pypicloud

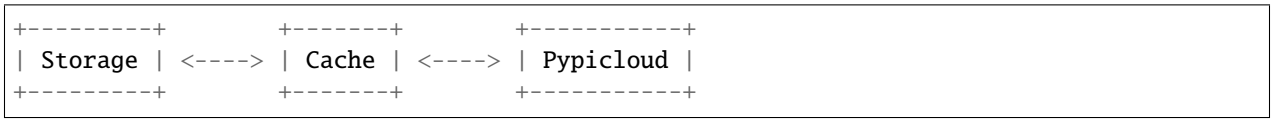
Release 1.3.7

unknown

Jul 14, 2022

CONTENTS

This is an implementation of the PyPI server for hosting your own python packages. It uses a three layer system for storing and serving files:



The **Storage** layer is where the actual package files will be kept and served from. This can be S3, GCS, Azure Blob Storage or a directory on the server running pypicloud.

The **Cache** layer stores information about which packages are in stored in Storage. This can be DynamoDB, Redis, or any SQL database.

The **Pypicloud** webserver itself is stateless, and you can have any number of them as long as they use the same Cache. (Scaling beyond a single cache requires some additional work.)

Pypicloud is designed to be easy to set up for small deploys, and easy to scale up when you need it. Go *get started!*

Code lives here: https://github.com/stevearc/pypicloud

CHAPTER

ONE

USER GUIDE

1.1 Getting Started

There is a docker container if you're into that sort of thing.

1.1.1 Installation

First create and activate a virtualenv to contain the installation:

```
$ virtualenv mypypi
New python executable in mypypi/bin/python
Installing setuptools.............done.
Installing pip...............done.
$ source mypypi/bin/activate
(mypypi)$
```

Now install pypicloud and waitress. To get started, we're using waitress as the WSGI server because it's easy to set up.

```
(mypypi)$ pip install pypicloud[server]
```

1.1.2 Configuration

Generate a server configuration file. Choose `filesystem` when it asks where you want to store your packages.

```
(mypypi)$ ppc-make-config -t server.ini
```

Warning: Note that this configuration should only be used for testing. If you want to set up your server for production, read the section on *deploying*.

1.1.3 Running

You can run the server using pserve

```
(mypypi)$ pserve server.ini
```

The server is running on port 6543. You can view the web interface at http://localhost:6543/

Packages will be stored in a directory named `packages` next to the `server.ini` file. Pypicloud will use a SQLite database in the same location to cache the package index. This is the simplest configuration for pypicloud because it is entirely self-contained on a single server.

1.1.4 Installing Packages

After you have the webserver started, you can install packages using:

```
pip install -i http://localhost:6543/simple/ PACKAGE1 [PACKAGE2 ...]
```

If you want to configure pip to always use pypicloud, you can put your preferences into the `$HOME/.pip/pip.conf` file:

```
[global]
index-url = http://localhost:6543/simple/
```

1.1.5 Uploading Packages

To upload packages, you will need to add your server as an index server inside your `$HOME/.pypirc`:

```
[distutils]
index-servers = pypicloud

[pypicloud]
repository: http://localhost:6543/simple/
username: <<username>>
password: <<password>>
```

Then you can run:

```
python setup.py sdist upload -r pypicloud
```

1.1.6 Searching Packages

After packages have been uploaded, you can search for them via pip:

```
pip search -i http://localhost:6543/pypi QUERY1 [QUERY2 ...]
```

If you want to configure pip to use pypicloud for search, you can update your preferences in the `$HOME/.pip/pip.conf` file:

```
[search]
index = http://localhost:6543/pypi
```

Note that this will ONLY return results from the pypicloud repository. The official PyPi repository will not be queried (regardless of your *fallback* setting)

1.2 Advanced Configurations

Now we're going to try something a bit more complicated. We're going to store the packages in S3 and cache the package index in DynamoDB.

Follow the same *installation instructions* as before.

1.2.1 AWS

If you have not already, create an access key and secret by following the AWS guide

The default configuration should work, but if you get permission errors or 403's, you will need to set a policy on your bucket.

1.2.2 Configuration

This time when you create a config file (`ppc-make-config -t server_s3.ini`), choose `S3` when it asks where you want to store your packages. Then add the following configuration (replacing the <> strings with the values you want)

```
pypi.fallback = redirect

pypi.db = dynamo
db.region_name = <region>

pypi.storage = s3
storage.bucket = <my_bucket>
storage.region_name = <region>
```

1.2.3 Running

Since you're using AWS services now, you need credentials. Put them somewhere that boto can find them. The easiest method is the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables, but you can also put them directly into the `server_s3.ini` file if you wish (see *dynamo* and *s3*)

Now you can run `pserve server_s3.ini`. On the first run it should create the S3 bucket and DynamoDB tables for you (you may need to tweak the provisioned capacity for the DynamoDB tables, depending on your expected load).

If you uploaded any packages to the first server and have them stored locally, you can migrate them to S3 using the `ppc-migrate` tool:

```
ppc-migrate server.ini server_s3.ini
```

1.3 Configuration Options

This is a list of all configuration parameters for pypicloud. In general, any of these can be overridden by environment variables. To override a setting, create an environment variable that is all uppercase, convert `.` to _, and prefix with `PPC_`. For example: `pypi.fallback = none` becomes `PPC_PYPI_FALLBACK=none`.

1.3.1 PyPICloud

pypi.fallback

Argument: {'redirect', 'cache', 'none'}, optional

This option defines what the behavior is when a requested package is not found in the database. (default 'redirect')

`redirect` - Return a 302 to the package at the `fallback_base_url`.

`cache` - Download the package from `fallback_base_url`, store it in the backend, and serve it. User must have `cache_update` permissions.

`none` - Return a 404

See also *pypi.always_show_upstream* below.

See fallback_detail for more detail on exactly how each fallback option will function.

pypi.always_show_upstream

Argument: bool, optional

Default `False`.

This adjusts the fallback behavior when one or more versions of the requested package are stored in pypicloud. If `False`, pypicloud will only show the client the versions that are stored. If `True`, the local versions will be shown with the versions found at the `fallback_base_url`.

pypi.fallback_url

DEPRECATED see `pypi.fallback_base_url`
Argument: string, optional

The index server to handle the behavior defined in `pypi.fallback` (default https://pypi.org/simple)

pypi.fallback_base_url

Argument: string, optional

This takes precendence over `pypi.fallback` by causing redirects to go to: `pypi.fallback_base_url/<simple|pypi>`. (default https://pypi.org)

pypi.use_json_scraper

Argument: bool, optional

There are two methods pypicloud uses to fetch package data from the fallback repo. The JSON scraper, and distlib. Distlib has an issue where it does not return the "Requires-Python" metadata, which can cause installation problems (see issue 219). If you are using a non-standard fallback that *supports* the `/json` endpoints (e.g. https://pypi.org/pypi/pypicloud/json), you may wish to set this to `true` so that you get the proper "Requires-Python" metadata.

Will default to true if `pypi.fallback_base_url` is not set, or is set to `https://pypi.org`.

pypi.disallow_fallback

Argument: list, optional

List of packages that should not be fetch from `pypi.fallback_base_url`. This is useful if private packages have the same name as a package in `pypi.fallback_base_url` and you don't want it to be replaced.

pypi.default_read

Argument: list, optional

List of groups that are allowed to read packages that have no explicit user or group permissions (default ['authenticated'])

pypi.default_write

Argument: list, optional

List of groups that are allowed to write packages that have no explicit user or group permissions (default no groups, only admin users)

pypi.cache_update

Argument: list, optional

Only used when `pypi.fallback = cache`. This is the list of groups that are allowed to trigger the operation that fetches packages from `fallback_base_url`. (default ['authenticated'])

pypi.calculate_package_hashes

Argument: bool, optional

Package SHA256 and MD5 hashes are now calculated by default when a package is uploaded. This option enables or disables the hash calculation (default true)

Scripts to calculate hashes on existing packages exist here: https://github.com/stevearc/pypicloud/tree/master/scripts

pypi.allow_overwrite

Argument: bool, optional

Allow users to upload packages that will overwrite an existing version (default False)

pypi.allow_delete

Argument: bool, optional

Allow users to delete packages (default True)

pypi.realm

Argument: string, optional

The HTTP Basic Auth realm (default 'pypi')

pypi.download_url

Argument: string, optional

Overide for the root server URL displayed in the banner of the homepage.

pypi.stream_files

Argument: bool, optional

Whether or not to stream the raw package data from the storage database, as opposed to returning a redirect link to the storage database. This is useful for taking advantage of the local *pip* cache, which caches based on the URL returned. **Note** that this will in most scenarios make fetching a package slower, since the server will download the full package data before sending it to the client.

pypi.package_max_age

Argument: int, optional

The *max-age* parameter (in seconds) to use in the *Cache-Control* header when downloading packages. If not set, the default will be *0*, which will tell *pip* not to cache any downloaded packages. In order to take advantage of the local *pip* cache, you should set this value to a relatively high number.

1.3.2 Storage

pypi.storage

Argument: string, optional

A dotted path to a subclass of *IStorage*. The default is *FileStorage*. Each storage option may have additional configuration options. Documentation for the built-in storage backends can be found at *Storage Backends*.

1.3.3 Cache

pypi.db

Argument: string, optional

A dotted path to a subclass of *ICache*. The default is *SQLCache*. Each cache option may have additional configuration options. Documentation for the built-in cache backends can be found at *Caching Backends*.

1.3.4 Access Control

pypi.auth

Argument: string, optional

A dotted path to a subclass of *IAccessBackend*. The default is *ConfigAccessBackend*. Each backend option may have additional configuration options. Documentation for the built-in backends can be found at *Access Control*.

1.3.5 Beaker

Beaker is the session manager that handles user auth for the web interface. There are many configuration options, but these are the only ones you need to know about.

session.encrypt_key

Argument: string

Encryption key to use for the AES cipher. Here is a reasonable way to generate one:

```
$ python -c 'import os, base64; print(base64.b64encode(os.urandom(32)))'
```

session.validate_key

Argument: string

Validation key used to sign the AES encrypted data.

session.secure

Argument: bool, optional

If True, only set the session cookie for HTTPS connections (default False). When running a production server, make sure this is always set to `true`.

1.4 Storage Backends

The storage backend is where the actual package files are kept.

1.4.1 Files

This will store your packages in a directory on disk. It's much simpler and faster to set up if you don't need the reliability and scalability of S3.

Set `pypi.storage = file` OR `pypi.storage = pypicloud.storage.FileStorage` OR leave it out completely since this is the default.

storage.dir

Argument: string

The directory where the package files should be stored.

1.4.2 S3

This option will store your packages in S3.

Note: Be sure you have set the correct s3_policy.

Set `pypi.storage = s3` OR `pypi.s3 = pypicloud.storage.S3Storage`

A few key, required options are mentioned below, but pypicloud attempts to support all options that can be passed to resource or to the Config object. In general you can simply prefix the option with `storage.` and pypicloud will pass it on. For example, to set the signature version on the Config object:

```
storage.signature_version = s3v4
```

Note that there is a `s3` option dict as well. Those options should also just be prefixed with `storage.`. For example:

```
storage.use_accelerate_endpoint = true
```

Will pass the Config object the option `Config(s3={'use_accelerate_endpoint': True})`.

Note: If you plan to run pypicloud in multiple regions, read more about syncing pypicloud caches using S3 notifications

storage.bucket

Argument: string

The name of the S3 bucket to store packages in.

storage.region_name

Argument: string, semi-optional

The AWS region your bucket is in. If your bucket does not yet exist, it will be created in this region on startup. If blank, the classic US region will be used.

> **Warning:** If your bucket name has a `.` character in it, or if it is in a newer region (such as `eu-central-1`), you *must* specify the `storage.region_name`!

storage.aws_access_key_id, storage.aws_secret_access_key

Argument: string, optional

Your AWS access key id and secret access key. If they are not specified then pypicloud will attempt to get the values from the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` or any other credentials source.

storage.prefix

Argument: string, optional

If present, all packages will be prefixed with this value when stored in S3. Use this to store your packages in a subdirectory, such as "packages/"

storage.prepend_hash

Argument: bool, optional

Prepend a 4-letter hash to all S3 keys (default True). This helps S3 load balance when traffic scales. See the AWS documentation on the subject.

storage.expire_after

Argument: int, optional

How long (in seconds) the generated S3 urls are valid for (default 86400 (1 day)). In practice, there is no real reason why these generated urls need to expire at all. S3 does it for security, but expiring links isn't part of the python package security model. So in theory you can bump this number up.

storage.redirect_urls

Argument: bool, optional

Leave this alone unless you're having problems using `easy_install`. It defaults to `True` and should not be changed unless you encounter issues.

The long story: redirect_detail

storage.server_side_encryption

Argument: str, optional

Enables AES-256 transparent server side encryption. See the AWS documention. Default is None.

storage.object_acl

Argument: string, optional

Sets uploaded object's "canned" ACL. See the AWS documentation. Default is "private", i.e. only the account owner will get full access. May be useful, if the bucket and pypicloud are hosted in different AWS accounts.

storage.public_url

Argument: bool, optional

If `true`, use public urls (in the form `https://us-east-1.s3.amazonaws.com/<bucket>/<path>`) instead of signed urls. If you configured your bucket to be public and are okay with anyone being able to read your packages, this will give you a speed boost (no expensive hashing operations) and should provide better HTTP caching behavior for the packages. Default is `false`.

1.4.3 CloudFront

This option will store your packages in S3 but use CloudFront to deliver the packages. This is an extension of the S3 storage backend and require the same settings as above, but also the settings listed below.

Set `pypi.storage = cloudfront` OR `pypi.s3 = pypicloud.storage.CloudFrontS3Storage`

storage.cloud_front_domain

Argument: string

The CloudFront domain you have set up. This CloudFront distribution must be set up to use your S3 bucket as the origin.

Example: `https://dabcdefgh12345.cloudfront.net`

storage.cloud_front_key_id

Argument: string, optional

If you want to protect your packages from public access you need to set up the CloudFront distribution to use signed URLs. This setting specifies the key id of the CloudFront key pair that is currently active on your AWS account.

storage.cloud_front_key_file

Argument: string, optional

Only needed when setting up CloudFront with signed URLs. This setting should be set to the full path of the CloudFront private key file.

storage.cloud_front_key_string

Argument: string, optional

The same as `cloud_front_key_file`, but contains the raw private key instead of a path to a file.

1.4.4 Google Cloud Storage

This option will store your packages in GCS.

Set `pypi.storage = gcs` OR `pypi.s3 = pypicloud.storage.GoogleCloudStorage`

Note: The gcs client libraries are not installed by default. To use this backend, you should install pypicloud with `pip install pypicloud[gcs]`.

This backend supports most of the same configuration settings as the S3 backend, and is configured in the same manner as that backend (via config settings of the form `storage.<key> = <value>`).

Settings supported by the S3 backend that are not currently supported by the GCS backend are `server_side_encryption` and `public_url`.

Pypicloud authenticates with GCS using the usual Application Default Credentials strategy, see the documentation for more details. For example you can set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable:

```
GOOGLE_APPLICATION_CREDENTIALS=/path/to/my/keyfile.json pserve pypicloud.ini
```

Pypicloud also exposes a config setting, `storage.gcp_service_account_json_filename`, documented below.

For more information on setting up a service account, see the GCS documentation.

