Craft Store

Canonical Group Ltd

Apr 19, 2024

CONTENTS

CHAPTER

ONE

TUTORIALS

If you want to learn the basics from experience, then our tutorials will help you acquire the necessary competencies from real-life examples with fully reproducible steps.

1.1 Login to the Snap Store

1.1.1 Prerequisites

- Python 3.8 or 3.9
- a clean virtual environment setup
- a text editor
- a developer account on https://snapcraft.io

1.1.2 Setup

Enable the virtual environment and then install Craft Store by running:

```
$ pip install craft-store
```

1.1.3 Code

Write following into a a text editor and save it as `snap_store_login.py`:

```
#!/usr/bin/env python
from craft_store import StoreClient, endpoints

store_client = StoreClient(
    base_url="https://dashboard.snapcraft.io",
    storage_base_url="https://upload.apps.staging.ubuntu.com",
    endpoints=endpoints.SNAP_STORE,
    user_agent="Craft Store Tutorial Agent",
    application_name="cart-store-tutorial"
)

store_client.login(
    permissions=["package_access"],
```

(continues on next page)

(continued from previous page)

```
    description="tutorial-client-login",
    ttl=1000
)
```

1.1.4 Run

Run the saved python module to login:

```
$ python snap_store_login.py
```

1.2 Login to the Snap Store using Ubuntu One

At the end of this tutorial you will have successfully written a script that can log you into the Snap Store using Ubuntu One (https://login.ubuntu.com) and have those credentials stored for the combination of the Snapcraft Dashboard and application name (`ubuntu1-dashboard`).

1.2.1 Prerequisites

- Python 3.8 or 3.9
- a clean virtual environment setup
- a text editor
- a developer account on https://snapcraft.io

1.2.2 Setup

Enable the virtual environment and then install Craft Store by running:

```
$ pip install craft-store click
```

1.2.3 Code

Write following into a a text editor and save it as `snap_store_login_ubuntu_one.py`:

```
import click

from craft_store import *

c = UbuntuOneStoreClient(
    base_url="https://dashboard.snapcraft.io",
    storage_base_url="https://upload.apps.staging.ubuntu.com",
    auth_url="https://login.ubuntu.com",
    endpoints=endpoints.U1_SNAP_STORE,
    application_name="ubuntu1-dashboard",
    user_agent="test",
```

(continues on next page)

(continued from previous page)

```
)

email = click.prompt("Email")
password = click.prompt("Password", hide_input=True)

try:
    c.login(
        permissions=[
            "package_access",
            "package_manage",
            "package_metrics",
            "package_push",
            "package_register",
            "package_release",
            "package_update",
        ],
        description="foo",
        ttl=1800,
        email=email,
        password=password,
    )
except errors.StoreServerError as server_error:
    if "twofactor-required" in server_error.error_list:
        otp = click.prompt("OTP")
        c.login(
            permissions=[
                "package_access",
                "package_manage",
                "package_metrics",
                "package_push",
                "package_register",
                "package_release",
                "package_update",
            ],
            description="foo",
            ttl=1800,
            email=email,
            password=password,
            otp=otp,
        )
```

1.2.4 Run

Run the saved python module to login:

```
$ python snap_store_login_ubuntu_one.py
```

1.3 Get Account email and id from the Snap Store

1.3.1 Prerequisites

- Completed *Login to the Snap Store*
- Shelled into the virtual environment created in *Login to the Snap Store*

1.3.2 Code

Write following into a a text editor and save it as `snap_store_whoami.py`:

```
#!/usr/bin/env python
from craft_store import StoreClient, endpoints

store_client = StoreClient(
    base_url="https://dashboard.snapcraft.io",
    storage_base_url="https://upload.apps.staging.ubuntu.com",
    endpoints=endpoints.SNAP_STORE,
    user_agent="Craft Store Tutorial Agent",
    application_name="cart-store-tutorial"
)

whoami = store_client.whoami()

print(f"email: {whoami['account']['email']}")
print(f"id: {whoami['account']['id']}")
```

1.3.3 Run

Run the saved python module to retrieved the account information for the login:

```
$ python snap_store_whoami.py
```

1.4 Upload a snap to storage

At the end of this tutorial you will be able to upload a snap to file storage and see simple progress and total length updated on the screen as the upload takes place.

