**Design a database for a mobile app with Amazon DynamoDB**

**Project overview** :

In this project, we learn advanced data modeling patterns in [Amazon DynamoDB](https://aws.amazon.com/dynamodb/) while building a DynamoDB-backed mobile application. When using DynamoDB, it is important to consider how we will access our data (our access patterns) before we model our data. To learn these patterns, we build our own data model for an example mobile application that includes a social network. we will learn how to design our own data model in DynamoDB to achieve fast, consistent performance.

Here we are using DynamoDB as my primary database because it is a nosql database and we don’t have to manage the table and datatype.

Here is some basic key point for which I am using It as a application database

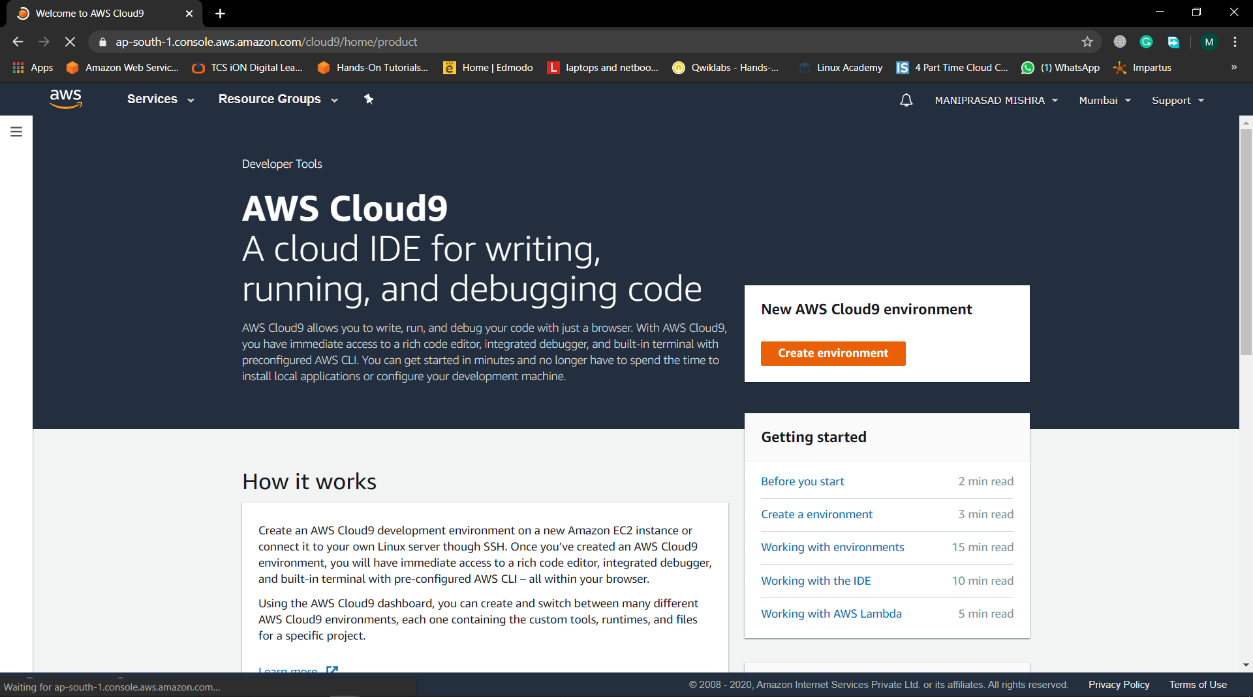
* Scalability: DynamoDB has no performance degradation as we scale to terabytes of data and beyond.
* A simple, REST-based API: DynamoDB is accessible over HTTP(S) and uses AWS Identity and Access Management (IAM) for authentication.
* Fully-managed: we don’t need to manage servers and apply patches to infrastructure. DynamoDB handles upgrades, backups, and other administrative tasks so that we can focus on developing of our application.

**Step 1: Create an AWS account :-**

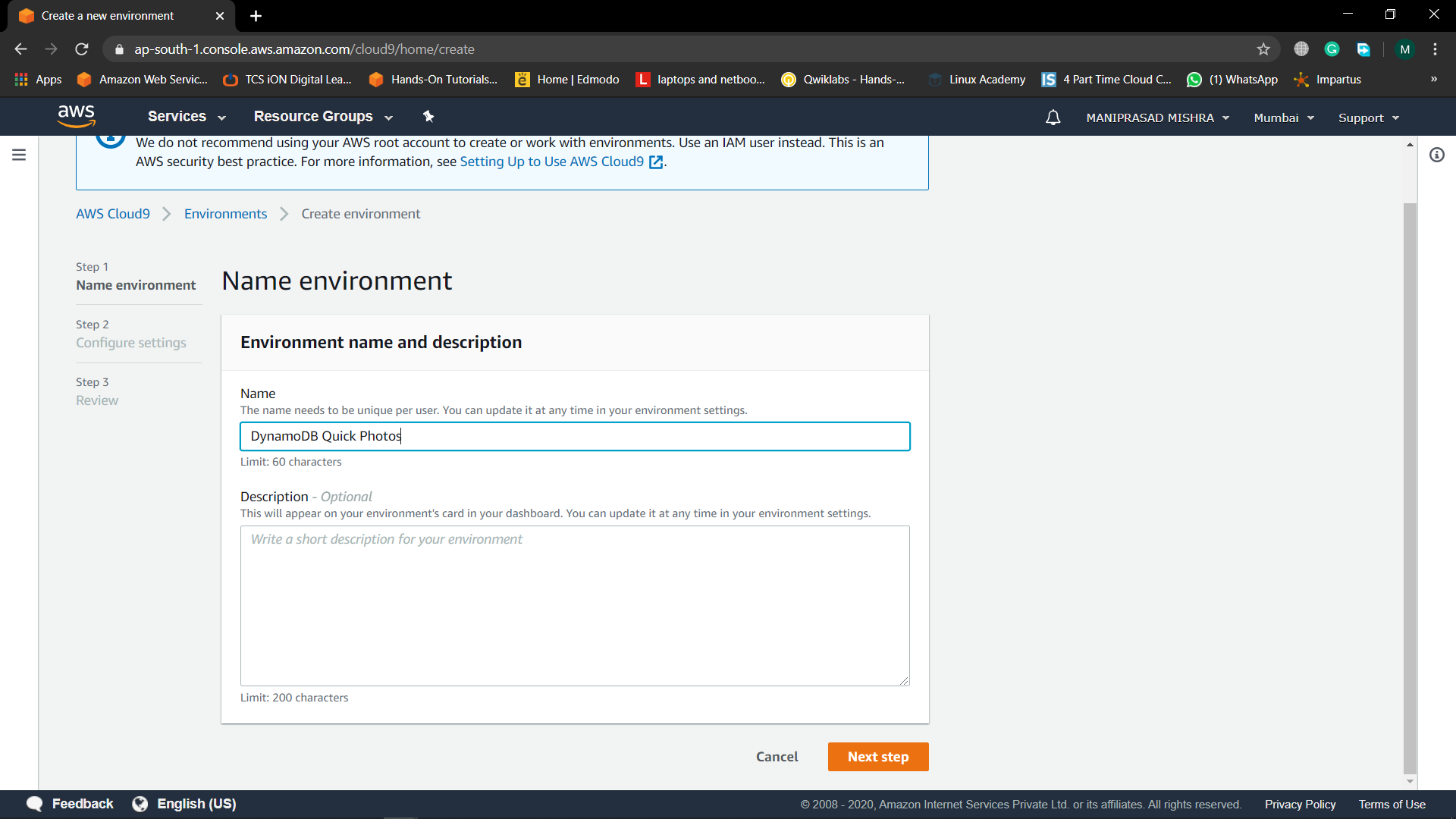
* Login in to aws account and go to service dropdown option and then Search for aws cloude 9 for creating a environment for our project.

**Step 2: Set up your AWS Cloud9 IDE :-**

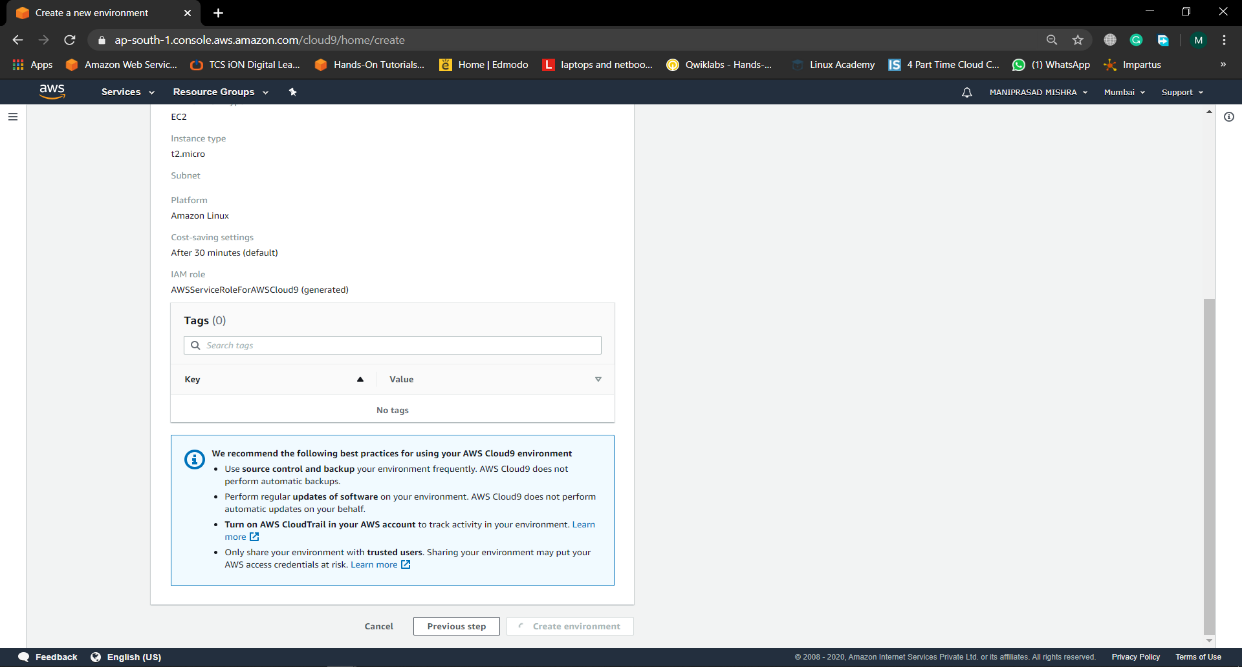
* Choose Create environment.



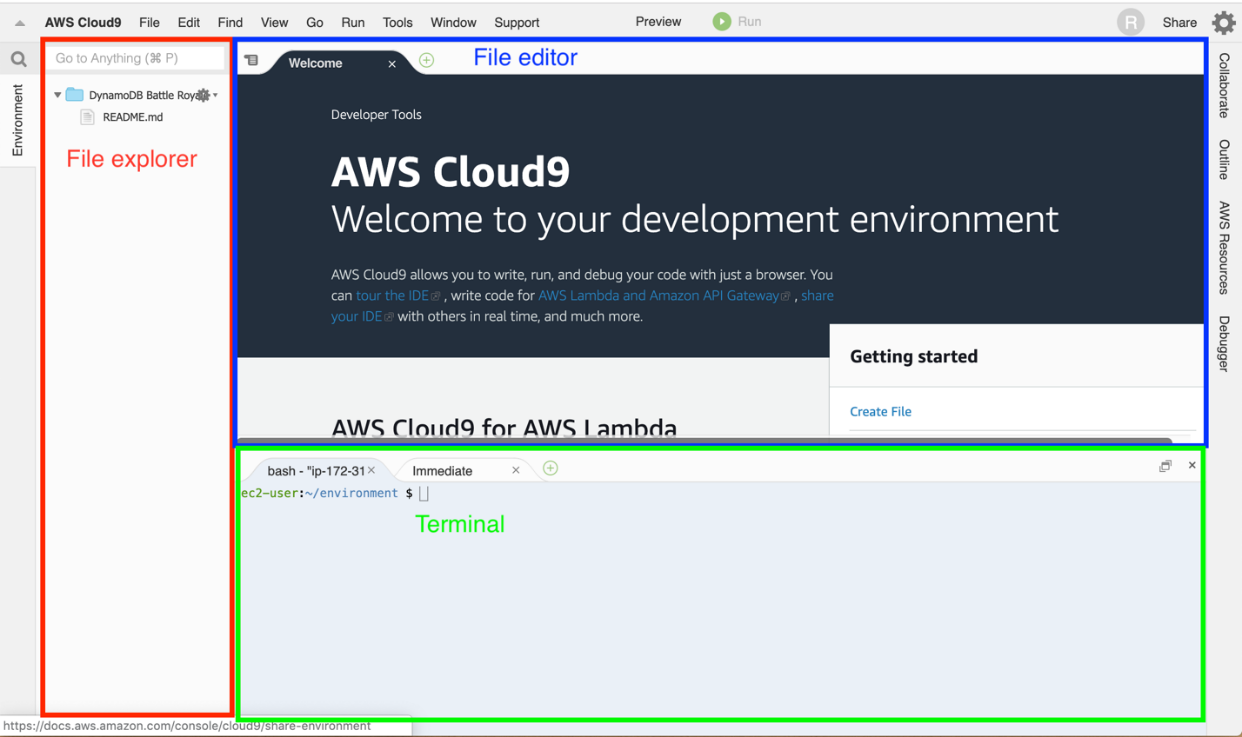
* Then choose any name for the environment.



