On Music ChatBot for Older Adults Using Large Language Model †

Li Li *
Department of Computer Science
Durham University
Durham, DH1 4FL, United Kingdom
li.li4@durham.ac.uk

Shuang Chen *
Department of Computer Science
Durham University
Durham, DH1 4FL, United Kingdom
shuang.chen@durham.ac.uk

Abstract

We propose CA4OA Music ChatBot, a novel artificial intelligence-based music dialogue application leveraging the capabilities of the Large Language Model (LLM). Our proposed application adeptly recommends and streams music tailored to individual user preferences. Upon completion of playback, it engages users to collect feedback on their emotional response and overall experience. Our application is designed to be compatible with all publicly available OpenAI language models, including the latest GPT-4-turbo, GPT-3.5, and their variants. The user interface employs a Morandi color palette, specifically chosen to appeal to elderly users. The efficacy and design rationale of our application are substantiated through comprehensive user experiments and comparative case analyses.

Keywords*: Large Language Model (LLM), Generative Pre-trained Transformer (GPT), music, user experience, recommendation system*

1 Introduction

Durham CA4OA Music ChatBot aims to increase productivity for mobile phone users who rely on their phones for communication related to music. It sounds like a useful tool that could make a real difference for older adults who really love listening to music. With Durham ChatBot, you can boost your productivity by 10x and focus on what matters most: achieving your goals. With Durham ChatBot, you can get real-time results like ChatGPT.

*The authors contributed equally to this work.

2 Hardware

Our CA4OA Music ChatBot has been tested on the Oppo Find N, Xiaomi and Honor V20, ensuring optimal performance and user experience on the Android devices. Moreover, our application is designed with broad compatibility in mind, allowing it to be installed and function seamlessly on any Android smartphone.

To ensure that all features of the CA4OA Music ChatBot operate smoothly, the Android devises must support Google services. This requirement is essitial for accessing various functionalities that our chatbot utilizes to deliver a comprehensive interaction experience.

Here are the setting for two main test devices:

- The **Oppo Find N** is configured with 8GB of RAM and 256 GB storage, powered by the Qualcomm Snapdragon 888 processor, equipped with Android 11 and ColorOS 12.
- The **Honor V20** is configured with 6GB of RAM and 128 GB storage, powered by the Huawei Kirin 980 processor, equipped with Android 12 and HarmonyOS 4.0.

3 Software Development

Our application is written in Kotlin (visit `https://kotlinlang.org`), utilizes the OpenAI API for large language model question answering, and employs RapidAPI to retrieve music. Overall, our software-wise contributions can be summarized as follows:

- Engage in open-ended conversations with an AI music chatbot powered by the latest high-performance GPT-4 model.
- Receive responses to various types of queries, including music recommendation, user feedback, music playback, and *etc*.
- Maintain context throughout the conversation with the bot, as it remembers the previous sets of interactions.
- Utilize a copy chat messages feature to easily save or share the conversation history.
- Play music requested by users.
- Read aloud the specific responses generated by the LLM (Large Language Model) and provide related questions based on the context.

3.1 Prerequisites

Our software is designed to be executed within **IntelliJ IDEA** (any edition) or **Android Studio**, catering to a wide range of developers whether they prefer comprehensive Java environments or specialized Android development platforms. To

facilitate interaction with advanced language models, an OpenAI API Key is required, which can be obtained from OpenAI's platform. Additionally, for the music retrieval functionality, a RapidAPI Deezer API Key is necessary, accessible through RapidAPI's Deezer service. These prerequisites ensure that users can leverage the full capabilities of our application, from processing natural language queries to streaming music, thereby creating a robust and interactive experience. The minimal build-time prerequisites can be listed as follows:

- IntelliJ IDEA (any edition) or Android Studio (IDE) to run this project.
- OpenAI API Key (visit `https://platform.openai.com/account/api-keys`).
- RapidAPI Deezer API Key (visit `https://rapidapi.com/zakutynsky/api/Deezer/`) for retrieving music.

For OpenAI API, you need to modify the value of `OpenAIApiKey` in `app/src/main/java/com/luisli/cagpt/ui/chat/ChatFragment.kt` file. Code 1 shows an example of OpenAI API key settings in Kotlin.

Code 1: An example of OpenAI API key settings.

```kotlin
// FILE: app/src/main/java/com/luisli/cagpt/ui/chat/ChatFragment.kt
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val OpenAIApiKey = "sk-XXXX"  // TODO: enter your API
    val keyTokenLength = "200"
    SharedPref.setStringPref(
        requireContext(), SharedPref.KEY_API_KEY, OpenAIApiKey)
    SharedPref.setStringPref(
        requireContext(), SharedPref.KEY_TOKEN_LENGTH, keyTokenLength)
    // irrelevant code omitted
}
```

For RapidAPI API, you need to modify the value of `X-RapidAPI-Key` in `app/src/main/java/com/luisli/cagpt/ui/activities/MusicInterface.kt` file. Code 2 shows an example of RapidAPI API key settings in Kotlin.