1.4.1 Prerequisites

- Python 3.8
- a clean virtual environment setup
- a text editor

1.4.2 Setup

Create a clean virtual environment:

```
$ pip3 -m venv ~/craft-store-upload
$ . ~/craft-store-upload/bin/activate
```

Install Craft Store by running:

```
$ pip install craft-store
```

Obtain a snap to upload by downloading one from the Snap Store and give it a predictable name:

```
$ snap download hello
$ mv hello_*.snap /tmp/hello.snap
```

1.4.3 Code for uploading

Open a text editor to add logic to instantiate a StoreClient for the Staging Snap Store:

```
#!/usr/bin/env python
from pathlib import Path

from craft_store import StoreClient, endpoints

store_client = StoreClient(
    base_url="https://dashboard.staging.snapcraft.io",
    storage_base_url="https://upload.apps.staging.ubuntu.com",
    endpoints=endpoints.SNAP_STORE,
    user_agent="Craft Store Tutorial Agent",
    application_name="craft-store-tutorial"
)

upload_id = store_client.upload_file(filepath=Path("/tmp/hello.snap"))

print(f"upload-id: {upload_id}")
```

Save the file as `snap_store_upload.py`:

1.4.4 Run

Run the saved python module to upload the *hello* snap and obtain an upload-id at the end:

```
$ python snap_store_upload.py
```

1.4.5 Adding progress

Now add a mechanism to view progress for the upload, open the recently saved `snap_store_upload.py` file and modify it so that it looks like the following:

```
#!/usr/bin/env python
from pathlib import Path

from craft_store import StoreClient, endpoints
from requests_toolbelt import MultipartEncoder, MultipartEncoderMonitor


def monitor_callback(encoder: MultipartEncoder) -> None:
    def progress_callback(monitor: MultipartEncoderMonitor) -> None:
        print(f"Uploaded: {monitor.bytes_read} of {monitor.len}")

    return progress_callback


store_client = StoreClient(
    base_url="https://dashboard.staging.snapcraft.io",
    storage_base_url="https://upload.apps.staging.ubuntu.com",
    endpoints=endpoints.SNAP_STORE,
    user_agent="Craft Store Tutorial Agent",
    application_name="craft-store-tutorial"
)

upload_id = store_client.upload_file(
     filepath=Path("/tmp/hello.snap"),
     monitor_callback=progress_callback
)

print(f"upload-id: {upload_id}")
```

Save the file.

1.4.6 Run

Run the saved python module again to upload the *hello* snap and obtain an upload-id at the end, but observing progress as the upload takes place:

```
$ python snap_store_upload.py
```

CHAPTER

TWO

HOW-TO GUIDES

2.1 Using credentials provided by an environment variable

2.1.1 Retrieving the credentials

Use the following snippet to obtain general credentials for your account:

```
#!/usr/bin/env python
from craft_store import StoreClient, endpoints

store_client = StoreClient(
    base_url="https://dashboard.snapcraft.io",
    endpoints=endpoints.SNAP_STORE,
    user_agent="Craft Store Tutorial Agent",
    application_name="cart-store-tutorial"
)

store_client.login(
    permissions=["package_access"],
    description="tutorial-client-login",
    ttl=1000
)

print(f"Exported credentials: {credentials}")
```

Note: The `craft_store.store_client.StoreClient.login()` method has some extra parameters such as `packages` and `channels` to restrict the credentials *reach* even further. Also take consideration into further locking down `permissions` (`craft_store.attenuations`).

