

# Assignment 2: Cassandra DB



[Github](#)

# PROJECT DOCUMENTATION

**DATE:** 25/12/2023

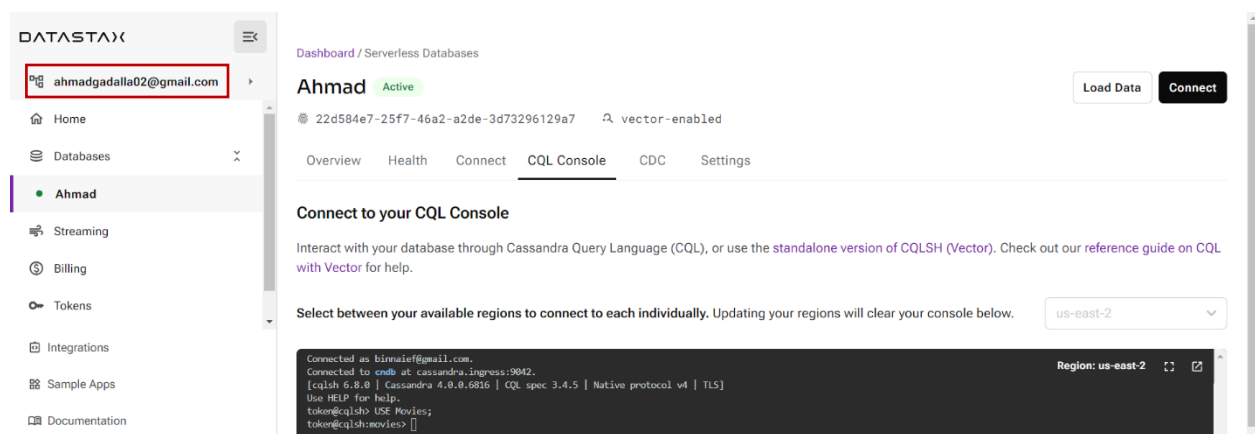
**PROJECT TITLE:** DataStax Astra & Python: Cassandra Showcase

**OBJECTIVE:** Utilizing DataStax Astra DB by executing key database tasks—creating keyspaces, populating data, and schema exploration. Additionally, aiming to demonstrate Python integration for image-to-blob transformations, querying based on directors/actors, and assessing TTL's impact on data persistence in Cassandra.

## GIVEN TASKS:

### 1. Create a Keyspace:

- Name: "Movies"
- Replication factor: 1
- Strategy: Simple strategy
- Screenshot required with your email visible.



## 2. Create a Column-Family "Movie":

- Columns: Id (int), name (text), movie-cast (map), movie-poster (blob)

```
CREATE TABLE movies.movie (  
  id int PRIMARY KEY,  
  movie_cast map<text, text>,  
  movie_poster blob,  
  name text  
)
```

- Set the first row TTL to 7 days.

```
USING TTL 604800;
```

## 3. Check the schema of the "Movie" column-family.

```
Select * from movie;
```

```
token@cqlsh:movies> select * from movie;  
  
id | movie_cast | movie_poster | name  
---+-----+-----+-----  
  
(0 rows)
```

```
describe movie;
```

- Ahmad
- Streaming
- Billing
- Tokens
- Settings

```
CREATE TABLE movies.movie (  
  id int PRIMARY KEY,  
  movie_cast map<text, text>,  
  movie_poster blob,  
  name text  
) WITH additional_write_policy = '99p'  
  AND bloom_filter_fp_chance = 0.01  
  AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}  
  AND comment = ''  
  AND compaction = {'class': 'org.apache.cassandra.db.compaction.UnifiedCompactionStrategy'}  
  AND compression = {'chunk_length_in_kb': '16', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}  
  AND crc_check_chance = 1.0  
  AND default_time_to_live = 604800  
  AND gc_grace_seconds = 864000  
  AND max_index_interval = 2048  
  AND memtable_flush_period_in_ms = 0  
  AND min_index_interval = 128  
  AND read_repair = 'BLOCKING'  
  AND speculative_retry = '99p';  
  
token@cqlsh:movies> []
```

