ChatGPT

Cliniko Voice App – Master Technical Build Guide

This document provides a comprehensive guide for implementing the **Cliniko Voice** prototype using **Rork** as the frontend technology. The goal is to build a mobile-first iOS app that lets clinicians authenticate with Supabase, input their Cliniko API key, select a patient, dictate a treatment note with real-time transcription via AssemblyAI, and submit that note to Cliniko. The guide covers the Rork development environment, screen-by-screen design, user flows, backend integration, real-time transcription handling, security best practices, component structure, and includes the original prototype feature plan for reference.

Overview of the Rork Development Environment

Rork is a platform that generates a cross-platform React Native app (using Expo) from a plain-English description [1]. Under the hood, a Rork app is a React Native project with Expo, using file-based routing (Expo Router) and TypeScript. It provides a starting point with common integrations (such as Supabase for auth and backend, and state management via Zustand or React Context). Key characteristics of the Rork environment include:

- **Project Structure:** Rork organizes the app in a conventional Expo structure. Screens are placed in the `/app` directory following Expo Router conventions (each screen is a file or folder in `app/`), there's a global layout file (`app/_layout.tsx`) for navigation configuration, and common features are modularized (contexts, hooks, components, etc.) [2] [3]. For example, a typical Rork project has folders for reusable `components/`, app logic in `services/` or `hooks/`, and configuration in `lib/` (like Supabase initialization).

- **Components and Screens:** Each screen is a React component (function) exported as default from a file in `app/`. Rork uses **Expo Router** for navigation, meaning screens can be nested in folders to represent navigational structure (e.g. `app/auth/sign-in.tsx` for a Sign-In screen). Reusable UI pieces are built as functional components in the `components/` directory (for instance, a `PatientListItem` component for displaying a patient in a list). Styles can be applied using React Native stylesheets or libraries (Rork often supports tools like NativeWind/Tailwind or plain StyleSheet API as seen in the generated code) [4] [5].

- **Routing and Navigation:** Navigation is declarative via Expo Router. Rork sets up a stack or tab navigator automatically. In this app, we will use a **Stack navigation** pattern for a linear flow (Login → API Key Input → Patient List → Dictation → Confirmation). The `app/_layout.tsx` file defines the root navigator and global providers. For example, a Rork app's root layout might wrap the app in context providers (Auth, global App context, QueryClient for React Query, etc.) and define a `<Stack>` with routes for main app screens and modals [6] [7]. We will leverage this to include our screens in the navigation flow (ensuring, for instance, that the Login and API Key screens appear modally or only when appropriate).

- **State Management:** Rork includes state management solutions out of the box. It typically uses **React Context** for auth and global app state (as seen with an `AuthContext` that wraps Supabase auth methods) [8] [9]. It also includes **Zustand** (a lightweight state management library) in its dependencies [10], which can be used for additional global state if needed (e.g., to manage UI state like the current transcription text or microphone status across components). For server state (data fetched from APIs), Rork projects integrate **@tanstack/react-query** for caching and synchronizing data from the backend [11] [12]. We can use React Query to manage data like patient lists or to refetch data after note submission, ensuring a responsive UI.

- **API Logic and Integration:** Rork apps are set up to call external APIs either directly from the frontend (using fetch/axios, often organized in a `services/` folder) or via backend functions. In our case, we have two external APIs: **Cliniko** and **AssemblyAI**. Rork's generated code includes examples of calling Supabase (which itself can act as our backend) and shows how to structure API calls in service modules. For instance, an `authService.ts` might wrap Supabase auth calls, and we can create similar services for Cliniko (e.g., `clinikoService.ts` to fetch patients or post notes) and for initiating the AssemblyAI stream. If needed, Rork also supports Supabase Edge Functions (serverless functions) in the project – the presence of a `supabase/functions` folder in Rork projects indicates we can offload certain logic to a secure backend environment if necessary. This could be useful for calling Cliniko or AssemblyAI from a server side to avoid exposing secrets.

In summary, working in Rork is much like working in any modern Expo/React Native project [13]. You write React components for UI, use hooks and context for state, and integrate APIs via fetch or Supabase. The benefit is that Rork jump-starts the app with initial code and configuration (Expo, Supabase setup, etc.), so you can focus on implementing features like our Cliniko Voice flow.

Screen-by-Screen Breakdown

We will implement the app through a series of screens, each corresponding to a step in the clinician's workflow. Below is a breakdown of each screen, including its purpose, components, navigation, and key UI/UX considerations:

1. Login Screen

Purpose: Allow the clinician to authenticate via Supabase using email/password or magic link. This is the entry point for users who haven't logged in.

Components & UI: The Login Screen (e.g., `app/auth/sign-in.tsx`) contains input fields for email and password and buttons to log in or request a magic link. It might also offer third-party login options if configured (though for this prototype we focus on email auth). In Rork's context, this screen uses the `useAuthContext()` to call Supabase auth functions [14] [15]. Key components include: - **Email and Password Inputs:** Use React Native `<TextInput>` fields for credentials. Include validation for proper email format and non-empty password, providing error messages below the fields if validation fails (as shown in the example Rork sign-in screen) [16] [17]. UX considerations: use `keyboardType="email-address"` for the email field and `secureTextEntry` for password [18] [19], and possibly a "show/hide password" toggle (e.g., an eye icon button) [20]. - **Login Button:** A primary call-to-action button that triggers login. When pressed, it calls a handler (e.g., `handleEmailSignIn`) which uses the Supabase client to sign

in. This may use a function like `supabase.auth.signInWithPassword({ email, password })` under the hood, but since our Rork environment likely wraps this in `signInWithEmail(email, password)` via an auth service [21] [22], we simply call the context method. If login fails, display an `Alert` or error message on screen [23]. - **Magic Link Option:** If using magic link, an alternative button can trigger `supabase.auth.signInWithOtp({ email })` which sends a login link to the email. In Expo, handling the magic link requires capturing the deep link (using Expo Linking). This adds complexity, so if included, we ensure the app is configured with a custom scheme (in `app.config.js`) and that the Supabase redirect URL is set. For prototype simplicity, email/password may suffice, but it's mentioned as an option. - **Sign-Up / Reset:** Optionally, link to a Sign-Up screen (if allowing self-registration) or a "Forgot password?" which calls Supabase's password reset email function [24] [25]. These improve UX but are ancillary to the main flow.

Navigation: After successful login, the user should be taken to the next step. In Rork/Expo Router, if the Login screen was launched modally (as in the example code) [7], we might simply `router.replace("/api-key")` or close the modal so that the next screen shows. Another approach is having the auth state control the initial route: for instance, the app's root layout could check `isAuthenticated` from `AuthContext` and either show the authenticated stack or redirect to `/auth/sign-in` [26] [27]. For clarity, we can implement it such that once `isAuthenticated` is true, the API Key Input Screen becomes accessible. On login success, calling `router.push("/api-key")` is an easy way to navigate forward.

UX Considerations: Keep the login form simple and clear. Provide feedback for loading (e.g., an `<ActivityIndicator>` when the login attempt is in progress) and errors. Since this is the first screen, it should also handle the case where the user might already be logged in from a previous session – in that case, it should immediately skip to the next screen. Supabase's session is persisted via AsyncStorage by default in Rork [28], so the app can auto-login if a session exists.

2. Cliniko API Key Input Screen

Purpose: Collect the clinician's Cliniko API key and store it securely for use in subsequent API calls. Cliniko's API uses an API Key for authentication (provided by the user from their Cliniko account), so we need this before fetching any patient data.

Components & UI: This screen consists of: - **API Key Text Field:** A single `<TextInput>` where the user pastes or types their Cliniko API key. The key is a long string (with region info, e.g., `MS0xNz...-auX` format). To improve UX, use a monospaced font or smaller text to accommodate long strings, and consider a "show/hide" toggle if treating it like a password (though it's not a password, hiding it from shoulder-surfers might be good). - **Save/Continue Button:** A button that when pressed, validates and saves the key. Validation can simply check that the input isn't empty and perhaps matches an expected pattern (Cliniko keys seem to start with `MS` and contain a region suffix like `-au1`, `-au2`, etc. – optional to validate this format). On save, we store the key for later use.

Storing the API Key securely: This is critical. Options: - **Secure Storage on Device:** Use Expo's SecureStore or encrypted storage to save the API key locally so it persists but is protected. This ensures the key isn't in plain AsyncStorage. If SecureStore is not already in the project, it can be added (Expo provides `expo-secure-store`). Usage would be e.g. `SecureStore.setItemAsync("clinikoApiKey", keyString)`. - **Supabase Database:** Alternatively, store the key in Supabase under the user's profile (for example, add a column in a `profiles` table for `cliniko_api_key`). If using this, ensure the Supabase policy restricts

that data to only the authenticated user (Supabase Row-Level Security can enforce that). Storing in the database allows the key to roam with the user (accessible from multiple devices once logged in) but introduces greater responsibility for protecting it (e.g., encrypt it in the database or mark it as sensitive). - **State Management:** Regardless of persistence, also keep the key in memory during the session (e.g., in a React Context or Zustand store) so that API calls can easily access it. We might create a simple context or include it in an existing App context that holds `clinikoApiKey` (populated after this screen).