2.1.2 Using retrieved credentials

If *craft_store.store_client.StoreClient* is initialized with `environment_auth` and the value is set then a in-memory keyring is used instead of the system keyring.

To make use of such thing, export `CREDENTIALS=<credentials>` where `<credentials>` is the recently retrieved credential. To make use of it and get information from your account:

```
#!/usr/bin/env python
from craft_store import StoreClient, endpoints

store_client = StoreClient(
    base_url="https://dashboard.snapcraft.io",
    endpoints=endpoints.SNAP_STORE,
    user_agent="Craft Store Tutorial Agent",
    application_name="cart-store-tutorial",
    environment_auth="CREDENTIALS",
)

whoami = store_client.whoami()

print(f"email: {whoami['account']['email']}")
print(f"id: {whoami['account']['id']}")
```

2.2 Using craft-cli for upload progress

Progress can be provided by use of craft-cli. This example will upload `./test.snap` with something that looks like the following:

The example requires setting the environment variable `SNAPCRAFT_STORE_CREDENTIALS` using a file generated by snapcraft export-login.

```
#!/usr/bin/env python
from pathlib import Path
import sys

from craft_cli import emit, EmitterMode
from craft_store import StoreClient, endpoints
from requests_toolbelt import MultipartEncoder, MultipartEncoderMonitor


emit.init(EmitterMode.BRIEF, "craft.store-howto", "Starting howto app.")

store_client = StoreClient(
    base_url="https://dashboard.staging.snapcraft.io",
    storage_base_url="https://upload.apps.staging.ubuntu.com",
    endpoints=endpoints.SNAP_STORE,
    user_agent="Craft Store Howto Agent",
    application_name="craft-store-upload",
    environment_auth="SNAPCRAFT_STORE_CREDENTIALS",
)
```

(continues on next page)

(continued from previous page)

```
def create_callback(encoder: MultipartEncoder):
    with emit.progress_bar("Uploading...", encoder.len, delta=False) as progress:

        def progress_callback(monitor: MultipartEncoderMonitor):
            progress.advance(monitor.bytes_read)

    return progress_callback


upload_id = store_client.upload_file(
    monitor_callback=create_callback,
    filepath=Path(sys.argv[1]),
)

emit.message(f"upload-id: {upload_id}")
emit.ended_ok()
```

2.3 Uploading and releasing a package with resources

One of the most common workflows when communicating with the store API is to release a package. This guide explains how to upload and release a package with an associated resource. We will use a charm as an example.

This performs roughly the same store operations as running:

```
charmcraft upload-resource my-charm my-file --filepath cat.gif
charmcraft upload my-charm.charm
charmcraft release my-charm
```

2.3.1 Get a Charmhub client

Create a `StoreClient` instance that points to the staging instance of CharmHub:

```
base_url: str = "https://api.staging.charmhub.io"
storage_base_url: str = "https://storage.staging.snapcraftcontent.com"
client = StoreClient(
    application_name="craft-store-demo",
    base_url=base_url,
    storage_base_url=storage_base_url,
    endpoints=endpoints.CHARMHUB,
    user_agent="craft-store-demo-app",
    environment_auth="CRAFT_STORE_CHARMCRAFT_CREDENTIALS",
)
```

2.3.2 Push the resource

Next, you'll need to:

1. Upload the resource file.
2. Connect that uploaded file with the charm resource.
3. Poll the status while CharmHub processes the file.
4. Retrieve the revision number assigned to the file.

```
resource_upload_id = client.upload_file(filepath=resource_path)
resource_status_url = client.push_resource(
    name=charm_name,
    resource_name=resource_name,
    upload_id=resource_upload_id,
    resource_type=CharmResourceType.FILE,
)
resource_status = check_status(client, base_url + resource_status_url)[0]
resource_revision = int(resource_status["revision"])
```