If using the service account provided automatically when running in GCE, GKE, etc, then due to a restriction with the gcloud library, the IAM signing service must be used:

```
storage.gcp_use_iam_signer=true
```

In addition, when using the IAM signing service, the service account used needs to have `iam.serviceAccounts.signBlob` on the storage bucket. This is available as part of `roles/iam.serviceAccountTokenCreator`.

storage.bucket

Argument: string

The name of the GCS bucket to store packages in.

storage.region_name

Argument: string, semi-optional

The GCS region your bucket is in. If your bucket does not yet exist, it will be created in this region on startup. If blank, a default US multi-regional bucket will be created.

storage.gcp_service_account_json_filename

Argument: string, semi-optional

Path to a local file containing a GCP service account JSON key. This argument is required unless the path is provided via the `GOOGLE_APPLICATION_CREDENTIALS` environment variable.

storage.gcp_use_iam_signer

Argument: bool, optional

If true, will use the IAM credentials to sign the generated package links (default `false`).

storage.iam_signer_service_account_email

Argument: string, optional

The email address to use for signing GCS links when `gcp_use_iam_signer = true`. If not provided, will fall back to the email in `gcp_service_account_json_filename`.

See issue 261 for more details

storage.gcp_project_id

Argument: string, optional

ID of the GCP project that contains your storage bucket. This is only used when creating the bucket, and if you would like the bucket to be created in a project other than the project to which your GCP service account belongs.

storage.prefix

Argument: string, optional

If present, all packages will be prefixed with this value when stored in GCS. Use this to store your packages in a subdirectory, such as "packages/"

storage.prepend_hash

Argument: bool, optional

Prepend a 4-letter hash to all GCS keys (default True). This may help GCS load balance when traffic scales, although this is not as well-documented for GCS as for S3.

storage.expire_after

Argument: int, optional

How long (in seconds) the generated GCS urls are valid for (default 86400 (1 day)). In practice, there is no real reason why these generated urls need to expire at all. GCS does it for security, but expiring links isn't part of the python package security model. So in theory you can bump this number up.

storage.redirect_urls

Argument: bool, optional

Leave this alone unless you're having problems using `easy_install`. It defaults to `True` and should not be changed unless you encounter issues.

The long story: redirect_detail

storage.object_acl

Argument: string, optional

Sets uploaded object's "predefined" ACL. See the GCS documentation. Default is "private", i.e. only the account owner will get full access. May be useful, if the bucket and pypicloud are hosted in different GCS accounts.

storage.storage_class

Argument: string, optional

Sets uploaded object's storage class. See the GCS documentation. Defaults to the default storage class of the bucket, if the bucket is preexisting, or "regional" otherwise.

storage.gcp_use_iam_signer

Argument: boolean, optional

Sign blobs using IAM backed signing, rather than using GCP application credentials. The service account used needs to have `iam.serviceAccounts.signBlob` on the storage bucket. This is available as part of `roles/iam.serviceAccountTokenCreator`.

1.4.5 Azure Blob Storage

This option will store your packages in a container in Azure Blob Storage.

Set `pypi.storage = azure-blob` OR `pypi.s3 = pypicloud.storage.AzureBlobStorage`

A few key, required options are mentioned below.

storage.storage_account_name

Argument: string

The name of the Azure Storage Account. If not present, will look for the `AZURE_STORAGE_ACCOUNT` environment variable.

storage.storage_account_key

Argument: string

A valid access key, either key1 or key2. If not present, will look for the `AZURE_STORAGE_KEY` environment variable.

storage.storage_container_name

Argument: string

Name of the container you wish to store packages in.

1.5 Caching Backends

PyPICloud stores the packages in a storage backend (typically S3), but that backend is not necessarily efficient for frequently reading metadata. So instead of hitting S3 every time we need to find a list of package versions, we store all that metadata in a cache. The cache does not have to be backed up because it is only a local cache of data that is permanently stored in the storage backend.

1.5.1 SQLAlchemy

Set `pypi.db = sql` OR `pypi.db = pypicloud.cache.SQLCache` OR leave it out completely since this is the default.

db.url

Argument: string

The database url to use for the caching database. Should be a SQLAlchemy url

- sqlite: `sqlite:///%(here)s/db.sqlite`
- mysql: `mysql://root@127.0.0.1:3306/pypi?charset=utf8mb4`
- postgres: `postgresql://postgres@127.0.0.1:5432/postgres`

Warning: You must specify the `charset=` parameter if you're using MySQL, otherwise it will choke on unicode package names. If you're using 5.5.3 or greater you can specify the `utf8mb4` charset, otherwise use `utf8`.

db.graceful_reload

Argument: bool, optional

When reloading the cache from storage, keep the cache in a usable state while adding and removing the necessary packages. Note that this may take longer because multiple passes will be made to ensure correctness. (default `False`)

db.poolclass

Argument: str, optional

Dotted path to the class to use for connection pooling. Set to `'sqlalchemy.pool.NullPool'` to disable connection pooling. See Connection Pooling for more information.

1.5.2 Redis

Set `pypi.db = redis` OR `pypi.db = pypicloud.cache.RedisCache`

You will need to `pip install redis` before running the server.

db.url

Argument: string

The database url to use for the caching database. The format looks like this: `redis://username:password@localhost:6379/0`

db.graceful_reload

Argument: bool, optional

When reloading the cache from storage, keep the cache in a usable state while adding and removing the necessary packages. Note that this may take longer because multiple passes will be made to ensure correctness. (default `False`)

1.5.3 DynamoDB

Set `pypi.db = dynamo` OR `pypi.db = pypicloud.cache.dynamo.DynamoCache`

Note: Make sure to `pip install pypicloud[dynamo]` before running the server to install the necessary DynamoDB libraries. Also, be sure you have set the correct dynamodb_policy.

Note: Pypicloud will create the DynamoDB tables if none exist. By default the tables will be named `DynamoPackage` and `PackageSummary` (this can be configured with `db.namespace` and `db.tablenames`). You may create and configure these tables yourself as long as they have the same schema.

Warning: When you reload the cache from the admin interface, the default behavior will drop the DynamoDB tables and re-create them. If you have configured the tables to have server-side encryption, or customized the throughput, you may find this undesirable. To avoid this, set `db.graceful_reload = true`

db.region_name

Argument: string

The AWS region to use for the cache tables.

db.aws_access_key_id, db.aws_secret_access_key

Argument: string, optional

Your AWS access key id and secret access key. If they are not specified then pypicloud will attempt to get the values from the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` or any other credentials source.

db.namespace

Argument: string, optional

If specified, all of the created Dynamo tables will have this as a prefix in their name. Useful to avoid name collisions.

db.tablenames

Argument: list<string>, optional

If specified, these will be the names of the two DynamoDB tables. Must be a 2-element whitespace-delimited list. Note that these names will still be prefixed by the `db.namespace`. (default `DynamoPackage PackageSummary`)

db.host

Argument: string, optional

The hostname to connect to. This is normally used to connect to a DynamoDB Local instance.

db.port

Argument: int, optional

The port to connect to when using `db.host` (default 8000)

db.secure

Argument: bool, optional

Force https connection when using `db.host` (default False)

db.graceful_reload

Argument: bool, optional

When reloading the cache from storage, keep the cache in a usable state while adding and removing the necessary packages. Note that this may take longer because multiple passes will be made to ensure correctness. (default `False`)

1.6 Access Control

PyPICloud has a complete access control system that allows you to fine-tune who has access to your packages. There are several choices for where to store your user credentials and access rules.

If you ever need to change your access backend, or you want to back up your current state, check out the *import/export* functionality.

If you want an in-depth look at your options for managing users, see the user_management section.

1.6.1 Users and Groups

The access control uses a combination of users and groups. A group is a list of users. There are also *admin* users, who always have read/write permissions for everything, and can do a few special operations besides. There are two special groups:

- `everyone` - This group refers to any anonymous user making a request
- `authenticated` - This group refers to all logged-in users

You will never need to specify the members of these groups, as membership is automatic.

1.6.2 Config File

The simplest access control available (which is the default) pulls user, group, and package permission information directly from the config file. Note that unlike other options in the config file, you can NOT override these settings with environment variables.

Here is a sample configuration to get you started:

```
# USERS
# user: stevearc, pass: gunface
user.stevearc = $5$rounds=80000$yiWi67QBJLDTvbI/$d6qIG/bIoM3hp0lxH8v/
↪vzxg8Qc4CJbxbxiUH4MlnE7
# user: dsa, pass: paranoia
user.dsa = $5$rounds=80000$U/lot7eW6gFvuvna$KDyrQvi40XXWzMRkBq1Z/0odJEXzqUVNaPIArL/W0s6
# user: donlan, pass: osptony
user.donlan = $5$rounds=80000$Qjz9eRNXrybydMz.$PoD.5vAR9Z2IYlOCPYbza1cKvQ.
↪8kuz1cP0zKl314g0
```

(continues on next page)

(continued from previous page)

```
# GROUPS
group.sharkfest =
    stevearc
    dsa
group.brotatos =
    donlan
    dsa

# PACKAGES
package.django_unchained.user.stevearc = rw
package.django_unchained.group.sharkfest = rw

package.polite_requests.user.dsa = rw
package.polite_requests.group.authenticated = r
package.polite_requests.group.brotatos = rw

package.pyramid_head.group.brotatos = rw
package.pyramid_head.group.everyone = r
```

Here is a table that describes who has what permissions on these packages. Note that if the entry is `none`, that user will not even see the package listed, depending on your `pypi.default_read` and `pypi.default_write` settings.

User	django_unchained	polite_requests	pyramid_head
stevearc	rw (user)	r (authenticated)	r (everyone)
dsa	rw (sharkfest)	rw (user)	rw (brotatos)
donlan	none	rw (brotatos)	rw (brotatos)
everyone	none	none	r (everyone)

Configuration

Set `pypi.auth = config` OR `pypi.auth = pypicloud.access.ConfigAccessBackend` OR leave it out completely since this is the default.

auth.scheme

Argument: str, optional

The default password hash to use. See the passlib docs for choosing a hash. Defaults to `sha512_crypt` on 64 bit systems and `sha256_crypt` on 32 bit systems.

Note this only matters for auth backends that allow dynamic user registration. If you are generating hashes for your config file with `pypicloud-gen-password`, you can configure this with the `-s` argument.

auth.rounds

Argument: int, optional

The number of rounds to use when hashing passwords. See PassLib's docs on choosing rounds values. The default rounds chosen by pypicloud are *significantly lower* than PassLib recommends; see passlib for why.

Note this only matters for auth backends that allow dynamic user registration. If you are generating hashes for your config file with `pypicloud-gen-password`, you can configure this with the `-r` argument.

user.<username>

Argument: string

Defines a single user login. You may specify any number of users in the file. Use `ppc-gen-password` to create the password hashes.

package.<package>.user.<user>

Argument: {`r`, `rw`}

Give read or read/write access on a package to a single user.

package.<package>.group.<group>

Argument: {`r`, `rw`}

Give read or read/write access on a package to a group of users. The group must be defined in a `group.`<group> field.

auth.admins

Argument: list

Whitespace-delimited list of users with admin privileges. Admins have read/write access to all packages, and can perform maintenance tasks.

group.<group>

Argument: list

Whitespace-delimited list of users that belong to this group. Groups can have separately-defined read/write permissions on packages.

1.6.3 SQL Database

You can opt to store all user and group permissions inside a SQL database. The advantages are that you can dynamically change these permissions using the web interface. The disadvantages are that this information is not stored anywhere else, so unlike the *cache database*, it actually needs to be backed up. There is an import/export command *that makes this easy*.

After you set up a new server using this backend, you will need to use the web interface to create the initial admin user.

Configuration

Set `pypi.auth = sql` OR `pypi.auth = pypicloud.access.sql.SQLAccessBackend`

The SQLite engine is constructed by calling engine_from_config with the prefix `auth.db.`, so you can pass in any valid parameters that way.

`auth.db.url`

Argument: string

The database url to use for storing user and group permissions. This may be the same database as `db.url` (if you are also using the SQL caching database).

`auth.db.poolclass`

Argument: str, optional

Dotted path to the class to use for connection pooling. Set to `'sqlalchemy.pool.NullPool'` to disable connection pooling. See Connection Pooling for more information.

`auth.rounds`

Argument: int, optional

The number of rounds to use when hashing passwords. See *auth.rounds*

`auth.signing_key`

Argument: string, optional

Encryption key to use for the token signing HMAC. You may also pass this in with the environment variable `PPC_AUTH_SIGNING_KEY`. Here is a reasonable way to generate a random key:

```
$ python -c 'import os, base64; print(base64.b64encode(os.urandom(32)))'
```

For more about generating and using tokens, see token_registration. Changing this value will retroactively apply to tokens issued in the past.

auth.token_expire

Argument: number, optional

How long (in seconds) the generated registration tokens will be valid for (default one week).

1.6.4 LDAP Authentication

You can opt to authenticate all users through a remote LDAP or compatible server. There is aggressive caching in the LDAP backend in order to keep chatter with your LDAP server at a minimum. If you experience a change in your LDAP layout, group modifications etc, restart your pypicloud process.

Note that you will need to `pip install pypicloud[ldap]` OR `pip install -e .[ldap]` (from source) in order to get the dependencies for the LDAP authentication backend.

At the moment there is no way for pypicloud to discern groups from LDAP, so it only has the built-in `admin`, `authenticated`, and `everyone` as the available groups. All authorization is configured using `pypi.default_read`, `pypi.default_write`, and `pypi.cache_update`. If you need to use groups, you can use the *auth.ldap.fallback* setting below.

Configuration

Set `pypi.auth = ldap` OR `pypi.auth = pypicloud.access.ldap_.LDAPAccessBackend`

auth.ldap.url

Argument: string

The LDAP url to use for remote verification. It should include the protocol and port, as an example: `ldap://10.0.0.1:389`

auth.ldap.service_dn

Argument: string, optional

The FQDN of the LDAP service account used. A service account is required to perform the initial bind with. It only requires read access to your LDAP. If not specified an anonymous bind will be used.

auth.ldap.service_password

Argument: string, optional

The password for the LDAP service account.

auth.ldap.service_username

Argument: string, optional

If provided, this will allow allow you to log in to the pypicloud interface as the provided `service_dn` using this username. This account will have admin privileges.

auth.ldap.user_dn_format

Argument: string, optional

This is used to find a user when they attempt to log in. If the username is part of the DN, then you can provide this templated string where `{username}` will be replaced with the searched username. For example, if your LDAP directory looks like this:

```
dn: CN=bob,OU=users
cn: bob
-
```

Then you could use the setting `auth.ldap.user_dn_format = CN={username},OU=users`.

This option is the preferred method if possible because you can provide the full DN when doing the search, which is more efficient. If your directory is not in this format, you will need to instead use `base_dn` and `user_search_filter`.

auth.ldap.base_dn

Argument: string, optional

The base DN under which all of your user accounts are organized in LDAP. Used in combination with the `user_search_filter` to find users. See also: `user_dn_format`.

`base_dn` and `user_search_filter` should be used if your directory format does not put the username in the DN of the user entry. For example:

```
dn: CN=Robert Paulson,OU=users
cn: Robert Paulson
unixname: bob
-
```

For that directory structure, you would use the following settings:

```
auth.ldap.base_dn = OU=users
auth.ldap.user_search_filter = (unixname={username})
```

auth.ldap.user_search_filter

Argument: string, optional

An LDAP search filter, which when used with the `base_dn` results a user entry. The string `{username}` will be replaced with the username being searched for. For example, `(cn={username})` or `(&(objectClass=person)(name={username}))`

Note that the result of the search must be exactly one entry.

auth.ldap.admin_field

Argument: string, optional

When fetching the user entry, check to see if the `admin_field` attribute contains any of `admin_value`. If so, the user is an admin. This will typically be used with the memberOf overlay.

