a deployment.

The *conduit* passed to the publish call provides the necessary methods to query the content in a repository. In the event a directory of the repository's content must be created, it is highly recommended to symlink from the unit's `storage_path` rather than copying it.

The conduit defines a `set_progress` call that should be used throughout the process to update the Pulp server with details on what has been accomplished and what remains to be done. The Pulp server does not require these calls. The progress message must be JSON-serializable (primitives, lists, dictionaries) but is otherwise entirely at the discretion of the plugin writer. The most recent progress report is saved in the database and made available to users as a means to track the progress of the publish.

When implementing the publish functionality, the importer's `cancel_sync_repo` method must be implemented as well. This call will be made on the same instance performing the publish, therefore it is valid to use an instance variable as a flag the publish process uses to determine if it should continue.

Consumer Payloads Method: `create_consumer_payload`

Depending on the distributor's implementation, it is possible that certain information needs to be given to consumers attempting to use it. For example, if a distributor supports multiple protocols such as HTTP and HTTPS, the consumer needs to know which protocol a given repository is configured to use. This information is referred to as a *consumer payload*.

Each time a consumer binds to a repository's distributor, the `create_consumer_payload` method is called. The format of the payload is up to the plugin writer.

Hosting Static Content You may host static content within the `/pulp` URL path. The convention with existing plugins is to allow content to be published over http, https, or both by symlinking content into corresponding directories on the filesystem. To accomplish this, you must create a basic Apache configuration.

Most of your configuration can go in a standard Apache config file like this one:

```
# /etc/httpd/conf.d/pulp_puppet.conf

# SSL-related directives can go right in the global config space. The
# corresponding non-SSL Alias directive must go in a separate config file.
Alias /pulp/puppet /var/www/pulp_puppet/https/repos

# directory where repos published for HTTPS get symlinked
<Directory /var/www/pulp_puppet/https/repos>
    Options FollowSymLinks Indexes
</Directory>

# directory where repos published for HTTP get symlinked
<Directory /var/www/pulp_puppet/http/repos>
    Options FollowSymLinks Indexes
</Directory>
```

However, directives such as Alias statements that are specific to the `<VirtualHost *:80>` block provided by the platform must go in a separate file. All files within `/etc/pulp/vhosts80/` have their directives "Included" in one `<VirtualHost *:80>` block.

```
# /etc/pulp/vhosts80/puppet.conf

# Directives in this file get included in the one authoritative
# <VirtualHost *:80> block provided by the platform.
Alias /pulp/puppet /var/www/pulp_puppet/http/repos
```

Plugin Conventions

Configuration

Validation It is up to the plugin writer to determine what configuration values are necessary for the plugin to function.

Pulp performs no validation on the configuration for a plugin. The `validate_config` method in each plugin subclass is used to verify the user-entered values for a repository. This is called when the plugin is first added to the repository and on all subsequent configuration changes. The configuration is sent to the Pulp server as a JSON document through its REST APIs and will be deserialized before being passed to the plugin.

This call must ensure the configuration to be used when running the plugin will be valid for the repository. If this call indicates an invalid configuration, the plugin will not be added to the repository (for the add call) or the configuration changes will not be saved to the database (for the update configuration call).

The docstring for the method describes the format of the returned value.

Format Each call into a plugin passes the configuration the call should use for its execution. The configuration is contained in a `pulp.plugins.config.PluginCallConfiguration` instance. This object is a wrapper on top of the three different locations a configuration value can come from:

- **Overrides** - Most calls allow the user to specify configuration values as a parameter when they are invoked. These values are made available to the plugin for the operation's execution, however they are not saved in the server.
- **Repository-level** - When an importer or distributor is attached to a repository, the Pulp server saves the configuration for that plugin with the repository. These values are only used for operations on that repository. For example, if an importer is configured to synchronize from an external feed, the URL of that feed would be stored on a per repository basis.
- **Plugin-level** - Each importer and distributor may be paired with a static configuration file on disk. These are JSON files that are loaded by the Pulp server when the plugins are initialized. Configuration values in this location are available to all instances of the importer/distributor.

The `PluginCallConfiguration` defines a method called `get(str)` that will retrieve the value for the given key. This call will check the three configuration locations in the order listed above. The first value found for the key is returned, removing the need for the plugin writer to apply this prioritization on their own.

Life Cycle Methods Both types of plugins define a number of methods related to the lifecycle of the plugin on a particular repository. These methods are called when the importer/distributor is added to or removed from a repository. Examples include `importer_added(repo, config` and `distributor_removed(repo, config).`

In many cases, these methods can be ignored. The default implementation will not raise an error. Their usage is typically to perform any initialization in the plugin's *working directory* that is necessary before the first plugin operation is invoked.

Metadata Method Both types of plugins require a metadata method to be overridden from the base class. The `metadata()` method is responsible for providing Pulp with information on how the plugin works. The following information must be returned from the metadata call. The docstring for the method describes the format of the returned value.

- **ID** - Unique ID that is used to refer to this type of plugin. This must be unique for all plugins installed in the Pulp server.
- **Display Name** - User-friendly description of what the plugin does.
- **Supported Types** - List of IDs for all content types that may be handled by the plugin. If there is no type definition found for any of the IDs referenced here, the server will fail to start.

Conduits A *conduit* is an object passed to a plugin when an method is executed. The conduit is used to access functionality in the Pulp server itself. Each method is given a custom conduit type depending on the needs of the method being invoked. Consult the docstrings for each method in the plugin base class for more information on the conduit class that will be used.

Warning: Plugins should not retain any state between calls. Conduits are typically scoped to the repository being used; reusing old conduit instances can lead to data corruption.

Scratchpads A *scratchpad* is used to store information across multiple operations run by the plugin. Each importer and distributor on a repository is given its own scratchpad. A plugin may only edit its own scratchpad for the repository being acted on.

The scratchpad is retrieved through the conduit's `get_scratchpad()` method and updated with `set_scratchpad(object)`. The scratchpad is stored in the database, therefore its value must be able to be pickled. It is recommended to use either a single string or a dictionary of string pairs.

Additionally, there exists a scratchpad at the repository level, accessible to *all* importers and distributors on the repository. This can be used to share information between different plugins. It is highly recommended to avoid using this wherever possible so as to not tightly couple plugins together. The repository scratchpad can be accessed using `get_repo_scratchpad()` and `set_repo_scratchpad(object)` and carries the same pickle restriction as described above.

Working Directories Each plugin on a repository is given a unique location on disk. This directory should be used for storing any temporary files that need to be created when the plugin is used. These directories are automatically deleted when the repository is deleted. The location of the working directory can be found in the repository instance (`pulp.plugins.model.Repository`) passed into each plugin call.

Installation There are two ways to install a plugin.

Entry Points The plugin may define a method that will serve as its entry point. The method must accept zero arguments and return a tuple of the following:

- Class of the plugin itself. This must be a subclass of either `pulp.plugins.importer.Importer` or `pulp.plugins.distributor.Distributor`.
- Plugin-level configuration to use for that plugin. See *Configuration* for more information on the scope of these configuration values.

A sample is as follows:

```
def entry_point():
    return DemoImporter, {}

class DemoImporter(Importer):
    ...
```

Python entry points are advertised within the package's `setup.py` file. Multiple entry points may be advertised by the same setup file. A sample from the Puppet plugins is below:

```
from setuptools import setup, find_packages

setup(
    name='pulp_puppet_plugins',
    version='2.0.0',
    license='GPLv2+',
    packages=find_packages(exclude=['test', 'test.*']),
    author='Pulp Team',
    author_email='pulp-list@redhat.com',
    entry_points = {
        'pulp.distributors': [
            'distributor = pulp_puppet.plugins.distributors.distributor:entry_point',
        ],
        'pulp.importers': [
            'importer = pulp_puppet.plugins.importers.importer:entry_point',
        ],
    }
)
```

Directory Loading For one-off testing purposes, the code for a plugin can be placed directly in a specific directory without the need to install to site-packages. The entry point method described above is the preferred way to integrate new plugins:

- Create directory in `/usr/lib/pulp/plugins/` under the appropriate plugin type.
- Add `__init__.py` to created directory.
- Add `importer.py` or `distributor.py` as appropriate.
- In the above module, add the classes that subclass `Importer` or `Distributor` as appropriate.

Additionally, for directory loaded plugins, Pulp will automatically load any configuration files found in the plugin's directory. The configuration within will be made available to each call as described in *Configuration*. The only restriction on the name of the configuration file is that it end with `.conf` and be placed in the directory created in the first step above.

Plugin Example

This example will cover the structure of a plugin, covering the type definition, importer, and distributor.

In a real importer implementation, the details of how the contents of an external repository are retrieved and how the files are downloaded are non-trivial. Similarly, the steps a distributor will take to publish the repository will vary in complexity based on the behavior of the publish implementation. As such, these examples will provide the basic structure of the plugins with stubs indicating where more complex logic would occur.

Note: For the purposes of this example, the code will be added to the subclasses directly. In a real implementation, multiple Python modules would be used for better organization.

Type Definition Content types are defined in a JSON file. Multiple types may be defined in the same definition file. More information on a definition's fields can be found in the Type Definitions section.

This document will use a modified version of the Puppet module type definition as an example. This version is simplified to use only the module name as the unit key.

```
{"types": [
    {
        "id" : "puppet_module",
        "display_name" : "Puppet Module",
        "description" : "Puppet Module",
        "unit_key" : "name",
        "search_indexes" : ["author", "tag_list"]
    }
]}
```

The type definition must be placed in the `/usr/lib/pulp/plugins/types` directory. The `pulp-manage-db` script must be run each time a definition is added or changed.

Importer Each importer must subclass the `pulp.plugins.importer.Importer` class. The following snippet contains the definition of that class and its implementation of the required `metadata()` method. More information on this method can be found *here*.

```
from pulp.plugins.importer import Importer

class PuppetModuleImporter(Importer):

    @classmethod
    def metadata(cls):
        return {
            'id' : 'puppet_importer',
            'display_name' : 'Puppet Importer',
            'types' : ['puppet_module'],
        }
```

Note: User-visible information, such as the `display_name` attribute above, should be run through an i18n conversion method before being returned from this call.

The puppet_module content type in the `types` field correlates to the name of the type defined above.

The importer implementation is also required to implement the `validate_config` method as *described here*. Implementations will vary by importer. For this example, a simple check to ensure a feed has been provided will be performed. If the feed is missing, the configuration is flagged as invalid and a message to be displayed to the user is returned. If the feed is present, the method indicates the configuration is valid (no user message is required).

```
def validate_config(self, repo, config, related_repos):
  if config.get('feed') is None:
    return False, 'Required attribute "feed" is missing'

  return True, None
```

At this point, other methods in `Importer` are subclassed depending on the desired functionality. This example will cover the `sync_repos` method.

The implementation below covers a very high-level view of what a repository sync call will do. The conduit is used to query the server for the current contents of the repository and add new units. It is also used to update the server on the progress of the sync.

```
def sync_repo(self, repo, sync_conduit, config):

  sync_conduit.set_progress('Downloading repository metadata')
  metadata = self._fetch_repo_metadata(repo, config)
  sync_conduit.set_progress('Metadata download complete')

  new_modules = self._resolve_modules_to_download(metadata, sync_conduit)

  sync_conduit.set_progress('Downloading modules')
  self._download_and_add_modules(new_modules, sync_conduit)
  sync_conduit.set_progress('Module download and import complete')

def _fetch_repo_metadata(repo, config):
  """
  Retrieves the listing of Puppet modules at the configured 'feed' location. The data returned from
  this call will vary based on the implementation but will likely be enough to identify each
  module in the repository.

  :return: list of module names in the external repository
  :rtype:  list
  """
  # Insert download and parse logic
  modules_in_repository = # Parse logic

  return modules_in_repository

def _resolve_modules_to_download(metadata, sync_conduit):
  """
  Analyzes the metadata describing modules in the external repository against those already in
  the Pulp repository. The conduit is used to query the Pulp server for the repository's modules.

  Similar to _fetch_repo_metadata, the format of the returned value needs to be enough that
  the download portion of the process can fetch them.

  :return: list of module names that need to be downloaded from the external repository
  :rtype:  list
  """
  # Units currently in the repository
  module_criteria = UnitAssociationCriteria(type_ids=['puppet_module'])
  existing_modules = sync_conduit.get_units(criteria=module_criteria)

  # Calculate the difference between existing_units and what is in the metadata
  module_names_to_download = # Difference logic

  return module_names_to_download

def _download_and_add_modules(new_modules, sync_conduit):
  """
  Performs the downloading of any missing modules and adds them to the Pulp server.
  """

  for module_name in new_modules:
    # Determine the unique identifier for the unit. This should use each of the fields for
    # the unit key as specified in the type definition.
    unit_key = {'name' : module_name}

    # Any extra information about the module is specified as its metadata. This may include
    # file size, checksum, description, etc. For this example, we'll simply leave it empty.
    metadata = {}

    # The relative path is the path and filename of the module. This must be unique across
    # all Puppet modules. Pulp will prefix this path as necessary to make it a full path
    # on the filesystem the file should reside.
    relative_path = 'modules/%s' % module_name

    # Allow Pulp to package the unit and perform any initialization it needs. This
    # initialization includes calculating the full path it will be stored at. The return
    # from this call is a pulp.plugins.Unit instance.
    pulp_unit = sync_conduit.init_unit('puppet_module', unit_key, metadata, relative_path)

    # Download the file to the Pulp-specified destination.
    # Download logic into pulp_unit.storage_path

    # If the download was successful, save the unit in Pulp's database and associate it with
    # the repository being synchronized (the conduit is scoped to the repository so it need
    # not be specified explicitly).
    sync_conduit.save_unit(pulp_unit)
```

Distributor This example will loosely describe the process of exposing a Pulp repository over the local web server.

Each distributor must subclass the `pulp.plugins.distributor.Distributor` class. The following snippet contains the definition of that class and its implementation of the required `metadata()` method. More information on this method can be found *here*.

```
from pulp.plugins.distributor import Distributor

class PuppetModuleDistributor(Distributor):

    @classmethod
    def metadata(cls):
        return {
            'id' : 'puppet_distributor',
            'display_name' : 'Puppet Distributor',
            'types' : ['puppet_module'],
        }
```

As with the importer, the type definition is referenced in the metadata as a supported type.

Also similar to the importer, the distributor implementation is required to implement the `validate_config` method as *described here*. For this example, the validation will ensure that the distributor is configured to publish over at least HTTP or HTTPS.

```
def validate_config(self, repo, config, related_repos):
  if config.get('serve-http') is None and config.get('serve-https') is None:
    return False, 'At least one of "serve-http" or "serve-https" must be specified'

  return True, None
```

The `publish_repo` method is implemented to support the publishing operation.

The implementation below covers a very high-level view of what a repository publish call will do. The conduit is used to query the server for the current contents of the repository and to update the server on the progress of the sync.

```
def publish_repo(self, repo, publish_conduit, config):

  publish_conduit.set_progress('Publishing modules')
  self._publish_modules(publish_conduit, config)
  publish_conduit.set_progress('Modules published')

  publish_conduit.set_progress('Generating repository metadata')
  self._generate_metadata(publish_conduit, config)
  publish_conduit.set_progress('Metadata generation complete')

def _publish_modules(publish_conduit, config):
  """
  For each module in the repository, creates a symlink from the location at which Pulp
  saved the module to a web-enabled directory.
  """

  criteria = UnitAssociationCriteria(type_ids=['puppet_module'])
  repo_modules = self.publish_conduit.get_units(criteria=criteria)

  # Each entry is a pulp.plugins.module.Unit instance
  for module in repo_modules:

    if config.get('serve-http') is True:
      # Create symlink from module.storage_path to HTTP-enabled directory

    if config.get('serve-https') is True:
      # Create symlink from module.storage_path to HTTPS-enabled directory

def _generate_metadata(publish_conduit, config):
  """
  Creates the files necessary to describe the contents of the published repository. This may
  not be necessary in all distributors. In this example, we're recreating the Puppet Forge
  repository on the Pulp server, so the corresponding JSON metadata files are created.
  These files are recreated instead of simply copied from Puppet Forge as the contents
  of the repository may have changed, for instance if modules were uploaded or copied
  from another repository.
  """

  # Metadata file creation logic, using the conduit to retrieve the modules in the repository
```

Installation Instructions on packaging and installing plugins for production deployment can be found at *Entry Points*. For development purposes, it may be simpler to install the plugin using the directory approach. More information can be found in the *Directory Loading* section of this guide.

Full Example

Type Definition

```
{"types": [
    {
        "id" : "puppet_module",
        "display_name" : "Puppet Module",
        "description" : "Puppet Module",
        "unit_key" : "name",
        "search_indexes" : ["author", "tag_list"]
    }
]}
```

Importer

```
from pulp.plugins.importer import Importer

class PuppetModuleImporter(Importer):

  @classmethod
  def metadata(cls):
      return {
          'id' : 'puppet_importer',
          'display_name' : 'Puppet Importer',
          'types' : ['puppet_module'],
      }

 def validate_config(self, repo, config, related_repos):
   if config.get('feed') is None:
     return False, 'Required attribute "feed" is missing'

   return True, None

 def sync_repo(self, repo, sync_conduit, config):

   sync_conduit.set_progress('Downloading repository metadata')
   metadata = self._fetch_repo_metadata(repo, config)
   sync_conduit.set_progress('Metadata download complete')

   new_modules = self._resolve_modules_to_download(metadata, sync_conduit)

   sync_conduit.set_progress('Downloading modules')
   self._download_and_add_modules(new_modules, sync_conduit)
   sync_conduit.set_progress('Module download and import complete')

 def _fetch_repo_metadata(repo, config):
   """
   Retrieves the listing of Puppet modules at the configured 'feed' location. The data returned from
   this call will vary based on the implementation but will likely be enough to identify each
   module in the repository.

   :return: list of module names in the external repository
   :rtype:  list
   """
   # Insert download and parse logic
   modules_in_repository = # Parse logic

   return modules_in_repository

 def _resolve_modules_to_download(metadata, sync_conduit):
   """
   Analyzes the metadata describing modules in the external repository against those already in
   the Pulp repository. The conduit is used to query the Pulp server for the repository's modules.

   Similar to _fetch_repo_metadata, the format of the returned value needs to be enough that
   the download portion of the process can fetch them.

   :return: list of module names that need to be downloaded from the external repository
   :rtype:  list
   """
   # Units currently in the repository
   module_criteria = UnitAssociationCriteria(type_ids=['puppet_module'])
   existing_modules = sync_conduit.get_units(criteria=module_criteria)
```

```
    # Calculate the difference between existing_units and what is in the metadata
    module_names_to_download = # Difference logic

    return module_names_to_download

  def _download_and_add_modules(new_modules, sync_conduit):
    """
    Performs the downloading of any missing modules and adds them to the Pulp server.
    """

    for module_name in new_modules:
      # Determine the unique identifier for the unit. This should use each of the fields for
      # the unit key as specified in the type definition.
      unit_key = {'name' : module_name}

      # Any extra information about the module is specified as its metadata. This may include
      # file size, checksum, description, etc. For this example, we'll simply leave it empty.
      metadata = {}

      # The relative path is the path and filename of the module. This must be unique across
      # all Puppet modules. Pulp will prefix this path as necessary to make it a full path
      # on the filesystem the file should reside.
      relative_path = 'modules/%s' % module_name

      # Allow Pulp to package the unit and perform any initialization it needs. This
      # initialization includes calculating the full path it will be stored at. The return
      # from this call is a pulp.plugins.Unit instance.
      pulp_unit = sync_conduit.init_unit('puppet_module', unit_key, metadata, relative_path)

      # Download the file to the Pulp-specified destination.
      # Download logic into pulp_unit.storage_path

      # If the download was successful, save the unit in Pulp's database and associate it with
      # the repository being synchronized (the conduit is scoped to the repository so it need
      # not be specified explicitly).
      sync_conduit.save_unit(pulp_unit)
```

Distributor

```
from pulp.plugins.distributor import Distributor

class PuppetModuleDistributor(Distributor):

  @classmethod
  def metadata(cls):
      return {
          'id' : 'puppet_distributor',
          'display_name' : 'Puppet Distributor',
          'types' : ['puppet_module'],
      }

 def validate_config(self, repo, config, related_repos):
   if config.get('serve-http') is None and config.get('serve-https') is None:
     return False, 'At least one of "serve-http" or "serve-https" must be specified'

   return True, None
```

```
def publish_repo(self, repo, publish_conduit, config):

  publish_conduit.set_progress('Publishing modules')
  self._publish_modules(publish_conduit, config)
  publish_conduit.set_progress('Modules published')

  publish_conduit.set_progress('Generating repository metadata')
  self._generate_metadata(publish_conduit, config)
  publish_conduit.set_progress('Metadata generation complete')

def _publish_modules(publish_conduit, config):
  """
  For each module in the repository, creates a symlink from the location at which Pulp
  saved the module to a web-enabled directory.
  """

  criteria = UnitAssociationCriteria(type_ids=['puppet_module'])
  repo_modules = self.publish_conduit.get_units(criteria=criteria)

  # Each entry is a pulp.plugins.module.Unit instance
  for module in repo_modules:

    if config.get('serve-http') is True:
      # Create symlink from module.storage_path to HTTP-enabled directory

    if config.get('serve-https') is True:
      # Create symlink from module.storage_path to HTTPS-enabled directory

def _generate_metadata(publish_conduit, config):
  """
  Creates the files necessary to describe the contents of the published repository. This may
  not be necessary in all distributors. In this example, we're recreating the Puppet Forge
  repository on the Pulp server, so the corresponding JSON metadata files are created.
  These files are recreated instead of simply copied from Puppet Forge as the contents
  of the repository may have changed, for instance if modules were uploaded or copied
  from another repository.
  """

  # Metadata file creation logic, using the conduit to retrieve the modules in the repository
```

2.5.2 Client Extensions

Overview

Both the Pulp Admin Client and the Pulp Consumer Client use an extension mechanism to allow additions and changes to be made depending on a developer's needs. For the plugin writer, these additions typically center around specific functionality for the types being supported by the plugin. For example, the configuration values for an *importer* are likely unique for each type of importer. Extensions are used to provide an interface catered to that specific configuration.

Documentation

Extensions

Overview The simplest way to describe the pieces of the client is to start with an example:

```
$ pulp-admin rpm repo list --details
```

In the above command:

- `pulp-admin` is the name of the client script itself.
- `rpm` and `repo` are *sections*. A section is used for organization and may contain one or more subsections and/or commands.
- `list` is the actual *command* that is being invoked.
- `--details` is a *flag* that is used to drive the command's behavior. Commands can accept flags or *options*, the difference being that the latter requires a value to be specified (e.g. `--repo-id demo-1`).

Extensions can add both sections and commands to the client. Commands can be added to existing sections, however it is typically preferred that commands be added to a section created by the extension itself (see *Conventions* for more information).

Framework Hook The starting point for an extension is a single method with the following signature:

```
def initialize(context):
  """
  Entry point into the extension, called when the extension is loaded. This
  call should make any additions or changes to the CLI required by the
  extension. The supplied context provides the necessary hooks to access the
  CLI instance as well as functionality for accessing the Pulp server and
  utilities for displaying information to the user.

  :param context: client context in which the extension is being loaded
  :type  context: pulp.client.extensions.core.ClientContext
  """
```

The sole argument is the client context in which the extension will run (more information can be found in *Client Context*. This method may then delegate to whatever other modules are necessary.

Client Context The client context is passed into the extension by the framework when the extension is initialized. The context is meant to provide all of the functionality necessary for the extension to interact with both the client framework and the server itself. The context contains the following pieces:

- `cli` - The cli attribute is the instance of the actual client framework itself. This object is used to add new sections to the client or retrieve existing ones.
- `prompt` - The prompt is a utility for writing output to the screen as well as for reading user input for interactive commands. A number of formatting methods are provided to provide a consistent look and feel for the client, such as methods to display a header or print an error message.
- `server` - The client framework creates and initializes an instance of the server bindings to connect to the configured Pulp server. Extensions should use these bindings when making calls against the server.
- `config` - The client framework will load all of the configuration files and make them accessible to the extension. A separate copy of the configuration is supplied to each extension, so changes may be made to this object without affecting other extensions.

Note: Currently, the API for the client context is not published. The code can be found at `ClientContext` in `client_lib/pulp/client/extensions/core.py`.

Conventions In order to prevent collisions among section names, it is suggested that each type support bundle create a section at the root of the CLI to contain its extensions. For example, when installed, the RPM support bundle adds a section called `rpm` in the root of the CLI. Commands within apply only to the plugins and consumers revolving around the RPM-related content types. The Puppet support bundle provides a similar structure under the root-level section `puppet`.

Naturally, there will be similarities between different support commands. Each support bundle will likely have commands to create a repository, customized with the configuration options relevant to that bundle's server plugins. Similarly, the command to sync a repository will be present in each bundle's section, with specific handling to render the progress report.

To facilitate those similarities, the `pulp.client.commands` package provides a number of reusable commands for common tasks, such as displaying a list of repositories or adding schedules for an operation. It is preferred that an extension subclass these reusable commands and use their integration points to customize them to your needs.

Note: Again, the API for these reusable commands is not yet published. The code can be found under `client_lib/pulp/client/commands`.

Installation

Entry Points Your extension should define a method that will be an entry point. It should accept one argument and return `None`. The convention is to use this definition:

```
from pulp.client.extensions.decorator import priority

@priority()
def initialize(context):
    """
    :type context: pulp.client.extensions.core.ClientContext
    """
    pass
```

The ClientContext instance includes a reference to everything you need to add new commands and sections to the CLI. Look at the Puppet Extensions for an example of how to add features to the CLI.

The `@priority()` decorator controls the order in which this extension will be loaded relative to other extensions. By not passing a value, this example accepts the default priority level. The default is found in `pulp.client.extensions.loader.DEFAULT_PRIORITY`, and its value is 5 as of this writing.

Python entry points are advertised within the package's `setup.py` file. As an example, here is that file from the Pulp Puppet Extensions Admin package.

```
from setuptools import setup, find_packages

setup(
    name='pulp_puppet_extensions_admin',
    version='2.0.0',
    license='GPLv2+',
    packages=find_packages(exclude=['test', 'test.*']),
    author='Pulp Team',
    author_email='pulp-list@redhat.com',
    entry_points = {
        'pulp.extensions.admin': [
            'repo_admin = pulp_puppet.extensions.admin.repo.pulp_cli:initialize',
        ]
    }
)
```

Notice that the entry point name is `pulp.extensions.admin`. That distinguishes it as an extension for the admin client. A consumer extension would use the name `pulp.extensions.consumer`. Technically these names could be changed, or new ones could be used if new CLI tools are developed. The "admin" and "consumer" portions of these names come from config files `/etc/pulp/admin/admin.conf` and `/etc/pulp/consumer/consumer.conf`. Each has a "[client]" section with a "role" setting. That said, the intent is for these to stay the same, and it is sufficient to assume that they will.

Directory Loading For one-off testing purposes, the code for an extension can be placed directly in a specific directory without the need to install to site-packages. The entry point method described above is the preferred way to integrate new extensions:

- Create directory in `/usr/lib/pulp/admin/extensions/` or `/usr/lib/pulp/consumer/extensions/`
- Add `__init__.py` to created directory.
- Add `pulp_cli.py` or `pulp_shell.py` as appropriate.
- In the above module, add a `def initialize(context)` method.
- The `context` object contains the CLI or shell instance that can be manipulated to add the extension's functionality.

