Muzz - Android Chat Exercise

Tech & Structure

- Kotlin, Jetpack Compose, Material 3
- MVVM + UDF
  - `ChatUiState` as a single source of truth for the screen
  - `ChatAction` sealed interface for all user events (typed event channel)
- Hilt for DI
- Room + Flow for local persistent storage and reactive updates

Feature-based package structure:

```
feature/chat/
    domain/        // Message, User, MessageRepository
    data/          // Room entities, DAO, MessageRepositoryImpl
    presentation/  // ChatViewModel, ChatUiState, ChatItem, ChatAction
    ui/            // ChatScreen and composable components
```

This keeps everything about "chat" together, without going full multi-module Clean Architecture, which would be overkill for a single-screen exercise.

Architecture & Data Flow

- Unidirectional Data Flow
- UI observes `StateFlow<ChatUiState>` from ChatViewModel.
- UI sends user events to the ViewModel as `ChatAction`.
- ViewModel transforms actions into repository calls + new `ChatUiState`.
- Composables only render state; no business logic or persistence code lives in the UI layer.

This keeps the screen easy to reason about and makes the state machine around message grouping/timestamps testable.

Repository

```
interface MessageRepository {
    fun observeMessages(): Flow<List<Message>>
    suspend fun sendMessage(text: String, from: User)
    suspend fun seedIfEmpty()
}
```

- `MessageRepositoryImpl` maps Room entities ⇄ domain models.

- `observeMessages()` returns a Flow from Room, so the UI automatically reacts to DB changes.
- `seedIfEmpty()` pre-populates the DB once so the conversation is not empty on first launch.

Room is treated as the single source of truth: the ViewModel never maintains its own "shadow" list of messages.

Message Grouping & Time Sections

The ViewModel converts raw messages into a flat list of `ChatItem`:

- Time separators are inserted when the gap from the previous message is more than 1 hour (formatted as `EEEE HH:mm`, e.g. Thursday 11:59).
- Bubble grouping: messages from the same user within 20 seconds are treated as a group and use compact spacing; otherwise they start a new group.

All of this logic lives in `buildUiItems` in the ViewModel. The UI only cares about `ChatItem` (separator vs bubble + spacing flags), which keeps layout code simple and makes the grouping rules easy to change or test later.

UI Decisions

- `ChatScreen` / `ChatScreenContent` split
  - `ChatScreen` owns the ViewModel (Hilt) and collects state.
  - `ChatScreenContent` is a stateless composable that takes `ChatUiState` + `(ChatAction) -> Unit`.
  - This keeps previewing/testing easier and avoids tying the UI directly to Hilt.
- Top bar (`ChatTopBar`)
  - Uses TopAppBar to leverage Material 3 defaults.
- Content is custom to match the PDF: back arrow, avatar + "Sarah" label, and a "more" icon.
  - For back/more, I use `Box` + `clickable` instead of `IconButton` so I can control the exact icon size and avoid the built-in padding that made the icons look smaller than in the design.
- Message list (`MessageList`)
  - `LazyColumn` + `rememberLazyListState`.
  - Automatically scrolls to the bottom when new messages arrive so the latest message is visible.
  - Adds a small spacer before the first item so the first bubble isn't glued to the top edge.
  - Draws subtle top/bottom gradient shadows to hint that the list is scrollable, without adding extra dividers or chrome.
- Bubbles (`MessageBubble`)
  - Alignment: `ME` → end (right), `SARAH` → start (left).

    - Uses `BoxWithConstraints + widthIn(max = maxWidth * 0.8f)` so:
        - Short messages only take minimal width.
        - Long messages are capped at 80% of the row, which matches the mock and reads better on large screens.
    - Colors come from `MaterialTheme.colorScheme`:
        - `primary` / `onPrimary` for my (outgoing) bubble.
        - `surfaceVariant` / `onSurfaceVariant` for Sarah's (incoming) bubble.
    - Outgoing messages show a `DoneAll` icon as a simple read receipt; it's purely visual and not wired to delivery states.
- Input bar (`MessageInputBar`)
    - Implemented on top of `BasicTextField` instead of `OutlinedTextField`.
        - `OutlinedTextField` would have simplified the code, but its built-in paddings and outline shape didn't match the PDF.
        - `BasicTextField` + a custom border lets me replicate the exact pill shape and inner padding from the design.
    - Border color reacts to focus (`primary` when focused, `outline` when not).
    - Uses a proper `IconButton` for the send action to keep touch target size and accessibility semantics, but with custom size and color to match the PDF; the button is disabled when the input is blank.
- Design tokens (`Dimens`)
    - Sizes, spacing, radii, and opacities are centralized in `Dimens` to avoid magic numbers in composables and to make future design tweaks easier.

Two-Way Messages

The second user is simulated:

- `ChatUiState.currentUser` tracks who is "active": `User.ME` or `User.SARAH`.
- Tapping the "more" icon triggers `ChatAction.OnMoreClicked`, which toggles `currentUser`.
- When sending a message, the current user is used and the message is written to Room; `observeMessages()` immediately pushes the updated conversation back to the UI.

This satisfies the "two-way messages" requirement without adding network calls or extra complexity.

Scope, limitations & what I'd add with more time

- The app intentionally models a single conversation between two fixed users; there's no navigation, login, or multi-chat support.
- Error/loading states are omitted to keep the exercise focused on layout, state handling, and persistence.

- `seedIfEmpty()` uses a hard-coded script to prefill the chat so the UI looks realistic on first launch.

If I had more time I would:

- Add unit tests for `ChatViewModel.buildUiItems()` (grouping, separators, edge cases).
- Extend the theme with a better dark-theme tuning.
- Add a few UX extras: per-message timestamps on long-press, animations on new messages, and accessibility refinements (TalkBack order, haptic feedback on send).