For example, if this is your LDAP directory:

```
dn: uid=user1,ou=test
cn: user1
objectClass: posixAccount

dn: cn=pypicloud_admin,dc=example,dc=org
objectClass: groupOfUniqueNames
uniqueMember: uid=user1,ou=test
```

You would use these settings:

```
auth.ldap.admin_field = uniqueMemberOf
auth.ldap.admin_value = cn=pypicloud_admin,dc=example,dc=org
```

Since the logic is just checking the value of an attribute, you could also use `admin_value` to specify the usernames of admins:

```
auth.ldap.admin_field = cn
auth.ldap.admin_value =
  user1
  user2
```

auth.ldap.admin_value

Argument: string, optional

See `admin_field`

auth.ldap.admin_group_dn

Argument: string, optional

An alternative to using `admin_field` and `admin_value`. If you don't have access to the `memberOf` overlay, you can provide `admin_group_dn`. When a user is looked up, pypicloud will search this group to see if the user is a member.

Note that to use this setting you must also use `user_dn_format`.

auth.ldap.cache_time

Argument: int, optional

When a user entry is pulled via searching with `base_dn` and `user_search_filter`, pypicloud will cache that entry to decrease load on your LDAP server. This value determines how long (in seconds) to cache the user entries for.

The default behavior is to cache users forever (clearing the cache requires a server restart).

auth.ldap.ignore_cert

Argument: bool, optional

If true then the ldap option to not verify the certificate is used. This is not recommended but useful if the cert name does not match the fqdn. Default is false.

auth.ldap.ignore_referrals

Argument: bool, optional

If true then the ldap option to not follow referrals is used. This is not recommended but useful if the referred servers does not work. Default is false.

auth.ldap.ignore_multiple_results

Argument: bool, optional

If true then the a warning is issued if multiple users are found. This is not recommended but useful if there are more than user matching a given search criteria. Default is false.

auth.ldap.fallback

Argument: string, optional

Since we do not support configuring groups or package permissions via LDAP, this setting allows you to use another system on top of LDAP for that purpose. LDAP will be used for user login and to determine admin status, but this other access backend will be used to determine group membership and package permissions.

Currently the only value supported is `config`, which will use the *Config File* values.

1.6.5 AWS Secrets Manager

This stores all the user data in a single JSON blob using AWS Secrets Manager.

After you set up a new server using this backend, you will need to use the web interface to create the initial admin user.

Configuration

Set `pypi.auth = aws_secrets_manager` OR `pypi.auth = pypicloud.access.aws_secrets_manager.AWSSecretsManagerAccessBackend`

The JSON format should look like this:

```
{
    "users": {
        "user1": "hashed_password1",
        "user2": "hashed_password2",
        "user3": "hashed_password3",
        "user4": "hashed_password4",
        "user5": "hashed_password5",
    },
    "groups": {
        "admins": [
        "user1",
        "user2"
        ],
        "group1": [
        "user3"
        ]
    },
    "admins": [
        "user1"
    ]
    "packages": {
        "mypackage": {
            "groups": {
                "group1": ["read', "write"],
                "group2": ["read"],
                "group3": [],
            },
            "users": {
                "user1": ["read", "write"],
                "user2": ["read"],
                "user3": [],
                "user5": ["read"],
            }
        }
    }
}
```

If the secret is not already created, it will be when you make edits using the web interface.

auth.region_name

Argument: string

The AWS region you're storing your secrets in

auth.secret_id

Argument: string

The unique ID of the secret

auth.aws_access_key_id, auth.aws_secret_access_key

Argument: string, optional

Your AWS access key id and secret access key. If they are not specified then pypicloud will attempt to get the values from the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` or any other credentials source.

auth.aws_session_token

Argument: string, optional

The session key for your AWS account. This is only needed when you are using temporary credentials. See more: http://boto3.readthedocs.io/en/latest/guide/configuration.html#configuration-file

auth.profile_name

Argument: string, optional

The credentials profile to use when reading credentials from the shared credentials file

auth.kms_key_id

Argument: string, optional

The ARN or alias of the AWS KMS customer master key (CMK) to be used to encrypt the secret. See more: https://docs.aws.amazon.com/secretsmanager/latest/apireference/API_CreateSecret.html

1.6.6 Remote Server

This implementation allows you to delegate all access control to another server. If you already have an application with a user database, this allows you to use that data directly.

You will need to `pip install requests` before running the server.

Configuration

Set `pypi.auth = remote` OR `pypi.auth = pypicloud.access.RemoteAccessBackend`

auth.backend_server

Argument: string

The base host url to connect to when fetching access data (e.g. http://myserver.com)

auth.user

Argument: string, optional

If provided, the requests will use HTTP basic auth with this user

auth.password

Argument: string, optional

If `auth.user` is provided, this will be the HTTP basic auth password

auth.uri.verify

Argument: string, optional

The uri to hit when verifying a user's password (default `/verify`).

params: `username`, `password`

returns: `bool`

auth.uri.groups

Argument: string, optional

The uri to hit to retrieve the groups a user is a member of (default `/groups`).

params: `username`

returns: `list`

auth.uri.group_members

Argument: string, optional

The uri to hit to retrieve the list of users in a group (default `/group_members`).

params: `group`

returns: `list`

auth.uri.admin

Argument: string, optional

The uri to hit to determine if a user is an admin (default `/admin`).

params: `username`

returns: `bool`

auth.uri.group_permissions

Argument: string, optional

The uri that returns a mapping of groups to lists of permissions (default `/group_permissions`). The permission lists can contain zero or more of ('read', 'write').

params: `package`

returns: `dict`

auth.uri.user_permissions

Argument: string, optional

The uri that returns a mapping of users to lists of permissions (default `/user_permissions`). The permission lists can contain zero or more of ('read', 'write').

params: `package`

returns: `dict`

auth.uri.user_package_permissions

Argument: string, optional

The uri that returns a list of all packages a user has permissions on (default `/user_package_permissions`). Each element is a dict that contains 'package' (str) and 'permissions' (list).

params: `username`

returns: `list`

auth.uri.group_package_permissions

Argument: string, optional

The uri that returns a list of all packages a group has permissions on (default `/group_package_permissions`). Each element is a dict that contains 'package' (str) and 'permissions' (list).

params: `group`

returns: `list`

`auth.uri.user_data`

Argument: string, optional

The uri that returns a list of users (default `/user_data`). Each user is a dict that contains a `username` (str) and `admin` (bool). If a username is passed to the endpoint, return just a single user dict that also contains `groups` (list).

params: `username`

returns: `list`

1.7 Deploying to Production

This section is geared towards helping you deploy this server properly for production use.

@powellc has put together an Ansible playbook for pypicloud, which can be found here: https://github.com/powellc/ansible-pypicloud

There is a docker container that you can deploy or use as a base image. The following configuration recommendations still apply.

1.7.1 Configuration

Remember when you generated a config file in *getting started*? Well we can do the same thing with a different flag to generate a default production config file.

```
$ ppc-make-config -p prod.ini
```

Warning: You should make sure that `session.secure` is `true`

You may want to tweak *auth.scheme* or *auth.rounds* for more speed or more security. See passlib for more context.

1.7.2 WSGI Server

You probably don't want to use waitress for your production server, though it will work fine for small deploys. I recommend using uWSGI. It's fast and mature.

After creating your production config file, it will have a section for uWSGI. You can run uWSGI with:

```
$ pip install uwsgi pastescript
$ uwsgi --ini-paste-logged prod.ini
```

Now uWSGI is running and listening on port 8080.

Warning: If you are using `pypi.fallback = cache`, make sure your uWSGI settings includes `enable-threads = true`. The package downloader uses threads.

1.7.3 HTTPS and Reverse Proxies

uWSGI has native support for SSL termination, but you may wish to use NGINX or an ELB to do the SSL termination plus load balancing. For this and other reverse proxy behaviors, you will need uWSGI to generate URLs that match what your proxy expects. You can do this with paste middleware. For example, to enforce https:

```
[app:main]
filter-with = proxy-prefix

[filter:proxy-prefix]
use = egg:PasteDeploy#prefix
scheme = https
```

To see more details about how this middleware works and what the other options are, the code can be found on github.

1.8 Upgrading

New versions of PyPICloud may require action in up to two locations:

1. The cache database
2. The access control backend

1.8.1 Cache Database

This storage system is designed to be ephemeral. After an upgrade, all you need to do is rebuild the cache from the storage backend and that will apply any schema changes needed. You can use the "rebuild" button in the admin interface, or you can hit the *REST endpoint* (note that this will not work if you have `db.graceful_reload = true`).

1.8.2 Access Control

If something has changed in the formatting of the access control between versions, there should be a note inside the changelog. If so, you will need to export your current data and import it to the new version.

```
$ ppc-export config.ini -o acl.json.gz
$ pip install --upgrade pypicloud
$ # Make any necessary changes to the config.ini file
$ ppc-import config.ini -i acl.json.gz
```

Note that this system also allows you to migrate your access rules from one backend to another.

```
$ ppc-export old_config.ini | ppc-import new_config.ini
```

1.8.3 Changing Storage

If you would like to change your storage backend, you will need to migrate your existing packages to the new location. Create a config file that uses the new storage backend, and then run:

```
ppc-migrate old_config.ini new_config.ini
```

This will find all packages stored in the old storage backend and upload them to the new storage backend.

1.9 Extending PyPICloud

Certain parts of PyPICloud were created to be pluggable. The storage backend, cache database, and access control backend can all be replaced very easily.

The steps for extending are:

1. Create a new implementation that subclasses the base class (*ICache*, *IStorage*, *IAccessBackend*/*IMutableAccessBackend*)
2. Put that implementation in a package and install that package in the same virtualenv as PyPICloud
3. Pass in a dotted path to that implementation for the approprate config field (e.g. `pypi.db`)

1.10 HTTP API

For all endpoints you may provide HTTP Basic Auth credentials. Here is a quick example that flushes and rebuilds the cache database:

```
curl https://myadmin:myadminpass@pypi.myserver.com/admin/rebuild
```

1.10.1 /simple/ (or /pypi/)

These endpoints are usually only used by `pip`

GET /simple/

Returns a webpage with links to all the pages for each unique package

Example:

```
curl myserver.com/simple/
```

POST /simple/

Upload a package

Parameters:

- `:action` - The only valid value is `'file_upload'`
- `name` - The name of the package being uploaded
- `version` - The version of the package being uploaded
- `content` (file) - The file object that contains the package data

Example:

```
curl -F ':action=file_upload' -F 'name=flywheel' -F 'version=0.1.0' \
-F 'content=@path/to/flywheel-0.1.0.tar.gz' myserver.com/simple/
```

GET /simple/<package>/

Returns a webpage with all links to all versions of this package.

If *fallback* is configured and the server does not contain the package, this will return either a `302` that points towards the fallback server (`redirect`), or a package index pulled from the fallback server (`cache`).

Example:

```
curl myserver.com/simple/flywheel/
```

GET /pypi/<package>/json

Returns information about all versions of the package in JSON format. This is similar to what PyPI does (ex: https://pypi.org/pypi/requests/json) but the information is more limited because pypicloud doesn't store as much package metadata.

Example:

```
curl myserver.com/pypi/flywheel/json/
```

1.10.2 /api/

These endpoints are used by the web interface

GET /api/package/[?verbose=true/false]

If `verbose` is False, return a list of all unique package names. If `verbose` is True, return a list of summarized data for each unique package name.

Parameters:

- `verbose` (bool) - Determines the return format (default False)

Example:

```
curl myserver.com/api/package/
curl myserver.com/api/package/?verbose=true
```

Sample Response

for `verbose=false`:

```
{
    "packages": [
        "flywheel",
        "pypicloud",
        "pyramid"
    ]
}
```

for `verbose=true`:

```
{
    "packages": [
        {
            "name": "flywheel",
            "stable": "0.1.0",
            "unstable": "0.1.0-2-g185e630",
            "last_modified": 1389945600
        },
        {
            "name": "pypicloud",
            "stable": "0.1.0",
            "unstable": "0.1.0-21-g4a739b0",
            "last_modified": 1390207478
        }
    ]
}
```

GET /api/package/<package>/

Get all versions of a package. Also returns if the user has write permissions for that package.

Example:

```
curl myserver.com/api/package/flywheel
```

Sample Response:

```
{
    "packages": [
        {
            "name": "flywheel",
            "last_modified": 1389945600
            "version": "0.1.0"
            "url": "https://pypi.s3.amazonaws.com/34c2/flywheel-0.1.0.tar.gz?Signature=
↪%2FSJidAjDkXbDojzXy8P1rFwe1kw%3D&Expires=1390262542"
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "name": "flywheel",
            "last_modified": 1390207478
            "version": "0.1.0-21-g4a739b0",
            "url": "https://pypi.s3.amazonaws.com/81f2/flywheel-0.1.0-21-g4a739b0.tar.gz?
↪Signature=%2FSJidAjDkXbDojzXy8P1rFwe1kw%3D&Expires=1390262542"
        },
    ],
    "write": true
}
```

POST /api/package/<package>/<filename>

Upload a package to the server. This is just a cleaner endpoint that does the same thing as the POST `/simple/` endpoint.

Parameters:

- `content` (file) - The file object that contains the package data

Example:

```
curl -F 'content=@path/to/flywheel-0.1.0.tar.gz' myserver.com/api/package/flywheel/
↪flywheel-0.1.0.tar.gz
```

DELETE /api/package/<package>/<filename>

Delete a package version from the server

Example:

```
curl -X DELETE myserver.com/api/package/flywheel/flywheel-0.1.0.tar.gz
```

PUT /api/user/<username>/

Register a new user account (if user registration is enabled). After registration the user will have to be confirmed by an admin.

If the server doesn't have any admins then the first user registered becomes the admin.

Parameters:

- `password` - The password for the new user account

Example:

```
curl -X PUT -d 'password=foobar' myserver.com/api/user/LordFoobar
```

POST /api/user/password

Change your password

Parameters:

- `old_password` - Your current password
- `new_password` - The password you are changing to

Example:

```
curl -d 'old_password=foobar&new_password=F0084RR' myserver.com/api/user/password
```

1.10.3 /admin/

These endpoints are used by the admin web interface. Most of them require you to be using a mutable *access backend*.

GET /admin/rebuild/

Flush the cache database and rebuild it by enumerating the storage backend

Example:

```
curl myserver.com/admin/rebuild/
```

GET /admin/acl.json.gz

Download the ACL as a gzipped-json file. This is equivalent to running `ppc-export`.

Example:

```
curl -o acl.json.gz myserver.com/admin/acl.json.gz
```

POST /admin/register/

Set whether registration is enabled or not

Parameters:

- `allow` (bool) - If True, allow new users to register

Example:

```
curl -d 'allow=true' myserver.com/admin/register/
```

GET /admin/pending_users/

Get a list of all users that are registered and need confirmation from an admin

Example:

```
curl myserver.com/admin/pending_users/
```

Sample Response:

```
[
    "LordFoobar",
    "TotallyNotAHacker",
    "Wat"
]
```

GET /admin/token/<username>/

Get a registration token for a username

Example:

```
curl myserver.com/admin/token/LordFoobar/
```

Sample Response:

```
{
    "token":
↪"LordFoobar:1522226377:2c3ad57edc6b73f3b9d16a48893ba4f7da7531a6abcf046c8d9c228ab50e4614
↪",
    "token_url": "http://myserver.com/login#/?
↪token=LordFoobar:1522226377:2c3ad57edc6b73f3b9d16a48893ba4f7da7531a6abcf046c8d9c228ab50e4614
↪"
}
```

GET /admin/user/

Get a list of all users and their admin status

Example:

```
curl myserver.com/admin/user/
```

Sample Response:

```
[
    {
        "username": "LordFoobar",
        "admin": true
    },
    {
        "username": "stevearc",
        "admin": false
```

(continues on next page)

(continued from previous page)

```
    }
]
```

GET /admin/user/<username>/

Get detailed data about a single user

Example:

```
curl myserver.com/admin/user/LordFoobar/
```

Sample Response:

```
{
    "username": "LordFoobar",
    "admin": true,
    "groups": [
        "cool_people",
        "group2"
    ]
}
```

GET /admin/user/<username>/permissions/

Get a list of packages that a user has explicit permissions on

Example:

```
curl myserver.com/admin/user/LordFoobar/permissions/
```

Sample Response:

```
[
    {
        "package": "flywheel",
        "permissions": ["read", "write"]
    },
    {
        "package": "pypicloud",
        "permissions": ["read"]
    }
]
```

DELETE /admin/user/<username>/

Delete a user

Example:

```
curl -X DELETE myserver.com/admin/user/chump/
```

PUT /admin/user/<username>/

Create a new user with a given password

Parameters:

- password (string) - The password for the new user

Example:

```
curl -X PUT -d 'password=abc123' myserver.com/admin/user/LordFoobar/
```

POST /admin/user/<username>/approve/

Mark a pending user as approved

Example:

```
curl -X POST myserver.com/admin/user/LordFoobar/approve/
```

POST /admin/user/<username>/admin/

Grant or revoke admin privileges for a user.