The snippet above uses a `check_status` helper function that polls CharmHub every three seconds while the file processes.

```
def check_status(client: StoreClient, status_url: str) -> list[dict[str, Any]]:
    """Check the status of an upload."""
    timeout = time.monotonic() + 120
    while time.monotonic() < timeout:
        resource_status = client.request("GET", status_url).json()
        done = True
        for resource_revision_status in resource_status["revisions"]:
            if resource_revision_status["status"] not in ("approved", "rejected"):
                done = False
        if done:
            return resource_status["revisions"]
        time.sleep(3)
    raise TimeoutError("Status was neither approved nor rejected after 120s")
```

This demo uploads the file silently, but the upload progress can also be monitored interactively through a callback, as demonstrated in *Using craft-cli for upload progress*. Likewise, polling the resource's status URL may be done in other (perhaps more user-friendly) ways.

For Charmhub, resources may optionally include a list of bases.

2.3.3 Push the package

This next segment is very similar, as it:

1. Uploads the charm
2. Connects that uploaded file with the charm revision
3. Polls Charmhub while it processes the charm.
4. Retrieves the revision number assigned to the charm upload.

```
charm_upload_id = client.upload_file(filepath=charm_path)
charm_status_url = client.notify_revision(
    name=charm_name,
    revision_request=RevisionsRequestModel(upload_id=charm_upload_id),
).status_url
charm_status = check_status(client, base_url + charm_status_url)[0]
charm_revision = charm_status["revision"]
```

2.3.4 Release the Kraken

Now that the charm and its resource have been uploaded, they can be released to a channel. Upon release, the revision is tied to the relevant resources.

```
client.release(
    name=charm_name,
    release_request=[
        ReleaseRequestModel(
            channel="edge",
            resources=[
                ResourceModel(name=resource_name, revision=resource_revision)
            ],
            revision=charm_revision,
        )
    ],
)
```

Below is a full file containing an executable version of the script in this guide.

```
#!/usr/bin/env python3
"""Demo code for uploading a charm to Charmhub."""
import argparse
import pathlib
import sys
import time
from typing import Any

from craft_store import endpoints, StoreClient
from craft_store.models import (
    RevisionsRequestModel,
    ReleaseRequestModel,
    ResourceModel,
    CharmResourceType,
)


def parse_args(argv: list[str]):
    parser = argparse.ArgumentParser(
        prog="upload_package",
        description="Uploads a charm and its resource",
    )
    parser.add_argument(
        "charm_name",
```

(continues on next page)

(continued from previous page)

```
        help="The name of the charm in the store",
    )
    parser.add_argument(
        "resource_name",
        help="The name of the resource in the store",
    )
    parser.add_argument(
        "--charm",
        type=pathlib.Path,
        required=True,
        help="The path of the charm to upload",
    )
    parser.add_argument(
        "--resource",
        type=pathlib.Path,
        required=True,
        help="The path of the resource file to upload",
    )
    return parser.parse_args(argv)


def check_status(client: StoreClient, status_url: str) -> list[dict[str, Any]]:
    """Check the status of an upload."""
    timeout = time.monotonic() + 120
    while time.monotonic() < timeout:
        resource_status = client.request("GET", status_url).json()
        done = True
        for resource_revision_status in resource_status["revisions"]:
            if resource_revision_status["status"] not in ("approved", "rejected"):
                done = False
        if done:
            return resource_status["revisions"]
        time.sleep(3)
    raise TimeoutError("Status was neither approved nor rejected after 120s")


def main(argv: list[str]):
    args = parse_args(argv)
    charm_path = args.charm.expanduser().resolve()
    resource_path = args.resource.expanduser().resolve()
    charm_name = args.charm_name
    resource_name = args.resource_name

    # [docs:get-client]
    base_url: str = "https://api.staging.charmhub.io"
    storage_base_url: str = "https://storage.staging.snapcraftcontent.com"
    client = StoreClient(
        application_name="craft-store-demo",
        base_url=base_url,
        storage_base_url=storage_base_url,
        endpoints=endpoints.CHARMHUB,
        user_agent="craft-store-demo-app",
```

(continues on next page)