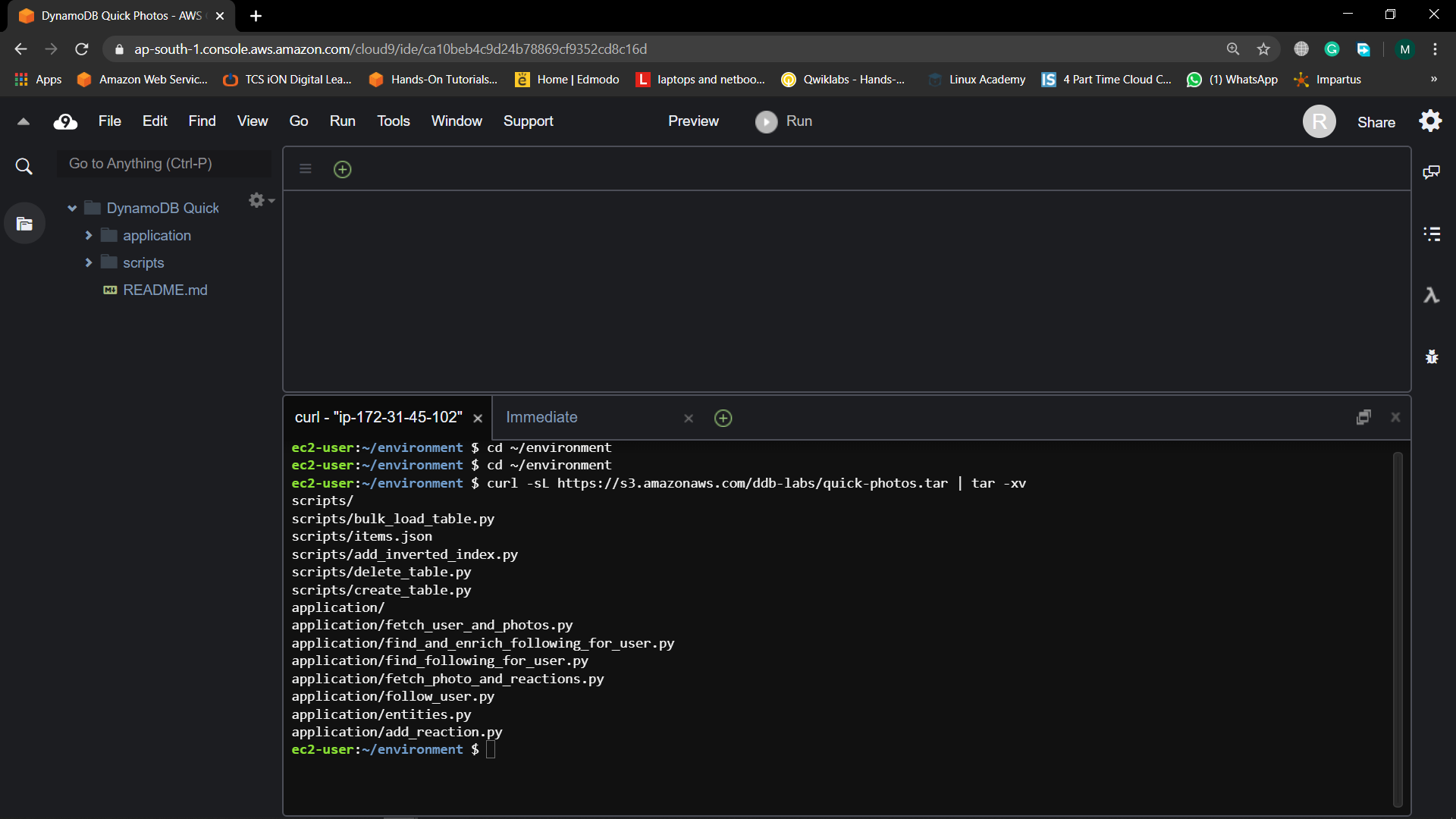
* Choose Next step.
* Leave the Environment settings at their defaults to create a new t2.micro EC2 instance, which will be hibernated after 30 minutes of inactivity.

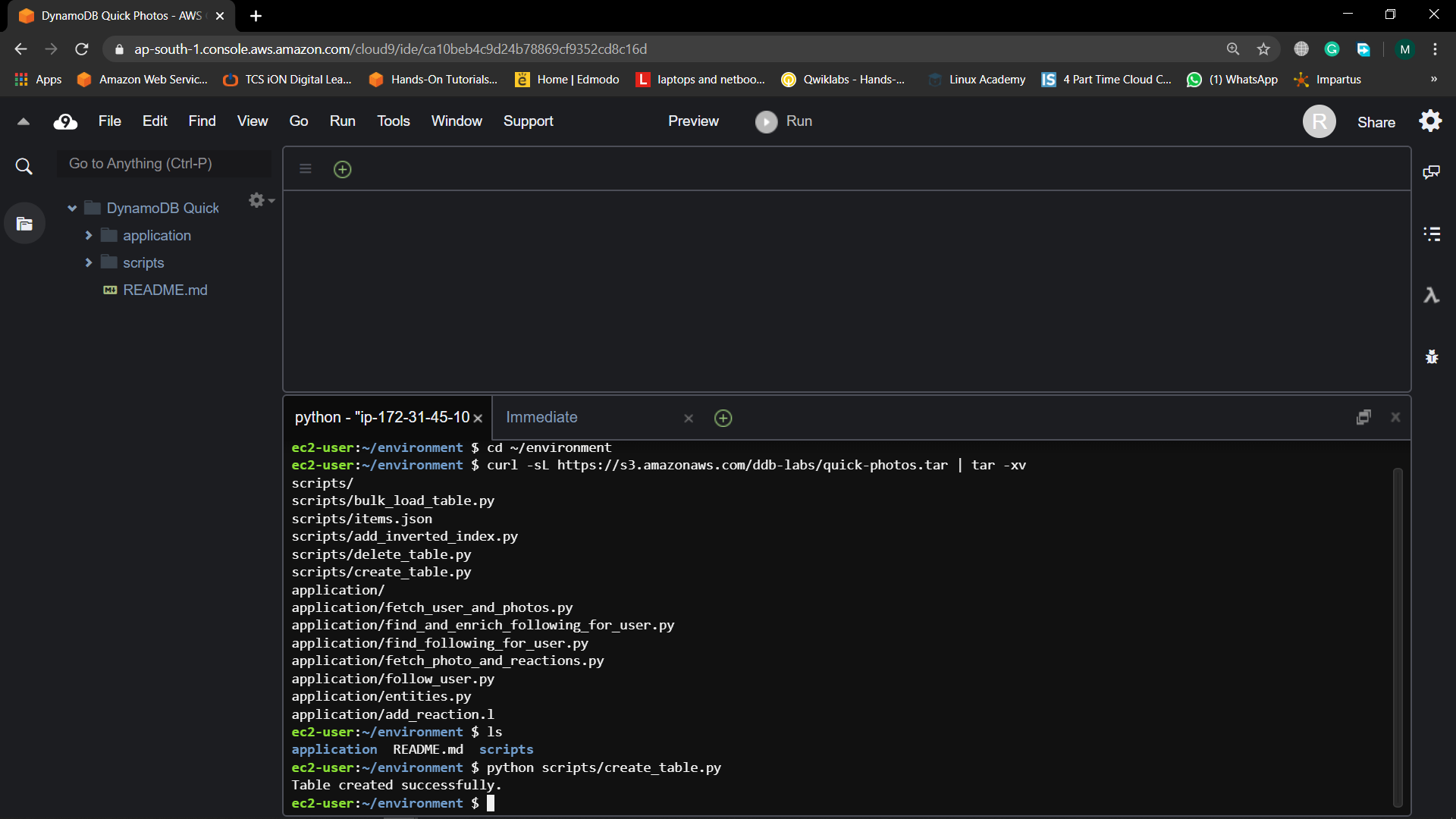


* Choose Next step.
* Review the environment name and settings, and choose Create environment. Your environment will be provisioned and prepared after several minutes.
* When it is ready, your IDE should open with a welcome note.

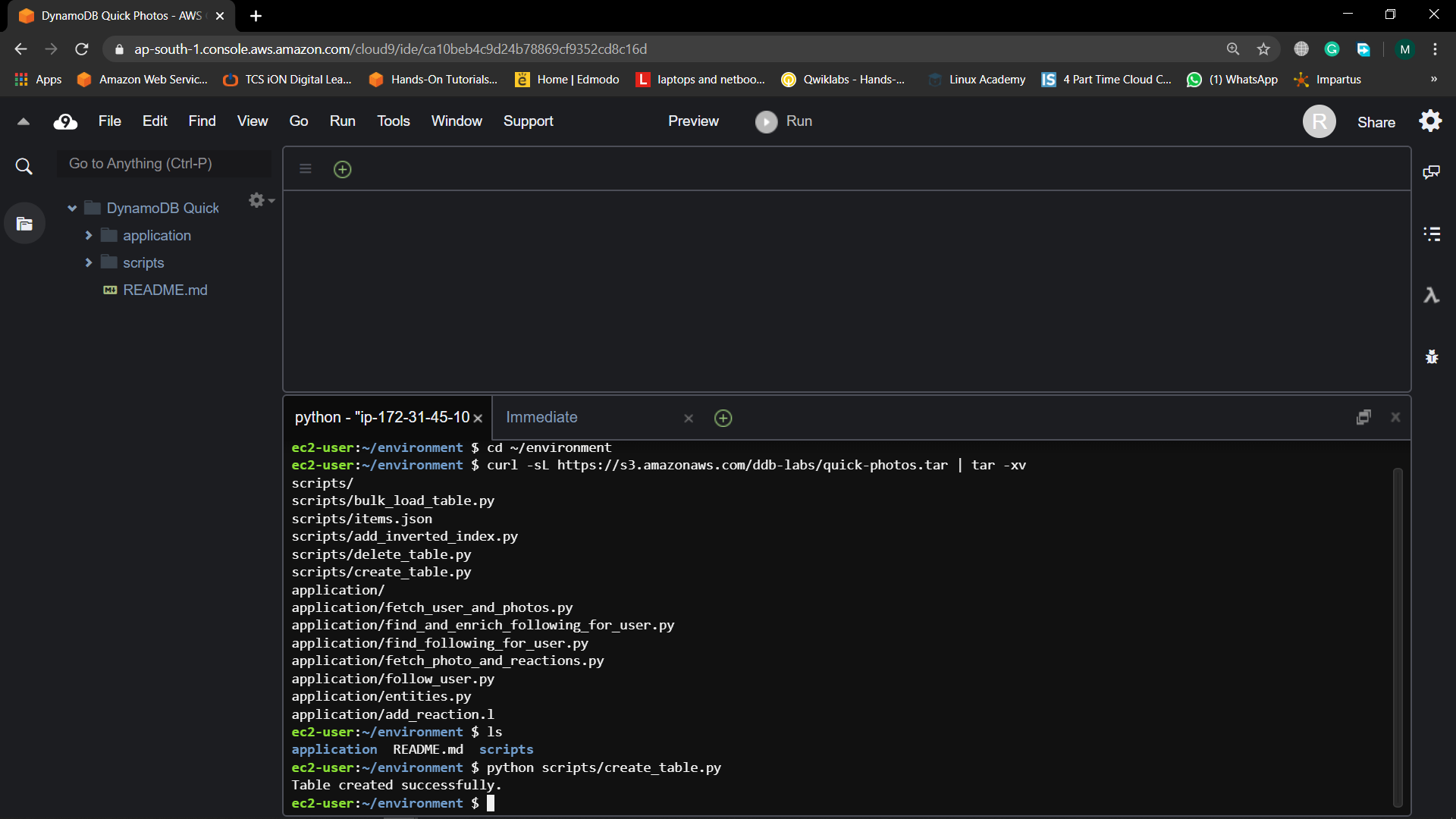


### **Step 3: Download the supporting code :-**

* In this project, we will use Python scripts to interact with the DynamoDB API. Run the following commands in our AWS Cloud9 terminal to download and unpack the module code which is stored inside the S3 bucket created by us.
* cd ~/environment
* It is use to extract the code stored in the S3 bucket.
* curl -sL https://s3.amazonaws.com/ddb-labs/quick-photos.tar | tar -xv



* To view the directories extracted from the S3 bucket we use
* Ls



* There should be Two directories and one readme file present in that machin it should be

application and script.

### **Step 4: Build your entity-relationship diagram as per the app :-**

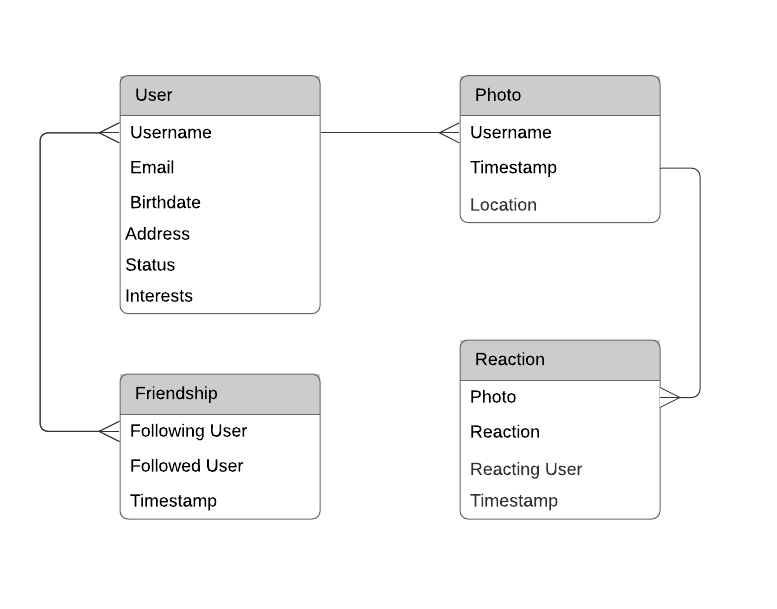
The first step of any data modeling exercise is to build a diagram to show the entities in your application and how they relate to each other.

In our application, we have the following entities:

* *User*
* *Photo*
* *Reaction*
* *Friendship*

A *User*can have many *Photos*, and a *Photo*can have many *Reactions*. Finally, the *Friendship*entity represents a many-to-many relationship between *Users*, as a *User*can follow multiple *Users*and be followed by multiple other *Users*.

With these entities and relationships in mind, our entity-relationship diagram is shown below.



### **Step 5: Consider user profile access patterns :-**

Now that we have our entity-relationship diagram, consider the access patterns around our entities. Let’s start with users.

The users of our mobile application will need to create user profiles. These profiles will include information such as a username, profile picture, location, current status, and interests for a given user.

Users will be able to browse the profile of other users. A user may want to browse the profile of another user to see if the user is interesting to follow or simply to read some background on an existing friend.

Over time, a user will want to update their profile to display a new status or to update their interests as they change.

Based on this information, we have three access patterns:

* Create user profile (*Write*)
* Update user profile (*Write*)
* Get user profile (*Read*)

### **Step 6: Consider photo access patterns :-**

Now, let’s look at the access patterns around photos.

Our mobile application allows users to upload and share photos with their friends, similar to [Instagram](https://www.instagram.com/) or [Snapchat](https://www.snapchat.com/). When users upload a photo, you will need to store information such as the time the photo was uploaded and the location of the file on your content delivery network (CDN).

When users aren’t uploading photos, they will want to browse photos of their friends. If they visit a friend’s profile, they should see the photos for a user with the most recent photos showing first. If they really like a photo, they can ‘react’ to the photo using one of four predefined reactions -- a heart, a smiley face, a thumbs up, or a pair of sunglasses. Viewing a photo should display the current reactions for the photo.

In this section, we have the following access patterns:

* Upload photo for user (*Write*)
* View recent photos for user (*Read*)
* React to a photo (*Write*)
* View photo and reactions (*Read*)

### **Step 7: Friendship access patterns :-**

Finally, let’s consider the access patterns around friendship.

Many popular mobile applications have a social network aspect. You can follow friends, view updates on your friends’ activities, and receive recommendations on other friends you may want to follow.

In your application, a friendship is a one-way relationship, like Twitter. One user can choose to follow another user, and that user may choose to follow the user back. For our application, we will call the users that follow a user “followers”, and we will call the users that a user is following the “followed”.

Based on this information, we have the following access patterns:

* Follow user (*Write*)
* View followers for user (*Read*)
* View followed for user (*Read*)

We have now mapped out all of our access patterns for our mobile application. In the following steps, we’ll implement these access patterns by using DynamoDB.

Note that the planning phase may take a few iterations. Start with a general idea of the access patterns your application needs. Map out the primary key, secondary indexes, and attributes in your table. Go back to the beginning and make sure all of your access patterns are satisfied. Once you are confident the planning phase is complete, then move forward with implementation.

### **Step 8. Design the primary key :-**

Let’s consider the different entities, as suggested in the preceding introduction. In the mobile application, we have the following entities:

* *Users*
* *Photos*
* *Reactions*
* *Friendship*

These entities show three different kinds of data relationships.

First, each user on your application will have a single user profile represented by a *User*entity in your table.

Next, a user will have multiple photos represented in your application, and a photo will have multiple reactions. These are both one-to-many relationships.

Finally, the *Friendship*entity is a representation of a many-to-many relationship. The *Friendship*entity represents when one user is following another user in your application. It is a many-to-many relationship as one user may follow multiple other users, and a user may have multiple followers.

Having a many-to-many mapping is usually an indication that you will want to satisfy two Query patterns, and our application is no exception. On the *Friendship*entity, we have an access pattern that needs to find all users that follow a particular user as well as an access pattern to find all of the users that a given user follows.

Because of this, we’ll use a composite primary key with both a *HASH*and *RANGE*value. The composite primary key will give us the *Query*ability on the *HASH*key to satisfy one of the query patterns we need. In the DynamoDB API specification, the partition key is called *HASH*and the sort key is called *RANGE*, and in this guide we will use the API terminology interchangeably and especially when we discuss the code or DynamoDB JSON wire format.

Note that the one-to-one entity -- *User*-- doesn’t have a natural property for the *RANGE*value. Because it’s a one-to-one mapping, the access patterns will be a basic key-value lookup. Since your table design requires a *RANGE*property, you can provide a filler value for the *RANGE*key.

|  |  |  |
| --- | --- | --- |
| Entity | HASH | RANGE |
| User | USER#<USERNAME> | #METADATA#<USERNAME> |
| Photo | USER#<USERNAME> | PHOTO#<USERNAME>#<TIMESTAMP> |
| Reaction | REACTION#<USERNAME>#<TYPE> | PHOTO#<USERNAME>#<TIMESTAMP> |
| Friendship | USER#<USERNAME> | #FRIEND#<FRIEND\_USERNAME> |

Let’s walk through the preceding table.

First, for the User entity, the *HASH*value will be *USER#<USERNAME>*. Notice that you’re using a prefix to identify the entity and prevent any possible collisions across entity types.

For the *RANGE*value on the User entity, we’re using a static prefix of *#METADATA#* followed by the username value. For the *RANGE*value, it’s important that you have a value that is known, such as the username. This allows for single-item actions such as *GetItem*, *PutItem*, and *DeleteItem*.

However, you also want a *RANGE*value with different values across different User entities to enable [even partitioning](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.Partitions.html) if you use this column as a *HASH*key for an index. For that reason, you append the username to the *RANGE*key.

Second, the Photo entity is a child entity of a particular User entity. The main access pattern for photos is to retrieve photos for a user ordered by date. Whenever you need something ordered by a particular property, you will need to include that property in your *RANGE*key to allow for sorting. For the Photo entity, use the same *HASH*key as the User entity, which will allow you to retrieve both a user profile and the user’s photos in a single request. For the *RANGE*key, use *PHOTO#<USERNAME>#<TIMESTAMP>* to uniquely identify a photo in your table.

Third, the Reaction entity is a child entity of a particular Photo entity. There is a one-to-many relationship to the Photo entity and thus will use similar reasoning as with the Photo entity. In the next module, you will see how to retrieve a photo and all of its reactions in a single query using a secondary index. For now, note that the *RANGE*key for a Reaction entity is the same pattern as the *RANGE*key for a Photo entity. For the *HASH*key, we use the username of the user that is creating the reaction as well as the type of reaction applied. Appending the type of reaction allows a user to add multiple reaction types to a single photo.

Finally, the Friendship entity uses the same *HASH*key as the User entity. This will allow you to fetch both the metadata for a user plus all of the user’s followers in a single query. The *RANGE*key for a Friendship entity is *#FRIEND#<FRIEND\_USERNAME>*. In Step 4 below, you will learn why to prepend the Friendship entity’s *RANGE*key with a “#”.

* In the next step, we create a table with this primary key design.

### **Step 9: Create a table**

Now that we have designed the primary key, let’s create a table.

The code you downloaded in Step 3 of Module 1 includes a Python script in the scripts/ directory named create\_table.py. The Python script’s contents are as follows:

* Then we use the under written code to Create the table for the application Database

Python script in the scripts/ directory named create\_table.py. The Python script’s contents are as follows:

import boto3

dynamodb = boto3.client('dynamodb')

try:

dynamodb.create\_table(

TableName='quick-photos',

AttributeDefinitions=[

{

"AttributeName": "PK",

"AttributeType": "S"

},

{

"AttributeName": "SK",

"AttributeType": "S"

}

],

KeySchema=[

{

"AttributeName": "PK",

"KeyType": "HASH"

},

{

"AttributeName": "SK",

"KeyType": "RANGE"

}

],

ProvisionedThroughput={

"ReadCapacityUnits": 5,

"WriteCapacityUnits": 5

}

)

print("Table created successfully.")

except Exception as e:

print("Could not create table. Error:")

print(e)

The preceding script uses the [CreateTable](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_CreateTable.html) operation using [Boto 3](https://boto3.amazonaws.com/v1/documentation/api/latest/index.html), the AWS SDK for Python. The operation declares two attribute definitions, which are typed attributes to be used in the primary key. Though DynamoDB is schemaless, you must declare the names and types of attributes that are used for primary keys. The attributes must be included on every item that is written to the table and thus must be specified as you are creating a table.

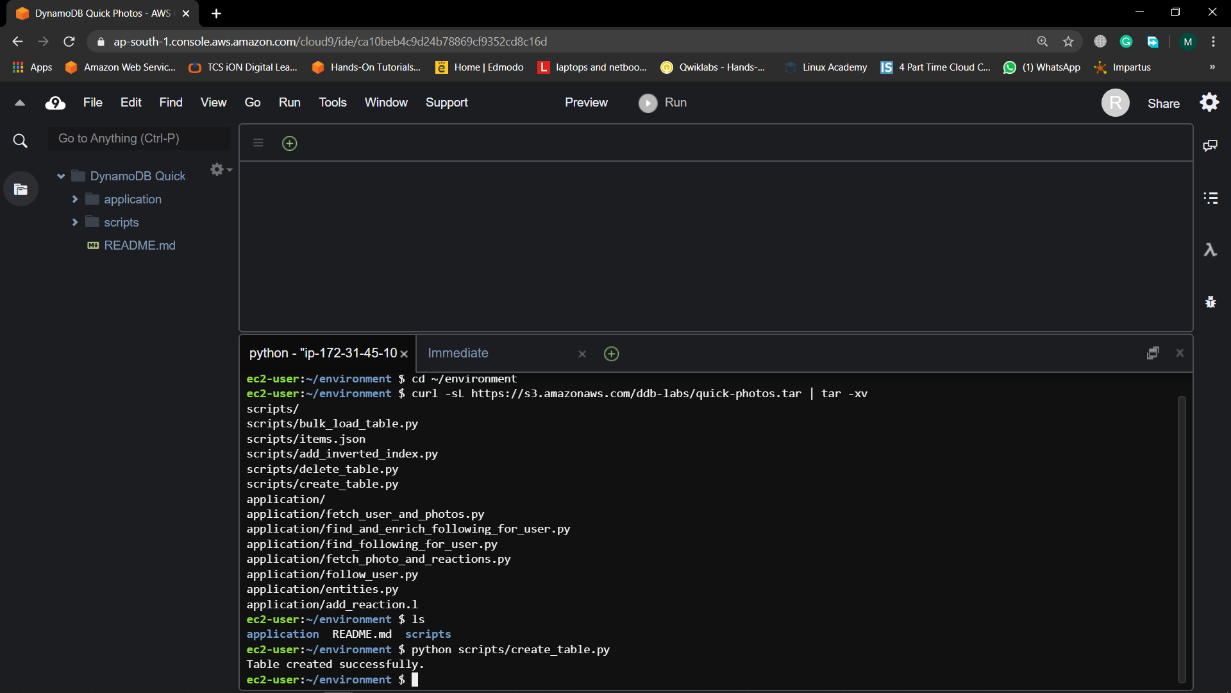
Because you’re storing different entities in a single table, your primary key can’t use attribute names like *UserId*. The attribute means something different based on the type of entity being stored. For example, the primary key for a user might be its *USERNAME*, and the primary key for a reaction might be its *TYPE*. Accordingly, we use generic names for the attributes -- PK (for partition key) and SK (for sort key).

After configuring the attributes in the key schema, we specify the [provisioned throughput](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html) for the table. DynamoDB has two capacity modes: provisioned and on-demand. In provisioned capacity mode, you specify exactly the amount of read and write throughput you want. You pay for this capacity whether you use it or not.  
  
In DynamoDB on-demand capacity mode, you can pay per request. The cost per request is slightly higher than if you were to use provisioned throughput fully, but you don’t have to spend time doing capacity planning or worrying about getting throttled. On-demand mode works great for spiky or unpredictable workloads. We’re using provisioned capacity mode in this lab because it fits within the DynamoDB free tier.

To creation of the table, run the Python script with the following command

python scripts/create\_table.py

The script should return this message: “Table created successfully.”



### **Step 10: Bulk-load data into the table**

In this step, we’re going to bulk load some data into the DynamoDB table we created in the preceding step. This means that in succeeding steps, we will have sample data to use.

In the scripts/ directory, there is a file called items.json. This file contains 967 sample items that were randomly generated for our project. These items include *User*, *Photo*, *Friendship*, and *Reaction*entities. You can open the file if you want to see some of the example data.

The scripts/ directory also has a file called bulk\_load\_table.py that will read the items in items.json and bulk write them to the DynamoDB table. The contents of that file are as follows in this code:

import json

import boto3

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('quick-photos')

items = []

with open('scripts/items.json', 'r') as f:

for row in f:

items.append(json.loads(row))

with table.batch\_writer() as batch:

for item in items:

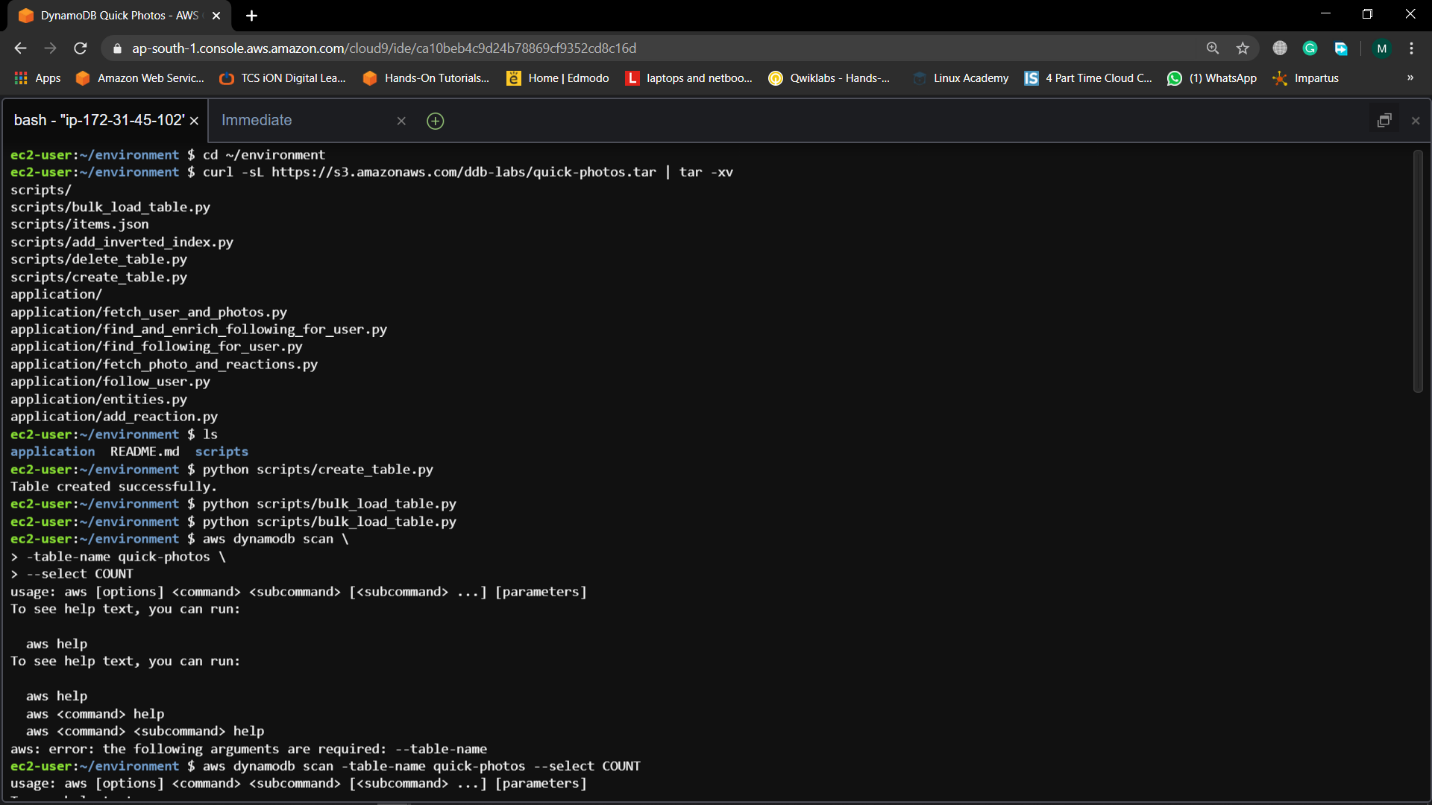
batch.put\_item(Item=item)

In this script, rather than using the low-level client in Boto 3, we use a higher-level [Resource object](https://boto3.amazonaws.com/v1/documentation/api/latest/guide/migration.html#resource-objects). *Resource* objects provide an easier interface for using the AWS APIs. The *Resource* object is useful in this situation because it batches our requests. The [BatchWriteItem](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchWriteItem.html" \t "_blank) [API](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchWriteItem.html) operation accepts up to 25 items in a single request. The *Resource*object will handle that batching for us rather than making us chop up our data into requests of 25 items or less.

The code is stored inside the S3 bucket and the data can be extracted from the S3 by using the underwritten small command.

python scripts/bulk\_load\_table.py

It will take some second to upload the Bulk data to the database.

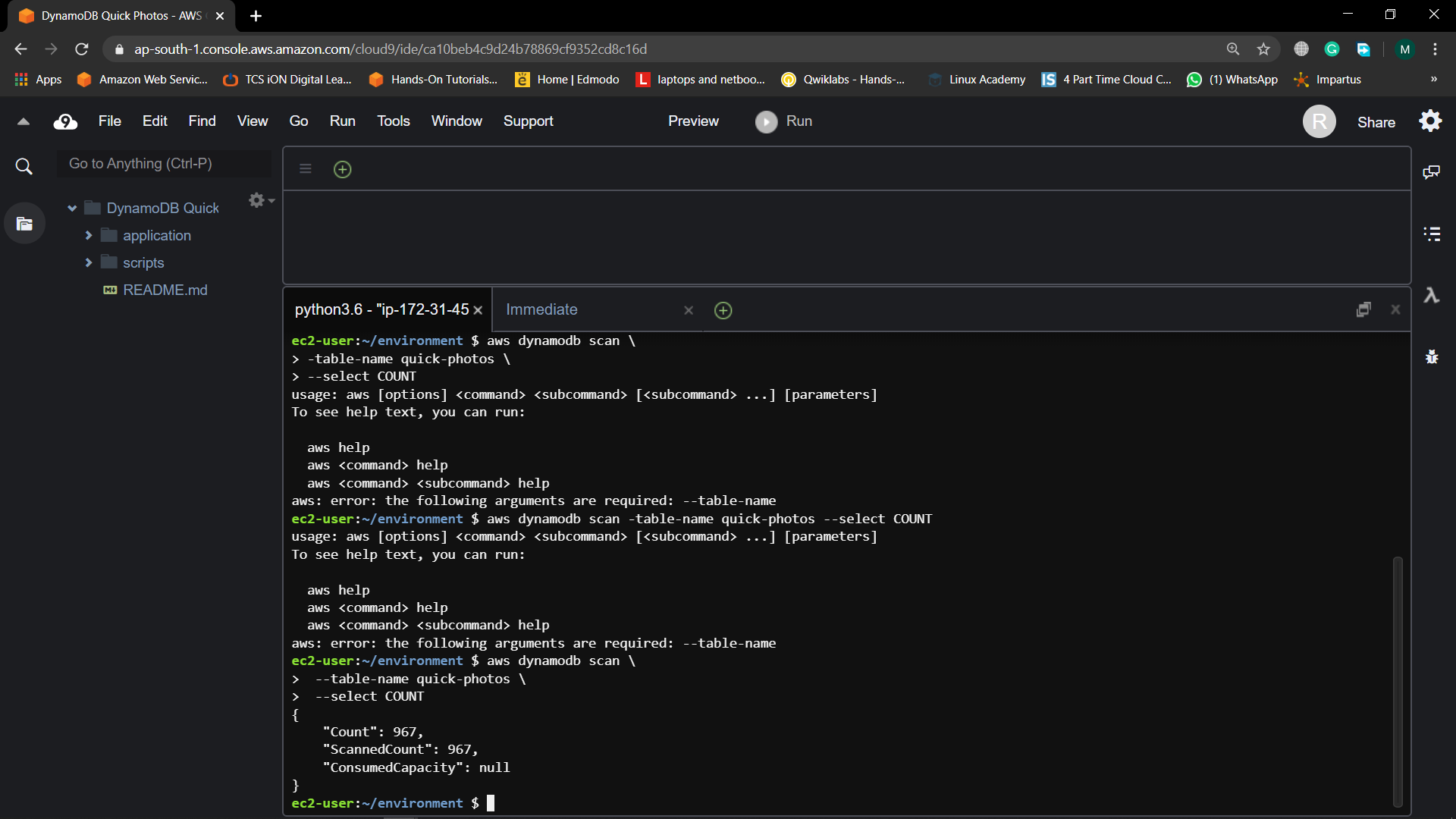


When You can ensure that all your data was loading by running a *Scan*operation and returning the count.

Run the underwritten command to use the AWS CLI to get the count:

aws dynamodb scan \--table-name quick-photos \--select COUNT

This should display the Scanned results.



{

"Count": 967,

"ScannedCount": 967,

"ConsumedCapacity": null

}

You should see a *Count*of 967, indicating all of your items were loading successfully.

### **Step 11: Retrieve multiple entity types in a single request :-**

As we said in the previous steps, you should optimize DynamoDB tables for the number of requests it receives. We also mentioned that DynamoDB does not have joins that a relational database has. Instead, you design your table to allow for join-like behavior in your requests.

In this step, we’ll see how to retrieve multiple entity types in a single request. In our application, we may want to fetch information about a user. This would include all of the information in the user’s profile on the *User*entity as well as all of the photos that have been uploaded by a user.

This request spans two entity types -- the *User*entity and the *Photo*entity. However, this doesn’t mean we need to make multiple requests.

In the code you downloaded, there is a file in the application/ directory called fetch\_user\_and\_photos.py. This script shows how you can structure your code to retrieve both a *User*entity and the *Photo*entities that were uploaded by the user in a single request.

The underwritten code composes the fetch\_user\_and\_photos.py script

import boto3

from entities import User, Photo

dynamodb = boto3.client('dynamodb')

USER = "jacksonjason"

def fetch\_user\_and\_photos(username):

resp = dynamodb.query(

TableName='quick-photos',

KeyConditionExpression="PK = :pk AND SK BETWEEN :metadata AND :photos",

ExpressionAttributeValues={

":pk": { "S": "USER#{}".format(username) },

":metadata": { "S": "#METADATA#{}".format(username) },

":photos": { "S": "PHOTO$" },

},

ScanIndexForward=True

)

user = User(resp['Items'][0])

user.photos = [Photo(item) for item in resp['Items'][1:]]

return user

user = fetch\_user\_and\_photos(USER)

print(user)

for photo in user.photos:

print(photo)

At the top, we import the Boto 3 library and some simple classes to represent the objects in our application code. You can see the definitions for those entities in the application/entities.py file if you’re interested.

The real work is happening in the *fetch\_user\_and\_photos*function that’s defined in the module. This is similar to a function you would define in your application to be used by any endpoints that need this data.

In this function, you first make a Query request to DynamoDB. The Query specifies a *HASH*key of *USER#<Username>* to isolate the returned items to a particular user.

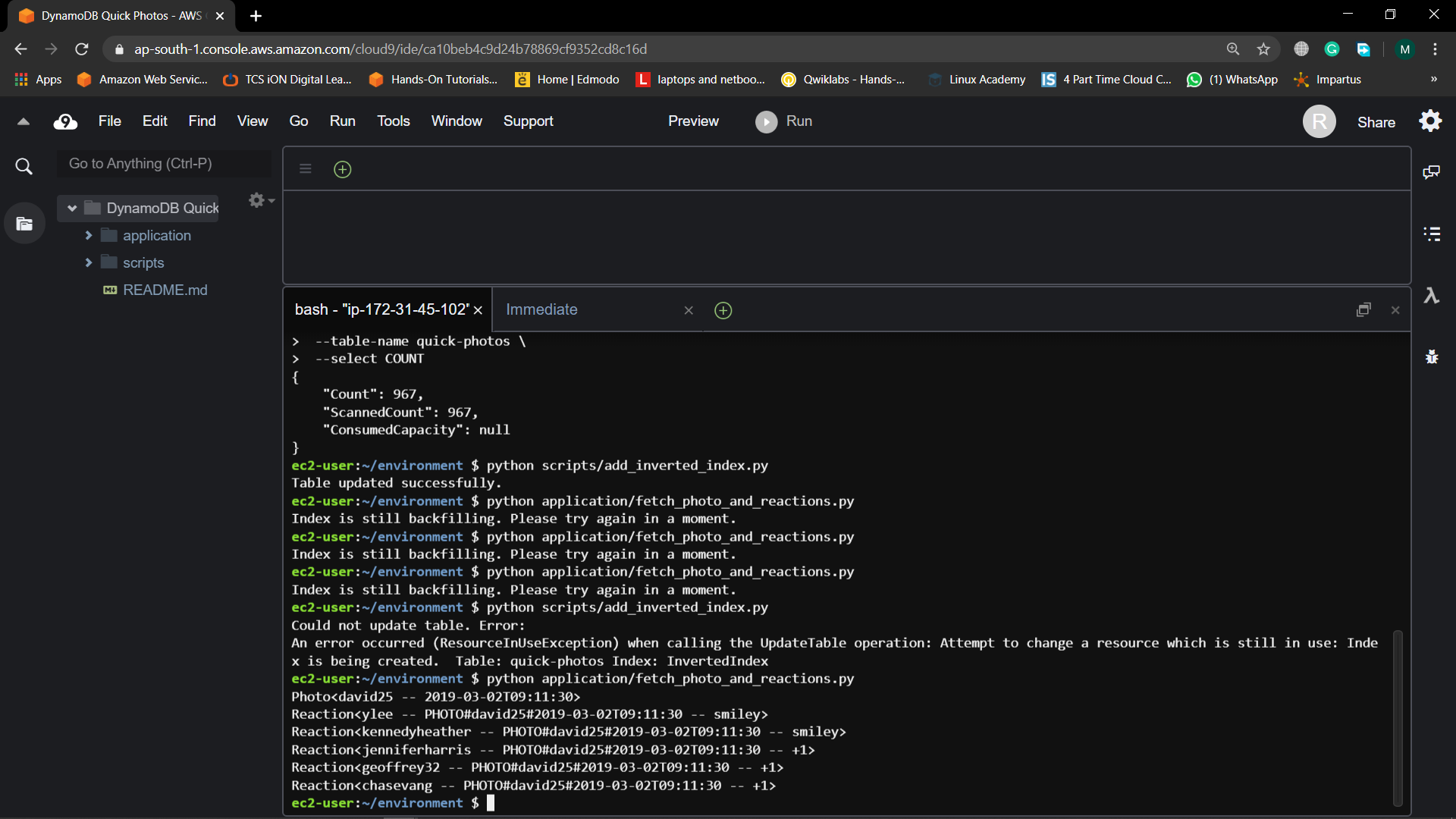
Then, the Query specifies a *RANGE*key condition expression that is between *#METADATA#<Username>* and *PHOTO$*. This Query will return a User entity, as its sort key is*#METADATA#<Username>*, as well as all of the Photo entities for this user, whose sort keys start with *PHOTO#*. Sort keys of the String type are sorted by ASCII character codes. The dollar sign ($) comes directly after the pound sign (#) in [ASCII](http://support.ecisolutions.com/doc-ddms/help/reportsmenu/ascii_sort_order_chart.htm), so this ensures that we will get all of the *Photo*entities.

Once we receive a response, we then assemble our items into objects known by our application. We know that the first item returned will be our *User*entity, so we create a *User*object from the item. For the remaining items, we create a *Photo*object for each one and then attach the array of users to the User object.

The end of the script shows the usage of the function and prints out the resulting objects. You can run the script in your terminal with the following command.

Using this fetch\_user\_and\_photos.py we can fetch the userdata stored in the database

It should print the *User*object and all *Photo*objects to the console:



User<jacksonjason -- John Perry>

Photo<jacksonjason -- 2018-05-30T15:42:38>

Photo<jacksonjason -- 2018-06-09T13:49:13>

Photo<jacksonjason -- 2018-06-26T03:59:33>

Photo<jacksonjason -- 2018-07-14T10:21:01>

Photo<jacksonjason -- 2018-10-06T22:29:39>

Photo<jacksonjason -- 2018-11-13T08:23:00>

Photo<jacksonjason -- 2018-11-18T15:37:05>

Photo<jacksonjason -- 2018-11-26T22:27:44>

Photo<jacksonjason -- 2019-01-02T05:09:04>

Photo<jacksonjason -- 2019-01-23T12:43:33>

Photo<jacksonjason -- 2019-03-03T02:00:01>

Photo<jacksonjason -- 2019-03-03T18:20:10>

Photo<jacksonjason -- 2019-03-11T15:18:22>

Photo<jacksonjason -- 2019-03-30T02:28:42>

Photo<jacksonjason -- 2019-04-14T21:52:36>

In these steps, we designed a primary key and created a table. Then, we bulk-loaded data into the table and saw how to query for multiple entity types in a single request.

With our current primary key design, we are able to satisfy the following access patterns:

* Create user profile (*Write*)
* Update user profile (*Write*)
* Get user profile (*Read*)
* Upload photo (*Write*)
* View photos for User (*Read*)
* View friends for a user (*Read*)

In the next module, we will add a secondary index and learn about the inverted index technique. Secondary indexes allow you to support additional access patterns on your DynamoDB table.

### **Step 12: Create a secondary index :-**

To create a secondary index, we have to specify the primary key of the index, just like when you were creating a table. Note that the primary key for a global secondary index does not have to be unique. DynamoDB then copies your items into the index based on the attributes specified, and you can query it just like your table.

An inverted index is a common pattern in DynamoDB where you create a secondary index that is the inverse of your table’s primary key. The HASH key for your table is specified as the *RANGE*key in your secondary index, and the *RANGE*key for your table is specified as the HASH key in your secondary index.

Creating a secondary index is similar to creating a table. In the code you downloaded, there’s a file in the scripts/ directory named add\_inverted\_index.py. The contents of that file are shown below.

import boto3

dynamodb = boto3.client('dynamodb')

try:

dynamodb.update\_table(

TableName='quick-photos',

AttributeDefinitions=[

{

"AttributeName": "PK",

"AttributeType": "S"

},

{

"AttributeName": "SK",

"AttributeType": "S"

}

],

GlobalSecondaryIndexUpdates=[

{

"Create": {

"IndexName": "InvertedIndex",

"KeySchema": [

{

"AttributeName": "SK",

"KeyType": "HASH"

},

{

"AttributeName": "PK",

"KeyType": "RANGE"

}

],

"Projection": {

"ProjectionType": "ALL"

},

"ProvisionedThroughput": {

"ReadCapacityUnits": 5,

"WriteCapacityUnits": 5

}

}

}

],

)

print("Table updated successfully.")

except Exception as e:

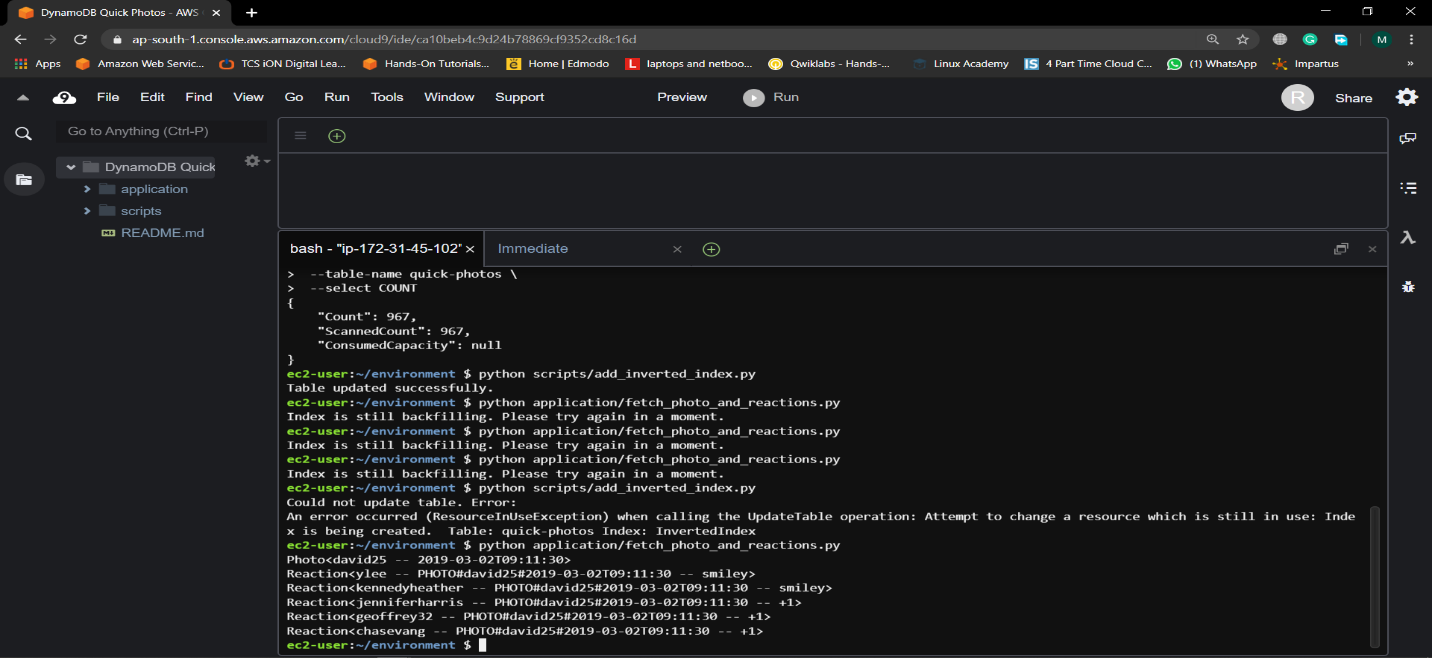
print("Could not update table. Error:")

print(e)

**python scripts/add\_inverted\_index.py**

This highlighted command will create a secondary index inside the database.

It will show the following message in the console: “Table updated successfully.”



### **Step 13: Query the inverted index to find a photo’s reactions**

Now that we have configured the secondary index, let’s use it to satisfy some of the access patterns.

To use a secondary index, you only have two API calls available -- [Query](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html) and [Scan](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Scan.html). With *Query*, you must specify the *HASH*key, and it returns a targeted result. With Scan, you don’t specify a *HASH*key, and the operation runs across your entire table. Scans are discouraged in DynamoDB except in specific circumstances because they access every item in your database. If you have a significant amount of data in your table, scanning can take a very long time

We can use the *Query*API against our secondary index to find all reactions on a particular photo. Like you saw in the previous module, you can use this query to retrieve two types of entities in a single command. In this query, you can retrieve both a photo and its reactions.

From the S3 bucket we have to extract Fetch\_photo\_and\_reactions.py file from the application directory. The contents of this script are shown below.

import boto3

from entities import Photo, Reaction

dynamodb = boto3.client('dynamodb')

USER = "david25"

TIMESTAMP = '2019-03-02T09:11:30'

def fetch\_photo\_and\_reactions(username, timestamp):

try:

resp = dynamodb.query(

TableName='quick-photos',

IndexName='InvertedIndex',

KeyConditionExpression="SK = :sk AND PK BETWEEN :reactions AND :user",

ExpressionAttributeValues={

":sk": { "S": "PHOTO#{}#{}".format(username, timestamp) },

":user": { "S": "USER$" },

":reactions": { "S": "REACTION#" },

},

ScanIndexForward=True

)

except Exception as e:

print("Index is still backfilling. Please try again in a moment.")

return False

items = resp['Items']

items.reverse()

photo = Photo(items[0])

photo.reactions = [Reaction(item) for item in items[1:]]

return photo

photo = fetch\_photo\_and\_reactions(USER, TIMESTAMP)

if photo:

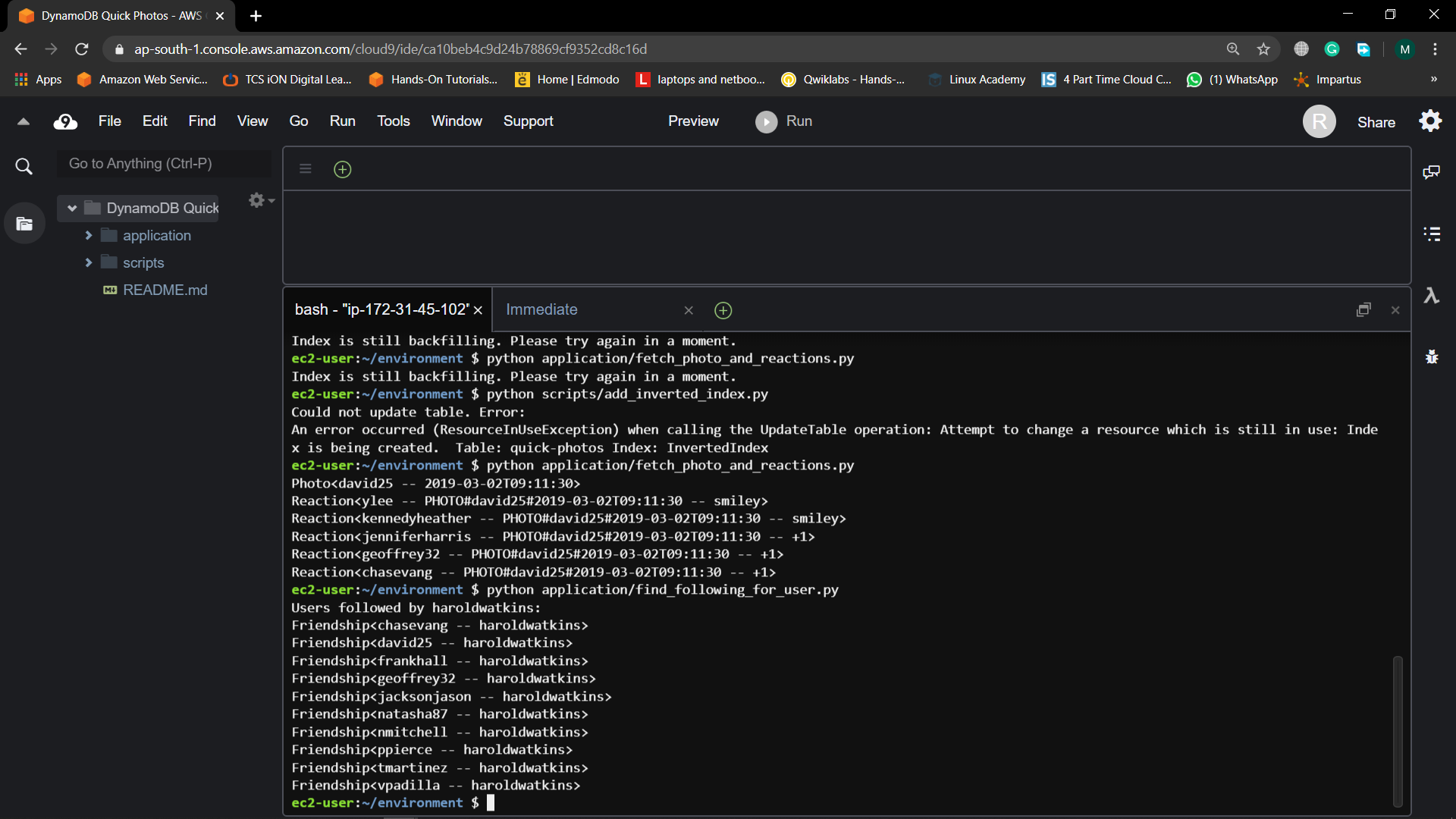
print(photo)

for reaction in photo.reactions:

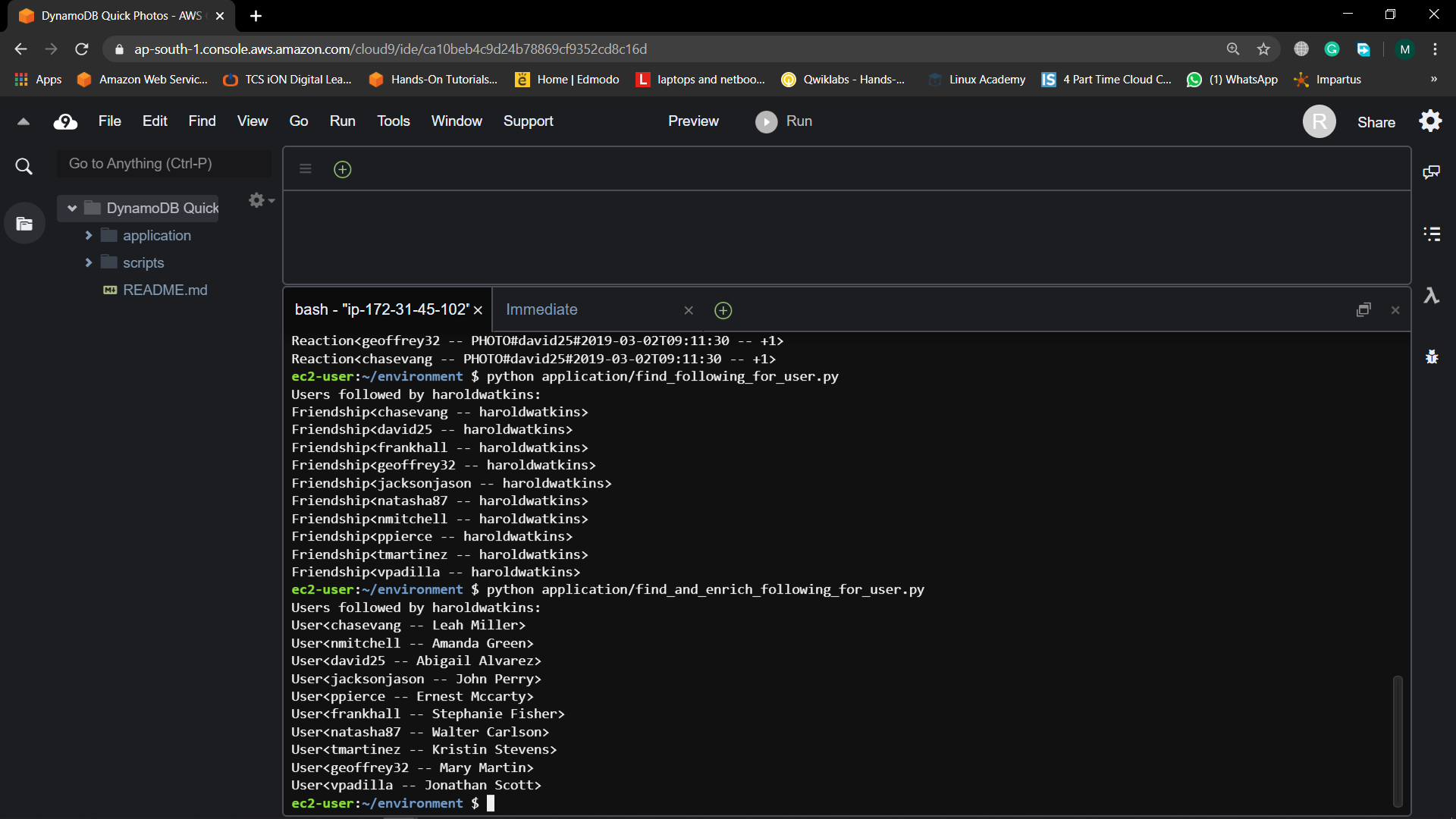
print(reaction)

The *fetch\_photo\_and\_reactions*function is similar to a function you would have in your application. The function accepts a username and timestamp and makes a query against the *InvertedIndex*to find the photo and reactions for the photo. Then it assembles the returned items into a *Photo*entity and multiple *Reaction*entities that can be used in your application. We have to use this command to do it properly in the CLI.

python application/fetch\_photo\_and\_reactions.py



We should see output a photo and its five reactions.



### **Step 14: Find followed users :-**

In the previous step, you saw how to use an inverted index to fetch a one-to-many relationship for an entity that was itself the subject of a one-to-many relationship. In this step, you will use the inverted index to fetch the “other” side of a many-to-many relationship.

The primary key in the table is allows you to find all of the followers of a particular user, but it won’t let you find all the users that someone is following. With the inverted index, it’s flipped -- you can find all the users followed by a particular user.

In the code we downloaded, there is a file in the application/ directory called find\_following\_for\_user.py

The file contains this code

import boto3

from entities import Friendship

dynamodb = boto3.client('dynamodb')

USERNAME = "haroldwatkins"

def find\_following\_for\_user(username):

resp = dynamodb.query(

TableName='quick-photos',

IndexName='InvertedIndex',

KeyConditionExpression="SK = :sk",

ExpressionAttributeValues={

":sk": { "S": "#FRIEND#{}".format(username) }

},

ScanIndexForward=True

)

return [Friendship(item) for item in resp['Items']]

follows = find\_following\_for\_user(USERNAME)

print("Users followed by {}:".format(USERNAME))

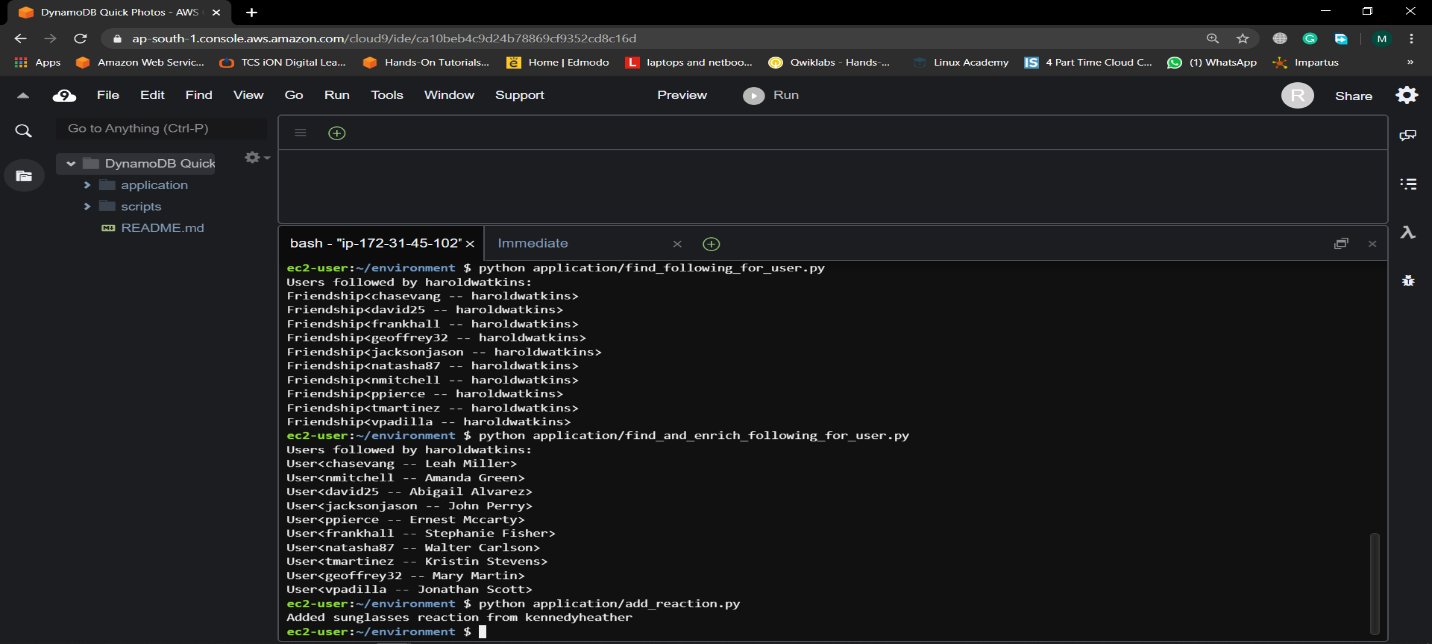
for follow in follows:

print(follow)

The *find\_following\_for\_user*function is similar to a function you would have in your application. The function accepts a username for whom you want to find the followed users. The function then queries the inverted index to find all *Friendship*entities where the following user is the given username.

Run the script by running the following command in your terminal.

python application/find\_following\_for\_user.py



In this cases, we added a secondary index to our table using the inverted index pattern. This satisfied two additional access patterns:

* View photo and reactions (*Read*)
* View followed for user (*Read*)

When retrieving all followed users for a user, we saw a problem that each Friendship entity was missing information about the followed user. In the next module, we will see how to use partial normalization to help with this access pattern.

### **Step 15: Use partial normalization to find followed users :-**

In this step, you will see how to find followed users. These are the users that a particular user is following in your application. You will also see how to retrieve all of the data about the users being followed.

As noted in the introduction to this module, you may want to use a partial normalization technique around friendships and users. Rather than storing the full information about each user in the *Friendship*entity, you can use the *BatchGetItem*API to retrieve information about a user in a *Friendship* entity.

In the code you downloaded, there is a file in the application/ directory called find\_and\_enrich\_following\_for\_user.py. The contents of this script are shown below.

import boto3

from entities import User

dynamodb = boto3.client('dynamodb')

USERNAME = "haroldwatkins"

def find\_and\_enrich\_following\_for\_user(username):

friend\_value = "#FRIEND#{}".format(username)

resp = dynamodb.query(

TableName='quick-photos',

IndexName='InvertedIndex',

KeyConditionExpression="SK = :sk",

ExpressionAttributeValues={":sk": {"S": friend\_value}},

ScanIndexForward=True

)

keys = [

{

"PK": {"S": "USER#{}".format(item["followedUser"]["S"])},

"SK": {"S": "#METADATA#{}".format(item["followedUser"]["S"])},

}

for item in resp["Items"]

]

friends = dynamodb.batch\_get\_item(

RequestItems={

"quick-photos": {

"Keys": keys

}

}

)

enriched\_friends = [User(item) for item in friends['Responses']['quick-photos']]

return enriched\_friends

follows = find\_and\_enrich\_following\_for\_user(USERNAME)

print("Users followed by {}:".format(USERNAME))

for follow in follows:

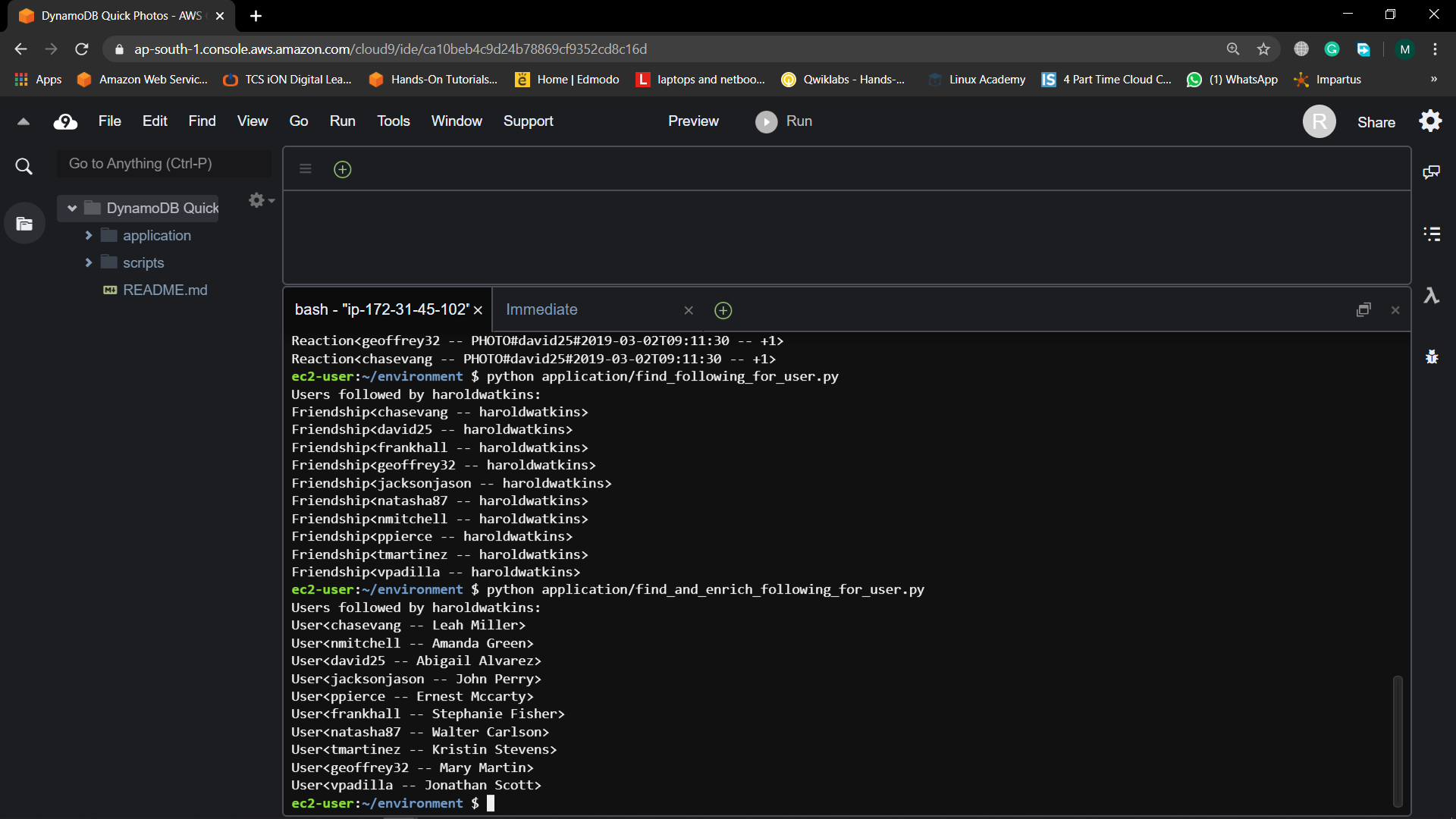
print(follow)

The *find\_and\_enrich\_following\_for\_user*function is similar to the *find\_follower\_for\_user*function you used in the last module. The function accepts a username for whom you want to find the followed users. The function first makes a *Query*request using the inverted index to find all of the users that the given username is following. It then assembles a *BatchGetItem*to fetch the full *User*entity for each of the followed users and returns those entities.

This results in two requests to DynamoDB, rather than the ideal of one. However, it’s satisfying a fairly complex access pattern, and it avoids the need to constantly update Friendship entities every time a user profile is updated. This partial normalization can be a great tool for your modeling needs.

Execute the script by running the following command in your terminal.

python application/find\_and\_enrich\_following\_for\_user.py



Note that here we are now dealing with *User*entities rather than *Friendship* entities. The User entity will have the most complete, up-to-date information about your user. While it took two requests to get there, it may still be a better option than full denormalization and the data integrity issues that result from it.

In these parts, we saw how partial normalization and the BatchGetItem API call can be helpful to maintain data integrity across objects while still keeping query requests low.

In the next parsts, we’ll use DynamoDB transactions as we add a reaction to a photo or follow a user.

### **Step 16: React to a photo:**

The first access pattern we will address in this module is reacting to a photo.

When adding a user’s reaction to a photo, we need to do a few things:

* Confirm that the user has not already used this reaction type on this photo
* Create a new *Reaction*entity to store the reaction
* Increment the proper reaction type in the *reactions*property on the *Photo*entity so that we can display the reaction details on a photo

Note that this requires write actions across two different items -- the existing *Photo*entity and the new Reaction entity -- as well as conditional logic for one of the items. This is the kind of operation that is a perfect fit for DynamoDB transactions.

In the code you downloaded, there is a script in the application/ directory called add\_reaction.py that includes a function for adding a reaction to a photo. The function in that file uses a DynamoDB transaction to add a reaction.

The contents of the file are as follows:

import datetime

import boto3

dynamodb = boto3.client('dynamodb')

REACTING\_USER = 'kennedyheather'

REACTION\_TYPE = 'sunglasses'

PHOTO\_USER = 'ppierce'

PHOTO\_TIMESTAMP = '2019-04-14T08:09:34'

def add\_reaction\_to\_photo(reacting\_user, reaction\_type, photo\_user, photo\_timestamp):

reaction = "REACTION#{}#{}".format(reacting\_user, reaction\_type)

photo = "PHOTO#{}#{}".format(photo\_user, photo\_timestamp)

user = "USER#{}".format(photo\_user)

try:

resp = dynamodb.transact\_write\_items(

TransactItems=[

{

"Put": {

"TableName": "quick-photos",

"Item": {

"PK": {"S": reaction},

"SK": {"S": photo},

"reactingUser": {"S": reacting\_user},

"reactionType": {"S": reaction\_type},

"photo": {"S": photo},

"timestamp": {"S": datetime.datetime.now().isoformat() }

},

"ConditionExpression": "attribute\_not\_exists(SK)",

"ReturnValuesOnConditionCheckFailure": "ALL\_OLD"

},

},

{

"Update": {

"TableName": "quick-photos",

"Key": {"PK": {"S": user}, "SK": {"S": photo}},

"UpdateExpression": "SET reactions.#t = reactions.#t + :i",

"ExpressionAttributeNames": {

"#t": reaction\_type

},

"ExpressionAttributeValues": {

":i": { "N": "1" },

},

"ReturnValuesOnConditionCheckFailure": "ALL\_OLD"

}

}

]

)

print("Added {} reaction from {}".format(reaction\_type, reacting\_user))

return True

except Exception as e:

print("Could not add reaction to photo")

add\_reaction\_to\_photo(REACTING\_USER, REACTION\_TYPE, PHOTO\_USER, PHOTO\_TIMESTAMP)

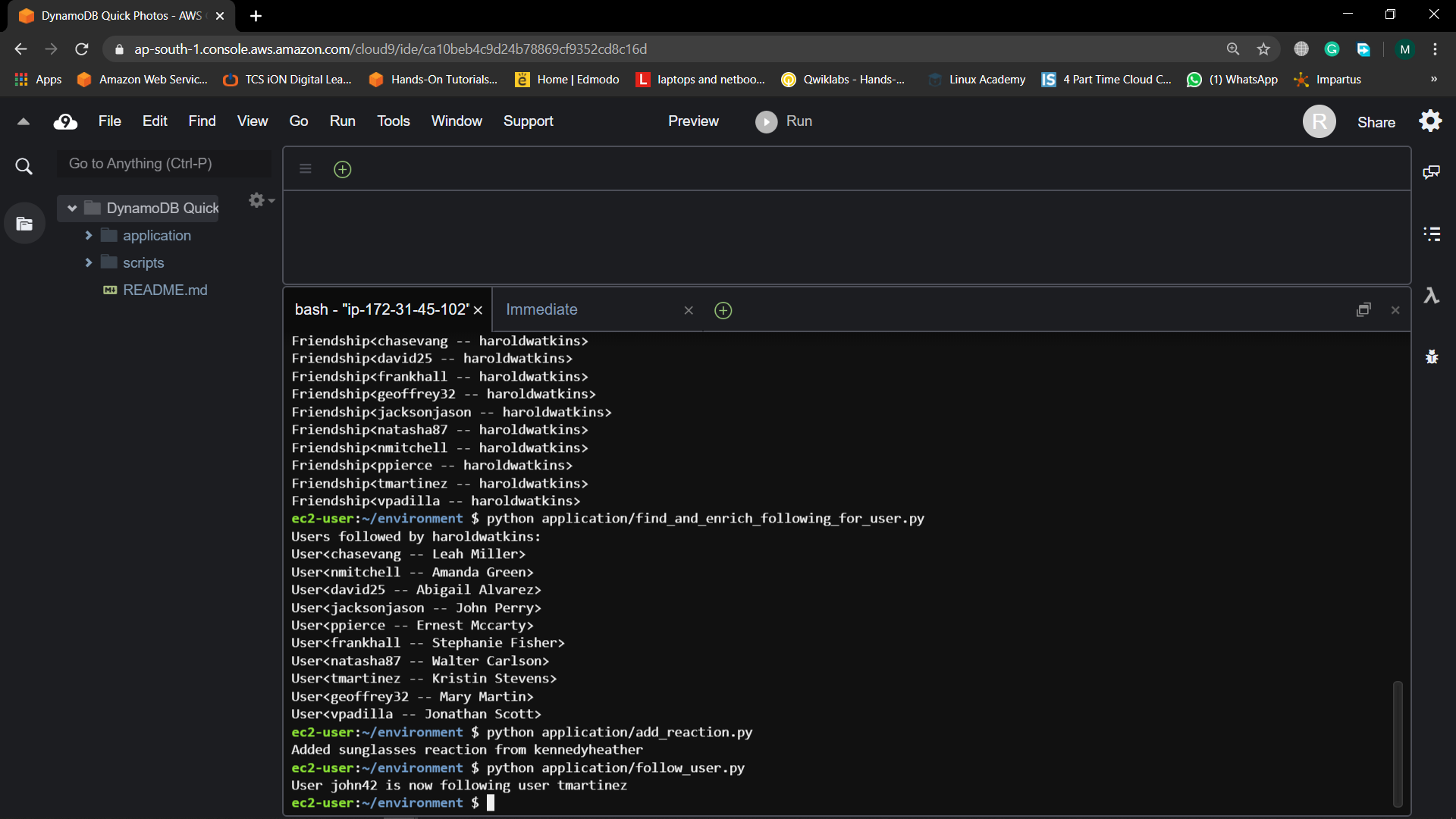
n the *add\_reaction\_to\_photo*function, we’re using the *transact\_write\_items()* method to perform a write transaction. Our transaction has two operations.

First, we’re doing a *Put*operation to insert a new *Reaction*entity. As part of that operation, we’re specifying a condition that the *SK*attribute should not exist for this item. This is a way to ensure that an item with this *PK*and *SK* doesn’t already exist. If it did, that would mean the user has already added this reaction to this photo.

The second operation is an *Update*operation on the *User*entity to increment the reaction type in the *reactions*attribute map. DynamoDB’s powerful [update expressions](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.UpdateExpressions.html) allow you to perform atomic increments without needing to first retrieve the item and then update it.

Run this script with the following command the terminal.

python application/add\_reaction.py



Note that if we try to run the script again, the function will fail. The user *kennedyheather*has already added this reaction to this photo, so trying to do it again would violate the condition expression in the operation to create the *Reaction*entity. In other words, the function is idempotent and repeated invocations of it with the same inputs will not have unintended consequences.

The addition of DynamoDB transactions greatly simplifies the workflow around complex operations like these. Previously, this would have required multiple API calls with complex conditions and manual rollbacks in the event of conflicts. Now it can be implemented with less than 50 lines of code.

In the next step, we’ll see how to handle our “Follow user” access pattern.

### **Step 17: Following a user :-**

In your application, one user can follow another user. When the application backend gets a request to follow a user, we need to do four things:

* Check that the following user is not already following the requested user;
* Create a *Friendship*entity to record the following relationship;
* Increment the follower count for the user being followed;
* Increment the following count for the user following.

In the code you downloaded, there is a file in the *application/* directory called *follow\_user.py*. The contents of the file are as follows:

import datetime

import boto3

dynamodb = boto3.client('dynamodb')

FOLLOWED\_USER = 'tmartinez'

FOLLOWING\_USER = 'john42'

def follow\_user(followed\_user, following\_user):

user = "USER#{}".format(followed\_user)

friend = "#FRIEND#{}".format(following\_user)

user\_metadata = "#METADATA#{}".format(followed\_user)

friend\_user = "USER#{}".format(following\_user)

friend\_metadata = "#METADATA#{}".format(following\_user)

try:

resp = dynamodb.transact\_write\_items(

TransactItems=[

{

"Put": {

"TableName": "quick-photos",

"Item": {

"PK": {"S": user},

"SK": {"S": friend},

"followedUser": {"S": followed\_user},

"followingUser": {"S": following\_user},

"timestamp": {"S": datetime.datetime.now().isoformat()},

},

"ConditionExpression": "attribute\_not\_exists(SK)",

"ReturnValuesOnConditionCheckFailure": "ALL\_OLD",

}

},

{

"Update": {

"TableName": "quick-photos",

"Key": {"PK": {"S": user}, "SK": {"S": user\_metadata}},

"UpdateExpression": "SET followers = followers + :i",

"ExpressionAttributeValues": {":i": {"N": "1"}},

"ReturnValuesOnConditionCheckFailure": "ALL\_OLD",

}

},

{

"Update": {

"TableName": "quick-photos",

"Key": {"PK": {"S": friend\_user}, "SK": {"S": friend\_metadata}},

"UpdateExpression": "SET following = following + :i",

"ExpressionAttributeValues": {":i": {"N": "1"}},

"ReturnValuesOnConditionCheckFailure": "ALL\_OLD",

}

},

]

)

print("User {} is now following user {}".format(following\_user, followed\_user))

return True

except Exception as e:

print(e)

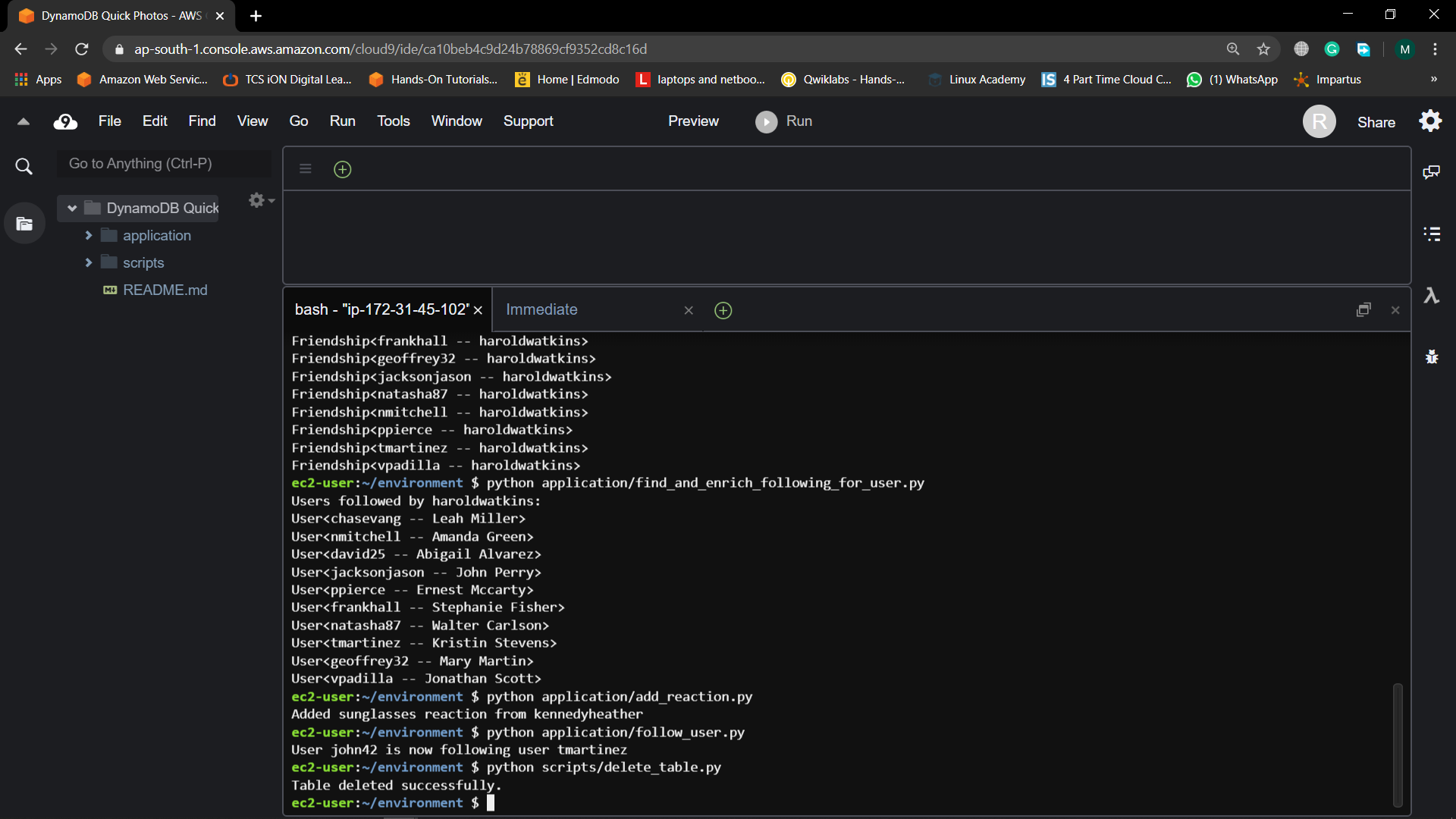
print("Could not add follow relationship")

follow\_user(FOLLOWED\_USER, FOLLOWING\_USER)

The follow\_user function in the file is similar to a function you would have in your application. It takes two usernames -- one of the followed user and one of the following user -- and it runs a request to create a *Friendship*entity and update the two *User*entities.

Run the script in your terminal with the following command.

python application/follow\_user.py



If we try to running the script a second time in your terminal. This time, you should get an error message indicating that you could not add the follow relationship. This is because this user now follows the requested user, so our request then failed the conditional check on the item.

In these parts, we saw how to satisfy two advanced write operations in our application. First, we used DynamoDB transactions for having a user react to a photo. With transactions, we handled a complex conditional write across multiple items in a single request. Further, we saw how to use DynamoDB update expressions to increment a nested attribute in a map property.

Second, we implemented the function for a user to follow another user. This required altering three items in a single request, while also performing a conditional check on one of the items. While this would normally be a difficult operation, DynamoDB makes it simple to handle this with DynamoDB transactions.

In the next module, we’ll clean up the resources we created and see some next steps in our DynamoDB learning path.

### 

### **Step 18: Delete the DynamoDB table :-**

As part of the cleanup process, you need to delete the DynamoDB table you used for this lab.

In the code you downloaded, there is a file in the scripts/ directory called delete\_table.py. The contents of that file are as follows.

import boto3

dynamodb = boto3.client('dynamodb')

try:

dynamodb.delete\_table(TableName='quick-photos')

print("Table deleted successfully.")

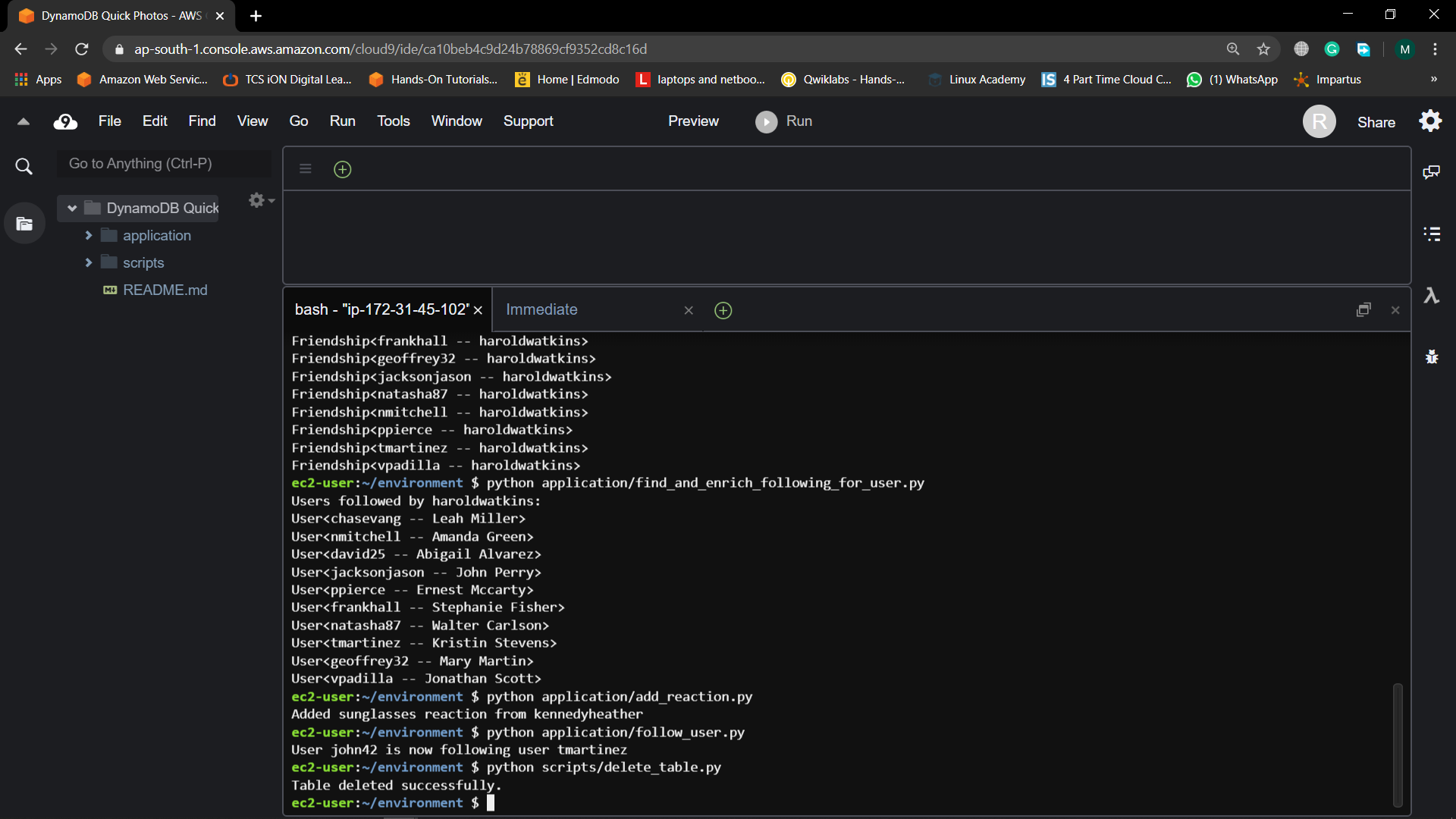
except Exception as e:

print("Could not delete table. Please try again in a moment. Error:")

print(e)

In the terminal, run the following command to run this script and delete the created table.

python scripts/delete\_table.py



### 

### **Step 19: Delete the AWS Cloud9 environment :-**

To delete the AWS Cloud9 environment that you used in this lab:

1. Navigate to the [AWS Cloud9 console](https://console.aws.amazon.com/cloud9/home).
2. Choose the DynamoDB Quick Photos environment and choose Delete
3. In the dialog box, type *Delete*in the box, and choose Delete.