Parameters:

- admin (bool) - If True, promote to admin. If False, demote to regular user.

Example:

```
curl -d 'admin=true' myserver.com/admin/user/LordFoobar/admin/
```

PUT /admin/user/<username>/group/<group>/

Add a user to a group

Example:

```
curl -X PUT myserver.com/admin/user/LordFoobar/group/cool_people/
```

DELETE /admin/user/<username>/group/<group>/

Remove a user from a group

Example:

```
curl -X DELETE myserver.com/admin/user/LordFoobar/group/cool_people/
```

GET /admin/group/

Get a list of all groups

Example:

```
curl myserver.com/admin/group/
```

Sample Response:

```
[
    "cool_people",
    "uncool_people",
    "marginally_cool_people"
]
```

GET /admin/group/<group>/

Get detailed information about a group

Example:

```
curl myserver.com/admin/group/cool_people
```

Sample Response:

```
{
    "members": [
        "LordFoobar",
        "stevearc"
    ],
    "packages": [
        {
            "package": "flywheel",
            "permissions": ["read", "write"]
        },
        {
            "package": "pypicloud",
            "permissions": ["read"]
        }
    ]
}
```

PUT /admin/group/<group>/

Create a new group

Example:

```
curl -X PUT myserver.com/admin/group/cool_people/
```

DELETE /admin/group/<group>/

Delete a group

Example:

```
curl -X DELETE myserver.com/admin/group/uncool_people/
```

GET /admin/package/<package>/

Get the user and group permissions for a package

Example:

```
curl myserver.com/admin/package/flywheel/
```

Sample Response:

```
{
    "user": [
        {
            "username": "LordFoobar",
            "permissions": ["read", "write"]
        },
        {
            "username": "stevearc",
            "permissions": ["read"]
        }
    ],
    "group": [
        {
            "group": "marginally_cool_people",
            "permissions": ["read"]
        },
        {
            "group": "cool_people",
            "permissions": ["read", "write"]
        }
    ]
}
```

PUT /admin/package/<package>/(user|group)/<name>/(read|write)/

Grant a permission to a user or a group on a package

Example:

```
curl -X PUT myserver.com/admin/package/flywheel/user/LordFoobar/read
curl -X PUT myserver.com/admin/package/flywheel/group/cool_people/write
```

DELETE /admin/package/<package>/(user|group)/<name>/(read|write)/

Revoke a permission for a user or a group on a package

Example:

```
curl -X DELETE myserver.com/admin/package/flywheel/user/LordFoobar/read
curl -X DELETE myserver.com/admin/package/flywheel/group/cool_people/write
```

1.11 Developing

To get set up:

```
$ git clone git@github.com:stevearc/pypicloud
$ cd pypicloud
$ virtualenv pypicloud_env
$ . pypicloud_env/bin/activate
$ pip install -r requirements_dev.txt
```

Run `ppc-make-config -d development.ini` to create a developer config file.

Now you can run the server with

```
$ pserve --reload development.ini
```

The unit tests require a redis server to be running on port 6379, MySQL on port 3306, and Postgres on port 5432. If you have docker installed you can use the `run-test-services.sh` script to start all the necessary servers. Run unit tests with:

```
$ python setup.py nosetests
```

or:

```
$ tox
```

1.12 Changelog

If you are upgrading an existing installation, read *the instructions*

1.12.1 1.3.7 - 2022/7/13

- Fix login issues in the web interface (issue 307)

1.12.2 1.3.6 - 2022/7/7

- Allow / to serve packages the same as */simple* (issue 305)

1.12.3 1.3.5 - 2022/6/20

- Handle all errors when fetching from upstream (pull 299)

1.12.4 1.3.4 - 2022/4/30

- Fix files storage backend on Windows (issue 297)

1.12.5 1.3.3 - 2021/11/12

- Add `db.poolclass` to configure SQLAlchemy connection pooling (issue 291)

1.12.6 1.3.2 - 2021/10/16

- Fix exception in JSON endpoint (issue 290)

1.12.7 1.3.1 - 2021/10/12

- Remove trailing slash from JSON scraper (issue 287)

1.12.8 1.3.0 - 2021/10/9

- Allow config options to be overridden by environment variables (issue 270)

1.12.9 1.2.4 - 2021/6/10

- Fix missing permissions for non-admin users (issue 284)

1.12.10 1.2.3 - 2021/6/9

- Add Pyramid>=2.0 to dependencies (issue 283)

1.12.11 1.2.2 - 2021/6/8

- Upgrade to Pyramid 2.0
- Remove the SQL index from package summary field (will take effect when you rebuild your cache, but a rebuild is not required)

1.12.12 1.2.1 - 2021/5/18

- Fix a XSS vulnerability (issue 280)
- Remove storage limit of package summary (pull 276) (will take effect when you rebuild your cache, but a rebuild is not required unless you hit this issue)

1.12.13 1.2.0 - 2021/3/1

- Add more package info to JSON API (pull 269)
- Stop normalizing metadata for Azure (pull 272)
- Provide Azure credentials via environment variable (issue 270)
- Pin the Pyramid version to avoid deprecation (issue 274)
- Dropping support for Python 3.5 and 3.6 due to difficulty with cryptography library

1.12.14 1.1.7 - 2020/11/16

- Fix a datetime crash when reloading the cache (issue 266)
- Fix a logic error with `db.graceful_reload` (pull 267)

1.12.15 1.1.6 - 2020/11/7

- Fix content-type when streaming packages (pull 260)
- JSON scraper doesn't throw exceptions if it receives a HTTP error (issue 264)
- Add config option for GCS IAM signing email (pull 262)

1.12.16 1.1.5 - 2020/9/19

- Add `pypi.allow_delete` to disable deleting packages (issue 259)

1.12.17 1.1.4 - 2020/9/13

- Fix concurrency bugs in GCS backend (issue 258)

1.12.18 1.1.3 - 2020/8/17

- Fix metadata storage issue with some S3-compatible backends (pull 255)
- Command line arg to generate password hash from stdin (pull 253)

1.12.19 1.1.2 - 2020/7/23

- Fix error when package in local storage but not in fallback repository (issue 251)

1.12.20 1.1.1 - 2020/6/14

- Fix an exception when `pypi.use_json_scraper = false` (issue 250)
- Allow passing in `auth.signing_key` as an environment variable (issue 247)
- Add some documentation about the DynamoDB cache (issue 249)

1.12.21 1.1.0 - 2020/5/31

- Drop support for Python 2 (pull 243)
- Add support for package hashes (pull 244)

1.12.22 1.0.16 - 2020/5/20

- Add support for Microsoft Azure Blob storage (pull 241)

1.12.23 1.0.15 - 2020/5/8

- Add `requests` as a dependency (pull 240)

1.12.24 1.0.14 - 2020/5/7

- Fix a bug with reloading Redis cache (pull 230)
- More graceful handling of non-package files in GCS (issue 232)
- Support for `requires_python` metadata (pull 234, issue 219)
- Add `pypi.use_json_scraper` setting for configuring
- Change default value of `storage.redirect_urls` to `True`
- Add *auth.scheme* setting to customize password hashing algorithm (issue 237)
- SIGNIFICANTLY LOWERED default password hashing rounds. Read about why in the docs

1.12.25 1.0.13 - 2020/1/1

- Add option to use IAM signer on GCS (pull 226)

1.12.26 1.0.12 - 2019/12/11

- Change default fallback url from `http://pypi.python.org` to `https://pypi.org` (pull 207)
- Add `pypi.disallow_fallback` option to disable fallback for specific packages (pull 216)
- Fix automatic bucket creation for all S3 regions (pull 225)

1.12.27 1.0.11 - 2019/4/5

- Add ability to stream files through pypicloud (pull 202)
- Support spaces in `auth.ldap.admin_value` values (pull 206)

1.12.28 1.0.10 - 2018/11/26

- Strip non-ASCII characters from summary for S3 backend (pull 197)
- Fix an issue with production log format (issue 198)
- Add `auth.ldap.fallback` to use config file configure groups and permissions with LDAP access backend (issue 199)

1.12.29 1.0.9 - 2018/9/6

- Fix: Exception during LDAP reconnect (pull 192)
- Fix: LDAP on Python 3 could not detect admins (pull 193)
- Feature: New `pypi.auth.admin_group_dn` setting for LDAP (for when memberOf is unavailable)

1.12.30 1.0.8 - 2018/8/27

- Feature: Google Cloud Storage support (pull 189)

1.12.31 1.0.7 - 2018/8/14

- Feature: `/health` endpoint checks health of connection to DB backends (issue 181)
- Feature: Options for LDAP access backend to ignore referrals and ignore multiple user results (pull 184)
- Fix: Exception when `storage.cloud_front_key_file` was set (pull 185)
- Fix: Bad redirect to the fallback url when searching the `/json` endpoint (pull 188)
- Deprecation: `pypi.fallback_url` has been deprecated in favor of `pypi.fallback_base_url` (pull 188)

1.12.32 1.0.6 - 2018/6/11

- Fix: Support `auth.profile_name` passing in a boto profile name (pull 172)
- Fix: Uploading package with empty description using twine crashes DynamoDB backend (issue 174)
- Fix: Config file generation for use with docker container (using %(here)s was not working)
- Use cryptography package instead of horrifyingly old and deprecated pycrypto (issue 179)
- Add `storage.public_url` to S3 backend (issue 173)

1.12.33 1.0.5 - 2018/4/24

- Fix: Download ACL button throws error in Python 3 (issue 166)
- New access backend: AWS Secrets Manager (pull 164)
- Add `storage.storage_class` option for S3 storage (pull 170)
- Add `db.tablenames` option for DynamoDB cache (issue 167)
- Reduce startup race conditions on empty caches when running multiple servers (issue 167)

1.12.34 1.0.4 - 2018/4/1

- Fix: Fix SQL connection issues with uWSGI (issue 160)
- Miscellaneous python 3 fixes

1.12.35 1.0.3 - 2018/3/26

- Fix: uWSGI hangs in python 3 (issue 153)
- Fix: Crash when using `ppc-migrate` to migrate from S3 to S3
- Add warnings and documentation for edge case where S3 bucket has a dot in it (issue 145)
- Admin can create signup tokens (issue 156)

1.12.36 1.0.2 - 2018/1/26

- Fix: Hang when rebuilding Postgres cache (issue 147)
- Fix: Some user deletes fail with Foreign Key errors (issue 150)
- Fix: Incorrect parsing of version for wheels (issue 154)
- Configuration option for number of rounds to use in password hash (issue 115)
- Make request errors visible in the browser (issue 151)
- Add a Create User button to admin page (issue 149)
- SQL access backend defaults to disallowing anonymous users to register

1.12.37 1.0.1 - 2017/12/3

- Support for LDAP anonymous bind (pull 142)
- Fix a crash in Python 3 (issue 141)

1.12.38 1.0.0 - 2017/10/29

- Python3 support thanks to boto3
- Removing stable/unstable version from package summary
- Changing and removing many settings
- Performance tweaks
- `graceful_reload` option for caches, to refresh from the storage backend while remaining operational
- Complete rewrite of LDAP access backend
- Utilities for hooking into S3 create & delete notifications to keep multiple caches in sync

NOTE Because of the boto3 rewrite, many settings have changed. You will need to review the settings for your storage, cache, and access backends to make sure they are correct, as well as rebuilding your cache as per usual.

1.12.39 0.5.6 - 2017/10/29

- Add `storage.object_acl` for S3 (pull 139)

1.12.40 0.5.5 - 2017/9/9

- Allow search endpoint to have a trailing slash (issue 133)

1.12.41 0.5.4 - 2017/8/10

- Allow overriding the displayed download URL in the web interface (pull 125)
- Bump up the DB size of the version field (SQL-only) (pull 128)

1.12.42 0.5.3 - 2017/4/30

- Bug fix: S3 uploads failing from web interface and when fallback=cache (issue 120)

1.12.43 0.5.2 - 2017/4/22

- Bug fix: The `/pypi` path was broken for viewing & uploading packages (issue 119)
- Update docs to recommend `/simple` as the install/upload URL
- Beaker session sets `invalidate_corrupt = true` by default

1.12.44 0.5.1 - 2017/4/17

- Bug fix: Deleting packages while using the Dynamo cache would sometimes remove the wrong package from Dynamo (issue 118)

1.12.45 0.5.0 - 2017/3/29

Upgrade breaks: SQL caching database. You will need to rebuild it.

- Feature: Pip search works now (pull 107)

1.12.46 0.4.6 - 2017/4/17

- Bug fix: Deleting packages while using the Dynamo cache would sometimes remove the wrong package from Dynamo (issue 118)

1.12.47 0.4.5 - 2017/3/25

- Bug fix: Access backend now works with MySQL family (pull 106)
- Bug fix: Return http 409 for duplicate upload to work better with twine (issue 112)
- Bug fix: Show upload button in interface if `default_write = everyone`
- Confirm prompt before deleting a user or group in the admin interface
- Do some basica sanity checking of username/password inputs

1.12.48 0.4.4 - 2016/10/5

- Feature: Add optional AWS S3 Server Side Encryption option (pull 99)

1.12.49 0.4.3 - 2016/8/2

- Bug fix: Rebuilding cache always ends up with correct name/version (pull 93)
- Feature: /health endpoint (nothing fancy, just returns 200) (issue 95)

1.12.50 0.4.2 - 2016/6/16

- Bug fix: Show platform-specific versions of wheels (issue 91)

1.12.51 0.4.1 - 2016/6/8

- Bug fix: LDAP auth disallows empty passwords for anonymous binding (pull 92)
- Config generator sets `pypi.default_read = authenticated` for prod mode

1.12.52 0.4.0 - 2016/5/16

Backwards incompatibility: This version was released to handle a change in the way pip 8.1.2 handles package names. If you are upgrading from a previous version, there are detailed instructions for how to upgrade safely.

1.12.53 0.3.13 - 2016/6/8

- Bug fix: LDAP auth disallows empty passwords for anonymous binding (pull 92)

1.12.54 0.3.12 - 2016/5/5

- Feature: Setting `auth.ldap.service_account` for LDAP auth (pull 84)

1.12.55 0.3.11 - 2016/4/28

- Bug fix: Missing newline in config template (pull 77)
- Feature: `pypi.always_show_upstream` for tweaking fallback behavior (issue 82)

1.12.56 0.3.10 - 2016/3/21

- Feature: S3 backend setting `storage.redirect_urls`

1.12.57 0.3.9 - 2016/3/13

- Bug fix: SQL cache works with MySQL (issue 74)
- Feature: S3 backend can use S3-compatible APIs (pull 72)

1.12.58 0.3.8 - 2016/3/10

- Feature: Cloudfront storage (pull 71)
- Bug fix: Rebuilding cache from storage won't crash on odd file names (pull 70)

1.12.59 0.3.7 - 2016/1/12

- Feature: `/packages` endpoint to list all files for all packages (pull 64)

1.12.60 0.3.6 - 2015/12/3

- Bug fix: Settings parsed incorrectly for LDAP auth (issue 62)

1.12.61 0.3.5 - 2015/11/15

- Bug fix: Mirror mode: only one package per version is displayed (issue 61)

1.12.62 0.3.4 - 2015/8/30

- Add docker-specific option for config creation
- Move docker config files to a separate repository

1.12.63 0.3.3 - 2015/7/17

- Feature: LDAP Support (pull 55)
- Bug fix: Incorrect package name/version when uploading from web (issue 56)

1.12.64 0.3.2 - 2015/7/7

- Bug fix: Restore direct links to S3 to fix easy_install (issue 54)

1.12.65 0.3.1 - 2015/6/18

- Bug fix: `pypi.allow_overwrite` causes crash in sql cache (issue 52)

1.12.66 0.3.0 - 2015/6/16

- Fully defines the behavior of every possible type of pip request. See Fallbacks for more detail.
- Don't bother caching generated S3 urls.

