Keg-Storage

Release 0.4.1

Oct 31, 2019

Documenation:

CHAPTER 1

Backends

1.1 Azure Block Blob

class keg_storage.backends.azure.**AzureStorage**(*account: str*, *key: str*, *bucket: str*, *name: str = 'azure'*)

> **delete**(*path: str*)
> Delete the remote file specified by *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], buffer_size: int = 10485760*) → 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.

class keg_storage.backends.azure.**AzureReader**(*container_name: str*, *path: str*, *mode: keg_storage.backends.base.FileMode*, *service: azure.storage.blob.blockblobservice.BlockBlobService*, *chunk_size=10485760*)

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**(*container_name: str*, *path: str*, *mode: keg_storage.backends.base.FileMode*, *service: azure.storage.blob.blockblobservice.BlockBlobService*, *chunk_size: int = 4194304*)

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**(*container_name: str*, *path: str*, *mode: keg_storage.backends.base.FileMode*, *service: azure.storage.blob.blockblobservice.BlockBlobService*, *chunk_size: int = 10485760*)
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 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'*)

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

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.

1.3 SFTP

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.

class keg_storage.backends.sftp.**SFTPStorage**(*host*, *username*, *key_filename*, *known_hosts_fpath*, *allow_agent=False*, *look_for_keys=False*, *name='sftp'*)

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

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.

1.4 Utilities

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

Index