MinIO APIS OVERVIEW

Object API's

`fput_object`

The fput_object method is used to upload a file from the local file system to a bucket on the MinIO server.

```
Minio.fput_object(
    bucket_name: str,
    object_name: str,
    file_path: str,
    content_type: Optional[str] = None,
    metadata: Optional[Dict[str, str]] = None,
    sse: Optional[Base] = None,
    progress: Optional[io.BytesIO] = None,
    part_size: Optional[int] = None,
    tags: Optional[Dict[str, str]] = None,
    retention: Optional[ObjectLock] = None,
    legal_hold: Optional[bool] = None,
    storage_class: Optional[str] = None
) -> ObjectWriteResult
```

Parameters

- **bucket_name (str):** The name of the bucket where the object will be stored. Must already exist.
- **object_name (str):** The name of the object to be created in the bucket. It can include prefixes (like directories).
- **file_path (str):** The local file path to the file that you want to upload.
- **content_type (Optional[str]):** The content type (MIME type) of the object, e.g., "application/octet-stream". If not specified, MinIO will attempt to infer it based on the file extension.
- **metadata (Optional[Dict[str, str]]):** A dictionary of user-defined metadata to be stored with the object. Metadata keys must start with "x-amz-meta-" to be stored correctly.

- **sse (Optional[Base]):** Server-side encryption configuration. You can use this to encrypt the object with server-side encryption.
- **progress (Optional[io.BytesIO]):** A file-like object to track the progress of the upload. This is useful for large files where you want to show progress to the user.
- **part_size (Optional[int]):** The part size (in bytes) to use for multipart uploads. If not specified, the SDK will calculate a part size based on the file size.
- **tags (Optional[Dict[str, str]]):** A dictionary of tags to be assigned to the object.
- **retention (Optional[ObjectLock]):** Object retention configuration. This allows you to specify the retention policy for the object, such as Governance or Compliance mode.
- **legal_hold (Optional[bool]):** Whether or not to place a legal hold on the object. A legal hold prevents the object from being deleted or overwritten until it is lifted.
- **storage_class (Optional[str]):** The storage class to be assigned to the object. Common options include "STANDARD" and "REDUCED_REDUNDANCY".

Return Value

ObjectWriteResult: The method returns an ObjectWriteResult object, which contains metadata about the uploaded object, such as the version ID and ETag

Example Code

```
from minio import Minio
from minio.error import S3Error
from minio.sse import SseCustomerKey, SseKms, SseS3
from minio.commonconfig import GOVERNANCE, LegalHold, RETENTION, Tags
import io
import os

# Initialize MinIO client
client = Minio(
    "play.min.io",  # Replace with your MinIO server endpoint
    access_key="YOUR-ACCESS-KEY",  # Replace with your access key
    secret_key="YOUR-SECRET-KEY",  # Replace with your secret key
       secure=True  # Use True if your MinIO server uses HTTPS
)

# Define bucket, object, and file path
bucket_name = "mybucket"
object_name = "myfile.txt"
file_path = "/path/to/local/file.txt"

# Set up a progress tracker (optional)
progress = io.BytesIO()

# Define server-side encryption (optional)
```

```python
# Example: Use SSE-S3 encryption (server-side encryption with S3 managed keys)
sse = SseS3()

# Define custom metadata (optional)
metadata = {
    "x-amz-meta-mykey": "myvalue",
    "x-amz-meta-anotherkey": "anothervalue"
}

# Define tags (optional)
tags = {
    "project": "minio",
      "user": "testuser"
}

# Define retention configuration (optional)
retention = RETENTION(GOVERNANCE, "2024-12-31T00:00:00Z")

# Define legal hold (optional)
legal_hold = LegalHold(True)

# Set part size (optional)
part_size = 5 * 1024 * 1024  # 5MB part size for multipart upload

# Upload the file with all parameters
try:
      result = client.fput_object(
        bucket_name=bucket_name,
        object_name=object_name,
        file_path=file_path,
        content_type="text/plain",
        metadata=metadata,
      sse=sse,
        progress=progress,
        part_size=part_size,
      tags=tags,
        retention=retention,
        legal_hold=legal_hold,
        storage_class="REDUCED_REDUNDANCY"
      )
      print(f"File uploaded successfully with ETag: {result.etag}")
except S3Error as exc:
    print("Error occurred:", exc)
```

Explanation of the Code:

1. **MinIO Client Initialization:**
    - The client is initialized with the MinIO server's endpoint, access key, secret key, and a flag indicating whether to use HTTPS.
2. **Progress Tracker:**
    - An `io.BytesIO()` object is used to track the upload progress. .
3. **Server-Side Encryption (SSE):**
    - The `SseS3()` object is used for server-side encryption with S3 managed keys. You can also use `SseKms()` for KMS-managed keys or `SseCustomerKey()` for client-side keys.
4. **Custom Metadata:**
    - Custom metadata is defined as a dictionary. Keys must start with `"x-amz-meta-"` to be recognized as metadata.
5. **Tags:**
    - Tags are defined as a dictionary. These are key-value pairs associated with the object for easier search and categorization.
6. **Retention Configuration:**
    - The `RETENTION` object is used to set a retention policy. In this example, Governance mode is applied until December 31, 2024.
7. **Legal Hold:**
    - A `LegalHold` object is used to prevent the object from being deleted or modified until the hold is lifted.
8. **Part Size:**
    - The `part_size` parameter defines the size of each part for multipart uploads. Here, it's set to 5MB.
9. **Storage Class:**
    - The `storage_class` parameter sets the storage class to `"REDUCED_REDUNDANCY"`, which offers lower redundancy and storage costs.
10. **File Upload:**
    - The `fput_object` method uploads the file with all specified parameters.

`fget_object`

The `fget_object` method allows you to download an object from a MinIO bucket to a specified file path on your local system.

```
Minio.fget_object(
    bucket_name: str,
    object_name: str,
    file_path: str,
    request_headers: Optional[Dict[str, str]] = None,
    sse: Optional[Base] = None,
    version_id: Optional[str] = None,
    progress: Optional[io.BytesIO] = None,
    offset: Optional[int] = 0,
    length: Optional[int] = 0
) -> ObjectWriteResult
```

Parameters

- **`bucket_name` (str):** The name of the bucket containing the object you want to download.
- **`object_name` (str):** The name of the object in the bucket that you want to download.
- **`file_path` (str):** The local file path where the downloaded object will be saved.
- **`request_headers` (Optional[Dict[str, str]]):** A dictionary of additional HTTP headers to include in the request. This is useful if you need to include specific headers for authentication or other purposes.
- **`sse` (Optional[Base]):** Server-side encryption configuration. If the object is encrypted on the server, you need to provide the appropriate decryption keys here.
    - `SseCustomerKey`: For SSE-C (Server-Side Encryption with Customer-Provided Keys).
    - `SseKms`: For SSE-KMS (Server-Side Encryption with Key Management Service).
    - `SseS3`: For SSE-S3 (Server-Side Encryption with S3-Managed Keys).
- **`version_id` (Optional[str]):** If versioning is enabled on the bucket, you can specify a particular version of the object to download.
- **`progress` (Optional[io.BytesIO]):** A file-like object to track the progress of the download. This is useful for monitoring the download of large objects.

- **`offset` (Optional[int]):** The starting byte position from which to download the object. This is useful for downloading a specific byte range of the object.
- **`length` (Optional[int]):** The total number of bytes to download from the object. Combined with `offset`, this allows you to download specific parts of the object.

Return Value

- **`ObjectWriteResult`:** The method returns an `ObjectWriteResult` object, which contains metadata about the downloaded object, such as the ETag and version ID (if applicable).

Example Code

```
from minio import Minio
from minio.error import S3Error
from minio.sse import SseCustomerKey
import io

# Initialize MinIO client
client = Minio(
    "play.min.io",  # Replace with your MinIO server endpoint
    access_key="YOUR-ACCESS-KEY",  # Replace with your access key
    secret_key="YOUR-SECRET-KEY",  # Replace with your secret key
    secure=True  # Use True if your MinIO server uses HTTPS
)

# Define bucket, object, and file path
bucket_name = "mybucket"
object_name = "myfile.txt"
file_path = "/path/to/local/file.txt"

# Set up a progress tracker (optional)
progress = io.BytesIO()

# Define server-side encryption (optional)
# Example: Use SSE-C encryption (server-side encryption with customer-provided
keys)
sse = SseCustomerKey(b"32byteslongsecretkeymustbeexactly32b")

# Define additional request headers (optional)
request_headers = {
    "x-amz-request-payer": "requester"
}

# Define specific version (optional)
version_id = "1234567890abcdef"
```

```
# Download the file with all parameters
try:
    client.fget_object(
        bucket_name=bucket_name,
        object_name=object_name,
        file_path=file_path,
        request_headers=request_headers,
        sse=sse,
        version_id=version_id,
        progress=progress,
        offset=0,  # Start from the beginning
        length=0   # Download the entire object
    )
    print(f"File downloaded successfully to {file_path}")
except S3Error as exc:
    print("Error occurred:", exc)
```

Explanation of the Code:

1. **MinIO Client Initialization:**
    - The client is initialized with the MinIO server's endpoint, access key, secret key, and a flag indicating whether to use HTTPS.
2. **Progress Tracker:**
    - An `io.BytesIO()` object is used to track the download progress.
3. **Server-Side Encryption (SSE):**
    - The `SseCustomerKey()` object is used for server-side encryption with customer-provided keys (SSE-C). If the object is encrypted on the server, you need to provide the correct decryption key to download the object.
4. **Request Headers:**
    - Custom HTTP headers can be added to the download request. For example, the `"x-amz-request-payer"` header is included to handle requester-pays buckets.
5. **Version ID:**
    - If versioning is enabled on the bucket, you can specify a particular version of the object to download by providing the `version_id`.
6. **Offset and Length:**
    - The `offset` and `length` parameters allow you to download a specific byte range of the object. In this example, the entire object is downloaded starting from byte 0.
7. **File Download:**
    - The `fget_object` method downloads the specified object from the bucket and saves it to the local file system at the specified `file_path`

When to use offsets and length

Resuming Interrupted Downloads

- **Scenario**: Suppose you're downloading a large file, and the download is interrupted. You don't want to start from the beginning again.
- **How to Know the Offset**:
    - Track the number of bytes successfully downloaded before the interruption.
    - If your last successful chunk ended at byte 5000, you would start the next download from byte 5001.

```
offset = 5001  # Start from byte 5001
client.fget_object("my-bucket", "my-large-object", "path/to/file",
offset=offset)
```

Processing Large Files in Chunks

- **Scenario**: You need to process a large log file or video in smaller, manageable chunks.
- **How to Know the Offset**:
    - Determine the size of each chunk based on your application's memory limits or processing logic.
    - For example, if you want to process 1 MB chunks, you would calculate the offset as `chunk_number * chunk_size`.

```
chunk_size = 1 * 1024 * 1024  # 1 MB chunk size
for i in range(number_of_chunks):
    offset = i * chunk_size
    client.fget_object("my-bucket", "my-large-object",
f"path/to/file_chunk_{i}", offset=offset, length=chunk_size)
```

Partial Retrieval Based on Known File Structure

- **Scenario**: You're working with a structured file format (like a video, database dump, or custom binary format) where specific data sections are located at known byte offsets.
- **How to Know the Offset**:
    - Refer to the file format specification to know where each section starts and ends.

- For example, a video file might have metadata at the beginning, and the actual content starts after a known number of bytes.

```
metadata_offset = 0
metadata_length = 1024  # Known length of the metadata section
client.fget_object("my-bucket", "my-video-file", "metadata.bin",
offset=metadata_offset, length=metadata_length)
```

`remove_object`

The remove_object function in MinIO's SDK is used to delete an object from a specified bucket. Optionally, if object versioning is enabled, you can also delete a specific version of the object. Additionally, you can choose to bypass governance mode when removing an object, if required.

```
def remove_object(
    bucket_name: str,
    object_name: str,
    version_id: Optional[str] = None,
    bypass_governance: bool = False
) -> None
```

Parameters:

- **`bucket_name`** (str): The name of the bucket from which the object should be deleted.
- **`object_name`** (str): The name of the object to be deleted.
- **`version_id`** (Optional[str], default=None): If versioning is enabled, the `version_id` specifies the version of the object to delete. If this is not provided, the latest version is deleted.
- **`bypass_governance`** (bool, default=False): If set to `True`, this allows the deletion of an object even when a governance mode retention policy is in place. This requires special permissions.

Examples

Deleting an object without versioning:

```
# Remove an object from a bucket
minioClient.remove_object("my-bucket", "my-object")
```

Deleting an object with versioning:

```
# Remove a specific version of an object
minioClient.remove_object("my-bucket", "my-object", version_id="3f79a9...")
```

Deleting an object with governance bypass:

```
# Remove an object, bypassing governance mode
minioClient.remove_object("my-bucket", "my-object", bypass_governance=True)
```

`remove_objects`

The `remove_objects` function in the MinIO SDK is used to **delete multiple objects** from a specified bucket in a single operation. This function is useful when you need to delete a list of objects efficiently, rather than removing them one by one.

```
def remove_objects(
    bucket_name: str,
    delete_object_list: Iterable[str]
) -> List[DeleteError]
```

Parameters:

- **`bucket_name`** (str): The name of the bucket from which the objects should be deleted.
- **`delete_object_list`** (Iterable[str]): A list or iterable of object names (strings) that you want to delete from the bucket.

Return Value:

- **Returns** a list of **`DeleteError`** objects, if any errors occur during the deletion process. Each `DeleteError` contains information about the object and the error encountered.

- If no errors occur, the function completes the deletion and returns an empty list.

Example

```
from minio import Minio
from minio.error import S3Error

# Initialize the MinIO client
minioClient = Minio(
    "play.min.io",
    access_key="your-access-key",
    secret_key="your-secret-key",
    secure=True
)

try:
    # List of objects to be deleted
    objects_to_delete = ["object1.txt", "object2.txt", "object3.txt"]

    # Remove multiple objects from the bucket
    errors = minioClient.remove_objects("my-bucket", objects_to_delete)

    if errors:
        for error in errors:
            print(f"Failed to delete {error.object_name}: {error}")
    else:
        print("All objects deleted successfully.")
except S3Error as err:
    print(f"Error: {err}")
```

`copy_object`

The `copy_object` function in the MinIO SDK is used to **copy an object from one bucket to another** or within the same bucket. It allows copying objects across different buckets, either on

the same MinIO server or across different servers, while optionally modifying metadata, adding storage class information, or enabling encryption.

```
def copy_object(
    bucket_name: str,
    object_name: str,
    copy_source: CopySource,
    metadata: Optional[dict] = None,
    sse: Optional[SseCustomerKey] = None,
    source_sse: Optional[SseCustomerKey] = None,
    storage_class: Optional[str] = None
) -> ObjectWriteResult
```

Parameters:

- **`bucket_name`** (str): The name of the destination bucket where the object will be copied.
- **`object_name`** (str): The name of the new object in the destination bucket.
- **`copy_source`** (`CopySource`): The source object (bucket and object name) to copy from. This includes the source bucket, object name, and optional version ID for versioned buckets.
- **`metadata`** (Optional[dict], default=None): Optional metadata to apply to the copied object. If provided, it replaces the source object's metadata.
- **`sse`** (Optional[SseCustomerKey], default=None): Server-side encryption (SSE) configuration for the destination object. Used when encrypting the object in the destination bucket.
- **`source_sse`** (Optional[SseCustomerKey], default=None): SSE configuration for the source object if it is encrypted.
- **`storage_class`** (Optional[str], default=None): The storage class to apply to the copied object (e.g., standard, reduced redundancy).

Return Value:

- **Returns** an `ObjectWriteResult`, which contains metadata about the newly copied object, such as the version ID (if versioning is enabled) and the ETag (a unique identifier for the object).

Example Code

Simple Object Copy Within the Same Bucket:

```
# Define the copy source (bucket and object)
copy_source = CopySource("my-bucket", "source-object")
# Copy object within the same bucket
result = client.copy_object("my-bucket", "destination-object", copy_source)
```

Copying an Object Between Different Buckets:

```
# Define the source bucket and object
copy_source = CopySource("source-bucket", "source-object")
# New metadata to apply to the copied object
metadata = {"Content-Type": "application/json"}
# Copy the object and update its metadata
result = minioClient.copy_object("destination-bucket", "destination-object",
copy_source, metadata=metadata)
```

Copying an Object with Encryption:

```
from minio.sse import SseCustomerKey

try:
    # Define the copy source
    copy_source = CopySource("source-bucket", "source-object")

    # Define the server-side encryption key
    sse = SseCustomerKey(b"32byteslongsecretkeymustprovided")

    # Copy the object with encryption
    result = minioClient.copy_object("destination-bucket", "encrypted-object",
copy_source, sse=sse)
    print("Object copied with encryption and ETag:", result.etag)
except Exception as err:
    print(f"Error: {err}")
```

composite_object

The compose_object function in the MinIO SDK is used to combine multiple source objects into a single destination object. This is particularly useful when you need to concatenate or merge several smaller objects into a larger one. It can also help when working with multipart uploads, where multiple object parts need to be merged into a single object.

```python
def compose_object(
    bucket_name: str,
    object_name: str,
    sources: List[ComposeSource],
    metadata: Optional[dict] = None,
    sse: Optional[SseCustomerKey] = None,
    storage_class: Optional[str] = None
) -> ObjectWriteResult
```

Parameters:

- **`bucket_name`** (str): The name of the destination bucket where the composed object will be stored.
- **`object_name`** (str): The name of the new composed object in the destination bucket.
- **`sources`** (List[`ComposeSource`]): A list of `ComposeSource` objects representing the source objects to be composed. Each source object can be from the same or different buckets.
- **`metadata`** (Optional[dict], default=None): Optional metadata to apply to the composed object. If provided, it replaces the metadata of the source objects.
- **`sse`** (Optional[SseCustomerKey], default=None): Server-side encryption (SSE) configuration for the composed object.
- **`storage_class`** (Optional[str], default=None): The storage class to apply to the composed object (e.g., standard, reduced redundancy).

Return Value:

- **Returns** an `ObjectWriteResult` containing metadata about the composed object, such as its version ID (if versioning is enabled) and the ETag (a unique identifier for the object).