1.12.67 0.2.13 - 2015/5/27

- Bug fix: Crash when mirror mode serves private packages

1.12.68 0.2.12 - 2015/5/14

- Bug fix: Mirror mode works properly with S3 storage backend

1.12.69 0.2.11 - 2015/5/11

- Bug fix: Cache mode will correctly download packages with legacy versioning (pull 45)
- Bug fix: Fix the fetch_requirements endpoint (commit 6b2e2db)
- Bug fix: Incorrect expire time comparison with IAM roles (pull 47)
- Feature: 'mirror' mode. Caches packages, but lists all available upstream versions.

1.12.70 0.2.10 - 2015/2/27

- Bug fix: S3 download links expire incorrectly with IAM roles (issue 38)
- Bug fix: `fallback = cache` crashes with distlib 0.2.0 (issue 41)

1.12.71 0.2.9 - 2014/12/14

- Bug fix: Connection problems with new S3 regions (issue 39)
- Usability: Warn users trying to log in over http when `session.secure = true` (issue 40)

1.12.72 0.2.8 - 2014/11/11

- Bug fix: Crash when migrating packages from file storage to S3 storage (pull 35)

1.12.73 0.2.7 - 2014/10/2

- Bug fix: First download of package using S3 backend and `pypi.fallback = cache` returns 404 (issue 31)

1.12.74 0.2.6 - 2014/8/3

- Bug fix: Rebuilding SQL cache sometimes crashes (issue 29)

1.12.75 0.2.5 - 2014/6/9

- Bug fix: Rebuilding SQL cache sometimes deadlocks (pull 27)

1.12.76 0.2.4 - 2014/4/29

- Bug fix: `ppc-migrate` between two S3 backends (pull 22)

1.12.77 0.2.3 - 2014/3/13

- Bug fix: Caching works with S3 backend (commit 4dc593a)

1.12.78 0.2.2 - 2014/3/13

- Bug fix: Security bug in user auth (commit 001e8a5)
- Bug fix: Package caching from pypi was slightly broken (commit 065f6c5)
- Bug fix: `ppc-migrate` works when migrating to the same storage type (commit 45abcde)

1.12.79 0.2.1 - 2014/3/12

- Bug fix: Pre-existing S3 download links were broken by 0.2.0 (commit 52e3e6a)

1.12.80 0.2.0 - 2014/3/12

Upgrade breaks: caching database

- Bug fix: Timestamp display on web interface (pull 18)
- Bug fix: User registration stores password as plaintext (commit 21ebe44)
- Feature: `ppc-migrate`, command to move packages between storage backends (commit 399a990)
- Feature: Adding support for more than one package with the same version. Now you can upload wheels! (commit 2f24877)
- Feature: Allow transparently downloading and caching packages from pypi (commit e4dabc7)
- Feature: Export/Import access-control data via `ppc-export` and `ppc-import` (commit dbd2a16)
- Feature: Can set default read/write permissions for packages (commit c9aa57b)
- Feature: New cache backend: DynamoDB (commit d9d3092)

- Hosting all js & css ourselves (no more CDN links) (commit 20e345c)
- Obligatory miscellaneous refactoring

1.12.81 0.1.0 - 2014/1/20

- First public release

CHAPTER

TWO

API REFERENCE

2.1 pypicloud package

2.1.1 Subpackages

pypicloud.access package

Submodules

pypicloud.access.aws_secrets_manager module

Backend that defers to another server for access control

class pypicloud.access.aws_secrets_manager.**AWSSecretsManagerAccessBackend**(*request=None*, *secret_id=None*, *kms_key_id=None*, *client=None*, ***kwargs*)

Bases: *IMutableJsonAccessBackend*

This backend allows you to store all user and package permissions in AWS Secret Manager

check_health()

Check the health of the access backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

classmethod configure(*settings:* EnvironSettings)

Configure the access backend with app settings

pypicloud.access.base module

The access backend object base class

class pypicloud.access.base.**IAccessBackend**(*request=None*, *default_read=None*, *default_write=None*, *disallow_fallback=()*, *cache_update=None*, *pwd_context=None*, *token_expiration=604800*, *signing_key=None*)

Bases: object

Base class for retrieving user and package permission data

```
ROOT_ACL = [('Allow', 'system.Authenticated', 'login'), ('Allow', 'admin',
<pyramid.security.AllPermissionsList object>), ('Deny', 'system.Everyone',
<pyramid.security.AllPermissionsList object>)]
```

allow_register() → bool

Check if the backend allows registration

This should only be overridden by mutable backends

Returns

allow
[bool]

allow_register_token() → bool

Check if the backend allows registration via tokens

This should only be overridden by mutable backends

Returns

allow
[bool]

allowed_permissions(*package: str*) → Dict[str, Tuple[str, ...]]

Get all allowed permissions for all principals on a package

Returns

perms
[dict] Mapping of principal to tuple of permissions

can_update_cache() → bool

Return True if the user has permissions to update the pypi cache

check_health() → Tuple[bool, str]

Check the health of the access backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

classmethod configure(*settings: EnvironSettings*) → Dict[str, Any]

Configure the access backend with app settings

dump() → Dict[str, Any]

Dump all of the access control data to a universal format

Returns

data
[dict]

`get_acl`(*package: str*) → List[Tuple[str, str, str]]

Construct an ACL for a package

`group_members`(*group: str*) → List[str]

Get a list of users that belong to a group

Parameters

group
[str]

Returns

users
[list] List of user names

`group_package_permissions`(*group: str*) → List[Dict[str, List[str]]]

Get a list of all packages that a group has permissions on

Parameters

group
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

`group_permissions`(*package: str*) → Dict[str, List[str]]

Get a mapping of all groups to their permissions on a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] mapping of group name to a list of permissions (which can contain 'read' and/or 'write')

`groups`(*username: Optional[str] = None*) → List[str]

Get a list of all groups

If a username is specified, get all groups that the user belongs to

Parameters

username
[str, optional]

Returns

groups
[list] List of group names

`has_permission`(*package: str*, *perm: str*) → bool

Check if this user has a permission for a package

`in_any_group`(*username: str*, *groups: List[str]*) → bool

Find out if a user is in any of a set of groups

Parameters

username
[str] Name of user. May be None for the anonymous user.

groups
[list] list of group names. Supports 'everyone', 'authenticated', and 'admin'.

Returns

member
[bool]

`in_group`(*username: Optional[str]*, *group: str*) → bool

Find out if a user is in a group

Parameters

username
[str, None] Name of user. May be None for the anonymous user.

group
[str] Name of the group. Supports 'everyone', 'authenticated', and 'admin'.

Returns

member
[bool]

`is_admin`(*username: str*) → bool

Check if the user is an admin

Parameters

username
[str]

Returns

is_admin
[bool]

`load`(*data*)

Idempotently load universal access control data.

By default, this does nothing on immutable backends. Backends may override this method to provide an implementation.

This method works by default on mutable backends with no override necessary.

`mutable = False`

`need_admin`() → bool

Find out if there are any admin users

This should only be overridden by mutable backends

Returns

need_admin
[bool] True if no admin user exists and the backend is mutable, False otherwise

classmethod postfork(***kwargs*)

This method will be called after uWSGI forks

user_data(*username=None*)

Get a list of all users or data for a single user

For Mutable backends, this MUST exclude all pending users

Returns

users
[list] Each user is a dict with a 'username' str, and 'admin' bool

user
[dict] If a username is passed in, instead return one user with the fields above plus a 'groups' list.

user_package_permissions(*username: str*) → List[Dict[str, List[str]]]

Get a list of all packages that a user has permissions on

Parameters

username
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

user_permissions(*package: str*) → Dict[str, List[str]]

Get a mapping of all users to their permissions for a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] Mapping of username to a list of permissions (which can contain 'read' and/or 'write')

user_principals(*username: Optional[str]*) → List[str]

Get a list of principals for a user

Parameters

username
[str]

Returns

principals
[list]

verify_user(*username: str*, *password: str*) → bool

Check the login credentials of a user

For Mutable backends, pending users should fail to verify

Parameters

username
[str]

password
[str]

Returns

valid
[bool] True if user credentials are valid, false otherwise

class pypicloud.access.base.**IMutableAccessBackend**(*request=None*, *default_read=None*, *default_write=None*, *disallow_fallback=()*, *cache_update=None*, *pwd_context=None*, *token_expiration=604800*, *signing_key=None*)

Bases: *IAccessBackend*

Base class for access backends that can change user/group permissions

allow_register()

Check if the backend allows registration

This should only be overridden by mutable backends

Returns

allow
[bool]

allow_register_token()

Check if the backend allows registration via tokens

This should only be overridden by mutable backends

Returns

allow
[bool]

approve_user(*username: str*) → None

Mark a user as approved by the admin

Parameters

username
[str]

create_group(*group: str*) → None

Create a new group

Parameters

group
[str]

delete_group(*group: str*) → None

Delete a group

Parameters

group
[str]

delete_user(*username: str*) → None

Delete a user

Parameters

username
[str]

dump()

Dump all of the access control data to a universal format

Returns

data
[dict]

edit_group_permission(*package_name: str*, *group: str*, *perm: Set[str]*, *add: bool*) → None

Grant or revoke a permission for a group on a package

Parameters

package_name
[str]

group
[str]

perm
[{'read', 'write'}]

add
[bool] If True, grant permissions. If False, revoke.

edit_user_group(*username: str*, *group: str*, *add: bool*) → None

Add or remove a user to/from a group

Parameters

username
[str]

group
[str]

add
[bool] If True, add to group. If False, remove.

edit_user_password(*username: str*, *password: str*) → None

Change a user's password

Parameters

username
[str]

password
[str]

edit_user_permission(*package_name: str*, *username: str*, *perm: Set[str]*, *add: bool*) → None

Grant or revoke a permission for a user on a package

Parameters

package_name
[str]

username
[str]

perm
[{'read', 'write'}]

add
[bool] If True, grant permissions. If False, revoke.

`get_signup_token`(*username: str*) → str

Create a signup token

Parameters

username
[str] The username to be created when this token is consumed

Returns

token
[str]

`load`(*data*)

Idempotently load universal access control data.

By default, this does nothing on immutable backends. Backends may override this method to provide an implementation.

This method works by default on mutable backends with no override necessary.

`mutable = True`

`need_admin`() → bool

Find out if there are any admin users

This should only be overridden by mutable backends

Returns

need_admin
[bool] True if no admin user exists and the backend is mutable, False otherwise

`pending_users`() → List[str]

Retrieve a list of all users pending admin approval

Returns

users
[list] List of usernames

`register`(*username: str*, *password: str*) → None

Register a new user

The new user should be marked as pending admin approval

Parameters

username
[str]

password
[str] This should be the plaintext password

set_allow_register(*allow: bool*) → None

Allow or disallow user registration

Parameters

allow
[bool]

set_user_admin(*username: str*, *admin: bool*) → None

Grant or revoke admin permissions for a user

Parameters

username
[str]

admin
[bool] If True, grant permissions. If False, revoke.

validate_signup_token(*token: str*) → Optional[str]

Validate a signup token

Parameters

token
[str]

Returns

username
[str or None] This will be None if the validation fails

pypicloud.access.base.**get_pwd_context**(*preferred_hash: Optional[str] = None*, *rounds: Optional[int] = None*) → LazyCryptContext

Create a passlib context for hashing passwords

pypicloud.access.base.**group_to_principal**(*group: str*) → str

Convert a group to its corresponding principal

pypicloud.access.base.**groups_to_principals**(*groups: List[str]*) → List[str]

Convert a list of groups to a list of principals

pypicloud.access.base_json module

Abstract backends that are backed by simple JSON

class pypicloud.access.base_json.**IJsonAccessBackend**(*request=None*, *default_read=None*, *default_write=None*, *disallow_fallback=()*, *cache_update=None*, *pwd_context=None*, *token_expiration=604800*, *signing_key=None*)

Bases: *IAccessBackend*

This backend reads the permissions from anything that can provide JSON data

Notes

JSON should look like this:

```
{
    "users": {
        "user1": "hashed_password1",
        "user2": "hashed_password2",
        "user3": "hashed_password3",
        "user4": "hashed_password4",
        "user5": "hashed_password5",
    },
    "groups": {
        "admins": [
        "user1",
        "user2"
        ],
        "group1": [
        "user3"
        ]
    },
    "admins": [
        "user1"
    ]
    "packages": {
        "mypackage": {
            "groups": {
                "group1": ["read', "write"],
                "group2": ["read"],
                "group3": [],
            },
            "users": {
                "user1": ["read", "write"],
                "user2": ["read"],
                "user3": [],
                "user5": ["read"],
            }
        }
    }
}
```

property db

Fetch JSON and cache it for future calls

group_members(*group*)

Get a list of users that belong to a group

Parameters

group
[str]

Returns

users
[list] List of user names

`group_package_permissions`(*group*)

Get a list of all packages that a group has permissions on

Parameters

group
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

`group_permissions`(*package*)

Get a mapping of all groups to their permissions on a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] mapping of group name to a list of permissions (which can contain 'read' and/or 'write')

`groups`(*username=None*)

Get a list of all groups

If a username is specified, get all groups that the user belongs to

Parameters

username
[str, optional]

Returns

groups
[list] List of group names

`is_admin`(*username*)

Check if the user is an admin

Parameters

username
[str]

Returns

is_admin
[bool]

`user_data`(*username=None*)

Get a list of all users or data for a single user

For Mutable backends, this MUST exclude all pending users

Returns

users
[list] Each user is a dict with a 'username' str, and 'admin' bool

user
[dict] If a username is passed in, instead return one user with the fields above plus a 'groups' list.

`user_package_permissions`(*username*)

Get a list of all packages that a user has permissions on

Parameters

username
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

`user_permissions`(*package*)

Get a mapping of all users to their permissions for a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] Mapping of username to a list of permissions (which can contain 'read' and/or 'write')

`class` pypicloud.access.base_json.`IMutableJsonAccessBackend`(*request=None*, *default_read=None*, *default_write=None*, *disallow_fallback=()*, *cache_update=None*, *pwd_context=None*, *token_expiration=604800*, *signing_key=None*)

Bases: *IJsonAccessBackend*, *IMutableAccessBackend*

This backend allows you to store all user and package permissions in a backend that is able to store a json file

Notes

The format is the same as *IJsonAccessBackend*, but with the additional fields:

```
{
    "pending_users": {
        "user1": "hashed_password1",
        "user2": "hashed_password2"
    },
    "allow_registration": true
}
```

`allow_register`()

Check if the backend allows registration

This should only be overridden by mutable backends

Returns

allow
[bool]

`approve_user`(*username*)

Mark a user as approved by the admin

Parameters

username
[str]

`create_group`(*group*)

Create a new group

Parameters

group
[str]

`delete_group`(*group*)

Delete a group

Parameters

group
[str]

`delete_user`(*username*)

Delete a user

Parameters

username
[str]

`edit_group_permission`(*package_name*, *group*, *perm*, *add*)

Grant or revoke a permission for a group on a package

Parameters

package_name
[str]

group
[str]

perm
[{'read', 'write'}]

add
[bool] If True, grant permissions. If False, revoke.