For simplicity, a good approach is: store in SecureStore and also set it in a global state variable. That way, after input, we can immediately use it to fetch patients and don't have to prompt every time.

Navigation: After saving the key, navigate to the Patient List screen (`router.push("/patients")`). We should only allow proceeding if a key is provided. Also, it's wise to test the key quickly: upon submission, we could attempt a small API call to Cliniko (e.g. fetch the first page of patients) to verify the key is valid. If the call fails (401 Unauthorized), we alert the user that the key is invalid, and do not navigate forward.

UI/UX Notes: Include a brief description or link on how to get their Cliniko API key (many users might not know where to find it). For example, a text like "Find your API key in Cliniko (Log into Cliniko > My Info > Manage API Keys)" [29] can guide them. If possible, provide a copy-paste friendly field (on iOS, ensure autocapitalization is off and maybe use `autoCorrect={false}` to avoid any keyboard alterations). The screen should reassure users that the key will be stored securely and only used to fetch their data.

3. Patient List Screen

Purpose: Allow the clinician to select a patient from their Cliniko account for whom they want to create a treatment note. This screen displays the list of patients retrieved from Cliniko using the provided API key.

Data & Components: On component mount, it should fetch the patient list from Cliniko's API. Cliniko has an endpoint `GET /patients` that returns a list of patients in paginated form [30]. The request must include the API key in the Authorization header (Basic Auth with the API key as username) [31]. We can implement this fetch in a service or directly using React Query: - For example, define a React Query hook `usePatients()` that uses `fetch` to call Cliniko. The fetch would be like:

```
const res = await fetch(`${baseUrl}/patients`, {
  headers: {
    "Authorization": "Basic " + btoa(clinikoApiKey + ":"),
    "Accept": "application/json",
    "User-Agent": "ClinikoVoiceApp (user@example.com)"
  }
});
```

Cliniko requires a `User-Agent` header with an email or app name [32] and the Accept header. The API key is sent as basic auth (base64 encoded key + colon) [31]. Using React Query, we'd call this in `useEffect` or a query on screen load. If using a Supabase Edge Function instead (see Backend Integration), that function would do the above and return data to the app.

- **Patient List UI:** Use a **FlatList** or **SectionList** to display patients. Each list item can show the patient's name (and possibly other info like ID or email if needed). Likely just the full name is enough for selection.
- Implement a `PatientListItem` component for reusability: it could be a `TouchableOpacity` row containing the patient's name, possibly an icon/avatar placeholder. When pressed, it triggers selection.

- If the list is long, adding a search bar at the top can improve UX (Cliniko's API supports query filtering by name [33], which we could use to implement search-as-you-type or simply filter the fetched list on the client side).

- **State and Loading:** While fetching patients, show a loading indicator (spinner or skeleton list). If the fetch fails (e.g., wrong API key or network issue), catch the error. In case of 401 Unauthorized, redirect back to API Key screen with an error message (the key might have been invalidated or expired). For other errors, show a retry option.

Selection Navigation: When a patient is tapped, navigate to the Dictation/Note screen for that patient. We need to pass along which patient was chosen. In Expo Router, we can do this by navigating to a route with a parameter or by using global state: - A simple method is to navigate to e.g. `/dictate` screen and have a global context/store hold the current patient (set it on tap). Or, - We could use route params: define the dictation screen at `app/dictate/[patientId].tsx`. Then do `router.push({ pathname: "/dictate/[patientId]", params: { patientId: id }})`. But managing complex objects via route params is not ideal, so storing the selected patient object in context (or a Zustand store) may be easier. Given Rork already uses context for Auth, we might extend an AppContext to hold `selectedPatient` and a setter.

UX Considerations: If the patient list is large, you might lazy load (Cliniko paginates results). But for a prototype, loading all patients is acceptable. Provide feedback if no patients are found ("No patients available" message). Also, show the patient's name clearly; if multiple patients have similar names, perhaps show an additional identifier (like ID or DOB) in smaller text to avoid confusion. Keep touch targets large enough for mobile.

4. Dictation/Note Screen

Purpose: Enable the clinician to dictate a treatment note via voice and see it transcribed in real time. The clinician can then review and confirm the transcribed note.

Layout & Components: This screen is the core of the app's functionality, combining audio capture, live transcription, and text editing: - **Patient Context:** At the top, display the selected patient's name (e.g., "Recording note for **John Smith**"). This confirms to the user which patient's note they are about to create. Possibly include a "back" option to re-select a patient if needed. - **Transcription Display:** A large text area or scrollable `<Text>` view that shows the real-time transcription of the clinician's speech. Initially, it might show a placeholder ("Tap the microphone and start speaking..."). As the user speaks, partial transcription results should appear here and update continuously. It's often useful to distinguish partial (in-progress) text from final text (for example, partial words could be italic or lighter color until finalized) [34]. Once the AssemblyAI API designates a final transcript for an utterance, that text can be rendered in normal weight,

and new partial text appears after it for the next segment. - **Microphone Button:** A prominent button to start/stop recording. Typically, this is a round button with a microphone icon. When recording is active, it could be red or filled, and when stopped, a different state (or a stop icon). This button toggles the audio streaming. The first tap will: 1. Request microphone permission if not already granted (using Expo's `Audio.requestPermissionsAsync()` or the Permissions API). 2. Begin capturing audio from the device microphone. 3. Open a WebSocket connection to AssemblyAI's streaming transcription API. 4. Stream the audio data to AssemblyAI and receive transcripts.

When tapped again (stop), it should stop audio capture and send a message to end the AssemblyAI stream (which will then return any final transcript segment and close the connection).

- **Control Buttons:** Besides the mic, include a button to **Submit** the note (enabled only when not currently recording, and perhaps only if there is some transcribed text). Also, you might include an **Edit** or **Clear** option:
- *Edit:* If the user sees a minor mistake in the transcription, allowing them to tap the text and edit it manually would be useful (maybe in a multi-line TextInput). This can be optional in a prototype, but it's a good UX consideration since speech recognition isn't perfect.
- *Clear:* In case they want to discard and re-dictate, a clear or reset button can wipe the text and reset the transcription session.

Real-Time Transcription (AssemblyAI Integration): This is the technical heart of this screen. AssemblyAI offers a realtime **WebSocket API** for streaming speech-to-text, which returns **partial and final transcripts** with very low latency [35] . Here's how to handle it: - **Starting the Stream:** When the user taps record, create a WebSocket connection to AssemblyAI's endpoint (e.g., `wss://api.assemblyai.com/v2/realtime/ws?sample_rate=16000&token=YOUR_API_TOKEN`). AssemblyAI requires an API token for auth; include it in the connection URL or headers. (If we want to avoid exposing this token, see the security section or consider a proxy). - **Capturing Audio:** In an Expo/RN app, capturing raw audio in real-time may require a native module. Expo's `Audio.Recording` API typically records to a file, not giving raw buffers easily in real-time. For streaming, one approach is to use a low-level audio API or a third-party native plugin that provides live PCM data. If we limit to iOS (as specified mobile-first iOS), we could use e.g. `expo-av` on iOS which might allow a recording callback. If Rork doesn't provide a built-in solution, we might integrate a community module (such as **react-native-audio** or **react-native-webrtc** to capture a microphone audio stream). Assuming we have a way to get audio chunks (16kHz PCM mono as required by AssemblyAI), we will continuously send them over the WebSocket. - **Receiving Transcripts:** Once the stream is open, AssemblyAI will start sending JSON messages for transcriptions. Each message will indicate if it's a partial transcript or a final transcript for a segment [36] . For example, it may send `{ "text": "hello wor", "message_type": "PartialTranscript" }` and then `{ "text": "hello world", "message_type": "FinalTranscript" }`. Our app should update the UI accordingly: - Append or update the text area with the `text` from each partial message. It's common to replace the last portion of text as the transcript gets refined (the API might send the full transcript so far, or just the new part – the exact format can vary, but typically one might replace the last partial). - When a Final transcript message arrives (`message_type: "FinalTranscript"`), treat that portion of text as finalized. Perhaps append a newline (if each utterance is separate) or simply continue the paragraph. Mark that segment as final (we no longer update it). - **Stopping the Stream:** When the user stops recording (or if they pause speaking for long enough), we'll close the WebSocket. AssemblyAI will send a termination message or final batch if needed. At this point, the full note text is available in the UI for review.