Example

Composing Multiple Objects into One:

```
from minio import Minio
from minio.commonconfig import ComposeSource

# Initialize the MinIO client
minioClient = Minio(
    "play.min.io",
    access_key="your-access-key",
    secret_key="your-secret-key",
    secure=True
)

try:
    # Define the source objects for composition
    sources = [
        ComposeSource("my-bucket", "part-1"),
        ComposeSource("my-bucket", "part-2"),
        ComposeSource("my-bucket", "part-3")
    ]

    # Compose multiple objects into one
    result = minioClient.compose_object("my-bucket", "final-object", sources)
    print("Object composed successfully with ETag:", result.etag)
except Exception as err:
    print(f"Error: {err}")
```

Composing Objects Across Buckets:

```
try:
    # Define the source objects from different buckets
    sources = [
        ComposeSource("source-bucket-1", "part-1"),
        ComposeSource("source-bucket-2", "part-2")
    ]

    # Compose objects from different buckets into one
    result = minioClient.compose_object("destination-bucket",
"final-object", sources)
    print("Object composed successfully with ETag:", result.etag)
except Exception as err:
    print(f"Error: {err}")
```

Composing Objects with Metadata Update:

```
try:
```

```
    # Define the source objects for composition
    sources = [
        ComposeSource("my-bucket", "part-1"),
        ComposeSource("my-bucket", "part-2")
    ]

    # New metadata to apply to the composed object
    metadata = {"Content-Type": "application/json"}

    # Compose multiple objects with metadata update
    result = minioClient.compose_object("my-bucket", "final-object", sources,
metadata=metadata)
    print("Object composed with new metadata and ETag:", result.etag)
except Exception as err:
    print(f"Error: {err}")
```

Composing Objects with Encryption:

```
from minio.sse import SseCustomerKey

try:
    # Define the source objects for composition
    sources = [
        ComposeSource("my-bucket", "part-1"),
        ComposeSource("my-bucket", "part-2")
    ]

    # Define server-side encryption for the composed object
    sse = SseCustomerKey(b"32byteslongsecretkeymustprovided")

    # Compose multiple objects with encryption
    result = minioClient.compose_object("my-bucket", "encrypted-object", sources,
sse=sse)
    print("Object composed with encryption and ETag:", result.etag)
except Exception as err:
    print(f"Error: {err}")
```

Bucket API's

MinIO's bucket APIs provide a set of operations for managing and interacting with buckets, which are containers for storing objects. These APIs are designed to be S3-compatible, allowing you to use them similarly to how you would with Amazon S3, while benefiting from MinIO's high-performance, distributed object storage capabilities.

`make_bucket`

Purpose: Creates a new bucket to store objects.

```
Minio.make_bucket(
    bucket_name: str,
    location: Optional[str] = None
) -> None
```

Parameters:

- `bucket_name`: The name of the bucket you want to create.
- `location`: (Optional) The region where the bucket will be created.

Returns: None

Example:

```
from minio import Minio

# Initialize a MinIO client
client = Minio(
    "play.min.io",
    access_key="YOUR-ACCESS-KEY",
    secret_key="YOUR-SECRET-KEY",
    secure=True
)

# Create a new bucket
client.make_bucket("my-new-bucket", location="us-east-1")
print("Bucket created successfully")
```

list_buckets

- **Purpose**: Lists all the buckets available in your MinIO instance.

```
Minio.list_buckets() -> List[Bucket]
```

Parameters: None

Returns: A list of `Bucket` objects, each containing the bucket name and creation date.

Example:

```python
from minio import Minio

# Initialize a MinIO client
client = Minio(
    "play.min.io",
    access_key="YOUR-ACCESS-KEY",
    secret_key="YOUR-SECRET-KEY",
    secure=True
)

# List all buckets
buckets = client.list_buckets()
for bucket in buckets:
    print(bucket.name, bucket.creation_date)
```

remove_bucket

- **Purpose**: Deletes an empty bucket.

```
Minio.remove_bucket(
    bucket_name: str
) -> None
```

Parameters:

- `bucket_name`: The name of the bucket you want to delete.

Returns: None

Example

```
from minio import Minio

# Initialize a MinIO client
client = Minio(
    "play.min.io",
    access_key="YOUR-ACCESS-KEY",
    secret_key="YOUR-SECRET-KEY",
    secure=True
)

# Delete a bucket
client.remove_bucket("my-old-bucket")
print("Bucket deleted successfully")
```

bucket_exists

- **Purpose**: Checks whether a specific bucket exists.

```
Minio.bucket_exists(
    bucket_name: str
) -> bool
```

Parameters:

- `bucket_name`: The name of the bucket you want to check.

Returns : True if the bucket exists, otherwise False .

Example

```python
from minio import Minio

# Initialize a MinIO client
client = Minio(
    "play.min.io",
    access_key="YOUR-ACCESS-KEY",
    secret_key="YOUR-SECRET-KEY",
    secure=True
)

# Check if a bucket exists
exists = client.bucket_exists("my-bucket")
if exists:
    print("Bucket exists!")
else:
    print("Bucket does not exist.")
```

list_objects

- **Purpose**: Lists the objects stored in a specific bucket.

```
Minio.list_objects(
    bucket_name: str,
    prefix: Optional[str] = None,
    recursive: bool = False
) -> Iterable[Object]
```

- **Parameters**:
    - `bucket_name`: The name of the bucket.
    - `prefix`: (Optional) Filter objects by prefix.
    - `recursive`: (Optional) List all objects recursively (default is `False`).

Returns: An iterable of `Object` instances.

Example:

```
from minio import Minio

# Initialize a MinIO client
client = Minio(
    "play.min.io",
    access_key="YOUR-ACCESS-KEY",
    secret_key="YOUR-SECRET-KEY",
    secure=True
)

# List objects in a bucket
objects = client.list_objects("my-bucket", prefix="photos/",
recursive=True)
for obj in objects:
    print(obj.object_name, obj.size, obj.last_modified)
```

Bucket Policy APi's

`set_bucket_policy`

- **Definition**: `set_bucket_policy(bucket_name, policy)`
- **Purpose**: Applies a policy to a bucket to control access permissions.
- **Parameters**:
    - `bucket_name`: The name of the bucket.
    - `policy`: The policy in JSON format to be applied to the bucket.

```
policy = {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {"AWS": "*"},
            "Action": ["s3:GetObject"],
            "Resource": ["arn:aws:s3:::my-bucket/*"]
        }
    ]
}
client.set_bucket_policy("my-bucket", policy)
```

`get_bucket_policy`

- **Definition**: `get_bucket_policy(bucket_name)`
- **Purpose**: Retrieves the current policy applied to a bucket.
- **Parameters**:
    - `bucket_name`: The name of the bucket.

Example:

```
policy = client.get_bucket_policy("my-bucket")
```

```
print(policy)
```

`delete_bucket_policy`

- **To remove all applied policy from a bucket**

Example:

```
minioClient.delete_bucket_policy("logs-bucket")
```

Bucket Versioning API's

`enable_bucket_versioning`

- **Definition**: `enable_bucket_versioning(bucket_name)`
- **Purpose**: Enables versioning on a bucket, allowing multiple versions of the same object to be stored.
- **Parameters**:
    - `bucket_name`: The name of the bucket.

Example:

```
client.enable_bucket_versioning("my-bucket")
```

`suspend_bucket_versioning`

- **Definition**: `suspend_bucket_versioning(bucket_name)`
- **Purpose**: Suspends versioning on a bucket. Previously existing versions are preserved, but new versions won't be created.
- **Parameters**:
    - `bucket_name`: The name of the bucket.

Example:

```
client.suspend_bucket_versioning("my-bucket")
```

Bucket Lifecycle API's

`set_bucket_lifecycle`

- **Definition**: `set_bucket_lifecycle(bucket_name, lifecycle)`
- **Purpose**: Defines rules for automatically transitioning or expiring objects in a bucket based on age or other criteria.
- **Parameters**:
    - `bucket_name`: The name of the bucket.
    - `lifecycle`: The lifecycle configuration in JSON format.

Example:

In below code, S3_STANDARD is the tier that we created in your minio

```
from minio import Minio
from minio.lifecycleconfig import LifecycleConfig, Rule, Expiration, Transition
from minio.commonconfig import ENABLED, Filter


# Initialize the MinIO client
minioClient = Minio(
    "127.0.0.1:9000",
    access_key="hgWVh2MUy0v7i2Hzq2NR",
    secret_key="hfmaMc57uRhEGy0d70XlPbqzeMdRnFyxmVSYMMdZ",
    secure=False
)

# Create LifecycleConfig rules
config = LifecycleConfig(
    [
        Rule(
            ENABLED,
            rule_filter=Filter(prefix="logs/"),
            rule_id="rule1",
            transition=Transition(days=30, storage_class="S3_STANDARD"),
        ),
        Rule(
            ENABLED,
            rule_filter=Filter(prefix="tmp/"),
            rule_id="rule2",
            expiration=Expiration(days=365),
        ),
```

```
    ],
)


try:
    # Set the bucket lifecycle configuration
    minioClient.set_bucket_lifecycle("logs-bucket", config)
    print("Lifecycle policy applied successfully.")

except Exception as err:
    print(f"Error: {err}")
```

get_bucket_lifecycle

- **Definition**: `get_bucket_lifecycle(bucket_name)`
- **Purpose**: Retrieves the lifecycle policy applied to a bucket.
- **Parameters**:
    - `bucket_name`: The name of the bucket.

Example:

```
lifecycle = client.get_bucket_lifecycle("my-bucket")
# Iterate over each rule in the lifecycle configuration
   for rule in lifecycle.rules:
```

```
        print(f"Rule ID: {rule.rule_id}")
        print(f"Status: {rule.status}")

        # Print expiration details if they exist
        if rule.expiration:
            print(f"Expiration: {rule.expiration.days} days")

        # Print transition details if they exist
        if rule.transition:
            print(f"Transition after: {rule.transition.days} days")
            print(f"Transition to storage class:
{rule.transition.storage_class}")

        # Print filter details if they exist
        if rule.rule_filter:
            print(f"Filter Prefix: {rule.rule_filter.prefix}")
```

`delete_bucket_lifecycle`

- **Definition**: `delete_bucket_lifecycle(bucket_name)`
- **Purpose**: Removes the lifecycle policy applied to a bucket.
- **Parameters**:
    - `bucket_name`: The name of the bucket.