`edit_user_group`(*username*, *group*, *add*)

Add or remove a user to/from a group

Parameters

username
[str]

group
[str]

add
[bool] If True, add to group. If False, remove.

`edit_user_permission`(*package_name*, *username*, *perm*, *add*)
Grant or revoke a permission for a user on a package

Parameters

package_name
[str]

username
[str]

perm
[{'read', 'write'}]

add
[bool] If True, grant permissions. If False, revoke.

`mutable = True`

`pending_users`()
Retrieve a list of all users pending admin approval

Returns

users
[list] List of usernames

`set_allow_register`(*allow*)
Allow or disallow user registration

Parameters

allow
[bool]

`set_user_admin`(*username*, *admin*)
Grant or revoke admin permissions for a user

Parameters

username
[str]

admin
[bool] If True, grant permissions. If False, revoke.

pypicloud.access.config module

Backend that reads access control rules from config file

`class` pypicloud.access.config.`ConfigAccessBackend`(*request=None*, *data=None*, ***kwargs*)
Bases: *IJsonAccessBackend*

Access Backend that uses values set in the config file

`classmethod configure`(*settings*)
Configure the access backend with app settings

load(*data*)

Idempotently load universal access control data.

By default, this does nothing on immutable backends. Backends may override this method to provide an implementation.

This method works by default on mutable backends with no override necessary.

pypicloud.access.ldap_ module

LDAP authentication plugin for pypicloud.

class pypicloud.access.ldap_.**LDAP**(*admin_field*, *admin_group_dn*, *admin_value*, *base_dn*, *cache_time*, *service_dn*, *service_password*, *service_username*, *url*, *user_search_filter*, *user_dn_format*, *ignore_cert*, *ignore_referrals*, *ignore_multiple_results*)

Bases: `object`

Handles interactions with the remote LDAP server

property admin_member_type

connect()

Initializes the python-ldap module and does the initial bind

get_user(*username*)

Get the User object or None

test_connection()

Binds to service. Will throw if bad connection

verify_user(*username*, *password*)

Attempts to bind as the user, then rebinds as service user again

class pypicloud.access.ldap_.**LDAPAccessBackend**(*request=None*, *conn=None*, *fallback_factory=None*, ***kwargs*)

Bases: *IAccessBackend*

This backend allows you to authenticate against a remote LDAP server.

check_health()

Check the health of the access backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

classmethod configure(*settings*)

Configure the access backend with app settings

property fallback

group_members(*group*)

Get a list of users that belong to a group

Parameters

group
[str]

Returns

users
[list] List of user names

`group_package_permissions`(*group*)

Get a list of all packages that a group has permissions on

Parameters

group
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

`group_permissions`(*package*)

Get a mapping of all groups to their permissions on a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] mapping of group name to a list of permissions (which can contain 'read' and/or 'write')

`groups`(*username=None*)

Get a list of all groups

If a username is specified, get all groups that the user belongs to

Parameters

username
[str, optional]

Returns

groups
[list] List of group names

`is_admin`(*username*)

Check if the user is an admin

Parameters

username
[str]

Returns

is_admin
[bool]

`user_data`(*username=None*)

Get a list of all users or data for a single user

For Mutable backends, this MUST exclude all pending users

Returns

users
[list] Each user is a dict with a 'username' str, and 'admin' bool

user
[dict] If a username is passed in, instead return one user with the fields above plus a 'groups' list.

`user_package_permissions`(*username*)

Get a list of all packages that a user has permissions on

Parameters

username
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

`user_permissions`(*package*)

Get a mapping of all users to their permissions for a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] Mapping of username to a list of permissions (which can contain 'read' and/or 'write')

`verify_user`(*username*, *password*)

Check the login credentials of a user

For Mutable backends, pending users should fail to verify

Parameters

username
[str]

password
[str]

Returns

valid
[bool] True if user credentials are valid, false otherwise

`class` pypicloud.access.ldap_.`User`(*username*, *dn*, *is_admin*)

Bases: `tuple`

`property dn`

Alias for field number 1

`property is_admin`

Alias for field number 2

property username

Alias for field number 0

pypicloud.access.ldap_.**reconnect**(*func*)

If the LDAP connection dies underneath us, recreate it

pypicloud.access.remote module

Backend that defers to another server for access control

class pypicloud.access.remote.**RemoteAccessBackend**(*request=None*, *settings=None*, *server=None*, *auth=None*, ***kwargs*)

Bases: *IAccessBackend*

This backend allows you to defer all user auth and permissions to a remote server. It requires the `requests` package.

classmethod configure(*settings*)

Configure the access backend with app settings

group_members(*group*)

Get a list of users that belong to a group

Parameters

group
[str]

Returns

users
[list] List of user names

group_package_permissions(*group*)

Get a list of all packages that a group has permissions on

Parameters

group
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

group_permissions(*package*)

Get a mapping of all groups to their permissions on a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] mapping of group name to a list of permissions (which can contain 'read' and/or 'write')

`groups`(*username=None*)

Get a list of all groups

If a username is specified, get all groups that the user belongs to

Parameters

username
[str, optional]

Returns

groups
[list] List of group names

`is_admin`(*username*)

Check if the user is an admin

Parameters

username
[str]

Returns

is_admin
[bool]

`user_data`(*username=None*)

Get a list of all users or data for a single user

For Mutable backends, this MUST exclude all pending users

Returns

users
[list] Each user is a dict with a 'username' str, and 'admin' bool

user
[dict] If a username is passed in, instead return one user with the fields above plus a 'groups' list.

`user_package_permissions`(*username*)

Get a list of all packages that a user has permissions on

Parameters

username
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

`user_permissions`(*package*)

Get a mapping of all users to their permissions for a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] Mapping of username to a list of permissions (which can contain 'read' and/or 'write')

`verify_user`(*username*, *password*)

Check the login credentials of a user

For Mutable backends, pending users should fail to verify

Parameters

username
[str]

password
[str]

Returns

valid
[bool] True if user credentials are valid, false otherwise

pypicloud.access.sql module

Access backend for storing permissions in using SQLAlchemy

`class pypicloud.access.sql.Group`(*name*)

Bases: `Base`

Group record

`name`

`class pypicloud.access.sql.GroupPermission`(*package*, *groupname*, *read=False*, *write=False*)

Bases: *Permission*

Permissions for a group on a package

`group`

`groupname`

`package`

`read`

`write`

`class pypicloud.access.sql.KeyVal`(*key*, *value*)

Bases: `Base`

Simple model for storing key-value pairs

`key`

`value`

`class pypicloud.access.sql.Permission`(*package*, *read*, *write*)

Bases: `Base`

Base class for user and group permissions

package = Column(None, String(length=255), table=None, primary_key=True, nullable=False)

property permissions

Construct permissions list

read = Column(None, Boolean(), table=None)

write = Column(None, Boolean(), table=None)

class pypicloud.access.sql.**SQLAccessBackend**(*request=None*, *dbmaker=None*, ***kwargs*)

Bases: *IMutableAccessBackend*

This backend allows you to store all user and package permissions in a SQL database

allow_register()

Check if the backend allows registration

This should only be overridden by mutable backends

Returns

allow
[bool]

approve_user(*username*)

Mark a user as approved by the admin

Parameters

username
[str]

check_health()

Check the health of the access backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

classmethod configure(*settings:* EnvironSettings)

Configure the access backend with app settings

create_group(*group*)

Create a new group

Parameters

group
[str]

property db

Lazy-create the DB session

delete_group(*group*)

Delete a group

Parameters

group
[str]

`delete_user`(*username*)

Delete a user

Parameters

username
[str]

`edit_group_permission`(*package_name*, *group*, *perm*, *add*)

Grant or revoke a permission for a group on a package

Parameters

package_name
[str]

group
[str]

perm
[{'read', 'write'}]

add
[bool] If True, grant permissions. If False, revoke.

`edit_user_group`(*username*, *group*, *add*)

Add or remove a user to/from a group

Parameters

username
[str]

group
[str]

add
[bool] If True, add to group. If False, remove.

`edit_user_permission`(*package_name*, *username*, *perm*, *add*)

Grant or revoke a permission for a user on a package

Parameters

package_name
[str]

username
[str]

perm
[{'read', 'write'}]

add
[bool] If True, grant permissions. If False, revoke.

`group_members`(*group*)

Get a list of users that belong to a group

Parameters

group
[str]

Returns

users
[list] List of user names

`group_package_permissions`(*group*)

Get a list of all packages that a group has permissions on

Parameters

group
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

`group_permissions`(*package*)

Get a mapping of all groups to their permissions on a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] mapping of group name to a list of permissions (which can contain 'read' and/or 'write')

`groups`(*username=None*)

Get a list of all groups

If a username is specified, get all groups that the user belongs to

Parameters

username
[str, optional]

Returns

groups
[list] List of group names

`is_admin`(*username*)

Check if the user is an admin

Parameters

username
[str]

Returns

is_admin
[bool]

`need_admin`()

Find out if there are any admin users

This should only be overridden by mutable backends

Returns

need_admin
[bool] True if no admin user exists and the backend is mutable, False otherwise

`pending_users()`

Retrieve a list of all users pending admin approval

Returns

users
[list] List of usernames

`classmethod postfork(**kwargs)`

This method will be called after uWSGI forks

`set_allow_register(allow)`

Allow or disallow user registration

Parameters

allow
[bool]

`set_user_admin(username, admin)`

Grant or revoke admin permissions for a user

Parameters

username
[str]

admin
[bool] If True, grant permissions. If False, revoke.

`user_data(username=None)`

Get a list of all users or data for a single user

For Mutable backends, this MUST exclude all pending users

Returns

users
[list] Each user is a dict with a 'username' str, and 'admin' bool

user
[dict] If a username is passed in, instead return one user with the fields above plus a 'groups' list.

`user_package_permissions(username)`

Get a list of all packages that a user has permissions on

Parameters

username
[str]

Returns

packages
[list] List of dicts. Each dict contains 'package' (str) and 'permissions' (list)

user_permissions(*package*)

Get a mapping of all users to their permissions for a package

Parameters

package
[str] The name of a python package

Returns

permissions
[dict] Mapping of username to a list of permissions (which can contain 'read' and/or 'write')

class pypicloud.access.sql.**User**(*username*, *password*, *pending=True*)

Bases: Base

User record

admin

groups

password

pending

username

class pypicloud.access.sql.**UserPermission**(*package*, *username*, *read=False*, *write=False*)

Bases: *Permission*

Permissions for a user on a package

package

read

user

username

write

Module contents

Classes that provide user and package permissions

pypicloud.access.**includeme**(*config*) → None

Configure the app

pypicloud.cache package

Submodules

pypicloud.cache.base module

Base class for all cache implementations

class pypicloud.cache.base.**ICache**(*request=None*, *storage=None*, *allow_overwrite=None*, *calculate_hashes=True*, *allow_delete=True*)

Bases: object

Base class for a caching database that stores package metadata

all(*name: str*) → List[*Package*]

Search for all versions of a package

Parameters

name
[str] The name of the package

Returns

packages
[list] List of all *Package* s with the given name

check_health() → Tuple[bool, str]

Check the health of the cache backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

clear(*package:* Package) → None

Remove this package from the caching database

Parameters

package
[*Package*]

clear_all() → None

Clear all cached packages from the database

classmethod configure(*settings*)

Configure the cache method with app settings

delete(*package:* Package) → None

Delete this package from the database and from storage

Parameters

package
[*Package*]

distinct() → List[str]

Get all distinct package names

Returns

names
[list] List of package names

`download_response`(*package:* Package)

Pass through to storage

`fetch`(*filename: str*) → *Package*

Get matching package if it exists

Parameters

filename
[str] Name of the package file

Returns

package
[*Package*]

`get_url`(*package:* Package) → str

Get the download url for a package

Parameters

package
[*Package*]

Returns

url
[str]

`new_package`(**args*, ***kwargs*) → *Package*

`classmethod postfork`(***kwargs*)

This method will be called after uWSGI forks

`reload_from_storage`(*clear: bool = True*) → None

Make sure local database is populated with packages

`reload_if_needed`() → None

Reload packages from storage backend if cache is empty

This will be called when the server first starts

`save`(*package:* Package) → None

Save this package to the database

Parameters

package
[*Package*]

`search`(*criteria: Dict[str, List[str]]*, *query_type: str*) → List[*Package*]

Perform a search from pip

Parameters

criteria
[dict] Dictionary containing the search criteria. Pip sends search criteria for "name" and "summary" (typically, both of these lists have the same search values).

Example:

```
{
    "name": ["value1", "value2", ..., "valueN"],
    "summary": ["value1", "value2", ..., "valueN"]
}
```

query_type
[str] Type of query to perform. By default, pip sends "or".

`summary()` → List[Dict[str, Any]]

Summarize package metadata

Returns

packages
[list] List of package dicts, each of which contains 'name', 'summary', and 'last_modified'.

`upload`(*filename: str*, *data: BinaryIO*, *name: Optional[str] = None*, *version: Optional[str] = None*, *summary: Optional[str] = None*, *requires_python: Optional[str] = None*) → *Package*

Save this package to the storage mechanism and to the cache

Parameters

filename
[str] Name of the package file

data
[file] File-like readable object

name
[str, optional] The name of the package (if not provided, will be parsed from filename)

version
[str, optional] The version number of the package (if not provided, will be parsed from filename)

summary
[str, optional] The summary of the package

requires_python
[str, optional] The Python version requirement

Returns

package
[*Package*] The Package object that was uploaded

Raises

e
[ValueError] If the package already exists and allow_overwrite = False

pypicloud.cache.dynamo module

Store package data in DynamoDB

class pypicloud.cache.dynamo.**DynamoCache**(*request=None*, *engine=None*, *graceful_reload=False*, ***kwargs*)

Bases: *ICache*

Caching database that uses DynamoDB

all(*name*)

Search for all versions of a package

Parameters

name
[str] The name of the package

Returns

packages
[list] List of all *Package* s with the given name

check_health()

Check the health of the cache backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

clear(*package*)

Remove this package from the caching database

Parameters

package
[*Package*]

clear_all()

Clear all cached packages from the database

classmethod configure(*settings*)

Configure the cache method with app settings

distinct()

Get all distinct package names

Returns

names
[list] List of package names

fetch(*filename*)

Get matching package if it exists

Parameters

filename
[str] Name of the package file

Returns

package
[*Package*]

new_package(**args*, ***kwargs*)

reload_from_storage(*clear=True*)

Make sure local database is populated with packages

save(*package*)

Save this package to the database

Parameters

package
[*Package*]

summary()

Summarize package metadata

Returns

packages
[list] List of package dicts, each of which contains 'name', 'summary', and 'last_modified'.

class pypicloud.cache.dynamo.**DynamoPackage**(*_, **__)

Bases: *Package*, Model

Python package stored in DynamoDB

data = <flywheel.fields.Field object>

filename = <flywheel.fields.Field object>

last_modified = <flywheel.fields.Field object>

meta_ = <flywheel.model_meta.ModelMetadata object>

name = <flywheel.fields.Field object>

summary = <flywheel.fields.Field object>

version = <flywheel.fields.Field object>

class pypicloud.cache.dynamo.**PackageSummary**(*_, **__)

Bases: Model

Aggregate data about packages

last_modified = <flywheel.fields.Field object>

meta_ = <flywheel.model_meta.ModelMetadata object>

name = <flywheel.fields.Field object>

summary = <flywheel.fields.Field object>

pypicloud.cache.redis_cache module

Store package data in redis

class pypicloud.cache.redis_cache.**RedisCache**(*request=None*, *db=None*, *graceful_reload=False*, ***kwargs*)

Bases: *ICache*

Caching database that uses redis

all(*name*)

Search for all versions of a package

Parameters

name
[str] The name of the package

Returns

packages
[list] List of all *Package* s with the given name

check_health()

Check the health of the cache backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

clear(*package*)

Remove this package from the caching database

Parameters

package
[*Package*]

clear_all()

Clear all cached packages from the database

classmethod configure(*settings*)

Configure the cache method with app settings

distinct()

Get all distinct package names

Returns

names
[list] List of package names

fetch(*filename*)

Get matching package if it exists

Parameters

filename
[str] Name of the package file

Returns

package
[*Package*]

redis_filename_set(*name*)

Get the key to a redis set of filenames for a package

redis_key(*key*)

Get the key to a redis hash that stores a package

redis_prefix = 'pypicloud:'

property redis_set

Get the key to the redis set of package names

redis_summary_key(*name*)

Get the redis key to a summary for a package

reload_from_storage(*clear=True*)

Make sure local database is populated with packages

save(*package*, *pipe=None*, *save_summary=True*)

Save this package to the database

Parameters

package
[*Package*]

summary()

Summarize package metadata

Returns

packages
[list] List of package dicts, each of which contains 'name', 'summary', and 'last_modified'.

pypicloud.cache.redis_cache.**summary_from_package**(*package*)

Create a summary dict from a package

pypicloud.cache.sql module

Store package data in a SQL database

class pypicloud.cache.sql.**JSONEncodedDict**(**args*, ***kwargs*)

Bases: TypeDecorator

Represents an immutable structure as a json-encoded string.

impl

alias of TEXT

process_bind_param(*value*, *dialect*)

Receive a bound parameter value to be converted.

