# Lab 2 - Implement and Optimize User Profile Data Management Workload on Amazon DocumentDB

1. Design a modern application on an AWS NoSQL database, Amazon DocumentDB.
2. Implement Amazon DocumentDB APIs to manage application data in Amazon DocumentDB tables.
3. Use indexes to optimize queries for performance and cost.
4. Design and enter an aggregation pipeline to support business uses requiring aggregated data.
5. Implement a cost-optimized Amazon DocumentDB change streams solution using Lambda event filtering.

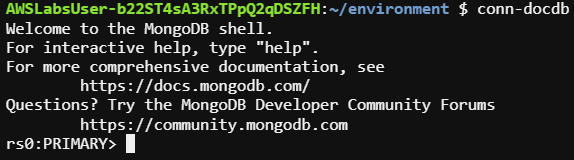
**Task 1: Review the development environment**

1.1 open cloud9 and connect to ec2



1.2 Connect to the DocumentDB cluster

Conn-docdb



**Task 2: Implement business requirements for data management of user profiles**

There is a lambda function called CreateUpdateProfile to handle creating and updating profiles.

It connects to the database, checks if you want to create (POST) or update (PUT) a profile, and then performs the action.

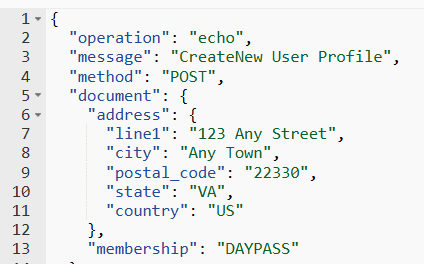
2.1 

2.2 .py file for CreateUpdateProfile

| import os  import pymongo  import boto3  import logging  import json  from datetime import datetime  logger = logging.getLogger()  logger.setLevel(logging.INFO)  SECRET\_VALUE = None  CACHE\_DBCONNECTION = None  def get\_credentials(secret\_name):  session = boto3.session.Session()  global SECRET\_VALUE  try:  logger.info('Retrieving secret {} from Secrets Manger.'.format(secret\_name))  client = session.client(service\_name='secretsmanager', region\_name=session.region\_name)    if SECRET\_VALUE is None:  logging.info('Cache miss: Getting fresh db credentials.')  SECRET\_VALUE = client.get\_secret\_value(SecretId=secret\_name)  else:  logging.info('Cache hit! db credentials.')  # SECRET\_VALUE = secret\_value  secret\_value = SECRET\_VALUE  secret\_json = json.loads(secret\_value['SecretString'])  username = secret\_json['username']  password = secret\_json['password']  cluster\_uri = secret\_json['host']  logger.info('Secret {} retrieved from Secrets Manger.'.format(secret\_name))  return (username, password, cluster\_uri)  except Exception as e:  logger.error('Failed to retrieve secret {} because: {}'.format(secret\_name, e))  def db\_connect():    global CACHE\_DBCONNECTION    try:  if CACHE\_DBCONNECTION:  return CACHE\_DBCONNECTION  else:    # Retrieve DocumentDB credentials and cluster uri from AWS Secrets Manager  secret\_name = os.environ['DOCDB\_SECRET\_NAME'] # Environment variable for secret name  (secret\_username, secret\_password, cluster\_uri) = get\_credentials(secret\_name)    # Retrieve database\_name and collection\_name from environment variables  database\_name = os.environ['DOCDB\_DATABASE']  collection\_name = os.environ['DOCDB\_COLLECTION']    # Connect to Amazon DocumentDB  logger.info('Creating new DocumentDB client.')  CACHE\_DBCONNECTION = pymongo.MongoClient(  cluster\_uri,  tls=True,  retryWrites=False,  tlsCAFile='/opt/python/global-bundle.pem',  username=secret\_username,  password=secret\_password,  authSource='admin')  logger.info('Successfully created new DocumentDB client.')    return CACHE\_DBCONNECTION    except Exception as e:  logger.error('An error occured: {}'.format(e))      def lambda\_handler(event, context):  try:  # Retrieve database\_name and collection\_name from environment variables  database\_name = os.environ['DOCDB\_DATABASE']  collection\_name = os.environ['DOCDB\_COLLECTION']    ## Connect to Amazon DocumentDB  client = db\_connect()  ## Set database to bikeapp  db = client[database\_name]  ## Set collection to userprofile  collection = db[collection\_name]    ## Parse API Gateway invocation containing userprofile JSON  document = event['document']  ## Parse API method  method = event['method']    document = event['document']    ## For Creating an user's profile (API METHOD: POST /userprofile)  if method == 'POST':  ## Generate unique userid: \_id  document\_id = f"U{str(round(datetime.now().timestamp()))[-7:]}"    ## Insert new userpfile document  #### Append primary key: \_id into the new userpofile document  document['\_id'] = document\_id  result = collection.insert\_one(document)  logger.info('Inserted document with ID: {}.'.format(result.inserted\_id))      ## Return API Gateway response  #### Status Code: 201  response = { 'status': 201, 'message': 'User Profile {} has been created!'.format(result.inserted\_id) }      ## For Updating an user's profile (API METHOD: PUT /userprofile)  elif method == 'PUT':  ## Parse userid from the event JSON  document\_id = event.get('userid')  # Update the document and retrieve the updated version  updated\_document = collection.find\_one\_and\_update(  {'\_id': document\_id},  {'$set': document},  return\_document=pymongo.collection.ReturnDocument.AFTER  )  logger.info('Updated document: {}'.format(updated\_document))  logger.info('Document updated successfully')    ### Returns an API gateway response with a status code and message of SUCCESS or ERROR.  response = { 'status': 200, 'message': 'User Profile {} has been updated!'.format(document\_id) }  return response  except Exception as e:  logger.error('An error occured: {}'.format(e)) |
| --- |