Example:

```
client.delete_bucket_lifecycle("my-bucket")
```

Bucket Notification API

`set_bucket_notification`

- **Definition**: `set_bucket_notification(bucket_name, notification)`

- **Purpose**: Configures notifications for specific events (like object creation, deletion, etc.) in a bucket.
- **Parameters**:
    - `bucket_name`: The name of the bucket.
    - `notification`: The notification configuration in JSON format.

Example:

```
notification = {
    "QueueConfigurations": [
        {
            "Id": "NewObjectCreated",
            "QueueArn": "arn:aws:sqs:us-east-1:123456789012:my-queue",
            "Events": ["s3:ObjectCreated:*"]
        }
    ]
}
client.set_bucket_notification("my-bucket", notification)
```

`get_bucket_notification`

- **Definition**: `get_bucket_notification(bucket_name)`
- **Purpose**: Retrieves the current notification configuration for a bucket.
- **Parameters**:
    - `bucket_name`: The name of the bucket.

Example:

```
notification = client.get_bucket_notification("my-bucket")
print(notification)
```

`delete_bucket_notification`

- **Definition**: `delete_bucket_notification(bucket_name)`
- **Purpose**: Removes all notifications configured for a bucket.

- **Parameters**:
    - `bucket_name`: The name of the bucket.

Example:

```
client.delete_bucket_notification("my-bucket")
```

Bucket Tags API's

`set_bucket_tags`

- **Definition**: `set_bucket_tags(bucket_name, tags)`
- **Purpose**: Adds or updates tags on a bucket, which can be used for categorization, management, or billing purposes.
- **Parameters**:
    - `bucket_name`: The name of the bucket.
    - `tags`: A dictionary of tags (key-value pairs) to apply to the bucket.

Example:

```
from minio.commonconfig import Tags

tags = Tags.new_bucket_tags()
tags["Project"] = "Project One"
tags["User"] = "jsmith"
minioClient.set_bucket_tags("logs-bucket", tags)
```

`get_bucket_tags`

- **Definition**: `get_bucket_tags(bucket_name)`

- **Purpose**: Retrieves the tags associated with a bucket.
- **Parameters**:
    - `bucket_name`: The name of the bucket.

Example:

```
tags = client.get_bucket_tags("my-bucket")
print(tags)
```

`delete_bucket_tags`

- **Definition**: `delete_bucket_tags(bucket_name)`
- **Purpose**: Removes all tags from a bucket.
- **Parameters**:
    - `bucket_name`: The name of the bucket.
- **Example**:

```
client.delete_bucket_tags("my-bucket")
```

Bucket Locking and Retention

`set_object_lock_config`

The `set_object_lock_config` function in the MinIO SDK is used to **set or update the object lock configuration** for a bucket. Object locking enables you to protect objects from being deleted or overwritten for a specified period or indefinitely (using legal hold or retention policies). This function configures the bucket to allow these features.

```
def set_object_lock_config(
    bucket_name: str,
    lock_config: ObjectLockConfig
) -> None
```

Parameters:

- **`bucket_name`** (str): The name of the bucket for which you want to configure object lock settings.
- **`lock_config`** (`ObjectLockConfig`): An `ObjectLockConfig` object that contains the configuration for object lock. This includes the mode (e.g., `GOVERNANCE` or `COMPLIANCE`) and the default retention period for the objects.

Usage Scenario:

- **Object Retention**: To enforce write-once-read-many (WORM) compliance by preventing objects from being modified or deleted for a specified time.
- **Governance Mode**: Allows deletion of locked objects with special permissions.
- **Compliance Mode**: Prevents deletion or modification of locked objects for the retention period.
- **Legal Hold**: Ensures objects are protected indefinitely, overriding retention policies.

Example

Set Object Lock Configuration for a Bucket:

```
from minio import Minio
from minio.commonconfig import GOVERNANCE
from minio.objectlockconfig import DAYS, ObjectLockConfig
# Initialize the MinIO client
minioClient = Minio(
    "127.0.0.1:9000",
    access_key="hgWVh2MUy0v7i2Hzq2NR",
    secret_key="hfmaMc57uRhEGy0d70XlPbqzeMdRnFyxmVSYMMdZ",
    secure=False
)
try:
    # Create a new bucket with object lock enabled
    minioClient.make_bucket("new-governance-bucket", object_lock=True)
    print("Bucket created successfully with object lock enabled.")

    # Set up object lock configuration for the bucket (Governance mode, 15 days)
    config = ObjectLockConfig(GOVERNANCE, 15, DAYS)

    # Apply object lock configuration to the bucket
    minioClient.set_object_lock_config("new-governance-bucket", config)
    print("Object lock configuration set successfully.")

except Exception as err:
    print(f"Error: {err}")
```

Set Object Lock in Compliance Mode:

```
from minio import Minio
from minio.commonconfig import COMPLIANCE
from minio.objectlockconfig import DAYS, ObjectLockConfig
# Initialize the MinIO client
minioClient = Minio(
    "127.0.0.1:9000",
    access_key="hgWVh2MUy0v7i2Hzq2NR",
    secret_key="hfmaMc57uRhEGy0d70XlPbqzeMdRnFyxmVSYMMdZ",
    secure=False
)
try:
    # Create a new bucket with object lock enabled
    minioClient.make_bucket("new-compliance-bucket", object_lock=True)
    print("Bucket created successfully with object lock enabled.")

    # Set up object lock configuration for the bucket (Governance mode, 15 days)
    config = ObjectLockConfig(COMPLIANCE, 15, DAYS)

    # Apply object lock configuration to the bucket
    minioClient.set_object_lock_config("new-compliance-bucket", config)
    print("Object lock configuration set successfully.")

except Exception as err:
    print(f"Error: {err}")
```

`get_object_lock_config`

The `get_object_lock_config` function in the MinIO SDK is used to **retrieve the object lock configuration** of a bucket. This allows you to check the current object lock settings, including the default retention mode and retention period, to ensure that the bucket is protected according to your requirements.

```
def get_object_lock_config(
    bucket_name: str
) -> ObjectLockConfig
```

Parameters:

- **`bucket_name`** (str): The name of the bucket whose object lock configuration you want to retrieve.

Return Value:

- **Returns** an `ObjectLockConfig` object, which contains details about the bucket's object lock configuration, such as the lock mode (`GOVERNANCE` or `COMPLIANCE`) and the default retention period.

Example

```
from minio import Minio
from minio.commonconfig import ObjectLockConfig

# Initialize the MinIO client
minioClient = Minio(
    "play.min.io",
    access_key="your-access-key",
    secret_key="your-secret-key",
    secure=True
)

try:
    # Get object lock configuration for the bucket
    config= minioClient.get_object_lock_config("my-bucket")

    # Check if object lock is configured
    if config:
         print(f"mode: {config._mode} duration: {config._duration}")
```

```
    else:
        print("No object lock configuration found for the bucket.")
except Exception as err:
    print(f"Error: {err}")
```

Check if Object Lock is Set in Compliance Mode:

```
from minio.commonconfig import COMPLIANCE

try:
    # Get object lock configuration
    lock_config = minioClient.get_object_lock_config("compliance-bucket")

    # Verify if compliance mode is enabled
    if lock_config and lock_config.mode == COMPLIANCE:
        print("Compliance mode is enabled for this bucket.")
    else:
        print("Compliance mode is not enabled.")
except Exception as err:
    print(f"Error: {err}")
```

Key Considerations:

- **Modes**:
    - **GOVERNANCE**: This mode allows objects to be protected from deletion but can be overridden by authorized users with special permissions.
    - **COMPLIANCE**: This mode strictly enforces retention policies, preventing any deletion or modification of objects for the specified retention period.
- **Retention Period**: The function returns the default retention period (in days or years) that applies to objects in the bucket.
- **No Object Lock Config**: If the bucket doesn't have an object lock configuration, the function returns None, indicating that object lock is not enabled for the bucket.
- **Pre-requisite**: Object lock must be enabled for the bucket at creation time to retrieve any configuration.

`delete_object_lock_config`

Remove Object Lock Configuration:

```
try:
    # Remove object lock configuration by passing None
    minioClient.delete_object_lock_config("new-compliance-bucket")
    print("Object lock configuration removed successfully.")
except Exception as err:
    print(f"Error: {err}")
```

Object Retention

`set_object_retention`

The `set_object_retention` function in the MinIO SDK is used to **apply or update a retention policy for a specific object** within a bucket. This policy prevents the object from being deleted or overwritten for a defined retention period, ensuring compliance with governance or regulatory requirements. The retention mode can either be set to `GOVERNANCE` or `COMPLIANCE`, and the function can be applied to versioned objects.

```
Minio.set_object_retention(bucket_name, object_name, config,
version_id=None)
```

Parameters:

- **`bucket_name`** (str): The name of the bucket where the object resides.
- **`object_name`** (str): The name of the object to which the retention policy will be applied.
- **`retention`** (`ObjectLockRetention`): The retention configuration object, specifying the mode (`GOVERNANCE` or `COMPLIANCE`) and the duration (in days or years).
- **`version_id`** (Optional[str], default=None): The version ID of the object (if versioning is enabled). If not provided, the latest version is used.

Usage Scenario:

- **Enforce Retention**: Apply a retention policy to prevent deletion or modification of objects for a specific duration.
- **Governance Mode**: Allows for deletion/modification with special permissions, ensuring that only authorized users can override the retention.
- **Compliance Mode**: Ensures that objects cannot be deleted or modified until the retention period expires, strictly enforcing compliance.
- **Versioning**: Can be applied to specific versions of objects in versioned buckets.

