Part D: Explanation - Lazy Loading and Transactional Integrity in Spring

In this part, we modified the `AccountService` class by adding the `@Transactional` annotation at the class level.
We also changed the entity relationships in the `Account` class to use `FetchType.LAZY`.

Why we don't need EAGER loading anymore and the application still works:

1. **Lazy Loading Explained**:
- When relationships like @ManyToOne or @OneToMany are set to `LAZY`, related entities are not loaded from the database until they are accessed.
- This is more efficient because it reduces the initial query load and only fetches the data we need when we actually access the related field.

2. **Transactional Context Enables Lazy Access**:
- Without `@Transactional`, accessing a lazily-loaded relationship outside of the initial method call will fail with a `LazyInitializationException`, because the Hibernate session is already closed.
- However, when we annotate the service class with `@Transactional`, all method calls execute inside a single Hibernate session. This means even lazily-loaded fields can be accessed without error as long as we're still inside that transaction boundary.

3. **Application Still Works Because**:
- All access to lazy fields happens within the transactional method, so the persistence context is open.
- For example, if we fetch an `Account`, and then immediately access `account.getCustomer()`

inside the same method, it works fine because the session is still open.

4. **Why Lazy is Preferred**:

- Using `LAZY` loading improves performance and avoids unnecessary joins and data loading.
- With proper transaction management, you get the benefits of `LAZY` without encountering lazy loading issues.

Conclusion:

We don't need `EAGER` loading anymore because Spring ensures that all method executions within the service class annotated with `@Transactional` occur inside an open persistence context. As long as related data is accessed within that method, Hibernate can lazily fetch it safely.