Implementing Android Auto in an Audio Player App

Introduction

Android Auto integration is essential for providing a seamless in-car audio experience. They help you bring your media app content to users in their car. In this article, I will walk through my approach to implementing Android Auto support in my audio player app, detailing how content is loaded, structured, and served to the system using MediaLibrarySession.Callback. This guide assumes that you already have a media app that plays audio on a phone. Please, do not confuse Android Auto for Android Automotive OS. See the official guide for the difference.

Overview of the Implementation

The core of the Android Auto integration in this conceptual app revolves around the `AudioService` class. This class extends MusicLibraryService and acts as the entry point for managing media playback and content browsing.

Inside `AudioService`, there is an open inner class called `MusicServiceCallback`, which implements `MediaLibrarySession.Callback`. This class is responsible for handling media browsing requests, including:

- `onGetLibraryRoot()`: Defines the root node of the media hierarchy.
- `onGetChildren()`: Provides the list of media items when a user browses a category or folder.
- `onGetItem()` : Retrieves a specific media item by its ID.
- `onSearch()`: Handles search queries from the user and determines how results are fetched.
- `onGetSearchResult()`: Retrieves and returns search results based on user input.
- `onSetMediaItems()`: Manages setting the media queue when playback is initialized.

Key terms and concepts

Media library service

An Android service implemented by your media app that complies with the MediaLibraryService API. Your app uses this service to expose its content.

Media browser

An API used by media apps to discover media browser services and display their content. Android Auto and Android Automotive OS use a media browser to find your app's media browser service.

Media item

The media browser organizes its content in a tree of MediaItem objects. A media item can have either or both of the following flags:

- FLAG_PLAYABLE: indicates that the item is a leaf on the content tree. The item represents a single sound stream, such as a song on an album, a chapter in an audio book, or an episode of a podcast.
- FLAG_BROWSABLE: indicates that the item is a node on the content tree and it has children. For example, the item represents an album, and its children are the songs on the album.

A media item that is both browsable and playable acts like a playlist. You can select the item itself to play all of its children, or you can browse its children.

Core Classes and Files

Assuming we are integrating Android Auto into a media player app and we are structuring our media browsing experience around **three main destinations**:

- **Home** – Displays dynamic sections loaded from an API.
- **Recently Played** – Loads recently played tracks from an API.
- **Library** – Contains a **Downloads** section with:
    - **Downloaded Tracks** – Individual tracks stored locally.
    - **Podcasts** – Podcasts loaded from the Room database.

To manage the **loading and mapping processes**, we will rely on a central class called `BrowseTree`, along with several Kotlin utility files for handling transformations and structuring the media hierarchy.

Key Classes and Their Responsibilities:

- **`BrowseTree`** – Loads data from APIs and Room local db and manages the overall structure of the media content, organizing it into a hierarchical format that Android Auto can browse.
- **`MappingUtils`** – Provides functions to convert domain models (API/DB data) into `MediaItem` objects.
- **`NodeUtils`** – Defines browsable roots, static nodes, and hierarchical structures (e.g., organizing Home, Library, and Downloads).
- **`MediaItemUtils`** – Generates different types of `MediaItem` instances (playable tracks, browsable folders, static UI elements).
- **`AndroidAutoStateRepository`** – Maintains media playback state across the app and Android Auto, ensuring queue-based playback.
- **`MediaPlaybackParams`** – Holds playback-related details such as track queue, playback position, and playback behavior settings.

Steps for Integrating Android Auto

To integrate Android Auto into your media app, follow these steps:

1. **Add support for Android Auto to your media app**:
    - Ensure your media app is compatible with Android Auto so that it is detected and displayed in the Android Auto launcher.
2. **Declare the Media Library Service in the Manifest:**
    - Add the necessary service declaration in your `AndroidManifest.xml` to inform Android Auto about your media service.
3. **(Optional) Specify Custom App Icons:**
    - Define Android Auto-specific app icons in your `res` folder. If no icons are specified, Android Auto will default to your standard app icons.
4. **Create the Media Library Service:**
    - Extend your music playback service class from `MusicLibraryService` to manage media sessions and playback.
5. **Implement Media Browsing Callbacks:**

    - Create an open inner class within your service that implements `MediaLibrarySession.Callback` to handle media browsing and playback requests efficiently.
6. **Build Your Content Hierarchy:**
    - Override `MediaLibrarySession.Callback`'s callbacks—`onGetLibraryRoot()`and `onGetChildren()`—to structure your media content into a hierarchical tree format, ensuring it is displayed correctly on Android Auto's screen.

