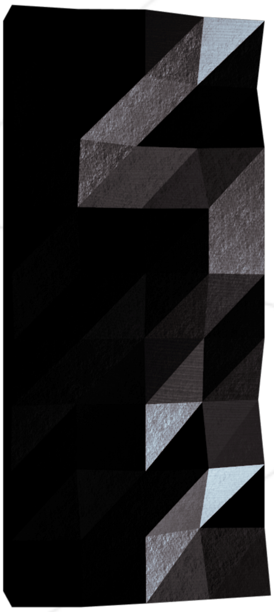


Building Applications with Python

Module 4

Summary:

Version: 1.1

Contents

Chapter I

	Exercise 01
NoSQL Database	
Turn-in directory: *ex*01/	
Files to turn in: `build_table.py, database.py`	
Allowed functions: `Standard library, boto3, python-dotenv`	

In this exercise, we will explore DynamoDB, AWS's NoSQL database. Unlike relational databases like PostgreSQL, which use separate tables and relationships, DynamoDB adopts an access-oriented model: data is organized according to query patterns, and not necessarily in normalized structures.

In this module, we will use a common approach in NoSQL databases: storing different types of data in the same table, differentiating them by composite keys. This modeling will allow us to re-implement the Ningi Points API in a more scalable way.

1. Start an instance of `dynamodb-local`, using Docker.

```
?> docker run -d \
    --name dynamodb-local \
    -p 8000:8000 \
    -v dynamodb-data:/home/dynamodblocal/data \
    amazon/dynamodb-local \
    -jar DynamoDBLocal.jar -sharedDb
```

2. In the `database.py` file, create a `get_client` function that returns a `dynamodb` client, as shown below:

```
dynamodb = boto3.client(
    'dynamodb',
    region_name='us-east-1',
    endpoint_url='http://localhost:8000',
    aws_access_key_id='dummy',
    aws_secret_access_key='dummy',
)
```

In real applications, credential configuration should be done with environment variables or AWS CLI profiles. Here, we use fixed values for simplicity and because they are dummy credentials with specific operation for the development environment with dynamodb-local.

3. Implement a program, in the `build_table.py` file, that, using the `dynamodb` client, creates a table called `ningipoints` with the following characteristics:

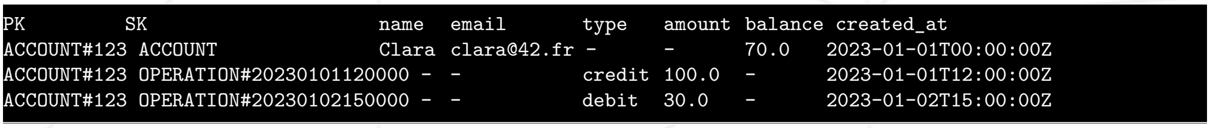
    - **TableName**: ’ningipoints’
    - **AttributeDefinitions**:
        - Must define the fields ’PK’, ’SK’ as Strings
    - **KeySchema**
        - Must configure the ’PK’ field as ’HASH’ (Partition Key)
        - Must configure the ’SK’ field as ’RANGE’ (Sort Key)
    - **BillingMode**: ’PAY_PER_REQUEST’

4. Your program should check if the table does not already exist and wait for the table creation before finishing.

In DynamoDB, we use PK *Partition Key* and SK *Sort Key* as a composite primary key. The PK defines where the data is stored, and the SK allows sorting and querying related items. In this project, we will follow the convention ‘PK = ACCOUNT#<id>‘ and ‘SK = ACCOUNT‘ for the main account item, and ‘SK = OPERATION#<timestamp>‘ for the operations. This approach is common in a single-table design because it is simple, scalable, and easy to query. Only PK and SK need to be defined when creating the table - the other attributes (name, email, amount, etc.) are added freely to each item. The use of the names PK and SK is just a convention adopted for clarity.

You can use the NoSQL database client Workbench for DynamoDB to explore your newly created table.
‘Amazon DynamoDB Launch > Operation Builder > Add Connection‘

Example of how the data will be stored in the table, in the next exercises:



```
PK          SK                        name  email       type   amount balance created_at
ACCOUNT#123 ACCOUNT                   Clara clara@42.fr -      -      70.0    2023-01-01T00:00:00Z
ACCOUNT#123 OPERATION#20230101120000 - -           credit 100.0  -       2023-01-01T12:00:00Z
ACCOUNT#123 OPERATION#20230102150000 - -           debit  30.0   -       2023-01-02T15:00:00Z
```

Chapter II

	Exercise 02
Add Account	
Turn-in directory: *ex*02/	
Files to turn in: `database.py`	
Allowed functions: `No restrictions`	

Use the ‘database.py‘ file created in the previous exercise.

1. Implement the `create_account` function, which inserts a new account in the database, with the prototype:

```
def create_account(account_id: int, name: str, email: str) -> None:
```

   - Creates an account, with `PK='ACCOUNT#<id>'` and `SK='ACCOUNT'`.
   - Uses a `ConditionExpression` to ensure that an exception is raised if an attribute of type `PK` with this same value already exists.