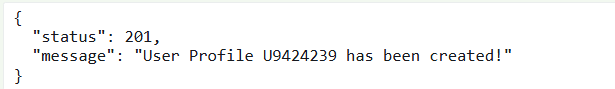
2.3 Verify creation of the user profile

2.4 in test section, create a user



Notice the POST method

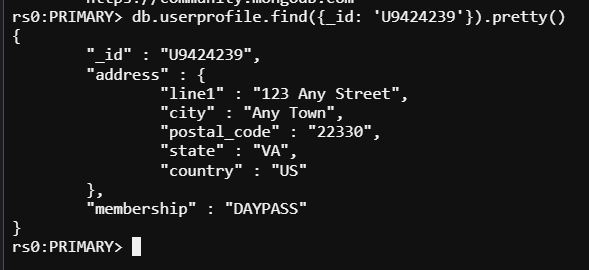
Op-



2.5 try updating the profile, from Cloud9

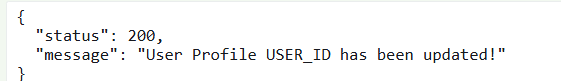
2.6 find the user

db.userprofile.find({\_id: 'USER\_ID'}).pretty()



2.7 update from lambda tab, using PUT





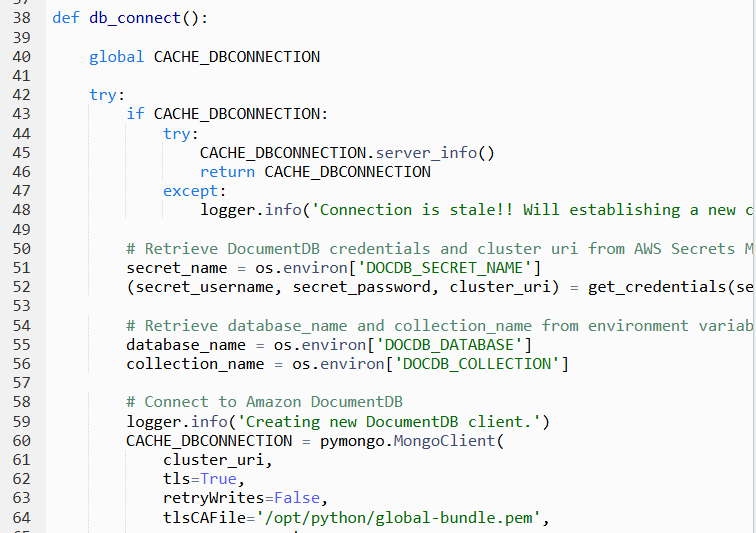
2.8 verify in Cloud9

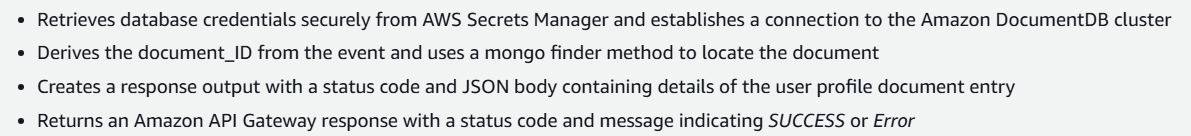


2.9 GetProfile lambda func to retrieve profile details.



Code -





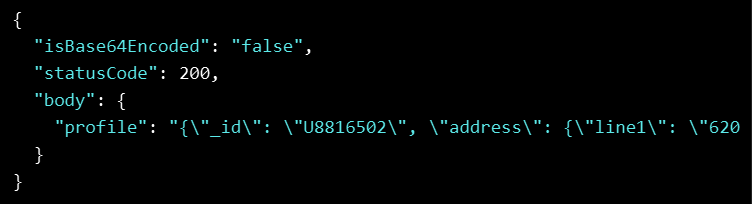
2.10 using “test” for this

{

"method": "GET",

"userid": "USER\_ID"

}



**Task 3: Implement business requirements for scoring customers**

3.1 Find customers with most scores