Apply Retention Policy to an Object:

```
from minio import Minio
from minio.commonconfig import GOVERNANCE
from minio.retention import Retention
from datetime import datetime, timedelta



# Apply object lock configuration to the object
config = Retention(GOVERNANCE, datetime.utcnow() + timedelta(days=10))
minioClient.set_object_retention("new-locked-bucket", "ps.png", config)
```

Apply Retention with Compliance Mode:

```
from minio import Minio
from minio.commonconfig import COMPLIANCE
from minio.retention import Retention
from datetime import datetime, timedelta



# Apply object lock configuration to the object
config = Retention(COMPLIANCE, datetime.utcnow() + timedelta(days=10))
minioClient.set_object_retention("new-locked-bucket", "ps.png", config)
```

Apply Retention to a Specific Version of an Object:

```
from minio import Minio
from minio.commonconfig import COMPLIANCE
from minio.retention import Retention
from datetime import datetime, timedelta
```

```
# Apply object lock configuration to the object
config = Retention(COMPLIANCE, datetime.utcnow() + timedelta(days=10))
minioClient.set_object_retention("new-locked-bucket", "ps.png", config,
version_id="12344")
```

`get_object_retention`

The `get_object_retention` function in the MinIO SDK is used to **retrieve the retention policy** for a specific object in a bucket. This function allows you to check the current retention settings (such as retention mode and retention duration) applied to an object. It is useful for verifying that an object is protected from deletion or modification for a specified period.

```
def get_object_retention(
    bucket_name: str,
    object_name: str,
    version_id: Optional[str] = None
) -> ObjectLockRetention
```

Parameters:

- **`bucket_name`** (str): The name of the bucket where the object resides.
- **`object_name`** (str): The name of the object whose retention policy you want to retrieve.
- **`version_id`** (Optional[str], default=None): The version ID of the object (if versioning is enabled). If not provided, the latest version of the object is checked for its retention policy.

Return Value:

- **Returns** an `ObjectLockRetention` object, which contains details about the object's retention policy, such as the retention mode (`GOVERNANCE` or `COMPLIANCE`) and the retention duration (in days or years).

Retrieve Retention Policy for an Object:

```
 config = minioClient.get_object_retention("new-locked-bucket", "ps.png")
 print(config._mode,config._retain_until_date)
```

Retrieve Retention Policy for a Specific Version:

```
try:
    # Get retention policy for a specific version of the object
    retention = minioClient.get_object_retention("my-bucket", "my-object",
version_id="1234abcd")

    if retention:
        print(f"Retention mode: {retention.mode}")
        print(f"Retention duration (days): {retention.retention_duration_days}")
    else:
        print("No retention policy found for this version of the object.")
except Exception as err:
    print(f"Error: {err}")
```

Object legal hold

`enable_object_legal_hold`

The `put_object_legal_hold` function in the MinIO SDK is used to **apply or update a legal hold** on a specific object. A legal hold overrides any retention policy and prevents the object from being deleted or modified indefinitely until the legal hold is explicitly removed. Legal holds are used in scenarios where data needs to be preserved for legal reasons, irrespective of any set retention periods.

```
def enable_object_legal_hold(
            self,
            bucket_name: str,
            object_name: str,
            version_id: str | None = None,
    ):
```

Parameters:

- **`bucket_name`** (str): The name of the bucket where the object resides.
- **`object_name`** (str): The name of the object on which you want to apply or update the legal hold.
- **`version_id`** (Optional[str], default=None): The version ID of the object (if versioning is enabled). If not provided, the latest version of the object will be affected.

Enable Legal Hold on an Object:

```
# Enable legal hold on the object
minioClient.put_object_legal_hold("my-bucket", "my-object", enable_legal_hold=True)
```

Enable Legal Hold on a Specific Version of an Object:

```
try:
    # Enable legal hold on a specific version of the object
    minioClient.put_object_legal_hold("my-bucket", "my-object",
enable_legal_hold=True, version_id="1234abcd")
    print("Legal hold enabled on specific version successfully.")
except Exception as err:
    print(f"Error: {err}")
```

`is_object_legal_hold_enabled`

The `is_object_legal_hold_enabled` function in the MinIO SDK is used to **retrieve the legal hold status** of a specific object. This function allows you to check whether a legal hold is applied to the object, preventing its deletion or modification.

```
def is_object_legal_hold_enabled(
            self,
            bucket_name: str,
            object_name: str,
            version_id: str | None = None,
    ) -> bool:
```

Parameters:

- **`bucket_name`** (str): The name of the bucket where the object resides.
- **`object_name`** (str): The name of the object whose legal hold status you want to retrieve.

- **`version_id`** (Optional[str], default=None): The version ID of the object (if versioning is enabled). If not provided, the legal hold status of the latest version is retrieved.

Return Value:

- **Returns** a `boolean` object that indicates whether the legal hold is there or not.

Retrieve Legal Hold Status for an Object:

```
if minioClient.is_object_legal_hold_enabled("new-locked-bucket", "logs_ps.png"):
        print("legal hold is enabled")
    else:
        print("legal hold is not enabled")
```

Retrieve Legal Hold Status for a Specific Version of an Object:

```
if minioClient.is_object_legal_hold_enabled("new-locked-bucket",
"logs_ps.png","version_id"):
        print("legal hold is enabled")
    else:
        print("legal hold is not enabled")
```

`disable_object_legal_hold`

```
 def disable_object_legal_hold(
            self,
            bucket_name: str,
            object_name: str,
            version_id: str | None = None,
    ):
```

Example:

```
minioClient.disable_object_legal_hold("new-locked-bucket", "logs_ps.png")
```

Presigned Operations

`get_presigned_url`

The `get_presigned_url` function in the MinIO SDK is used to **generate a presigned URL** for accessing an object in a bucket. This URL allows clients to access the object without requiring authentication or direct access to the MinIO server. Presigned URLs are useful for granting temporary access to an object, and they can be used for uploading, downloading, or modifying objects based on the method specified.

```python
def get_presigned_url(
    method: str,
    bucket_name: str,
    object_name: str,
    expires: Optional[timedelta] = None,
    response_headers: Optional[dict] = None,
    request_date: Optional[datetime] = None,
    version_id: Optional[str] = None,
    extra_query_params: Optional[dict] = None
) -> str
```

Parameters:

- **`method`** (str): The HTTP method to use for the presigned URL (e.g., `GET`, `PUT`, `DELETE`, etc.). Determines the type of access the presigned URL provides (e.g., read, write, delete).
- **`bucket_name`** (str): The name of the bucket where the object resides.
- **`object_name`** (str): The name of the object for which you want to generate the presigned URL.
- **`expires`** (Optional[`timedelta`], default=None): The duration (as a `timedelta` object) for which the presigned URL will remain valid. If not provided, a default expiration time is used (usually 7 days).
- **`response_headers`** (Optional[dict], default=None): Custom response headers to be included in the presigned URL (useful for setting `Content-Type`, etc.).
- **`request_date`** (Optional[`datetime`], default=None): The specific date and time for the request, mainly used for advanced use cases.
- **`version_id`** (Optional[str], default=None): The version ID of the object (if versioning is enabled). If not provided, the presigned URL will be generated for the latest version.

- **`extra_query_params`** (Optional[dict], default=None): Additional query parameters to include in the presigned URL.

Return Value:

- **Returns** a presigned URL (as a `str`) that provides temporary access to the specified object.

Usage Scenario:

- **Granting Temporary Access**: Generate a presigned URL to allow external users or clients temporary access to an object without needing MinIO credentials.
- **Object Upload/Download**: Use presigned URLs to securely share object upload or download links with external users.
- **Time-limited Access**: Define the duration for which the URL will be valid (e.g., for secure access that expires after a given time).

Generate a Presigned URL for Downloading an Object (GET Request):

```
from minio import Minio
from datetime import timedelta

# Initialize the MinIO client
minioClient = Minio(
    "play.min.io",
    access_key="your-access-key",
    secret_key="your-secret-key",
    secure=True
)

try:
    # Generate a presigned URL for downloading an object, valid for 1 day
    presigned_url = minioClient.get_presigned_url(
        method="GET",
        bucket_name="my-bucket",
        object_name="my-object",
        expires=timedelta(days=1)
    )

    print("Presigned URL:", presigned_url)
except Exception as err:
    print(f"Error: {err}")
```

Generate a Presigned URL for Uploading an Object (PUT Request):

```
try:
    # Generate a presigned URL for uploading an object, valid for 1 hour
    presigned_url = minioClient.get_presigned_url(
        method="PUT",
        bucket_name="my-bucket",
        object_name="my-upload-object",
        expires=timedelta(hours=1)
    )

    print("Presigned URL for upload:", presigned_url)
except Exception as err:
    print(f"Error: {err}")
```

Python code to upload a file using presigned url.

```
import requests

# Presigned URL (from the first step)
presigned_url =
"https://play.min.io/my-bucket/my-upload-object?X-Amz-Algorithm=..."

# Upload the file to the presigned URL
with open("/path/to/file.txt", "rb") as file_data:
    response = requests.put(presigned_url, data=file_data)

if response.status_code == 200:
    print("Upload successful")
else:
    print(f"Upload failed: {response.status_code}")
```

Generate a Presigned URL with Custom Response Headers and for specific version id:

```
try:
    # Generate a presigned URL with custom response headers (e.g., content type)
    presigned_url = minioClient.get_presigned_url(
        method="GET",
        bucket_name="my-bucket",
        object_name="my-object",
        response_headers={"response-content-type": "application/pdf"},
        version_id="1234abcd"
        expires=timedelta(minutes=30)
    )
```

`presigned_get_object`

Purpose: Specialized for **downloading (GET request)** objects from a bucket.