Extension Example

This example will cover creating an extension with a single command, found in a new section at the root of the CLI. The full command will be:

```
$ pulp-admin example demo --name Jay --show-date
```

Framework Hook The first step is to create the method the framework will invoke when loading the extension. This method should create the necessary objects to populate the CLI with the extension's additions.

For this demo, the context is held in a global variable so it can be accessed when the command is run.

```
CONTEXT = None
def initialize(context):
    global CONTEXT
    CONTEXT = context
```

Sections There are two ways to add a new section to either the root of the CLI directly or to another section. One approach is to manually instantiate the `pulp.client.extensions.extensions.PulpCliSection` class, optionally subclassing it to add any needed enhancements. The instantiated object is then added to the CLI using the `add_section` call or to a parent section with the `add_subsection` call.

The other approach is to use the `create_section` helper method found in the CLI instance itself or the `create_subsection` call in other `PulpCliSection` instances. The demo will use the latter approach.

The `create_section` call takes the name of the section (i.e. the text that will be used on the command line directly) and a description for help text purposes. The `PulpCliSection` instance is returned from the call so it can be further manipulated. The CLI instance is retrieved from the client context.

```
ex_section = context.cli.create_section('example', 'Example section')
```

At this point, if this extension was installed, the new section would appear in the usage. Installation is covered later in this document.

```
$ pulp-admin
Usage: pulp-admin [SUB_SECTION, ..] COMMAND

 Available Sections:
   auth      - manage users, roles and permissions
   bindings  - search consumer bindings
   event     - subscribe to event notifications
   example   - Example section
   orphan    - find and remove orphaned content units
   puppet    - manage Puppet-related content and features
   repo      - list repositories and manage repo groups
   rpm       - manage RPM-related content and features
   server    - display info about the server
   tasks     - list and cancel server-side tasks
```

Commands Commands associate the user request with the method that will handle it. Commands are added to sections using a similar approach as with sections. The command can be manually instantiated from the `pulp.client.extensions.extensions.PulpCliCommand` class and added to a section with the `add_command` method.

Alternatively, a helper method named `create_command` can be used to do both the instantiation and add it to a section. This call accepts three parameters. The first is the name of the command which is used to invoke it from the command line. The second is the description, displayed when viewing the usage of the command. The third is a reference to the method to run when the command is executed.

The following snippet creates our demo command and ties it to the `run_demo` method.

```
demo_command = ex_section.create_command('demo', 'Demo command', run_demo)
```

The referenced `run_demo` must be defined as a method, otherwise the extension will fail to load. We'll expand on this in the next section, but a simple implementation is as follows.

```
def run_demo(**kwargs):
  CONTEXT.prompt.write('Hello World')
```

Options and Flags While some commands can simply be executed as is, many will need to accept user input. These are referred to as *options* and *flags*. Both can be created by running the appropriate `create_*` method on the command instance.

For the demo, we'll add an option that accepts the user's name and a flag that toggles whether or not the date is printed.

```
demo_command.create_option('--name', 'Name of the user', required=True)
demo_command.create_flag('--show-date', 'If specified, the date will be displayed')
```

The above snippet configures the `--name` option as required. The client framework will enforce that, displaying the usage text to the user in the event it is not specified.

```
$ pulp-admin example demo
Command: demo
Description: Demo command

Available Arguments:

  --name      - (required) Name of the user
  --show-date - If specified, the date will be displayed
```

```
The following options are required but were not specified:
  --name
```

The client framework will capture the input and make it available to the command's execution method in its `kwargs` argument. The name of the option/flag is used as the key and the user input is the value (or `True` in the case of a flag). Below is the `run_demo` method from above, enhanced to take advantage of our newly added options and flags.

```
def run_demo(**kwargs):
    CONTEXT.prompt.write('Hello %s' % kwargs['name'])
    if kwargs['show-date']:
        CONTEXT.prompt.write(datetime.datetime.now())
```

Example usage:

```
$ pulp-admin example demo --name Jay
Hello Jay

$ pulp-admin example demo --name Jay --show-date
Hello Jay
2013-02-07 14:54:14.587727
```

Installation Instructions on packaging and installing extensions for production deployment can be found at *Entry Points*.

For simplicity, this demo will install the extension using the directory approach. More information can be found in the *Directory Loading* section of this guide.

- Create `/usr/lib/pulp/admin/extensions/example`
- Create an empty file in that directory named `__init__.py`
- Copy the file containing this demo code to that directory, naming it `pulp_cli.py`

When the `pulp-admin` script is run, the usage text will show the `example` section created from this demo.

Full Example

```
import datetime

CONTEXT = None

def initialize(context):
    global CONTEXT
    CONTEXT = context

    ex_section = context.cli.create_section('example', 'Example section')
    demo_command = ex_section.create_command('demo', 'Demo command', run_demo)
    demo_command.create_option('--name', 'Name of the user', required=True)
    demo_command.create_flag('--show-date', 'If specified, the date will be displayed')

def run_demo(**kwargs):
    CONTEXT.prompt.write('Hello %s' % kwargs['name'])
    if kwargs['show-date']:
        CONTEXT.prompt.write(datetime.datetime.now())
```

2.5.3 Agent Handlers

Overview

The Pulp agent processes messages sent from the server to a consumer. These messages include informing the consumer of a new binding to a repository or a request to install one or more content units.

The implementation for how those requests are found will vary depending on the types of units involved. The agent supports writing *handlers* that will be used depending on the data in the operation. For example, when a bind request is received for a yum repository, a specific handler is invoked to edit the appropriate repository definition file.

An example extension can be found on the Handler Example page.

Documentation

Handlers

Overview The pulp agent supports an API for remote operations. These operations can be categorized as those that are type-specific and those that are not. In this context, the term type-specific includes a wide variety of Pulp conceptual types. Just as the handling of these types is pluggable within the Pulp server, it is also pluggable within the Pulp agent. The agent delegates the implementation of type-specific operations to the appropriate handler.

The collection of type-specific operations is logically grouped into named capabilities to support a good division of responsibility within handlers. An agent handler provides an implementation of one or more of these predefined capabilities.

Handler capabilities are as follows:

- The **bind** capability is a collection of agent operations responsible for updating the consumer's configuration so that it can consumer content from a specific Pulp repository. Handler classes are mapped to the *bind* capability by distributor type ID.
- The **content** capability is a collection of agent operations responsible for installing, updating, and uninstalling content on the consumer. Handler classes are mapped to the *content* capability by content type ID.
- The **system** capability is a collection of agent operations responsible for operating system level operations. Handler classes are mapped to the *system* capability using the operating system type as defined to the Python interpreter.

Each handler is defined using a configuration file, called a *descriptor*. In addition to handler configuration, the descriptor associates capabilities with Python classes that are contributed by the handler. The mapping of capabilities to handler classes is qualified by a *type* ID that is appropriate to the capability.

When a remote operation request is received by the agent, the implementation is delegated to an appropriate handler based on the *type* of the object that is the subject of the operation.

For example, the installation of a content unit of type `tarball` would be delegated to the `install()` method on an instance of the handler class mapped to the *content* capability and type ID of `tarball`.

Handler Descriptors The handler descriptor declares and configures an agent handler. It is an INI formatted text file that is installed into `/etc/Pulp/agent/conf.d/`. A descriptor has two required sections. The `[main]` section defines global handler properties, and the `[types]` section is used to map types to handler classes.

The `[main]` section has a required `enabled` property. The handler is loaded into the agent if value of enabled is one of: *(true|yes|1)*.

The `[types]` section supports three optional properties that correspond to handler capabilities. The `[content]` property is a comma delimited list of Pulp content types that are supported by the handler. The values listed must correspond to content types defined within the Pulp platform `types` inventory. The `bind` property is a comma delimited list of distributor types supported by the handler. The values listed must correspond to the distributor_type_id of a distributor plugin currently installed on the Pulp server. Lastly, the system property is a single value that must correspond to the operating system name as reported in python by the `os.uname()` function.

For each type listed in the content, bind and system properties, there must be a corresponding section with the same name. This section has a required property named `class` that is used to specify a class that provides the capability for the specified type. In addition to the `class` property, these sections support the inclusion of arbitrary property names and values which are passed to the specified handler class as configuration.

Let's take a look at a descriptor example:

```
[main]
enabled=1

[types]
content=rpm,puppet,tar
bind=yum
system=Linux

[rpm]
class=pulp_rpm.agent.handler.PackageHandler
import_key=1
permit_reboot=1

[puppet]
class=pulp_puppet.agent.handler.PuppetHandler

[tar]
class=pulp_tar.agent.handler.TarHandler
preserve_permissions=1

[yum]
class=pulp_rpm.agent.handler.YumBindHandler
ssl_verify=0

[Linux]
class=pulp.agent.handler.LinuxHandler
reboot_delay=10
```

In this example, the `[types]` section lists support for the `rpm`, `puppet`, and `tar` content types. Notice that there are the corresponding sections named `[rpm]` `[puppet]`, and `[tar]` that map the handler class and specify-type specific configuration. This pattern is replicated for the `bind`, and `system` properties.

Handler Classes The functionality contributed by agent handlers is implemented in handler classes. The required API for each class is dictated by the capability to which it's mapped. For each capability there is a corresponding abstract base class.

The base classes for each capability are as follows:

- Classes that provide the **content** capability must extend the `ContentHandler` base class and override each method.
- Classes that provide the **bind** capability must extend the `BindHandler` base class and override each method.
- Classes that provide the **system** capability must extend the `SystemHandler` base class and override each method.

Note: Currently, the APIs for the handler base classes are not published. The code can be found in `platform/src/pulp/agent/lib/handler.py`.

By convention, each handler class method signature contains two standard parameters. The `conduit` parameter is an object that provides access to objects within the agent's environment, such as the consumer configuration, Pulp server API bindings, the consumer's ID, and a progress reporting object. The `options` parameter is a dictionary that defines options used to influence the operation's implementation.

Note: Currently, the APIs for the conduit are not published. The code can be found in `platform/src/pulp/agent/lib/conduit.py`.

Reports The agent handler framework defines a set of report classes. Each method implementation must return the appropriate report object. The `HandlerReport` has three main attributes. The `succeeded` flag is boolean indicating the overall success of the operation. The definition of success is entirely at the discretion of the handler writer. The `details` attribute is a dictionary containing the detailed result of the operation. Last, the `num_changes` attribute indicates the total number of changes made to the consumer as a result of the operation. It is intended that the handler writer use either the `set_succeeded()` or the `set_failed()` methods to update the report. The default value fo the `succeeded` attribute is True. Table mapping types, handler classes, and report classes:

Type	Class	Method	Report
content	ContentHandler	install()	ContentReport
		update()	ContentReport
		uninstall()	ContentReport
		profile()	ProfileReport
bind	BindHandler	bind()	BindReport
		unbind()	BindReport
		clean()	CleanReport
system	SystemHandler	reboot()	RebootReport

Note: Currently, the APIs for the reports are not published. The code can be found in `platform/src/pulp/agent/lib/report.py`.

Exception Handling Exceptions raised during handler class method invocation should be caught and either handled or incorporated into the result report. Uncaught exceptions are caught by the agent handler framework, logged, and used to construct and return the appropriate handler report object. In this report object, the `succeeded` attribute is set to False, and the `details` attribute is updated to contain the following keys:

- message - The exception message.
- trace - A string representation of the stack trace.

Installation The two components of an agent handler are installed as follows. The *Handler Descriptors* are installed in `/etc/pulp/agent/conf.d`. The modules containing *Handler Classes* can be either installed in the python path or installed in the `/usr/lib/pulp/agent/handlers`. directory. If installed in the python path, the `class` property in the descriptor must be package qualified as needed to be found within the python path.

The Pulp agent must be restarted for handler changes to take effect.

Logging The Pulp agent is implemented using Gofer plugins. Agent handler log messages are written to syslog.

Handler Example

The following handler example provides an implementation of the *content*, *bind*, and *system* capabilities. The choice to define all of them in a single descriptor is arbitrary. They could also have been defined in separate descriptors.

Handler

Handler Descriptor Define the handler descriptor.

```
[main]
enabled=1

[types]
content=rpm
bind=yum
system=Linux

[rpm]
class=example.agent.handler.PackageHandler

[yum]
class=example.agent.handler.YumBindHandler

[Linux]
class=example.agent.handler.LinuxHandler
```

Content Handler Define the *PackageHandler* class.

```python
from pulp.agent.lib.handler import ContentHandler
from pulp.agent.lib.report import ProfileReport, ContentReport


class PackageHandler(ContentHandler):
    """
    An RPM content handler.
    Install, update, and uninstall RPMs.
    """

    def install(self, conduit, units, options):
        """
        Install RPM content units.  Each unit specifies an RPM that is to be installed.
        :param conduit: A handler conduit.
        :type conduit: pulp.agent.lib.conduit.Conduit
        :param units: A list of content unit keys.
        :type units: list
        :param options: Unit install options.
        :type options: dict
        :return: An installation report.
        :rtype: ContentReport
        """
        report = ContentReport()

        #
        # RPMs installed here
        #
        # succeeded = <did it succeed>
        # details = <the details of the installed packages here>
        #

        if succeeded:
            report.set_succeeded(details)
        else:
            report.set_failed(details)

        return report

    def update(self, conduit, units, options):
        """
        Update RPM content units.  Each unit specifies an RPM that is to be updated.
        :param conduit: A handler conduit.
        :type conduit: pulp.agent.lib.conduit.Conduit
        :param units: A list of content unit keys.
        :type units: list
        :param options: Unit update options.
        :type options: dict
        :return: An update report.
        :rtype: ContentReport
        """
        report = ContentReport()

        #
        # RPMs updated here
        #
        # succeeded = <did it succeed>
        # details = <the details of the updated packages here>
        #

        if succeeded:
            report.set_succeeded(details)
        else:
            report.set_failed(details)

        return report

    def uninstall(self, conduit, units, options):
        """
        Uninstall RPM content units.  Each unit specifies an RPM that is to be uninstalled.
        :param conduit: A handler conduit.
        :type conduit: pulp.agent.lib.conduit.Conduit
        :param units: A list of content unit_keys.
        :type units: list
        :param options: Unit uninstall options.
        :type options: dict
        :return: An uninstall report.
        :rtype: ContentReport
        """
        report = ContentReport()

        #
        # RPMs uninstalled here
        #
        # succeeded = <did it succeed>
        # details = <the details of the uninstalled packages here>
        #

        if succeeded:
            report.set_succeeded(details)
        else:
            report.set_failed(details)

        return report

    def profile(self, conduit):
        """
        Get the installed package profile.
        :param conduit: A handler conduit.
        :type conduit: pulp.agent.lib.conduit.Conduit
        :return: An profile report.
        :rtype: ProfileReport
        """
        report = ProfileReport()

        #
        # Assemble the report here
        #
        # succeeded = <did it succeed>
        # details = <the package profile here>
        #

        if succeeded:
            report.set_succeeded(details)
        else:
            report.set_failed(details)

        return report
```

Bind Handler Define the *YumBindHandler* class.

```
from pulp.agent.lib.handler import BindHandler
from pulp.agent.lib.report import BindReport


class YumBindHandler(BindHandler):
    """
    A yum repository bind request handler.
    Manages the /etc/yum.repos.d/abc.repo based on bind requests.
    """

    def bind(self, conduit, binding, options):
        """
        Bind a repository.
        :param conduit: A handler conduit.
        :type conduit: pulp.agent.lib.conduit.Conduit
        :param binding: A binding to add/update.
          A binding is: {type_id:<str>, repo_id:<str>, details:<dict>}
        :type binding: dict
        :param options: Bind options.
        :type options: dict
        :return: A bind report.
        :rtype: BindReport
        """
        repo_id = binding['repo_id']
        report = BindReport(repo_id)

        #
        # Update the abc.repo file here
        #
        # succeeded = <did it succeed>
        # details = <the details of the bind here>
        #

        if succeeded:
            report.set_succeeded(details)
        else:
            report.set_failed(details)

        return report

    def unbind(self, conduit, repo_id, options):
        """
        Bind a repository.
        :param conduit: A handler conduit.
        :type conduit: pulp.agent.lib.conduit.Conduit
        :param repo_id: A repository ID.
        :type repo_id: str
        :param options: Unbind options.
        :type options: dict
        :return: An unbind report.
        :rtype: BindReport
        """
        report = BindReport(repo_id)

        #
        # Update a abc.repo file here
        #
        # succeeded = <did it succeed>
        # details = <the details of the unbind here>
        #

        if succeeded:
            report.set_succeeded(details)
        else:
            report.set_failed(details)

        return report
```

System Handler Define the *LinuxHandler* class.

```
from pulp.agent.lib.handler import SystemHandler
from pulp.agent.lib.report import RebootReport


class LinuxHandler(SystemHandler):
    """
    Linux system handler
    Provides support for operating system specific operations.
    """
    def reboot(self, conduit, options):
        """
        Schedule a system reboot.
        :param conduit: A handler conduit.
        :type conduit: pulp.agent.lib.conduit.Conduit
        :param options: reboot options
        :type options: dict
        """
        report = RebootReport()

        #
        # Schedule the reboot here
        #
        # succeeded = <did it succeed>
        # details = <the details of the reboot here>
        #

        if succeeded:
            report.set_succeeded(details)
        else:
            report.set_failed(details)

        return report
```

Installation The descriptor is installed into the */etc/pulp/agent/conf.d/* directory.

The example handler classes are installed into *site-packages/example/agent/handler/* where *site-packages* can be any directory in the python path.

After installation, restart the goferd service.

2.6 Integrating with Pulp

2.6.1 REST API

Authentication

Calls to the REST API must be authenticated as a *User*.

Basic Authentication

Any call to the REST API may use HTTP basic authentication to provide a username and password.

User Certificates

You can "login" to Pulp by providing basic auth credentials and receiving a client SSL certificate to use for future requests. Although this is a POST operation, there is no data required in the request. The resulting certificate can then be used as a client-side SSL certificate for authentication of future requests to the REST API.

Method: POST

Path: `/pulp/api/v2/actions/login/`
Permission: read

Response Codes:

- **200** - credentials were accepted
- **401** - credentials were not accepted

Return: response body contains a client SSL certificate and private key

Sample 200 Response Body:

```
{
 "key":
   "-----BEGIN RSA PRIVATE KEY-----
   MIICXAIBAAKBgQC/AW1iSiMbwAeHJcwCQedMHaKg8/3aBA88pkYGwJL1cxlmN5Hr
   OL2WYUi3Kbkt51n56LiBc5wetQ3O2WDARaLTuk4j9LJDVsN065F6q4NuwYLx8lar
   U5ZQVfxE/CP/2KW2ymp4YPFksoo1yZJDvComteuVk2n20o4MKtE7VvYCSwIDAQAB
   AoGAfeJ57hLAitSH4ZmWmFJJF9BcU8obH2oXhLhtZJvc/2npboXnZOjTgt4BJ76W
   7lsQ4PVxTNgeJ9raC98WtgHvKooTyPagIudVBFMszTbseJU7XV8gfC66sG/j4h5U
   d9eJDClgjHbPTSgzFG4Y4Wv/2s4wMl8S/1t0svV4QiS/cdECQQDrGZ5qoV8mWNMp
   gl9NanpfhyRCej0bpdAJ/BDfrYuuE9FJ8r108cDVlIM+XWV6vRGi2W05YZBo7ys/
   KPg/PiDpAkEAz/xPkRCmseVQhF8I2oXAYEbnx7Yxwc+6/MYyc0zK7I03TuEQUHd1
   TfFNSCjkjnrbiTkwU+JrAlgjhRvnYsppEwJABa6o1Yrw8cxTzj0IcKaSLpzlk3XA
   5FotnRAqmD1pktuHw3HKgnkVYBQm1+sJ+N14/6ahrTFefCrLsMsctOqbgQJAFhqV
   hjBD1wIs9XR4J2kxkcnXVjU5woRGNhkGQZS2uD8l0p8+sZ6Qe/EaKoIWEEJkVIgc
   Z73Xa49cbwgRJkGmuwJBALEDimytIFUSzXCJZj1s6/5Uvldfae3297pbdU0ByBHf
   /WZ1p6+u8FthSMnmq4DI4UDdxDQfNjHdcGcWPQb4yko=
   -----END RSA PRIVATE KEY-----
   ",
 "certificate":
   "-----BEGIN CERTIFICATE-----
   MIICMzCCARsCASgwDQYJKoZIhvcNAQEFBQAwFDESMBAGA1UEAxMJbG9jYWxob3N0
   MB4XDTEzMDIyMjE2MjExOFoXDTEzMDMwMTE2MjExOFowLzEtMCsGA1UEAxMkYWRt
   aW46YWRtaW46NTA1YTJiZDFlMTlhMDA2MzViMDAwMDA5MIGfMA0GCSqGSIb3DQEB
   AQUAA4GNADCBiQKBgQC/AW1iSiMbwAeHJcwCQedMHaKg8/3aBA88pkYGwJL1cxlm
   N5HrOL2WYUi3Kbkt51n56LiBc5wetQ3O2WDARaLTuk4j9LJDVsN065F6q4NuwYLx
   8larU5ZQVfxE/CP/2KW2ymp4YPFksoo1yZJDvComteuVk2n20o4MKtE7VvYCSwID
   AQABMA0GCSqGSIb3DQEBBQUAA4IBAQB1JfB8iVv8m/jqROjU2oKtvBCj5RdBELZp
   tz/9TLXUgg7WpGezGIiKfss+hJW7QV1kuOfYS/5kO5XE8rYKg2FB5Tdx5fs4MTPT
   Th7h+kyg6On8y2o1J/uCQ2PSb3Ex5ajbY+PBNqWngcPLIi+Xn0iRmJmgRUO7QZ08
   GXqvcA0wsM0+07WcMNINxfQ1RuEmxWPNtJ861akNyGP8ZsmT0ABd5Q+pUq/nuBQ9
   7jwhi90WftYsQDHik9Ek43ltDVjfhDhQFWg3QKM7Xg2BkYkYYGB6ld6+v/jpOxtp
   Bg9xsQGTzaPcxGKAAwRHnEJ8vcBK+DIH5CqKOmhxxEveBDFWSNAI
   -----END CERTIFICATE-----
   "
}
```

Consumer APIs

Register, Update, and Unregister

Register a Consumer Register a new consumer with the Pulp server. Consumer IDs must be unique across all consumers registered to the server.

Part of the reply to this call is an x.509 certificate that is used to identify the consumer for operations performed on its behalf. The certificate should be stored with the consumer and used as its client-side SSL certificate to convey its identification.

Method: POST
Path: `/pulp/api/v2/consumers/`
Permission: create
Request Body Contents:

- **id** (string) - unique identifier for the consumer
- **display_name** (string) - *(optional)* user-friendly name for the consumer
- **description** (string) - *(optional)* user-friendly text describing the consumer
- **notes** (object) - *(optional)* key-value pairs to programmatically tag the consumer

Response Codes:

- **201** - if the consumer was successfully registered
- **400** - if one or more of the parameters is invalid
- **409** - if there is already a consumer with the given ID

Return: details of registered consumer along with the certificate used to identify the consumer in the future

Sample Request:

```
{
 "notes": {"arch": "x86_64", "os": "fedora16"},
 "display_name": null,
 "id": "test-consumer",
 "description": "Fedora 16 test build machine"
}
```

Sample 201 Response Body:

```
{
 "consumer": {
   "display_name": "test-consumer",
   "description": "Fedora 16 test build machine",
   "_ns": "consumers",
   "notes": {},
   "rsa_pub": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgw...K7\newIDAP//\n-----END PUBLIC KEY-----\n",
   "capabilities": {},
   "_id": {
     "$oid": "5367e982e13823076517f976"
   },
   "id": "test-consumer",
   "_href": {
     "_href": "/pulp/api/v2/consumers/test-consumer/"
   }
 },
 "certificate": "-----BEGIN RSA PRIVATE KEY-----\nMIICX...at9E1vT0=\n-----END CERTIFICATE-----"
}
```

Update a Consumer The update consumer call is used to change the details of an existing consumer.

Method: PUT
Path: `/pulp/api/v2/consumers/<consumer_id>/`
Permission: update
Request Body Contents: The body of the request is a JSON document with a root element called `delta`. The contents of delta are the values to update. Only changed parameters need be specified. The following keys are allowed in the delta object. Descriptions for each parameter can be found under the register consumer API:

- **display_name**
- **description**
- **notes**

Response Codes:

- **200** - if the update was executed and successful
- **404** - if there is no consumer with the given ID
- **400** - if one or more of the parameters is invalid

Return: database representation of the consumer after changes made by the update

Sample Request:

```
{
 "delta": {"display-name": "Test Consumer",
           "notes": {"arch": "x86_64"},
           "description": "QA automation testing machine"}
}
```

Sample 200 Response Body:

```
{
 "consumer": {
   "display_name": "Test Consumer",
   "description": "QA automation testing machine",
   "_ns": "consumers",
   "notes": {
     "arch": "x86_64"
   },
   "rsa_pub": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgw...K7\newIDAP//\n-----END PUBLIC KEY-----\n",
   "capabilities": {},
   "_id": {
     "$oid": "5367e982e13823076517f976"
   },
   "id": "test-consumer",
   "_href": {
     "_href": "/pulp/api/v2/consumers/test-consumer/"
   }
 },
 "certificate": "-----BEGIN RSA PRIVATE KEY-----\nMIICX...at9E1vT0=\n-----END CERTIFICATE-----"
}
```

Unregister a Consumer Unregister a consumer from the Pulp server. If the consumer is configured with messaging capabilities, it will be notified of its unregistration.

Method: DELETE
Path: `/pulp/api/v2/consumers/<consumer_id>/`
Permission: delete
Query Parameters:
Response Codes:

- **200** - if the consumer was successfully unregistered
- **404** - if there is no consumer with the given ID

Return: null

Repository Binding

Bind a Consumer to a Repository Bind a *consumer* to a *repository's distributor* for the purpose of consuming published content. Binding the consumer is performed in the following steps:

1. Create the *binding* on server.
2. Optionally send a request to the consumer to create the binding.

The distributor may support configuration options that it may use for that particular binding. These options would be used when generating the payload that is sent to consumers so they may access the repository. See the individual distributor's documentation for more information on the format.

Method: POST
Path: `/pulp/api/v2/consumers/<consumer_id>/bindings/`
Permission: create
Request Body Contents:

- **repo_id** (string) - unique identifier for the repository
- **distributor_id** (string) - identifier for the distributor
- **options** (object) - *(optional)* options passed to the handler on the consumer

- **notify_agent** (boolean) - *(optional)* indicates if the consumer should be sent a message about the new binding; defaults to true if unspecified
- **binding_config** (object) - *(optional)* options to be used by the distributor for this binding

Response Codes:

- **200** - if the bind request was fully processed on the server
- **202** - if an additional task was created to update consumer agents
- **400** - if one or more of the parameters is invalid
- **404** - if the consumer does not exist

Return: a *Call Report* if any tasks were spawned. In the event of a 200 response the body will be be the binding that was created.