#### 4. Populate the Movie table:

- Insert data for 3 movies (ID, name, movie-cast, excluding movie-poster).

```
USE Movies;
```

```
INSERT INTO movie (id, movie_cast, movie_poster, name) VALUES
(1, {'director': 'David Frankel', 'actors': 'Meryl Streep, Anne
Hathaway', 'music_cast': 'Theodore Shapiro'}, null, 'The Devil Wears
Prada');
```

```
INSERT INTO movie (id, movie_cast, movie_poster, name) VALUES
(2, {'director': 'Tim Burton', 'actors': 'Johnny Depp, Freddie
Highmore', 'music_cast': 'Danny Elfman'}, null, 'Charlie and the
Chocolate Factory');
```

```
INSERT INTO movie (id, movie_cast, movie_poster, name) VALUES
(3, {'director': 'Pete Docter', 'actors': 'John Goodman, Billy
Crystal', 'music_cast': 'Randy Newman'}, null, 'Monsters, Inc.');
```

- Check the schema.

```
Select id, name, movie_cast, from movie;
```

```
id | name | movie_cast
---+---+---
1 | The Devil Wears Prada | {'actors': 'Meryl Streep, Anne Hathaway', 'director': 'David Frankel', 'music_cast': 'Theodore Shapiro'}
2 | Charlie and the Chocolate Factory | {'actors': 'Johnny Depp, Freddie Highmore, hi, Jana '}
3 | Monsters, Inc. | {'actors': 'John Goodman, Billy Crystal', 'director': 'Pete Docter', 'music_cast': 'Randy Newman'}

(3 rows)
token@cqlsh:movies> describe movie;

CREATE TABLE movies.movie (
  id int PRIMARY KEY,
  movie_cast map<text, text>,
  movie_poster blob,
  name text
) WITH additional_write_policy = '99p'
AND bloom_filter_fp_chance = 0.01
AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
AND comment = ''
AND compaction = {'class': 'org.apache.cassandra.db.compaction.UnifiedCompactionStrategy'}
AND compression = {'chunk_length_in_kb': '16', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
AND crc_check_chance = 1.0
AND default_time_to_live = 604800
AND gc_grace_seconds = 864000
AND max_index_interval = 2048
AND memtable_flush_period_in_ms = 0
AND min_index_interval = 128
AND read_repair = 'BLOCKING'
AND speculative_retry = '99p';

token@cqlsh:movies> |
```

## 5. Write a Python function to update movie-poster column:

- Function connects to Keyspace, transforms movie posters (from a local folder) to blob datatype, and updates the movie-poster column for the 3 inserted rows.

```
def connect_and_update_blob(folder_path):
    # Load credentials from JSON file
    with open("ahmadgadalla02@gmail.com-token.json") as f:
        secrets = json.load(f)

    CLIENT_ID = secrets["clientId"]
    CLIENT_SECRET = secrets["secret"]

    # Connection setup using secure connect bundle
    cloud_config = {
        'secure_connect_bundle': 'secure-connect-ahmad.zip'
    }

    auth_provider = PlainTextAuthProvider(CLIENT_ID, CLIENT_SECRET)

    # Create the cluster and session
    cluster = Cluster(cloud=cloud_config, auth_provider=auth_provider)
    session = cluster.connect()

    # Keyspace to use
    keyspace_name = "movies"

    # Use the keyspace
    session.set_keyspace(keyspace_name)

    # Get all files in the folder
    files = os.listdir(folder_path)[:3] # Select the first three
files in the folder

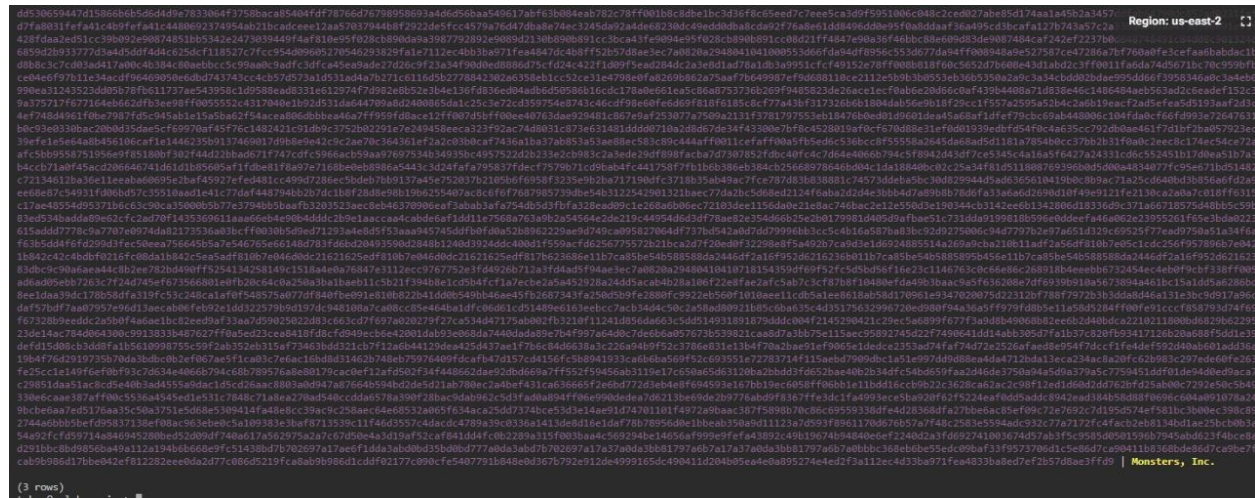
    # Update movie-poster column for the three inserted rows
    for idx, file_name in enumerate(files, 1):
        # Read the image as binary data
        with open(os.path.join(folder_path, file_name), 'rb') as
image_file:
            image_data = image_file.read()

        # Update the movie-poster column for the specific row
        query = f"UPDATE movie SET movie_poster = %s WHERE id = {idx}"
        session.execute(query, (image_data,)) # Assuming 'movie'
table and 'id' column

    cluster.shutdown()
```



```
Select * from movie;
```