Throughout this process, provide feedback: - While recording, maybe show a “Listening…” indicator or an audio waveform animation to indicate the app is capturing audio. - Possibly show partial text in gray and then turn to black when finalized to indicate confirmed text. - If any error occurs (e.g., lost internet, or AssemblyAI returns an error), show an alert to the user and stop the transcription.

Review & Edit: After stopping, the user should review the transcribed note. They can tap on the text to edit any mistakes. Implement the text display as an editable text field when not actively transcribing: for example, you could toggle between a Text component and a TextInput component on tap (or have an “Edit” mode). This way, the clinician can correct medical terms or formatting before submission. Keep this editing intuitive – maybe auto-focus the TextInput if entering edit mode, and provide a “Done” button to finish editing.

Navigation: The next step is submission. When the user is satisfied and taps **Submit**, we will call the Cliniko API to create a treatment note for the selected patient, then navigate to the Confirmation screen. (Alternatively, the Confirmation screen could be skipped if we decide to just use an alert, but a screen is nicer for UX and extensibility.)

5. Confirmation Screen

Purpose: Inform the user that the treatment note was successfully submitted to Cliniko (or show the result of the submission) and conclude the workflow. It may also present some details of the saved note or next steps.

Components & UI: This is typically a simple screen with a success message: - A checkmark icon or success illustration could be shown (if available). - Text like “**Success!** Your treatment note has been saved to Cliniko.” Possibly include the patient’s name and maybe timestamp. - Perhaps show a snippet of the note content for confirmation (so the clinician knows which note was submitted, especially if they plan to dictate multiple notes in a session). - If the app allows multiple sequential notes, provide a button to “Create Another Note” which could take them back to patient selection (or directly to dictation for the same patient if desired). Or if done, provide a way to go back to main menu or log out.

Submission Details (from previous screen): When navigating to this screen, we should pass or fetch the result of the note submission: - If the submission was successful, maybe pass a flag or the created note’s ID to display. - If the submission failed (network issue or API error), perhaps this screen can show an error message instead, with a “Try Again” option. Alternatively, handle errors on the Dictation screen itself with a retry without navigating to a success page.

Cliniko Treatment Note Creation: To recap, on hitting Submit in the Dictation screen, we perform: - A POST request to Cliniko’s `treatment_notes` endpoint. According to Cliniko API, creating a treatment note likely requires a patient ID and a treatment note template ID and content [37]. In the Cliniko API, a Treatment Note must usually conform to a template (the template defines the fields of the note). For simplicity, the prototype might use a predefined template that has just a free-text field for the note content. We’ll need the template’s ID. This could be set in configuration or fetched from Cliniko if not known. (Cliniko’s API has endpoints for treatment note templates [38].) - The POST JSON might look like: `{ "patient_id": 123, "treatment_note_template_id": 456, "content": "the transcribed note text..." }` (actual field names may differ; checking Cliniko API docs is necessary for exact format). - Include the same auth headers as earlier (API key via Basic Auth, etc.) [39].

If using a Supabase Edge Function to handle Cliniko calls, the function would accept e.g. patient_id and note content and perform this POST using a secret Cliniko key (though since the key is user-specific, probably we just send the key as well or retrieve it from DB in the function). For a prototype, a direct call from the app is fine, just be mindful of security.

Navigation/State: After showing confirmation, if the user taps “Done” or “Back to start”, we might navigate them to either the Patient List (if they want to do another note) or maybe all the way back to some home screen. Since our flow is linear, perhaps going back to Patient List is logical (allowing another note for possibly a different patient). We should also clear or reset any state (like the transcribed text and selected patient) if starting over.

UX Considerations: This screen should reassure the user that their action is complete. Use clear, positive language for success. If we include any identifiers (like Cliniko’s note reference or so), that might help them later. If the prototype plan is just one note at a time, this screen might also suggest that the user can close the app or log out now if they’re done. Provide a log-out option somewhere (perhaps in a menu or on this screen) so the clinician can securely end their session if on a shared device.

Each of these screens and their transitions form the backbone of Cliniko Voice. Next, we detail how the data flows through these screens and how we manage the state as the user moves through the app.

User Journeys and State Flows

This section describes the end-to-end user journey and how the app’s state transitions at each step. We outline the primary flows: user authentication, patient data retrieval, voice dictation, and note submission. Understanding these flows ensures the app’s screens and components work together coherently.

1. Authentication Flow: When the app launches, it checks if the user is already authenticated (Supabase can restore a saved session automatically) [40] [41] . - **Not Authenticated:** The app will present the Login screen. The user enters credentials and triggers Supabase auth. On success, Supabase returns a session (with user ID, access token, etc.), which the app stores in the `AuthContext` state (`user` and `session` become non-null) [42] [43] . The app then transitions to the next step (API Key input). The `AuthProvider` context now marks `isAuthenticated` = true so guarded routes become available. - **Authenticated (Returning User):** If a session token is still valid, the app may bypass login. In our navigation logic, the root layout could detect a valid session on load and navigate directly to either the API Key screen or Patient list. However, we need to consider if the API key is already stored from before. If yes, the user might jump straight to patient list without needing to re-enter the key (improving UX). We can implement a check: e.g., in AppContext or a splash screen, attempt to retrieve a stored Cliniko API key. If found, skip to Patient List; if not, go to API Key screen. - **Sign Out:** The user should be able to log out (perhaps from a menu or profile area not detailed in our screens). Logging out would clear the Supabase session (`supabase.auth.signOut()`), which our AuthContext listens for and sets `user` to null [44] . The app then should navigate back to the Login screen and also clear any stored Cliniko API key from secure storage for safety.

2. Cliniko API Key Flow: After login, the user enters their Cliniko API key. - The state at this point includes the Supabase user (so we know who is logged in, possibly we have a user ID for storing any user-specific

data). The API key provided is tied to the Cliniko account of that user (not to Supabase user directly, but we assume one-to-one mapping). - On submitting the key, we store it (in SecureStore and maybe in a context state like `clinikoApiKey` within an AppContext). The journey can branch here: - **Key already on file:** It's possible the key was stored from a previous session (as mentioned). If so, the user might not even see this screen. But if they do (maybe they want to update it), saving it should overwrite the old one. - **Testing the Key:** Optionally, right after storing, the app can attempt a lightweight Cliniko API call (like GET `/patients?per_page=1`) just to verify the key works. If it fails, inform the user and keep them on this screen to re-enter. - Once a valid key is confirmed, proceed to fetch patients.

3. Patient Data Fetch & Selection Flow: When the Patient List screen loads, it triggers data fetch: - The app uses the stored Cliniko API key to call `GET /patients` (either directly or via an edge function). On the first load, we might set a loading state `patientsLoading = true`. When data returns successfully, `patients` state is populated (an array of patient objects). - We should also handle pagination if there are many patients: Cliniko's response likely includes a way to get the next page (like `links.next` or similar). A simple implementation might fetch all pages in sequence (loop until no more pages) and combine results. Or implement infinite scroll: load more when the list scrolls near bottom. - If the fetch fails (e.g. network down or invalid key), show an error state. If it's an auth error from Cliniko (key revoked), consider redirecting back to API Key screen with a message. - Upon the user selecting a patient, update the state to record which patient is active. For instance, set `selectedPatient` in a global context/store. Also, possibly store `selectedPatientId` in a local state of the dictation screen via route params. The journey then continues to dictation.

4. Dictation & Transcription Flow: This flow is a bit more complex, involving real-time updates: - **Starting Dictation:** User taps the mic to start. We update a state `isRecording = true` (could be local component state or global if other components need to know). We might also set up some state for the transcription text, e.g. `transcriptText` (initially empty). - **Streaming State:** As the audio streams, we continually update `transcriptText`. One way is to keep the full text in state and replace it whenever a new partial transcript arrives. Another approach is to maintain two pieces of state: `finalText` and `partialText` – where `finalText` accumulates all confirmed speech and `partialText` holds the latest interim words. Then in the render, you display `finalText + partialText` concatenated, styling partialText differently. For example:

```
const [finalText, setFinalText] = useState("");
const [partialText, setPartialText] = useState("");
```

On a partial result message: `setPartialText(message.text)`. On a final result:

```
setFinalText(prev => prev + (prev.endsWith(" ") ? "" : " ") + message.text + "
");
setPartialText("");
```

This way, each final segment is appended to the accumulated text. (Ensure spacing is handled so words don't mash together.) - **Voice Segmentation:** AssemblyAI's streaming will typically finalize transcripts when the speaker pauses (end-of-utterance) [45]. The state `partialText` will go to empty when final text moves to `finalText`. This segmentation might insert punctuation or line breaks depending on AssemblyAI's

capabilities (some models auto-punctuate). - **Stopping Dictation:** When user taps stop, `isRecording` goes false. We then close the WebSocket connection. We should also stop the device's recording mechanism. The final transcript chunk might come in just as we stop, so ensure we handle any last message. - Now the `transcriptText` is ready in full (in whatever state we kept it, probably in `finalText`). We could merge final+partial if partial had something unfinalized (though AssemblyAI usually finalizes everything on end-of-stream). - At this point, the user journey is to review the note. If the user is satisfied, they hit Submit. If not, they could press record again to append more (this would open a new stream and continue from where left off – or we could clear and restart). For MVP, it might be one-shot recording; for a nicer UX, allow multiple segments: e.g. a "Append" or just reusing the mic to continue recording, which would open a new stream and append further text.