Use Case: It is specifically designed to generate a presigned URL for **retrieving an object** from MinIO using the HTTP `GET` method. It is simpler to use for download scenarios.

Method Parameter: No need to pass the HTTP method (`GET` is implicitly used). It is solely focused on generating a presigned URL for downloading objects.

```python
def presigned_get_object(
    bucket_name: str,
    object_name: str,
    expires: Optional[timedelta] = None,
    response_headers: Optional[dict] = None,
    request_date: Optional[datetime] = None,
    version_id: Optional[str] = None,
    extra_query_params: Optional[dict] = None
) -> str
```

Parameters:

- **`bucket_name`** (str): The name of the bucket where the object resides.
- **`object_name`** (str): The name of the object for which you want to generate the presigned URL.
- **`expires`** (Optional[`timedelta`], default=None): The duration (as a `timedelta` object) for which the presigned URL will remain valid. If not provided, a default expiration time is used (usually 7 days).
- **`response_headers`** (Optional[dict], default=None): Custom response headers to include in the presigned URL (useful for setting `Content-Type`, `Content-Disposition`, etc.).
- **`request_date`** (Optional[`datetime`], default=None): The specific date and time for the request, mainly used for advanced use cases.
- **`version_id`** (Optional[str], default=None): The version ID of the object (if versioning is enabled). If not provided, the presigned URL will be generated for the latest version.
- **`extra_query_params`** (Optional[dict], default=None): Additional query parameters to include in the presigned URL.

Return Value:

- **Returns** a presigned URL (as a `str`) that provides temporary access to the specified object for downloading.

Generate a Presigned URL for Downloading an Object:

```
# Generate a presigned URL for downloading an object, valid for 1 day
presigned_url = minioClient.presigned_get_object(
        bucket_name="my-bucket",
        object_name="my-object",
        expires=timedelta(days=1)
```

Generate a Presigned URL for a Specific Version of an Object:

```
# Generate a presigned URL for downloading an object, valid for 1 day
presigned_url = minioClient.presigned_get_object(
        bucket_name="my-bucket",
        object_name="my-object",
        version_id="1234abcd"
        expires=timedelta(days=1)
```

`presigned_put_object`

Purpose: Specialized for **uploading(PUT request)** objects to a bucket.

The presigned_put_object function in the MinIO SDK is used to generate a presigned URL for uploading (PUT) an object to a bucket. This URL allows clients to upload an object to the specified bucket without needing direct authentication or access to the MinIO server. The presigned URL is valid for a limited time, providing temporary access to upload the object.

```
def presigned_put_object(
    bucket_name: str,
    object_name: str,
    expires: Optional[timedelta] = None
) -> str
```

Parameters:

- **`bucket_name`** (str): The name of the bucket where the object will be uploaded.
- **`object_name`** (str): The name of the object to be uploaded via the presigned URL.

- **`expires`** (Optional[`timedelta`], default=None): The duration (as a `timedelta` object) for which the presigned URL will remain valid. If not provided, a default expiration time is used (usually 7 days).

Return Value:

- **Returns** a presigned URL (as a `str`) that provides temporary access to upload the specified object using an HTTP PUT request.

Example

```
# Generate a presigned URL for uploading an object, valid for 1 hour
presigned_url = minioClient.presigned_put_object(
        bucket_name="my-bucket",
        object_name="my-upload-object",
        expires=timedelta(hours=1)
    )
```

Multipart Upload Operations

Usage Scenario:

- **Large File Uploads**: Use multipart upload to upload large files efficiently by splitting them into smaller parts.
- **Parallel Uploads**: The multipart upload allows uploading parts in parallel, reducing the total time for large file uploads.
- **Upload Recovery**: In case of a failure, only the failed part needs to be re-uploaded, rather than restarting the entire upload process.

Multipart Upload Process:

- **Initiate**: The multipart upload process starts with `initiate_multipart_upload`, which returns an `upload_id`.
- **Upload Parts**: Use the `upload_id` to upload individual parts with the `upload_part` method.
- **Complete**: After all parts are uploaded, the upload must be completed using `complete_multipart_upload`.

- **Abort**: If necessary, the multipart upload can be aborted using `abort_multipart_upload`.

`initiate_multipart_upload`

This function initiates the multipart upload and returns an `upload_id`, which is required for uploading individual parts and completing the multipart upload.

```
def initiate_multipart_upload(
    bucket_name: str,
    object_name: str,
    metadata: Optional[dict] = None,
    sse: Optional[SseCustomerKey] = None
) -> str
```

Initiate a Multipart Upload: In this example, the multipart upload is initiated for an object, and the returned `upload_id` is stored for uploading parts.

```
from minio import Minio

# Initialize the MinIO client
minioClient = Minio(
    "play.min.io",
    access_key="your-access-key",
    secret_key="your-secret-key",
    secure=True
)

try:
    # Initiate multipart upload for 'my-large-object'
    upload_id = minioClient.initiate_multipart_upload("my-bucket",
"my-large-object")

    print("Multipart upload initiated successfully.")
    print("Upload ID:", upload_id)
except Exception as err:
    print(f"Error: {err}")
```

upload_part

Upload Parts Using the Multipart Upload ID: After initiating the multipart upload, the returned `upload_id` is used to upload parts sequentially or in parallel.

```
import os

try:
    # Path to the large file
    file_path = "/path/to/large-file.bin"
    file_size = os.path.getsize(file_path)

    # Open the file and upload it in parts
    with open(file_path, "rb") as file_data:
        part_number = 1
        part_size = 5 * 1024 * 1024  # 5MB parts
        uploaded_parts = []

        while file_size > 0:
            data = file_data.read(part_size)
            etag = minioClient.upload_part("my-bucket", "my-large-object",
upload_id, part_number, data)
            uploaded_parts.append({"PartNumber": part_number, "ETag": etag})

            part_number += 1
            file_size -= part_size

        print("All parts uploaded successfully.")
        print("Uploaded Parts:", uploaded_parts)
except Exception as err:
    print(f"Error: {err}")
```

complete_multipart_upload

Complete the Multipart Upload: Once all parts are uploaded, the multipart upload process must be completed using the `upload_id` and the list of uploaded parts.

```
try:
    # Complete the multipart upload
    minioClient.complete_multipart_upload("my-bucket", "my-large-object",
upload_id, uploaded_parts)

    print("Multipart upload completed successfully.")
except Exception as err:
    print(f"Error: {err}")
```

abort_multipart_upload

The `abort_multipart_upload` function in the MinIO SDK is used to **abort an ongoing multipart upload**. If an upload is interrupted or incomplete, this function ensures that the multipart upload is terminated and all uploaded parts are deleted. This is useful for cleaning up failed or abandoned uploads and avoiding incomplete or orphaned uploads that could consume storage

```
def abort_multipart_upload(
    bucket_name: str,
    object_name: str,
    upload_id: str
) -> None
```

```
from minio import Minio

# Initialize the MinIO client
minioClient = Minio(
    "play.min.io",
    access_key="your-access-key",
    secret_key="your-secret-key",
    secure=True
)

try:
    # Step 1: Initiate multipart upload
    upload_id = minioClient.initiate_multipart_upload("my-bucket",
"my-large-object")
    print("Multipart upload initiated. Upload ID:", upload_id)

except Exception as err:
    In case of error: Abort the multipart upload
    minioClient.abort_multipart_upload("my-bucket", "my-large-object", upload_id)
    print("Multipart upload aborted successfully.")
    print(f"Error: {err}")
```

`list_multipart_uploads`

The `list_multipart_uploads` function in the MinIO SDK is used to **list all ongoing or incomplete multipart uploads** within a specific bucket. This function helps identify multipart uploads that are still in progress or have been abandoned, allowing you to manage and potentially abort incomplete uploads that are no longer needed.

```python
def list_multipart_uploads(
    bucket_name: str,
    prefix: Optional[str] = None,
    key_marker: Optional[str] = None,
    upload_id_marker: Optional[str] = None,
    delimiter: Optional[str] = None,
    recursive: bool = False
) -> Iterator[Upload]
```

Parameters:

- **`bucket_name`** (str): The name of the bucket where the multipart uploads are being listed.
- **`prefix`** (Optional[str], default=None): Limits the multipart uploads returned to those that begin with the specified `prefix`. Useful for filtering uploads by object name or path.
- **`key_marker`** (Optional[str], default=None): Specifies the object name to start listing from. Used for pagination.
- **`upload_id_marker`** (Optional[str], default=None): Specifies the `upload_id` to start listing from. Used for pagination when there are multiple uploads with the same key.
- **`delimiter`** (Optional[str], default=None): A character used to group object names (like `/` for folders in a hierarchical structure).
- **`recursive`** (bool, default=False): When set to `True`, lists all multipart uploads under the specified bucket without grouping them based on the `delimiter`.

Return Value:

- **Returns** an **iterator of `Upload` objects**, each containing details about an ongoing or incomplete multipart upload (e.g., `key`, `upload_id`, `initiated_time`).

