Magenta Chat

Magenta chat is a simple application that simulates a chat feature between 2 users.

Video

Project decisions

The application was made following the MVVM architecture and the Google guidelines for android development.

Used libraries and frameworks

- *Jetpack Compose* for construct the interfaces
- *Hilt* for dependency injection
- *Room* for data persistence
- *Kotlin Coroutines* for managing asynchronous tasks and data flows
- *JUnit* for test setup and assertions
- *Mockk* for creating mock objects for testing
- *Kotlin fixture* for creating dummy objects for testing

Other decisions

Conventional commits Conventional commits is an specification of how we should write commit messages in order to make it more readable and organized. It also make possible automatic changelog creation. The full documentation can be found here.

In a normal situation I would setup some gradle plugins like this commit lint plugin and this git hook plugin to ensure that everyone is following this pattern. I decided not to do it in this project to keep it simpler.

Composable components The UI parts were made as independent components in order to make ir more easily reusable.

State hoisting The composables were made to be the most stateless as possible. Receiving data models through the arguments and passing the event handling to the parent composable.

Flow exposure on ViewModel Flows are exposed on the ViewModel so the UI can observe it and react when the view model value changes.

Room DAO returning Flow Flows are being exposed by the DAO interface. This way we can take advantage of the Room-Coroutines integration and make our query observable. Doing this, everytime the query dataset changes, the UI is automatically updated.

UI Models During the message fetching, the `MessageEntity` objects are mapped to `ChatUiModel`. Thus the presentation layer receives only the data that will be shown, already parsed and user readable. Then the presentation layer can build the component list based on the models it receives.

In a bigger application, I would introduce another kind of model: the domain models. The domain models would be purely data classes. Without any framework specific annotation or reference.

So the domain logic would interact only the domain ones and we can detach all the database/ui/network specific classes from the business logic. I didnt do that because of time constraints.

Resource provider The resource provider was created to detach the Android resource manipulation classes from the mapping logic. This way we can test logic that uses android resources without having an actual Android `Context`.

Next steps...

- Component to be shown when there is no message on the list. Some icon and a text like "Say hi to your new match!"
- Confirmation dialog when the user clicks on back arrow.
- Domain models
- Detekt and Ktlint integration
- Commit lint integration