5. Note Submission Flow: On submit: - Trigger a loading state (e.g., disable inputs, show a spinner on the submit button or nearby). - Use the Cliniko API to create the treatment note. We need the `patient_id` (from the selected patient) and the note content. Also the `treatment_note_template_id`: this could be a constant configured earlier. For example, say we have a template called "Voice Note Template" with ID 789. We would include that. - If calling directly from the app, use `fetch` with method POST to `.../treatment_notes` with JSON body. Authorization header with API key as before [39]. If using an edge function, call it (e.g., via `supabase.functions.invoke("create_note", {...})` with necessary data). The edge function approach lets us keep the API key usage off the client after the initial fetch (the function could fetch the key from DB by user ID), but it's more setup. In our prototype, direct fetch is acceptable with caution. - On success (HTTP 201 Created), the Cliniko API will return the created note object or at least a 200-series status. At this moment, update any relevant state (if we track notes or if we want to mark that patient now has an extra note – though that might be beyond our scope). - Navigate to Confirmation screen with a param or context indicating success. Possibly pass along the note content or ID to display. Reset transcription state (e.g., clear `finalText` state in the dictation component so it's fresh if they go back). - On failure (non-2xx response), handle it: could stay on the dictation screen and show an error alert ("Failed to save note. Please check your internet and try again."). Only navigate to confirmation if success. Alternatively, navigate to a Confirmation screen that is an error variant. But usually staying and letting the user retry submit is better.

State Summary Across Screens: We can maintain a simple global state in an `AppContext` or similar: - `auth.user` – from AuthContext (Supabase user info). - `auth.session` – Supabase session/token. - `clinikoApiKey` – stored string. - `selectedPatient` – object or at least ID of the patient chosen. - `transcriptText` or similar – we might keep this local to the dictation screen since it's not needed elsewhere. - Possibly a global `patients` list state if we want to cache it (React Query would cache it anyway). React Query by default would store the patients list in memory cache keyed by the query (e.g., `["patients"]`), so if we navigate away and back, we don't necessarily re-fetch unless stale.

The user journey in summary: 1. **Launch app** – If not logged in, go to Login. If logged in, skip to API Key (or Patient list if key is known). 2. **Login** – Supabase Auth, then navigate forward. 3. **Enter API Key** – Save key securely, maybe verify, then navigate. 4. **Load Patients** – Fetch and display list. User selects one. 5. **Dictate Note** – Start recording, stream to AssemblyAI, live transcript appears. Stop recording, review text. 6. **Submit Note** – Send to Cliniko. On success, show Confirmation. 7. **Confirmation** – Note saved. Option to do another or logout.

Throughout these flows, the app's state transitions ensure that: - An unauthenticated user cannot reach patient data or dictation (AuthContext `isAuthenticated` gating those screens). - A user without entering

an API key cannot fetch patients (we ensure `clinikoApiKey` is set before patient list). - A user cannot submit without selecting a patient and having transcript text. - After logout, all sensitive state (API key, transcripts) is cleared from memory.

Backend Integration Points

This prototype relies on integration with three backend services: **Supabase** (for authentication and optional data storage), **Cliniko API** (for patient data and submitting notes), and **AssemblyAI** (for speech-to-text). Here we outline how and where these integrations occur in the app, including alternative approaches and best practices.

1. Supabase (Authentication & Storage): - *Authentication:* Supabase provides email/password auth, which we use on the Login screen via the Supabase JS client. In the Rork environment, the Supabase client is initialized in `lib/supabase.ts` with the project URL and anon key [46]. It's configured to use React Native's AsyncStorage for storing the session token securely and to auto-refresh tokens [28]. We interact with it through context hooks (e.g., `useAuthContext().signInWithEmail`) which internally call `supabase.auth.signInWithPassword` etc. Supabase handles the heavy lifting of verifying credentials and maintaining session state [40] [47]. The app should also be set up to handle Supabase's deep linking if magic links are used (i.e., capturing the link and passing the `access_token` to `supabase.auth.setSession()`), but that requires configuring the redirect URL in Supabase and Expo's linking, which can be done via `expo-linking` and `app.config.js` (beyond basic scope). - *Database & Edge Functions:* While not strictly required, Supabase can store the Cliniko API key or notes if desired. A secure approach is to create a table `cliniko_keys (user_id UUID references auth.users, api_key text encrypted)` and an RPC or function to retrieve it. Then the app could just call a Supabase function like `getClinikoPatients()` without exposing the key. Supabase Edge Functions are written in TypeScript (Deno runtime) and can easily make fetch calls to external APIs, including Cliniko and AssemblyAI. In fact, Rork's inclusion of **Hono** (a minimal web framework) suggests a pattern: building a Supabase Edge Function to serve as a proxy or custom backend logic. For example, we could have: - `functions/fetch_patients.ts`: which reads the user's Cliniko key from DB and calls Cliniko's `/patients`, returning the list. - `functions/create_note.ts`: which takes patientId and note content, and uses the stored key to POST to Cliniko. - `functions/assembly_token.ts`: perhaps to generate a short-lived AssemblyAI token if needed (AssemblyAI might not need this; usually the API token is static).

Using edge functions means our mobile app calls these via `supabase.functions.invoke("fetch_patients")` rather than hitting Cliniko directly. This keeps API keys off the client after initial entry and allows implementing any server-side logic (like logging or transforming data). The tradeoff is complexity and a slight performance hit (extra hop via Supabase). For a prototype, direct calls are fine, but for production security, this pattern is recommended.

- *Usage in App:* The Supabase integration point in the app is primarily at login (auth) and possibly at storing the API key. If we decide to store the key in Supabase DB, then the API Key screen upon submission would call something like:

```
supabase.from('cliniko_keys').upsert({ user_id: user.id, api_key:
encryptedKey });
```

and then our data fetch calls would go through functions that retrieve this key. Without that, the app just keeps the key locally and uses it directly.

2. Cliniko API (Patients & Treatment Notes): - *Direct API Calls:* Cliniko's API is RESTful and expects Basic Auth with the API key [31]. All calls are over HTTPS to a region-specific endpoint (the API key itself encodes the region, e.g., `api.au1.cliniko.com` for -au1 keys [48]). We integrate with it by using `fetch` in JavaScript. There is no official JavaScript SDK to my knowledge, so raw REST calls are the way. We must include required headers in each request: - `Authorization: Basic <base64(api_key:)>` (no password, just a colon) [31]. - `Accept: application/json` [32]. - `Content-Type: application/json` for writes [32]. - `User-Agent: <AppName> (<email>)` as per Cliniko's requirements [32]. We can use the user's email from Supabase or a generic contact email for our app here.

In the app, these calls can be wrapped in functions: e.g.,

```
async function fetchPatients(apiKey: string): Promise<Patient[]> {
  const res = await fetch(`${clinikoBaseUrl}/patients`, { headers:
{...authHeaders} });
  if(!res.ok) throw new Error(await res.text());
  return res.json();
}
```

Similar functions for creating a note:

```
async function createTreatmentNote(apiKey: string, patientId: number,
templateId: number, content: string) {
  const body = JSON.stringify({ patient_id: patientId,
treatment_note_template_id: templateId, content });
  const res = await fetch(`${clinikoBaseUrl}/treatment_notes`, { method:
'POST', headers: {...authHeaders, "Content-Type":"application/json"}, body });
  if(!res.ok) throw new Error(await res.text());
  return res.json();
}
```

These can be placed in a `clinikoService.ts` and used in React Query or directly on events.

- *Via Supabase Function:* If using an edge function, the integration moves to the server side. For instance, a `fetch_patients` function would do the same fetch logic in the Deno environment (which supports fetch). It would need the API key; it can get it from an environment variable (not ideal since key is user-specific), or from the database as mentioned. The function then returns the data (the Supabase function acts as a tiny proxy). The mobile app calls `await supabase.functions.invoke('fetch_patients')` and gets the result. This adds security (key isn't exposed in app network calls, except perhaps in that first upsert) and possibly performance downsides if the Supabase edge location is far from Cliniko's servers (likely negligible).

- *Which to choose:* For this prototype documentation, we can mention that direct calls are simpler to implement and fine for initial testing (with the known risk that the API key is in memory and used on device). For a more secure approach or to avoid CORS issues (if in a web context later), the Supabase edge function approach is advisable.