Sample Request:

```
{
  "repo_id": "test-repo",
  "distributor_id": "dist-1"
}
```

Tags: Each task created to add the binding to a *consumer* will be created with the following tags: `"pulp:repository:<repo_id>"`, `"pulp:consumer:<consumer_id>"` `"pulp:repository_distributor:<distributor-id>"` `"pulp:action:bind"`

Unbind a Consumer Remove a binding between a *consumer* and a *repository's distributor*.

Unbinding the consumer is performed in the following steps:

1. Mark the *binding* as deleted on the server.
2. Send a request to the consumer to remove the binding.
3. Once the consumer has confirmed that the binding has been removed, it is permanently deleted on the server.

The steps for a forced unbind are as follows:

1. The *binding* is deleted on the server. This happens synchronously with the call.
2. Send a request to the consumer to remove the binding. The ID of the request to the consumer is returned via the spawned_tasks field of the *Call Report*.

If the notify_agent parameter was set to false when the binding was created, no request is sent to the consumer to remove the binding, so the binding is immediately deleted.

Method: DELETE
Path: `/pulp/api/v2/consumers/<consumer_id>/bindings/<repo_id>/<distributor_id>`
Permission: delete
Query Parameters: The consumer ID, repository ID and distributor ID are included in the URL itself.

- **force** (boolean) - *(optional)* delete the binding immediately and discontinue tracking consumer actions

- **options** (object) - *(optional)* options passed to the handler on the consumer

Response Codes:

- **200** - if notify_agent was set to false for the binding and it was immediately deleted
- **202** - the unbind request was accepted
- **400** - if one or more of the parameters is invalid
- **404** - if the consumer, repo, or distributor IDs don't exist, or if the binding does not exist

Return: a *Call Report* if any tasks were spawned.

Tags: Each task created to delete the binding from a *consumer* will be created with the following tags: `"pulp:repository:<repo_id>"`, `"pulp:consumer:<consumer_id>"` `"pulp:repository_distributor:<distributor-id>"` `"pulp:action:unbind"`

Retrieve a Single Binding Retrieves information on a single binding between a consumer and a repository.

Method: GET
Path: `/pulp/api/v2/consumers/<consumer_id>/bindings/<repo_id>/<distributor_id>`
Permission: read
Query Parameters: None; the consumer ID, repository ID and distributor ID are included in the URL itself. There are no supported query parameters.
Response Codes:

- **200** - if the bind exists
- **404** - if the given IDs don't exist, or if no bind exists with the given IDs

Return: database representation of the matching bind

Sample 200 Response Body:

```
{
  "repo_id": "test-repo",
  "consumer_id": "test-consumer",
  "_ns": "consumer_bindings",
  "_id": {"$oid": "5008604be13823703800003e"},
  "distributor_id": "dist-1",
  "id": "5008604be13823703800003e"
}
```

Retrieve All Bindings Retrieves information on all bindings for the specified consumer.

Method: GET

Path: `/pulp/api/v2/consumers/<consumer_id>/bindings/`
Permission: read
Query Parameters: None; the consumer ID is included in the URL itself. There are no supported query parameters.
Response Codes:

- **200** - if the consumer exists

Return: an array of database representations of the matching binds

Sample 200 Response Body:

```
[
  {
    "repo_id": "test-repo",
    "consumer_id": "test-consumer",
    " _ns": "consumer_bindings",
    "_id": {"$oid": "5008604be13823703800003e"},
    "distributor_id": "dist-1",
    "id": "5008604be13823703800003e"
  },
    "repo_id": "test-repo2",
    "consumer_id": "test-consumer",
    " _ns": "consumer_bindings",
    "_id": {"$oid": "5008604be13823703800003e"},
    "distributor_id": "dist-1",
    "id": "5008604be13823703800003e"
  },
 ]
```

Retrieve Binding By Consumer And Repository Retrieves information on all bindings between a consumer and a repository.

Method: GET
Path: `/pulp/api/v2/consumers/<consumer_id>/bindings/<repo_id>/`
Permission: read
Query Parameters: None; the consumer and repository IDs are included in the URL itself. There are no supported query parameters.
Response Codes:

- **200** - if both the consumer and repository IDs are valid
- **404** - if one or both of the given ids are not valid

Return: an array of objects, where each object represents a binding

Sample 200 Response Body:

```
[
 {
   "notify_agent": true,
   "repo_id": "test_repo",
   "_href": "/pulp/api/v2/consumers/test_consumer/bindings/test_repo/test_distributor/",
   "type_id": "test_distributor",
   "consumer_actions": [
     {
       "status": "pending",
       "action": "bind",
       "id": "3a8713bb-6902-4f11-a725-17c7f1f6586a",
       "timestamp": 1402688658.785708
     }
   ],
   "_ns": "consumer_bindings",
   "distributor_id": "test_distributor",
   "consumer_id": "test_consumer",
   "deleted": false,
   "binding_config": {},
   "details": {
     "server_name": "pulp.example.com",
     "ca_cert": null,
     "relative_path": "/pulp/repos/test_repo",
     "gpg_keys": [],
     "client_cert": null,
     "protocols": [
       "https"
     ],
     "repo_name": "test_repo"
   },
   "_id": {
     "$oid": "539b54927bc8f6388640871d"
   },
   "id": "539b54927bc8f6388640871d"
 }
]
```

Retrieval

Retrieve a Single Consumer Retrieves information on a single Pulp consumer. The returned data includes general consumer details.

Method: GET
Path: `/pulp/api/v2/consumers/<consumer_id>/`
Permission: read
Query Parameters:

- **details** (boolean) - include all details about the consumer
- **bindings** (boolean) - include information about consumer bindings

Response Codes:

- **200** - if the consumer exists
- **404** - if no consumer exists with the given ID

Return: database representation of the matching consumer with the addition of repository bindings information for the consumer

Sample 200 Response Body:

```
{
  "display_name": "test-consumer",
  "description": null,
  "certificate": "-----BEGIN CERTIFICATE-----\nMIICHDCCAQQCATowDQYJKoZIhvcNAQEFBQAwFDESMBAGA1UEAxMJb
  "_ns": "consumers",
  "notes": {"arch":"i386"},
  "capabilities": {},
  "unit_profile": [],
  "bindings": [],
  "_id": {
    "$oid": "4fbd3540e5e7102dae000015"
  },
  "id": "test-consumer"
}
```

Retrieve All Consumers Returns information on all consumers in the Pulp server. Eventually this call will support query parameters to limit the results and provide searching capabilities. This call will never return a 404; an empty array is returned in the case where there are no consumers.

Method: GET
Path: `/pulp/api/v2/consumers/`
Permission: read
Query Parameters:

- **details** (boolean) - include all details about the consumer
- **bindings** (boolean) - include information about consumer bindings

Response Codes:

- **200** - containing the array of consumers

Return: the same format as retrieving a single consumer, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
 {
   "display_name": "test-consumer",
   "description": null,
   "certificate": "-----BEGIN CERTIFICATE-----\nMIICHDCCAQQCATowDQYJKoZIhvcNAQEFBQAwFDESMBAGA1UEAxMJk
   "_ns": "consumers",
   "notes": {"arch":"i386"},
   "capabilities": {},
   "unit_profile": [],
   "bindings": [],
   "_id": {
     "$oid": "4fbd3540e5e7102dae000015"
   },
   "id": "test-consumer"
 },
 {
   "display_name": "test-consumer1",
   "description": null,
   "certificate": "-----BEGIN CERTIFICATE-----\nMIICHDCCAQQCATowDQYJKoZIhvcNApCEFBQAwFDESMBAGA1UEAxM
   "_ns": "consumers",
   "notes": {},
   "capabilities": {},
   "unit_profile": [],
   "bindings": [],
   "_id": {
     "$oid": "4fbd3540e5e7102dae00000d"
   },
   "id": "test-consumer1"
 }
]
```

Advanced Search for Consumers Please see *Search API* for more details on how to perform these searches.

Returns information on consumers in the Pulp server that match your search parameters. It is worth noting that this call will never return a 404; an empty array is returned in the case where there are no consumers.

Method: POST
Path: `/pulp/api/v2/consumers/search/`
Permission: read
Request Body Contents: include the key "criteria" whose value is a mapping structure as defined in *Search Criteria*. Optionally include the key "bindings" with any value that evaluates to True to have the "bindings" attribute added to each returned consumer.

- **criteria** (object) - the search criteria defined in *Search Criteria*
- **details** (boolean) - include all details about the consumer
- **bindings** (boolean) - include information about consumer bindings

Response Codes:

- **200** - containing the array of consumers

Return: the same format as retrieving a single consumer, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
 {
   "display_name": "test-consumer",
   "description": null,
   "certificate": "-----BEGIN CERTIFICATE-----\nMIICHDCCAQQCATowDQYJKoZIhvcNAQEFBQAwFDESMBAGA1UEAxMJ
   "_ns": "consumers",
   "notes": {"arch":"i386"},
   "capabilities": {},
   "unit_profile": [],
   "bindings": [],
   "_id": {
     "$oid": "4fbd3540e5e7102dae000015"
   },
   "id": "test-consumer"
 },
 {
   "display_name": "test-consumer1",
   "description": null,
   "certificate": "-----BEGIN CERTIFICATE-----\nMIICHDCCAQQCATowDQYJKoZIhvcNApCEFBQAwFDESMBAGA1UEAxM
   "_ns": "consumers",
   "notes": {},
   "capabilities": {},
   "unit_profile": [],
   "bindings": [],
   "_id": {
     "$oid": "4fbd3540e5e7102dae00000d"
   },
   "id": "test-consumer1"
 }
]
```

Returns information on consumers in the Pulp server that match your search parameters. It is worth noting that this call will never return a 404; an empty array is returned in the case where there are no consumers.

This method is slightly more limiting than the POST alternative, because some filter expressions may not be serializable as query parameters.

Method: GET
Path: `/pulp/api/v2/consumers/search/`
Permission: read
Query Parameters: query params should match the attributes of a Criteria object as defined in *Search Criteria*. For example: /v2/consumers/search/?field=id&field=display_name&limit=20' Include the key 'bindings' to have the 'bindings' attribute, an array of related bindings, added to each returned consumer.
Response Codes:

- **200** - containing the array of consumers

Return: the same format as retrieving a single consumer, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
 {
   "display_name": "test-consumer",
   "description": null,
   "certificate": "-----BEGIN CERTIFICATE-----\nMIICHDCCAQQCATowDQYJKoZIhvcNAQEFBQAwFDESMBAGA1UEAxMJ
   "_ns": "consumers",
   "notes": {"arch":"i386"},
   "capabilities": {},
   "unit_profile": [],
   "bindings": [],
   "_id": {
     "$oid": "4fbd3540e5e7102dae000015"
   },
   "id": "test-consumer"
 },
 {
   "display_name": "test-consumer1",
   "description": null,
   "certificate": "-----BEGIN CERTIFICATE-----\nMIICHDCCAQQCATowDQYJKoZIhvcNApCEFBQAwFDESMBAGA1UEAxM
   "_ns": "consumers",
   "notes": {},
   "capabilities": {},
   "unit_profile": [],
   "bindings": [],
   "_id": {
     "$oid": "4fbd3540e5e7102dae00000d"
   },
   "id": "test-consumer1"
 }
]
```

Content Management

Install Content on a Consumer Install one or more content units on a consumer. This operation is asynchronous. If dependencies are automatically installed or updated, it is reflected in the installation report.

The units to be installed are specified in an array. Each unit in the array of *units* is an object containing two required attributes. The first is the **type_id** which a string that defines the unit's content type. The value is unrestricted by the Pulp server but must match a type mapped to a content *handler* in the agent. The second is the **unit_key** which identifies the unit or units to be installed. Both the structure and content are handler specific.

The caller must also pass an *options* object, which specifies additional install options. Both the structure and content are handler specific. The options drive how the handler performs the operation.

Method: POST
Path: `/pulp/api/v2/consumers/<consumer_id>/actions/content/install/`
Permission: create
Request Body Contents:

- **units** (array) - array of content units to install
- **options** (object) - install options

Response Codes:

- **202** - the install request has been accepted
- **400** - if one or more of the parameters is missing or invalid
- **404** - if the consumer does not exist

Return: a *Call Report*

Sample Request:

```
{
  "units": [
    {"unit_key": {"name": "zsh", "version": "4.3.17"}, "type_id": "rpm"}
  ],
  "options": {
    "apply": true, "reboot": false, "importkeys": false
  }
}
```

Tags: The task created will have the following tags: `"pulp:action:unit_install"`, `"pulp:consumer:<consumer_id>"`

Update Content on a Consumer Update one or more content units on a consumer. This operation is asynchronous. If dependencies are automatically installed or updated, it is reflected in the update report.

The units to be updated are specified in an array. Each unit in the array of *units* is an object containing two required attributes. The first is the **type_id** which a string that defines the unit's content type. The value is unrestricted by the Pulp server but must match a type mapped to a content *handler* in the agent. The second is the **unit_key** which identifies the unit or units to be updated. Both the structure and content are handler specific.

The caller must also pass an *options* object, which specifies additional update options. Both the structure and content are handler specific. The options drive how the handler performs the operation.

Method: POST
Path: `/pulp/api/v2/consumers/<consumer_id>/actions/content/update/`
Permission: create
Request Body Contents:

- **units** (array) - array of content units to update
- **options** (object) - update options

Response Codes:

- **202** - the update request has been accepted
- **400** - if one or more of the parameters is missing or invalid
- **404** - if the consumer does not exist

Return: a *Call Report*

Sample Request:

```
{
  "units": [
    {"unit_key": {"name": "zsh"}, "type_id": "rpm"}
  ],
  "options": {
    "apply": true, "reboot": false, "all": false, "importkeys": false
  }
}
```

Tags: The task created will have the following tags: `"pulp:action:unit_update"`, `"pulp:consumer:<consumer_id>"`

Uninstall Content on a Consumer Uninstall one or more content units on a consumer. This operation is asynchronous. If dependencies are automatically removed, it is reflected in the uninstall report.

The units to be uninstalled are specified in an array. Each unit in the array of *units* is an object containing two required attributes. The first is the **type_id** which a string that defines the unit's content type. The value is unrestricted by the Pulp server but must match a type mapped to a content *handler* in the agent. The second is the **unit_key** which identifies the unit or units to be uninstalled. The value is completely defined by the handler mapped to the unit's type_id.

The caller must also pass an *options* object, which specifies additional uninstall options. Both the structure and content are handler specific. The options drive how the handler performs the operation.

Method: POST
Path: `/pulp/api/v2/consumers/<consumer_id>/actions/content/uninstall/`
Permission: create
Request Body Contents:

- **units** (array) - array of content units to uninstall
- **options** (object) - uninstall options

Response Codes:

- **202** - the uninstall request has been accepted
- **400** - if one or more of the parameters is missing or invalid
- **404** - if the consumer does not exist

Return: a *Call Report*

Sample Request:

```
{
  "units": [
    {"unit_key": {"name": "zsh"}, "type_id": "rpm"}
  ],
  "options": {
    "apply": true, "reboot": false
  }
}
```

Tags: The task created will have the following tags: `"pulp:action:unit_uninstall"`, `"pulp:consumer:<consumer_id>"`

Scheduled Content Management

Pulp has the ability to schedule content unit installs, updates, and uninstalls on a given consumer. The schedules can be created, manipulated, and queried with with the following APIs.

Note that all schedule resources are in the format described in *Scheduled Tasks*.

For each request, `<action>` should be one of `install`, `update`, or `uninstall`.

Listing Schedules

Method: GET
Path: `/pulp/api/v2/consumers/<consumer id>/schedules/content/<action>/`
Permission: read
Response Codes:

- **200** - if the consumer exists
- **404** - if the consumer does not exist

Return: (possibly empty) array of schedule resources

Sample 200 Response Body:

```
[
 {
   "next_run": "2014-01-28T16:33:26Z",
   "task": "pulp.server.tasks.consumer.update_content",
   "last_updated": 1390926003.828128,
   "first_run": "2014-01-28T10:35:08Z",
   "schedule": "2014-01-28T10:35:08Z/P1D",
   "args": [
     "me"
   ],
   "enabled": true,
   "last_run_at": null,
   "_id": "52e7d8b3dd01fb0c8428b8c2",
   "total_run_count": 0,
   "failure_threshold": null,
   "kwargs": {
     "units": [
       {
         "unit_key": {
           "name": "pulp-server"
         },
         "type_id": "rpm"
       }
     ],
     "options": {}
   },
   "units": [
     {
       "unit_key": {
         "name": "pulp-server"
       },
       "type_id": "rpm"
     }
   ],
   "resource": "pulp:consumer:me",
   "remaining_runs": null,
   "consecutive_failures": 0,
   "options": {},
   "_href": "/pulp/api/v2/consumers/me/schedules/content/update/52e7d8b3dd01fb0c8428b8c2/"
 }
]
```

Creating a Schedule

Method: POST
Path: `/pulp/api/v2/consumers/<consumer id>/schedules/content/<action>/`
Permission: create
Request Body Contents:

- **schedule** (string) - schedule in iso8601 interval format
- **failure_threshold** (integer) - *(optional)* number of consecutive failures allowed before automatically disabling
- **enabled** (boolean) - *(optional)* whether or not the schedule is enabled (enabled by default)
- **options** (object) - *(optional)* key - value options to pass to the install agent
- **units** (array) - array of units to install

Response Codes:

- **201** - if the schedule was successfully created
- **400** - if any of the required params are missing or any params are invalid
- **404** - if the consumer does not exist

Return: resource representation of the new schedule

Sample Request:

```
{"schedule": "R1/P1DT",
 "units": [{"type_id": "rpm", "unit_keys": {"name": "gofer"}}]
}
```

Sample 201 Response Body:

```
{
 "next_run": "2012-09-22T14:15:00Z",
 "task": "pulp.server.tasks.consumer.update_content",
 "last_updated": 1390926003.828128,
 "first_run": "2012-09-22T14:15:00Z",
 "schedule": "R1/P1DT",
 "args": [
   "me"
 ],
 "enabled": true,
 "last_run_at": null,
 "_id": "52e7d8b3dd01fb0c8428b8c2",
 "total_run_count": 0,
 "failure_threshold": null,
 "kwargs": {
   "units": [
     {
       "unit_key": {
         "name": "gofer"
       },
       "type_id": "rpm"
     }
   ],
   "options": {}
 },
 "units": [
   {
     "unit_key": {
       "name": "gofer"
     },
     "type_id": "rpm"
   }
 ],
 "resource": "pulp:consumer:me",
 "remaining_runs": 1,
 "consecutive_failures": 0,
 "options": {},
 "_href": "/pulp/api/v2/consumers/me/schedules/content/update/52e7d8b3dd01fb0c8428b8c2/"
}
```

Retrieving a Schedule

Method: GET
Path: `/pulp/api/v2/consumers/<consumer id>/schedules/content/<action>/<schedule id>/`
Permission: read
Response Codes:

- **200** - if both the consumer and the scheduled install exist
- **404** - if either the consumer or scheduled install does not exist

Return: schedule resource representation

Sample 200 Response Body:

```
{
   "_href": "/pulp/api/v2/consumers/me/schedules/content/update/52e7d8b3dd01fb0c8428b8c2/",
   "_id": "52e7d8b3dd01fb0c8428b8c2",
   "args": [
       "consumer1"
   ],
   "consecutive_failures": 0,
   "enabled": true,
   "failure_threshold": null,
   "first_run": "2014-01-28T10:35:08Z",
   "kwargs": {
       "options": {},
       "units": [
           {
               "type_id": "rpm",
               "unit_key": {
                   "name": "pulp-server"
               }
           }
       ]
   },
   "last_run_at": null,
   "last_updated": 1390926003.828128,
   "next_run": "2014-01-28T16:50:47Z",
   "options": {},
   "remaining_runs": null,
   "resource": "pulp:consumer:me",
   "schedule": "2014-01-28T10:35:08Z/P1D",
   "task": "pulp.server.tasks.consumer.update_content",
   "total_run_count": 0,
   "units": [
       {
           "type_id": "rpm",
           "unit_key": {
               "name": "pulp-server"
           }
       }
   ]
}
```

Updating a Schedule

Method: PUT

Path: `/pulp/api/v2/consumers/<consumer id>/schedules/content/<action>/<schedule id>/`

Permission: update

Request Body Contents:

- **schedule** (string) - *(optional)* schedule as an iso8601 interval (specifying a recurrence will affect remaining_runs)
- **failure_threshold** (integer) - *(optional)* number of allowed consecutive failures before the schedule is disabled
- **remaining_runs** (integer) - *(optional)* number of remaining runs for schedule
- **enabled** (boolean) - *(optional)* whether or not the schedule is enabled

- **options** (object) - *(optional)* key - value options to pass to the install agent
- **units** (array) - *(optional)* array of units to install

Response Codes:

- **200** - if the schedule was successfully updated
- **400** - if any of the params are invalid
- **404** - if the consumer or schedule does not exist

Return: resource representation of the schedule

Sample Request:

```
{
 "units": [{"type_id": "rpm", "unit_keys": {"name": "grinder"}},
           {"type_id": "rpm", "unit_keys": {"name": "gofer"}}]
}
```

Sample 200 Response Body:

```
{
 "next_run": "2014-01-28T16:54:26Z",
 "task": "pulp.server.tasks.consumer.update_content",
 "last_updated": 1390928066.995197,
 "first_run": "2014-01-28T10:35:08Z",
 "schedule": "2014-01-28T10:35:08Z/P1D",
 "args": [
   "me"
 ],
 "enabled": false,
 "last_run_at": null,
 "_id": "52e7d8b3dd01fb0c8428b8c2",
 "total_run_count": 0,
 "failure_threshold": null,
 "kwargs": {
   "units": [
     {
       "unit_key": {
         "name": "grinder"
       },
       "type_id": "rpm"
     },
     {
       "unit_key": {
         "name": "gofer"
       },
       "type_id": "rpm"
     }
   ],
   "options": {}
 },
 "units": [
   {
     "unit_key": {
       "name": "grinder"
     },
     "type_id": "rpm"
   },
   {
     "unit_key": {
       "name": "gofer"
     },
     "type_id": "rpm"
   }
 ],
 "resource": "pulp:consumer:me",
 "remaining_runs": null,
 "consecutive_failures": 0,
 "options": {},
 "_href": "/pulp/api/v2/consumers/me/schedules/content/update/52e7d8b3dd01fb0c8428b8c2/"
}
```

Deleting a Schedule

Method: DELETE
Path: `/pulp/api/v2/consumers/<consumer id>/schedules/content/<action>/<schedule id>/`
Permission: delete
Response Codes:

- **200** - if the schedule was deleted successfully
- **404** - if the consumer or schedule does not exist

Return: null

Consumer History

Retrieve Consumer Event History Retrieves the history of events that occurred on a consumer. The array can be filtered by a number of fields including the event type and event timestamp data. Pagination support in the form of limits and skips is also provided.

Valid values for the event type filtering are as follows:

- consumer_registered
- consumer_unregistered
- repo_bound
- repo_unbound
- content_unit_installed
- content_unit_uninstalled
- unit_profile_changed
- added_to_group
- removed_from_group

Method: GET
Path: `/pulp/api/v2/consumers/<consumer_id>/history/`
Permission: read
Query Parameters:

- **event_type** (string) - *(optional)* type of event to retrieve; must be one of the values enumerated above
- **limit** (string) - *(optional)* maximum number of results to retrieve
- **sort** (string) - *(optional)* direction of sort by event timestamp; possible values: 'ascending', 'descending'
- **start_date** (string) - *(optional)* earliest date of events that will be retrieved; format: yyyy-mm-dd
- **end_date** (string) - *(optional)* latest date of events that will be retrieved; format: yyyy-mm-dd

Response Codes:

- **200** - for the successful retrieval of consumer history
- **404** - if the given consumer is not found

Return: array of event history objects

Sample Request:

```
/pulp/api/v2/consumers/test-consumer/history/?sort=descending&limit=2&event_type=consumer_registered
```

Sample 200 Response Body:

```
[
 {
   "originator": "SYSTEM",
   "timestamp": "2012-05-23T19:06:40Z",
   "consumer_id": "test-consumer",
   "details": null,
   "_ns": "consumer_history",
   "_id": {
     "$oid": "4fbd3540e5e7102dae000016"
   },
   "type": "consumer_registered",
   "id": "4fbd3540e5e7102dae000016"
 },
 {
   "originator": "SYSTEM",
   "timestamp": "2012-05-23T19:03:29Z",
   "consumer_id": "test-consumer1",
   "details": null,
   "_ns": "consumer_history",
   "_id": {
     "$oid": "4fbd3481e5e7102dae00000f"
   },
   "type": "consumer_registered",
   "id": "4fbd3481e5e7102dae00000f"
 }
]
```

Unit Profiles

Create A Profile Create a *unit profile* and associate it with the specified *consumer*. Unit profiles are associated to consumers by content type. Each consumer may be associated with one profile of a given content type at a time. If a profile of the specified content type is already associated with the consumer, it is replaced with the profile supplied in this call.

Method: POST
Path: `/pulp/api/v2/consumers/<consumer_id>/profiles/`
Permission: create
Request Body Contents:

- **content_type** (string) - the content type ID
- **profile** (object) - the content profile

Response Codes:

- **201** - if the profile was successfully created
- **400** - if one or more of the parameters is invalid
- **404** - if the consumer does not exist

Return: the created unit profile object

Sample Request:

```
{
  "content_type": "rpm",
  "profile": [{"arch": "i686",
               "epoch": 0,
               "name": "glib2",
               "release": "2.fc17",
               "vendor": "Fedora Project",
               "version": "2.32.4"},
              {"arch": "x86_64",
               "epoch": 0,
               "name": "rpm-libs",
               "release": "8.fc17",
               "vendor": "Fedora Project",
               "version": "4.9.1.3"}]
}
```

Sample 201 Response Body:

```
{
  "profile": [{"arch": "i686",
               "epoch": 0,
               "name": "glib2",
               "release": "2.fc17",
               "vendor": "Fedora Project",
                "version": "2.32.4"},
               {"arch": "x86_64",
                "epoch": 0,
                "name": "rpm-libs",
                "release": "8.fc17",
                "vendor": "Fedora Project",
                "version": "4.9.1.3"}],
  "_ns": "consumer_unit_profiles",
  "consumer_id": "test-consumer",
  "content_type": "rpm",
  "_href": "/pulp/api/v2/consumers/test-consumer/profiles/test-consumer/rpm/",
  "profile_hash": "2ecdf09a0f1f6ea43b5a991b468866bc07bcf8c2ac8251395ef2d78adf6e5c5b",
  "_id": {"$oid": "5008500ae138230abe000095"},
  "id": "5008500ae138230abe000095"
}
```

Replace a Profile Replace a *unit profile* associated with the specified *consumer*. Unit profiles are associated to consumers by content type. Each consumer may be associated to one profile of a given content type at one time. If no unit profile matching the specified content type is currently associated to the consumer, the supplied profile is created and associated with the consumer using the specified content type.

Method: PUT
Path: `/pulp/api/v2/consumers/<consumer_id>/profiles/<content-type>/`
Permission: update
Request Body Contents:

- **profile** (object) - the content profile

Response Codes:

- **201** - if the profile was successfully updated
- **400** - if one or more of the parameters is invalid
- **404** - if the consumer does not exist

Return: the created unit profile object

Sample Request:

```
{
  "profile": [{"arch": "i686",
                "epoch": 0,
                "name": "glib2",
                "release": "2.fc17",
                "vendor": "Fedora Project",
                "version": "2.32.4"},
               {"arch": "x86_64",
                "epoch": 0,
                "name": "rpm-libs",
                 "release": "8.fc17",
                 "vendor": "Fedora Project",
                 "version": "4.9.1.3"}]
}
```

Sample 201 Response Body:

```
{
  "profile": [{"arch": "i686",
                 "epoch": 0,
                 "name": "glib2",
                 "release": "2.fc17",
                 "vendor": "Fedora Project",
                 "version": "2.32.4"},
                {"arch": "x86_64",
                 "epoch": 0,
                 "name": "rpm-libs",
                 "release": "8.fc17",
                 "vendor": "Fedora Project",
                 "version": "4.9.1.3"}],
  "_ns": "consumer_unit_profiles",
  "consumer_id": "test-consumer",
  "content_type": "rpm",
  "_href": "/pulp/api/v2/consumers/test-consumer/profiles/test-consumer/rpm/",
  "profile_hash": "2abcf09a0f1f6ea43b5a991b468866bc07bcf8c2ac8251395ef2d78adf6e5c5b",
  "_id": {"$oid": "5008500ae138230abe000095"},
  "id": "5008500ae138230abe000095"
}
```

Retrieve All Profiles By Consumer Id Retrieves information on all unit profile‘s associated with a :term:‘consumer.

Method: GET
Path: `/pulp/api/v2/consumers/<consumer_id>/profiles/`
Permission: read
Query Parameters: None; There are no supported query parameters
Response Codes:

- **200** - regardless of whether any profiles exist
- **404** - if the consumer does not exist

Return: an array of unit profile objects or an empty array if none exist

Sample 200 Response Body:

```
[
 {"_href": "/pulp/api/v2/consumers/test-consumer/profiles/test-consumer/test-content-type/",
  "_id": {"$oid": "521d92b1e5e7102f7500004a"},
  "_ns": "consumer_unit_profiles",
  "consumer_id": "test-consumer",
  "content_type": "test-content-type",
  "id": "521d92b1e5e7102f7500004a",
  "profile": [{"arch": "i686",
               "epoch": 0,
               "name": "glib2",
               "release": "2.fc17",
               "vendor": "Fedora Project",
               "version": "2.32.4"},
              {"arch": "x86_64",
               "epoch": 0,
               "name": "rpm-libs",
               "release": "8.fc17",
               "vendor": "Fedora Project",
               "version": "4.9.1.3"}],
  "profile_hash": "15df1c6105edacd6b167d2e9dd87311b069f50cebb2f7968ef185c1d6eae5197"
 },
 {"_href": "/pulp/api/v2/consumers/test-consumer/profiles/test-consumer/rpm/",
  "_id": {"$oid": "5217d77de5e710796700000c"},
  "_ns": "consumer_unit_profiles",
  "consumer_id": "test-consumer",
  "content_type": "rpm",
  "id": "5217d77de5e710796700000c",
  "profile": [{"arch": "i686",
               "epoch": 0,
               "name": "glib2",
               "release": "2.fc17",
               "vendor": "Fedora Project",
               "version": "2.32.4"}],
  "profile_hash": "15df1c6105edacd6b167d2e9dd87311b069f50cebb2f7968ef185c1d6eae5197"
 }
]
```

Retrieve a Profile By Content Type Retrieves a *unit profile* associated with a *consumer* by content type.

Method: GET
Path: `/pulp/api/v2/consumers/<consumer_id>/profiles/<content_type>/`
Permission: read
Query Parameters: None; There are no supported query parameters
Response Codes:

- **200** - regardless of whether any profiles exist
- **404** - if the consumer or requested profile does not exists

Return: the requested unit profile object

Sample 200 Response Body:

```
{
  "_href": "/pulp/api/v2/consumers/test-consumer/profiles/test-consumer/test-content-type/",
  "_id": {"$oid": "521d92b1e5e7102f7500004a"},
  "_ns": "consumer_unit_profiles",
  "consumer_id": "test-consumer",
  "content_type": "test-content-type",
  "id": "521d92b1e5e7102f7500004a",
  "profile": [{"arch": "i686",
               "epoch": 0,
               "name": "glib2",
               "release": "2.fc17",
               "vendor": "Fedora Project",
               "version": "2.32.4"},
              {"arch": "x86_64",
               "epoch": 0,
               "name": "rpm-libs",
               "release": "8.fc17",
               "vendor": "Fedora Project",
               "version": "4.9.1.3"}],
  "profile_hash": "15df1c6105edacd6b167d2e9dd87311b069f50cebb2f7968ef185c1d6eae5197"
}
```

Retrieve All Profiles Retrieves information on all :term:‘unit profile‘s

Method: GET
Path: `/pulp/api/v2/consumers/profile/search/`
Permission: read
Query Parameters: None; There are no supported query parameters
Response Codes:

- **200** - containing the array of items

Return: array of unit profiles

Sample 200 Response Body:

```
[
  {
    "profile": [{"arch": "i686",
                 "epoch": 0,
                 "name": "glib2",
                 "release": "2.fc17",
                 "vendor": "Fedora Project",
                 "version": "2.32.4"},
                {"arch": "x86_64",
                 "epoch": 0,
                 "name": "rpm-libs",
                 "release": "8.fc17",
                 "vendor": "Fedora Project",
                 "version": "4.9.1.3"}],
    "_ns": "consumer_unit_profiles",
    "profile_hash": "d20dc2d0fce88a064a2f7309863da7ebd068969de0150fd8ff83c220a0785d8a",
    "consumer_id": "test-consumer",
    "content_type": "rpm",
    "_id": {"$oid": "545cacf09cd4ca28c83dd9f5"},
    "id": "545cacf09cd4ca28c83dd9f5"
  }
]
```

Searching Profiles Returns items that match the search parameters. To understand how to use the SearchAPI look at the *Search Criteria* page. This call will never return a 404; an empty array is returned in the case where there are no items in the database.

Method: POST
Path: `/pulp/api/v2/consumers/profile/search/`
Permission: read
Request Body Contents: include the key "criteria" whose value is a mapping structure as defined in *Search Criteria*
Response Codes:

- **200** - containing the result of the query

Return: elements matching the query

Sample Request:

```
{
  "criteria" : {
    "filters" : { "profile.name"  : "pulp-consumer-client" },
    "fields" : [ "consumer_id", "profile.$" ]
  }
}
```

Sample 200 Response Body:

```
{
  "profile": [
    {
      "vendor": "Koji",
      "name": "pulp-consumer-client",
      "epoch": 0,
      "version": "2.4.3",
      "release": "1.el6",
      "arch": "noarch"
    }
  ],
  "_id": {"$oid": "545cacf09cd4ca28c83dd9f5"},
  "id": "545cacf09cd4ca28c83dd9f5",
  "consumer_id": "pulp-test"
}
```

Content Applicability

Generate Content Applicability for Updated Consumers This API regenerates *applicability data* for a given set of consumers matched by a given *Search Criteria* asynchronously and saves it in the Pulp database. It should be used when a consumer profile is updated or consumer-repository bindings are updated. Applicability data is regenerated for all unit profiles associated with given consumers and for all content types that define applicability. Generated applicability data can be queried using the *Query Content Applicability* API described below.

The API will return a *Call Report*. Users can check whether the applicability generation is completed using task id in the *Call Report*. You can run a single applicability generation task at a time. If an applicability generation task is running, any new applicability generation tasks requested are queued and postponed until the current task is completed.

Method: POST
Path: `/pulp/api/v2/consumers/actions/content/regenerate_applicability/`
Permission: create
Request Body Contents:

- **consumer_criteria** (object) - a consumer criteria object defined in *Search Criteria*

Response Codes:

- **202** - if applicability regeneration is queued successfully
- **400** - if one or more of the parameters is invalid

:return:a *Call Report* representing the current state of the applicability regeneration

Sample Request:

```
{
 "consumer_criteria": {
  "filters": {"id": {"$in": ["sunflower", "voyager"]}}
 }
}
```

Tags: The task created will have the following tag: `"pulp:action:content_applicability_regeneration"`

Generate Content Applicability for Updated Repositories This API regenerates *applicability data* for a given set of repositories matched by a given *Search Criteria* asynchronously and saves it in the Pulp database. It should be used when a repository's content is updated. Only *existing* applicability data is regenerated for given repositories. If applicability data for a consumer-repository combination does not already exist, it should be generated using the API *Generate Content Applicability for Updated Consumers*.

If any new content types that support applicability are added to the given repositories, applicability data is generated for them as well. Generated applicability data can be queried using the *Query Content Applicability* API described below.

The API will return a *Call Report*. Users can check whether the applicability generation is completed using task id in the *Call Report*. You can run a single applicability generation task at a time. If an applicability generation task is running, any new applicability generation tasks requested are queued and postponed until the current task is completed.

Method: POST
Path: `/pulp/api/v2/repositories/actions/content/regenerate_applicability/`
Permission: create
Request Body Contents:

- **repo_criteria** (object) - a repository criteria object defined in *Search Criteria*

Response Codes:

- **202** - if applicability regeneration is queued successfully
- **400** - if one or more of the parameters is invalid

:return:a *Call Report* representing the current state of the applicability regeneration

Sample Request:

```
{
 "repo_criteria": {
  "filters": {"id": {"$in": ["test-repo", "test-errata"]}}
 }
}
```

Tags: The task created will have the following tag: `"pulp:action:content_applicability_regeneration"`

Generate Content Applicability for a single Consumer This API regenerates *applicability data* for the given consumer asynchronously and saves it in the Pulp database. It can be used by a consumer when its profile is updated or its consumer-repository bindings are updated. Applicability data is regenerated for all unit profiles associated with te given consumer and for all content types that define applicability. Generated applicability data can be queried using the *Query Content Applicability* API described above.

The API will return a *Call Report*. If an applicability generation task is running for a given consumer, any new applicability generation tasks requested are queued and postponed until the current task is completed.

Method: POST
Path: `/pulp/api/v2/consumers/<consumer_id>/actions/content/regenerate_applicability/`
Permission: create
Request Body Contents:

Response Codes:

- **202** - if applicability regeneration is queued successfully
- **404** - if a consumer with given consumer_id does not exist

:return:a *Call Report* representing the current state of the applicability regeneration

Tags: The task created will have the following tag: `"pulp:action:consumer_content_applicability_regeneration"`

Query Content Applicability This method queries Pulp for the applicability data that applies to a set of consumers matched by a given *Search Criteria*. The API user may also optionally specify an array of content types to which they wish to limit the applicability data.

Note: The criteria is used by this API to select the consumers for which Pulp needs to find applicability data. The `sort` option can be used in conjunction with `limit` and `skip` for pagination, but the `sort` option will not influence the ordering of the returned applicability reports since the consumers are collated together.

The applicability API will return an array of objects in its response. Each object will contain two keys, `consumers` and `applicability`. `consumers` will index an array of consumer ids. These grouped consumer ids will allow Pulp to collate consumers that have the same applicability together. `applicability` will index an object. The applicability object will contain content types as keys, and each content type will index an array of unit ids.

Each *applicability report* is an object:

- **consumers** - array of consumer ids
- **applicability - object with content types as keys, each indexing an** array of applicable unit ids

Method: POST
Path: `/pulp/api/v2/consumers/content/applicability/`
Permission: read
Request Body Contents:

- **criteria** (object) - a consumer criteria object defined in *Search Criteria*
- **content_types** (array) - an array of content types that the caller wishes to limit the applicability report to (optional)

Response Codes:

- **200** - if the applicability query was performed successfully
- **400** - if one or more of the parameters is invalid

Return: an array of applicability reports

Sample Request:

```
{
 "criteria": {
  "filters": {"id": {"$in": ["sunflower", "voyager"]}}
 },
 "content_types": ["type_1", "type_2"]
}
```

Sample 200 Response Body:

```
[
   {
       "consumers": ["sunflower"],
       "applicability": {"type_1": ["unit_1_id", "unit_2_id"]}
   },
   {
       "consumers": ["sunflower", "voyager"],
       "applicability": {"type_1": ["unit_3_id"], "type_2": ["unit_4_id"]}
   }
]
```

Consumer Group APIs

Create, Delete, and Update

Create a Consumer Group Creates a new consumer group. Group IDs must be unique across all consumer groups defined on the server. A group can be initialized with an array of consumers by passing their IDs to the create call. Consumers can be added or removed at anytime using the membership calls.

Method: POST
Path: `/pulp/api/v2/consumer_groups/`
Permission: create
Request Body Contents:

- **id** (string) - unique identifier of the consumer group
- **display_name** (string) - *(optional)* display-friendly name for the consumer group
- **description** (string) - *(optional)* description of the consumer group
- **consumer_ids** (array) - *(optional)* array of consumer ids initially associated with the group
- **notes** (object) - *(optional)* key-value pairs associated with the consumer group

Response Codes:

- **201** - consumer group successfully created
- **400** - if one or more of the parameters is invalid
- **409** - if a consumer group with the given id already exists

Return: representation of the consumer group resource

Sample Request:

```
{
 "id": "test-group",
 "description": "A test group of consumers",
 "consumer_ids": ["first-consumer", "second-consumer"]
}
```

Sample 201 Response Body:

```
{
 "_id": {"oid": "50407df0cf211b30c37c29f4"},
 "_ns": "consumer_groups",
 "_href": "/v2/consumer_groups/test-group/",
 "id": "test-group",
 "display_name": null,
 "description": "A test group of consumers",
 "consumer_ids": ["first-consumer", "second-consumer"],
 "notes": {}
}
```

Update a Consumer Group Metadata about the consumer group itself (not the members) can be updated with this call. Consumer members can be added or removed at anytime using the membership calls.

Method: PUT
Path: `/pulp/api/v2/consumer_groups/<consumer_group_id>/`
Permission: update
Request Body Contents:

- **display_name** (string) - *(optional)* same as in create call
- **description** (string) - *(optional)* same as in create call
- **notes** (object) - *(optional)* same as in create call

Response Codes:

- **200** - if the update executed immediately and was successful
- **400** - if one or more of the parameters is invalid
- **404** - if the group does not exist

Return: updated representation of the consumer group resource

Sample Request:

```
{
 "display_name": "new-name",
 "description": "new-description",
 "notes": {"new-note": "new-value"}
}
```

Sample 200 Response Body:

```
{
 "scratchpad": null,
 "display_name": "new-name",
 "description": "new-description",
 "_ns": "consumer_groups",
 "notes": {"new-note": "new-value"},
 "consumer_ids": [],
 "_id": {"$oid": "5344df87e5e71066d17aa889"},
 "id": "test-consumer-group",
 "_href": "/pulp/api/v2/consumer_groups/test-consumer-group/"
}
```

Delete a Consumer Group Deleting a consumer group has no effect on the consumers that are members of the group, apart from removing them from the group.

Method: DELETE
Path: `/pulp/api/v2/consumer_groups/<consumer_group_id>/`
Permission: delete
Response Codes:

- **200** - if the delete executed immediately and was successful
- **404** - if the specified group does not exist

Return: null

Group Membership

Consumers can be associated and unassociated with any existing consumer group at any time using the following REST API.

Associate a Consumer with a Group Associate the consumers specified by the *Search Criteria* with a consumer group. This call is idempotent; if a consumer is already a member of the group, no changes are made and no error is raised.

Method: POST
Path: `/pulp/api/v2/consumer_groups/<consumer_group_id>/actions/associate/`
Permission: execute
Request Body Contents:

- **criteria** (object) - criteria used to specify the consumers to associate

Response Codes:

- **200** - the consumers were successfully associated
- **400** - if one or more of the parameters is invalid
- **404** - if the group does not exist

Return: array of consumer IDs for all consumers in the group

Sample Request:

```
{
 "criteria": {
   "filters": {
     "id": {
       "$in": [
         "lab1",
         "lab2"
       ]
     }
   }
 }
}
```

Sample 200 Response Body:

```
["lab0", "lab1", "lab2"]
```

Unassociate a Consumer from a Group Unassociate the consumers specified by the *Search Criteria* from a consumer group. If a consumer satisfied by the criteria is not a member of the group, no changes are made and no error is raised.

Method: POST
Path: `/pulp/api/v2/consumer_groups/<consumer_group_id>/actions/unassociate/`
Permission: execute
Request Body Contents:

- **criteria** (object) - criteria used to specify the consumers to associate

Response Codes:

- **200** - the consumers were successfully unassociated
- **400** - if one or more of the parameters is invalid
- **404** - if the group does not exist

Return: array of consumer IDs for all consumers in the group

Sample Request:

```
{
 "criteria": {
   "filters": {
     "id": {
       "$in": [
         "lab1",
         "lab2"
       ]
     }
   }
 }
}
```

Sample 200 Response Body:

```
["lab0"]
```

Repository Binding

Bind a Consumer Group to a Repository Bind a *consumer* group to a *repository's distributor* for the purpose of consuming published content. Binding each consumer in the group is performed through the following steps:

1. Create each *binding* on server.
2. Send a request to each consumer to create the binding. A separate task is created for each unique combination of *consumer*, *repository*, and *distributor*.

The distributor may support configuration options that it may use for that particular binding. These options are used when generating the payload that is sent to consumers so they may access the repository. See the individual distributor's documentation for more information on the format.

Method: POST
Path: `/pulp/api/v2/consumer_groups/<group_id>/bindings/`
Permission: create
Request Body Contents:

- **repo_id** (string) - unique identifier for the repository
- **distributor_id** (string) - identifier for the distributor
- **options** (object) - *(optional)* options passed to the handler on each consumer
- **notify_agent** (boolean) - *(optional)* indicates if the consumer should be sent a message about the new binding; defaults to true if unspecified
- **binding_config** (object) - *(optional)* options to be used by the distributor for this binding

Response Codes:

- **202** - if the bind request was accepted
- **400** - if one or more of the parameters is invalid
- **404** - if the consumer group does not exist

Return: a *Call Report*

Sample Request:

```
{
  "repo_id": "test-repo",
  "distributor_id": "dist-1"
}
```

Tags: Each task created to add the binding to a *consumer* will be created with the following tags: `"pulp:repository:<repo_id>"`, `"pulp:consumer:<consumer_id>"` `"pulp:repository_distributor:<distributor-id>"` `"pulp:action:bind"`

Unbind a Consumer Group Remove a binding between each consumer in a *consumer* group and a *repository's distributor*.

Unbinding each consumer in the group is performed through the following steps:

1. Mark each *binding* as deleted on the server.
2. Send a request to each consumer to remove the binding.
3. Once each consumer has confirmed that the binding has been removed, it is permanently deleted on the server.

The steps for a forced unbind are as follows:

1. Each *binding* is deleted on the server.
2. Send a request to each consumer to remove the binding. The result of each consumer request discarded.

In either case step 2 results in a separate task created for each unique combination of *consumer*, *repository*, and *distributor*.

Method: DELETE
Path: `/pulp/api/v2/consumer_groups/<group_id>/bindings/<repo_id>/<distributor_id>`
Permission: delete
Query Parameters: The consumer ID, repository ID and distributor ID are included in the URL itself.

- **force** (boolean) - *(optional)* delete the binding immediately and discontinue tracking consumer actions
- **options** (object) - *(optional)* options passed to the handler on each consumer

Response Codes:

- **202** - the unbind request was accepted
- **400** - if one or more of the parameters is invalid
- **404** - if the consumer group, repository, or distributor does not exist

Return: a *Call Report*

Tags: Each task created to remove the binding from a *consumer* will be created with the following tags: `"pulp:repository:<repo_id>"`, `"pulp:consumer:<consumer_id>"` `"pulp:repository_distributor:<distributor-id>"` `"pulp:action:unbind"`

Content Management

Install Content on a Consumer Group Install one or more content units on each consumer in the group. This operation is asynchronous.

The units to be installed are specified in an array. Each unit in the array of *units* is an object containing two required attributes. The first is the **type_id** which a string that defines the unit's content type. The value is unrestricted by the Pulp server but must match a type mapped to a content *handler* in the agent. The second is the **unit_key** which identifies the unit or units to be installed. Both the structure and content are handler specific.

The caller can pass additional options using an *options* object. Both the structure and content are handler specific. The options drive how the handler performs the operation.

Method: POST
Path: `/pulp/api/v2/consumer_groups/<group_id>/actions/content/install/`
Permission: create
Request Body Contents:

- **units** (array) - array of content units to install
- **options** (object) - install options

Response Codes:

- **202** - the install request has been accepted
- **400** - if one or more of the parameters is invalid
- **404** - if the consumer group does not exist

Return: a *Call Report* that lists each of the tasks that were spawned.

Sample Request:

```
{
  "units": [
    {"unit_key": {"name": "zsh", "version": "4.3.17"}, "type_id": "rpm"},
    {"unit_key": {"id": "ERRATA-123"}, "type_id": "erratum"},
    {"unit_key": {"name": "web-server"}, "type_id": "package_group"}
  ],
  "options": {
    "apply": true, "reboot": false, "importkeys": false
  }
}
```

Tags: Each task created to install content on a *consumer* will be created with the following tags: `"pulp:consumer:<consumer_id>", "pulp:action:unit_install"`

Update Content on a Consumer Group Update one or more content units on each consumer in the group. This operation is asynchronous.

The units to be updated are specified in an array. Each unit in the array of *units* is an object containing two required attributes. The first is the **type_id** which a string that defines the unit's content type. The value is unrestricted by the Pulp server but must match a type mapped to a content *handler* in the agent. The second is the **unit_key** which identifies the unit or units to be updated. Both the structure and content are handler specific.

The caller can pass additional options using an *options* object. Both the structure and content are handler specific. The options drive how the handler performs the operation.

Method: POST
Path: `/pulp/api/v2/consumer_groups/<group_id>/actions/content/update/`

Permission: create
Request Body Contents:

- **units** (array) - array of content units to update
- **options** (object) - update options

Response Codes:

- **202** - the update request has been accepted
- **400** - if one or more of the parameters is invalid
- **404** - if the consumer group does not exist

Return: a *Call Report* that lists each of the tasks that were spawned.

Sample Request:

```
{
  "units": [
    {"unit_key": {"name": "zsh", "version": "4.3.17"}, "type_id": "rpm"},
    {"unit_key": {"id": "ERRATA-123"}, "type_id": "erratum"},
    {"unit_key": {"name": "web-server"}, "type_id": "package_group"}
  ],
  "options": {
    "apply": true, "reboot": false, "importkeys": false
  }
}
```

Tags: Each task created to update content on a *consumer* will be created with the following tags: `"pulp:consumer:<consumer_id>", "pulp:action:unit_update"`

Uninstall Content on a Consumer Group Uninstall one or more content units on each consumer in the group. This operation is asynchronous. If dependencies are automatically removed, it is reflected in the uninstall report.

The units to be uninstalled are specified in an array. Each unit in the array of *units* is an object containing two required attributes. The first is the **type_id** which a string that defines the unit's content type. The value is unrestricted by the Pulp server but must match a type mapped to a content *handler* in the agent. The second is the **unit_key** which identifies the unit or units to be uninstalled. The value is completely defined by the handler mapped to the unit's type_id.

The caller can pass additional options using an *options* object. Both the structure and content are handler specific. The options drive how the handler performs the operation.

Method: POST
Path: `/pulp/api/v2/consumer_groups/<group_id>/actions/content/uninstall/`
Permission: create
Request Body Contents:

- **units** (array) - array of content units to uninstall

- **options** (object) - uninstall options

Response Codes:

- **202** - the uninstall request has been accepted
- **400** - if one or more of the parameters is invalid
- **404** - if the consumer group does not exist

Return: a *Call Report* that lists each of the tasks that were spawned.

Sample Request:

```
{
  "units": [
    {"unit_key": {"name": "zsh", "version": "4.3.17"}, "type_id": "rpm"},
    {"unit_key": {"id": "ERRATA-123"}, "type_id": "erratum"},
    {"unit_key": {"name": "web-server"}, "type_id": "package_group"}
  ],
  "options": {
    "apply": true, "reboot": false
  }
}
```

Tags: Each task created to uninstall content on a *consumer* will be created with the following tags: `"pulp:consumer:<consumer_id>", "pulp:action:unit_uninstall"`

Repository APIs

Creation, Deletion, and Configuration

Create a Repository Creates a new repository in Pulp. This call accepts optional parameters for importer and distributor configuration. More detailed description of these parameters can be found below in the documentation of APIs to associate an importer or a distributor to an already existing repository. If these parameters are not passed, the call will only create the repository in Pulp. The real functionality of a repository isn't defined until importers and distributors are added. Repository IDs must be unique across all repositories in the server.

Method: POST
Path: `/pulp/api/v2/repositories/`
Permission: create
Request Body Contents:

- **id** (string) - unique identifier for the repository
- **display_name** (string) - *(optional)* user-friendly name for the repository
- **description** (string) - *(optional)* user-friendly text describing the repository's contents
- **notes** (object) - *(optional)* key-value pairs to programmatically tag the repository
- **importer_type_id** (string) - *(optional)* type id of importer being associated with the repository

- **importer_config** (object) - *(optional)* configuration the repository will use to drive the behavior of the importer
- **distributors** (array) - *(optional)* array of objects containing values of distributor_type_id, repo_plugin_config, auto_publish, and distributor_id

Response Codes:

- **201** - the repository was successfully created
- **400** - if one or more of the parameters is invalid
- **409** - if there is already a repository with the given ID
- **500** - if the importer or distributor raises an error during initialization

Return: database representation of the created repository

Sample Request:

```
{
 "display_name": "Harness Repository: harness_repo_1",
 "id": "harness_repo_1",
 "importer_type_id": "harness_importer",
 "importer_config": {
   "num_units": "5",
   "write_files": "true"
 },
 "distributors": [{
                "distributor_id": "dist_1",
                "distributor_type_id": "harness_distributor",
                "distributor_config": {
                "publish_dir": "/tmp/harness-publish",
                "write_files": "true"
                },
                "auto_publish": false
       }],
}
```

Sample 201 Response Body:

```
{
 "scratchpad": {},
 "display_name": "Harness Repository: harness_repo_1",
 "description": null,
 "_ns": "repos",
 "notes": {},
 "content_unit_counts": {},
 "_id": {
   "$oid": "52280416e5e71041ad000066"
 },
 "id": "harness_repo_1",
 "_href": "/pulp/api/v2/repositories/harness_repo_1/"
}
```

Update a Repository Much like create repository is simply related to the repository metadata (as compared to the associated importers/distributors), the update repository call is centered around updating only that metadata.

Method: PUT
Path: `/pulp/api/v2/repositories/<repo_id>/`
Permission: update
Request Body Contents: The body of the request is a JSON document with three possible root elements:

- **delta** (object) - object containing keys with values that should be updated on the repository
- **importer_config** (object) - *(optional)* object containing keys with values that should be updated on the repository's importer config
- **distributor_configs** (object) - *(optional)* object containing keys that are distributor ids, and values that are objects containing keys with values that should be updated on the specified distributor's config

Response Codes:

- **200** - if the update was executed and successful
- **202** - if the update was executed but additional tasks were created to update nested distributor configurations
- **400** - if one or more of the parameters is invalid
- **404** - if there is no repository with the give ID

Return: a *Call Report* containing the database representation of the repository (after changes made by the update) and any tasks spawned to apply the consumer bindings for the repository. See *Bind a Consumer to a Repository* for details on the bindings tasks that will be generated.

Sample Request:

```
{
 "delta": {
  "display_name" : "Updated"
 },
 "importer_config": {
  "demo_key": "demo_value"
 },
 "distributor_configs": {
  "demo_distributor": {
    "demo_key": "demo_value"
  }
 }
}
```

Sample result value: The result field of the *Call Report* contains the database representation of the repository

```
{
...
"result": {
   "display_name": "zoo",
   "description": "foo",
    "_ns": "repos",
    "notes": {
      "_repo-type": "rpm-repo"
    },
    "content_unit_counts": {
      "package_group": 2,
      "package_category": 1,
      "rpm": 32,
      "erratum": 4
    },
    "_id": {
      "$oid": "5328b2983738202945a3bb47"
    },
    "id": "zoo",
    "_href": "/pulp/api/v2/repositories/zoo/"

  },
  ...
}
```

Associate an Importer to a Repository Configures an *importer* for a previously created Pulp repository. Each repository maintains its own configuration for the importer which is used to dictate how the importer will function when it synchronizes content. The possible configuration values are contingent on the type of importer being added; each importer type will support a different set of values relevant to how it functions.

Only one importer may be associated with a repository at a given time. If a repository already has an associated importer, the previous association is removed. The removal is performed before the new importer is initialized, thus there is the potential that if the new importer initialization fails the repository is left without an importer.

Adding an importer performs the following validation steps before confirming the addition:

- The importer plugin is contacted and asked to validate the supplied configuration for the importer. If the importer indicates its configuration is invalid, the importer is not added to the repository.
- The importer's importer_added method is invoked to allow the importer to do any initialization required for that repository. If the plugin raises an exception during this call, the importer is not added to the repository.
- The Pulp database is updated to store the importer's configuration and the knowledge that the repository is associated with the importer.

The details of the added importer are returned from the call.

Method: POST
Path: `/pulp/api/v2/repositories/<repo_id>/importers/`
Permission: create
Request Body Contents:

- **importer_type_id** (string) - indicates the type of importer being associated with the repository; there must be an importer installed in the Pulp server with this ID
- **importer_config** (object) - configuration the repository will use to drive the behavior of the importer

Response Codes:

- **202** - if the association was queued to be performed
- **400** - if one or more of the required parameters is missing, the importer type ID refers to a non-existent importer, or the importer indicates the supplied configuration is invalid
- **404** - if there is no repository with the given ID
- **500** - if the importer raises an error during initialization

Return: a *Call Report* containing the current state of the association task

Sample Request:

```
{
 "importer_type_id": "harness_importer",
 "importer_config": {
   "num_units": "5",
   "write_files": "true"
 }
}
```

Sample result value for the Task Report: The result field of the *Task Report* will contain the database representation of the importer (not the full repository details, just the importer)

```
{
 "scratchpad": null,
 "_ns": "repo_importers",
 "importer_type_id": "harness_importer",
 "last_sync": null,
 "repo_id": "harness_repo_1",
 "_id": "bab0f9d5-dfd1-45ef-bd1d-fd7ea8077d75",
 "config": {
   "num_units": "5",
   "write_files": "true"
 },
 "id": "harness_importer"
}
```

Tags: The task created will have the following tags: `"pulp:action:update_importer"`, `"pulp:repository:<repo_id>"`, `"pulp:repository_importer:<importer_type_id>`

Associate a Distributor with a Repository Configures a *distributor* for a previously created Pulp repository. Each repository maintains its own configuration for the distributor which is used to dictate how the distributor will function when it publishes content. The possible configuration values are contingent on the type of distributor being added; each distributor type will support a different set of values relevant to how it functions.

Multiple distributors may be associated with a repository at a given time. There may be more than one distributor with the same type. The only restriction is that the distributor ID must be unique across all distributors for a given repository.

Adding a distributor performs the following validation steps before confirming the addition:

- If provided, the distributor ID is checked for uniqueness in the context of the repository. If not provided, a unique ID is generated.
- The distributor plugin is contacted and asked to validate the supplied configuration for the distributor. If the distributor indicates its configuration is invalid, the distributor is not added to the repository.

- The distributor's distributor_added method is invoked to allow the distributor to do any initialization required for that repository. If the plugin raises an exception during this call, the distributor is not added to the repository.
- The Pulp database is updated to store the distributor's configuration and the knowledge that the repository is associated with the distributor.

The details of the added distributor are returned from the call.

Method: POST
Path: `/pulp/api/v2/repositories/<repo_id>/distributors/`
Permission: create
Request Body Contents:

- **distributor_type_id** (string) - indicates the type of distributor being associated with the repository; there must be a distributor installed in the Pulp server with this ID
- **distributor_config** (object) - configuration the repository will use to drive the behavior of the distributor
- **distributor_id** (string) - *(optional)* if specified, this value will be used to refer to the distributor; if not specified, one will be randomly assigned to the distributor
- **auto_publish** (boolean) - *(optional)* if true, this distributor will automatically have its publish operation invoked after a successful repository sync. Defaults to false if unspecified

Response Codes:

- **201** - if the distributor was successfully added
- **400** - if one or more of the required parameters is missing, the distributor type ID refers to a non-existent distributor, or the distributor indicates the supplied configuration is invalid
- **404** - if there is no repository with the given ID
- **500** - if the distributor raises an error during initialization

Return: database representation of the distributor (not the full repository details, just the distributor)

Sample Request:

```
{
 "distributor_id": "dist_1",
 "distributor_type_id": "harness_distributor",
 "distributor_config": {
   "publish_dir": "/tmp/harness-publish",
   "write_files": "true"
 },
 "auto_publish": false
}
```

Sample 201 Response Body:

```
{
 "scratchpad": null,
 "_ns": "repo_distributors",
 "last_publish": null,
 "auto_publish": false,
 "distributor_type_id": "harness_distributor",
 "repo_id": "harness_repo_1",
 "publish_in_progress": false,
 "_id": "cfdd6ab9-6dbe-4192-bde2-d00db768f268",
 "config": {
   "publish_dir": "/tmp/harness-publish",
   "write_files": "true"
 },
 "id": "dist_1"
}
```

Update an Importer Associated with a Repository Update the configuration for an *importer* that has already been associated with a repository.

Any importer configuration value that is not specified remains unchanged.

Method: PUT
Path: `/pulp/api/v2/repositories/<repo_id>/importers/<importer_id>/`
Permission: update
Request Body Contents:

- **importer_config** (object) - object containing keys with values that should be updated on the importer

Response Codes:

- **202** - if the request was accepted by the server to update the importer when the repository is available
- **404** - if there is no repository or importer with the specified IDs

Return: a *Call Report* which includes a spawned task that should be polled for a *Task Report*

Sample Request:

```
{
  "importer_config": {
    "demo_key": "demo_value"
  }
}
```

Sample result value for the Task Report: The result field of the *Task Report* contains the database representation of the importer. This does not include the full repository details.

```
{
  "scratchpad": null,
  "_ns": "repo_importers",
  "importer_type_id": "demo_importer",
  "last_sync": "2013-10-03T14:08:35Z",
  "scheduled_syncs": [],
  "repo_id": "demo_repo",
  "_id": {
    "$oid": "524db282dd01fb194283e53f"
  },
  "config": {
    "demo_key": "demo_value"
  },
  "id": "demo_importer"
}
```

Tags: The task created will have the following tags: `"pulp:action:update_importer"`, `"pulp:repository:<repo_id>"`, `"pulp:repository_importer:<importer_id>`

Disassociate an Importer from a Repository

Method: DELETE
Path: `/pulp/api/v2/repositories/<repo_id>/importers/<importer_id>/`
Permission: delete

Response Codes:

- **202** - if the request was accepted by the server to disassociate when the repository is available
- **404** - if there is no repository or importer with the specified IDs

Return: a *Call Report*

Tags: The task created will have the following tags: `"pulp:action:delete_importer"`, `"pulp:repository:<repo_id>"`, `"pulp:repository_importer:<importer_id>`

Update a Distributor Associated with a Repository Update the configuration for a *distributor* that has already been associated with a repository. This performs the following actions:

1. Updates the distributor on the server.
2. Rebinds any bound consumers.

Any distributor configuration value that is not specified remains unchanged.

The first step is represented by a *Call Report*. Upon completion of step 1 the spawned_tasks field will be populated with links to any tasks required to complete step 2. Updating a distributor causes each binding associated with that repository to be updated as well. See *Bind a Consumer to a Repository* for details.

Method: PUT
Path: `/pulp/api/v2/repositories/<repo_id>/distributors/<distributor_id>/`
Permission: update

Response Codes:

- **202** - if the request was accepted by the server to update the distributor when the repository is available

- **404** - if there is no repository or distributor with the specified IDs

Return: a *Call Report*

Sample Request:

```
{
 "distributor_config": {
   "demo_key": "demo_value"
 },
 "delta": {
   "auto_publish": true
 }
}
```

Tags: The task created to update the distributor will have the following tags: `"pulp:action:update_distributor", "pulp:repository:<repo_id>", "pulp:repository_distributor:<distributor_id>` Information about the binding tasks can be found at *Bind a Consumer to a Repository*.

Disassociate a Distributor from a Repository Disassociating a distributor performs the following actions:

1. Remove the association between the distributor and the repository.
2. Unbind all bound consumers.

The first step is represented by a *Call Report*. Upon completion of step 1 the spawned_tasks field will be populated with links to any tasks required complete step 2. The total number of spawned tasks depends on how many consumers are bound to the repository.

Method: DELETE
Path: `/pulp/api/v2/repositories/<repo_id>/distributors/<distributor_id>/`
Permission: delete

Response Codes:

- **202** - if the request was accepted by the server to disassociate when the repository is available
- **404** - if there is no repository or distributor with the specified IDs
- **500** - if the server raises an error during disassociation

Return: a *Call Report*

Tags: The task created to delete the distributor will have the following tags: `"pulp:action:remove_distributor","pulp:repository:<repo_id>", "pulp:repository_distributor:<distributor_id>`

Delete a Repository When a repository is deleted, it is removed from the database and its local working directory is deleted. The content within the repository, however, is not deleted. Deleting content is handled through the orphaned unit process.

Deleting a repository is performed in the following major steps:

1. Delete the repository.
2. Unbind all bound consumers.

The first step is represented by a *Call Report*. Upon completion of step 1 the spawned_tasks field will be populated with links to any tasks required to complete step 2. The total number of spawned tasks depends on how many consumers are bound to the repository.

Method: DELETE
Path: `/pulp/api/v2/repositories/<repo_id>/`
Permission: delete
Response Codes:

- **202** - if the request was accepted by the server to delete the repository
- **404** - if the requested repository does not exist

Return: a *Call Report*

Tags: The task created to delete the repository will have the following tags: `"pulp:action:delete","pulp:repository:<repo_id>"`

Retrieval

Retrieve a Single Repository Retrieves information on a single Pulp repository. The returned data includes general repository metadata and metadata describing any *importers* and *distributors* associated with it.

Method: GET
Path: `/pulp/api/v2/repositories/<repo_id>/`
Permission: read

Query Parameters:

- **details** (boolean) - *(optional)* shortcut for including both distributors and importers
- **importers** (boolean) - *(optional)* include the "importers" attribute on each repository
- **distributors** (boolean) - *(optional)* include the "distributors" attribute on each repository

Response Codes:

- **200** - if the repository exists

- **404** - if no repository exists with the given ID

Return: database representation of the matching repository

Sample 200 Response Body:

```
{
 "display_name": "Harness Repository: harness_repo_1",
 "description": null,
 "distributors": [
   {
     "scratchpad": 1,
     "_ns": "repo_distributors",
     "last_publish": "2012-01-25T15:26:32Z",
     "auto_publish": false,
     "distributor_type_id": "harness_distributor",
     "repo_id": "harness_repo_1",
     "publish_in_progress": false,
     "_id": "addf9261-345e-4ce3-ad1e-436ba005287f",
     "config": {
       "publish_dir": "/tmp/harness-publish",
       "write_files": "true"
     },
     "id": "dist_1"
   }
 ],
 "notes": {},
 "content_unit_counts": {},
 "last_unit_added": "2012-01-25T15:26:32Z",
 "last_unit_removed": "2012-01-25T15:26:32Z",
 "importers": [
   {
     "scratchpad": 1,
     "_ns": "repo_importers",
     "importer_type_id": "harness_importer",
     "last_sync": "2012-01-25T15:26:32Z",
     "repo_id": "harness_repo_1",
     "sync_in_progress": false,
     "_id": "bbe81308-ef7c-4c0c-b684-385fd627d99e",
     "config": {
       "num_units": "5",
       "write_files": "true"
     },
     "id": "harness_importer"
   }
 ],
 "id": "harness_repo_1"
}
```

Retrieve All Repositories Returns information on all repositories in the Pulp server. It is worth noting that this call will never return a 404; an empty array is returned in the case where there are no repositories.

Method: GET
Path: `/pulp/api/v2/repositories/`

Permission: read
Query Parameters:

- **details** (boolean) - *(optional)* shortcut for including both distributors and importers
- **importers** (boolean) - *(optional)* include the "importers" attribute on each repository
- **distributors** (boolean) - *(optional)* include the "distributors" attribute on each repository

Response Codes:

- **200** - containing the array of repositories

Return: the same format as retrieving a single repository, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
 {
   "display_name": "Harness Repository: harness_repo_1",
   "description": null,
   "last_unit_added": "2012-01-25T15:26:32Z",
   "last_unit_removed": null,
   "distributors": [
     {
       "scratchpad": 1,
       "_ns": "repo_distributors",
       "last_publish": "2012-01-25T15:26:32Z",
       "auto_publish": false,
       "distributor_type_id": "harness_distributor",
       "repo_id": "harness_repo_1",
       "publish_in_progress": false,
       "_id": "addf9261-345e-4ce3-ad1e-436ba005287f",
       "config": {
         "publish_dir": "/tmp/harness-publish",
         "write_files": "true"
       },
       "id": "dist_1"
     }
   ],
   "notes": {},
   "content_unit_counts": {},
   "importers": [
     {
       "scratchpad": 1,
       "_ns": "repo_importers",
       "importer_type_id": "harness_importer",
       "last_sync": "2012-01-25T15:26:32Z",
       "repo_id": "harness_repo_1",
       "sync_in_progress": false,
       "_id": "bbe81308-ef7c-4c0c-b684-385fd627d99e",
       "config": {
         "num_units": "5",
         "write_files": "true"
```

```
        },
        "id": "harness_importer"
      }
    ],
    "id": "harness_repo_1"
  }
]
```

Advanced Search for Repositories Please see *Search API* for more details on how to perform these searches.

Returns information on repositories in the Pulp server that match your search parameters. It is worth noting that this call will never return a 404; an empty array is returned in the case where there are no repositories.

Method: POST
Path: `/pulp/api/v2/repositories/search/`
Permission: read
Request Body Contents:

- **importers** (boolean) - *(optional)* include the "importers" attribute on each repository
- **distributors** (boolean) - *(optional)* include the "distributors" attribute on each repository

Response Codes:

- **200** - containing the array of repositories

Return: the same format as retrieving a single repository, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
  {
    "display_name": "Harness Repository: harness_repo_1",
    "description": null,
    "distributors": [
      {
        "scratchpad": 1,
        "_ns": "repo_distributors",
        "last_publish": "2012-01-25T15:26:32Z",
        "auto_publish": false,
        "distributor_type_id": "harness_distributor",
        "repo_id": "harness_repo_1",
        "publish_in_progress": false,
        "_id": "addf9261-345e-4ce3-ad1e-436ba005287f",
        "config": {
          "publish_dir": "/tmp/harness-publish",
          "write_files": "true"
        },
        "id": "dist_1"
      }
    ],
    "notes": {},
    "content_unit_counts": {},
    "last_unit_added": null,
    "last_unit_removed": null,
    "importers": [
      {
        "scratchpad": 1,
        "_ns": "repo_importers",
        "importer_type_id": "harness_importer",
        "last_sync": "2012-01-25T15:26:32Z",
        "repo_id": "harness_repo_1",
        "sync_in_progress": false,
        "_id": "bbe81308-ef7c-4c0c-b684-385fd627d99e",
        "config": {
          "num_units": "5",
          "write_files": "true"
        },
        "id": "harness_importer"
      }
    ],
    "id": "harness_repo_1"
  }
]
```

Returns information on repositories in the Pulp server that match your search parameters. It is worth noting that this call will never return a 404; an empty array is returned in the case where there are no repositories.

This method is slightly more limiting than the POST alternative, because some filter expressions may not be serializable as query parameters.

Method: GET
Path: `/pulp/api/v2/repositories/search/`
Permission: read
Query Parameters: query params should match the attributes of a Criteria object as defined in *Search Criteria*. The exception is the 'fields' parameter, which should be specified in singular form as follows: For example: /v2/repositories/search/?field=id&field=display_name&limit=20'

- **details** (boolean) - *(optional)* shortcut for including both distributors and importers
- **importers** (boolean) - *(optional)* include the "importers" attribute on each repository
- **distributors** (boolean) - *(optional)* include the "distributors" attribute on each repository

Response Codes:

- **200** - containing the array of repositories

Return: the same format as retrieving a single repository, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
 {
   "display_name": "Harness Repository: harness_repo_1",
   "description": null,
   "distributors": [
     {
       "scratchpad": 1,
       "_ns": "repo_distributors",
       "last_publish": "2012-01-25T15:26:32Z",
       "auto_publish": false,
       "distributor_type_id": "harness_distributor",
       "repo_id": "harness_repo_1",
       "publish_in_progress": false,
       "_id": "addf9261-345e-4ce3-ad1e-436ba005287f",
       "config": {
         "publish_dir": "/tmp/harness-publish",
         "write_files": "true"
       },
       "id": "dist_1"
     }
   ],
   "notes": {},
   "content_unit_counts": {},
   "last_unit_added": null,
   "last_unit_removed": null,
   "importers": [
     {
       "scratchpad": 1,
       "_ns": "repo_importers",
       "importer_type_id": "harness_importer",
       "last_sync": "2012-01-25T15:26:32Z",
       "repo_id": "harness_repo_1",
       "sync_in_progress": false,
       "_id": "bbe81308-ef7c-4c0c-b684-385fd627d99e",
       "config": {
         "num_units": "5",
         "write_files": "true"
       },
       "id": "harness_importer"
     }
   ],
   "id": "harness_repo_1"
 }
]
```

Retrieve Importers Associated with a Repository Retrieves the *importer* (if any) associated with a repository. The array will either be empty (no importer configured) or contain a single entry.

Method: GET
Path: `/pulp/api/v2/repositories/<repo_id>/importers/`
Permission: read
Query Parameters: None
Response Codes:

- **200** - containing an array of importers
- **404** - if there is no repository with the given ID; this will not occur if the repository exists but has no associated importers

Return: database representation of the repository's importer or an empty list

Sample 200 Response Body:

```
[
 {
   "scratchpad": 1,
   "_ns": "repo_importers",
   "importer_type_id": "harness_importer",
   "last_sync": "2012-01-25T15:26:32Z",
   "repo_id": "harness_repo_1",
   "sync_in_progress": false,
   "_id": "bbe81308-ef7c-4c0c-b684-385fd627d99e",
   "config": {
     "num_units": "5",
     "write_files": "true"
   },
   "id": "harness_importer"
 }
]
```

Retrieve an Importer Associated with a Repository Retrieves the given *importer* (if any) associated with a repository.

Method: GET
Path: `/pulp/api/v2/repositories/<repo_id>/importers/<importer_id>/`
Permission: read
Query Parameters: None
Response Codes:

- **200** - containing the details of the importer
- **404** - if there is either no repository or importer with a matching ID.

Return: database representation of the repository's importer

Sample 200 Response Body:

```
{
  "scratchpad": 1,
  "_ns": "repo_importers",
  "importer_type_id": "harness_importer",
  "last_sync": "2012-01-25T15:26:32Z",
  "repo_id": "harness_repo_1",
  "sync_in_progress": false,
  "_id": {"$oid": "bbe81308-ef7c-4c0c-b684-385fd627d99e"},
  "config": {
    "num_units": "5",
    "write_files": "true"
  },
  "id": "harness_importer"
}
```

Retrieve Distributors Associated with a Repository Retrieves all *distributors* associated with a repository. If the repository has no associated distributors, an empty array is returned.

Method: GET
Path: `/pulp/api/v2/repositories/<repo_id>/distributors/`
Permission: read
Query Parameters: None
Response Codes:

- **200** - containing an array of distributors
- **404** - if there is no repository with the given ID; this will not occur if the repository exists but has no associated distributors

Return: database representations of all distributors on the repository

Sample 200 Response Body:

```
[
 {
   "scratchpad": 1,
   "_ns": "repo_distributors",
   "last_publish": "2012-01-25T15:26:32Z",
   "auto_publish": false,
   "distributor_type_id": "harness_distributor",
   "repo_id": "harness_repo_1",
   "publish_in_progress": false,
   "_id": "addf9261-345e-4ce3-ad1e-436ba005287f",
   "config": {
     "publish_dir": "/tmp/harness-publish",
     "write_files": "true"
   },
   "id": "dist_1"
 }
]
```

Retrieve a Distributor Associated with a Repository Retrieves a single *distributors* associated with a repository.

Method: GET
Path: `/pulp/api/v2/repositories/<repo_id>/distributors/<distributor_id>/`
Permission: read

Query Parameters: None
Response Codes:

- **200** - containing the details of a distributors
- **404** - if there is either no repository or distributor with a matching ID.

Return: database representation of the distributor

Sample 200 Response Body:

```
{
  "scratchpad": 1,
  "_ns": "repo_distributors",
  "last_publish": "2012-01-25T15:26:32Z",
  "auto_publish": false,
  "distributor_type_id": "harness_distributor",
  "repo_id": "harness_repo_1",
  "publish_in_progress": false,
  "_id": {"$oid": "addf9261-345e-4ce3-ad1e-436ba005287f"},
  "config": {
    "publish_dir": "/tmp/harness-publish",
    "write_files": "true"
  },
  "id": "dist_1"
}
```

Synchronization

Sync a Repository Syncs content into a repository from a feed source using the repository's *importer*.

Method: POST
Path: `/pulp/api/v2/repositories/<repo_id>/actions/sync/`
Permission: execute
Request Body Contents:

- **override_config** (object) - importer configuration values that override the importer's default configuration for this sync

Response Codes:

- **202** - if the sync is set to be executed
- **404** - if repo does not exist

Return: a *Call Report*

Sample Request:

```
{
  "override_config": {"verify_checksum": false,
                      "verify_size": false},
}
```

Tags: The task created will have the following tags: `"pulp:action:sync"`, `"pulp:repository:<repo_id>"`

Scheduling a Sync A repository can be synced automatically using an *iso8601 interval*. To create a scheduled sync, the interval, sync override config, and other schedule options must be set on the repository's *importer*.

Method: POST
Path: `/pulp/api/v2/repositories/<repo_id>/importers/<importer_id>/schedules/sync/`
Permission: create
Request Body Contents:

- **schedule** (string) - the schedule as an iso8601 interval
- **override_config** (object) - *(optional)* the overridden configuration for the importer to be used on the scheduled sync
- **failure_threshold** (number) - *(optional)* consecutive failures allowed before this scheduled sync is disabled
- **enabled** (boolean) - *(optional)* whether the scheduled sync is initially enabled (defaults to true)

Response Codes:

- **201** - if the schedule was successfully created
- **400** - if one or more of the parameters are invalid
- **404** - if there is no repository or importer with the specified IDs

Return: schedule report representing the current state of the scheduled call

Sample Request:

```
{
 "override_config": {},
 "schedule": "00:00:00Z/P1DT",
 "failure_threshold": 3,
}
```

Sample 201 Response Body:

```
{
 "next_run": "2014-01-27T21:41:50Z",
 "task": "pulp.server.tasks.repository.sync_with_auto_publish",
 "last_updated": 1390858910.292712,
 "first_run": "2014-01-27T21:41:50Z",
 "schedule": "PT1H",
 "args": [
   "demo"
 ],
 "enabled": true,
 "last_run_at": null,
 "_id": "52e6d29edd01fb70bd0d9c37",
 "total_run_count": 0,
 "failure_threshold": 3,
 "kwargs": {
   "overrides": {}
 },
 "resource": "pulp:importer:demo:puppet_importer",
 "remaining_runs": null,
 "consecutive_failures": 0,
 "_href": "/pulp/api/v2/repositories/demo/importers/puppet_importer/schedules/sync/52e6d29edd01fb70bc
}
```

Updating a Scheduled Sync The same parameters used to create a scheduled sync may be updated at any point.

Method: PUT
Path: `/pulp/api/v2/repositories/<repo_id>/importers/<importer_id>/schedules/sync/<schedule_id>/`
Permission: create
Request Body Contents:

- **schedule** (string) - *(optional)* new schedule as an iso8601 interval
- **override_config** (object) - *(optional)* new overridden configuration for the importer to be used on the scheduled sync
- **failure_threshold** (number) - *(optional)* new consecutive failures allowed before this scheduled sync is disabled
- **enabled** (boolean) - *(optional)* whether the scheduled sync is enabled

Response Codes:

- **200** - if the schedule was successfully updated
- **400** - if one or more of the parameters are invalid
- **404** - if there is no repository, importer or schedule with the specified IDs

Return: schedule report representing the current state of the scheduled call (see sample response of Scheduling a Sync for details)

Deleting a Scheduled Sync Delete a scheduled sync to remove it permanently from the importer.

Method: DELETE

Path:
`/pulp/api/v2/repositories/<repo_id>/importers/<importer_id>/schedules/sync/<schedule_id>/`
Permission: delete

Response Codes:

- **200** - if the schedule was deleted successfully
- **404** - if there is no repository, importer or schedule with the specified IDs

Return: null

Listing All Scheduled Syncs All of the scheduled syncs for a given importer may be listed.

Method: GET
Path:
`/pulp/api/v2/repositories/<repo_id>/importers/<importer_id>/schedules/sync/`
Permission: read

Response Codes:

- **200** - if repo, importer exist
- **404** - if there is no repository or importer with the specified IDs

Return: array of schedule reports for all scheduled syncs defined

Sample 200 Response Body:

```
[
   {
       "_href": "/pulp/api/v2/repositories/test/importers/yum_importer/schedules/sync/54d8852245ef48
       "_id": "54d8852245ef4876fade7cc2",
       "args": [
           "test"
       ],
       "consecutive_failures": 0,
       "enabled": true,
       "failure_threshold": null,
       "first_run": "2015-02-09T10:00:02Z",
       "kwargs": {
           "overrides": {}
       },
       "last_run_at": "2015-02-09T10:00:23Z",
       "last_updated": 1423476133.825821,
       "next_run": "2015-02-10T10:00:02Z",
       "remaining_runs": null,
       "resource": "pulp:importer:test:yum_importer",
       "schedule": "P1DT",
       "task": "pulp.server.tasks.repository.sync_with_auto_publish",
       "total_run_count": 1
   }
]
```

Listing a Single Scheduled Sync Each scheduled sync may be inspected.

