Invites

Introduction

We're building an app that lets members create events (e.g. parties, book protests, clubs, dining out). The organizers will invite other members to those events, and members will be able to indicate if they're attending or not.

Some of the code has already been written, so we've got a good foundation, and you'll be dealing with a few important use cases involving members being deleted, and events being canceled.

For privacy reasons, we'll want to delete member records completely if a member requests it. But when an event is canceled, we don't want to delete the record outright; we'll marked it as "canceled" instead.

The three key entities in our app right now are:

- Members: The people who can invite one another to events
  - Partition key: id
- Events: The events that people are being invited to
  - Partition key: id
- Invites: A request for a particular Member to attend a particular Event
  - Partition key: eventId
  - Sort key: memberId

Our service will support a number of requests regarding these three entity types. The relevant cases at this point are:

- Members
  - CreateMember
  - DeleteMember: actually deletes the member permanently
  - GetMember: by member ID
- Events
  - CancelEvent
  - CreateEvent
  - GetEvent: by event ID
- Invites
  - CreateInvite
  - GetInvite: by event ID + member ID
  - GetInvitesForMember: returns List of Invites **sent to** a member

The code base follows the Activity-DAO-DynamoDBMapper pattern that we've come to know and love. The various Activity classes each implement one operation that our service supports. That Activity may depend on several of `MemberDao`'s, `EventDao`'s, `InviteDao`'s methods to accomplish their use cases. Each DAO is responsible for one model type, and only interacts with that model's DynamoDB table.

Disclaimer: One difference that you'll notice here is that our Activity classes don't yet accept/return Response/Result objects. They're accepting/returning individual values. We'll do this until it's time to create

our service infrastructure and create the necessary models. That retrofit is beyond the scope of this activity, but will be fairly easily accomplished in the Activity classes themselves when the time comes.

You'll primarily be updating DAO and Activity code, but will touch a few tests as well, where appropriate (we'll guide you!).

For each "Phase" of the activity, there's a test that ensure you've satisfied the requirements for that phase. Also make sure that the relevant unit tests for the classes you're working on are passing (typically one Activity and 1-3 DAOs).

Helpful hint: If a test is failing for one of the phases, and you want to see more about the test failure, use the file URL that is printed out on the commandline to see more detail!

Phase 0: Preliminaries

1. Create the tables we'll be using for this activity by running these aws CLI commands:

```
aws cloudformation create-stack --region us-west-2 --stack-name
dynamodbdeleteiterators-eventstable01 --template-body
file://cloudformation/events_table.yaml --capabilities CAPABILITY_IAM
aws cloudformation create-stack --region us-west-2 --stack-name
dynamodbdeleteiterators-invitestable01 --template-body
file://cloudformation/invites_table.yaml --capabilities CAPABILITY_IAM
aws cloudformation create-stack --region us-west-2 --stack-name
dynamodbdeleteiterators-memberstable01 --template-body
file://cloudformation/members_table.yaml --capabilities CAPABILITY_IAM
```

2. Discuss the different attributes with your team to make sure you all understand what they represent.
3. As a final verification, run the `Phase0Test` test file and be sure it passes.

GOAL: Events, Invites, and Members tables are all created in your AWS Account, and the attributes make sense

Phase 0 is complete when:

- You understand the 3 data types and their relationships
- Events, Invites, Members tables exist with some sample data
- `Phase0Test` passes

Phase 1: Delete Member

For privacy reasons, we're required to allow a member to completely delete their data from our database. We're still pulling the service together, so we're just implementing the Activity classes at this point.

Deleting members hasn't been implemented at all yet. You're looking for your next task in this sprint, and this one's yours. We'll tackle this in two steps:

- Start off by writing a unit test for the relevant DAO, then implement the method in the DAO.
- Integrate the DAO into the relevant Activity class.

Let's get started!

1. Implement a unit test in `MemberDaoTest` to cover the "happy path" for the deletePermanently() method that's declared (but not implemented yet) in `MemberDao`
    1. Mock your com.amazon.ata.dynamodbdeleteiterators.classroom.dependency on `DynamoDBMapper` and `verify()` that it receives a call to `DynamoDBMapper`'s `delete()` method
2. Run the test and verify it fails before updating `MemberDao`
3. Implement `MemberDao`'s `deletePermanently()` so that the test passes
4. Try running `DeleteMemberActivityTest` (already implemented for you) and make sure that it fails
5. Use the `MemberDao` method in `DeleteMemberActivity` as appropriate
6. Make sure `DeleteMemberActivityTest` passes now
7. Run the `Phase1Test` make sure it passes!

GOAL: Your `DeleteMemberActivity` is working!

Phase 1 is complete when:

- You've written a happy path (member is deleted) `MemberDao` unit test for `deletePermanently()`, and the test passes
- `DeleteMemberActivityTest` tests pass
- `Phase1Test` tests pass

Phase 2: Delete Member's Invites (too)

To uphold the privacy promise to our customers, we'll also want to delete any invitations that the member has received (it can still reveal a lot about someone by seeing what they've been invited to).

We've provided a way to fetch all invites sent to a member: `InviteDao.getInvitesSentToMember()`. You can use this in `DeleteMemberActivity` to find the invites to delete.

For now, we're going to leave any invites in the database that **have** been accepted, so that the organizer can see who accepted. We'll have a future task coming up soon to figure out how to handle accepted invites that are deleted when a member is deleted. This means, you'll need to **conditionally** delete invites sent to the deleted member only if the invite has `isAttending` false.

Data can change between when an item is fetched from DynamoDB and when it is updated (or deleted), so don't trust the value in the `Invite` object you receive from the DAO--it might've updated from underneath you by some other request! In your Activity class, attempt to delete each `Invite` regardless of the value of `isAttending` in the `Invite` object that you have. Let the DAO handle the conditional delete logic.

Remember that an exception is thrown when a condition is violated, so in this case, you'll catch the exception and carry on.

1. Implement `InviteDao`'s `deleteInvite()` method to conditionally delete an `Invite` by `eventId` (the partition key) and `memberId` (sort key), only if `isAttending` is false. Take a look at AttributeValue's Javadoc to see how to create a boolean attribute type.
2. Verify that `InviteDaoTest` (implemented for you) passes
3. Update `DeleteMemberActivity` to use `InviteDao` to fetch and delete the invitations.
    1. To fetch the invites, use `InviteDao.getInvitesSentToMember()`

    1. Use your implementation of `InviteDao.deleteInvite()` to conditionally delete each invite.
    2. HINT: Because you're adding a com.amazon.ata.dynamodbdeleteiterators.classroom.dependency here, you'll might need to do a couple of things to avoid getting a `NullPointerException` when running this unit test: 1. Add the following to the `DeleteMemberActivityTest` to make sure that `DeleteMemberActivity` gets its proper `InviteDao` mock injected (rather than `null`): `{.java} @Mock private InviteDao inviteDao;` 1. You may need to rebuild your project to re-generate Dagger classes.
2. NOTE: You'll need to update `DeleteMemberActivityTest` to provide the `InviteDao` com.amazon.ata.dynamodbdeleteiterators.classroom.dependency to `DeleteMemberActivity` under test. Follow the pattern and add an `InviteDao` mock that gets injected as `MemberDao` does.
    1. For completeness, you should also stub the response that the `inviteDao` mock should return to `getInvitesSentToMember()` -- feel free to stub it to return empty list, but more realistic would be to return an Invite and verify that `deleteInvite()` is also called with that invite's event/member IDs.

GOAL: `DeleteMemberActivity` also deletes invites to the deleted member, but only if the invite hasn't been accepted.

Phase 2 is complete when:

- You've implemented `deleteInvite()`
- `InviteDaoTest` passes
- `DeleteMemberActivityTest` tests still all pass
- `Phase2Test` tests pass

Phase 3: The Softer, Gentler Delete

Through the discussions of performing hard deletes for members, your team realizes you should actually go back and update the existing implementation of event-canceling. The current implementation hard deletes an event when it's canceled, but this loses potentially valuable data (especially if the organizer decides they've made a mistake and want the event back!).

Update `EventDao`'s `cancelEvent()` method to perform a soft delete, marking the event as `isCanceled` = true, instead of deleting it outright.

One of your teammates is an avid Test-Driven Developer (TDD), so they've actually updated `EventDaoTest`, which is now failing. So you only need to update `EventDao` and be on your way!

GOAL: Canceling an event only marks the event item as `isCanceled` = true, and does **not** delete from the Events table.

Phase 3 is complete when:

- You've updated `EventDao.cancelEvent()` to soft delete rather than hard delete the event.
- `EventDaoTest` is passing
- `Phase3Test` tests pass

Phase 4: Continuous Data Improvement

A member can make a request to see all invitations that they've received through the application. Sounds great, but wait, didn't we just change event canceling from a delete into `isCanceled`, so those will get returned now too?

Yes, and there are a couple of things we can do about that. One, of course, is to proactively mark all such invites as canceled right away. That can become costly, especially if a member never comes back to the app and we've updated an item in DynamoDB for nothing!

The other downside of relying only on this approach is that we might not mark all invites as canceled fast enough to make sure that a member looking at their invites sees the most up-to-date invites. Or worse, we somehow fail to update a particular invite and it still goes on thinking it's pointing to an event that's still happening.

A good practice is to make your data "self-healing", meaning that in the course of running the app, any data inconsistencies get cleaned up along the way.

That's what we're going to do here: when fetching all invites for a member, we'll check to see if the event is canceled. If it is canceled, we also mark the invite as canceled (in DynamoDB too).

Further, we actually want to **replace** the Invite object that gets passed back so that it is actually a `CanceledInvite` object that our UI knows how to render properly. (The main benefit of `CanceledInvite` is that its setters have been disabled, so it can't be updated.)

Plan of attack:

1. In `GetInvitesForMemberActivity`, update `handleRequest()` to:
    1. Collect all of the event IDs from the invites returned for the member (fetching the invites is already implemented, by calling `InviteDao.getInvitesSentToMember()`)
    2. Use the provided `EventDao.getEvents()` method to get all of the events in a batch load (more efficient than requesting the events one at at time)
    3. Iterate through each of the Invites:
        1. If the corresponding Event is marked `isCanceled` = true, then mark the **`Invite`** as canceled, using `InviteDao.cancelInvite()`
        2. (You'll probably want a way to look up the right `Event` for a given `Invite` -- consider a `Map` for this...what would your key/values be?)
        3. Also: If the `Invite` is already marked `isCanceled` = true, you don't need to update it again
2. Ensure that the `getInvitesForMemberActivity_withCanceledEvent_marksCorrespondingInviteAsCanceled` test in `Phase4Test` passes.
3. Go back to that iteration through the invites for the member, and update the iteration to use an iterator so that as we're iterating we can:
    1. Remove the `Invite`s that point to a canceled `Event`
    2. Insert `CanceledInvite` objects in their place
        1. You may find the `CanceledInvite(Invite)` constructor particularly handy here.
4. Check that all tests in `Phase4Test` now pass

GOAL: Mark all invites for a canceled event as canceled, and return `CanceledInvite` in place of any `Invite` in `GetInvitesForMemberActivity`

Phase 4 is complete when:

- You've updated `GetInvitesForMemberActivity` to detect Invites that point to canceled events, and to mark those Invites as canceled.
- You've updated `GetInvitesForMemberActivity` to also replace `Invite`s with `CanceledInvite`s in these cases (**using an iterator**).
- `Phase4Test` tests pass

Extensions

These should be doable independently, so do not need to be done in any order.

1. Allow your `Invite.isAttending` to be null if invitee hasn't responded yet, to differentiate between "I haven't answered yet" and "I'm not coming". You might need to change your conditional delete logic in `InviteDao` to accommodate this. Hint: To keep it simple, you might be able to use a `ComparisonOperator` on your `ExpectedAttributeValue` to indicate that `isAttending` must NOT be true (but could be null or false). Add a relevant integration test to verify this case.

2. To make the DynamoDB interaction a little more efficient in `GetInvitesForMemberActivity`, if an `Invite` is already marked as canceled, exclude that `Invite`'s `Event` from the collection of event IDs passed into `EventDao.getEvents()`. (Only query the events for invites that are still marked as not-canceled). Write a corresponding integration test (either in a new class or in the Phase4 test, as you prefer).

3. Implement "cold storage" for the older events (archiving the events that are less likely to be accessed in a separate table (or datastore) from more current events). This reduces the size of the actively accessed table, which can decrease costs.

   Add an Activity to scan through all existing events. All of the events that are over 30 days old should be removed from the Events table and added to a PastEvents table. You'll need to create the PastEvents table yourself in the AWS console. We recommend creating the PastEvent first, and only after that succeeds, delete from the Events table. This way, there's less risk of losing data. Add relevant unit and integration tests.

4. Similarly, archive old invites to "cold storage" to store invites for events more than 30 days in the past. Add unit and integration tests.