3. AssemblyAI (Real-time Transcription): - *API Authentication:* AssemblyAI requires an API token for any requests. Typically, you'd obtain a token from your AssemblyAI account and include it in a header (`Authorization: Bearer YOUR_TOKEN`) for their standard API. For the streaming WebSocket, AssemblyAI's docs indicate including the token in the query params or as a header upon connection [36]. We have to integrate this securely: - Hard-coding the token in the app is okay for development, but in production, this is a secret that could be extracted from the app binary. A safer method is to store it in a secure location or fetch it from a secure source at runtime. We might, for example, store it in Supabase (as a secret accessible to edge functions or even in `Remote Config` if such existed). Alternatively, since AssemblyAI might allow creating ephemeral session tokens, a supabase function could generate a short-lived token to use. - For now, assume we use a config value. Possibly, in the Expo app config, we set `ASAI_API_KEY` as an environment variable (but be cautious: Expo public env vars will be embedded; better use a private config not exposed to client or supply at runtime).

- *WebSocket Streaming:* The integration with AssemblyAI is done client-side (unless we create a proxy server to relay audio which is unnecessary). So in the app:
- We use the WebSocket API available in React Native (which is basically the browser WebSocket polyfill). Example:

```
const ws = new WebSocket(`wss://api.assemblyai.com/v2/realtime/ws?
sample_rate=16000&token=${ASSEMBLYAI_TOKEN}`);
ws.onopen = () => { /* start sending audio */ };
ws.onmessage = (message) => { /* process transcription */ };
ws.onclose = () => { /* cleanup */ };
```

- When `ws.onopen` fires, we need to pump audio data. Ideally, we get PCM audio chunks from the microphone at 16kHz. If we had an audio library that provides a stream callback (e.g., every X milliseconds gives a chunk of PCM bytes), we'd take those bytes and do `ws.send(audioChunk)`. The data should be sent in the format the API expects (likely raw PCM or a specific container like WAV chunks). AssemblyAI's docs should specify if the audio must be base64 or raw PCM in the WebSocket – typically, it's raw binary PCM 16-bit little-endian. We ensure our audio recorder provides that format. If not, some conversion may be needed.

- As messages arrive (`ws.onmessage`), they come as JSON strings. We parse them and update state accordingly as discussed.

- *AssemblyAI Edge Cases:* The streaming will have a session timeout (often 5 minutes or so per connection). For dictation of a single note, that's fine (notes likely shorter). If the user speaks for an extremely long time, the socket could close – for a prototype, not likely an issue.

- We should handle if `ws.onclose` happens unexpectedly (maybe show a message "Transcription session ended. Tap the mic to continue.").

- Also handle `ws.onerror` (if network drops).
- Another integration point is AssemblyAI's *punctuation and formatting options*. By default, Universal-Streaming might add punctuation. We can enable or disable certain features via query params or an initial config message. For instance, AssemblyAI allows parameters like `format_text=true` to get nicely formatted text. We might want to ensure those settings to reduce the need for manual editing.

- *Testing in Simulator:* Note that if a developer is running in an iOS simulator, microphone might not work or need special handling. On device, ensure to ask permission. This integration should be tested thoroughly on real devices for accuracy and performance.

Summary of Integration Points in Code: - **Supabase**: initialized in `lib/supabase.ts` [28], used in AuthContext and possibly for data calls. - **Cliniko**: integrated via either direct fetch in service functions (called from PatientList and Submit actions) or via Supabase function calls from those places. - **AssemblyAI**: integrated in the Dictation screen component via WebSocket and a native audio module. This part of the code will be most intricate and platform-specific. If Rork has a plugin, use it; otherwise, plan to use Expo's custom development build to add a native module for audio if needed (since standard Expo might not support stream recording out-of-the-box). The **Expo Dev Client** can include modules like `react-native-audio` or `react-native-voice` if needed, which requires building a dev build (Rork's documentation suggests using EAS for custom builds when adding native modules [49] [50]).

Real-Time Transcription Handling in Rork

Implementing real-time transcription requires combining React Native capabilities with AssemblyAI's streaming API. This section focuses on **how** to handle audio streaming and transcription within the Rork app:

- **Accessing Microphone in React Native:** Rork (via Expo) uses the Permissions API to request microphone access. Before starting recording, call `await Audio.requestPermissionsAsync()` (from `expo-av`) or the newer `ExpoPermissions.askAsync(Permissions.AUDIO_RECORDING)`. Once granted, an `Audio.Recording` instance can be prepared. In standard Expo, you might do:

```
const recording = new Audio.Recording();
await
recording.prepareToRecordAsync(Audio.RECORDING_OPTIONS_PRESET_HIGH_QUALITY);
recording.setOnRecordingStatusUpdate(status => { ... });
await recording.startAsync();
```

  However, `OnRecordingStatusUpdate` won't give raw PCM data; it mainly gives status like duration. We need to retrieve the audio data. Unfortunately, Expo's managed workflow doesn't surface the live PCM frames. We may need to **eject** or use a custom module for this specific need, unless using a workaround.

- **Possible Workarounds:** One approach for streaming without ejecting is to use a WebRTC library: `react-native-webrtc` can capture microphone audio as a stream and then we could pipe that stream's data into a WebSocket. But integrating that in Expo might require a custom dev build (since it's a native module).

- **Rork Toolkit:** Rork is positioned for AI apps; it's possible the Rork toolkit (`@rork-ai/toolkit-sdk`) might include some utility for audio or an example of voice transcription. If such a utility exists, we should use it. If not, we proceed with custom integration.

- **Native Module Approach:** For iOS, we could write a small native module that uses `AVAudioEngine` to get the microphone input and send PCM to JavaScript. This is advanced and likely outside the scope of what a single engineer would do in the initial prototype timeframe. Instead, we might choose a simpler approach: use short recordings in sequence. For example:

- Record audio for a short interval (say 2 seconds), then stop and fetch the PCM buffer, send to AssemblyAI, then immediately resume recording next chunk. This would create small gaps but might suffice if continuous streaming is too hard.

- **Handling Partial vs Final Transcripts:** As AssemblyAI sends messages, we update the UI:

- **Partial Transcripts:** display immediately, replacing the prior partial. The user will see the text almost as they speak, with minor lag of a few hundred milliseconds [35].

- **Final Transcripts:** once a final segment is received (AssemblyAI typically indicates an `is_final:true` or similar flag), we append it to the transcript permanently. This finalized text will not change further, and it likely includes proper punctuation. We can even play a short confirmation sound or haptic when a segment finalizes, to give user feedback.

- **Threading and Performance:** Streaming audio and updating UI in real-time can be performance intensive. Keep the UI thread free by doing minimal work on each transcription message. The heavy audio encoding is done on native side (or separate thread via the recording API). Ensure that state updates for transcript use batched updates (multiple partials per second could overwhelm if not handled efficiently). Using a debouncing mechanism for partial updates might be wise (e.g., update state at most 10 times per second).

- **Stopping Criteria:** If needed, auto-stop recording after a certain silence duration. AssemblyAI might not automatically end the stream on silence (or it might after some timeout). We could implement detection: e.g., if no transcript update has come in 5 seconds, consider that the user is done speaking and stop the recording. This would mimic how some voice assistants work. However, it might cut off if the user is just thinking. We can leave manual control (the user presses stop) for simplicity.

- **Testing the Transcription:** It's important to test with medical terminology (since clinicians might use complex terms). AssemblyAI's accuracy will vary; ensure the chosen model is appropriate (they have a "Universal" model which is general; if there's a specific healthcare model or the ability to boost certain vocabulary via hints, consider that in future).

- **Displaying Live Text Nicely:** The app should scroll the text view as new lines come in (if it exceeds screen). If using a ScrollView or FlatList of transcript segments, ensure it always scrolls to bottom on update. If using a TextInput for editing, be careful to not focus it during live transcription (that would bring up keyboard and distract). It might be better to display in a non-editable Text component while recording, then swap to an editable TextInput after recording stops.

- **Audio Feedback:** Some apps play a short sound when recording starts and stops, to cue the user. We can use Expo's `Audio.Sound` to play a "ding" when they tap record (assuming volume isn't muted). Haptic feedback on iPhone (via `expo-haptics`) is another nice touch for start/stop.

In summary, Rork (with Expo) can handle real-time transcription but might need a custom development build for true streaming. The code architecture likely involves: - A custom hook `useTranscription` that manages the WebSocket and audio, providing `transcript`, `isRecording`, `startRecording()`, `stopRecording()` to the component. - The Dictation screen uses this hook to start/stop and display `transcript` updates. - The hook encapsulates all the logic for connecting to AssemblyAI and parsing messages, so the component code remains clean.