⚠️All steps will be explained in detail in the following sections.

Content Loading and Mapping

To efficiently load and structure media content, my implementation relies on the `BrowseTree` class. This acts as the base class for managing the hierarchical organization of media items.

How Content is Processed

1. **Fetching Data:**
    - Content is retrieved from both the back-end and local database.
2. **Mapping to Media Items:**
    - The `BrowseTree` class processes this content and utilizes utility classes to transform domain objects into `MediaItem` objects.
3. **Serving Content:**
    - Once the items are structured into a tree format, they are returned via the `onGetChildren()` callback.

Steps for Integration in detail

1. Add support for Android Auto to your media app

Declare media support for Android Auto by configuring the app manifest's files.
Use the following manifest entry to declare that your phone app supports Android Auto:

```
None
```

```
<application>

    ...

    <meta-data android:name="com.google.android.gms.car.application"

        android:resource="@xml/automotive_app_desc"/>

    ...

</application>
```

This manifest entry refers to an XML file that declares what automotive capabilities your app supports. To indicate that you have a media app, add an XML file named automotive_app_desc.xml to the res/xml/ directory in your project. This file should include the following content:

```
None
```

```
<automotiveApp>

    <uses name="media"/>

</automotiveApp>
```

After doing this the app icon appears on Android Auto screen(infotainment system of the car that is connected to the user's phone which has the app installed) but if you open the app, it will show a blank screen.

2. Declare the Media Library Service in the Manifest

Knowing that we have the AudioService class that implements the MediaLibraryService.
Android Auto connects to your app through your media library service to browse media items. Declare your media library service in your manifest to let Android Auto discover the service and connect to your app.
The following code snippet shows how we declare our media library service in the manifest.

None

```
<application>

...

<service

android:name="com.es.example.infrastructure.audioPlayer.service.Audio
   Service"

   android:enabled="true"

   android:exported="true"

   android:foregroundServiceType="mediaPlayback">

   <intent-filter>

       <action
   android:name="androidx.media3.session.MediaLibraryService"/>

       <action
   android:name="android.media.browse.MediaBrowserService" />

   </intent-filter>

</service>

...

</application>
```

⚠️ Please make sure to set your own correct package name and class names.

3. Specify app icons(optional)

You need to specify app icons that Android Auto and Android Automotive OS can use to represent your app in the system UI. Two icon types are required:

- Launcher icon
- Attribution icon

⚠️ For more information see the Official documentation.

4. Create the Media Library Service

You create a media library service by extending the MediaLibraryService class. Android Auto can then use your service to do the following:

- Browse your app's content hierarchy to present a menu to the user.
- Get the token for your app's SessionToken and MediaSessionCompat object to control audio playback.

You can also use your media library service to let other clients access media content from your app. These media clients might be other apps on a user's phone, or they can be other remote clients.

Design guidelines: As you implement your media browser service, refer to the following app design guidelines:

- Plan navigation tabs
- Media apps

Media Library Service Workflow

This section outlines how Android Auto interacts with a media library service during a typical user workflow.

1. User Launches the Media App on Android Auto

- Android Auto detects the media app and contacts its media library service.

2. Service Initialization (`onCreate()`)

- The media library service initializes a `MediaLibrarySession` to handle playback and browsing requests.
- In `onCreate()`, a `MediaLibrarySession.Callback` instance is assigned to manage media interactions.

Example Implementation:

```
None
override fun onCreate() {
    super.onCreate()
    // Build the MediaLibrarySession and set session activity
    mediaSession = MediaLibrarySession.Builder(
        this.applicationContext, mediaPlayer, getCallback()
    )
```

```
        .setId(UUID.randomUUID().toString())
        .setSessionActivity(sessionActivityPendingIntent)
        .build()
}
// Returns the MediaLibrarySession.Callback instance
private fun getCallback(): MediaLibrarySession.Callback {
    return MediaServiceCallback()
}
```

To make sure Android Auto detects and interacts with your media session, you must override `onGetSession()` in your media service. If this method is not properly implemented, **nothing will appear on Android Auto**.