## 6. A Python function to query movies based on name/director/actor:

- Discuss the feasibility of using "Like" or "Contains" CQL operators for this query.

*In Cassandra, direct 'LIKE' or 'CONTAINS' operators aren't supported. Instead, the 'ALLOW FILTERING' option enables queries on non-indexed columns like name, director, or actor. A Python function is used:*

```
SELECT * FROM movie ALLOW FILTERING;
```

*to perform queries based on these criteria, compensating for the absence of traditional string-matching operators.*

```
token@cqlsh:movies> SELECT * FROM movie WHERE movie_cast CONTAINS 'Johnny Depp';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"
```

- Function takes **movie name input**, queries the DB, displays the row results including the image.

```
def query_and_display_movies_name(search_term, output_folder):
    with open("ahmadgadalla02@gmail.com-token.json") as f:
        secrets = json.load(f)
```

```
CLIENT_ID = secrets["clientId"]
CLIENT_SECRET = secrets["secret"]
```

```

cloud_config = {
    'secure_connect_bundle': 'secure-connect-ahmad.zip'
}

auth_provider = PlainTextAuthProvider(CLIENT_ID, CLIENT_SECRET)

cluster = Cluster(cloud=cloud_config, auth_provider=auth_provider)
session = cluster.connect()

try:
    keyspace_name = "movies"

    session.set_keyspace(keyspace_name)

    query = f"SELECT id, movie_cast, movie_poster, name FROM movie
WHERE name = '{search_term}' ALLOW FILTERING;"

    rows = session.execute(query)
    os.makedirs(output_folder, exist_ok=True)

    for row in rows:
        print(f"Movie ID: {row.id}, Name: {row.name}, Movie Cast:
{row.movie_cast}\n")
        print(f"_____")

        # Save the image to the output folder
        if row.movie_poster:
            image_data = row.movie_poster
            image = Image.open(BytesIO(image_data))
            image_path = os.path.join(output_folder,
f"{row.id}_{row.name}_poster.jpg")
            image.save(image_path)
            print(f"Image saved to: {image_path}")

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        cluster.shutdown()