Security Best Practices

Given the sensitive nature of the data (patient information and clinical notes are considered Protected Health Information) and the involvement of API keys, we must enforce strong security practices:

- **Protecting the Cliniko API Key:**
- **Local Storage:** Use a secure storage mechanism on the device. Expo's SecureStore is the preferred choice to store the API key after the user enters it. This ensures the key is encrypted at rest on the device [51]. Do not store this key in plaintext AsyncStorage or in any logs.
- **Memory Handling:** Once the key is in memory (in a context state), treat it as sensitive. Do not log it to console. If using Redux or Zustand devtools, exclude it from being shown in debuggers.
- **Transmission:** The key is used in Authorization headers over HTTPS when calling Cliniko. Always ensure `fetch` calls use `https://` endpoints (which Cliniko's base URLs do) [48]. This encrypts the key in transit. Also, because it's Basic Auth, an attacker intercepting traffic (without HTTPS break-in) still can't read it. But if any error or response might echo something, be mindful.
- **In Supabase (if stored):** If you opt to store the API key in Supabase, do so encrypted. Supabase doesn't have a built-in encryption, but you could encrypt on the client side with a passphrase. Or at least mark the column as `text` and never expose it in any RPC without auth. Use RLS policies so only the row's owner (and maybe service role) can access it. **Never expose the key via a supabase public API or in a JSON the client can read.** The best method is to only use it server-side (edge function) if stored.

- **Key Rotation:** If the user generates a new Cliniko API key, they should update it in the app. Perhaps offer a way to edit the stored key (like returning to the API Key screen, or in settings). When they do so, overwrite and remove the old key completely.

- **Handling PHI (Protected Health Info):** The patient data and notes may be subject to privacy regulations (e.g., HIPAA if applicable).

- We must ensure data is not stored or transmitted to unauthorized services. For instance, when we use AssemblyAI, we are sending patient note content (the dictated speech) to a third party. If this is a concern, AssemblyAI would likely be a HIPAA-compliant service (to be verified). Regardless, we should **not send any explicit patient identifiers in the audio**. The clinician might mention a patient name while dictating – that's inevitable and will be transcribed. That means AssemblyAI's service will handle PHI. Typically, that's allowed under a BAA if AssemblyAI provides one. From the app side, just be transparent about it and use secure channels (which we do).
- After transcription, if we store the note content anywhere (e.g., if we accidentally kept it in Supabase or logs), avoid that. We directly send it to Cliniko and not keep copies.
- Ensure the network calls to Cliniko and AssemblyAI are using TLS (they are).

- Avoid writing sensitive data to disk. For example, if audio is recorded to a file (even temporarily), make sure to delete it after use, or use the OS's temp directory.

- **Supabase Security:**

- Use the **Supabase Auth** session for all user-specific data calls. Supabase's JWT tokens mean every request to Supabase (if we had any for data) is tied to the user. Since we might not use Supabase DB much in this prototype, the main point is to secure the Supabase credentials. The anon key is in the app (which is fine because RLS ensures it can only do what we allow). If we had a service role key, never include that in the app.

- Limit Supabase permissions – if storing keys in a table, use policies to restrict access. Possibly only allow `select` via a secure function (so the key never goes to client).

- **AssemblyAI Key Protection:**

- The AssemblyAI API key, if included in the app, is a secret that could be extracted. While not as sensitive as patient data, misuse of it could incur costs or allow others to transcribe arbitrarily on your account. If concerned, do not embed it directly. Instead, consider one of:
    - Proxy through a server that injects the key (e.g., a supabase function that returns a one-time token or simply relays the audio).
    - Or use an environment variable that is not expo-public (but in a mobile app, any bundled variable is essentially public once someone decompiles it).
    - Monitor usage of the key on AssemblyAI's dashboard to spot anomalies.

- If this were production, you'd likely move transcription to an edge function as well (user's device uploads audio stream to your server, which then forwards to AssemblyAI). This is heavy for real-time though, so you might at least secure the app distribution and limit knowledge of the key. In our documentation context, we can mention that and proceed with using it client-side for simplicity.

- **General Mobile App Security:**

- Ensure the app enforces secure connections (Expo by default will block HTTP calls unless you allow cleartext; we won't).
- Code obfuscation isn't provided by default, but since our logic holds API keys, consider using tools or at least not including keys directly in strings in the code (store them in some encoded form or in native code).

- Turn off any debug logging in production builds, especially anything that might log headers or responses from Cliniko. For instance, if an error occurs, do not log the entire response which might include sensitive info.

- **Logout Flow Cleanup:** As part of best practices, on logout, clear any sensitive data in memory and storage. Supabase will remove its session, but explicitly also remove the Cliniko API key from SecureStore (to prevent someone with device access from retrieving it). Also clear the patients list and any cached notes from memory. Essentially reset the app to initial state.

- **HIPAA considerations:** If aiming for a HIPAA-compliant solution, ensure to sign BAAs with Supabase (if storing PHI there) and AssemblyAI. Also, consider using Cliniko's guidelines for integration: Cliniko likely expects third-party apps to handle data carefully. Their privacy policy notes that when a customer provides an API key to a third-party app, it allows that app to access Cliniko data [52] – we need to uphold their trust by not leaking that data anywhere else.

By adhering to these security practices, we help protect both the application's integrity and the confidential health information it handles. Always keep libraries up to date (for security patches) and follow platform security updates (like iOS requiring NSMicrophoneUsageDescription in app config for microphone permission, which Expo handles in `app.config.js`).

Component and Screen Implementation Templates

This section provides examples of how to define the components and screens for reusability and clarity within the Rork project. We follow a component-driven approach, meaning common UI elements or logic are encapsulated as separate components or hooks. Below are templates and patterns for key components/screens in the Cliniko Voice app:

- **Login Screen Component Structure:** (simplified example in TypeScript/JSX)

```
// File: app/auth/sign-in.tsx
import { useAuthContext } from '@/contexts/AuthContext';
import { SafeAreaView, TextInput, TouchableOpacity, Text, View } from
'react-native';
import { useRouter } from 'expo-router';
import { useState } from 'react';

export default function SignInScreen() {
  const { signInWithEmail, isLoading } = useAuthContext();
  const router = useRouter();
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleSignIn = async () => {
    const result = await signInWithEmail(email, password);
    if(result.success) {
      router.replace('/api-key'); // navigate to API Key screen
```

```
    } else {
      // handle error (e.g., show alert)
    }
  };

  return (
    <SafeAreaView style={{ flex: 1, justifyContent: 'center', padding:
16 }}>
      <Text style={{ fontSize: 24, fontWeight: 'bold', marginBottom: 16 }}
>Sign In</Text>
      <TextInput
        placeholder="Email"
        autoCapitalize="none"
        keyboardType="email-address"
        value={email} onChangeText={setEmail}
        style={{ /* styling */ }}
      />
      <TextInput
        placeholder="Password"
        secureTextEntry
        value={password} onChangeText={setPassword}
        style={{ /* styling */ }}
      />
      <TouchableOpacity onPress={handleSignIn} disabled={isLoading}
style={{ /* button styling */ }}>
        <Text>{ isLoading ? 'Signing in...' : 'Sign In' }</Text>
      </TouchableOpacity>
    </SafeAreaView>
  );
}
```

This pseudocode illustrates usage of context for auth and navigation on success. It omits detailed styles for brevity. In our actual app, we'd also include Magic Link logic and possibly Apple Sign-In as in Rork's example [53] .

- **API Key Input Screen Component:**

```
// File: app/api-key.tsx (assuming we name it api-key)
import { useState } from 'react';
import { View, Text, TextInput, TouchableOpacity, Alert } from 'react-
native';
import * as SecureStore from 'expo-secure-store';
import { useRouter } from 'expo-router';

export default function ApiKeyScreen() {
  const [apiKey, setApiKey] = useState('');
```

```
  const router = useRouter();

  const handleSaveKey = async () => {
    if(!apiKey) {
      Alert.alert("Required", "Please enter your Cliniko API Key.");
      return;
    }
    try {
      // Save to secure storage
      await SecureStore.setItemAsync('clinikoApiKey', apiKey);
      // Optionally, store in some global state or context as well
      // e.g., AppContext.setClinikoApiKey(apiKey);
      // Maybe test the key with a small fetch:
      // const ok = await testClinikoKey(apiKey);
      // if(!ok) { Alert.alert("Invalid Key", "Please check the key and try
again"); return; }
      router.replace('/patients');  // go to patient list
    } catch (err) {
      console.error("Failed to save key", err);
      Alert.alert("Error", "Could not save the API key. Please try
again.");
    }
  };

  return (
    <View style={{ flex: 1, justifyContent: 'center', padding: 16 }}>
      <Text style={{ fontSize: 18, marginBottom: 8 }}>
        Enter your Cliniko API Key
      </Text>
      <TextInput
        placeholder="Cliniko API Key"
        value={apiKey}
        onChangeText={setApiKey}
        autoCapitalize="none"
        autoCorrect={false}
        style={{ /* styling */ }}
      />
      <TouchableOpacity onPress={handleSaveKey} style={{ /* button styling
*/ }}>
        <Text>Save & Continue</Text>
      </TouchableOpacity>
    </View>
  );
}
```

This demonstrates capturing the API key and storing it (with SecureStore). In a real implementation, `testClinikoKey` could be a small fetch to `/users/me` or `/patients` to verify validity.

