## Design Approach for Loading Initial Messages in the Chat Application:

**Startup API Fetch:** The application initiates an API request to retrieve the initial set of messages from the backend server when the chat screen is first rendered.

**State Management:** A custom message store (useMessageStore) is used to store the retrieved messages in a centralized state (in my case, Zustand). This method eliminates the need for needless prop drilling and guarantees that messages are readily available and updateable throughout the application.

**Effective Rendering:** The most recent messages appear at the bottom of the chat view, following standard chat UX patterns, because state messages are sorted (often in ascending order by sentAt) prior to rendering.

**Important Points:**

1. **Data Fetch in _layout.tsx**:
   The application leverages Expo Router's app/_layout.tsx file to trigger data loading as soon as the navigation stack mounts. This ensures that all required data (session info, participants, messages) is available before any child screens render.
2. **Centralized Initialization**:
   By using a useEffect in _layout.tsx, we call the store methods (fetchInfo, fetchParticipants, fetchLatest) once when the stack loads. This prevents duplicate or unnecessary requests.
3. **State Management (Zustand)**:
   Fetched data is stored in Zustand global stores, making it accessible to all screens/components without prop drilling or context chains.
4. **Why Not in App.tsx?**
   Putting initialization in App.tsx caused issues because navigation wasn't mounted yet leading to timing/race issues and unreliable data access. _layout.tsx guarantees the navigation context is ready, and children will always have fresh data.
5. **Async and Batched Fetch**:
   The three fetches are executed sequentially with await to ensure correct ordering and reduce race conditions (e.g., fetching participants before rendering messages).
6. **Offline Support**:
   After fetching, data can be cached locally (using AsyncStorage) for offline access and faster subsequent loads. This can be achieved by updating the Zustand fetch functions to write to/read from storage.