Keg-Storage

Release 0.5.11

Jan 16, 2023

Documentation:

CHAPTER 1

Backends

1.1 Azure Block Blob

class keg_storage.backends.azure.**AzureStorage**(*account: Optional[str] = None*, *key: Optional[str] = None*, *bucket: Optional[str] = None*, *sas_container_url: Optional[str] = None*, *sas_blob_url: Optional[str] = None*, *chunk_size=5242880*, *name: str = 'azure'*)

copy(*path: str*, *new_path: str*)
Copy the remote file specified by *path* to *new_path*.

create_download_url(*path: str, expire: Union[arrow.arrow.Arrow, datetime.datetime]*)
Create an SAS URL that can be used to download a blob without any additional authentication. This url may be accessed directly to download the blob:

requests.get(url)

create_upload_url(*path: str, expire: Union[arrow.arrow.Arrow, datetime.datetime]*)
Create an SAS URL that can be used to upload a blob without any additional authentication. This url can be used in following way to authenticate a client and upload to the pre-specified path:

client = BlobClient.from_blob_url(url) client.upload_blob(data)

delete(*path: str*)
Delete the remote file specified by *path*.

download(*path: str, file_obj: IO, *, progress_callback: Optional[Callable[[int], None]] = None*)
Copies a remote file at *path* to a file-like object *file_obj*.

If desired, a progress callback can be supplied. The function should accept an int parameter, which will be the number of bytes downloaded so far.

get(*path: str*, *dest: str*) → None
Copies a remote file at *path* to the *dest* path given on the local filesystem.

link_to(*path: str, operation: Union[keg_storage.backends.base.ShareLinkOperation, str], expire: Union[arrow.arrow.Arrow, datetime.datetime], output_path: Optional[str] = None, content_type: Optional[str] = None*) → str
Returns a URL allowing direct the specified operations to be performed on the given path

list(*path: str*) → List[keg_storage.backends.base.ListEntry]
Returns a list of *ListEntry's representing files available under the directory or prefix given in 'path*.

open(*path: str, mode: Union[keg_storage.backends.base.FileMode, str]*) → keg_storage.backends.azure.AzureFile
Returns a instance of RemoteFile for the given *path* that can be used for reading and/or writing depending on the *mode* given.

put(*path: str*, *dest: str*) → None
Copies a local file at *path* to a remote file at *dest*.

upload(*file_obj: IO, path: str, *, progress_callback: Optional[Callable[[int], None]] = None*)
Copies the contents of a file-like object *file_obj* to a remote file at *path*

If desired, a progress callback can be supplied. The function should accept an int parameter, which will be the number of bytes uploaded so far.

class keg_storage.backends.azure.**AzureReader**(*mode: keg_storage.backends.base.FileMode*, *blob_client: azure.storage.blob._blob_client.BlobClient*, *chunk_size=5242880*)
The Azure reader uses byte ranged API calls to fill a local buffer to avoid lots of API overhead for small read sizes.

read(*size: int*) → bytes
Read and return up to *size* bytes from the remote file. If the end of the file is reached this should return an empty bytes string.

class keg_storage.backends.azure.**AzureWriter**(*mode: keg_storage.backends.base.FileMode*, *blob_client: azure.storage.blob._blob_client.BlobClient*, *chunk_size=5242880*)
We are using Azure Block Blobs for all operations. The process for writing them is substantially similar to that of S3 with a couple of differences.

1. We generate the IDs for the blocks
2. **There is no separate call to instantiate the upload. The first call to put_block will create** the blob.

close()
Cleanup and deallocate any held resources. This method may be called multiple times on a single instance. If the file was already closed, this method should do nothing.

write(*data: bytes*) → None
Write the data buffer to the remote file.

class keg_storage.backends.azure.**AzureFile**(*mode: keg_storage.backends.base.FileMode*, *blob_client: azure.storage.blob._blob_client.BlobClient*, *chunk_size=5242880*)
Base class for Azure file interface. Since read and write operations are very different and integrating the two would introduce a lot of complexity there are distinct subclasses for files opened for reading and writing.