Make sure to add the following code to your `MediaLibraryService` implementation:

```
None
override fun onGetSession(controllerInfo: MediaSession.ControllerInfo):
MediaLibrarySession? {
       return if ("android.media.session.MediaController" ==
controllerInfo.packageName
        || packageValidator.isKnownCaller(controllerInfo.packageName,
controllerInfo.uid)
    ) {
        mediaSession
    }
}
```

This function ensures that only known callers (such as Android Auto) can access the media session. The `isKnownCaller()` function must be correctly implemented to validate incoming connections. If this step is missed, Android Auto will not show any media content.

⚠️ To test your implementation first you can just return the `mediaSession` without validation.

```
None
override fun onGetSession(controllerInfo: MediaSession.ControllerInfo):
MediaLibrarySession? {
     return mediaSession
}
```

3. Retrieving the Root Media Item (`onGetLibraryRoot()`)

- Android Auto requests the root media item to establish the content hierarchy.
- This root is not displayed but is used to fetch available media categories(the rest of the tree).

4. Browsing Content (`onGetChildren()`)

- If the user navigates a browsable category (e.g., playlists, albums, or podcasts), `onGetChildren()` is triggered to load its media items.

5. Playing Media (`onGetItem()`)

- When the user selects a playable item (e.g., a song or a podcast episode), the system retrieves it and begins playback.

6. Searching for Media (`onSearch()`) *(if supported)*

- Users can search for specific media content using voice or text input.
- Android Auto triggers `onSearch()`, allowing the service to return matching media items.

This structured workflow ensures that a media app provides a seamless browsing and playback experience when integrated with Android Auto.

5. Implement Media Browsing Callbacks (Build the content hierarchy)

Android Auto and Android Automotive OS call your app's media library service to find out what content is available. You need to implement two methods in your media browser service to support this: onGetLibraryRoot and onGetChildren

Implement onGetRoot

Your service's onGetLibraryRoot method returns information about the root node of your content hierarchy. Android Auto and Android Automotive OS use this root node to request the rest of your content using the onGetChildren method.
onGetLibraryRoot

This function, onGetLibraryRoot, is an override of a method in MediaLibraryService. It is responsible for handling requests from media browser clients (such as Android Auto, Google Assistant, or other media browsing apps) to retrieve the root media item of the library. Here's a breakdown of how it works:

onGetLibraryRoot()

Function Logic for Handling Media Library Requests

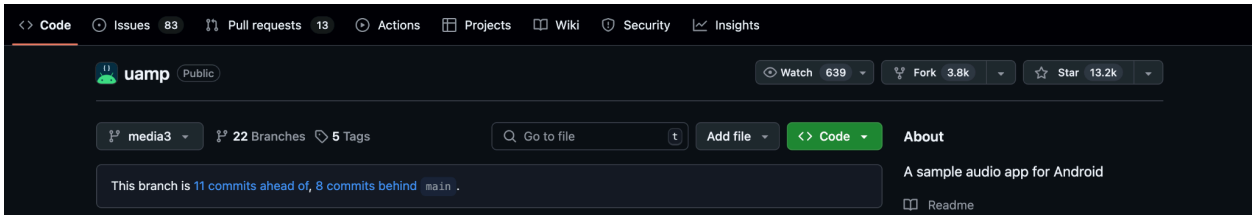
1. Verify the Caller

Before processing a media request, ensure that the requesting application is authorized.

```
None

val isKnownCaller = packageValidator.isKnownCaller(browser.packageName,
browser.uid)
```

This check ensures that only trusted clients can interact with the media service.
⚠️PackageValidator is a class from the official github repo of UAMP or validate in any other custom way.
⚠️Make sure you are on the **media3** branch.