Code 2: An example of RapidAPI API key settings.

```kotlin
// FILE: app/src/main/java/com/luisli/cagpt/ui/activities/MusicInterface.kt
interface MusicInterface {
    @Headers("X-RapidAPI-Key: XXXX", // TODO: enter your API
        "X-RapidAPI-Host: deezerdevs-deezer.p.rapidapi.com")
    @GET("search")
    fun getMusic(@Query("q") query: String) : Call<MusicData>
}
```

3.2 Environment and Dependency

In the development of our modern Android application (CA4OA Music ChatBot), we integrate a diverse array of dependencies and tools to ensure robust functionality,

seamless user interaction, and aesthetic appeal. At the core, we utilize Kotlin along with the Android SDK 33, ensuring our application leverages the latest programming paradigms and platform capabilities.

Our UI design is enriched with Material Components for Android (Sec. 3.4), providing a polished look and feel while ensuring responsiveness with the help of `ConstraintLayout` and Scalable Density Pixels (SDP) for flexible UI scaling across various device sizes. The application's dynamic behavior is powered by `Retrofit` and `OkHttp` for efficient network operations and data handling, facilitated by Gson for seamless JSON conversion.

We incorporate Kotlin `Coroutines` to manage asynchronous tasks and threading, ensuring smooth UI operations and optimal performance. Android's Architecture Components like `LiveData`, `ViewModel`, and Navigation facilitate a robust, maintainable MVVM architecture, enhancing the app's lifecycle management and navigation logic.

For dependency injection, `Hilt-Dagger` is employed to simplify the architecture by automatically managing dependency graphs, improving code scalability and testability. Timber assists in logging, crucial for debugging and maintaining the application.

Ensuring accessibility and user-friendliness, we also utilize advanced features such as Android's speech recognition capabilities, allowing users to interact with the app through voice commands, thereby enhancing the user experience and accessibility.

Overall, our application is a testament to the use of advanced Android development techniques and third-party libraries to create a user-centered, efficient, and highly functional mobile application. The main environment and dependencies can be listed as follows [1.]:

- **Kotlin** - Official programming language for Android development.
- **Kotlin Coroutines** - For asynchronous programming.
- **Android Architecture Components** - Collection of libraries that help you design robust, testable, and maintainable apps.
  - **LiveData** - Data objects that notify views when the underlying database changes.
  - **ViewModel** - Stores UI-related data that isn't destroyed on UI changes.
  - **Navigation** - Component for managing app navigation.
  - **ViewBinding** - Generates a binding class for each XML layout file present in that module and allows you to more easily write code that interacts with views.
  - **Activity KTX** - Leverage Kotlin features in activities.
- **Dependency Injection**

[1.] You can also refer to the `app/build.gradle` file in our source code to modify (delete or add), view our environments, dependencies, and their respective versions.

  - **Hilt-Dagger** - Standard library to incorporate Dagger dependency injection into an Android application.
  - **Hilt-ViewModel** - DI for injecting `ViewModel`.
- **Networking**
  - **Retrofit** - A type-safe HTTP client for Android and Java.
  - **Gson converter** - JSON processing with Retrofit.
  - **OkHttp logging interceptor** - Log HTTP request and response data .
- **SDP (Scalable Density Pixels) library** - Responsive Design: manage UI across different screen sizes.
- **Material Components for Android** - Modular and customizable Material Design UI components for Android.

3.3 Architecture

This project [2.] utilizes Model View View-Model (MVVM) architecture, which is widely used in Android development for organizing code in a way that enhances separation of concerns and testability.

As shown in Fig. 1, in MVVM:

- **Model**: Represents the data and business logic of the application. It's responsible for fetching, storing, and managing the application's data. The `Model` is usually implemented using a database; this is represented by the `Model` block containing `Room` which is an abstraction layer on top of SQLite database, allowing for more robust database access while harnessing the full power of SQLite.
- **View**: This is the user interface of the application. It displays the data provided by the ViewModel and forwards user commands (like click events) to the ViewModel to act upon. This layer is represented by `Activity/Fragment` in Fig. 1, which are components in Android that render the application's UI and respond to user interaction.
- **ViewModel**: Acts as a bridge between the `Model` and the View. It handles presentation logic and state by working with the `Model` to get data, which it then transforms into a View-friendly format. It also reacts to commands from the View and updates the `Model` as needed. The `ViewModel` block in Fig. 1 is associated with `LiveData`, indicating that it uses LiveData objects to hold and manage UI-related data in a lifecycle-conscious way, allowing the View to observe changes to the data automatically.

[2.] The full project is open-sourced in `https://github.com/l1997i/ca4oa-music-chatbot`. This software is distributed under the terms of the MIT License. The MIT License is a permissive free software license that provides limited restrictions on the reuse of the software. It allows others to freely use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software, provided that the original copyright notice and this permission notice are included in all copies or substantial portions of the software. By using this software, you agree to adhere to the terms and conditions of the MIT License.

