Captain’s Log

0) One-page overview

Core flows:

1. **Record** (press & talk) → audio chunks stream → live transcript appears → release to finalize.
2. **Notes** Viewable collection of notes across 4 tabs: personal, friends, public, for me (shared to me) - sorted by date.
3. **Friends** search + add a friend, share a note with selected friends, view a friend's profile.
4. **Search** search across note transcripts for keywords and provide summary capabilities.

High-level architecture:

- **Android app** → WebSocket to **Ktor API** for auth, streaming upload, and live translation.
- **Ktor server** → Routes audio to **Speech translation service** (ML) for partial/final text.
- **Postgres (+pgvector) service** Persist user, note, and friend data, and support search.
- **Object storage (MinIO)** Persist audio blobs.
- **LDAP service** Handle user creation and auth.

---

1) Android app

1.1 Screens & navigation

- **LoginScreen** (Enter username/password + login/signup)
- **RecordScreen** (Record button, live transcript, “Go to Notes” button)
- **NotesScreen** (Pannable tabs: personal, friends, public, for me; search field (filter notes on current tab by keyword))
- **FriendsScreen** (Search/Add by username)
- **ProfileScreen** (View selected user's profile and a list of their public notes)
- **NoteDetailScreen** (playback + transcript + timestamp + share (to pop-up with checklist of friends + share/cancel buttons)

1.2 Permissions & libs

- `RECORD_AUDIO`, `POST_NOTIFICATIONS`, `INTERNET`.
- Jetpack Compose, ViewModel, Kotlin Coroutines/Flows.
- Media3 (AudioRecorder/Player) or low-level `AudioRecord`.
- WebSockets using Ktor Client.
- JSON: Kotlinx Serialization.

1.3 Proposed package layout

```
ui/
  login/
  record/
```

```
  notes/
  friends/
  profile/
  note_detail/
  components/ (misc components)
data/
  api/ (serializable models)
  repo/ (repositories for handling server fetched data)
  model/ (view models)
```

1.4 Data models (client)

```
@Serializable
data class UserDto(val id: String, val displayName: String)

@Serializable
data class FriendDto(val userId: String, val friendId: String)

@Serializable
data class NoteDto(
  val id: String,
  val ownerId: String,
  val createdAt: Long,
  val transcript: String,
  val audioUrl: String?, //reference to blob storage url
  val durationMs: Long,
  val sharedWith: List<String> = emptyList()
)

@Serializable
//used for sending audio chunks for real time translation
data class TranscriptChunkDto(
  val noteSessionId: String,
  val sequence: Long,
  val text: String,
  val isFinal: Boolean
)
```

1.5 ViewModels

```
class AuthViewModel(private val repo: AuthRepository) : ViewModel() {
  val uiState: StateFlow<AuthState>
  fun login(username: String, password: String)
  fun create(username: String, password: String)
  fun logout()
}

class RecordViewModel(
  private val recorder: AudioRecorder,
```

```
    private val ws: HttpClient,
    private val notesRepo: NotesRepository
) : ViewModel() {
    val liveText: StateFlow<String>
    val isRecording: StateFlow<Boolean>
    val elapsedMs: StateFlow<Long>
    fun startRecording()
    fun stopRecording(): Result<String>
}

class NotesViewModel(private val repo: NotesRepository) : ViewModel() {
    val myNotes: StateFlow<List<NoteDto>>
    val sharedNotes: StateFlow<List<NoteDto>>
    fun refresh()
    fun search(query: String): Flow<List<NoteDto>>
    fun share(noteId: String, friendIds: List<String>)
}

class FriendsViewModel(private val repo: FriendsRepository) : ViewModel()
{
    val friends: StateFlow<List<UserDto>>
    fun searchUsers(username: String): Flow<List<UserDto>>
    fun addFriend(username: String)
}

class ProfileViewModel(
    private val userRepo: UserRepository,
    private val notesRepo: NotesRepository
) : ViewModel() {
    val selectedUser: StateFlow<UserDto?>
    val publicNotes: StateFlow<List<NoteDto>>
    fun loadUserProfile(userId: String)
    fun refreshNotes()
}

class NoteDetailViewModel(
    private val notesRepo: NotesRepository,
    private val friendsRepo: FriendsRepository
) : ViewModel() {
    val note: StateFlow<NoteDto?>
    val transcript: StateFlow<List<TranscriptSegment>>
    val isPlaying: StateFlow<Boolean>
    val friendsList: StateFlow<List<UserDto>>
    fun loadNoteDetail(noteId: String)
    fun play()
    fun pause()
    fun share(noteId: String, friendIds: List<String>)
}
```

