You need to use DynamoDB to build a mobile application and will be doing so by using DynamoDB to build the data model as well as a social network

Step 1: Set up AWS Cloud9 IDE

* Navigate to the Cloud9 service
* Choose create environment
* Type DynamoDB Quick Photos in the **Name** box
* Choose next step
* Leave the **Environment settings** at their defaults to create a new t2.micro ec2 instance
* Choose **Next step**
* Select **Create environment**

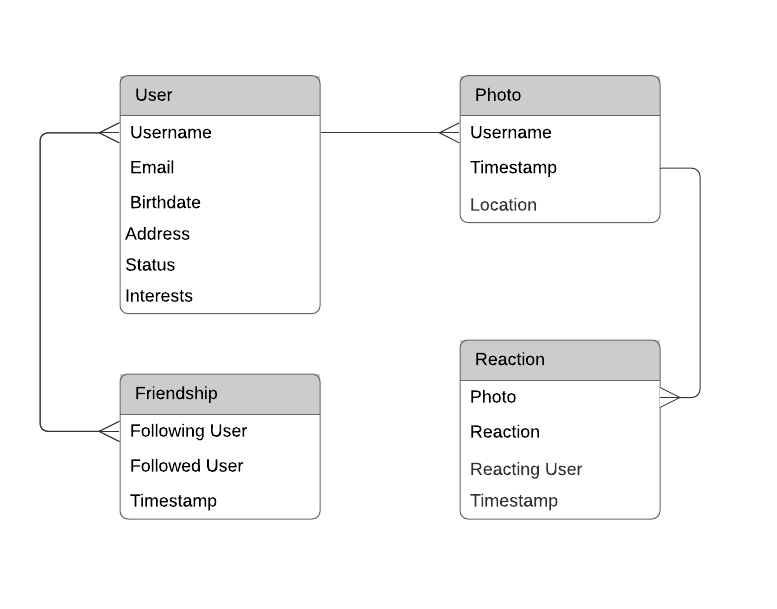
Step 2: Download supporting code

* Run this code in your Cloud9 terminal to download and unpack module code
  + cd ~/environment
  + Curl -sL <https://s3.amazonaws.com/ddb-labs/quick-photos.tar> | tar -xv
* Run the following command to look at your directories
  + ls
* There should be two directories now, application and scripts

Step 3: Build your entity-relationship diagram

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 4: 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 5: 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 or Snapchat. 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 6: 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*)

Step 7: 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.

With this in mind, let’s use the following pattern for HASH and RANGE values for each entity type:

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 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 8: Create a table

* The code you downloaded earlier in the lab includes a 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)

To create the table, run the Python script with the following command.

* python scripts/create\_table.py

Step 9: Bulk-load data into the table

* 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)

Run the bulk\_load\_table.py script and load your table with data by running the following command in the terminal.

* python scripts/bulk\_load\_table.py

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

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

This should display the following results.

{

"Count": 967,

"ScannedCount": 967,

"ConsumedCapacity": null

}

Step 10: Retrieve multiple entity types in a single request

* The following 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)
* You can run the script in your terminal with the following command.
  + python application/fetch\_user\_and\_photos.py
* 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>

Step 11: Create a 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)

Whenever attributes are used in a primary key for the table or secondary index, they must be defined in *AttributeDefinitions*. Then, we *Create* a new secondary index in the *GlobalSecondaryIndexUpdates* property. For this secondary index, we specify the index name, the schema of the primary key, the provisioned throughput, and the attributes we want to project.

Note that an inverted index is a name of a design pattern rather than an official property in DynamoDB. Creating an inverted index is just like creating any other secondary index.

* Create your inverted index by running the command below.
  + python scripts/add\_inverted\_index.py

Step 12: Query the inverted index to find a photo’s reactions

* In the code you downloaded, there is a file in the application/ directory called fetch\_photo\_and\_reactions.py. 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.
* Run this command
  + python application/fetch\_photo\_and\_reactions.py
* You should see it output a photo and its five reactions.
  + Photo<david25 -- 2019-03-02T09:11:30>
  + Reaction<ylee -- PHOTO#david25#2019-03-02T09:11:30 -- smiley>
  + Reaction<kennedyheather -- PHOTO#david25#2019-03-02T09:11:30 -- smiley>
  + Reaction<jenniferharris -- PHOTO#david25#2019-03-02T09:11:30 -- +1>
  + Reaction<geoffrey32 -- PHOTO#david25#2019-03-02T09:11:30 -- +1>
  + Reaction<chasevang -- PHOTO#david25#2019-03-02T09:11:30 -- +1>
* Note that the secondary index takes a moment to backfill. You may get an error message indicating that backfilling is in progress. If so, try again in a few minutes.

Step 13: 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 you downloaded, there is a file in the application/ directory called find\_following\_for\_user.py. The contents of this script follows.
  + 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)
* Run the script by running the following command in your terminal.
  + python application/find\_following\_for\_user.py
* Your console should output a list of users followed by the given username:
  + Users followed by haroldwatkins:
  + Friendship<chasevang -- haroldwatkins>
  + Friendship<david25 -- haroldwatkins>
  + Friendship<frankhall -- haroldwatkins>
  + Friendship<geoffrey32 -- haroldwatkins>
  + Friendship<jacksonjason -- haroldwatkins>
  + Friendship<natasha87 -- haroldwatkins>
  + Friendship<nmitchell -- haroldwatkins>
  + Friendship<ppierce -- haroldwatkins>
  + Friendship<tmartinez -- haroldwatkins>
  + Friendship<vpadilla -- haroldwatkins>

Step 14: Use partial normalization to find followed users

* 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
* Your console should output a list of users followed by the given username.
  + Users followed by haroldwatkins:
  + User<ppierce -- Ernest Mccarty>
  + User<vpadilla -- Jonathan Scott>
  + User<david25 -- Abigail Alvarez>
  + User<jacksonjason -- John Perry>
  + User<chasevang -- Leah Miller>
  + User<frankhall -- Stephanie Fisher>
  + User<nmitchell -- Amanda Green>
  + User<tmartinez -- Kristin Stevens>
  + User<natasha87 -- Walter Carlson>
  + User<geoffrey32 -- Mary Martin>

Step 15: React 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
* 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)
* In 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 allow you to perform atomic increments without needing to first retrieve the item and then update it.
* Run this script with the following command in your terminal.
  + python application/add\_reaction.py
* The output in your terminal should indicate that the reaction was added to the photo.
  + Added sunglasses reaction from kennedyheather

Step 16: 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
* You should see output in your terminal indicating that the operation succeeded.
  + User john42 is now following user tmartinez

**Clean up:**

* In the previous sections, we’ve satisfied the following access patterns in our application:
  + Create user profile (*Write*)
  + Update user profile (*Write*)
  + Get user profile (*Read*)
  + Upload photo (*Write*)
  + View recent photos for user (*Read*)
  + React to a photo (*Write*)
  + View photo and reactions (*Read*)
  + Follow user (*Write*)
  + View followers for user (*Read*)
  + View followed for user (*Read*)
* The strategies we used to satisfy these patterns included:
  + A single-table design that combined multiple entity types in one table.
  + A composite primary key that allow for many-to-many relationships.
  + An inverted index to allow reverse lookups on our many-to-many entity.
  + Partial normalization to keep our data fresh while remaining performant.\
  + DynamoDB transactions to handle complex write patterns across multiple items.

Step 17: 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 your terminal, run the following command to run this script and delete your table.
  + python scripts/delete\_table.py

Step 18: Delete the AWS Cloud9 environment

* Navigate to the AWS Cloud9 console
* Choose the DynamoDB Quick Photos environment and choose **Delete**
* In the dialog box, type *Delete* in the box and choose **Delete**

Congratulations, you completed this project