• **Patient List Screen & Patient List Item:**

```
// File: app/patients.tsx
import { useEffect, useState } from 'react';
import { FlatList, ActivityIndicator, TouchableOpacity, Text, View } from
'react-native';
import { useRouter } from 'expo-router';
import { fetchPatients } from '@/services/clinikoService'; // hypothetical
service function

export default function PatientListScreen() {
  const router = useRouter();
  const [patients, setPatients] = useState<Patient[]>([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const loadPatients = async () => {
      try {
        const data = await fetchPatients();  // fetchPatients would use
stored apiKey internally or get it from SecureStore
        setPatients(data);
      } catch (err) {
        console.error("Failed to load patients", err);
        // handle error: could alert and/or navigate back to key screen
      } finally {
        setLoading(false);
      }
    };
    loadPatients();
  }, []);

  const selectPatient = (patient: Patient) => {
    // Maybe store globally
    // AppContext.setSelectedPatient(patient);
    router.push(`/dictate?patientId=${patient.id}&name=$
{encodeURIComponent(patient.full_name)}`);
  };

  if(loading) {
    return <ActivityIndicator style={{ marginTop: 50 }} />;
  }

  return (
    <FlatList
      data={patients}
      keyExtractor={(item) => item.id.toString()}
      renderItem={({ item }) => (
```

```
        <PatientListItem patient={item} onSelect={() =>
selectPatient(item)} />
      )}
    />
  );
}

// A simple PatientListItem component can be defined in the same file or in
components/PatientListItem.tsx
function PatientListItem({ patient, onSelect }: { patient: Patient,
onSelect: () => void }) {
  return (
    <TouchableOpacity onPress={onSelect} style={{ padding: 16,
borderBottomWidth: 1, borderColor: '#ccc' }}>
      <Text style={{ fontSize: 16 }}>{patient.full_name}</Text>
      {/* could include patient.email or other info in a sub Text for
clarity */}
    </TouchableOpacity>
  );
}
```

Here we assume a `Patient` type with at least `id` and `full_name`. The `selectPatient` function passes patient info to the next screen. We used a query parameter approach (`?patientId=...&name=...`) for demonstration, but as noted earlier, we might handle passing via context instead.

- **Dictation/Note Screen (core parts):**

```
// File: app/dictate.tsx (if not using dynamic route)
import { useState, useRef } from 'react';
import { View, Text, ScrollView, TouchableOpacity } from 'react-native';
import { useRouter, useSearchParams } from 'expo-router';
import { startStreamingTranscription, stopStreamingTranscription } from '@/
services/transcriptionService';

export default function DictationScreen() {
  const router = useRouter();
  const { patientId, name: patientName } = useSearchParams();  // from
route params
  const [transcript, setTranscript] = useState<string>('');    //
accumulated transcript
  const [isRecording, setIsRecording] = useState(false);

  // useRef to hold partial transcript if needed, or hold the streaming
connection
  const streamRef = useRef<any>(null);
```

```
  const handleStart = async () => {
    setIsRecording(true);
    try {
      streamRef.current = await startStreamingTranscription(
        (partial) => {
          // update partial transcript in UI
          setTranscript(prev => prev + partial);
        },
        (final) => {
          // finalize a segment
          setTranscript(prev => prev + final + ' ');
        }
      );
    } catch(err) {
      console.error("Transcription start failed", err);
      setIsRecording(false);
    }
  };
  const handleStop = async () => {
    setIsRecording(false);
    if(streamRef.current) {
      await stopStreamingTranscription(streamRef.current);
    }
    // Now we have final transcript in state
  };
  const handleSubmit = async () => {
    // call Cliniko API to create note
    try {
      await createTreatmentNote(patientId, transcript);  // assume this
uses stored key internally
      router.push('/confirmation');
    } catch(err) {
      alert('Failed to submit note. Please try again.');
    }
  };

  return (
    <View style={{ flex: 1, padding: 16 }}>
      <Text style={{ fontSize: 18, fontWeight: 'bold', marginBottom: 8 }}>
        {patientName ? patientName : 'Selected Patient'}
      </Text>
      <ScrollView style={{ flex: 1, backgroundColor: '#f9f9f9', padding:
10, marginBottom: 8 }}>
        <Text>{transcript || (isRecording ? "Listening..." : "Tap the mic
to start dictating your note.")}</Text>
      </ScrollView>
      <View style={{ flexDirection: 'row', justifyContent: 'space-
```

```
between', alignItems: 'center' }}>
        <TouchableOpacity
          onPress={isRecording ? handleStop : handleStart}
          style={{ /* style for round mic button */ }}>
          <Text style={{ fontSize: 18 }}>{ isRecording ? "■" : " " }</Text>
          {/* Ideally use an icon: mic icon when idle, stop icon when
recording */}
        </TouchableOpacity>
        <TouchableOpacity onPress={handleSubmit} disabled={!transcript}
style={{ /* submit button style */ }}>
          <Text>Submit Note</Text>
        </TouchableOpacity>
      </View>
    </View>
  );
}
```

In this pseudo-code:

- We retrieve `patientName` and `patientId` via router params for display (assuming they were passed).
- We have `startStreamingTranscription` and `stopStreamingTranscription` as abstractions (these would handle WebSocket and audio inside a service or hook, invoking callbacks with partial/final results).
- The UI shows a ScrollView for the transcript (allowing it to scroll if long). We append to `transcript` string directly for simplicity, though in a real app you might maintain it differently to avoid too frequent string concatenations (using arrays of segments maybe).
- Two buttons: one toggles recording (displaying either a mic or a stop symbol), another for submit which is disabled until there's some transcript text.

We omitted error handling details and user permission prompts for brevity. In practice, `startStreamingTranscription` would handle requesting mic permission, establishing the WebSocket, and possibly running on a separate thread if heavy processing is needed.

- **Confirmation Screen Component:**

```
// File: app/confirmation.tsx
import { View, Text, TouchableOpacity } from 'react-native';
import { useRouter } from 'expo-router';

export default function ConfirmationScreen() {
  const router = useRouter();
  // We might fetch some info via route params or context if needed, e.g.,
patient's name or a snippet
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems:
'center', padding: 16 }}>
```

```
      <Text style={{ fontSize: 24, marginBottom: 16 }}> Note Saved!</Text>
      <Text style={{ fontSize: 16, textAlign: 'center', marginBottom: 24 }}
>
        Your treatment note has been submitted to Cliniko.
      </Text>
      <TouchableOpacity onPress={() => router.replace('/patients')}
style={{ /* button style */ }}>
        <Text>Record Another Note</Text>
      </TouchableOpacity>
      <TouchableOpacity onPress={() => router.replace('/')} style={{
marginTop: 16 /* go to home or login if needed */ }}>
        <Text>Done</Text>
      </TouchableOpacity>
    </View>
  );
}
```

This provides a straightforward confirmation with two options: either start another note (go back to patient list) or finish (could go to a home screen or sign the user out, depending on design).

Modularization and Reuse: We have created small, focused components like `PatientListItem` and could do similar for others: - A `MicButton` component could encapsulate the look and behavior of the round microphone toggle (taking `isRecording` as a prop and toggling internally or via props). It could also handle permission prompts internally, but it's often clearer to manage that in the screen logic. - A `TranscriptionViewer` component could be made to handle showing partial vs final text with formatting. For example, it could accept `finalText` and `partialText` as props and render them with different styles. - Even the Confirmation message could be a component if we had variations (success vs error). - All network calls should be in a `services/` folder, not directly in component files (to separate concerns and facilitate testing). We referenced hypothetical functions like `fetchPatients` and `createTreatmentNote` – those would live in something like `services/clinikoService.ts`. Similarly, assembly streaming could be in `services/transcriptionService.ts` or a hook file like `hooks/useTranscription.tsx`.

By organizing code in this way, we make the app easier to maintain. Rork's default project already suggests such structure (contexts, hooks, services directories) [3]. We follow that pattern.