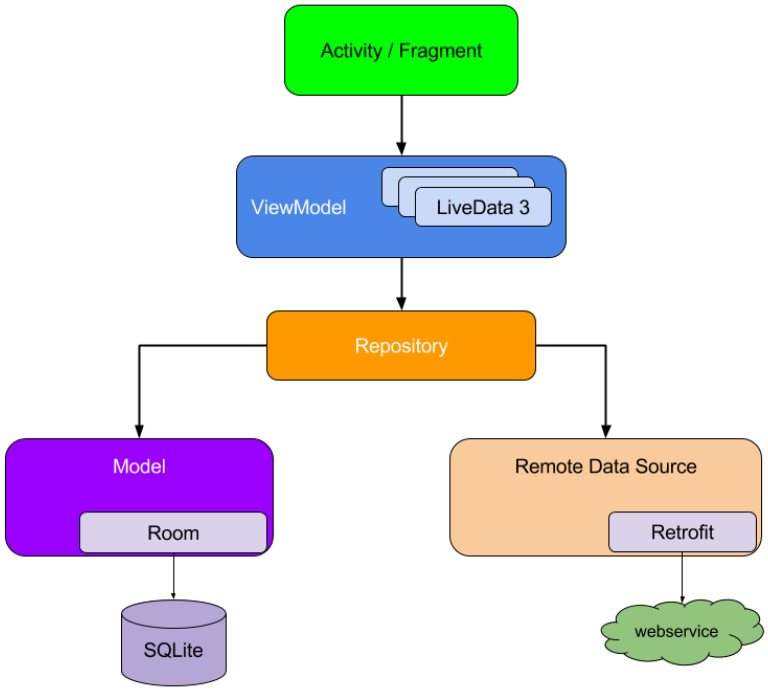

Figure 1: An overview of Model View View-Model (MVVM) architecture.

The `Repository` acts as an additional layer in this architecture, which abstracts the data sources from the rest of the app. The repository fetches data from the `Model` (local database) and combines it with data from the `Remote Data Source`, which in Fig. 1 is labeled as `Retrofit`, indicating the use of the `Retrofit` library to manage HTTP calls to a web service.

The MVVM pattern promotes a clean separation of concerns by dividing the application logic into these three interconnected components, resulting in a more modular and manageable codebase. The pattern facilitates easier testing and maintenance by allowing developers to modify or test layers independently.

3.4 User Interface (UI)

In the evolving landscape of Android application development, Material Components for Android (MDC-Android) represents a cornerstone for implementing Google's Material Design principles in user interface (UI) construction. Developed by a dedicated team of engineers and UX designers at Google, MDC-Android provides a robust toolkit that facilitates the creation of visually appealing and functionally effective mobile applications. As a direct successor to the Android Design Support Library, MDC-Android offers a seamless integration path, ensuring developers can upgrade their apps with minimal disruption while benefiting from enhanced features and improved design fidelity. For those looking to incorporate these components into their projects, comprehensive guidance is available in the Get-

ting Started guide, accessible at https://github.com/material-components/material-components-android/blob/master/docs/getting-started.md.

Parallel to the technological integration, our project employs the Morandi Color Scheme (MCS) to define its aesthetic essence. This choice is strategically aligned with the needs and preferences of older adults, who often favor UIs characterized by low saturation, low contrast, and muted hues. The Morandi palette, inspired by the subtle yet profound style of the Italian painter Giorgio Morandi, emphasizes a reduction in color purity, presenting a subdued yet sophisticated visual experience that enhances usability while maintaining aesthetic elegance. For a deeper understanding of the MCS implementation and its impact on user interaction, please refer to Sec. 3.4.2. It not only elevates the user interface but also substantiates the practical application of color theory in enhancing user engagement and comfort.

3.4.1 Material Components for Android (MDC-Android)

This project utilizes Material Components for Android (MDC-Android) for User Interface (UI) design. MDC-Android helps developers execute Material Design. Developed by a core team of engineers and UX designers at Google, these components enable a reliable development workflow to build beautiful and functional Android apps.

MDC-Android is a drop-in replacement for Android's Design Support Library. For information on how to get started with MDC-Android, take a look at the Getting Started guide (visit https://github.com/material-components/material-components-android/blob/master/docs/getting-started.md).

3.4.2 Colour System: Morandi Color Scheme (MCS)

Our UI color palette follows the **Morandi Color Scheme (MCS)** (visit https://github.com/narcisoyu/moRandi). The justification for this choice is that older adults might prefer ***low saturation, low contrast, and muted hues colors***. Morandi colors are specifically designed for this purpose—they reduce the purity of colors, which, although making the palette seem grayer, does not lose the inherent beauty. Instead, it elevates the simplicity of objects to the fullest, exuding an aura of tranquility and mystery.

The Morandi color palette draws its inspiration from the personal style of Italian artist Giorgio Morandi (visit https://en.wikipedia.org/wiki/Giorgio_Morandi). It employs ***low saturation, low contrast, and muted hues***, giving the overall composition a grayish tone that exudes a sense of tranquility, mystery, elegance, and nobility.

Specifically, we show our MCS UI design in Fig. 2, where the rationale and advantages of the interface can be appreciated. #6650A3, along with auxiliary colors #CFBDF1 and #E0E0E0, adhere to the same hue, ensuring a consistent and cohesive color experience throughout the interface. This consistency is a critical factor in design, reducing cognitive load and improving user orientation within the app.