Method: GET
Permission: read
Path:
`/pulp/api/v2/repositories/<repo_id>/importers/<importer_id>/schedules/sync/<schedule_id>/`

Response Codes:

- **200** - if repo, importer, schedule exist
- **404** - if there is no repository, importer or schedule with the specified IDs

Return: a schedule report for the scheduled sync

Sample 200 Response Body:

```
{
    "_href": "/pulp/api/v2/repositories/test/importers/yum_importer/schedules/sync/54d8852245ef4876fa
    "_id": "54d8852245ef4876fade7cc2",
    "args": [
        "test"
    ],
    "consecutive_failures": 0,
    "enabled": true,
    "failure_threshold": null,
    "first_run": "2015-02-09T10:00:02Z",
    "kwargs": {
        "overrides": {}
    },
    "last_run_at": "2015-02-09T10:00:23Z",
    "last_updated": 1423476133.825821,
    "next_run": "2015-02-10T10:00:02Z",
    "remaining_runs": null,
    "resource": "pulp:importer:test:yum_importer",
    "schedule": "P1DT",
    "task": "pulp.server.tasks.repository.sync_with_auto_publish",
    "total_run_count": 1
}
```

Retrieving Sync History Retrieve sync history for a repository. Each sync performed on a repository creates a history entry.

Method: GET
Permission: read
Path: `/pulp/api/v2/repositories/<repo_id>/history/sync/`

Query Parameters:

- **limit** (integer) - *(optional)* the maximum number of history entries to return; if not specified, the entire history is returned
- **sort** (string) - *(optional)* options are 'ascending' and 'descending'; the array is sorted by the sync timestamp
- **start_date** (iso8601 datetime) - *(optional)* any entries with a timestamp prior to the given date are not returned
- **end_date** (iso8601 datetime) - *(optional)* any entries with a timestamp after the given date are not returned

Response Codes:

- **200** - if the history was successfully retrieved
- **404** - if the repository id given does not exist

Return: an array of sync history entries

Sample 200 Response Body:

```
[
 {
  "result": "success",
  "importer_id": "my_demo_importer",
  "exception": null,
  "repo_id": "demo_repo",
  "traceback": null,
  "started": "1970:00:00T00:00:00Z",
  "completed": "1970:00:00T00:00:01Z",
  "importer_type_id": "demo_importer",
  "error_message": null,
 }
]
```

Publication

Publish a Repository Publish content from a repository using a repository's *distributor*. This call always executes asynchronously and will return a *call report*.

Method: POST
Path: `/pulp/api/v2/repositories/<repo_id>/actions/publish/`
Permission: execute
Request Body Contents:

- **id** (string) - identifies which distributor on the repository to publish
- **override_config** (object) - *(optional)* distributor configuration values that override the distributor's default configuration for this publish

Response Codes:

- **202** - if the publish is set to be executed
- **404** - if repo does not exist

Return: a *Call Report* representing the current state of the sync

Sample Request:

```
{
  "id": "distributor_1",
  "override_config": {},
}
```

Tags: The task created will have the following tags: `"pulp:action:publish"`,`"pulp:repository:<repo_id>"`

Scheduling a Publish A repository can be published automatically using an *iso8601 interval*. To create a scheduled publish, the interval, publish override config, and other schedule options must be set on a repository's *distributor*.

Method: POST
Path: `/pulp/api/v2/repositories/<repo_id>/distributors/<distributor_id>/schedules/publish/`
Permission: create
Request Body Contents:

- **schedule** (string) - the schedule as an iso8601 interval
- **override_config** (object) - *(optional)* the overridden configuration for the distributor to be used on the scheduled publish
- **failure_threshold** (number) - *(optional)* consecutive failures allowed before this scheduled publish is disabled
- **enabled** (boolean) - *(optional)* whether the scheduled publish is initially enabled (defaults to true)

Response Codes:

- **201** - if the schedule was successfully created
- **400** - if one or more of the parameters are invalid
- **404** - if there is no repository or distributor with the specified IDs

Return: schedule report representing the current state of the scheduled call

Sample Request:

```
{
 "override_config": {},
 "schedule": "PT1H",
 "failure_threshold": 3,
}
```

Sample 201 Response Body:

```
{
 "next_run": "2014-01-27T21:27:56Z",
 "task": "pulp.server.tasks.repository.publish",
 "last_updated": 1390858076.682694,
 "first_run": "2014-01-27T21:27:56Z",
 "schedule": "PT1H",
 "args": [
   "demo",
   "puppet_distributor"
 ],
 "enabled": true,
 "last_run_at": null,
 "_id": "52e6cf5cdd01fb70bd0d9c34",
 "total_run_count": 0,
 "failure_threshold": 3,
 "kwargs": {
   "overrides": {}
 },
 "resource": "pulp:distributor:demo:puppet_distributor",
 "remaining_runs": null,
 "consecutive_failures": 0,
 "_href": "/pulp/api/v2/repositories/demo/distributors/puppet_distributor/schedules/publish/52e6cf5cc
}
```

Updating a Scheduled Publish The same parameters used to create a scheduled publish may be updated at any point.

Method: PUT
Path: `/pulp/api/v2/repositories/<repo_id>/distributors/<distributor_id>/schedules/publish/<schedu`
Permission: create
Request Body Contents:

- **schedule** (string) - *(optional)* new schedule as an iso8601 interval
- **override_config** (object) - *(optional)* new overridden configuration for the importer to be used on the scheduled sync
- **failure_threshold** (number) - *(optional)* new consecutive failures allowed before this scheduled sync is disabled
- **enabled** (boolean) - *(optional)* whether the scheduled sync is enabled

Response Codes:

- **200** - if the schedule was successfully updated
- **400** - if one or more of the parameters are invalid
- **404** - if there is no repository, distributor or schedule with the specified IDs

Return: schedule report representing the current state of the scheduled call (see sample response of Scheduling a Publish for details)

Deleting a Scheduled Publish Delete a scheduled publish to remove it permanently from the distributor.

Method: DELETE
Path:
`/pulp/api/v2/repositories/<repo_id>/distributors/<distributor_id>/schedules/publish/<schedu`
Permission: delete

Response Codes:

- **200** - if the schedule was deleted successfully
- **404** - if there is no repository, distributor or schedule with the specified IDs

Return: null

Listing All Scheduled Publishes All of the scheduled publishes for a given distributor may be listed.

Method: GET
Path:
`/pulp/api/v2/repositories/<repo_id>/distributors/<distributor_id>/schedules/publish/`
Permission: read

Response Codes:

- **200** - if repo, distributor exist
- **404** - if there is no repository or distributor with the specified IDs

Return: array of schedule reports for all scheduled publishes defined (see sample response of Scheduling a Publish for details)

Sample 200 Response Body:

```
{
    "_href": "/pulp/api/v2/repositories/test/distributors/yum_distributor/schedules/publish/54d88df04
    "_id": "54d88df045ef4876fb50c994",
    "args": [
        "test",
        "yum_distributor"
    ],
    "consecutive_failures": 0,
    "enabled": true,
    "failure_threshold": null,
    "first_run": "2015-02-09T10:37:36Z",
    "kwargs": {
        "overrides": {}
    },
    "last_run_at": "2015-02-09T10:38:23Z",
    "last_updated": 1423478256.805917,
    "next_run": "2015-02-10T10:37:36Z",
    "remaining_runs": null,
    "resource": "pulp:distributor:test:yum_distributor",
    "schedule": "P1DT",
    "task": "pulp.server.tasks.repository.publish",
    "total_run_count": 1
}
```

]

Listing a Single Scheduled Publish Each scheduled publish may be inspected.

Method: GET
Permission: read
Path:
`/pulp/api/v2/repositories/<repo_id>/distributors/<distributor_id>/schedules/publish/<schedu`

Response Codes:

- **200** - if repo, distributor or schedule exist
- **404** - if there is no repository, distributor or schedule with the specified IDs

Return: a schedule report for the scheduled publish (see sample response of Scheduling a Publish for details)

Sample 200 Response Body:

```
{
   "_href": "/pulp/api/v2/repositories/test/distributors/yum_distributor/schedules/publish/54d88df045
   "_id": "54d88df045ef4876fb50c994",
   "args": [
       "test",
       "yum_distributor"
   ],
   "consecutive_failures": 0,
   "enabled": true,
    "failure_threshold": null,
    "first_run": "2015-02-09T10:37:36Z",
    "kwargs": {
        "overrides": {}
    },
    "last_run_at": "2015-02-09T10:38:23Z",
    "last_updated": 1423478256.805917,
    "next_run": "2015-02-10T10:37:36Z",
    "remaining_runs": null,
    "resource": "pulp:distributor:test:yum_distributor",
    "schedule": "P1DT",
    "task": "pulp.server.tasks.repository.publish",
    "total_run_count": 1
}
```

Retrieving Publish History Retrieve publish history for a repository. Each publish performed on a repository creates a history entry.

Method: GET
Permission: read
Path: `/pulp/api/v2/repositories/<repo_id>/history/publish/<distributor_id>/`

Query Parameters:

- **limit** (integer) - *(optional)* the maximum number of history entries to return; if not specified, the entire history is returned
- **sort** (string) - *(optional)* options are 'ascending' and 'descending'; the array is sorted by the publish timestamp
- **start_date** (iso8601 datetime) - *(optional)* any entries with a timestamp prior to the given date are not returned
- **end_date** (iso8601 datetime) - *(optional)* any entries with a timestamp after the given date are not returned

Response Codes:

- **200** - if the history was successfully retrieved
- **404** - if the repository id given does not exist

Return: an array of publish history entries

Sample 200 Response Body:

```
[
 {
  "result": "success",
  "distributor_id": "my_demo_distributor",
  "distributor_type_id": "demo_distributor",
  "exception": null,
  "repo_id": "demo_repo",
  "traceback": null,
  "started": "1970:00:00T00:00:00Z",
  "completed": "1970:00:00T00:00:01Z",
  "error_message": null,
 }
]
```

Content Retrieval

Advanced Unit Search A *Unit Association Criteria* can be used to search for units within a repository.

Method: POST
Path: `/pulp/api/v2/repositories/<repo_id>/search/units/`
Permission: read
Request Body Contents:

- **criteria** (object) - a UnitAssociationCriteria

Response Codes:

- **200** - if the search executed
- **400** - if the criteria is missing or not valid
- **404** - if the repository is not found

Return: array of objects representing content unit associations

Sample Request:

```
{
  "criteria": {
    "fields": {
      "unit": [
        "name",
        "version"
      ]
    },
    "type_ids": [
      "rpm"
    ],
    "limit": 1
  }
}
```

Sample 200 Response Body:

```
[
  {
    "updated": "2013-09-04T22:12:05Z",
    "repo_id": "zoo",
    "created": "2013-09-04T22:12:05Z",
    "_ns": "repo_content_units",
    "unit_id": "4a928b95-7c4a-4d23-9df7-ac99978f361e",
    "metadata": {
      "_id": "4a928b95-7c4a-4d23-9df7-ac99978f361e",
      "version": "4.1",
      "name": "bear"
    },
    "unit_type_id": "rpm",
    "owner_type": "importer",
    "_id": {
      "$oid": "522777f5e19a002faebebf79"
    },
    "id": "522777f5e19a002faebebf79",
    "owner_id": "yum_importer"
  }
]
```

Repository Group APIs

Creation, Delete, and Update

Create a Repository Group Creates a new repository group. Group IDs must be unique across all repository groups in the server. The group can be initialized with an array of repositories by passing in their IDs during the create call. Repositories can later be added or removed from the group using the membership calls.

Method: POST
Path: `/pulp/api/v2/repo_groups/`
Permission: create
Request Body Contents:

- **id** (string) - unique identifier for the group
- **display_name** (string) - *(optional)* user-friendly name for the repository group
- **description** (string) - *(optional)* user-friendly text describing the group's purpose
- **repo_ids** (array) - *(optional)* array of repositories to add to the group
- **notes** (object) - *(optional)* key-value pairs to programmatically tag the group

Response Codes:

- **201** - the group was successfully created
- **400** - if one or more of the parameters is invalid
- **404** - if any id in the given repo_ids does not belong to a valid repository
- **409** - if there is already a group with the given ID

Return: database representation of the created group

Sample Request:

```
{
 "id": "demo-group"
}
```

Sample 201 Response Body:

```
{
 "scratchpad": null,
 "display_name": null,
 "description": null,
 "_ns": "repo_groups",
 "notes": {},
 "repo_ids": [],
 "_id": {
   "$oid": "500ed9888a905b04e9000021"
 },
 "id": "demo-group",
 "_href": "/pulp/api/v2/repo_groups/demo-group/"
}
```

Delete a Repository Group Deleting a repository group does not affect the underlying repositories; it simply removes the group and its relationship to all repositories.

Method: DELETE
Path: `/pulp/api/v2/repo_groups/<group_id>/`
Permission: delete
Response Codes:

- **200** - if the repository group was successfully deleted
- **404** - if the specified group does not exist

Return: null

Update a Repository Group Once a repository group is created, its display name, description, and notes can be changed at a later time. The repositories belonging to the group do not fall under this call and are instead modified using the membership calls.

Only changes to notes need to be specified. Unspecified notes in this call remain unaffected. A note is removed by specifying its key with a value of null.

Method: PUT
Path: `/pulp/api/v2/repo_groups/<group_id>/`
Permission: update
Request Body Contents:

- **display_name** (string) - *(optional)* user-friendly name for the repository group
- **description** (string) - *(optional)* user-friendly text describing the group's purpose

- **notes** (object) - *(optional)* changes to key-value pairs to programmatically tag the group

Response Codes:

- **200** - if the update executed immediately and was successful
- **400** - if one of the parameters is invalid
- **404** - if the group does not exist

Return: updated database representation of the group

Sample Request:

```
{
 "display_name": "Demo Group"
}
```

Sample 200 Response Body:

```
{
 "scratchpad": null,
 "display_name": "Demo Group",
 "description": null,
 "_ns": "repo_groups",
 "notes": {},
 "repo_ids": [],
 "_id": {
   "$oid": "500ee4028a905b04e900002e"
 },
 "id": "demo-group",
 "_href": "/pulp/api/v2/repo_groups/demo-group/"
}
```

Retrieval

Retrieve a Single Repository Group Retrieves information on a single repository group.

Method: GET
Path: `/pulp/api/v2/repo_groups/<group_id>/`
Permission: read
Query Parameters: None
Response Codes:

- **200** - if the repository group is found
- **404** - if the group cannot be found

Return: database representation of the matching repository group

Sample 200 Response Body:

```
{
 "scratchpad": null,
 "display_name": "Demo Group",
 "description": null,
 "_ns": "repo_groups",
 "notes": {},
 "repo_ids": [
   "dest-2"
 ],
 "_id": {
   "$oid": "500ee4028a905b04e900002e"
 },
 "id": "demo-group",
 "_href": "/pulp/api/v2/repo_groups/demo-group/"
}
```

Retrieve All Repository Groups Retrieves information on all repository groups in the Pulp server. This call will never return a 404; an empty array is returned in the event there are no groups defined.

This call supports the search query parameters as described in *the search API conventions*.

Method: GET
Path: `/pulp/api/v2/repo_groups/`
Permission: read
Query Parameters:
Response Codes:

- **200** - containing the array of repository groups

Return: array of groups in the same format as retrieving a single group; empty array if there are none defined

Sample 200 Response Body:

```
[
 {
   "scratchpad": null,
   "display_name": null,
   "description": null,
   "_ns": "repo_groups",
   "notes": {},
   "repo_ids": [],
   "_id": {
     "$oid": "500ead8a8a905b04e9000019"
   },
   "id": "g1",
   "_href": "/pulp/api/v2/repo_groups/g1/"
 },
 {
   "scratchpad": null,
   "display_name": "Demo Group",
   "description": null,
   "_ns": "repo_groups",
   "notes": {},
   "repo_ids": [
     "dest-2"
   ],
   "_id": {
     "$oid": "500ee4028a905b04e900002e"
   },
   "id": "demo-group",
   "_href": "/pulp/api/v2/repo_groups/demo-group/"
 }
]
```

Repository Membership

Add Repositories to a Group One or more repositories can be added to an existing group. The repositories to be added are specified using a *search criteria document*. This call is idempotent; if a repository is already a member of the group, no changes are made and no error is raised.

Method: POST
Path: `/pulp/api/v2/repo_groups/<group_id>/actions/associate/`
Permission: execute
Request Body Contents:

- **criteria** (object) - a unit association search criteria document

Response Codes:

- **200** - if the associate was successfully performed
- **400** - if the criteria document is invalid
- **404** - if the group cannot be found

Return: array of repository IDs for all repositories in the group

Sample Request:

```
{
  "criteria": {
    "filters": {
      "id": {"$in": ["dest-1", "dest-2"]}
    }
  }
}
```

Sample 200 Response Body:

```
["dest-2", "dest-1"]
```

Remove Repositories from a Group In the same fashion as adding repositories to a group, repositories to remove are specified through a *search criteria document*. The repositories themselves are unaffected; this call simply removes the association to the given group.

Path: `/pulp/api/v2/repo_groups/<group_id>/actions/unassociate/`
Permission: execute
Request Body Contents:

- **criteria** (object) - a unit association search criteria document

Response Codes:

- **200** - if the removal was successfully performed
- **400** - if the criteria document is invalid
- **404** - if the group cannot be found

Return: array of repository IDs for all repositories in the group

Sample Request:

```
{
 "criteria": {
   "filters": {
     "id": "dest-1"
   }
 }
}
```

Sample 200 Response Body:

```
["dest-2", "dest-1"]
```

Repository Group Distributors

List Repository Group Distributors Retrieves all distributors associated with a given group

Method: GET
Path: `/pulp/api/v2/repo_groups/<group_id>/distributors/`
Permission: read

Response Codes:

- **200** - if the group exists

Return: an array of objects that represent distributors

Sample 200 Response Body: :

```
[
 {
   "scratchpad": null,
   "repo_group_id": "test_group",
   "_ns": "repo_group_distributors",
   "last_publish": "2014-06-12T14:38:05Z",
   "distributor_type_id": "group_test_distributor",
   "_id": {
     "$oid": "5399f38b7bc8f60c78d856bf"
   },
   "config": {
     "config_value1": false,
     "config_value2": true
   },
   "id": "2a146bdf-384b-4951-987e-8d42c7c4317f",
   "_href": "2a146bdf-384b-4951-987e-8d42c7c4317f"
 }
]
```

Add a Distributor to a Repository Group Configures a *distributor* for a previously created Pulp repository group. Each repository group maintains its own configuration for the distributor which is used to dictate how the distributor will function when it publishes content. The possible configuration values are contingent on the type of distributor being added; each distributor type will support a different set of values relevant to how it functions.

Multiple distributors may be associated with a repository group at a given time. There may be more than one distributor with the same type. The only restriction is that the distributor ID must be unique across all distributors for a given repository group.

Adding a distributor performs the following validation steps before confirming the addition:

- If provided, the distributor ID is checked for uniqueness in the context of the repository. If not provided, a unique ID is generated.
- The distributor plugin is contacted and asked to validate the supplied configuration for the distributor. If the distributor indicates its configuration is invalid, the distributor is not added to the repository.
- The distributor's distributor_added method is invoked to allow the distributor to do any initialization required for that repository. If the plugin raises an exception during this call, the distributor is not added to the repository.
- The Pulp database is updated to store the distributor's configuration and the knowledge that the repository is associated with the distributor.

The details of the added distributor are returned from the call.

Method: POST
Path: `/pulp/api/v2/repo_groups/<group_id>/distributors/`
Permission: create

Request Body Contents:

- **distributor_type_id** (string) - indicates the type of distributor being associated with the repository group; there must be a distributor installed in the Pulp server with this ID
- **distributor_config** (object) - configuration the repository will use to drive the behavior of the distributor
- **distributor_id** (string) - *(optional)* if specified, this value will be used to refer to the distributor; if not specified, one will be generated

Response Codes:

- **201** - if the distributor was successfully added
- **400** - if one or more of the required parameters is missing, the distributor type ID refers to a non-existent distributor, or the distributor indicates the supplied configuration is invalid
- **404** - if there is no repository with the given ID

Return: an object that represents the newly added distributor

Sample 201 Response Body: :

```
{
 "scratchpad": null,
 "repo_group_id": "test_group",
 "_ns": "repo_group_distributors",
 "last_publish": null,
 "distributor_type_id": "group_test_distributor",
 "_id": {
   "$oid": "5399fb527bc8f60c77d7c82a"
 },
 "config": {
   "config_value1": false,
   "config_value2": true
 },
 "id": "test_id",
 "_href": "/pulp/api/v2/repo_groups/test_group/distributors/unique_distributor_id/"
}
```

Retrieve a Repository Group Distributor Retrieve a specific distributor that is associated with a group.

Method: GET
Path: `/pulp/api/v2/repo_groups/<group_id>/distributors/<distributor_id>/`
Permission: read

Response Codes:

- **200** - containing an object representing the distributor
- **404** - if either the group_id or the distributor_id do not exist on the server

Return: an object that represents the specified distributor

Sample 200 Response Body: :

```
{
 "scratchpad": null,
 "repo_group_id": "test_group",
 "_ns": "repo_group_distributors",
 "last_publish": null,
 "distributor_type_id": "group_test_distributor",
 "_id": {
   "$oid": "5399fb527bc8f60c77d7c82a"
 },
 "config": {
   "config_value1": false,
   "config_value2": true
 },
 "id": "test_id",
 "_href": "/pulp/api/v2/repo_groups/test_group/distributors/test_id/"
}
```

Update a Repository Group Distributor Update the configuration for a *distributor* that has already been associated with a repository group.

Any distributor configuration value that is not specified remains unchanged.

Method: PUT
Path: `/pulp/api/v2/repo_groups/<group_id>/distributors/<distributor_id>/`
Permission: update

Request Body Contents:

- **distributor_config** (object) - configuration values to change for the distributor

Response Codes:

- **200** - if the configuration was successfully updated
- **404** - if there is no repository group or distributor with the specified IDs

Return: an object that represents the updated distributor

Sample Request: :

```
{
 'distributor_config': {
   "config_value2": false
 }
}
```

Sample 200 Response Body: :

```
{
 "scratchpad": null,
 "repo_group_id": "test_group",
 "_ns": "repo_group_distributors",
 "last_publish": null,
 "distributor_type_id": "group_test_distributor",
 "_id": {
   "$oid": "5399fb527bc8f60c77d7c82a"
 },
 "config": {
   "config_value1": false,
   "config_value2": false
 },
 "id": "test_id",
 "_href": "/pulp/api/v2/repo_groups/test_group/distributors/test_id/"
}
```

Disassociate a Repository Group Distributor Remove a repository group *distributor* from a repository group

Method: DELETE
Path: `/pulp/api/v2/repo_groups/<group_id>/distributors/<distributor_id>/`
Permission: delete

Response Codes:

- **200** - if the distributor was successfully disassociated from the repository group
- **404** - if the given group does not have a distributor with the given distributor id, or if the given group does not exist

Return: null will be returned if the distributor was successfully removed

Publication

Publish a Repository Group Publish content from a repository group using a repository group's *distributor*. This call always executes asynchronously and returns a *call report*.

Method: POST
Path: `/pulp/api/v2/repo_groups/<repo_group_id>/actions/publish/`
Permission: execute
Request Body Contents:

- **id** (string) - the ID of the distributor to use when publishing; this is not the distributor type ID
- **override_config** (object) - *(optional)* distributor configuration values that override the distributor's default configuration for this publish

Response Codes:

- **202** - if the publish is set to be executed
- **404** - if the repository group ID given does not exist

Return: a *Call Report* containing a list of spawned tasks that can be polled for a *Task Report*

Sample Request:

```
{
  "id": "demo_distributor_id",
  "override_config": {}
}
```

Sample result value for the Task Report:

```
{
 "_href": "/pulp/api/v2/tasks/a520a839-96ac-4c63-85e4-19a088c81807/",
 "_id": {
  "$oid": "53e243f9b53073e66875efa3"
 },
 "_ns": "task_status",
 "error": null,
 "exception": null,
 "finish_time": "2014-08-06T15:04:25Z",
 "id": "53e243f97bc8f602856d69b9",
 "progress_report": {},
 "result": null,
 "spawned_tasks": [],
 "start_time": "2014-08-06T15:04:25Z",
 "state": "finished",
 "tags": [
  "pulp:repository_group:demo_repo_group",
  "pulp:repository_group_distributor:demo_distributor_id",
  "pulp:action:publish"
 ],
 "task_id": "a520a839-96ac-4c63-85e4-19a088c81807",
 "task_type": "pulp.server.managers.repo.group.publish.publish",
 "traceback": null
}
```

Tags: The task created will have the following tags: `pulp:action:publish`, `pulp:repository_group:<repo_group_id>`, `pulp:repository_group_distributor:<group_distributor_`

Content Manipulation APIs

Uploading Content

Uploading a unit into a repository is a four step process:

- Create an upload request in Pulp. Pulp will provide an ID that is used for subsequent operations.
- Upload the bits for the new file. For large files, this can be done through multiple calls. This step is entirely optional; it is possible that a unit purely consists of metadata and possibly creating relationships to other units.

- Once the file is uploaded, metadata about the unit itself is provided to Pulp along with a destination repository. The repository's importer is contacted to perform the import which adds the unit to the database and associates it to the repository.
- Once the caller is finished importing the uploaded file, a request is sent to Pulp to delete the uploaded file from Pulp's temporary storage.

Uploaded files are not in the Pulp inventory until the import step. Units must be imported into a repository that has an importer associated with it that is capable of handling the unit type.

Creating an Upload Request Informs Pulp of the desire to upload a new content unit. Pulp will perform any preparation steps in the server and return an upload ID that is used to further work with the upload request.

Method: POST
Path: `/pulp/api/v2/content/uploads/`
Permission: create
Request Body Contents: None
Response Codes:

- **201** - if the request to upload a file is granted
- **500** - if the server cannot initialize the storage location for the file to be uploaded

Return: upload ID to identify this upload request in future calls

Sample 201 Response Body:

```
{
 "_href": "/pulp/api/v2/content/uploads/cfb1fed0-752b-439e-aa68-fba68eababa3/",
 "upload_id': "cfb1fed0-752b-439e-aa68-fba68eababa3"
}
```

Upload Bits Sends a portion of the contents of the file being uploaded to the server. If the entire file cannot be sent in a single call, the caller may divide up the file and provide offset information for Pulp to use when assembling it.

Method: PUT
Path: `/pulp/api/v2/content/uploads/<upload_id>/<offset/`
Permission: update
Request Body Contents: The body of the request is the content to store in the file starting at the offset specified in the URL.
Response Codes:

- **200** - if the content was successfully saved to the file

Return: none

Import into a Repository Provides metadata about the uploaded unit and requests Pulp import it into the inventory and associate it with the given repository. This call is made on the repository itself and the URL reflects this accordingly.