List All Multipart Uploads in a Bucket: In this example, we list all multipart uploads that are currently ongoing or incomplete in the specified bucket.

```
from minio import Minio

# Initialize the MinIO client
minioClient = Minio(
    "play.min.io",
    access_key="your-access-key",
    secret_key="your-secret-key",
    secure=True
)

try:
    # List all multipart uploads in the 'my-bucket' bucket
    uploads = minioClient.list_multipart_uploads("my-bucket")

    for upload in uploads:
        print(f"Object: {upload.object_name}, Upload ID: {upload.upload_id},
Initiated: {upload.initiated_time}")
except Exception as err:
    print(f"Error: {err}")
```

List Multipart Uploads with a Specific Prefix: You can filter multipart uploads by specifying a prefix. This is useful for listing uploads related to a specific path or folder-like structure.

```
try:
    # List multipart uploads that begin with 'uploads/'
    uploads = minioClient.list_multipart_uploads("my-bucket", prefix="uploads/")

    for upload in uploads:
        print(f"Object: {upload.object_name}, Upload ID: {upload.upload_id},
Initiated: {upload.initiated_time}")
except Exception as err:
    print(f"Error: {err}")
```

List Multipart Uploads with Pagination (Using Key and Upload ID Markers): If there are many multipart uploads, you can paginate through the results using the `key_marker` and `upload_id_marker`

```
try:
    # Start listing from a specific object name and upload ID
    uploads = minioClient.list_multipart_uploads("my-bucket",
key_marker="my-object", upload_id_marker="abcd1234")

    for upload in uploads:
        print(f"Object: {upload.object_name}, Upload ID: {upload.upload_id},
Initiated: {upload.initiated_time}")
except Exception as err:
    print(f"Error: {err}")
```

Key Considerations:

- **Filtering by Prefix**: You can use the `prefix` parameter to filter uploads and list only those that match a specific object name prefix or path, making it easier to manage uploads in a specific directory or with a certain name.
- **Pagination**: The `key_marker` and `upload_id_marker` parameters are useful for paginating through large lists of multipart uploads when there are too many to display in a single response.
- **Recursive Listing**: By setting `recursive=True`, you can list all multipart uploads under the specified bucket without grouping them by `delimiter`. This is useful for flat structures where you don't want hierarchical grouping.
- **Managing Incomplete Uploads**: This function helps track incomplete or abandoned multipart uploads, allowing you to identify uploads that can be resumed or aborted.

`list_parts`

The `list_parts` function in the MinIO SDK is used to **list all the parts that have been uploaded** for a specific multipart upload. It allows you to track the status of a multipart upload by listing the parts that have already been successfully uploaded. This is useful for resuming an interrupted upload, validating uploaded parts, or determining which parts still need to be uploaded.

Usage Scenario:

- **Tracking Multipart Upload Progress**: List all the parts that have already been uploaded to verify the status and progress of a multipart upload.
- **Resuming an Interrupted Upload**: If an upload is interrupted, use this function to list the parts that have been successfully uploaded and resume uploading the remaining parts.
- **Validating Uploaded Parts**: Ensure that all parts are uploaded correctly by listing and verifying the uploaded parts.

```python
def list_parts(
    bucket_name: str,
    object_name: str,
    upload_id: str,
    part_number_marker: Optional[int] = None,
    max_parts: Optional[int] = None
) -> Iterator[Part]
```

Parameters:

- **`bucket_name`** (str): The name of the bucket where the multipart upload is being done.
- **`object_name`** (str): The name of the object associated with the multipart upload.
- **`upload_id`** (str): The `upload_id` returned by the `initiate_multipart_upload` function, which uniquely identifies the multipart upload.
- **`part_number_marker`** (Optional[int], default=None): Specifies the part number after which listing should begin. This is useful for pagination when there are many parts.
- **`max_parts`** (Optional[int], default=None): Limits the number of parts returned in the response, used for pagination.

Return Value:

- **Returns** an **iterator of `Part` objects**, each representing a part that has been uploaded, containing details such as part number, ETag, and size.

  List All Uploaded Parts for a Multipart Upload: In this example, we list all the parts that have been uploaded for a specific multipart upload

```
from minio import Minio

# Initialize the MinIO client
minioClient = Minio(
    "play.min.io",
    access_key="your-access-key",
    secret_key="your-secret-key",
    secure=True
)

try:
    # List all parts for a multipart upload
    parts = minioClient.list_parts("my-bucket", "my-large-object",
"your-upload-id-here")

    for part in parts:
        print(f"Part Number: {part.part_number}, ETag: {part.etag}, Size:
{part.size}")
except Exception as err:
    print(f"Error: {err}")
```

Encryption

`put_bucket_encryption`

The `put_bucket_encryption` function in the MinIO SDK is used to **set default encryption configuration** for a specific bucket. This ensures that all objects uploaded to the bucket are automatically encrypted with a specified server-side encryption configuration, providing data protection by encrypting objects at rest.

```
def put_bucket_encryption(
    bucket_name: str,
    encryption_config: dict
) -> None
```

Parameters:

- **`bucket_name`** (str): The name of the bucket for which you want to configure the default encryption.
- **`encryption_config`** (dict): A dictionary containing the server-side encryption (SSE) configuration. This defines how objects in the bucket will be encrypted (e.g., using AWS SSE-S3, SSE-KMS, or SSE-C).

Return Value:

- **Returns**: None (the function applies the encryption configuration to the specified bucket).

Example 1: Set Default Bucket Encryption with SSE-S3 (Server-Side Encryption with S3 Managed Keys): In this example, the bucket is configured to use SSE-S3 encryption, which means MinIO will automatically encrypt objects using its own managed keys.

```
try:
    # Define the SSE-S3 encryption configuration
    encryption_config = {
        "Rule": {
            "ApplyServerSideEncryptionByDefault": {
                "SSEAlgorithm": "AES256"
            }
        }
    }

    # Apply the encryption configuration to the bucket
    minioClient.put_bucket_encryption("my-bucket", encryption_config)
```

```
    print("Bucket encryption set successfully.")
except Exception as err:
    print(f"Error: {err}")
```

Example 2: Set Default Bucket Encryption with SSE-KMS (Server-Side Encryption with KMS): In this example, the bucket is configured to use SSE-KMS encryption, where encryption keys are managed by a Key Management Service (KMS).

```
try:
    # Define the SSE-KMS encryption configuration
    encryption_config = {
        "Rule": {
            "ApplyServerSideEncryptionByDefault": {
                "SSEAlgorithm": "aws:kms",
                "KMSMasterKeyID": "your-kms-key-id"
            }
        }
    }

    # Apply the encryption configuration to the bucket
    minioClient.put_bucket_encryption("my-bucket", encryption_config)

    print("Bucket encryption with SSE-KMS set successfully.")
except Exception as err:
    print(f"Error: {err}")
```

Key Considerations:

- **Encryption Configurations**:
    - **SSE-S3**: Server-side encryption with S3-managed keys (AES256). The encryption keys are managed by MinIO.
    - **SSE-KMS**: Server-side encryption with AWS KMS or other Key Management Service providers, where the encryption keys are managed externally.
    - **SSE-C**: Server-side encryption with customer-provided keys (SSE-C), where the customer provides and manages the encryption keys.

`get_bucket_encryption`

`remove_bucket_encryption`

`Put_object_sse`

(SSE-C) Server-Side Encryption with Customer-Provided Keys (SSE-C):

SSE-C (Server-Side Encryption with Customer-Provided Keys) is a form of server-side encryption in which the customer provides the encryption keys used to encrypt and decrypt objects stored in MinIO. With SSE-C, MinIO performs the encryption and decryption operations, but the encryption key is managed entirely by the customer, providing additional control over data security.

In SSE-C:

- **MinIO performs encryption and decryption**, but it does not store or manage the encryption keys.
- **The customer must provide the encryption key** with each request to upload or retrieve an object. Without the correct key, MinIO cannot decrypt the object.
- This approach allows **greater control over encryption keys**, as the keys remain in the customer's control, even though MinIO manages the encryption process.

Example of How SSE-C Works:

- **Upload with SSE-C**: The customer provides the encryption key during the upload request, and MinIO uses this key to encrypt the object before storing it.
- **Download with SSE-C**: The customer must provide the same encryption key during the download request. MinIO uses this key to decrypt the object before returning it to the client.

Example Code: Uploading and Downloading an Object with SSE-C

Upload an Object with SSE-C: In this example, the object is uploaded to a bucket using a customer-provided encryption key.

```
from minio import Minio
from minio.sse import SseCustomerKey

# Initialize the MinIO client
```

```
minioClient = Minio(
    "play.min.io",
    access_key="your-access-key",
    secret_key="your-secret-key",
    secure=True
)

try:
    # Define a 32-byte encryption key
    sse_c_key = SseCustomerKey(b"32byteslongsecretkeymustprovided")

    # Upload a file using SSE-C encryption
    minioClient.fput_object(
        "my-bucket", "my-encrypted-object", "/path/to/my-file.txt", sse=sse_c_key
    )

    print("Object uploaded with SSE-C successfully.")
except Exception as err:
    print(f"Error: {err}")
```

1.

Download an Object with SSE-C: When retrieving the object, you must provide the same encryption key that was used during upload.

```
try:
    # Download the object using the same SSE-C encryption key
    minioClient.fget_object(
        "my-bucket", "my-encrypted-object", "/path/to/downloaded-file.txt",
sse=sse_c_key
    )

    print("Object downloaded and decrypted successfully.")
except Exception as err:
    print(f"Error: {err}")
```

Key Considerations:

- **Key Management**: The customer is responsible for managing and securely storing the encryption key. MinIO does not store or retain customer-provided keys.
- **Consistency**: The same encryption key must be provided for both upload and download operations. If the wrong key is used, the object cannot be decrypted.