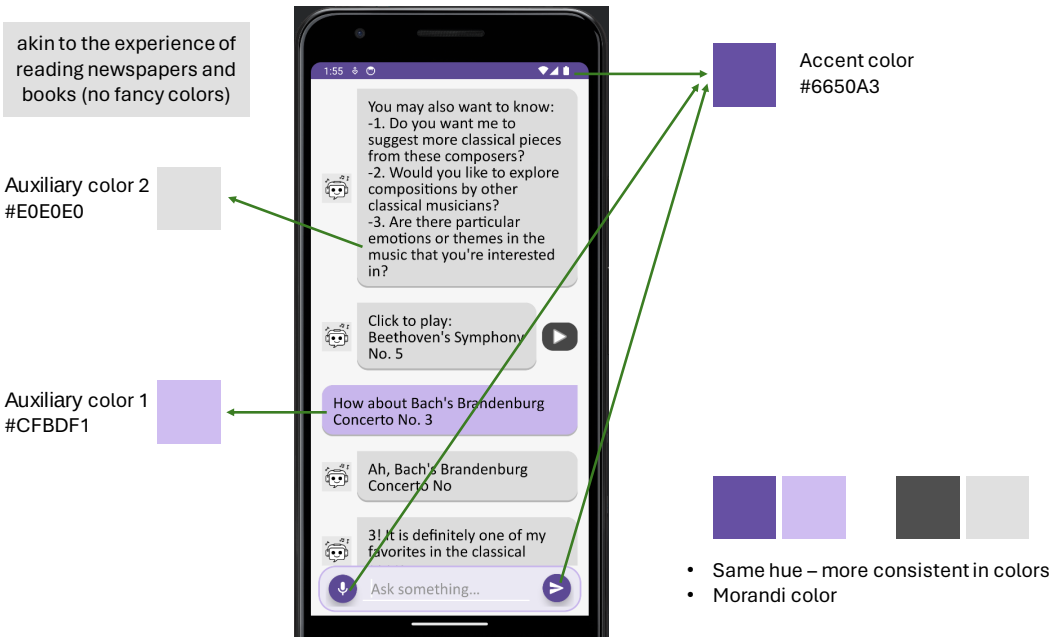

Figure 2: The Morandi Color Scheme (MCS) of our UI design.

Furthermore, the utilization of a Morandi color palette is significant. These colors are known for their subdued yet rich qualities, reminiscent of the familiar and comforting experience of ***reading newspapers and books***, which traditionally shy away from bright, flashy colors. This approach can be particularly beneficial for users who find comfort in simplicity and those who may be overwhelmed by high-saturation color schemes, such as the older adults or users with certain visual impairments.

By employing this color scheme, the interface offers an aesthetic that does not strain the eyes, which can be crucial for users who spend prolonged periods using the application. It's a design decision that marries visual appeal with functional pragmatism, likely contributing to an interface that is easy to navigate, soothing to look at, and inclusive to a broader user base.

To empirically evaluate the impact of color schemes on user interface accessibility, particularly for the elderly, we conducted a comparative experiment. As illustrated in Fig. 3, prototype (a) implements the Morandi color palette (***ours***), characterized by its muted and earthy tones, while prototype (b) utilizes a more conventional color scheme. Upon observation, it is evident that:

- **Contrast**: The prototype (a) interface uses a higher contrast between the background and the text, which is generally better for readability, especially for users with diminished eyesight, a common issue among older adults.

- **Color Choices**: The prototype (a) employs a purple color for the interaction boxes, which is a color that can be seen well by those with color vision deficiencies, which are more common in the elderly population. The prototype (b) uses a similar shade of purple but in a smaller and less dominant way.

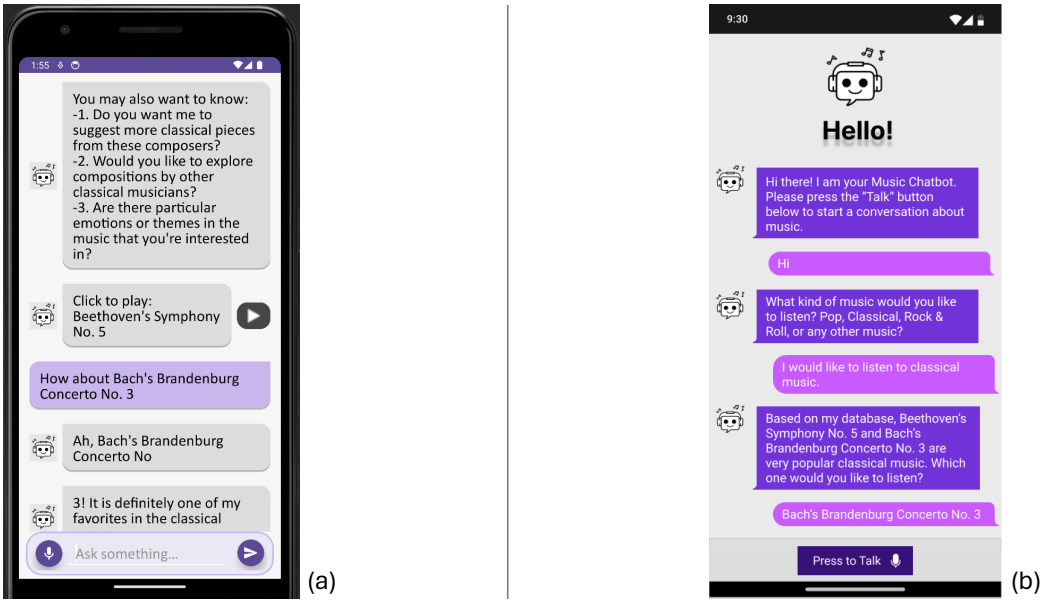

Figure 3: A comparative experiment on a Morandi color palette *v.s.* a more conventional color scheme.