```

- Function **director/actor input**, queries the DB, displays the row results including the image.

```
def query_and_display_movies_by_director_or_actor(search_term,
output_folder):

    with open("ahmadgadalla02@gmail.com-token.json") as f:
        secrets = json.load(f)

    CLIENT_ID = secrets["clientId"]
    CLIENT_SECRET = secrets["secret"]

    cloud_config = {
        'secure_connect_bundle': 'secure-connect-ahmad.zip'
    }

    auth_provider = PlainTextAuthProvider(CLIENT_ID, CLIENT_SECRET)

    cluster = Cluster(cloud=cloud_config, auth_provider=auth_provider)
    session = cluster.connect()

    try:
        keyspace_name = "movies"

        session.set_keyspace(keyspace_name)

        query = f"SELECT * FROM movie ALLOW FILTERING;"

        rows = session.execute(query)

        # Create the output folder if it doesn't exist
        os.makedirs(output_folder, exist_ok=True)

        for row in rows:
            movie_id = row.id
            movie_name = row.name
            movie_cast = row.movie_cast
            movie_cast = str(movie_cast)
            if search_term in movie_cast:
                movie_poster = row.movie_poster
                print(f"Movie ID: {movie_id}, Name: {movie_name},
Cast:{movie_cast}")
                image_data = row.movie_poster
                image = Image.open(BytesIO(image_data))
                image_path = os.path.join(output_folder,
f"{row.id}_{row.name}_poster.jpg")
```



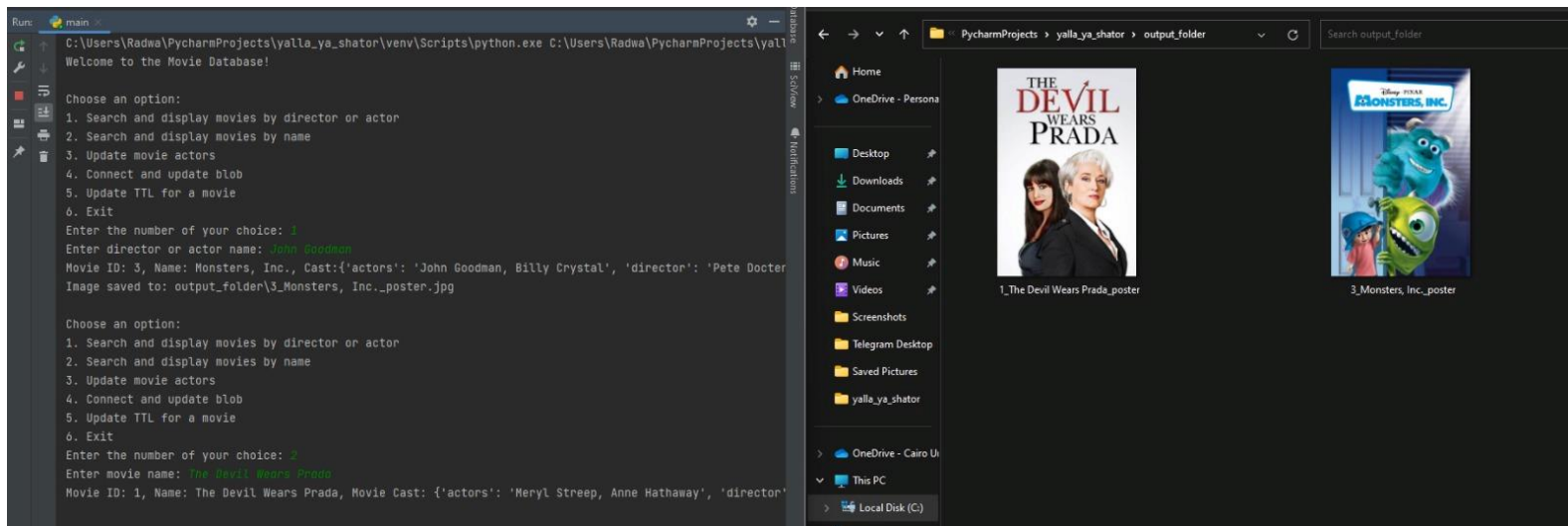
```

        image.save(image_path)
        print(f"Image saved to: {image_path}")
    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        cluster.shutdown()

```

- Output Check.



## 7. Update a certain movie's actors list:

- Add another actor to the existing list.

```

def update_movie_actors(movie_id, new_actor):
    with open("ahmadgadalla02@gmail.com-token.json") as f:
        secrets = json.load(f)

    CLIENT_ID = secrets["clientId"]
    CLIENT_SECRET = secrets["secret"]
    cloud_config = {
        'secure_connect_bundle': 'secure-connect-ahmad.zip'
    }

    auth_provider = PlainTextAuthProvider(CLIENT_ID, CLIENT_SECRET)

    cluster = Cluster(cloud=cloud_config, auth_provider=auth_provider)
    session = cluster.connect()