2. Set Up Metadata for Media Browsing

Define a bundle of extras to inform the client about supported features like media search and content styling.

```
None

val rootExtras = Bundle().apply {

    putBoolean(MEDIA_SEARCH_SUPPORTED, true)

    putBoolean(CONTENT_STYLE_SUPPORTED, true)

    putInt(CONTENT_STYLE_BROWSABLE_HINT, CONTENT_STYLE_LIST)

    putInt(CONTENT_STYLE_PLAYABLE_HINT, CONTENT_STYLE_LIST)

    putBoolean(BROWSER_SERVICE_EXTRAS_KEY_SEARCH_SUPPORTED, true)

}
```

These extras guide the media browsing experience by defining how items should be displayed.

3. Construct Library Parameters

Ensure that the response includes the appropriate metadata about supported capabilities.

```
None

val libraryParams =
LibraryParams.Builder().setExtras(rootExtras).build()
```

This step packages the browsing parameters for the client.

4. Determine User Authentication Status(optional-based on your requirements)

Before proceeding, check if the user is logged into the app.

```
None

val isLoggedIn = LoginStateRepository.isLoggedIn.value
```

This helps in deciding whether to grant access to media content. If not the app will show a static MediaItem that requests login from the user to give him access to the app content.

5. Select the Appropriate Root Media Item

Decide what content to return based on login status and caller validation.

```
None

val rootMediaItem = when {

    !isLoggedIn -> browseTree.getErrorRootNode() // Return an error if
not logged in

    !isKnownCaller -> getPlaceholderMediaItem() // Return a placeholder
for unknown callers

    params?.isRecent == true -> {

        if (player.currentTimeline.isEmpty) {

            storage.loadRecentSong()?.let {

                preparePlayerForResumption(it)

            }

        }

        browseTree.getBrowsableRoot()

    }

    else -> browseTree.getBrowsableRoot() // Return the root node for
browsing

}
```

- If the user is not logged in, return an error message.
- If the caller is not recognized, return a placeholder item.
- If the request is for recent media, check if there is an active timeline; otherwise, retrieve the last played song.
- Otherwise, return the standard browsable root.

⚠️ storage is an instance of shared preference class that is mainly used to save recent song data and retrieve it when needed.
⚠️ BrowseTree class and its functions will be explained at a later point.

6. Return the Result

Finally, wrap the selected media item inside a LibraryResult and return it immediately.

```
None

return Futures.immediateFuture(LibraryResult.ofItem(rootMediaItem,
libraryParams))
```

This ensures that the response is processed efficiently and returned without delay.

Summary

- This function decides what media item to return as the root based on:
  - Whether the caller is recognized.
  - Whether the user is logged in.
  - Whether the client is requesting recent content.
- It provides metadata (rootExtras) about search and UI styling.
- If the user is logged out, an error root is returned.
- If the caller is unknown, a placeholder is returned.
- If the caller is known and requesting recent media, it loads the last played song if needed.
- Otherwise, it returns the standard browsable root.

This is crucial for ensuring secure access control and customizing the media library experience based on the client's capabilities and user authentication status.

onGetChildren()

Android Auto calls your service's onGetChildren() method to get the children of the root media item. Android Auto displays these media items as the top level of content items.

Structure the root menu

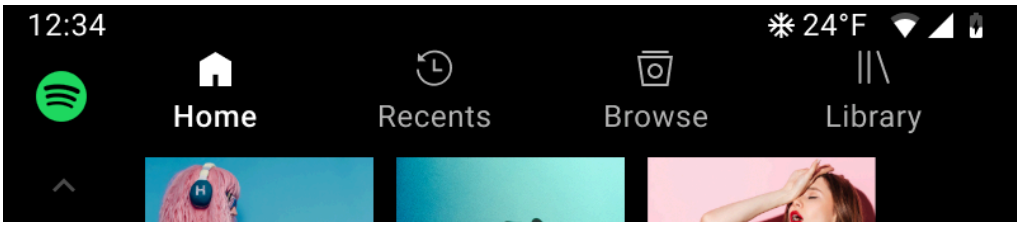


Figure 2. Root content displayed as navigational tabs.