*db.usertrips.aggregate(*

*[*

*{$match: {"tripdate": {$gte: ISODate('2023-01-01')}}},*

*{$project: { \_id: 0, userid: 1, fare: 1}},*

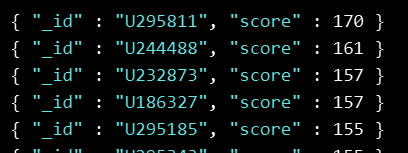
*{$group: { \_id: "$userid", score: { $sum: "$fare" }}},*

*{$out: "userscore" }*

*]*

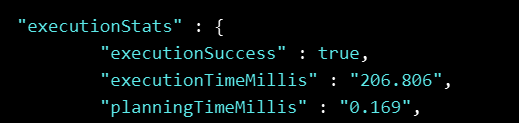
*)*

3.2 db.userscore.find().sort({score:-1}).limit(20)



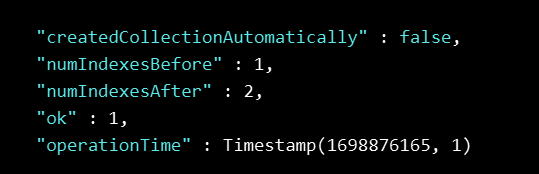
3.3 get the execution stats

db.userscore.explain('executionStats').find().sort({score: -1}).limit(20)

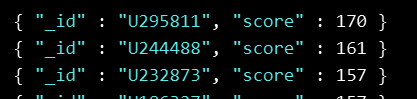


3.4 now, create and use an index for sorting

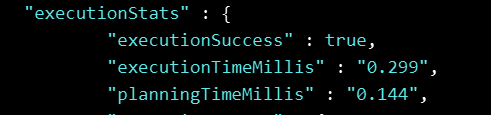
*db.userscore.createIndex({score:-1},{"name":"idx\_userscore"})*



*db.userscore.find().sort({score:-1}).limit(20)*



3.5 stats with index



Much reduced