(continued from previous page)

```
        environment_auth="CRAFT_STORE_CHARMCRAFT_CREDENTIALS",
    )
    # [docs:get-client-end]
    # [docs:upload-resource]
    resource_upload_id = client.upload_file(filepath=resource_path)
    resource_status_url = client.push_resource(
        name=charm_name,
        resource_name=resource_name,
        upload_id=resource_upload_id,
        resource_type=CharmResourceType.FILE,
    )
    resource_status = check_status(client, base_url + resource_status_url)[0]
    resource_revision = int(resource_status["revision"])
    # [docs:upload-resource-end]
    # [docs:upload-charm]
    charm_upload_id = client.upload_file(filepath=charm_path)
    charm_status_url = client.notify_revision(
        name=charm_name,
        revision_request=RevisionsRequestModel(upload_id=charm_upload_id),
    ).status_url
    charm_status = check_status(client, base_url + charm_status_url)[0]
    charm_revision = charm_status["revision"]
    # [docs:upload-charm-end]

    # [docs:release]
    client.release(
        name=charm_name,
        release_request=[
            ReleaseRequestModel(
                channel="edge",
                resources=[
                    ResourceModel(name=resource_name, revision=resource_revision)
                ],
                revision=charm_revision,
            )
        ],
    )
    # [docs:release-end]


if __name__ == "__main__":
    main(sys.argv[1:])
```

You can run it as such:

```
  python3 upload_package.py \
    --charm=test.charm \
    --resource=empty-file \
    $my_charm \
    my-file
```

CHAPTER

THREE

REFERENCE

3.1 Automatically-generated Code Documentation

Interact with Canonical services such as Charmhub and the Snap Store.

class craft_store.**Auth**(*application_name: str*, *host: str*, *ephemeral: bool = False*, *environment_auth: str | None = None*)

Auth wraps around the keyring to store credentials.

The application_name and host are used as key/values in the keyring to set, get and delete credentials.

If environment_auth is set on initialization of this class, then a `MemoryKeyring` is setup in lieu of the system one.

Credentials are base64 encoded into the keyring and decoded on retrieval.

Variables

- **application_name** – name of the application using this library.
- **host** – specific host for the store used.

static decode_credentials(*encoded_credentials: str*) → str

Decode base64 encoded credentials.

Raises

errors.CredentialsNotParseable – when the credentials are incorrectly encoded.

del_credentials() → None

Delete credentials from the keyring.

static encode_credentials(*credentials: str*) → str

Encode credentials to base64.

ensure_no_credentials() → None

Check that no credentials exist.

Raises

- **errors.CredentialsAvailable** – if credentials have already been set.
- **errors.KeyringUnlockError** – if the keyring cannot be unlocked.

get_credentials() → str

Retrieve credentials from the keyring.

set_credentials(*credentials: str*, *force: bool = False*) → None

Store credentials in the keyring.

Parameters

- **credentials** – token to store.
- **force** – overwrite existing credentials.

class craft_store.**BaseClient**(*, *base_url: str*, *storage_base_url: str*, *endpoints: Endpoints*, *application_name: str*, *user_agent: str*, *environment_auth: str | None = None*, *ephemeral: bool = False*)

Encapsulates API calls for the Snap Store or Charmhub.

Parameters

- **base_url** – the base url of the API endpoint.
- **storage_base_url** – the base url for storage.
- **endpoints** – `endpoints.CHARMHUB` or `endpoints.SNAP_STORE`.
- **application_name** – the name application using this class, used for the keyring.
- **user_agent** – User-Agent header to use for HTTP(s) requests.
- **environment_auth** – environment variable to use for credentials.
- **ephemeral** – keep everything in memory.

Raises

errors.NoKeyringError – if there is no usable keyring.

get_list_releases(*, *name: str*) → MarshableModel

Query the list_releases endpoint and return the result.

list_registered_names(*, *include_collaborations: bool = False*) → List[RegisteredNameModel]

List the registered names available to the logged in account.