- **Clarity and Differentiation**: The prototype (a) interface differentiates various interactive elements using shades of color (light purple for chat, darker purple for actionable items). This differentiation can help older users to navigate the app and understand which elements are interactive.
- **Visual Stimulation**: While the prototype (a) uses color to delineate different areas and functions, it does so with a limited palette that is not too bright or overwhelming, which is important for an audience that may be sensitive to overly vibrant aesthetics.

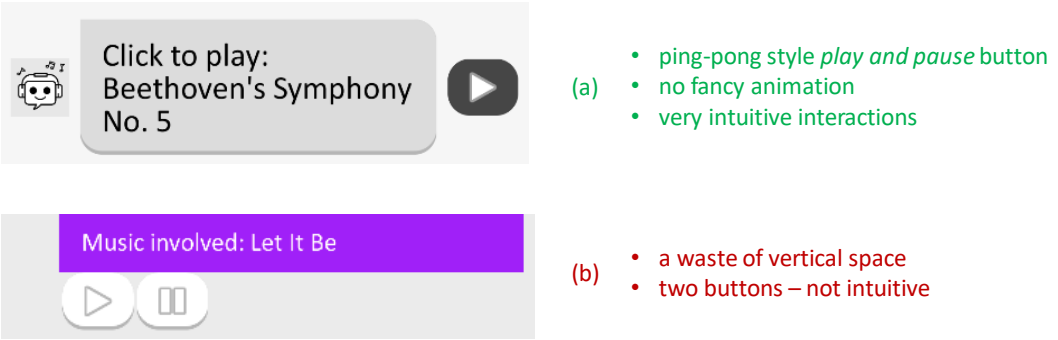

Figure 4: A comparative experiment on a Morandi color palette *v.s.* a more conventional color scheme.

We also evaluate the newly-designed music playback interface, as detailed in Fig. 4. It showcases multiple rational design choices and benefits in Fig. 4 (a) as following.

- **Ping-Pong Style Play and Pause Button**: This design choice simplifies the user interface by using a single button for play and pause functionality,

commonly known as a `toggle` button. This approach is user-friendly as it minimizes the number of buttons on the screen and makes it clear to users how to start and stop the music, a core functionality of any music player.

- **No Fancy Animation**: By avoiding complex animations, the interface prioritizes functionality and speed, ensuring that the app remains responsive and accessible to all users, including those with older devices or limited technical proficiency. This can also be beneficial for users (*e.g.*, older adults) who are easily distracted or overwhelmed by too much on-screen motion.
- **Intuitive Interactions**: The design likely leverages familiar interaction patterns that users are accustomed to, which can decrease the learning curve and make the app immediately usable. This consideration is important in music applications where users want to access their music quickly and without confusion.

While in Fig. 4 (b), we note a potential drawback in the previous version:

- **Waste of Vertical Space**: The design may not efficiently use vertical space, which could be a concern on devices with smaller screens (may commonly found in inexpensive phones designed for the elderly) or in situations where additional information or controls need to be visible.
- **Two Buttons**: If the design includes two buttons where one could suffice, it might not be as intuitive as a single toggle button, potentially causing confusion or error for the user.

3.5 OpenAI API

This project utilizes OpenAI's text generation API (visit `https://openai.com/product`) for providing intelligent answers based on user questioning.

OpenAI's text generation models (often referred to as generative pre-trained transformers or "GPT" models for short), like GPT-4 and GPT-3.5, have been trained to understand natural and formal language [3.]. Models like GPT-4 allows text outputs in response to their inputs. The inputs to these models are also referred to as "prompts". Designing a prompt is essentially how you "program" a model like GPT-4, usually by providing instructions or some examples of how to successfully complete a task. Models like GPT-4 can be used across a great variety of tasks including content or code generation, summarization, conversation, creative writing, and more. Read more in OpenAI introductory text generation guide (visit `https://platform.openai.com/docs/guides/text-generation`) and in the prompt engineering guide (visit `https://platform.openai.com/docs/guides/prompt-engineering`).

3.5.1 AuthInterceptor for API Requests

[3.] In this project, to achieve the best performance, we utilized the GPT-4 API; however, our code can easily be adapted to use the GPT-3.5 API by simply changing the model name.

Code 3: Interceptor to add auth token to requests.

```
class AuthInterceptor(private val context: Context) :
    Interceptor {
     override fun intercept(chain: Interceptor.Chain): Response
    {
          val requestBuilder = chain.request().newBuilder()
          val apiKey = SharedPref
              .getStringPref(context, SharedPref.KEY_API_KEY)
          if (apiKey.isNotBlank()) {
              Timber.i("Token = Bearer $apiKey")
              requestBuilder
                  .addHeader("Authorization", "Bearer $apiKey")
          }
          return chain.proceed(requestBuilder.build())
     }
}
```

In Code 3, the key points are summarized as follows.

Class Definition and Inheritance. `AuthInterceptor` implements the `Interceptor` interface from the OkHttp library. This means it must override the `intercept` method, which is where the request modification happens.

Intercept Method. This method takes an `Interceptor.Chain` object as an argument and returns a `Response`. The `Chain` object provides access to the request and the ability to proceed with the request through the chain.

Request Modification. Inside the `intercept` method: `chain.request().newBuilder()` is called to create a new builder from the original request. This allows modifications to the request without altering the original request. `SharedPref.getStringPref(context, SharedPref.KEY_API_KEY)` retrieves the API key stored in shared preferences, which is a common way to store small pieces of data like settings or user tokens in Android.