Method: POST
Path: `/pulp/api/v2/repositories/<repo_id>/actions/import_upload/`
Permission: update
Request Body Contents:

- **upload_id** (string) - identifies the upload request being imported
- **unit_type_id** (string) - identifies the type of unit the upload represents
- **unit_key** (object) - unique identifier for the new unit; the contents are contingent on the type of unit being uploaded
- **unit_metadata** (object) - *(optional)* extra metadata describing the unit; the contents will vary based on the importer handling the import
- **override_config** (object) - *(optional)* importer configuration values that override the importer's default configuration

Response Codes:

- **202** - if the request for the import was accepted but postponed until later

Return: a *Call Report* The result field in the call report will be defined by the importer used

Tags: The task created will have the following tags. `"pulp:repository:<repo_id>"`, `"pulp:action:import_upload"`

Sample Request:

```
{
   "override_config": {},
   "unit_type_id": "srpm",
   "upload_id": "768b18c1-fef1-4443-bd5b-a4cc9bda3b03",
   "unit_key": {},
   "unit_metadata": {"checksum_type": null}
}
```

Delete an Upload Request Once the uploaded file has been successfully imported and no further operations are desired, the caller should delete the upload request from the server.

Method: DELETE
Path: `/pulp/api/v2/content/uploads/<upload_id>/`
Permission: delete
Query Parameters: None
Response Codes:

- **200** - if the upload was successfully deleted
- **404** - if the given upload ID is not found

Return: none

List All Upload Requests Returns an array of IDs for all upload requests currently in the server.

Method: GET
Path: `/pulp/api/v2/content/uploads/`
Permission: read
Query Parameters: None
Response Codes:

- **200** - for a successful lookup

Return: array of IDs for all upload requests on the server; empty array if there are none

Sample 200 Response Body:

```
{
 "upload_ids': ["cfb1fed0-752b-439e-aa68-fba68eababa3"]
}
```

Copying Units Between Repositories

Pulp provides the ability to copy units between repositories. Units to copy are specified through a *unit association criteria* applied to a source repository. The `filters` field is used to match units, and the `fields` field is optionally used to limit which fields will be loaded into RAM and handed off to the importer. Limiting which fields are loaded can reduce the consumption of RAM, especially when the units have a lot of metadata. All matching units are imported into the destination repository.

The only restriction is that the destination repository must be configured with an importer that supports the type of units being copied.

Method: POST
Path: `/pulp/api/v2/repositories/<destination_repo_id>/actions/associate/`
Permission: update
Request Body Contents:

- **source_repo_id** (string) - repository from which to copy units
- **criteria** (criteria document) - *(optional)* filters which units to copy from the source repository
- **override_config** (object) - *(optional)* importer configuration values that override the importer's default configuration

Response Codes:

- **202** - if the request was accepted by the server to execute asynchronously

Return: a *Call Report*

Sample Request:

```
{
  'source_repo_id' : 'pulp-f17',
  'criteria': {
    'type_ids' : ['rpm'],
    'filters' : {
      'unit' : {
        '$and': [{'name': {'$regex': 'p.*'}}, {'version': {'$gt': '1.0'}}]
      }
    }
  },
 'override_config': {
   'resolve_dependencies: true,
   'recursive': true
  },
}
```

Sample result value:

```
"result": {
  "units_successful": [
    {
      "unit_key": {
        "name": "whale",
        "checksum": "3b34234afc8b8931d627f8466f0e4fd352145a2512681ec29db0a051a0c9d893",
        "epoch": "0",
        "version": "0.2",
        "release": "1",
        "arch": "noarch",
        "checksumtype": "sha256"
      },
      "type_id": "rpm"
    }
  ]
}
```

Tags: The task created will have the following tags. `"pulp:repository:<source_repo_id>"`, `"pulp:consumer:<destination_repo_id>"`, `"pulp:action:associate"`

Unassociating Content Units from a Repository

Pulp also provides the ability to unassociate units from a repository. Units to unassociate are specified through a *Unit Association Criteria* applied to the repository. All matching units are unassociated from the repository.

The only restriction is that the content units can only be unassociated by the same person that originally associated the units with the repository.

Note that there is a bug related to this call in which criteria with no type_ids field will remove all units in a repository.

Method: POST
Path: `/pulp/api/v2/repositories/<repo_id>/actions/unassociate/`
Permission: update
Request Body Contents:

- **criteria** (criteria document) - filters which units to unassociate from the repository

Response Codes:

- **202** - if the request was accepted by the server to execute asynchronously

Return: a *Call Report*

Tags: The task created will have the following tags. `"pulp:repository:<repo_id>"`, `"pulp:action:unassociate"`

Orphaned Content

Content units (see *content unit*) in Pulp are brought in as part of repository sync and content upload operations. However, because content can be associated with more than one repository, content is not removed when the repositories it is associated with are removed or when the content is disassociated with repositories.

Instead, if content is no longer associated with any repositories, it is considered **orphaned**.

Orphaned content may be viewed and removed from Pulp using the following REST calls.

Content types are defined by type definitions.

Viewing Orphaned Content

View All Orphaned Content Get a summary view of the orphaned units by content type

Method: GET
Path: `/pulp/api/v2/content/orphans/`
Permission: read
Response Codes:

- **200** - even if no orphaned content is found

Return: summary of orphaned packages by content type

Sample 200 Response Body:

```
{
 {'rpm': {'count': 21,
          '_href': '/pulp/api/v2/content/orphans/rpm/'},
 {'drpm': {'count': 0,
           '_href': '/pulp/api/v2/content/orphans/drpm/'},
}
```

View Orphaned Content by Type List all the orphaned content of a particular content type.

Method: GET
Path: `/pulp/api/v2/content/orphans/<content_type_id>/`
Permission: read
Response Codes:

- **200** - even if no orphaned content is found
- **404** - if the content type does not exist

Return: (possibly empty) array of content units

Sample 200 Response Body:

```
{
 [
 {'_content_type_id': 'rpm',
  '_href': '/pulp/api/v2/content/orphans/rpm/d0dc2044-1edc-4298-bf10-a472ea943fe1/',
  '_id': 'd0dc2044-1edc-4298-bf10-a472ea943fe1',
  '_ns': 'units_rpm',
  '_storage_path': '/var/lib/pulp/content/rpm/.//gwt/2.3.0/1.fc16/noarch/c55f30d742a5dade6380a499df9
  'arch': 'noarch',
  'buildhost': 'localhost',
  'checksum': 'c55f30d742a5dade6380a499df9fbf5e6bf35a316acf3774b261592cc8e547d5',
  'checksumtype': 'sha256',
  'description': 'Writing web apps today is a tedious and error-prone process.  Developers can\nspen
  'epoch': '0',
  'filename': 'gwt-2.3.0-1.fc16.noarch.rpm',
  'license': 'ASL 2.0',
  'name': 'gwt',
  'relativepath': 'gwt-2.3.0-1.fc16.noarch.rpm',
  'release': '1.fc16',
  'vendor': '',
  'version': '2.3.0'},
 {'_content_type_id': 'rpm',
  '_href': '/pulp/api/v2/content/orphans/rpm/5b8982b3-1d57-4822-92e5-effa0d4f0a17/',
  '_id': '5b8982b3-1d57-4822-92e5-effa0d4f0a17',
  '_ns': 'units_rpm',
  '_storage_path': '/var/lib/pulp/content/rpm/.//gwt-javadoc/2.3.0/1.fc16/noarch/00da925d1a828f7e398
  'arch': 'noarch',
  'buildhost': 'localhost',
  'checksum': '00da925d1a828f7e3985683ff68043523fe42ec3f1030f449cfddcc5854f6de1',
  'checksumtype': 'sha256',
  'description': 'Javadoc for gwt.',
  'epoch': '0',
  'filename': 'gwt-javadoc-2.3.0-1.fc16.noarch.rpm',
  'license': 'ASL 2.0',
  'name': 'gwt-javadoc',
  'relativepath': 'gwt-javadoc-2.3.0-1.fc16.noarch.rpm',
  'release': '1.fc16',
  'vendor': '',
  'version': '2.3.0'},
 {'_content_type_id': 'rpm',
  '_href': '/pulp/api/v2/content/orphans/rpm/228762de-9762-4384-b41a-4ccc594467f9/',
  '_id': '228762de-9762-4384-b41a-4ccc594467f9',
  '_ns': 'units_rpm',
  '_storage_path': '/var/lib/pulp/content/rpm/.//autotest/0.13.0/6.fc16/noarch/1c0009934068204b3937e4
  'arch': 'noarch',
  'buildhost': 'localhost',
  'checksum': '1c0009934068204b3937e49966b987ae925924b0922656640f39bcd0e85d52cd',
  'checksumtype': 'sha256',
  'description': u"Autotest is a framework for fully automated testing. It is designed primarily\nto
  'epoch': '0',
  'filename': 'autotest-0.13.0-6.fc16.noarch.rpm',
  'license': 'GPLv2 and BSD and LGPLv2.1+',
  'name': 'autotest',
  'relativepath': 'autotest-0.13.0-6.fc16.noarch.rpm',
  'release': '6.fc16',
  'vendor': '',
  'version': '0.13.0'},
 ]
}
```

The individual fields of the content units returned will vary by type. The above sample is provided as a demonstration only and does not necessarily reflect the exact return types of all calls. However all fields beginning with a _ will be available in all content units, regardless of type.

View an Individual Orphaned Content Unit Retrieve an individual orphaned content unit by content type and content id.

Method: GET
Path: `/pulp/api/v2/content/orphans/<content_type_id>/<content_unit_id>/`
Permission: read
Response Codes:

- **200** - if the orphaned content unit is found
- **404** - if the orphaned content unit does not exist

Return: content unit

Removing Orphaned Content Removing orphans may entail deleting contents from disk and, as such, may possibly be long-running process, so all these calls run asynchronously and return a *Call Report*

Remove All Orphaned Content Remove all orphaned content units, regardless of type.

Method: DELETE
Path: `/pulp/api/v2/content/orphans/`
Permission: delete
Response Codes:

- **202** - even if no content is to be deleted

Return: a *Call Report*

Tags: The task created will have the following tag. `"pulp:content_unit:orphans"`

Remove Orphaned Content by Type Remove all the orphaned content of a particular content type.

Method: DELETE
Path: `/pulp/api/v2/content/orphans/<content_type_id>/`
Permission: delete
Response Codes:

- **202** - even if no content is to be deleted

Return: a *Call Report*

Tags: The task created will have the following tag. `"pulp:content_unit:orphans"`

Remove an Individual Orphaned Content Unit Remove and individual orphaned content unit by content type and content id.

Method: DELETE
Path: `/pulp/api/v2/content/orphans/<content_type_id>/<content_unit_id>/`
Permission: delete
Response Codes:

- **202** - if the content unit is to be deleted
- **404** - if the content does not exist

Return: a *Call Report*

Tags: The task created will have the following tag. `"pulp:content_unit:orphans"`

Remove Orphaned Content Units by Type and Id Deprecated since version 2.4: Please use */v2/content/orphans/* instead for deletions.

Individual content units across types may be deleted by this call. The body of the call consists of an array of JSON objects with the fields:

- content_type_id: also known as the content_type_id
- unit_id: also known as the content_unit_id

Method: POST
Path: `/pulp/api/v2/content/actions/delete_orphans/`
Permission: delete
Request Body Contents:

- (array) - jSON object containing the content_type_id and unit_id fields

Response Codes:

- **202** - even if not content is to be deleted

Return: a *Call Report*

Sample Request:

```
{
 [{'content_type_id': 'rpm', 'unit_id': 'd0dc2044-1edc-4298-bf10-a472ea943fe1'},
  {'content_type_id': 'rpm', 'unit_id': '228762de-9762-4384-b41a-4ccc594467f9'}]
}
```

Tags: The task created will have the following tag. `"pulp:content_unit:orphans"`

Retrieval

Retrieve a Single Unit Returns information about a single content unit.

Method: GET
Path: `/pulp/api/v2/content/units/<content_type>/<unit_id>/`
Permission: read
Response Codes:

- **200** - if the unit is found
- **404** - if there is no unit at the given id

Return: the details of the unit

Sample 200 Response Body:

```
{
  "_id": "046ca98d-5977-400d-b4de-a5bb57c8b7e2",
  "_content_type_id": "type-2",
  "_last_updated": "2013-09-05T17:49:41Z",
  "_storage_path": "/var/lib/pulp/content/type-2/A",
  "key-2a": "A",
  "key-2b": "B",
}
```

Note: In the above example, the fields that begin with _ are consistent across content types. All other data, such as the example data "key-2a", is contingent on the type of unit being retrieved.

Search for Units Please see *Search API* for more details on how to perform these searches.

Returns information on content units in the Pulp server that match your search parameters. It is worth noting that this call will never return a 404; an empty array is returned in the case where there are no content units. This is even the case when the content type specified in the URL does not exist.

Method: POST
Path: `/pulp/api/v2/content/units/<content_type>/search/`
Permission: read
Request Body Contents:

- **criteria** (object) - mapping structure as defined in *Search Criteria*
- **include_repos** (boolean) - *(optional)* adds an extra per-unit attribute "repository_memberships" that lists IDs of repositories of which the unit is a member.

Response Codes:

- **200** - containing the array of content units

Return: the same format as retrieving a single content unit, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
  {
    "key-2a": "A",
    "_ns": "units_type-2",
    "_id": "046ca98d-5977-400d-b4de-a5bb57c8b7e2",
    "key-2b": "A",
    "_content_type_id": "type-2",
    "repository_memberships": ["repo1", "repo2"]
  },
  {
    "key-2a": "B",
    "_ns": "units_type-2",
    "_id": "2cc5b44a-c5d7-4751-9505-c54ad4f43497",
    "key-2b": "C",
    "_content_type_id": "type-2",
    "repository_memberships": ["repo1"]
  }
]
```

Returns information on content units in the Pulp server that match your search parameters. It is worth noting that this call will never return a 404; an empty array is returned in the case where there are no content units. This is even the case when the content type specified in the URL does not exist.

This method is slightly more limiting than the POST alternative, because some filter expressions may not be serializable as query parameters.

Method: GET
Path: `/pulp/api/v2/content/units/<content_type>/search/`
Permission: read
Query Parameters: query params should match the attributes of a Criteria object as defined in *Search Criteria*. For example: /v2/content/units/deb/search/?field=id&field=display_name&limit=20'

- **include_repos** (boolean) - *(optional)* adds an extra per-unit attribute "repository_memberships" that lists IDs of repositories of which the unit is a member.

Response Codes:

- **200** - containing the array of content units

Return: the same format as retrieving a single content unit, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
  {
    "key-2a": "A",
    "_ns": "units_type-2",
    "_id": "046ca98d-5977-400d-b4de-a5bb57c8b7e2",
    "key-2b": "A",
    "_content_type_id": "type-2",
    "repository_memberships": ["repo1", "repo2"]
  },
  {
    "key-2a": "B",
    "_ns": "units_type-2",
    "_id": "2cc5b44a-c5d7-4751-9505-c54ad4f43497",
    "key-2b": "C",
    "_content_type_id": "type-2",
    "repository_memberships": ["repo1"]
  }
]
```

Content Sources

Pulp's Content Sources represent external sources of content.

List All Sources Get all content sources.

Method: GET
Path: `/pulp/api/v2/content/sources/`
Permission: read
Query Parameters: None
Response Codes:

- **200** - on success

Return: a list of content source objects

Sample 200 Response Body:

```
[
  {
    "paths": "el7-x86_64/ pulp-el7-x86_64/",
    "name": "Local Content",
    "type": "yum",
    "ssl_validation": "true",
    "expires": "3d",
    "enabled": "1",
    "base_url": "file:///opt/content/disk/",
    "priority": "0",
    "source_id": "disk",
    "max_concurrent": "2",
    "_href": "/pulp/api/v2/content/sources/disk/"
  }
]
```

Get Source By ID Get a content source by ID.

Method: GET
Path: `/pulp/api/v2/content/sources/<source-id>/`
Permission: read
Query Parameters: None
Response Codes:

- **200** - on success

Return: the requested content source object

Sample 200 Response Body:

```
{
"paths": "el7-x86_64/ pulp-el7-x86_64/",
"name": "Local Content",
"type": "yum",
"ssl_validation": "true",
"expires": "3d",
"enabled": "1",
"base_url": "file:///opt/content/disk/",
"priority": "0",
"source_id": "disk",
"max_concurrent": "2",
"_href": "/pulp/api/v2/content/sources/disk/"
}
```

Refresh All Sources Get all content sources.

Method: POST
Path: `/pulp/api/v2/content/sources/action/refresh/`
Permission: update
Query Parameters: None
Response Codes:

- **202** - on success

Return: a spawned task id

Sample 202 Response Body:

```
[
  {
    "spawned_tasks": [
      {
        "_href": "/pulp/api/v2/tasks/1d893293-5849-47d8-830d-f6f888d347e6/",
        "task_id": "1d893293-5849-47d8-830d-f6f888d347e6"
      }
    ],
    "result": null,
    "error": null
  }
]
```

Refresh Single Source Get all content sources.

Method: POST
Path: `/pulp/api/v2/content/sources/<source-id>/action/refresh/`
Permission: update
Query Parameters: None

Response Codes:

- **202** - on success

Return: a spawned task id

Sample 202 Response Body:

```
[
  {
    "spawned_tasks": [
      {
        "_href": "/pulp/api/v2/tasks/7066c9f0-8606-4842-893a-297d435fe11a/",
        "task_id": "7066c9f0-8606-4842-893a-297d435fe11a"
      }
    ],
    "result": null,
    "error": null
  }
]
```

Catalog

The content catalog contains information about content that is provided by Pulp's alternate content sources.

Deleting Entries Delete entries from the catalog by content source by ID.

Method: DELETE
Path: `/pulp/api/v2/content/catalog/<source-id>/`
Permission: delete
Query Parameters: None
Response Codes:

- **200** - even if no entries matched and deleted

Return: a summary of entries deleted

Sample 200 Response Body:

```
{"deleted": 10}
```

Dispatch APIs

Task Management

Pulp can execute almost any call asynchronously and some calls are always executed asynchronously. Pulp provides REST APIs to inspect and manage the tasks executing these calls.

Task Report The task information object is used to report information about any asynchronously executed task.

- **_href** *(string)* - uri path to retrieve this task report object.
- **state** *(string)* - the current state of the task. The possible values include: 'waiting', 'skipped', 'running', 'suspended', 'finished', 'error', 'canceled', and 'timed out'.
- **task_id** *(string)* - the unique id of the task that is executing the asynchronous call
- **task_type** *(string)* - **deprecated** the fully qualified (package/method) type of the task that is executing the asynchronous call. The field is empty for tasks performed by consumer agent.
- **progress_report** *(object)* - arbitrary progress information, usually in the form of an object
- **result** *(any)* - the return value of the call, if any
- **exception** *(null or string)* - **deprecated** the error exception value, if any
- **traceback** *(null or array)* - **deprecated** the resulting traceback if an exception was raised
- **start_time** *(null or string)* - the time the call started executing
- **finish_time** *(null or string)* - the time the call stopped executing
- **tags** *(array)* - arbitrary tags useful for looking up the call report
- **spawned_tasks** *(array)* - List of objects containing the uri and task id for any tasks that were spawned by this task.
- **worker_name** *(string)* - The worker associated with the task. This field is empty if a worker is not yet assigned.
- **queue** *(string)* - The queue associated with the task. This field is empty if a queue is not yet assigned.
- **error** *(null or object)* - Any, errors that occurred that did not cause the overall call to fail. See *Error Details*.

Note: The **exception** and **traceback** fields have been deprecated as of Pulp 2.4. The information about errors that have occurred will be contained in the error block. See *Error Details* for more information.

Example Task Report:

```
{
 "_href": "/pulp/api/v2/tasks/0fe4fcab-a040-11e1-a71c-00508d977dff/",
 "state": "running",
 "worker_name": "reserved_resource_worker-0@your.domain.com",
 "task_id": "0fe4fcab-a040-11e1-a71c-00508d977dff",
 "task_type": "pulp.server.tasks.repository.sync_with_auto_publish",
 "progress_report": {}, # contents depend on the operation
 "result": null,
 "start_time": "2012-05-17T16:48:00Z",
 "finish_time": null,
 "exception": null,
 "traceback": null,
 "tags": [
   "pulp:repository:f16",
   "pulp:action:sync"
 ],
 "spawned_tasks": [{"href": "/pulp/api/v2/tasks/7744e2df-39b9-46f0-bb10-feffa2f7014b/",
                    "task_id": "7744e2df-39b9-46f0-bb10-feffa2f7014b" }],
 "error": null
}
```

Polling Task Progress Poll a task for progress and result information for the asynchronous call it is executing. Polling returns a *Task Report*

Method: GET
Path: `/pulp/api/v2/tasks/<task_id>/`
Permission: read

Response Codes:

- **200** - if the task is found
- **404** - if the task is not found

Return: a *Task Report* representing the task queried

Cancelling a Task Some asynchronous tasks may be cancelled by the user before they complete. A task must be in the *waiting* or *running* states in order to be cancelled.

Note: It is possible for a task to complete or experience an error before the cancellation request is processed, so it is not guaranteed that a task's final state will be 'canceled' as a result of this call. In these instances this method call will still return a response code of 200.

Method: DELETE
Path: `/pulp/api/v2/tasks/<task_id>/`
Permission: delete

Response Codes:

- **200** - if the task cancellation request was successfully received
- **404** - if the task is not found

Return: null

Listing Tasks All currently running and waiting tasks may be listed. This returns an array of *Task Report* instances. the array can be filtered by tags.

Method: GET
Path: `/pulp/api/v2/tasks/`
Permission: read
Query Parameters:

- **tag** (string) - *(optional)* only return tasks tagged with all tag parameters

Response Codes:

- **200** - containing an array of tasks

Return: array of *Task Report*

Searching for Tasks API callers may also search for tasks. This uses a *search criteria document*.

Method: POST
Path: `/pulp/api/v2/tasks/search/`
Permission: read
Request Body Contents: include the key "criteria" whose value is a mapping structure as defined in *Search Criteria*
Response Codes:

- **200** - containing the list of tasks

Return: the same format as retrieving a single task, except the base of the return value is a list. If no results are found, an empty list is returned.

Method: GET
Path: `/pulp/api/v2/tasks/search/`
Permission: read
Query Parameters: query params should match the attributes of a Criteria object as defined in *Search Criteria*. The exception is that field names should be specified in singular form with as many 'field=foo' pairs as needed.

For example:

```
/pulp/api/v2/tasks/search/?field=id&field=task_type&limit=20
```

Response Codes:

- **200** - containing the array of tasks.

Event Listener APIs

Event Listener Creation and Configuration

Learn about *Events*

Create an Event Listener Creates a new listener in the server that will be notified of any events of the configured event types. Each listener must specify a notifier type to handle the event and any configuration necessary for that notifier type. An array of event types is also specified; the newly created listener is only notified when events of the given types are fired.

Method: POST
Path: `/pulp/api/v2/events/`
Permission: create
Request Body Contents:

- **notifier_type_id** (string) - one of the supported notifier type IDs
- **notifier_config** (object) - configuration values the notifier will use when it handles an event
- **event_types** (array) - array of event type IDs that this listener will handle. “*” matches all types

Response Codes:

- **201** - the event listener was successfully created
- **400** - if one of the required parameters is missing or an invalid event type is specified

Return: database representation of the created event listener, including its ID

Sample Request:

```
{
  "notifier_type_id" : "http",
  "notifier_config" : {
    "url" : "http://localhost/api"
  },
  "event_types" : ["repo.sync.finish", "repo.publish.finish"]
}
```

Sample 201 Response Body:

```
{
  "_href": "/pulp/api/v2/events/4ff708048a905b7016000008/",
  "_id": {"$oid": "4ff708048a905b7016000008"},
  "_ns": "event_listeners",
  "event_types": [
    "repo.sync.finish",
    "repo.publish.finish"
  ],
  "id": "4ff708048a905b7016000008",
  "notifier_config": {
    "url": "http://localhost/api"
  },
  "notifier_type_id": "http"
}
```

Retrieve All Event Listeners Returns an array of all event listeners in the server.

Method: GET
Path: `/pulp/api/v2/events/`
Permission: read

Response Codes:

- **200** - array of event listeners, empty array if there are none

Return: database representation of each event listener

Sample 200 Response Body:

```
[
 {
   "_href": "/pulp/api/v2/events/4ff708048a905b7016000008/",
   "_id": {"$oid": "4ff708048a905b7016000008"},
   "_ns": "event_listeners",
   "event_types": [
     "repo.sync.finish",
     "repo.publish.finish"
   ],
   "id": "4ff708048a905b7016000008",
   "notifier_config": {
     "url": "http://localhost/api"
   },
   "notifier_type_id": "http"
 }
]
```

Retrieve a single Event Listener Returns a single event listener from the server.

Method: GET
Path: `/pulp/api/v2/events/<event_listener_id>/`
Permission: read

Response Codes:

- **200** - the event listener detail

- **404** - if the given event listener does not exist

Return: database representation of the event listener

Sample 200 Response Body:

```
{
 "_href": "/pulp/api/v2/events/4ff708048a905b7016000008/",
 "_id": {"$oid": "4ff708048a905b7016000008"},
 "_ns": "event_listeners",
 "event_types": [
   "repo.sync.finish",
   "repo.publish.finish"
 ],
 "id": "4ff708048a905b7016000008",
 "notifier_config": {
   "url": "http://localhost/api"
 },
 "notifier_type_id": "http"
}
```

Delete an Event Listener Deletes an event listener. The event listener is identified by its ID which is found either in the create response or in the data returned by listing all event listeners.

Method: DELETE
Path: `/pulp/api/v2/events/<event_listener_id>/`
Permission: delete

Response Codes:

- **200** - if the event listener was successfully deleted
- **404** - if the given event listener does not exist

Return: none

Update an Event Listener Configuration Changes the configuration for an existing event listener. The notifier type cannot be changed. The event listener being updated is referenced by its ID which is found either in the create response or in the data returned by listing all event listeners.

If the notifier configuration is updated, the following rules apply:

- Configuration keys that are not mentioned in the updated configuration remain unchanged.
- Configuration keys with a value of none are removed entirely from the server-side storage of the notifier's configuration.
- Any configuration keys with non-none values are saved in the configuration, overwriting the previous value for the key if one existed.

Updating the event types is simpler; if present, the provided event types array becomes the new array of event types for the listener. The previous array is overwritten.

Method: PUT
Path: `/pulp/api/v2/events/<event_listener_id>/`
Permission: update
Request Body Contents:

- **notifier_config** (object) - *(optional)* dictates changes to the configuration as described above
- **event_types** (array) - *(optional)* array of new event types for the listener to listen for. "*" matches all types.

Response Codes:

- **200** - if the listener was successfully updated
- **400** - if an invalid event type is specified
- **404** - if the given event listener does not exist

Return: database representation of the updated listener

Sample Request:

```
{
  "event_types" : ["repo.sync.start"]
}
```

Sample 200 Response Body:

```
{
  "_href": "/pulp/api/v2/events/4ff73d598a905b777d000014/",
  "_id": {"$oid": "4ff73d598a905b777d000014"},
  "_ns": "event_listeners",
  "event_types": ["repo.sync.start"],
  "id": "4ff73d598a905b777d000014",
  "notifier_config": {"url": "http://localhost/api"},
  "notifier_type_id": "http"
}
```

User APIs

Create, Update, and Delete

Create a User Create a new user. User logins must be unique across all users.

