Transactional key-value store submission guide

Assignment brief

The assignment is to build an interactive command line interface to a transactional key value store. A user should be able to compile and run this program and get an interactive shell with a prompt where they can type commands. The user can enter commands to set/get/delete key/value pairs and count values. All values can be treated as strings, no need to differentiate by type. The key/value data only needs to exist in memory for the session, it does not need to be written to disk.

The interface should also allow the user to perform operations in transactions, which allows the user to commit or roll back their changes to the key value store. That includes the ability to nest transactions and rollback and commit within nested transactions. The solution shouldn't depend on any third party libraries.

The interface should support the following commands:

```
SET <key> <value> : store the value for key
GET <key> : return the current value for key
DELETE <key> : remove the entry for key
COUNT <value> : return the number of keys that have the given value
BEGIN : start a new transaction
COMMIT : complete the current transaction
ROLLBACK : revert to state prior to BEGIN call
```

Assignment submission

Android app overview

The submission for this assignment is an Android app coded in Kotlin – with its source code provided.

App stack

- View bindings for accessing views
- Coroutines for “backend” asynchronous programming
- Kotlin Flow for asynchronous programming between view and viewModel
- Hilt for dependency injection
- Mock and Kluent for unit tests

App layers

The app is divided into three layers: presentation, domain, and data.

The presentation layer receives input from the user and displays data to the user. It implements a Model-View-ViewModel (MVVM) design pattern.

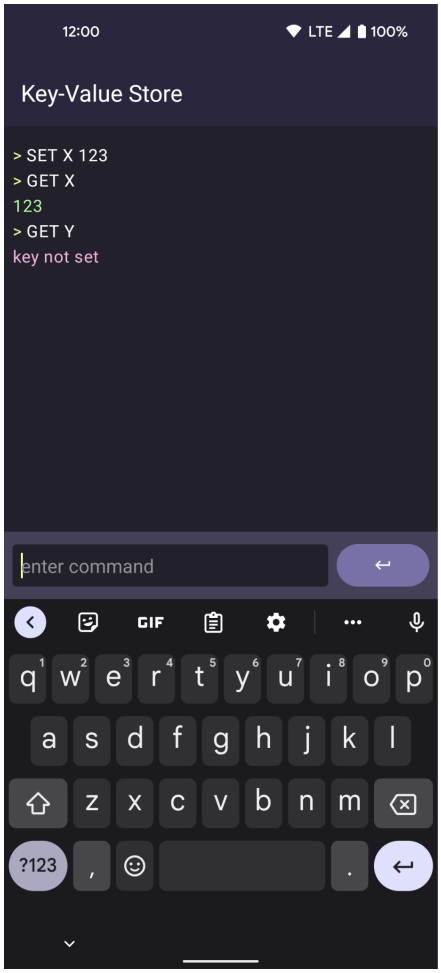
The domain layer is the highest level layer and contains the business logic - e.g. use cases, domain models, and data layer interfaces. It sits between the presentation layer and the data layer.

The data layer is responsible for storing and retrieving data. It contains the specific key-value store implementation.

App interface

The app has a simple user interface that consists of:

1. Command line display that covers most of the screen real estate
2. Command input box at the bottom left
3. Command submit button at the bottom right



New commands are entered into the input box and are then submitted with either the submit button or the Enter key on the keyboard. When a new command is submitted it is added to the command line display and executed. If the command's execution has a "success" response, the response is added in green to the command line display. If the command's execution has an error response, the response is added in red to the command line response.

Key-value store implementation

The store (`ConcurrentMemoryStore` implementing interface `KeyValueStore)` which this app implements, exists only in memory and it is thread safe - i.e. it can be accessed concurrently from multiple threads at the same time. This store makes use of coroutines for concurrency.

Each transaction is represented by a `Transaction` class that contains two hashmaps, a hashmap for the key-value mappings and a hashmap for the mapping from value to its count in the transaction. This representation allows most store operations to happen in constant time. The one exception is the COMMIT operation where the complexity depends on the number of entries in the transaction. The constant complexity for the COUNT operation is achieved with the inclusion of the hashmap of values to their count in the transaction class.

To simulate multiple, nested transactions a single `ArrayDeque<Transaction>` instance is used. It is used because it can simulate a stack data structure with a Last In First Out (LIFO) principle efficiently - i.e. it is optimized for the functionality where the last transaction added is the first to be removed. Each new transaction adds a `Transaction` to the `ArrayDequeue`. Note, there is always at least one `Transaction` in the `Dequeue` which is the base transaction of the store and it cannot be rolled back or commited.

To ensure thread safety on the `ArrayDeque` and hashmaps - which by themself are not thread safe - the `Mutex` from `kotlinx.coroutines.sync` is used. This `Mutex` supports coroutines and suspends them when they are waiting for access instead of blocking them.

Most of the store commands are performed on the caller's thread since they perform in constant time and the coroutines are simply suspended if waiting for access. The exception again is the COMMIT command which is explicitly performed on a background thread, since it might take a not-insignificant amount of time on a large store.

Final thoughts

I set out to find a sensible solution to the brief and hope I achieved that. Please let me know if your experience does not match what is outlined in this document - criticism is always welcomed.