2. Implement the `get_account` function, which retrieves an account from the database, whose `PK='ACCOUNT#<id>'` and `SK='ACCOUNT'`, with the prototype:

```
def get_account(account_id: int) -> dict | None:
```

3. Your functions should behave as follows:

```
?> python3
>>> from database import create_account, get_account
>>> create_account(123, 'clara', 'clara@42sp.org.br')
>>> get_account(123)
{'SK': {'S': 'ACCOUNT'}, 'name': {'S': 'clara'}, 'created_at': {'S': '2025-07-09T15:26:12.959863Z'}, 'PK': {'S': 'ACC
```

Chapter III

	Exercise 03
Add Operation	
Turn-in directory: *ex*03/	
Files to turn in: `database.py`	
Allowed functions: `No restrictions`	

Use the ‘database.py‘ file created in the previous exercise.

In this exercise, you will extend the data structure of the system by adding **credit and debit operations** for existing accounts.

1. Implement the `create_operation` function, which inserts a new operation into the table:

   ```
   def create_operation(account_id: int, type: str, amount: int) -> None:
   ```

2. Implement the `get_operations` function, which retrieves the operations of a given account:

   ```
   def get_operations(account_id: int) -> list[dict]:
   ```

Your ‘create_operation‘ function should update the account’s balance (balance) appropriately, according to the operation type.

Use the query operation, with KeyConditionExpression and ExpressionAttributeValues, combining a fixed PK and SK that starts with 'OPERATION#': ‘PK = :pk AND begins_with(SK, :sk)‘.

Example of expected behavior:

```
?> python3
>>> from database import create_operation, get_operations
>>> create_operation(123, 'credit', 100, '2025-07-09T15:00:00Z')
>>> create_operation(123, 'debit', 50, '2025-07-10T10:30:00Z')
>>> get_operations(123)
[
   {'SK': {'S': 'OPERATION#2025-07-09T15:52:24.683244Z'}, 'created_at': {'S': '2025-07-09T15:52:24.683244Z'}, 'amount': {'N
   {'SK': {'S': 'OPERATION#2025-07-09T15:52:33.694799Z'}, 'created_at': {'S': '2025-07-09T15:52:33.694799Z'}, 'amount': {'N
]
```

Chapter IV

	Exercise 04
Get and Delete	
Turn-in directory: *ex*04/	
Files to turn in: `database.py`	
Allowed functions: `No restrictions`	

Use the ‘database.py‘ file created in the previous exercise.

1. Implement the `delete_account` function, which removes the appropriate account (and all its operations):

```
def delete_account(account_id: int) -> None:
```

2. Implement the `get_accounts` function that returns all accounts (without operations):

```
def get_accounts() -> list[dict]:
```

Chapter V

Validation

- If you have reached this point and rigorously executed the exercises, you can already validate this module with a final result of 80% and already possess the necessary knowledge to move on to the following modules.

- The next exercises make it possible to achieve a result of 100% and 125%, respectively. Evaluate the completion of these exercises considering the ease/difficulty encountered in the previous exercises. Seek a balance between challenging yourself and moving on to the next modules.

Chapter VI

	Exercise 05
Atomic Transactions	
Turn-in directory: *ex*05/	
Files to turn in: `database.py`	
Allowed functions: `No restrictions`	

Currently, creating operations and updating the balance are two separate transactions. This can lead to inconsistencies if one operation fails after the other has succeeded.

1. Update the `create_operation` function to use atomic transactions, ensuring that:
    - The balance is only updated IF the operation is registered.
    - The operation is only registered IF the balance is updated.

> ⚠ `You must use the 'transact_write_items' function of the dynamodb client, and perform both transactions concurrently.`

Chapter VII

Bonus

	Exercise 06
Global Secondary Indexes (GSI)	
Turn-in directory: *ex*06/	
Files to turn in: `build_table.py, database.py`	
Allowed functions: `Standard Library, boto3`	

In this exercise, we will recreate our table, adding a Global Secondary Index - GSI that will allow searching for accounts by email address.

1. Modify the `build_table.py` file to create the `ningipoints` table with a GSI called `EmailIndex` that will have the `email` attribute as its partition key.

2. Implement the `get_account_by_email` function in the `database.py` file, which returns the first account found with the email sent as a parameter:

```
def get_account_by_email(email: str) -> dict | None:
```

Chapter VIII

Peer Review and Submission

- Submit your project to your *Git* repository available on the project page on the intranet.
- Only the work within your repository will be evaluated during the defense. Do not hesitate to check your file and folder names to ensure they are correct.
- At the time of evaluation, the evaluator will go to the workstation of the student to be evaluated to perform the tests. A clone of the repository will be made in a new folder, and these are the files that will be evaluated.