1.2 Local Filesystem

class keg_storage.backends.filesystem.**LocalFSStorage**(*root: Union[str, pathlib.Path], linked_endpoint: Optional[str] = None, secret_key: Optional[bytes] = None, name: str = None*)

copy(*path: str*, *new_path: str*)
Copy the remote file specified by *path* to *new_path*.

create_link_token(**, path: str, operation: Union[keg_storage.backends.base.ShareLinkOperation, str], expire: Union[arrow.arrow.Arrow, datetime.datetime]*)
Create a signed JWT authorizing the user to perform the specified operations

delete(*path: str*)
Delete the remote file specified by *path*.

deserialize_link_token(*token: str*) → keg_storage.backends.base.InternalLinkTokenData
Verify a JWT and extract the path and allowed operations

download(*path: str, file_obj: IO, *, progress_callback: Optional[Callable[[int], None]] = None*)
Copies a remote file at *path* to a file-like object *file_obj*.

If desired, a progress callback can be supplied. The function should accept an int parameter, which will be the number of bytes downloaded so far.

get(*path: str*, *dest: str*) → None
Copies a remote file at *path* to the *dest* path given on the local filesystem.

link_to(*path: str, operation: Union[keg_storage.backends.base.ShareLinkOperation, str], expire: Union[arrow.arrow.Arrow, datetime.datetime], output_path: Optional[str] = None, content_type: Optional[str] = None*) → str
Create a URL pointing to the given *linked_endpoint* containing a JWT authorizing the user user to perform the given operations.

This is currently only implemented for flask based apps but you may override this method in your own subclass to support other frameworks.

To use this method you must provide *secret_key* and *linked_endpoint* to the constructor.

Note: *content_type* parameter is ignored for this backend.

list(*path: str*) → List[keg_storage.backends.base.ListEntry]
Returns a list of *ListEntry's representing files available under the directory or prefix given in 'path*.

open(*path: str, mode: Union[keg_storage.backends.base.FileMode, str]*)
Returns a instance of RemoteFile for the given *path* that can be used for reading and/or writing depending on the *mode* given.

put(*path: str*, *dest: str*) → None
Copies a local file at *path* to a remote file at *dest*.

upload(*file_obj: IO, path: str, *, progress_callback: Optional[Callable[[int], None]] = None*)
Copies the contents of a file-like object *file_obj* to a remote file at *path*

If desired, a progress callback can be supplied. The function should accept an int parameter, which will be the number of bytes uploaded so far.

class keg_storage.backends.filesystem.**LocalFSFile**(*path: pathlib.Path, mode: keg_storage.backends.base.FileMode*)

close()
Cleanup and deallocate any held resources. This method may be called multiple times on a single instance. If the file was already closed, this method should do nothing.

read(*size: int*) → bytes
Read and return up to *size* bytes from the remote file. If the end of the file is reached this should return an empty bytes string.

write(*data: bytes*) → None
Write the data buffer to the remote file.

1.3 S3 Backend

class keg_storage.backends.s3.**S3Storage**(*bucket*, *aws_region*, *aws_access_key_id=None*, *aws_secret_access_key=None*, *aws_profile=None*, *name='s3'*)

copy(*current_file*, *new_file*)
Copy the remote file specified by *path* to *new_path*.

delete(*path*)
Delete the remote file specified by *path*.

download(*path: str, file_obj: IO, *, progress_callback: Optional[Callable[[int], None]] = None*)
Copies a remote file at *path* to a file-like object *file_obj*.

If desired, a progress callback can be supplied. The function should accept an int parameter, which will be the number of bytes downloaded so far.

get(*path: str*, *dest: str*) → None
Copies a remote file at *path* to the *dest* path given on the local filesystem.

link_to(*path: str, operation: Union[keg_storage.backends.base.ShareLinkOperation, str], expire: Union[arrow.arrow.Arrow, datetime.datetime], output_path: Optional[str] = None, content_type: Optional[str] = None*) → str
Returns a URL allowing direct the specified operations to be performed on the given path

list(*path*)
Returns a list of *ListEntry's representing files available under the directory or prefix given in 'path*.

open(*path: str, mode: Union[keg_storage.backends.base.FileMode, str]*)
Returns a instance of RemoteFile for the given *path* that can be used for reading and/or writing depending on the *mode* given.

put(*path: str*, *dest: str*) → None
Copies a local file at *path* to a remote file at *dest*.

upload(*file_obj: IO, path: str, *, progress_callback: Optional[Callable[[int], None]] = None*)
Copies the contents of a file-like object *file_obj* to a remote file at *path*

If desired, a progress callback can be supplied. The function should accept an int parameter, which will be the number of bytes uploaded so far.

class keg_storage.backends.s3.**S3Reader**(*bucket*, *filename*, *client*)

close()
Cleanup and deallocate any held resources. This method may be called multiple times on a single instance. If the file was already closed, this method should do nothing.

read(*size: int*)
Read and return up to *size* bytes from the remote file. If the end of the file is reached this should return an empty bytes string.

class keg_storage.backends.s3.**S3Writer**(*bucket*, *filename*, *client*, *chunk_size=10485760*)
Writes to S3 are quite a bit more complicated than reads. To support large files, we cannot write in a single operation and the API does not encourage streaming writes so we make use of the multipart API methods.