Parameters

include_collaborations – if True, includes names the user is a collaborator on but does not own.

list_resource_revisions(*name: str*, *resource_name: str*) → List[CharmResourceRevision]

List the revisions for a specific resource of a specific name.

list_revisions(*name: str*) → List[RevisionModel]

Get the list of existing revisions for a package.

Parameters

name – the package to lookup.

Returns

a list of revisions that have been uploaded for this package.

Charmhub example: https://api.charmhub.io/docs/default.html#list_revisions

login(*, *permissions: Sequence[str]*, *description: str*, *ttl: int*, *packages: Sequence[Package] | None = None*, *channels: Sequence[str] | None = None*, ***kwargs*) → str

Obtain credentials to perform authenticated requests.

Credentials are stored on the system's keyring, handled by *craft_store.auth.Auth*.

The list of permissions to select from can be referred to on `craft_store.attenuations`.

The login process requires 3 steps:

- request an initial macaroon on `endpoints.Endpoints.tokens`.
- discharge that macaroon using Candid
- send the discharge macaroon to `endpoints.Endpoints.tokens_exchange` to obtain final authorization of the macaroon

This last macaroon is stored into the system's keyring to perform authenticated requests.

Parameters

- **permissions** – Set of permissions to grant the login.
- **description** – Client description to refer to from the Store.
- **ttl** – time to live for the credential, in other words, how long until it expires, expressed in seconds.
- **packages** – Sequence of packages to limit the credentials to.
- **channels** – Sequence of channel names to limit the credentials to.

Raises

errors.CredentialsAlreadyAvailable – if credentials already exist.

logout() → None

Clear credentials.

Raises

errors.CredentialsUnavailable – if credentials cannot be found.

notify_revision(*, *name: str*, *revision_request: RevisionsRequestModel*) → RevisionsResponseModel

Post to the revisions endpoint to notify the store about an upload.

This request usually takes place after a successful `upload`.

push_resource(*name: str*, *resource_name: str*, *, *upload_id: str*, *resource_type: CharmResourceType | None = None*, *bases: ConstrainedListValue[RequestCharmResourceBase] | None = None*) → str

Push a resource revision to the server.

Parameters

- **name** – the (snap, charm, etc.) name to attach the upload to
- **resource_name** – The name of the resource.
- **upload_id** – The ID of the upload (the output of `upload`)
- **resource_type** – If necessary for the namespace, the type of resource.
- **bases** – A list of bases that this file supports.

Returns

The path and query string (as a single string) of the status URL.

API docs: http://api.staging.charmhub.io/docs/default.html#push_resource

The status URL returned is likely a pointer to `list_upload_reviews`: http://api.staging.charmhub.io/docs/default.html#list_upload_reviews

`register_name`(*name: str*, *, *entity_type: Literal['charm', 'bundle', 'snap'] | None = None*, *private: bool = False*, *team: str | None = None*) → str

Register a name on the store.

Parameters

- **name** – the name to register.
- **entity_type** – The type of package to register (e.g. charm or snap)
- **private** – Whether this entity is private or not.
- **team** – An optional team ID to register the name with.

Returns

the ID of the registered name.

`release`(*, *name: str*, *release_request: Sequence[ReleaseRequestModel]*) → None

Request a release of name.

Parameters

- **name** – name to release.
- **release_request** – sequence of items to release.

`request`(*method: str*, *url: str*, *params: Dict[str, str] | None = None*, *headers: Dict[str, str] | None = None*, ***kwargs*) → Response

Perform an authenticated request if auth_headers are True.

Parameters

- **method** – HTTP method used for the request.
- **url** – URL to request with method.
- **params** – Query parameters to be sent along with the request.
- **headers** – Headers to be sent along with the request.

Raises

- **errors.StoreServerError** – for error responses.
- **errors.NetworkError** – for lower level network issues.
- **errors.CredentialsUnavailable** – if credentials cannot be found.