Custom subclasses of _types.TypeDecorator should override this method to provide custom behaviors for incoming data values. This method is called at **statement execution time** and is passed the literal Python data value which is to be associated with a bound parameter in the statement.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

See also:

types_typedecorator

_types.TypeDecorator.process_result_value()

process_result_value(*value*, *dialect*)

Receive a result-row column value to be converted.

Custom subclasses of _types.TypeDecorator should override this method to provide custom behaviors for data values being received in result rows coming from the database. This method is called at **result fetching time** and is passed the literal Python data value that's extracted from a database result row.

The operation could be anything desired to perform custom behavior, such as transforming or deserializing data.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

See also:

types_typedecorator

_types.TypeDecorator.process_bind_param()

class pypicloud.cache.sql.**MutableDict**

Bases: Mutable, dict

SQLAlchemy dict field that tracks changes

classmethod coerce(*key*, *value*)

Convert plain dictionaries to MutableDict.

class pypicloud.cache.sql.**SQLCache**(*request=None*, *dbmaker=None*, *graceful_reload=False*, ***kwargs*)

Bases: *ICache*

Caching database that uses SQLAlchemy

all(*name*)

Search for all versions of a package

Parameters

name
[str] The name of the package

Returns

packages
[list] List of all *Package* s with the given name

`check_health()`

Check the health of the cache backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

`clear(package)`

Remove this package from the caching database

Parameters

package
[*Package*]

`clear_all()`

Clear all cached packages from the database

`classmethod configure(settings: EnvironSettings)`

Configure the cache method with app settings

`distinct()`

Get all distinct package names

Returns

names
[list] List of package names

`fetch(filename)`

Get matching package if it exists

Parameters

filename
[str] Name of the package file

Returns

package
[*Package*]

`new_package(*args, **kwargs)`

`classmethod postfork(**kwargs)`

This method will be called after uWSGI forks

`reload_from_storage(clear=True)`

Make sure local database is populated with packages

`reload_if_needed()`

Reload packages from storage backend if cache is empty

This will be called when the server first starts

`save(package)`

Save this package to the database

Parameters

package
[*Package*]

search(*criteria*, *query_type*)

Perform a search.

Queries are performed as follows:

For the AND query_type, queries within a column will utilize the AND operator, but will not conflict with queries in another column.

(column1 LIKE '%a%' AND column1 LIKE '%b%') OR (column2 LIKE '%c%' AND column2 LIKE '%d%')

For the OR query_type, all queries will utilize the OR operator:

(column1 LIKE '%a%' OR column1 LIKE '%b%') OR (column2 LIKE '%c%' OR column2 LIKE '%d%')

summary()

Summarize package metadata

Returns

packages
[list] List of package dicts, each of which contains 'name', 'summary', and 'last_modified'.

class pypicloud.cache.sql.**SQLPackage**(*name*, *version*, *filename*, *last_modified=None*, *summary=None*, ***kwargs*)

Bases: *Package*, Base

Python package stored in SQLAlchemy

data

filename

last_modified

name

summary

version

class pypicloud.cache.sql.**TZAwareDateTime**(**args*, ***kwargs*)

Bases: TypeDecorator

cache_ok = True

Indicate if statements using this ExternalType are "safe to cache".

The default value None will emit a warning and then not allow caching of a statement which includes this type. Set to False to disable statements using this type from being cached at all without a warning. When set to True, the object's class and selected elements from its state will be used as part of the cache key. For example, using a TypeDecorator:

```
class MyType(TypeDecorator):
    impl = String

    cache_ok = True

    def __init__(self, choices):
        self.choices = tuple(choices)
        self.internal_only = True
```

The cache key for the above type would be equivalent to:

```
>>> MyType(["a", "b", "c"])._static_cache_key
(<class '__main__.MyType'>, ('choices', ('a', 'b', 'c')))
```

The caching scheme will extract attributes from the type that correspond to the names of parameters in the `__init__()` method. Above, the "choices" attribute becomes part of the cache key but "internal_only" does not, because there is no parameter named "internal_only".

The requirements for cacheable elements is that they are hashable and also that they indicate the same SQL rendered for expressions using this type every time for a given cache value.

To accommodate for datatypes that refer to unhashable structures such as dictionaries, sets and lists, these objects can be made "cacheable" by assigning hashable structures to the attributes whose names correspond with the names of the arguments. For example, a datatype which accepts a dictionary of lookup values may publish this as a sorted series of tuples. Given a previously un-cacheable type as:

```
class LookupType(UserDefinedType):
    '"a custom type that accepts a dictionary as a parameter.

    this is the non-cacheable version, as "self.lookup" is not
    hashable.

    '"

    def __init__(self, lookup):
        self.lookup = lookup

    def get_col_spec(self, **kw):
        return "VARCHAR(255)"

    def bind_processor(self, dialect):
        # ...  works with "self.lookup" ...
```

Where "lookup" is a dictionary. The type will not be able to generate a cache key:

```
>>> type_ = LookupType({"a": 10, "b": 20})
>>> type_._static_cache_key
<stdin>:1: SAWarning: UserDefinedType LookupType({'a': 10, 'b': 20}) will not
produce a cache key because the ``cache_ok`` flag is not set to True.
Set this flag to True if this type object's state is safe to use
in a cache key, or False to disable this warning.
symbol('no_cache')
```

If we **did** set up such a cache key, it wouldn't be usable. We would get a tuple structure that contains a dictionary inside of it, which cannot itself be used as a key in a "cache dictionary" such as SQLAlchemy's statement cache, since Python dictionaries aren't hashable:

```
>>> # set cache_ok = True
>>> type_.cache_ok = True

>>> # this is the cache key it would generate
>>> key = type_._static_cache_key
>>> key
(<class '__main__.LookupType'>, ('lookup', {'a': 10, 'b': 20}))
```

(continues on next page)

(continued from previous page)

```
>>> # however this key is not hashable, will fail when used with
>>> # SQLAlchemy statement cache
>>> some_cache = {key: "some sql value"}
Traceback (most recent call last): File "<stdin>", line 1,
in <module> TypeError: unhashable type: 'dict'
```

The type may be made cacheable by assigning a sorted tuple of tuples to the ".lookup" attribute:

```
class LookupType(UserDefinedType):
    '"a custom type that accepts a dictionary as a parameter.

    The dictionary is stored both as itself in a private variable,
    and published in a public variable as a sorted tuple of tuples,
    which is hashable and will also return the same value for any
    two equivalent dictionaries.  Note it assumes the keys and
    values of the dictionary are themselves hashable.

    '"

    cache_ok = True

    def __init__(self, lookup):
        self._lookup = lookup

        # assume keys/values of "lookup" are hashable; otherwise
        # they would also need to be converted in some way here
        self.lookup = tuple(
            (key, lookup[key]) for key in sorted(lookup)
        )

    def get_col_spec(self, **kw):
        return "VARCHAR(255)"

    def bind_processor(self, dialect):
        # ...  works with "self._lookup" ...
```

Where above, the cache key for `LookupType({"a": 10, "b": 20})` will be:

```
>>> LookupType({"a": 10, "b": 20})._static_cache_key
(<class '__main__.LookupType'>, ('lookup', (('a', 10), ('b', 20))))
```

New in version 1.4.14: - added the `cache_ok` flag to allow some configurability of caching for `TypeDecorator` classes.

New in version 1.4.28: - added the `ExternalType` mixin which generalizes the `cache_ok` flag to both the `TypeDecorator` and `UserDefinedType` classes.

See also:

sql_caching

impl

alias of `DateTime`

process_bind_param(*value*, *dialect*)

Receive a bound parameter value to be converted.

Custom subclasses of _types.TypeDecorator should override this method to provide custom behaviors for incoming data values. This method is called at **statement execution time** and is passed the literal Python data value which is to be associated with a bound parameter in the statement.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

See also:

types_typedecorator

_types.TypeDecorator.process_result_value()

process_result_value(*value*, *dialect*)

Receive a result-row column value to be converted.

Custom subclasses of _types.TypeDecorator should override this method to provide custom behaviors for data values being received in result rows coming from the database. This method is called at **result fetching time** and is passed the literal Python data value that's extracted from a database result row.

The operation could be anything desired to perform custom behavior, such as transforming or deserializing data.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be None.
- **dialect** – the Dialect in use.

See also:

types_typedecorator

_types.TypeDecorator.process_bind_param()

pypicloud.cache.sql.**create_schema**(*engine*)

Create the database schema if needed

Parameters

engine
[sqlalchemy.Engine]

Notes

The method should only be called after importing all modules containing models which extend the `Base` object.

`pypicloud.cache.sql.`**`drop_schema`**(*engine*)

Drop the database schema

Parameters

engine
[sqlalchemy.Engine]

Notes

The method should only be called after importing all modules containing models which extend the `Base` object.

Module contents

Caching database implementations

`pypicloud.cache.`**`get_cache_impl`**(*settings*)

Get the cache class from settings

`pypicloud.cache.`**`includeme`**(*config*)

Get and configure the cache db wrapper

pypicloud.storage package

Submodules

pypicloud.storage.azure_blob module

Store packages in Azure Blob Storage

class `pypicloud.storage.azure_blob.`**`AzureBlobStorage`**(*request*, *expire_after=None*, *path_prefix=None*, *redirect_urls=None*, *storage_account_name=None*, *storage_account_key=None*, *storage_container_name=None*)

Bases: *IStorage*

Storage backend that uses Azure Blob Storage

`check_health`()

Check the health of the storage backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

`classmethod configure`(*settings*)

Configure the storage method with app settings

delete(*package*)

Delete a package file

Parameters

package
[*Package*] The package metadata

download_response(*package*)

Return a HTTP Response that will download this package

This is called from the download endpoint

get_path(*package*)

Get the fully-qualified bucket path for a package

list(*factory=<class 'pypicloud.models.Package'>*)

Return a list or generator of all packages

open(*package*)

Get a buffer object that can read the package data

This should be a context manager. It is used in migration scripts, not directly by the web application.

Parameters

package
[*Package*]

Examples

```
with storage.open(package) as pkg_data:
    with open('outfile.tar.gz', 'w') as ofile:
        ofile.write(pkg_data.read())
```

test = False

upload(*package*, *datastream*)

Upload a package file to the storage backend

Parameters

package
[*Package*] The package metadata

datastream
[file] A file-like object that contains the package data

pypicloud.storage.base module

Base class for storage backends

class pypicloud.storage.base.**IStorage**(*request: Request*)

Bases: object

Base class for a backend that stores package files

check_health() → Tuple[bool, str]

Check the health of the storage backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

classmethod configure(*settings*)

Configure the storage method with app settings

delete(*package:* Package) → None

Delete a package file

Parameters

package
[*Package*] The package metadata

download_response(*package:* Package)

Return a HTTP Response that will download this package

This is called from the download endpoint

get_url(*package:* Package) → str

Create or return an HTTP url for a package file

By default this will return a link to the download endpoint

/api/package/<package>/<filename>

Returns

link
[str] Link to the location of this package file

list(*factory: ~typing.Type[~pypicloud.models.Package] = <class 'pypicloud.models.Package'>*) → List[*Package*]

Return a list or generator of all packages

open(*package:* Package)

Get a buffer object that can read the package data

This should be a context manager. It is used in migration scripts, not directly by the web application.

Parameters

package
[*Package*]

Examples

```
with storage.open(package) as pkg_data:
    with open('outfile.tar.gz', 'w') as ofile:
        ofile.write(pkg_data.read())
```

upload(*package:* Package, *datastream:* *BinaryIO*) → None

Upload a package file to the storage backend

Parameters

package
[*Package*] The package metadata

datastream
[file] A file-like object that contains the package data

pypicloud.storage.files module

Store packages as files on disk

class pypicloud.storage.files.**FileStorage**(*request=None*, ***kwargs*)

Bases: *IStorage*

Stores package files on the filesystem

classmethod configure(*settings*)

Configure the storage method with app settings

delete(*package*)

Delete a package file

Parameters

package
[*Package*] The package metadata

download_response(*package*)

Return a HTTP Response that will download this package

This is called from the download endpoint

get_metadata_path(*package*)

Get the fully-qualified file path for a package metadata file

get_path(*package*)

Get the fully-qualified file path for a package

list(*factory=<class 'pypicloud.models.Package'>*)

Return a list or generator of all packages

open(*package*)

Get a buffer object that can read the package data

This should be a context manager. It is used in migration scripts, not directly by the web application.

Parameters

package
[*Package*]

Examples

```
with storage.open(package) as pkg_data:
    with open('outfile.tar.gz', 'w') as ofile:
        ofile.write(pkg_data.read())
```

path_to_meta_path(*path*)

Construct the filename for a metadata file

upload(*package*, *datastream*)

Upload a package file to the storage backend

Parameters

package
[*Package*] The package metadata

datastream
[file] A file-like object that contains the package data

pypicloud.storage.gcs module

Store packages in GCS

class pypicloud.storage.gcs.**GoogleCloudStorage**(*request=None*, *bucket_factory=None*, *service_account_json_filename=None*, *project_id=None*, *use_iam_signer=False*, *iam_signer_service_account_email=None*, ***kwargs*)

Bases: *ObjectStoreStorage*

Storage backend that uses GCS

property bucket

delete(*package*)

Delete the package

classmethod get_bucket(*bucket_name*, *settings*)

list(*factory=<class 'pypicloud.models.Package'>*)

Return a list or generator of all packages

classmethod package_from_object(*obj*, *factory*)

Create a package from a GCS object

test = False

upload(*package*, *datastream*)

Upload the package to GCS

pypicloud.storage.object_store module

Store packages in S3

class pypicloud.storage.object_store.**ObjectStoreStorage**(*request=None*, *expire_after=None*, *bucket_prefix=None*, *prepend_hash=None*, *redirect_urls=None*, *sse=None*, *object_acl=None*, *storage_class=None*, *region_name=None*, *public_url=False*, ***kwargs*)

Bases: *IStorage*

Storage backend base class containing code that is common between supported object stores (S3 / GCS)

calculate_path(*package*)

Calculates the path of a package

classmethod configure(*settings*)

Configure the storage method with app settings

download_response(*package*)

Return a HTTP Response that will download this package

This is called from the download endpoint

get_path(*package*)

Get the fully-qualified bucket path for a package

get_url(*package*)

Create or return an HTTP url for a package file

By default this will return a link to the download endpoint

/api/package/<package>/<filename>

Returns

link
[str] Link to the location of this package file

open(*package*)

Get a buffer object that can read the package data

This should be a context manager. It is used in migration scripts, not directly by the web application.