Method: POST
Path: `/pulp/api/v2/users/`
Permission: create

Request Body Contents:

- **login** (string) - unique login for the user
- **name** (string) - *(optional)* name of the user
- **password** (string) - *(optional)* password of the user used for authentication

Response Codes:

- **201** - if the user was successfully created
- **400** - if one or more of the parameters is invalid
- **409** - if there is already a user with the given login

Return: details of created user

Sample Request:

```
{
 "login": "test-login",
 "password": "test-password",
 "name": "test name"
}
```

Sample 201 Response Body:

```
{
 "name": "test name",
 "roles": [],
 "_ns": "users",
 "login": "test-login",
 "_id": {
   "$oid": "502c83f6e5e7100b0a000035"
 },
 "id": "502c83f6e5e7100b0a000035",
 "_href": "/pulp/api/v2/users/test-login/"
}
```

Update a User The update user call is used to change the details of an existing consumer.

Method: PUT
Path: `/pulp/api/v2/users/<user_login>/`
Permission: update
Request Body Contents: The body of the request is a JSON document with a root element called `delta`. The contents of delta are the values to update. Only changed parameters need be specified. The following keys are allowed in the delta object. Descriptions for each parameter can be found under the create user API:

- **password**
- **name**

- **roles** (array) - *(optional)* array of roles to update the user to. In this case, relevant permissions for the user will be updated as well.

Response Codes:

- **200** - if the update was executed and successful
- **404** - if there is no user with the given login
- **400** - if one or more of the parameters is invalid

Return: database representation of the user including changes made by the update

Sample Request:

```
{
 "delta": {"name": "new name", "password": "new-password"}
}
```

Sample 200 Response Body:

```
{
 "name": "new name",
 "roles": [],
 "_ns": "users",
 "login": "test-login",
 "_id": {
   "$oid": "502c83f6e5e7100b0a000035"
 },
 "id": "502c83f6e5e7100b0a000035"
}
```

Delete a User Deletes a user from the Pulp server. Permissions granted to the user are revoked as well.

Method: DELETE
Path: `/pulp/api/v2/users/<user_login>/`
Permission: delete
Query Parameters:
Response Codes:

- **200** - if the user was successfully deleted
- **404** - if there is no user with the given login

Return: null

Retrieval

Retrieve a Single User Retrieves information on a single Pulp user. The returned data includes general user details.

Method: GET
Path: `/pulp/api/v2/users/<user_login>/`
Permission: read
Query Parameters:

Response Codes:

- **200** - if the user exists
- **404** - if no user exists with the given ID

Return: database representation of the matching user excluding user password

Sample 200 Response Body:

```
{
 "name": "admin",
 "roles": [
   "super-users"
 ],
 "_ns": "users",
 "login": "admin",
 "_id": {
   "$oid": "502c47ace5e7100b0a000008"
 },
 "id": "502c47ace5e7100b0a000008",
 "_href": "/pulp/api/v2/users/admin/"
}
```

Retrieve All Users Returns information on all users in the Pulp server. An empty array is returned in the case where there are no users.

Method: GET
Path: `/pulp/api/v2/users/`
Permission: read
Query Parameters:

Response Codes:

- **200** - containing the array of users

Return: the same format as retrieving a single user, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
 {
   "name": "admin",
   "roles": [
     "super-users"
   ],
   "_ns": "users",
   "login": "admin",
   "_id": {
     "$oid": "502c47ace5e7100b0a000008"
   },
   "id": "502c47ace5e7100b0a000008",
   "_href": "/pulp/api/v2/users/admin/"
 },
 {
   "name": "test name",
   "roles": [],
   "_ns": "users",
   "login": "test-login",
   "_id": {
     "$oid": "502c8c08e5e7100b0a000049"
   },
   "id": "502c8c08e5e7100b0a000049",
   "_href": "/pulp/api/v2/users/test-login/"
 }
]
```

Advanced Search for Users Please see *Search API* for more details on how to perform these searches.

Role APIs

Create, Update, and Delete

This page details role creation, updates and deletion.

If you are updating a role for something besides its name and description, it is likely you want to update the permissions associated with the role instead. Please see the Permissions API for more detail.

Create a Role Create a new role. Role id must be unique across all roles.

Method: POST
Path: `/pulp/api/v2/roles/`
Permission: create
Request Body Contents:

- **role_id** (string) - unique id for the role
- **display_name** (string) - *(optional)* user-friendly name for the role
- **description** (string) - *(optional)* user-friendly text describing the role

Response Codes:

- **201** - if the role was successfully created
- **400** - if one or more of the parameters is invalid
- **409** - if there is already a role with the given id

Return: details of created role

Sample Request:

```
{
 "display_name": "Role Test",
 "description": "Demo Role",
 "role_id": "role-test"
}
```

Sample 201 Response Body:

```
{
 "display_name": "Role Test",
 "description": "Demo Role",
 "_ns": "roles",
 "_href": "/pulp/api/v2/roles/role-test/",
 "_id": {
   "$oid": "502cb2d7e5e710772d000049"
 },
 "id": "role-test",
 "permissions": {}
}
```

Update a Role The update role call is used to change the details of an existing role.

Method: PUT
Path: `/pulp/api/v2/roles/<role_id>/`
Permission: update
Request Body Contents: The body of the request is a JSON document with a root element called `delta`. The contents of delta are the values to update. Only changed parameters need be specified. The following keys are allowed in the delta object.

- **display_name** (string) - *(optional)* user-friendly name for the role
- **description** (string) - *(optional)* user-friendly text describing the role

Response Codes:

- **200** - if the update was executed and successful
- **404** - if there is no role with the given id

Return: database representation of the role including changes made by the update

Sample Request:

```
{
 "delta": {
   "display_name": "New Role Test",
   "description": "New Demo Role"
 }
}
```

Sample 200 Response Body:

```
{
 "display_name": "New Role Test",
 "description": "New Demo Role",
 "_ns": "roles",
 "_href": "/pulp/api/v2/roles/role-test/",
 "_id": {
   "$oid": "502cb2d7e5e710772d000049"
 },
 "id": "role-test"
}
```

Delete a Role Deletes a role from the Pulp server. Users bindings are removed from the role and permissions granted to the users because of the role are revoked as well unless those permissions are granted by other role as well.

Method: DELETE
Path: `/pulp/api/v2/roles/<role_id>/`
Permission: delete
Query Parameters:
Response Codes:

- **200** - if the role was successfully deleted
- **404** - if there is no role with the given id

Return: null

Add a User to a Role Add a user to an existing role. Note that user with given login is NOT created as part of this operation. User with a given login should already exist.

Method: POST
Path: `/pulp/api/v2/roles/<role_id>/users/`
Permission: update
Request Body Contents:

- **login** (string) - login of the user to be added to the role

Response Codes:

- **200** - if the user was successfully added
- **400** - if one or more of the parameters is invalid
- **404** - if there is no role with the given id

Return: null

Sample Request:

```
{
 "login": "test-login"
}
```

Remove a User from a Role Removes a user from an existing role.

Method: DELETE
Path: `/pulp/api/v2/roles/<role_id>/users/<user_login>/`
Permission: delete
Request Body Contents:

Response Codes:

- **200** - if the user was successfully deleted
- **404** - if there is no role with the given id

Return: null

Retrieval

Retrieve a Single Role Retrieves information on a single Role. The returned data includes general role details.

Method: GET
Path: `/pulp/api/v2/roles/<role_id>/`
Permission: read
Query Parameters:

Response Codes:

- **200** - if the role exists
- **404** - if no role exists with the given ID

Return: database representation of the matching role

Sample 200 Response Body:

```
{
 "display_name": "Super Users",
 "description": "Role indicates users with admin privileges",
 "_ns": "roles",
 "_href": "/pulp/api/v2/roles/super-users/",
 "users": [
   "admin"
 ],
 "_id": {
   "$oid": "502ca7afe5e7106ef1000007"
 },
 "id": "super-users",
 "permissions": {
   "/": [
     "CREATE",
     "READ",
     "UPDATE",
     "DELETE",
     "EXECUTE"
   ]
 }
}
```

Retrieve All Roles Returns information on all the roles. An empty array is returned in the case where there are no roles.

Method: GET
Path: `/pulp/api/v2/roles/`
Permission: read
Query Parameters:

Response Codes:

- **200** - containing the array of roles

Return: the same format as retrieving a single role, except the base of the return value is an array of them

Sample 200 Response Body:

```
[
 {
   "display_name": "Super Users",
   "description": "Role indicates users with admin privileges",
   "_ns": "roles",
   "_href": "/pulp/api/v2/roles/super-users/",
   "users": [
     "admin"
   ],
   "_id": {
     "$oid": "502ca7afe5e7106ef1000007"
   },
   "id": "super-users",
   "permissions": {
     "/": [
       "CREATE",
       "READ",
       "UPDATE",
       "DELETE",
       "EXECUTE"
     ]
   }
 },
 {
   "display_name": "test",
   "description": "foo",
   "_ns": "roles",
   "_href": "/pulp/api/v2/roles/test-role1/",
   "users": [
     "test-login"
   ],
   "_id": {
     "$oid": "502caa28e5e71073ae000017"
   },
   "id": "test-role1",
   "permissions": {}
 }
]
```

Permission APIs

Grant/Revoke permissions from User or Role

Grant to user Grants permissions to a user.

Method: POST
Path: `/pulp/api/v2/permissions/actions/grant_to_user/`
Permission: execute
Request Body Contents:

- **login** (string) - login of existing user
- **resource** (string) - resource URI
- **operations** (array) - array of operation strings;valid operations:'CREATE','READ','UPDATE','DELETE','EXECUTE'

Response Codes:

- **200** - if permissions were successfully granted to the user

- **404** - if any of the parameters are invalid

Return: null

Sample Request:

```
{
 "operations": ["CREATE", "READ", "DELETE"],
 "login": "test-login",
 "resource": "/v2/repositories/"
}
```

Revoke from user   Revokes permissions from a user.

Method: POST
Path: `/pulp/api/v2/permissions/actions/revoke_from_user/`
Permission: execute
Request Body Contents:

- **login** (string) - login of existing user
- **resource** (string) - resource URI
- **operations** (array) - array of operation strings;valid operations:'CREATE','READ','UPDATE','DELETE','EXECUTE'

Response Codes:

- **200** - if permissions were successfully revoked from the user
- **404** - if any of the parameters are invalid

Return: null

Sample Request:

```
{
 "operations": ["CREATE", "DELETE"],
 "login": "test-login",
 "resource": "/v2/repositories/"
}
```

Grant to role   Grants permissions to a role. This will add permissions to all users belonging to the role. Note that users added to the role after granting permissions will inherit these permissions from the role as well.

Method: POST
Path: `/pulp/api/v2/permissions/actions/grant_to_role/`

Permission: execute
Request Body Contents:

- **role_id** (string) - id of an existing role
- **resource** (string) - resource URI
- **operations** (array) - array of operation strings;valid operations:'CREATE','READ','UPDATE','DELETE','EXECUTE'

Response Codes:

- **200** - if permissions were successfully granted to the role
- **404** - if any of the parameters are invalid

Return: null

Sample Request:

```
{
 "operations": ["CREATE", "READ", "DELETE"],
 "resource": "/v2/repositories/",
 "role_id": "test-role"
}
```

Revoke from role Revokes permissions from a role. This will revoke permissions from all users belonging to the role unless they are granted by other roles as well.

Method: POST
Path: `/pulp/api/v2/permissions/actions/revoke_from_role/`
Permission: execute
Request Body Contents:

- **role_id** (string) - id of an existing role
- **resource** (string) - resource URI
- **operations** (array) - array of operation strings;valid operations:'CREATE','READ','UPDATE','DELETE','EXECUTE'

Response Codes:

- **200** - if permissions were successfully revoked from the role
- **404** - if any of the parameters are invalid

Return: null

Sample Request:

```
{
 "operations": ["CREATE", "READ", "DELETE"],
 "resource": "/v2/repositories/",
 "role_id": "test-role"
}
```

Retrieval

Retrieve Permissions for particular resource If a resource is specified, permissions for the particular resource are returned. In this case the array will contain a single item.

Method: GET
Path: `/pulp/api/v2/permissions/`
Permission: read
Query Parameters: Resource path URI should be specifield. For example to retrieve permissions for "/v2/actions/login/": /v2/permissions/?resource=%2Fv2%2Factions%2Flogin%2F

Response Codes:

- **200** - containing the array of permissions for specified resource

Return: array of database representation of permissions for specified resource

Sample 200 Response Body:

```
[
```

{

"_id": { "$oid": "546a6ece6754762f1c34b1db"

}, "_ns": "permissions", "id": "546a6ece6754762f1c34b1db", "resource": "/v2/actions/login/", "users": {

"admin": ["READ", "UPDATE"

]

}

}

]

Retrieve Permissions for all resources Returns information on permissions for all resources.

Method: GET
Path: `/pulp/api/v2/permissions/`
Permission: read

Query Parameters:

Response Codes:

- **200** - containing the array of permissions

Return: array of database representation of permissions

Sample 200 Response Body:

```
[
 {
   "_ns": "permissions",
   "_id": {
     "$oid": "5035917fe5e7106f4100000c"
   },
   "resource": "/v2/actions/login/",
   "id": "5035917fe5e7106f4100000c",
   "users": {
     "admin": [
       "READ",
       "UPDATE"
     ]
   }
 },
 {
   "_ns": "permissions",
   "_id": {
     "$oid": "5035917fe5e7106f4100000d"
   },
   "resource": "/v2/actions/logout/",
   "id": "5035917fe5e7106f4100000d",
   "users": {
     "admin": [
       "READ",
       "UPDATE"
     ]
   }
 },
 {
   "_ns": "permissions",
   "_id": {
     "$oid": "5035917fe5e7106f41000010"
   },
   "resource": "/",
   "id": "5035917fe5e7106f41000010",
   "users": {
     "admin": [
       "CREATE",
       "READ",
       "UPDATE",
       "DELETE",
       "EXECUTE"
     ]
```

```
    }
  }
]
```

Status

Getting the Server Status

An unauthenticated resource that shows the current status of the Pulp server. A 200 response shows that the server is up and running. Users of this API may want to examine `pulp_messaging_connection`, `pulp_database_connection` and `known_workers` to get more detailed status information.

Warning: Clustered Pulp installations have additional monitoring concerns. See *Cluster Monitoring* for more information.

Note: This API is meant to provide an "at-a-glance" status to aid debugging of a Pulp deployment, and is not meant to replace monitoring of Pulp components in a production environment.

A healthy Pulp installation will contain exactly one record for "resource_manager" and "scheduler" in the worker list, and one or more "reserved_resource_worker" records. It will also have `messaging_connection` and `database_connection` entries that contain `{connected: True}`. Note that if the scheduler is not running, other workers may be running but not updating their last heartbeat record.

The version of Pulp is also returned via `platform_version` in the `versions` object. This field is calculated from the "pulp-server" python package version. Do not use the deprecated `api_version` record.

Method: GET
Path: `/pulp/api/v2/status/`
Permission: none

Response Codes:

- **200** - pulp server is up and running

Return: jSON document showing current server status

Sample 200 Response Body:

```
{
   "api_version": "2",
   "database_connection": {
       "connected": true
   },
   "known_workers": [
       {
           "last_heartbeat": "2015-01-02T20:39:58Z",
           "name": "scheduler@status-info-net0.default.virt"
```

```
        },
        {
            "last_heartbeat": "2015-01-02T20:40:34Z",
            "name": "reserved_resource_worker-0@status-info-net0.default.virt"
        },
        {
            "last_heartbeat": "2015-01-02T20:40:36Z",
            "name": "resource_manager@status-info-net0.default.virt"
        }
    ],
    "messaging_connection": {
        "connected": true
    },
    "versions": {
        "platform_version": "2.6.0"
    }
}
```

2.6.2 Events

The Pulp server has the ability to fire events as a response to various actions taking place in the server. Event listeners are configured to respond to these events. The event listener's "notifier" is the action that will handle the event, such as sending a message describing the event over a message bus or invoking an HTTP callback. Each event listener is the pairing of a notifier type, its configuration, and one or more event types to listen for.

Notifiers

Email Notifier

The Email notifier is used to send an email to specified recipients every time a particular event type fires. The body of the email will be a JSON serialized version of the event's payload.

Configuration The Email notifier is used by specifying the notifier type as `email`.

The following configuration values are supported when using the Email notifier:

`subject` Required. The text of the email's subject.

`addresses` Required. This is a tuple or list of email addresses that should receive emails.

Body The body of an inbound event notification will be a JSON document containing the following keys:

`event_type` Indicates the type of event that is being sent.

`payload` JSON document describing the event. This will vary based on the type of event.

`call_report` JSON document giving the *Call Report*, if the event was triggered within the context of a task. Otherwise this field will be *null*.

HTTP Notifier

The HTTP notifier is used to trigger a callback to a URL when the event fires. The callback is a POST operation, and the body of the call will be the contents of the event (and thus vary by type).

Note: This was previously known as a "REST API" notifier in development versions of Pulp 2.0. The first build to include the new name was version 2.0.6-0.12.beta

Configuration The HTTP notifier is used by specifying the notifier type as `http`.

The following configuration values are supported when using the HTTP notifier:

`url` Required. Full URL to invoke to send the event information.

`username` If specified, this value will be passed as basic authentication credentials when the HTTP request is made.

`password` If specified, this value will be passed as basic authentication credentials when the HTTP request is made.

Body The body of an inbound event notification will be a JSON document containing the following keys:

`event_type` Indicates the type of event that is being sent.

`payload` JSON document describing the event. This will vary based on the type of event.

`call_report` JSON document giving the *Call Report*, if the event was triggered within the context of a task. Otherwise this field will be *null*.

AMQP Notifier

The AMQP notifier is used to send Pulp events to an AMQP message broker. Messages are sent to a topic exchange. Each message "subject" starts with "pulp.server" and is then followed by the full message type, such as "repo.sync.finish" to yield a "subject" (or "topic") of "pulp.server.repo.sync.finish".

The default exchange will be "amq.topic", which is guaranteed to exist. A new default may be specified in server.conf, and that may be overridden by the configuration described below.

Configuration The AMQP notifier is used by specifying the notifier type as `amqp`.

The following configuration values are supported when using the AMQP notifier:

`exchange` Optional. The name of an AMQP exchange to use. The exchange must be of type "topic".

Body The body of an inbound event notification will be a JSON document containing the following keys:

`event_type` Indicates the type of event that is being sent.

`payload` JSON document describing the event. This will vary based on the type of event.

`call_report` JSON document giving the *Call Report*, if the event was triggered within the context of a task. Otherwise this field will be *null*.

Event Types

Repository Synchronize and Publish Events

The following events are related to repo sync and publish operations.

Repository Sync Started Fired when a repository sync begins.

- **Type ID:** `repo-sync-started`
- **Body:** Contains the ID of the repository being synchronized.

Example:

```
{
  "repo_id": "pulp-f17"
}
```

Repository Sync Finished Fired when a repository sync completes. This event is fired for both successful and failed sync operations and the body will describe the result.

- **Type ID:** `repo-sync-finished`
- **Body:** Contains the results of the sync process. The contents will vary based on the success or failure of the process.

Example Success:

```
{
  "importer_type_id": "yum_importer",
  "importer_id": "yum_importer",
  "exception": null,
  "repo_id": "pulp-f16",
  "removed_count": 0,
  "started": "2012-07-06T15:49:11Z",
  "_ns": "repo_sync_results",
  "completed": "2012-07-06T15:49:14Z",
  "traceback": null,
  "summary": {
    <data is contingent on the importer and removed for space>
  },
  "added_count": 0,
  "error_message": null,
  "updated_count": 0,
  "details": {
    <data is contingent on the importer and removed for space>
  },
  "id": "4ff7413a8a905b777d000072",
  "result": "success"
}
```

Example Failure:

```
{
  "importer_type_id": "yum_importer",
  "importer_id": "yum_importer",
  "exception": null,
  "repo_id": "pulp-f17",
  "removed_count": 0,
  "started": "2012-07-06T12:06:02Z",
  "_ns": "repo_sync_results",
  "completed": "2012-07-06T12:06:02Z",
  "traceback": null,
  "summary": {
    <data is contingent on the importer and removed for space>
  },
  "added_count": 0,
  "error_message": null,
  "updated_count": 0,
  "details": null,
  "id": "4ff70cea8a905b777d00000c",
  "result": "failed"
}
```

Repository Publish Started Fired when a repository publish operation begins. This includes if a repository is configured to automatically publish after a sync.

- **Type ID:** `repo-publish-started`
- **Body:** Contains the ID of the repository and the ID of the distributor performing the publish.

Example:

```
{
  "repo_id": "pulp-f16",
  "distributor_id": "yum_distributor"
}
```

Repository Publish Finished Fired when a repository publish completes. This event is fired for both successful and failed publish operations and the body will describe the result.

- **Type ID:** `repo-publish-finished`
- **Body:** Contains the result of the publish process. The contents will vary based on the success or failure of the process.

Example Success:

```
{
  "exception": null,
  "repo_id": "pulp-f16",
  "started": "2012-07-06T15:53:41Z",
  "_ns": "repo_publish_results",
  "completed": "2012-07-06T15:53:43Z",
  "traceback": null,
  "distributor_type_id": "yum_distributor",
  "summary": {
    <data is contingent on the distributor and removed for space>
  },
  "error_message": null,
  "details": {
    <data is contingent on the distributor and removed for space>
  },
  "distributor_id": "yum_distributor",
  "id": "4ff742478a905b777d00008b",
  "result": "success"
}
```

2.6.3 Pulp Nodes

Pulp *Nodes* management is performed using the platform REST API. This document identifies the specific APIs and defines the data values needed for each call. For more information on *Nodes* concepts, see the Pulp User Guide.

Activation

Activation is stored as a note on the consumer.

Activate

To activate a consumer as a child node, add a special note to the consumer using the *consumer update API*.

Notes:

_child-node The value `true` indicates the consumer is a child node.

_node-update-strategy The value specifies the *node-level* synchronization strategy.

Sample POST body:

```
{
  "delta": {
    "notes": {
      "_child-node": true,
      "_node-update-strategy": "additive"

    }
  }
}
```

Deactivate

To deactivate a child node, remove the special notes from the consumer using the *consumer update API*.

Sample POST body:

```
{
  "delta": {
    "notes": {
      "_child-node": null,
      "_node-update-strategy": null

    }
  }
}
```

Repositories

Enabling

Repositories are enabled for use with child nodes by associating the *Nodes* distributor with the repository using the *distributor association API*. The `distributor_type_id` is `nodes_http_distributor`.

Sample POST body:

```
{
  "distributor_id": "nodes_http_distributor",
  "distributor_type_id": "nodes_http_distributor",
  "distributor_config": {},
  "auto_publish": true
}
```

Disabling

Repositories are disabled for use with child nodes by disassociating the *Nodes* distributor and the repository using the *distributor disassociation API*. The `distributor_id` in the URL is `nodes_http_distributor`.

Publishing

Manually publishing the *Nodes* data necessary for child node synchronization can be done using the *repository publish API*.

Sample POST body:

```
{"override_config": {}, "id": "nodes_http_distributor"}
```

Binding

Bind

Binding a child node to a repository can be done using the *bind API*. In the POST body, the `notify_agent` must be set to `false` because node bindings do not require agent participation. The `binding_config` can be used to specify the *repository-level* synchronization strategy. The default is `additive` if not specified.

Sample POST body:

```
{
  "notify_agent": false,
  "binding_config": {"strategy": "additive"},
  "repo_id": "elmer",
  "distributor_id": "nodes_http_distributor"
}
```

Unbind

Unbinding a child node from a repository can be done using the *unbind API*. The `distributor_id` in the URL is `nodes_http_distributor`.

Synchronization

The synchronization of a child node is seen by the parent server as a content update on a consumer. In this case, the consumer is a child node.

Run

An immediate synchronization of a child node can be initiated using the *content update API*. In the POST body, an array of (1) unit with `type_id` of `node` and `unit_key` of `null` is specified.

Sample POST body:

```
{
  "units": [{"type_id": "node", "unit_key": null}],
  "options": {}
}
```

To skip the repository synchronization phase of the update, specify the `skip_content_update` option with a value of `true`.

Sample POST body:

```
{
  "units": [{"type_id": "node", "unit_key": null}],
  "options": {"skip_content_update": true}
}
```

To synchronize individual repositories, use the `type_id` of `repository` and specify the repository ID using the `repo_id` keyword in the `unit_key`.

Sample POST body:

```
{
  "units": [{"type_id": "repository", "unit_key": {"repo_id": "abc"}}],
  "options": {}
}
```

2.7 Glossary

applicability data Applicability data for a consumer consists of arrays of applicable *content unit* ids, keyed by a content unit type. The definition of applicability itself defers for each content type. For example, in case of an rpm, a content unit is considered applicable to a consumer when an older version of the content unit installed on that consumer can be updated to the given content unit.

binding An association between a *consumer* and a *repository distributor* for the purpose of installing *content units* on the specified consumer.

call report A JSON object describing metadata, progress information, and the final result of any asynchronous task being executed by Pulp.

conduit Object passed to a plugin when it is invoked. The conduit contains methods the plugin should use to access the Pulp Server or Pulp Agent.

consumer A managed system that is the consumer of content. Consumption refers to the installation of software contained within a *repository* and published by an associated *distributor*.

content unit An individual piece of content managed by the Pulp server. A unit does not necessarily correspond to a file. It is possible that a content unit is defined as the aggregation of other content units as a grouping mechanism.

distributor Server-side plugin that takes content from a repository and publishes it for consumption. The process by which a distributor publishes content varies based on the desired approach of the distributor. A repository may have more than one distributor associated with it at a given time.

extension Client-side command line interface plugin that provides additional commands within the command-line clients.

handler Agent plugin that implements content type specific or operating system specific operations on the consumer.

importer Server-side plugin that provides support for synchronizing content from an external source and importing that content into the Pulp server. Importers are added to repositories to define the supported functionality of that repository.

iso8601 interval ISO Date format that is able to specify an optional number of recurrences, an optional start time, and a time interval. There are a number of equivalent formats. Pulp supports: R<optional recurrences>/<optional start time>/P<interval> Examples:

- simple daily interval: P1DT
- 6 hour interval with 3 recurrences: R3/PT6H
- 10 minute interval with start time: 2012-06-22T12:00:00Z/PT10M

Further reading and more examples: http://en.wikipedia.org/wiki/ISO_8601#Time_intervals

platform Short for the "Pulp Platform", which refers to the generic framework functionality provided by Pulp. The platform has no type-specific knowledge; all type-specific functionality is provided through plugins to the platform.

repository A collection of content units. A repository's supported types is dictated by the configured *importer*. A repository may have multiple *distributors* associated which are used to publish its content to multiple destinations, formats, or protocols.

scratchpad Persisted area in which a plugin may store information to be retained across multiple invocations. Each scratchpad is scoped to an individual plugin on a repository.

unit profile An array of *content unit* installed on a *consumer*. The structure and content of each item in the profile varies based on the unit type.

2.8 Troubleshooting

2.8.1 Trailing Slashes

The REST API for Pulp is inconsistent with the use of trailing slashes in REST calls. If you are having trouble, try adding or removing the trailing slash.

CHAPTER 3

Indices and tables

- genindex
- search

Index