Returns

Response from the request.

`unregister_name`(*name: str*) → str

Unregister a name with no published packages.

Parameters

name – The name to unregister.

Returns

the ID of the deleted name.

`update_resource_revision`(*name: str*, *resource_name: str*, *, *revision: int*, *bases: ConstrainedListValue[RequestCharmResourceBase]*) → int

Update a single resource revision.

update_resource_revisions(**updates: CharmResourceRevisionUpdateRequest*, *name: str*, *resource_name: str*) → int

Update one or more resource revisions.

Parameters

- **name** – The package.
- **resource_name** – The resource name to update.
- **updates** – The updates to make of any revisions

Returns
The number of revisions updated.

upload_file(*, *filepath: Path*, *monitor_callback: Callable | None = None*) → str

Upload filepath to storage.

The monitor_callback is a method receiving one argument of type `MultipartEncoder`, the total length of the upload can be accessed from this encoder from the `len` attribute to setup a progress bar instance.

The callback is to return a function that receives a `MultipartEncoderMonitor` from which the `.bytes_read` attribute can be read to update progress.

The simplest implementation can look like:

```
def monitor_callback(encoder: requests_toolbelt.MultipartEncoder):

    # instantiate progress class with total bytes encoder.len

    def progress_printer(monitor: requests_toolbelt.MultipartEncoderMonitor):
        # Print progress using monitor.bytes_read

    return progress_printer
```

Parameters
monitor_callback – a callback to monitor progress.

whoami() → Dict[str, Any]

Return whoami json data queyring `endpoints.Endpoints.whoami`.

class craft_store.**HTTPClient**(*, *user_agent: str*)

Generic HTTP Client to communicate with Canonical's Developer Gateway.

This client has a requests like interface, it creates a requests.Session on initialization to handle retries over HTTP and HTTPS requests.

The default number of retries is set in `REQUEST_TOTAL_RETRIES` and can be overridden with the `CRAFT_STORE_RETRIES` environment variable.

The backoff factor has a default set in `REQUEST_BACKOFF` and can be overridden with the `CRAFT_STORE_BACKOFF` environment variable.

Retries are done for the following return codes: `500`, `502`, `503` and `504`.

Variables
user_agent – User-Agent header to identify the client.

get(**args*, ***kwargs*) → Response

Perform an HTTP GET request.

post(**args*, ***kwargs*) → Response

Perform an HTTP POST request.

put(**args*, ***kwargs*) → Response

Perform an HTTP PUT request.

request(*method: str*, *url: str*, *params: Dict[str, str] | None = None*, *headers: Dict[str, str] | None = None*, ***kwargs*) → Response

Send a request to url.

`user_agent` is set as part of the headers for the request. All requests are logged through a debug logs, headers matching Authorization and Macaroons have their value replaced.

Parameters

- **method** – HTTP method used for the request.
- **url** – URL to request with method.
- **params** – Query parameters to be sent along with the request.
- **headers** – Headers to be sent along with the request.

Raises

- **errors.StoreServerError** – for error responses.
- **errors.NetworkError** – for lower level network issues.

Returns

Response from the request.

class craft_store.**StoreClient**(***, *base_url: str*, *storage_base_url: str*, *endpoints: Endpoints*, *application_name: str*, *user_agent: str*, *environment_auth: str | None = None*, *ephemeral: bool = False*)

Encapsulates API calls for the Snap Store or Charmhub.

class craft_store.**UbuntuOneStoreClient**(***, *base_url: str*, *storage_base_url: str*, *auth_url: str*, *endpoints: Endpoints*, *application_name: str*, *user_agent: str*, *environment_auth: str | None = None*, *ephemeral: bool = False*)

Encapsulates API calls for the Snap Store or Charmhub with Ubuntu One.

request(*method: str*, *url: str*, *params: Dict[str, str] | None = None*, *headers: Dict[str, str] | None = None*, ***kwargs*) → Response

Make a request to the store.