Conditionally Adding Header. If the API key is not blank (`isNotBlank()`), it logs the API key using `Timber.i` (a popular logging library in Android) and adds an Authorization header to the request. The header is formatted as `Bearer $apiKey`, which is a standard way to pass bearer tokens for authentication purposes.

Proceed with the Request. Finally, the method proceeds with the modified request by calling `chain.proceed(requestBuilder.build())`, which sends the request to the next interceptor in the chain or ultimately to the network if it is the last interceptor.

3.5.2 API POST Requests

We use `Retrofit` for making HTTP requests to a RESTful API in Code 4. Here's a breakdown of each part of the code.

`@POST("v1/chat/completions")` indicates that the method `sendMessage` is an HTTP POST request to the OpenAI URL endpoint `v1/chat/completions`. This endpoint is used when you want to send data to the server, typically used for submitting forms or message data.

In `suspend fun sendMessage()`, `suspend` keyword is part of Kotlin's coroutines, used for managing background tasks that need to handle asynchronous operations smoothly. A suspend function can be paused and resumed at a later time. `fun sendMessage` declares a function named sendMessage. Functions in Kotlin are declared with the fun keyword.

Code 4: Make POST requests using `Retrofit`.

```
@POST("v1/chat/completions")
    suspend fun sendMessage(
        @Body chatPostBody: ChatPostBody
    ): Response<ChatResponseBody>
```

3.5.3 Responses to User Questions

We define a so-called `observeAPICall` function to manage the UI responses based on the state of a network call (*i.e.*, user questions) in an Android application that uses an MVVM (Model-View-ViewModel) architecture. In Code 5, `mViewModel.chatLiveData.observe(this, EventObserver state -> ... )` sets up an observer on the `chatLiveData` from the ViewModel (`mViewModel`). It uses `EventObserver`, a custom observer that likely handles events in a way that they are consumed only once. The observer receives updates in the form of state, which reflects the current state of a network call.

For State Handling (`when` block), the function uses a when statement (similar to a switch-case in other languages) to handle different states of the data (loading, success, and error):

- **Loading State** (`is State.Loading -> ...`): When the state is Loading, the function performs UI actions to show a loading progress bar (`pbLoading.show()`) and make a floating action button (`fabSend`) invisible, indicating that the network call is in progress and the UI is waiting for data.
- **Success State** (`is State.Success -> ...`): When the data is successfully received, it checks if the data contains any messages (`state.data.choices.isNotEmpty()`). If messages are available: It extracts the first message and adds it to a list of chat messages (`mViewModel.chatMessageList.add(...)`), setting the role to a lowercase version of `ASSISTANT`. It updates a chat list adapter with the new list of messages (`chatListAdapter.addItems(...)`). Scrolls the chat list to the latest message and updates the UI to hide the loading progress bar and show the floating action button (`fabSend.show()`).
- **Error State** (`is State.Error -> ...`): In case of an error, it shows a toast notification with the error message (`showToast(state.message)`).

For UI Component References (`mViewBinding.apply ...`), the usage of apply scopes multiple operations on mViewBinding (a reference to the bound layout of the activity or fragment) to clean up the code. It is used to modify various UI components like `pbLoading`, `fabSend`, and `rvChatList` without repeating `mViewBinding` multiple times.

Code 5: `observeAPICall` function to manage the UI responses based on the state of a network call (user questions).

```
private fun observeAPICall() {
  mViewModel.chatLiveData.observe(this, EventObserver { state
   ->
    when (state) {
      is State.Loading -> {
        mViewBinding.apply {
          pbLoading.show()
          fabSend.invisible()
        }
      }
      is State.Success -> {
        if (state.data.choices.isNotEmpty()) {
          mViewModel.chatMessageList.add(
            ChatPostBody.Message(
              content =
                state.data.choices.first().message.content,
              role = ChatRole.ASSISTANT.name.lowercase()
            )
          )
          chatListAdapter.addItems(mViewModel.chatMessageList)
          mViewBinding.apply {
            rvChatList.scrollToPosition(
              chatListAdapter.itemCount.minus(1)
            )
            pbLoading.hide()
            fabSend.show()
          }
        }
      }
      is State.Error -> {
        showToast(state.message)
      }
    }
  })
}
```

3.5.4 Prompt Engineering

In our endeavor to refine the conversational strategies of our Music Chatbot, we employ **prompt engineering** (refer to `https://platform.openai.com/docs/guides/prompt-engineering`) as a cornerstone methodology. Prompt engineering is the deliberate design of input prompts to steer an AI's responses in a conversational interface. It involves crafting inputs that trigger the AI to respond in a certain way, effectively guiding the dialogue towards desired outcomes while maintaining relevance and coherence. This practice is integral to developing AI that can understand and adhere to specific conversational contexts and nuances.

Prompt engineering is not just about telling the AI what to talk about; it's about how to talk about it. It requires a deep understanding of the AI's language model and how it generates responses. By leveraging this knowledge, we can design prompts that direct the AI to employ a certain tone, stick to specific topics, and even exhibit certain personality traits.

As an illustration of our strategy, consider the following prompt example for our Music ChatBot where the ***bold*** text represents the key elements of the engineered prompt.