- **Security**: SSE-C gives customers full control over encryption keys, but with that control comes the responsibility to protect and manage the keys securely.
- **Key Size**: The encryption key must be exactly 32 bytes long for AES-256 encryption.

Complete API List

1. Bucket Operations

- **`make_bucket(bucket_name: str, location: Optional[str] = None) -> None`**: Create a new bucket.
- **`list_buckets() -> List[Bucket]`**: List all buckets.
- **`remove_bucket(bucket_name: str) -> None`**: Delete an empty bucket.
- **`bucket_exists(bucket_name: str) -> bool`**: Check if a bucket exists.
- **`set_bucket_policy(bucket_name: str, policy: str) -> None`**: Set a bucket policy.
- **`get_bucket_policy(bucket_name: str) -> str`**: Get the bucket policy.
- **`remove_bucket_policy(bucket_name: str) -> None`**: Remove the bucket policy.
- **`set_bucket_lifecycle(bucket_name: str, lifecycle: str) -> None`**: Set a bucket lifecycle policy.
- **`get_bucket_lifecycle(bucket_name: str) -> str`**: Get the bucket lifecycle policy.
- **`remove_bucket_lifecycle(bucket_name: str) -> None`**: Remove the bucket lifecycle policy.
- **`set_bucket_notification(bucket_name: str, notification: str) -> None`**: Set bucket notifications.
- **`get_bucket_notification(bucket_name: str) -> str`**: Get bucket notifications.
- **`remove_bucket_notification(bucket_name: str) -> None`**: Remove bucket notifications.
- **`set_bucket_tags(bucket_name: str, tags: Dict[str, str]) -> None`**: Set tags on a bucket.
- **`get_bucket_tags(bucket_name: str) -> Dict[str, str]`**: Get the tags associated with a bucket.

- **remove_bucket_tags(bucket_name: str) -> None**: Remove all tags from a bucket.
- **enable_bucket_versioning(bucket_name: str) -> None**: Enable versioning on a bucket.
- **suspend_bucket_versioning(bucket_name: str) -> None**: Suspend versioning on a bucket.

2. Object Operations

- **put_object(bucket_name: str, object_name: str, data: io.BytesIO, length: int, content_type: Optional[str] = None, metadata: Optional[Dict[str, str]] = None, sse: Optional[Base] = None, progress: Optional[io.BytesIO] = None, part_size: Optional[int] = None, num_parallel_uploads: int = 1) -> ObjectWriteResult**: Upload an object.
- **get_object(bucket_name: str, object_name: str, offset: Optional[int] = 0, length: Optional[int] = 0, sse: Optional[Base] = None, version_id: Optional[str] = None, extra_query_params: Optional[Dict[str, str]] = None) -> HTTPResponse**: Retrieve an object.
- **fget_object(bucket_name: str, object_name: str, file_path: str, request_headers: Optional[Dict[str, str]] = None, sse: Optional[Base] = None, version_id: Optional[str] = None, progress: Optional[io.BytesIO] = None, offset: Optional[int] = 0, length: Optional[int] = 0) -> ObjectWriteResult**: Download an object to a file.
- **remove_object(bucket_name: str, object_name: str, version_id: Optional[str] = None, bypass_governance: bool = False) -> None**: Delete an object.
- **remove_objects(bucket_name: str, delete_object_list: Iterable[str]) -> List[DeleteError]**: Delete multiple objects.
- **copy_object(bucket_name: str, object_name: str, source: CopySource, sse: Optional[Base] = None, metadata: Optional[Dict[str, str]] = None) -> ObjectWriteResult**: Copy an object from a source to a destination.
- **stat_object(bucket_name: str, object_name: str, sse: Optional[Base] = None, version_id: Optional[str] = None) -> Object**: Get metadata/statistics of an object.

- **compose_object(bucket_name: str, object_name: str, sources: List[ComposeSource], sse: Optional[Base] = None, metadata: Optional[Dict[str, str]] = None) -> ObjectWriteResult**: Create an object by concatenating multiple source objects.
- **get_presigned_url(method: str, bucket_name: str, object_name: str, expires: timedelta, response_headers: Optional[Dict[str, str]] = None, request_date: Optional[datetime] = None) -> str**: Get a presigned URL to access an object.
- **put_presigned_object(bucket_name: str, object_name: str, data: io.BytesIO, length: int, headers: Optional[Dict[str, str]] = None) -> None**: Upload an object using a presigned URL.
- **select_object_content(bucket_name: str, object_name: str, expression: str, request_progress: bool = False, input_serialization: Optional[Dict[str, str]] = None, output_serialization: Optional[Dict[str, str]] = None) -> SelectObjectReader**: Run SQL queries on a CSV, JSON, or Parquet object.

3. Multipart Upload Operations

- **initiate_multipart_upload(bucket_name: str, object_name: str, metadata: Optional[Dict[str, str]] = None) -> str**: Initiate a new multipart upload and get the upload ID.
- **upload_part(bucket_name: str, object_name: str, data: io.BytesIO, length: int, upload_id: str, part_number: int, sse: Optional[Base] = None) -> str**: Upload a part of an object.
- **complete_multipart_upload(bucket_name: str, object_name: str, upload_id: str, etags: List[str], sse: Optional[Base] = None) -> ObjectWriteResult**: Complete a multipart upload.
- **abort_multipart_upload(bucket_name: str, object_name: str, upload_id: str) -> None**: Abort a multipart upload.
- **list_multipart_uploads(bucket_name: str, prefix: Optional[str] = None, key_marker: Optional[str] = None, upload_id_marker: Optional[str] = None, delimiter: Optional[str] = None, max_uploads: Optional[int] = None) -> Iterable[MultipartUpload]**: List ongoing multipart uploads.
- **list_parts(bucket_name: str, object_name: str, upload_id: str, part_number_marker: Optional[int] = None, max_parts:**

Optional[int] = None) -> Iterable[Part]: List the parts of a multipart upload.

4. Object Locking and Retention

- **set_object_lock_config(bucket_name: str, config: ObjectLockConfig) -> None**: Set the object lock configuration for a bucket.
- **get_object_lock_config(bucket_name: str) -> ObjectLockConfig**: Get the object lock configuration for a bucket.
- **put_object_retention(bucket_name: str, object_name: str, retention: Retention) -> None**: Set the object retention policy on an object.
- **get_object_retention(bucket_name: str, object_name: str) -> Retention**: Get the object retention policy on an object.
- **put_object_legal_hold(bucket_name: str, object_name: str, status: str) -> None**: Set the legal hold status on an object.
- **get_object_legal_hold(bucket_name: str, object_name: str) -> str**: Get the legal hold status on an object.

5. Encryption

- **put_bucket_encryption(bucket_name: str, config: BucketEncryptionConfig) -> None**: Set bucket encryption configuration.
- **get_bucket_encryption(bucket_name: str) -> BucketEncryptionConfig**: Get bucket encryption configuration.
- **remove_bucket_encryption(bucket_name: str) -> None**: Remove bucket encryption configuration.
- **put_object_sse(bucket_name: str, object_name: str, sse: Base) -> None**: Set server-side encryption for an object.

6. Monitoring and Administration

- **get_bucket_policy(bucket_name: str) -> str**: Retrieve the current policy applied to a bucket.
- **set_bucket_policy(bucket_name: str, policy: str) -> None**: Apply a policy to a bucket.

- **get_bucket_notification(bucket_name: str) -> str**: Retrieve the current notification configuration for a bucket.
- **set_bucket_notification(bucket_name: str, notification: str) -> None**: Configure notifications for specific events (like object creation, deletion, etc.) in a bucket.
- **remove_bucket_notification(bucket_name: str) -> None**: Remove all notifications configured for a bucket.
- **set_bucket_tags(bucket_name: str, tags: Dict[str, str]) -> None**: Add or update tags on a bucket.
- **get_bucket_tags(bucket_name: str) -> Dict[str, str]**: Retrieve the tags associated with a bucket.
- **remove_bucket_tags(bucket_name: str) -> None**: Remove all tags from a bucket.
- **list_incomplete_uploads(bucket_name: str, prefix: Optional[str] = None, recursive: bool = False) -> Iterable[Upload]**: List incomplete uploads.
- **clear_incomplete_uploads(bucket_name: str, prefix: Optional[str] = None, recursive: bool = False) -> None**: Remove all incomplete uploads in a bucket.

7. Miscellaneous

- **remove_objects(bucket_name: str, delete_object_list: Iterable[str]) -> List[DeleteError]**: Delete multiple objects from a bucket.
- **download_object(bucket_name: str, object_name: str, file_path: str) -> None**: Download an object from a bucket and save it to a file.
- **upload_file(bucket_name: str, object_name: str, file_path: str) -> None**: Upload a file to a bucket.
- **upload_part_copy(bucket_name: str, object_name: str, source: CopySource, upload_id: str, part_number: int, sse: Optional[Base] = None) -> str**: Upload a part of an object by copying data from an existing object.

8. Presigned Operations

- **get_presigned_url(method: str, bucket_name: str, object_name: str, expires: timedelta, response_headers: Optional[Dict[str,**

str]] = None, request_date: Optional[datetime] = None) -> str: Get a presigned URL to access an object.

- **presigned_get_object(bucket_name: str, object_name: str, expires: timedelta, response_headers: Optional[Dict[str, str]] = None) -> str**: Get a presigned URL for retrieving an object.
- **presigned_put_object(bucket_name: str, object_name: str, expires: timedelta) -> str**: Get a presigned URL for uploading an object.
- **presigned_post_policy(post_policy: PostPolicy) -> Tuple[str, Dict[str, str]]**: Generate a presigned URL and form data for a POST policy.