Finally, we integrate everything in the Expo Router. Ensure that `app/_layout.tsx` includes our new screens in the navigation stack (if using explicit routes configuration as in Rork's template) [7]. If we put these screens as top-level files (`api-key.tsx`, `patients.tsx`, `dictate.tsx`, `confirmation.tsx`), Expo Router will automatically have them in the stack navigator (assuming the RootLayout uses a `<Stack>` and doesn't explicitly whitelist screens). In Rork's provided example, they explicitly listed screens like `auth/sign-in`, etc., in RootLayoutNav [54]. We would add:

```
<Stack.Screen name="api-key" options={{ title: 'Cliniko API Key' }} />
<Stack.Screen name="patients" options={{ title: 'Select Patient' }} />
<Stack.Screen name="dictate" options={{ title: 'Dictate Note' }} />
```

```
<Stack.Screen name="confirmation" options={{ title: 'Note Saved', headerShown:
false /* maybe hide header on success screen */ }} />
```

This ensures proper header titles or hiding where needed.

With these components and templates, the lead mobile engineer can quickly grasp how to implement the Cliniko Voice app in Rork. The code can be expanded from here, but the patterns should remain consistent.

Prototype Feature Plan (Incorporated)

To provide full context, below is the original **Cliniko Voice Prototype Feature Plan**, outlining what the app needs to achieve. This serves as a single source of truth for functionality and is now integrated with the technical guidance above:

- **User Authentication:** The app allows clinicians to log in via Supabase authentication. They can use email/password or a magic link to sign in. On successful login (or if a session is remembered), the user enters the app's main flow. If the user is new, they may need to sign up for an account (handled by Supabase). Authentication is required to securely associate data (like API keys) with a user account.

- **Cliniko API Key Input:** After logging in, the clinician is prompted to provide their Cliniko API key. This API key is unique to their Cliniko account and allows the app to access Cliniko's data on their behalf. The app will store this key securely (so the user typically only enters it once). Storing is done either locally (encrypted on device) and/or in the user's profile in Supabase (encrypted/secured). This setup step is critical as it links the app to the clinician's Cliniko data. The UI should guide the user on where to find their API key in Cliniko's settings [52] .

- **Patient Selection:** Once the API key is set, the app retrieves a list of patients from the clinician's Cliniko account via the Cliniko API. The clinician is presented with an alphabetized list (or search interface) of their patients. They can scroll or search to find the patient for whom they want to create a note. Upon selecting a patient, the app will proceed to the note dictation interface. This step ensures the note will be attached to the correct patient record in Cliniko.

- **Voice Dictation with Real-Time Transcription:** The clinician can record a treatment note by speaking instead of typing. When they begin dictation, the app starts capturing the audio and sends it to AssemblyAI's real-time transcription service. As the clinician speaks, the app displays the transcribed text on screen in real time – allowing them to see the note being composed word by word. They can pause or stop when done. The transcription should handle medical terminology reasonably well and format the note (with punctuation and appropriate capitalization) for readability. Clinicians should be able to review and, if necessary, edit the transcribed text before submission (e.g., correct any errors or add details the voice recognition missed). This feature dramatically speeds up documentation by leveraging voice technology, and the real-time feedback lets users ensure the note is being captured correctly.

- **Submit Treatment Note to Cliniko:** After reviewing the transcribed note, the clinician submits it. The app will take the final text and send it to Cliniko via the API, creating a new **treatment note**

entry for the selected patient. This submission uses a predefined template in Cliniko – meaning the note might be inserted into a specific format or category. (For instance, if there's a template called "General Consultation Note", the note will be created under that template ID in Cliniko.) On successful submission, the note is officially saved in the clinician's Cliniko account as part of the patient's record, just as if they had typed it directly into Cliniko. The app should show a confirmation that the note was saved successfully. If Cliniko returns any error (like an invalid template or network issue), the app informs the user to retry or check the entered information.

- **Flow Summary:** In practical use, a clinician would: open the app, log in (if not already), provide the Cliniko API key (if not already saved), pick a patient from their list, tap a button and start speaking their note while seeing it transcribe live, then hit submit to save the note to Cliniko. The app streamlines the entire treatment note creation process into a few taps and voice input, saving time compared to manual typing.

- **Additional Considerations:** The prototype focuses on the core flow described above. Future enhancements could include: ability to add formatting or structured sections to the note (if Cliniko templates have multiple fields), support for multiple practitioners (if a user account has access to multiple Cliniko practitioner profiles), and integration of other AssemblyAI features like summarization or highlighting important phrases. For now, the scope is strictly one note at a time, tied to one patient, with the clinician's own API key and account.

By incorporating this feature plan directly into the technical guide, we ensure that **the development stays aligned with the intended functionality**. Each step in the user journey is backed by technical details in the earlier sections of this document. The lead engineer can cross-reference the plan with implementation sections (e.g., "Voice Dictation" in the plan corresponds to the Dictation Screen and Real-Time Transcription handling in the guide). This alignment makes the document a single source of truth for both what to build and how to build it.

Conclusion

Building the Cliniko Voice prototype with Rork, Supabase, Cliniko API, and AssemblyAI is a cutting-edge project combining mobile app development with AI services. We have reviewed the Rork environment (Expo React Native with file-based routing and integrated backend tools), detailed each screen and its functionality, described the user flow from login to note submission, and outlined the integration points with external services. Real-time transcription is a centerpiece of the app, and we've gone through how to implement and optimize it in a React Native context. Security considerations have been emphasized at each stage to protect sensitive health data and API keys. Finally, the full feature requirements were embedded to keep development on track with the original vision.

Using this guide, a lead mobile engineer should be able to set up the project, implement each component and service, and understand the rationale behind each step. The result will be a **mobile-first iOS app** that significantly streamlines clinicians' workflow by allowing voice-driven note-taking with seamless integration into their Cliniko practice management system. By following the best practices and patterns in this document, the app will be robust, user-friendly, and secure – ready to move from prototype to a potential production tool.

Sources:

- Rork & Expo documentation for project structure and technologies [13] [2]
- Rork example project (Resulta) for integration patterns with Supabase and Expo Router [3] [7]
- Cliniko API documentation and usage examples for patient fetch and note creation [31] [30]
- AssemblyAI real-time transcription overview for handling partial and final transcripts [35]
- Rork FAQ on building voice transcription apps (general capabilities) [1]

---

[1] [51] General - Rork
https://docs.rork.com/faq/general

[2] [3] [13] [49] [50] README.md
https://github.com/jackhunterking/rork-beauty-content-creator/blob/6802793f9b61fabeb5f2f7e75548b837a4139308/README.md

[4] [5] [14] [15] [16] [17] [18] [19] [20] [23] [24] [25] [53] sign-in.tsx
https://github.com/jackhunterking/rork-beauty-content-creator/blob/6802793f9b61fabeb5f2f7e75548b837a4139308/app/auth/sign-in.tsx

[6] [7] [12] [26] [27] [54] _layout.tsx
https://github.com/jackhunterking/rork-beauty-content-creator/blob/6802793f9b61fabeb5f2f7e75548b837a4139308/app/_layout.tsx

[8] [9] AuthContext.tsx
https://github.com/jackhunterking/rork-beauty-content-creator/blob/6802793f9b61fabeb5f2f7e75548b837a4139308/contexts/AuthContext.tsx

[10] [11] package.json
https://github.com/jackhunterking/rork-beauty-content-creator/blob/6802793f9b61fabeb5f2f7e75548b837a4139308/package.json

[21] [22] [40] [41] [42] [43] [44] [47] useAuth.ts
https://github.com/jackhunterking/rork-beauty-content-creator/blob/6802793f9b61fabeb5f2f7e75548b837a4139308/hooks/useAuth.ts

[28] [46] supabase.ts
https://github.com/jackhunterking/rork-beauty-content-creator/blob/6802793f9b61fabeb5f2f7e75548b837a4139308/lib/supabase.ts

[29] Setting Up Cliniko Integration | Complete Guide - Patient Notes
https://www.patientnotes.app/documentation/setting-up-cliniko-integration

[30] [31] [32] [33] [39] cliniko-rest-examples.md
https://github.com/cryptoleek-eth/cliniko-api-client/blob/1c892b26efa4597e9e33edb40ad779dae7f87eaa/cliniko-rest-examples.md

[34] [35] Speech-to-Text API - AssemblyAI
https://www.assemblyai.com/products/speech-to-text

[36] Best Practices for building Voice Agents | AssemblyAI | Documentation
https://assemblyai.com/docs/voice-agent-best-practices

37 38 Create treatment note template - Cliniko API
https://docs.api.cliniko.com/openapi/treatment-note-template/createtreatmentnotetemplate-post

45 Transcribe phone calls in real-time using C# .NET with AssemblyAI ...
https://www.twilio.com/en-us/blog/developers/tutorials/integrations/transcribe-phone-calls-real-time-csharp-assemblyai-twilio

48 Create treatment note
https://docs.api.cliniko.com/openapi/treatment-note/createtreatmentnote-post

52 cliniko.com.md
https://github.com/citp/privacy-policy-historical/blob/4002d8c5640023da70a24a63af63e4c17b74f6cd/c/cl/cli/cliniko.com.md