The process can be summarized as:

- Create a multipart upload and get an upload key to use with subsequent calls.
- Upload "parts" of the file using the upload key and get back an ID for each part.
- Combine the parts using the upload key and all the part IDs from the above steps.

The chunked nature of the uploads should be mostly invisible to the caller since S3Writer maintains a local buffer.

Because creating a multipart upload itself has an actual cost and there is no guarantee that anything will actually be written, we initialize the multipart upload lazily.

abort()
Use if for some reason you want to discard all the data written and not create an S3 object

close()
Cleanup and deallocate any held resources. This method may be called multiple times on a single instance. If the file was already closed, this method should do nothing.

write(*data: bytes*)
Write the data buffer to the remote file.

class keg_storage.backends.s3.**S3FileBase**(*mode*, *bucket*, *filename*, *client*)
Read and write operations for S3 are very different so individual subclasses are used for each. Read+Write mode is not available for this backend.

1.4 SFTP

class keg_storage.backends.sftp.**SFTPStorage**(*host*, *username*, *key_filename*, *known_hosts_fpath*, *port=22*, *allow_agent=False*, *look_for_keys=False*, *linked_endpoint=None*, *secret_key=None*, *name='sftp'*)

copy(*path: str*, *new_path: str*)
Copy the remote file specified by *path* to *new_path*.

create_link_token(**, path: str, operation: Union[keg_storage.backends.base.ShareLinkOperation, str], expire: Union[arrow.arrow.Arrow, datetime.datetime]*)
Create a signed JWT authorizing the user to perform the specified operations

delete(*path: str*)
Delete the remote file specified by *path*.

deserialize_link_token(*token: str*) → keg_storage.backends.base.InternalLinkTokenData
Verify a JWT and extract the path and allowed operations

download(*path: str, file_obj: IO, *, progress_callback: Optional[Callable[[int], None]] = None*)
Copies a remote file at *path* to a file-like object *file_obj*.

If desired, a progress callback can be supplied. The function should accept an int parameter, which will be the number of bytes downloaded so far.

get(*path: str*, *dest: str*) → None
Copies a remote file at *path* to the *dest* path given on the local filesystem.

link_to(*path: str, operation: Union[keg_storage.backends.base.ShareLinkOperation, str], expire: Union[arrow.arrow.Arrow, datetime.datetime], output_path: Optional[str] = None, content_type: Optional[str] = None*) → str
Create a URL pointing to the given *linked_endpoint* containing a JWT authorizing the user user to perform the given operations.

This is currently only implemented for flask based apps but you may override this method in your own subclass to support other frameworks.

To use this method you must provide *secret_key* and *linked_endpoint* to the constructor.

Note: *content_type* parameter is ignored for this backend.

list(*path: str*)
Returns a list of *ListEntry's representing files available under the directory or prefix given in 'path*.

open(*path: str, mode: Union[keg_storage.backends.base.FileMode, str]*)
Returns a instance of RemoteFile for the given *path* that can be used for reading and/or writing depending on the *mode* given.

put(*path: str*, *dest: str*) → None
Copies a local file at *path* to a remote file at *dest*.

upload(*file_obj: IO, path: str, *, progress_callback: Optional[Callable[[int], None]] = None*)
Copies the contents of a file-like object *file_obj* to a remote file at *path*

If desired, a progress callback can be supplied. The function should accept an int parameter, which will be the number of bytes uploaded so far.

class keg_storage.backends.sftp.**SFTPRemoteFile**(*mode*, *path*, *client*)

close()
Cleanup and deallocate any held resources. This method may be called multiple times on a single instance. If the file was already closed, this method should do nothing.

read(*size: int*)
Read and return up to *size* bytes from the remote file. If the end of the file is reached this should return an empty bytes string.

write(*data: bytes*)
Write the data buffer to the remote file.