```

```

try:

    keyspace_name = "movies"

    session.set_keyspace(keyspace_name)

    select_query = f"SELECT movie_cast FROM movie WHERE id = {movie_id};"
    result = session.execute(select_query).one()

    existing_actors_str = result.movie_cast.get('actors', '') if
result.movie_cast else ''

    updated_actors_str = f"{existing_actors_str}, {new_actor}" if
existing_actors_str else new_actor

    update_query = f"UPDATE movie SET movie_cast = {{ 'actors':
'{updated_actors_str}' }} WHERE id = {movie_id};"
    session.execute(update_query)

    print(f"Updated movie {movie_id} with {new_actor} added to the
actor list.")

except Exception as e:
    print(f"An error occurred: {e}")

finally:
    cluster.shutdown()

```

- Check Schema.

```

Choose an option:
1. Search and display movies by director or actor
2. Search and display movies by name
3. Update movie actors
4. Connect and update blob
5. Update TTL for a movie
6. Exit
Enter the number of your choice: 3
Enter movie ID: 2
Enter new actor's name: Jana
Updated movie 2 with Jana added to the actor list.

Choose an option:
1. Search and display movies by director or actor
2. Search and display movies by name
3. Update movie actors
4. Connect and update blob
5. Update TTL for a movie
6. Exit
Enter the number of your choice: 1
Enter director or actor name: Jana
Movie ID: 2, Name: Charlie and the Chocolate Factory, Cast: {'actors': 'Johnny Depp, Freddie Highmore, hi, Jana '}
Image saved to: output_folder\2_Charlie and the Chocolate Factory_poster.jpg

```



1\_The Devil Wears Prada\_poster



2\_Charlie and the Chocolate Factory\_poster



```
select id, name, movie_cast from movie;
```

```
Connected to cqlsh at cassandra:19000.
[cqlsh 5.0.0 | Cassandra 3.0.8.003 | CQL spec 3.4.3 | Native protocol v4 | TLS]
Use HELP for help.
tokens@cqlsh:USE movies;
tokens@cqlsh:select id, name, movie_cast from movie;

id | name | movie_cast
--+-----+-----
1 | The Devil Wears Prada | ('actors': 'Meryl Streep, Anne Hathaway', 'director': 'David Frankel', 'music_cast': 'Theodore Shapiro')
2 | Charlie and the Chocolate Factory | ('actors': 'Johnny Depp, Freddie Highmore, Hilary Swank')
3 | Monsters, Inc. | ('actors': 'John Goodman, Billy Crystal', 'director': 'Pete Docter', 'music_cast': 'Randy Newman')

(3 rows)
tokens@cqlsh:describe movie;

CREATE TABLE movies.movie (
  id int PRIMARY KEY,
  movie_cast map<text, text>,
  movie_poster blob,
  name text
) WITH additional_write_policy = '99p'
AND bloom_filter_fp_chance = 0.01
AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
AND comment = ''
AND compaction = {'class': 'org.apache.cassandra.db.compaction.UnifiedCompactionStrategy'}
AND compression = {'chunk_length_in_kb': '1K', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
AND crc_check_chance = 1.0
AND default_time_to_live = 86400
AND gc_grace_seconds = 864000
AND max_index_interval = 2048
AND memtable_flush_period_in_ms = 0
AND min_index_interval = 128
AND read_repair = 'BLOCKING'
AND speculative_retry = '99p';

tokens@cqlsh:quit;
```

## 8. Update the first row TTL to 3 seconds:

- Re-query the table after the TTL expires to observe the changes.

```
def update_ttl(movie_id, ttl):

    with open("ahmadgadalla02@gmail.com-token.json") as f:
        secrets = json.load(f)

    CLIENT_ID = secrets["clientId"]
    CLIENT_SECRET = secrets["secret"]

    cloud_config = {
        'secure_connect_bundle': 'secure-connect-ahmad.zip'
    }

    auth_provider = PlainTextAuthProvider(CLIENT_ID, CLIENT_SECRET)

    cluster = Cluster(cloud=cloud_config, auth_provider=auth_provider)
    session = cluster.connect()

    keyspace_name = "movies"
    session.set_keyspace(keyspace_name)

    # Step 1: Read the existing row with the desired movie_id
    query = "SELECT * FROM movie WHERE id = %s"
    result = session.execute(query, (movie_id,))
    for row in result:
        movie_details = row

    if not movie_details:
        print(f"No movie found with ID: {movie_id}")
```

```

        return
    try:
        # Step 2: Delete the old row
        delete_query = "DELETE FROM movie WHERE id = %s"
        session.execute(delete_query, (movie_id,))

        # Step 3: Insert the row back with the new TTL value
        insert_query = """
        INSERT INTO movie (id, name, movie_cast, movie_poster)
        VALUES (%s, %s, %s, %s) USING TTL %s
        """
        session.execute(insert_query, (movie_details.id,
        movie_details.name, movie_details.movie_cast,
        movie_details.movie_poster, ttl))
        print(f"TTL updated successfully for movie with ID:
        {movie_id}")

    except Exception as e:
        print(f"An error occurred: {e}")

    session.shutdown()
    cluster.shutdown()