Parameters

package
[*Package*]

Examples

```
with storage.open(package) as pkg_data:
    with open('outfile.tar.gz', 'w') as ofile:
        ofile.write(pkg_data.read())
```

classmethod package_from_object(*obj*, *factory*)

Subclasses must implement a method for constructing a Package instance from the backend's storage object format

test = False

pypicloud.storage.s3 module

Store packages in S3

class pypicloud.storage.s3.**CloudFrontS3Storage**(*request=None*, *domain=None*, *crypto_pk=None*, *key_id=None*, ***kwargs*)

Bases: *S3Storage*

Storage backend that uses S3 and CloudFront

classmethod configure(*settings*)

Configure the storage method with app settings

class pypicloud.storage.s3.**S3Storage**(*request=None*, *bucket=None*, ***kwargs*)

Bases: *ObjectStoreStorage*

Storage backend that uses S3

check_health()

Check the health of the storage backend

Returns

(healthy, status)
[(bool, str)] Tuple that describes the health status and provides an optional status message

delete(*package*)

Delete a package file

Parameters

package
[*Package*] The package metadata

classmethod get_bucket(*bucket_name: str*, *settings:* EnvironSettings) → boto3.s3.Bucket

list(*factory=<class 'pypicloud.models.Package'>*)

Return a list or generator of all packages

classmethod package_from_object(*obj*, *factory*)

Create a package from a S3 object

test = False

upload(*package*, *datastream*)

Upload a package file to the storage backend

Parameters

package
[*Package*] The package metadata

datastream
[file] A file-like object that contains the package data

Module contents

Storage backend implementations

pypicloud.storage.**get_storage_impl**(*settings*) → Callable[[Any], Any]

Get and configure the storage backend wrapper

pypicloud.views package

Submodules

pypicloud.views.admin module

API endpoints for admin controls

class pypicloud.views.admin.**AdminEndpoints**(*request*)

Bases: object

Collection of admin endpoints

approve_user()

Approve a pending user

create_group()

Create a group

create_user(*password*)

Create a new user

delete_group()

Delete a group

delete_user()

Delete a user

download_access_control()

Download the ACL data as a gzipped-json file

edit_permission()

Edit user permission on a package

generate_token()

Create a signup token for a user

get_group()
Get the members and package permissions for a group

get_groups()
Get the list of groups

get_package_permissions()
Get the user and group permissions set on a package

get_pending_users()
Get the list of pending users

get_user()
Get a single user

get_user_permissions()
Get the package permissions for a user

get_users()
Get the list of users

mutate_group_member()
Add a user to a group

rebuild_package_list()
Rebuild the package cache in the database

set_admin_status(*admin*)
Set a user to be or not to be an admin

toggle_allow_register(*allow*)
Allow or disallow user registration

pypicloud.views.api module

Views for simple api calls that return json data

pypicloud.views.api.**all_packages**(*request*, *verbose=False*)
List all packages

pypicloud.views.api.**change_password**(*request*, *old_password*, *new_password*)
Change a user's password

pypicloud.views.api.**delete_package**(*context*, *request*)
Delete a package

pypicloud.views.api.**download_package**(*context*, *request*)
Download package, or redirect to the download link

pypicloud.views.api.**fetch_dist**(*request*, *url*, *name*, *version*, *summary*, *requires_python*)
Fetch a Distribution and upload it to the storage backend

pypicloud.views.api.**package_versions**(*context*, *request*)
List all unique package versions

pypicloud.views.api.**register**(*request*, *password*)
Register a user

pypicloud.views.api.**upload_package**(*context*, *request*, *content*, *summary=None*, *requires_python=None*)
Upload a package

pypicloud.views.login module

Render views for logging in and out of the web interface

pypicloud.views.login.**do_forbidden**(*request*)
Intercept 403's and return 401's when necessary

pypicloud.views.login.**do_login**(*request*, *username*, *password*)
Check credentials and log in

pypicloud.views.login.**do_token_register**(*request*, *token*, *password*)
Consume a signed token and create a new user

pypicloud.views.login.**get_login_page**(*request*)
Catch login and redirect to login wall

pypicloud.views.login.**handle_register_request**(*request*, *username*, *password*)
Process a request to register a new user

pypicloud.views.login.**logout**(*request*)
Delete the user session

pypicloud.views.login.**register**(*request*, *username*, *password*)
Check credentials and log in

pypicloud.views.login.**register_new_user**(*access*, *username*, *password*)
Register a new user & handle duplicate detection

pypicloud.views.packages module

View for cleaner buildout calls

pypicloud.views.packages.**list_packages**(*request*)
Render the list for all versions of all packages

pypicloud.views.simple module

Views for simple pip interaction

pypicloud.views.simple.**get_fallback_packages**(*request*, *package_name*, *redirect=True*)
Get all package versions for a package from the fallback_base_url

pypicloud.views.simple.**package_versions**(*context*, *request*)
Render the links for all versions of a package

pypicloud.views.simple.**package_versions_json**(*context*, *request*)
Render the package versions in JSON format

pypicloud.views.simple.**packages_to_dict**(*request*, *packages*)
Convert a list of packages to a dict used by the template

pypicloud.views.simple.**search**(*request*, *criteria*, *query_type*)

Perform searches from pip. This handles XML RPC requests to the "pypi" endpoint (configured as /pypi/) that specify the method "search".

pypicloud.views.simple.**simple**(*request*)

Render the list of all unique package names

pypicloud.views.simple.**upload**(*request*, *content*, *name=None*, *version=None*, *summary=None*, *requires_python=None*)

Handle update commands

Module contents

Views

pypicloud.views.**format_exception**(*context*, *request*)

Catch all app exceptions and render them nicely

This will keep the status code, but will always return parseable json

Returns

error
[str] Identifying error key

message
[str] Human-readable error message

stacktrace
[str, optional] If pyramid.debug = true, also return the stacktrace to the client

pypicloud.views.**get_index**(*request*)

Render a home screen

pypicloud.views.**health_endpoint**(*request*)

Simple health endpoint

2.1.2 Submodules

pypicloud.auth module

Utilities for authentication and authorization

class pypicloud.auth.**PypicloudSecurityPolicy**

Bases: `object`

authenticated_userid(*request*)

Return a userid string identifying the trusted and verified user, or `None` if unauthenticated.

If the result is `None`, then `pyramid.request.Request.is_authenticated` will return `False`.

forget(*request*, ***kw*)

Return a set of headers suitable for 'forgetting' the current user on subsequent requests. An individual security policy and its consumers can decide on the composition and meaning of `**kw`.

identity(*request*)

Return the identity of the current user. The object can be of any shape, such as a simple ID string or an ORM object.

permits(*request*, *context*, *permission*)

Return an instance of `pyramid.security.Allowed` if a user of the given identity is allowed the `permission` in the current `context`, else return an instance of `pyramid.security.Denied`.

remember(*request*, *userid*, ***kw*)

Return a set of headers suitable for 'remembering' the userid named `userid` when set in a response. An individual security policy and its consumers can decide on the composition and meaning of `**kw`.

pypicloud.auth.**get_basicauth_credentials**(*request*)

Get the user/password from HTTP basic auth

pypicloud.auth.**includeme**(*config*)

Configure the app

pypicloud.lambda_scripts module

Helpers for syncing packages into the cache in AWS Lambda

pypicloud.lambda_scripts.**build_lambda_bundle**(*argv=None*)

Build the zip bundle that will be deployed to AWS Lambda

pypicloud.lambda_scripts.**create_sync_scripts**(*argv=None*)

Set bucket notifications and create AWS Lambda functions that will sync changes in the S3 bucket to the cache

pypicloud.lambda_scripts.**make_virtualenv**(*env*)

Create a virtualenv

pypicloud.locator module

Simple replacement for distlib SimpleScrapingLocator

class pypicloud.locator.**FormattedScrapingLocator**(*url*, *timeout=None*, *num_workers=10*, ***kwargs*)

Bases: `SimpleScrapingLocator`

get_releases(*project_name*)

class pypicloud.locator.**SimpleJsonLocator**(*base_index*)

Bases: `object`

Simple replacement for distlib SimpleScrapingLocator

get_releases(*project_name*)

pypicloud.locator.**is_compatible**(*wheel*, *tags=None*)

Hacked function to monkey patch into distlib

pypicloud.models module

Model objects

class pypicloud.models.**Package**(*name*, *version*, *filename*, *last_modified=None*, *summary=None*, ***kwargs*)

Bases: object

Representation of a versioned package

Parameters

name
[str] The name of the package (will be normalized)

version
[str] The version number of the package

filename
[str] The name of the package file

last_modified
[datetime, optional] The datetime when this package was uploaded (default now)

summary
[str, optional] The summary of the package

****kwargs**
Metadata about the package

get_metadata()

Returns the package metadata as a dict

get_url(*request*)

Create path to the download link

property is_prerelease

Returns True if the version is a prerelease version

property parsed_version

Parse and cache the version using pkg_resources

static read_metadata(*blob*)

Read metadata from a blob

search_summary()

Data to return from a pip search

pypicloud.route module

Tools and resources for traversal routing

class pypicloud.route.**APIPackageFileResource**(*request*, *name*, *filename*)

Bases: object

Resource for api endpoints dealing with a single package version

class pypicloud.route.**APIPackageResource**(*request*, *name*)

Bases: *IResourceFactory*

Resource for requesting package versions

class pypicloud.route.**APIPackagingResource**(*request*)

Bases: *IResourceFactory*

Resource for api package queries

class pypicloud.route.**APIResource**(*request*)

Bases: *IStaticResource*

Resource for api calls

subobjects = {'package': <class 'pypicloud.route.APIPackagingResource'>}

class pypicloud.route.**AccountResource**(*request*)

Bases: object

Resource for login/logout endpoints

class pypicloud.route.**AdminResource**(*request*)

Bases: *IStaticResource*

Resource for admin calls

class pypicloud.route.**IResourceFactory**(*request*)

Bases: object

Resource that generates child resources from a factory

class pypicloud.route.**IStaticResource**(*request*)

Bases: object

Simple resource base class for static-mapping of paths

subobjects = {}

class pypicloud.route.**PackagesResource**(*request*)

Bases: *IStaticResource*

Resource for cleaner buildout config

class pypicloud.route.**Root**(*request*)

Bases: *IStaticResource*

Root context for PyPI Cloud

subobjects = {'acct': <class 'pypicloud.route.AccountResource'>, 'admin': <class 'pypicloud.route.AdminResource'>, 'api': <class 'pypicloud.route.APIResource'>, 'packages': <class 'pypicloud.route.PackagesResource'>, 'pypi': <class 'pypicloud.route.SimpleResource'>, 'simple': <class 'pypicloud.route.SimpleResource'>}

class pypicloud.route.**SimplePackageResource**(*request*, *name*)

Bases: object

Resource for requesting simple endpoint package versions

class pypicloud.route.**SimpleResource**(*request*)

Bases: object

Resource for simple pip calls

pypicloud.scripts module

Commandline scripts

pypicloud.scripts.**bucket_validate**(*name*)

Check for valid bucket name

pypicloud.scripts.**export_access**(*argv=None*)

Dump the access control data to a universal format

pypicloud.scripts.**gen_password**(*argv=None*)

Generate a salted password

pypicloud.scripts.**import_access**(*argv=None*)

Load the access control data from a dump file or stdin

This operation is idempotent and graceful. It will not clobber your existing ACL.

pypicloud.scripts.**make_config**(*argv=None*)

Create a server config file

pypicloud.scripts.**migrate_packages**(*argv=None*)

Migrate packages from one storage backend to another

Create two config.ini files that are configured to use different storage backends. All packages will be migrated from the storage backend in the first to the storage backend in the second.

ex: pypicloud-migrate-packages file_config.ini s3_config.ini

pypicloud.scripts.**prompt**(*msg*, *default=<object object>*, *validate=None*)

Prompt user for input

pypicloud.scripts.**prompt_option**(*text*, *choices*, *default=<object object>*)

Prompt the user to choose one of a list of options

pypicloud.scripts.**promptyn**(*msg*, *default=None*)

Display a blocking prompt until the user confirms

pypicloud.scripts.**storage_account_name_validate**(*name*)

Check for valid storage account name

pypicloud.scripts.**wrapped_input**(*msg*)

Wraps input for tests

pypicloud.util module

Utilities

class pypicloud.util.**EnvironSettings**(*settings: Dict[str, Any]*, *env: Optional[Dict[str, str]] = None*)

Bases: `object`

clone() → *EnvironSettings*

get(*key: str*, *default: Optional[Any] = None*) → Any

get_as_dict(*prefix: str*, ***kwargs: Callable[[Any], Any]*) → Dict[str, Any]

Convenience method for fetching settings

Returns a dict; any settings that were missing from the config file will not be present in the returned dict (as opposed to being present with a None value)

Parameters

prefix
[str] String to prefix all keys with when fetching value from settings

****kwargs**
[dict] Mapping of setting name to conversion function (e.g. str or asbool)

items() → ItemsView[str, Any]

keys() → KeysView[str]

pop(*key: str*, *default: ~typing.Any = <object object>*) → Any

read_prefix_from_environ(*prefix: str*) → None

setdefault(*key: str*, *value: Any*) → Any

exception pypicloud.util.**PackageParseError**

Bases: ValueError

class pypicloud.util.**TimedCache**(*cache_time: Optional[int]*, *factory: Optional[Callable[[Any], Any]] = None*)

Bases: dict

Dict that will store entries for a given time, then evict them

Parameters

cache_time
[int or None] The amount of time to cache entries for, in seconds. 0 will not cache. None will cache forever.

factory
[callable, optional] If provided, when the TimedCache is accessed and has no value, it will attempt to populate itself by calling this function with the key it was accessed with. This function should return a value to cache, or None if no value is found.

get(*key*, *default=None*)

Return the value for key if key is in the dictionary, else default.

set_expire(*key*, *value*, *expiration*)

Set a value in the cache with a specific expiration

Parameters

key
[str]

value
[value]

expiration
[int or None] Sets the value to expire this many seconds from now. If None, will never expire.

pypicloud.util.**create_matcher**(*queries: List[str]*, *query_type: str*) → Callable[[str], bool]

Create a matcher for a list of queries

Parameters

queries
[list] List of queries

query_type: str
Type of query to run: ["or"|"and"]

Returns

Matcher function

pypicloud.util.**get_packagetype**(*name: str*) → str

Get package type out of a filename

pypicloud.util.**normalize_metadata**(*metadata: Dict[str, Union[str, bytes]]*) → Dict[str, str]

Strip non-ASCII characters from metadata values and replace "_" in metadata keys to "-"

pypicloud.util.**normalize_metadata_value**(*value: Union[str, bytes]*) → str

Strip non-ASCII characters from metadata values

pypicloud.util.**normalize_name**(*name: str*) → str

Normalize a python package name

pypicloud.util.**parse_filename**(*filename: str*, *name: Optional[str] = None*) → Tuple[str, str]

Parse a name and version out of a filename

2.1.3 Module contents

S3-backed pypi server

pypicloud.**includeme**(*config*)

Set up and configure the pypicloud app

pypicloud.**main**(*config*, ***settings*)

This function returns a Pyramid WSGI application.

pypicloud.**to_json**(*value*)

A json filter for jinja2

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W