1.5 Utilities

class keg_storage.backends.base.**FileMode**
An enumeration.

class keg_storage.backends.base.**ShareLinkOperation**
An enumeration.

class keg_storage.backends.base.**InternalLinkTokenData**(*path*, *operations*)

operations
Alias for field number 1

path
Alias for field number 0

class keg_storage.backends.base.**InternalLinksStorageBackend**(**, linked_endpoint: Optional[str], secret_key: Optional[bytes], name: str*)

Base class for storage backends that do not have their own direct method of creating download/upload/deletion URLs. To use the link_to feature for such backends, the app must provide it's own endpoint to handle the requests. See plugin.LinkViewMixin for a base implementation of such an endpoint.

create_link_token(**, path: str, operation: Union[keg_storage.backends.base.ShareLinkOperation, str], expire: Union[arrow.arrow.Arrow, datetime.datetime]*)
Create a signed JWT authorizing the user to perform the specified operations

deserialize_link_token(*token: str*) → keg_storage.backends.base.InternalLinkTokenData
Verify a JWT and extract the path and allowed operations

link_to(*path: str, operation: Union[keg_storage.backends.base.ShareLinkOperation, str], expire: Union[arrow.arrow.Arrow, datetime.datetime], output_path: Optional[str] = None, content_type: Optional[str] = None*) → str
Create a URL pointing to the given *linked_endpoint* containing a JWT authorizing the user user to perform the given operations.

This is currently only implemented for flask based apps but you may override this method in your own subclass to support other frameworks.

To use this method you must provide *secret_key* and *linked_endpoint* to the constructor.

Note: *content_type* parameter is ignored for this backend.

CHAPTER 2

Configuration

2.1 Storage Profiles

Configure storage backends using the `KEG_STORAGE_PROFILES` setting. This should be a list of 2-tuples, matching a `keg_storage.backends.StorageBackend` with a dict of initialization arguments.

For an example, refer to `keg_storage_ta.config.DefaultProfile`.

CHAPTER 3

Usage

3.1 S3

3.1.1 Pre-signed URLs

The *link_to* function for the S3 backend creates a temporary, pre-signed URL that can be used for uploads or downloads.

Uploads

- PUT request required
- **Must have a header of *content-type: application/octet-stream* set**
  - If header doesn't match the expected value, you will get a 400 error
- **Make sure you have permissions to the key you are creating**
  - The SDK will happily generate pre-signed URLs that are not available to the generating user
- Body is file contents

JavaScript example:

```
const resp = await axios.default.put(storageUrl, file, {
    headers: { "content-type": "application/octet-stream" },
});
```

3.2 StorageOperations wrapper/mixin

class keg_storage.**StorageOperations**

Ops wrapper for storage operations that will typically occur in a flask app.

Assumes the storage plugin is being used and configured with storage profiles.

Class properties storage_location and storage_profile may be assigned defaults in a subclass direct any of the operations to that folder path or configured interface. storage_location is expected to be an Enum.

Each method will also take storage_location and storage_profile, so they can be provided directly for one-offs. So, this class can be used directly or as a mixin.

classmethod storage_delete_file(*filename*, *storage_location=None*, *storage_profile=None*)

Remove file data from storage.

classmethod storage_download_file(*filename*, *storage_location=None*, *storage_profile=None*)

Pull file data from storage, return BytesIO stream.

classmethod storage_duplicate_file(*filename*, *storage_location=None*, *storage_profile=None*)

Copy file data already in storage to a new file object. Generates the new filename using a UUID.

static storage_generate_filename(*filename*)

Generate a UUID-based filename for an object, typically for upload to prevent path collisions. If the provided original filename has an extension, honor that extension.

classmethod storage_get_download_link(*filename*, *expire_minutes*, *storage_location=None*, *storage_profile=None*, ***kwargs*)

Generate an expiring download link to pass to client for a stored object.

classmethod storage_get_profile(*storage_profile=None*)

Get configured storage interface. Either specify which interface via the storage_profile kwarg, or it will fall back to the first defined profile.

classmethod storage_get_upload_link(*filename*, *expire_minutes*, *storage_location=None*, *storage_profile=None*)

Generate an expiring upload link to pass to client for data to be stored.

static storage_prefix_path(*location*, *filename*)

Join the location path with the filename to get the full object path

classmethod storage_upload_file(*file_object*, *filename*, *preserve_filename=False*, *storage_location=None*, *storage_profile=None*)

Push file data to storage. A UUID-based filename will be generated to prevent path collisions unless preserve_filename is set.

classmethod storage_upload_form_file(*form_field: str*, *storage_location=None*, *storage_profile=None*)

Shortcut to push file data from posted form to storage.

Index

I

L

O

P

R

S

U

W