CIS 4010 Assignment 1, Task 4 | Christian Cornelis - ccorneli@uoguelph.ca

Procedure	**AWS Solution**	**AWS Explanation**	**Azure Solution**	**Azure Explanation**
Container Client Instantiation	boto3.resource() and boto3.client	Requires a populated ~/.aws/credentials file to instantiate correctly. A resource is a wrapper on the client class and was required to retrieve a bucket by name. The client was used to create buckets with the .create_bucket() method. The .head_bucket() method was used to check if a bucket existed.	azure.blob.storage.blob .BlobServiceClient()	Requires an environment variable to be set with the BlobStorage Connection string. Can be called via the .from_connection_string() method using this env variable.
Object Upload	boto3.client().upload_file()	This method simply required the path to the desired file to be uploaded, the bucket name, and the desired name of the object in the bucket.	BlobServiceClient.get_blob_client() BlobClient .upload_blob()	This method was used with the desired container and the object to be uploaded being sent as parameters. AWS did not require such an instantiation of a second client for object uploading. This second method utilized an open file in byte-reading move to upload the object to the container specified by the get_blob_client() method.
List all containers	boto3.client().list_buckets( )	Simply lists all buckets that the client has access to.	BlobServiceClient.list_containers()	Lists all containers the client has access to.
List all objects in a specified container	boto3.resource().Bucket()	Instantiates a connection to a specified bucket. Throws a ClientError if the bucket does not exist. Bucket.objects.all() retrieves all objects in the bucket instantiated.	BlobServiceClient.get_container_client()	Instantiates a connection to a specified container via a ContainerClient. Throws a ResourceNotFoundError when the container does not exist. ContainerClient.list_blobs() can be used to see all objects in the container.
Download an object	boto3.client() .download_file()	Takes in a bucket name, object name, and desired downloaded-file name as parameters. Writes the file to the file	BlobClient.download_blob ().readall()	Requires a BlobClient to be instantiated with the container and blob specified as parameters.

		system automatically.		.download_blob() is called, which returns a StorageStreamDownloader object. .readall() Reads all bytes in the blob. This method must be used in conjunction with opening a new file in mode 'wb' and writing the return of .readall() to the open file. This was not necessary on AWS.

Database Client Instantiation	boto3.resource()
	
	and
	
	boto3.client()
		Same as above, except 'dynamodb' is specified as the resource.
		
		Region_name was also necessary in order to instantiate the client/resource on DynamoDB instances in the region necessary. This was not necessary in Azure.
			azure.cosmosdb.table.tableservice.TableService	A TableService object is instantiated by passing in the connection string to the Azure CosmosDB account.

Database Table Instantiation	boto3.resource().create_table()	create_table() takes in parameters specifying the Table name, key schema, attributes, and throughput provisioning. Attribute types can be defined, as well as the sort and partition keys. Returns a connection to the table. The table status is 'CREATING' until it is ready, at which point it changes to 'ACTIVE' and the table can be populated or queried-upon.	TableService.create_table()	.create_table() is called passing the table name to the Service. No key or attribute definitions are required, which made instantiation much simpler.
Database Table Population	boto3.resource().put_item()	Takes a dictionary specified as the Item parameter that will be added to a new row in the table. The sort and partition keys must be specified, but all other rows are optional.	TableService.insert_entity()	Takes an Entity Object, or a literal definition of a dictionary containing 'PartitionKey' and 'RowKey' as keys, as a param. Entity objects can be used to create rows (entities) by defining their attributes and values as a dictionary. 'PartitionKey' has many limitations on the characters it can contain, so encoding the data is required. For defining Entity attributes that require floating-point decimals,

				the EntityProperty class can be instantiated and defined as type EdmType.DOUBLE to avoid headaches with this type of data. Care should be taken with the types that Entity attributes are cast to, as int() in Python gets cast to Int64 in a CosmosDB Table. All other attributes aside from the PartitionKey and RowKey are optional.
Database Query	boto3.resource().scan()	Scan takes in a very large number of parameters, but I only required use of FilterExpression, ProjectionExpression, ExpressionAttributeNames, and ExclusiveStartKey. FilterExpression represents the filter to apply to the table. It is built using boto3.dynamodb.conditions.Attr() objects, which have built-in operations, and are quite easy to chain together once created. ProjectionExpression represents the attributes to project the filter on in a comma-separated string. ExpressionAttributeNames were used to get around the issue involving reserved-keywords as attribute names (ex. Year and rank). An alias was created in the ProjectionExpression string, and a dictionary defining the alias used in ProjectionExpression (ex. #y for year) was used as the ExpressionAttributeNames parameter. ExclusiveStartKey was used to represent where a scan should begin querying the table. This was used when the maximum response size was reached and an additional scan() call was necessary.	TableService.query_entities()	.query_entities() can be called passing the table name, filter, and a comma-separated list of attributes to filter from. The filter was quite easy to build as it is entirely string-based and used operators such as ‘gt’, ‘lt’, and ‘eq’ for ranges. Querying via the CosmosDB SDK was much simpler due to the lack of an issue regarding response-size, as AWS had.

References

[1] D. Stacey, “CIS 4010 AWS and Azure Course Notes/Sample Code”, https://courselink.uoguelph.ca/d2l/home/606015, 2020, online; accessed 20 Jan 2020.
[2] Microsoft. “Querying tables and entities” https://docs.microsoft.com/en-us/rest/api/storageservices/querying-tables-and-entities, online, unknown; accessed 24 Jan 2020.
[3] GitHub. “Azure/azure-cosmos-python” https://github.com/Azure/azure-cosmos-python, online, 2019; accessed 24 Jan 2020.
[4] Microsoft. “TableService class” “https://docs.microsoft.com/en-ca/python/api/azure-cosmosdb-table/azure.cosmosdb.table.tableservice.tableservice?view=azure-python#create-table-table-name--fail-on-exist-false--timeout-none-, online, unknown; accessed 25 Jan 2020.
[5] Amazon Web Services Inc. “DynamoDB”. https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/dynamodb.html, online, 2019; accessed 27 Jan 2020.
[6] Amazon Webservices Inc. “S3 - Boto 3 Docs 1.11.9 documentation” https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html, 2019, online; accessed 20 Jan 2020.