1.6 Repositories

```
class NotesRepository {
  suspend fun myNotes(): List<NoteDto>
  suspend fun sharedNotes(): List<NoteDto>
  suspend fun publicNotes(): List<NoteDto>
  suspend fun getNote(noteId: String): NoteDto
  suspend fun share(noteId: String, friendIds: List<String>)
  suspend fun search(query: String): List<NoteDto>
}

class FriendsRepository {
  suspend fun getFriends(): List<UserDto>
  fun searchUsers(query: String): Flow<List<UserDto>>
  suspend fun addFriend(userId: String)
}

class TranscriptionRepository {
  suspend fun connect(): Unit
  suspend fun sendChunk(bytes: ByteArray, seq: Long)
  val transcript: Flow<TranscriptChunkDto>
  suspend fun close()
}
```

1.7 Composables (signatures)

```
@Composable fun LoginScreen(vm: AuthViewModel, onLoggedIn: () -> Unit)
@Composable fun RecordScreen(vm: RecordViewModel, onGoToNotes: () -> Unit)
@Composable fun NotesScreen(vm: NotesViewModel, onOpenNote: (String) ->
Unit)
@Composable fun FriendsScreen(vm: FriendsViewModel, onOpenProfile:
(String) -> Unit)
@Composable fun ProfileScreen(userId: String, vm: ProfileViewModel,
onOpenNote: (String) -> Unit, onBack: () -> Unit)
@Composable fun NoteDetailScreen(noteId: String, vm: NoteDetailViewModel,
onBack: () -> Unit)
@Composable fun FriendsSearchBar(onQuery: (String) -> Unit)
@Composable fun NoteCard(note: NoteDto, onShare: (String) -> Unit, onOpen:
() -> Unit)
```

1.8 Audio streaming

- PCM 16-bit mono, 16kHz (server resamples if needed).
- WebSocket binary frames with `{seq:uint64, ts:uint64}`.
- Client shows **partial** text; replaces with **final** text when segment ends.

---

2) Backend: Ktor API (Kotlin)

2.1 Auth model

- LDAP bind for username/password.
- Ktor issues **JWT** (access + refresh).
- All routes require `Authorization: Bearer <jwt>`.

2.2 API routes

```
POST   /auth/login
POST   /auth/signup
POST   /auth/refresh
POST   /auth/logout

WS     /transcribe
GET    /audio/{noteId}

POST   /notes
GET    /notes/my
GET    /notes/shared
GET    /notes/public
GET    /notes/{id}
POST   /notes/{id}/share
POST   /notes/{id}/visibility
PUT    /notes/{id}
DELETE /notes/{id}
GET    /notes/search?q=...

GET    /friends
POST   /friends
DELETE /friends/{id}
GET    /friends/requests
POST   /friends/requests/{id}/accept

GET    /users/search?q=...
GET    /users/{id}
GET    /users/{id}/notes/public
```

2.3 Services

- `AuthService` (LDAP bind + JWT)
- `TranscriptionService` (audio forwarding + transcript)
- `NotesService` (CRUD + sharing + presigned URLs)
- `SearchService` (keyword + pgvector semantic search)
- `StorageClient` (MinIO)

2.4 WebSocket protocol

- Client → Server: `AudioChunk(seq, bytes)`
- Server → Client: `TranscriptChunkDto` (partial/final)
- Stopping recording closes the session

---

3) Transcription Service

3.1 Model choice

- **Whisper (large-v3)** view on huggingface

3.2 API

```
POST /chunk (rapid translation)
GET /final (fetch full text translation)
```

3.3 Outputs

- Partial + final transcripts.

---

4) Data & Storage

4.1 Postgres schema

```
CREATE TABLE users (...);
CREATE TABLE friends (...);
CREATE TABLE notes (...);
CREATE TABLE note_shares (...);
ALTER TABLE notes ADD COLUMN tsv tsvector;
ALTER TABLE notes ADD COLUMN embedding vector(384);
```

4.2 Object storage (MinIO/S3)

- Audio stored as single object per note.
- Presigned URLs for playback.

---

5) Search design

1. Keyword search: `tsvector`.
2. Semantic search: `pgvector` cosine similarity.

---

6) Containers & networking

docker-compose services:

- `api` (Ktor)
- `translate` (ML audio translation)
- `postgres` (Persistence and search)
- `minio` (Audio persistence)

- `ldap` (authentiation)

Network: private Docker network; only `nginx` public.

---

7) Auth & Permissions

- JWT tokens validated per request.
- Owner or shared user may access note/audio.
- Presigned URLs expire quickly.

---

8) Recording & transcription lifecycle

1. App starts `sessionId`.
2. Streams chunks.
3. Server sends partials.
4. Partials pushed to UI
5. Stop recording → store transcript + embedding + audio.

11) Testing

- Unit tests: ViewModels, repos.
- Integration: routes, JWT, MinIO.