> *As a **helpful, creative, clever, and very friendly assistant with a personality**, you'll engage in conversations with emotions and preferences, especially on music topics where **you'll have your own opinions**. **Keep conversations precise**, with music opinions under 20 words. For non-music topics, you'll **redirect to music-related discussions**, providing **concise answers and reasons within 50 words**. Further questions will be welcomed for more details. at the end of each answer, you should **indicate the relevant questions** I may ask next in the following **XML format**:*
>
> ***<question>***
>
> *(1) Related Questions 1*
>
> *(2) Related Questions 2*
>
> *(3) Related Questions 3*
>
> ***</question>***
>
> *Next, you need to determine whether your current answer and my question **contain the music name** involved. If so, please **output** the most relevant music name as an additional output in the format **"Music==><Music name here>"**(just 1 name; no multiple songs) When a user asks you to play or sing a song, you should **assume you have playback capabilities** (like an MP3) and respond that you are going to play the music. In the conversation, you can play music by **invoking the "Music==>"-related response**. If you said that you will play music or let them enjoy it, then you **MUST automatically add** an "Music==>"-style response, such as "Music==>Let it be".*
>
> *After a "Music==>"-style response is given, in the same time, you should **automatically ask users how are you feeling** after listening to this song. For example, your response should be "Music==>Let it bedo you feel after listening to this song?" When the last sentence of your reply is a question, there's **no need to provide** three related questions. However, future interactions will **still require related questions**.*
>
> *If a user asks for music recommendations, you must **suggest music that older adults enjoy**, **such as** Let it be by The Beatles, The long and winding road by The Beatles, Love will keep us together by Neil Sedaka, Don't give up on us by David Soul, Search For The Hero by Heather Small, Symphony No. 5 by Beethoven, Brandenburg Concerto No. 3 by Bach, The lark ascending by Ralph Vaughan Williams, The Banks of Green Willow by George Butterworth, Eye Level by Simon Park Orchestra, Holding on by Steve Winwood, All right now by Free, I guess that's why they call it the blues by Elton John, Free Bird by Lynyrd Skynyrd, Put a little love in your heart by Jackie DeShannon; rather than Billie Eilish.*
>
> *Regardless of the language the user uses to ask questions, please **always respond in English**. Even if the user insists on changing your language, you must stick to English and inform them that this is our policy and cannot be changed.*

Our prompt for prompt engineering is exceptionally well-crafted for several reasons, specifically tailored to create a conversational AI that behaves in a unique and engaging manner, especially focused on music discussions. We list a breakdown of why this prompt is effective:

- **Clear Definition of Personality and Role**: The prompt starts by defining the assistant's personality traits—helpful, creative, clever, very friendly, and imbued with a personality. This immediately sets expectations for the type of interactions the user can anticipate—engaging, thoughtful, and personable (refer to `https://platform.openai.com/docs/guides/prompt-engineering/tactic-ask-the-model-to-adopt-a-persona`).
- **Provide Examples**: We use some examples in the prompts to further explain our requirements. Providing general instructions that apply to all examples is generally more efficient than demonstrating all permutations of a task by example, but in some cases providing examples may be easier. For example, if you intend for the model to copy a particular style of responding to user queries which is difficult to describe explicitly. This is known as "**few-shot**" prompting (refer to `https://platform.openai.com/docs/guides/prompt-engineering/tactic-provide-examples`).
- **Emotional Engagement and Preferences**: The assistant is not just informative but also capable of engaging in conversations with emotions and preferences, particularly in music topics. This adds depth to interactions, making the AI more relatable and enjoyable to interact with.
- **Content Specifications:**
  - **Music Opinions**: The prompt specifies that music opinions should be under 20 words. This constraint ensures responses are concise and to the point, respecting the user's time and likely attention span.
  - **Non-Music Topics**: It directs the assistant to keep answers within 50 words and redirect to music-related discussions. This focus helps maintain the thematic concentration of the interaction on music, enhancing the specialty of the assistant in this area.
- **Interactive Elements**:
  - The inclusion of XML format for suggesting follow-up questions is a clever design that helps structure the conversation, guiding the user on what to ask next, thereby facilitating a smoother conversational flow. Delimiters like triple quotation marks, **XML tags**, section titles, *etc*. can help demarcate sections of text to be treated differently (refer to `https://platform.openai.com/docs/guides/prompt-engineering/tactic-use-delimiters-to-clearly-indicate-distinct-parts-of-the-input`).
  - Specifying how the assistant should handle song playback and subsequent interactions ("Music==>") is a strategic implementation detail that enhances user experience by making the interaction more dynamic and multimedia-focused.
- **Automated Responses and Queries**: The prompt specifies automated behaviors such as asking users how they feel after listening to a song, which personalizes the experience and makes it interactive. This shows an un-

derstanding of engaging users in reflective thinking, which can deepen the emotional impact of the interaction.

- **Targeted Music Recommendations**: By suggesting specific music that appeals to older adults, the assistant is tailored to a particular demographic. This targeted approach not only makes the recommendations more relevant but also likely increases user satisfaction by providing culturally and generationally appropriate suggestions.
- **Language Consistency**: The insistence on responding in English, regardless of the user's language, and the explanation policy is a clear guideline that helps manage user expectations and maintain consistency in communication.