3.2 Indices and tables

- genindex
- modindex

CHAPTER

FOUR

EXPLANATION

CHAPTER

FIVE

CHANGELOG

5.1 2.6.1 (2024-03-26)

- Remove dependency on `protobuf`
- Explicitly note incompatibility with `keyring` v25.0

5.2 2.6.0 (2024-01-02)

- Add support for listing revisions for a name
- Add support for listing resource revisions
- Add support for updating metadata for resource revisions
- Add support for uploading a resource
- Document the workflow for uploading and releasing a package

5.3 2.5.0 (2023-11-23)

- Add a fallback mechanism for when the system keyring fails, such as the Secret Service keyring (gnome-keyring). The fallback is to write to a file based backend, provided by `craft_store.auth.FileKeyring`
- Removed setup.cfg, fully using pyproject.toml

5.4 2.4.0 (2023-04-13)

- Add support for registering, unregistering, and listing names, with usage examples in integration tests.
  - `craft_store.base_client.BaseClient.register_name`
  - `craft_store.base_client.BaseClient.unregister_name`
  - `craft_store.base_client.BaseClient.list_registered_names`
- Handle keyring unlocking errors

Full Changelog

5.5 2.3.0 (2022-10-07)

- Add support for exporting the new credentials format (which is backwards compatible with the existing one)

5.6 2.2.1 (2022-08-25)

- Export `craft_store.models.SnapListReleasesModel` and `craft_store.models.CharmListReleasesModel`
- Remove incorrectly exported `SnapChannelMapModel` and `CharmChannelMapModel`
- Make bases optional in `craft_store.models.SnapListReleasesModel`

5.7 2.2.0 (2022-08-11)

- Refactor common code in `endpoints`
- Export new symbols in craft_store.models:
  - `craft_store.models.CharmChannelMapModel`
  - `craft_store.models.MarshableModel`
  - `craft_store.models.ReleaseRequestModel`
  - `craft_store.models.RevisionsRequestModel`
  - `craft_store.models.RevisionsResponseModel`
  - `craft_store.models.SnapChannelMapModel`
- Catch the correct `JSONDecodeError`

5.8 2.1.1 (2022-04-26)

- Update macaroon refresh logic for *craft_store.UbuntuOneStoreClient*

5.9 2.1.0 (2022-03-19)

- Support for ephemeral logins in *craft_store.BaseClient*
- New endpoint to complete the upload experience *craft_store.BaseClient.notify_revision()*
- New endpoint to release *craft_store.BaseClient.release()* and retrieve release information *craft_store.BaseClient.get_list_releases()*
- Support for Python 3.10

5.10 2.0.1 (2022-02-10)

- Convert login expiration to a ISO formatted datetime for Ubuntu endpoints
- Raise `craft_store.errors.CredentialsNotParseable` on base64 decode errors
- Use network location as keyring storage location instead of full base url in *craft_store.base_client.BaseClient*

5.11 2.0.0 (2022-02-07)

- New endpoint for uploads to storage, *craft_store.StoreClient* and *craft_store.UbuntuOneStoreClient* require a new initialization new parameter
- Setting credentials while credentials are already set is no longer allowed `craft_store.errors.CredentialsAlreadyAvailable` is raised if credentials already exist
- NotLoggedIn exception renamed to `craft_store.errors.CredentialsUnavailable`
- Early checks are now in place for keyring availability before a login attempt takes place

5.12 1.2.0 (2021-12-09)

- New whoami endpoint for `craft_store.endpoints.CHARMHUB`
- New class to provide login support for Ubuntu One SSO *craft_store.UbuntuOneStoreClient*

5.13 1.1.0 (2021-11-19)

- Support for channels and packages in endpoints
- *craft_store.store_client.StoreClient* support for retrieving credentials from an environment variable
- Login credentials now returned from *craft_store.BaseClient.login()*

5.14 1.0.0 (2021-10-21)

- Initial release

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX