Deliverer Mapper Pattern Documentation

Introduction

This document explains the transformation of the mapper pattern implementation for the Deliverer entity and its entity mapping.
It will compare how the code was before and after the adaptation, and provide reasoning for each change.

Before: Initial Implementation

Before adaptation, the mapper logic was implemented using individual classes for each entity transformation.

Example of Mapper Classes:

```
class DelivererToDelivererEntityMapper : Mapper<Deliverer, DelivererEntity> {
    override fun map(from: Deliverer): DelivererEntity {
        return DelivererEntity(from.delivererId, from.name)
    }
}

class DelivererEntityToDelivererMapper : Mapper<DelivererEntity, Deliverer> {
    override fun map(from: DelivererEntity): Deliverer {
        return Deliverer(from.delivererId, from.name)
    }
}
```

Repository example:

```
class DelivererRepositoryImpl(
    private val delivererDao :DelivererDao,
    private val mapper: Mapper<Deliverer, DelivererEntity>
) : DelivererRepository {
    override suspend fun insert(deliverer: Deliverer) {
        delivererDao.insertDeliverer(mapper.map(deliverer))
    }
}
```

After: Adapted Implementation

After adaptation, the mapper logic was refactored to improve code readability, reduce the need for object instantiation, and increase
the reusability of the mappers by using singleton objects. The mappers were modified to follow the Singleton pattern by using Kotlin objects
instead of classes.

Updated Mapper Classes:

```
object DelivererToEntityMapper : Mapper<Deliverer, DelivererEntity> {
    override fun map(from: Deliverer): DelivererEntity {
        return DelivererEntity(from.delivererId, from.name)
    }
}
```

```
object DelivererEntityToDomainMapper : Mapper<DelivererEntity, Deliverer> {
    override fun map(from: DelivererEntity): Deliverer {
        return Deliverer(from.delivererId, from.name)
    }
}
```

This allows for the reusability of these mappers without the need to instantiate them each time.

Repository example:

```
class DelivererRepositoryImpl(
    private val delivererDao: DelivererDao
) : DelivererRepository {

    override suspend fun insert(deliverer: Deliverer) {
        delivererDao.insertDeliverer(DelivererToEntityMapper.map(deliverer))
    }

    override suspend fun update(deliverer: Deliverer) {
        delivererDao.updateDeliverer(DelivererToEntityMapper.map(deliverer))
    }

    override suspend fun delete(deliverer: Deliverer) {
        delivererDao.deleteDeliverer(DelivererToEntityMapper.map(deliverer))
    }

    override fun getAllDeliverers(): Flow<List<Deliverer>> {
```

```
        return delivererDao.getAllDeliverers()
            .map { DelivererEntityToDomainMapper.mapAll(it) }
    }
}
```

The `mapAll` function, which is an extension of the Mapper interface, allows for transforming lists of items.

Reasons for Changes

The adaptation of the mapper pattern includes several reasons:

1. **Improved Code Readability:**
    - By converting the mappers into singleton `object` classes, we make the code cleaner, more concise, and easier to maintain.

2. **Better Memory Efficiency:**
    - By using `object` instead of `class`, we avoid unnecessary instantiations of the mappers. Each `object` is a singleton, ensuring the mapper is used only once.

3. **Reusability:**
    - The mappers are now reusable across the project without the need to create new instances every time they are required. This reduces the boilerplate code.

4. **Support for Bulk Transformation with `mapAll()`:**
    - The `mapAll()` function was introduced to simplify bulk transformations for lists of items.

5. **Flexible Flow Mapping:**

- The `mapAll` function was also adapted for usage with `Flow` to handle asynchronous data streams like database queries. This allows for better scalability and efficient memory usage with large datasets.