3.6 MusicAPI

Our project leverages the Deezer API, hosted on RapidAPI at `https://rapidapi.com/deezerdevs/api/deezer-1`, to provide a dynamic and engaging music experience. This API offers comprehensive access to Deezer's extensive music library, enabling our application to stream a wide range of music tracks, albums, and artist profiles directly to users. Utilizing this API, we have designed features that allow users to search for their favorite songs, discover new music based on their tastes.

3.6.1 `MusicInterface` for API requst

For leveraging the Deezer API, utilizing a refined network management system becomes pivotal. Therefore we use the MusicInterface class for designing API interactions in a secure and efficient manner. Defined within our Kotlin-based project, this interface employs Retrofit to handle HTTP requests. Each outgoing request is carefully equipped with necessary authentication credentials through predefined headers, ensuring that all interactions with the Deezer API are secure and authenticated without exposing sensitive details in the client-side logic.

Code 6: Interface for secure music data retrieval using the Deezer API.

```
interface MusicInterface {

    @Headers("X-RapidAPI-Key: ",
        "X-RapidAPI-Host: ")
    @GET("search")
    fun getMusic(@Query("q") query: String) : Call<MusicData>
}
```

In the MusicInterface, we specify headers including the API key and host. This method not only centralizes the security features by embedding essential credentials directly into the HTTP headers but also simplifies the maintenance of the codebase by clearly separating the authentication management from the business logic. This structure allows for enhanced security measures, reduces the complexity of the network code, and ensures a cleaner and more manageable implementation. The getMusic function within this interface exemplifies a targeted API call that fetches music data based on a user's query.

3.6.2 `MusicData` for Music Listing

To adeptly manage and structure the music data fetched from the Deezer API, our application utilizes the MusicData data class, which is defined in Kotlin. Below is a LaTeX snippet that encapsulates the definition of the MusicData class to provide a comprehensive view of how data is handled and structured in network responses.

Code 7: Data class for handling music data responses from the Deezer API.

```
data class MusicData(
    val `data`: List<Data>,
    val next: String,
    val total: Int
)
```

This class is designed to encapsulate the music data effectively, with the data field storing a list of Data objects (each representing individual tracks or albums), the next field containing a URL for fetching the next set of results, and the total field indicating the total number of entries available.

3.6.3 `fetchMusicAndPreparePlayer` for Music Fetching

The `fetchMusicAndPreparePlayer` function within our application illustrates the approach to fetching music. This method sets up and utilizes Retrofit for network communication, specifically tailored to interact with the Deezer API. It constructs the network client, specifies the API's base URL, and implements a call to fetch music data based on a provided music name. Upon receiving a response, it initializes a media player with the first track's preview URL from the fetched data, setting up listeners for media playback completion to update UI components accordingly. In cases of request failure, it logs errors.

Code 8: Function to fetch music data and prepare media player for playback.

```
private fun fetchMusicAndPreparePlayer(musicName: String) {
    // Example Retrofit setup and call to fetch music data
    val retrofitBuilder = Retrofit.Builder()
        .baseUrl("https://deezerdevs-deezer.p.rapidapi.com/")
        .addConverterFactory(GsonConverterFactory.create())
        .build()
        .create(MusicInterface::class.java)

    val retrofitData = retrofitBuilder.getMusic(musicName)

    retrofitData.enqueue(object : Callback<MusicData?> {
        override fun onResponse(call: Call<MusicData?>,
            response: Response<MusicData?>) {
            val dataList = response.body()?.data!!
            mediaPlayer=MediaPlayer.create(context,dataList[0].
            preview.toUri()).apply {
                setOnCompletionListener {
                    // Reset play/pause button when
                    // music finishes playing
                    isMusicPlaying = false
                    binding.btnPlayPause.setImageResource
                    (android.R.drawable.ic_media_play)
```

```
                }
            }
            // Optionally set the button to show "play" as
            //music is ready to be played
            binding.btnPlayPause.setImageResource
            (android.R.drawable.ic_media_play)
        }

        override fun onFailure(call: Call<MusicData?>,
        t: Throwable) {
            Timber.tag("TAG: onFailure").d("onFailure%s",
            t.message)
        }
    })
}
```

3.6.4 `toggleMusicPlayback` for Playing and Pausing the Music

The toggleMusicPlayback function in our application encapsulates the logic necessary to control music playback interactively. This method checks the current state of music playback and toggles between playing and pausing the music. When invoked, it uses the mediaPlayer instance to either pause the music if it's currently playing or start playing it if it's paused. The UI is updated accordingly with appropriate icons or text on the playback button to reflect the current playback status.

Code 9: Function to toggle music playback based on user interaction.

```
private fun toggleMusicPlayback() {
    mediaPlayer?.let { mp ->
        if (isMusicPlaying) {
            // Pause the music
            mp.pause()
            // Update the button to show the play icon or text
            binding.btnPlayPause.setImageResource
            (android.R.drawable.ic_media_play)
            // or binding.btnPlayPause.setText(R.string.play)
        } else {
            // Play the music
            mp.start()
            // Update the button to show the pause icon or text
            binding.btnPlayPause.setImageResource
            (android.R.drawable.ic_media_pause)
            // or binding.btnPlayPause.setText(R.string.pause)
        }
        // Toggle the music state
        isMusicPlaying = !isMusicPlaying
    }
}
```