```

- Check output and schema.

```

Run: main x
D:\AdvAssign2\venv\Scripts\python.exe D:/AdvAssign2/main.py
Welcome to the Movie Database!

Choose an option:
1. Search and display movies by director or actor
2. Search and display movies by name
3. Update movie actors
4. Connect and update blob
5. Update TTL for a movie
6. Exit
Enter the number of your choice: 5
Enter movie ID: 2
Enter new TTL (time to live) in seconds: 3
TTL updated successfully for movie with ID: 2

```

```
select TTL (name) from movie where id = 1;
```

```
(1 rows)
token@cqlsh:movies> select TTL (name) from movie where id = 1;

ttl(name)
-----
10

(1 rows)
token@cqlsh:movies> select TTL (name) from movie where id = 1;

ttl(name)
-----
7

(1 rows)
token@cqlsh:movies> select TTL (name) from movie where id = 1;

ttl(name)
-----
4

(1 rows)
token@cqlsh:movies> select TTL (name) from movie where id = 1;

ttl(name)
-----
3

(1 rows)
token@cqlsh:movies> select TTL (name) from movie where id = 1;

ttl(name)
-----
3

(0 rows)
token@cqlsh:movies> █
```

## 9. Menu:

```
def main_menu():
    print("Welcome to the Movie Database!")

    while True:
        print("\nChoose an option:")
        print("1. Search and display movies by director or actor")
        print("2. Search and display movies by name")
        print("3. Update movie actors")
        print("4. Connect and update blob")
        print("5. Update TTL for a movie")
        print("6. Exit")

        choice = input("Enter the number of your choice: ")

        if choice == "1":
            search_term = input("Enter director or actor name: ")
            output_folder = input("Enter output folder: ")
            query_and_display_movies_by_director_or_actor(search_term,
output_folder)

            elif choice == "2":
```

```

        search_term = input("Enter movie name: ")
        output_folder = input("Enter output folder: ")
        query_and_display_movies_name(search_term, output_folder)

    elif choice == "3":
        movie_id = input("Enter movie ID: ")
        new_actor = input("Enter new actor's name: ")
        update_movie_actors(int(movie_id), new_actor)

    elif choice == "4":
        folder_path = input("Enter folder path: ")
        connect_and_update_blob(folder_path)

    elif choice == "5":
        movie_id = input("Enter movie ID: ")
        ttl = input("Enter new TTL (time to live) in seconds: ")
        update_ttl(int(movie_id), int(ttl))

    elif choice == "6":
        print("Exiting the Movie Database. Goodbye!")
        break

    else:
        print("Invalid choice. Please enter a valid number.")

main_menu()

```

- Check output.

```

Welcome to the Movie Database!

Choose an option:
1. Search and display movies by director or actor
2. Search and display movies by name
3. Update movie actors
4. Connect and update blob
5. Update TTL for a movie
6. Exit
Enter the number of your choice: 4
Enter folder path: p

Choose an option:
1. Search and display movies by director or actor
2. Search and display movies by name
3. Update movie actors
4. Connect and update blob
5. Update TTL for a movie
6. Exit
Enter the number of your choice: 6
Exiting the Movie Database. Goodbye!
|

```





**Cooked by:**

- |                         |          |
|-------------------------|----------|
| 1- Menna Elminshawy     | 20217011 |
| 2- Ahmad Wael Abdelaziz | 20216016 |
| 3- Ahmed Khaled Ahmed   | 20216004 |
| 4- Jana Mohamed Nayef   | 20216129 |
| 5- Radwa Belal Abdallah | 20217005 |

**Group ID: DS2**