Android Auto and Android Automotive OS have specific constraints about the structure of the root menu. These are communicated to the MediaLibraryService through root hints, which can be read through the Bundle argument passed into onGetLibraryRoot(). Following these hints lets the system display the root content optimally as navigational tabs. If you don't follow these hints, some root content might be dropped or made less discoverable by the system. Two hints are sent:

- A limit on the number of root children: for the majority of cases, you can expect this number to be four. This means that over four tabs cannot be shown.
- Supported flags on the root children: you can expect this value to be MediaItem#FLAG_BROWSABLE. This means that only browsable items—not playable items—can be shown as tabs.

Guide to implementing onGetChildren()

In Android's `MediaLibrarySession`, the `onGetChildren` function is responsible for retrieving media items based on a `parentId`. This function is crucial in structuring a media app's library, allowing users to browse playlists, albums, and other content dynamically.

Here we are fetching our domain objects using our BrowsTree functions and also you can use it to load objects from local db and map them into MediaItems then return them.

The returned result here MUST be deferred since we are fetching data in background and it takes some time.

This guide provides a sample implementation, breaks down the logic, and explains best practices for handling media items efficiently.

```
None
override fun onGetChildren(
    session: MediaLibrarySession,
    browser: MediaSession.ControllerInfo,
    parentId: String,
    page: Int,
    pageSize: Int,
    params: LibraryParams?,
): ListenableFuture<LibraryResult<ImmutableList<MediaItem>>> {
```

```
    val deferredResult =
SettableFuture.create<LibraryResult<ImmutableList<MediaItem>>>()

    when (parentId) {
        ROOT_ID -> {
            val children = if (LoginStateRepository.isLoggedIn.value) {
                browseTree.getRootItems()
            } else {
                listOf(browseTree.getErrorLoggedOutMediaItem())
            }
            setDeferredResult(deferredResult, params, children)
        }

        HOME_ID -> {
            serviceScope.launch {
                browseTree.homeResult.collect { result ->
                    handleResource(result, onSuccess = {
                        val items =
browseTree.mapHomeToMediaItems(result?.data)
                        setDeferredResult(deferredResult, params, items)
                    }, onEmpty = {
                        setEmptyDeferredResult(deferredResult, params)
                    })
                }
            }
        }

        LIBRARY_PLAYLISTS -> {
            serviceScope.launch {
                browseTree.playlistsResponse.collect { response ->
                    handleResource(response, onSuccess = {
                        val items =
browseTree.mapPlaylistsToMediaItems(response?.data)
                        setDeferredResult(deferredResult, params, items)
                    }, onEmpty = {
                        setEmptyDeferredResult(deferredResult, params)
                    })
                }
            }
        }

        else -> {
            val parentItem = browseTree.getMediaItemByMediaId(parentId)
            if (parentItem == null) {
```

```
                setEmptyDeferredResult(deferredResult, params)
            } else {
                handleSpecificItem(parentItem, parentId, deferredResult,
params)
            }
        }
    }
    return deferredResult
}

private fun handleSpecificItem(
    parentItem: MediaItem,
    parentId: String,
    deferredResult: SettableFuture<LibraryResult<ImmutableList<MediaItem>>>,
    params: LibraryParams?
) {
    when (parentItem.mediaMetadata.extras?.getString(EXTRA_ITEM_TYPE)) {
        TYPE_PLAYLIST -> listenToPlaylistTracksResponse(parentId,
deferredResult, params)
        TYPE_ALBUM -> listenToAlbumTracksResponse(parentId, deferredResult,
params)
        TYPE_PODCAST -> listenToPodcastEpisodesResponse(parentId,
deferredResult, params)
        else -> setEmptyDeferredResult(deferredResult, params)
    }
}
```

Function Signature

None

```
override fun onGetChildren(

    session: MediaLibrarySession,

    browser: MediaSession.ControllerInfo,

    parentId: String,

    page: Int,

    pageSize: Int,
```

```
    params: LibraryParams?,

): ListenableFuture<LibraryResult<ImmutableList<MediaItem>>>
```

Parameters

- **session**: The active MediaLibrarySession handling the request.
- **browser**: Information about the app requesting the media items.
- **parentId**: The ID of the parent media item whose children are being requested.
- **page** & **pageSize**: Used for paginated loading.
- **params**: Additional request parameters, such as recent content filters.

Return Type

- The function returns a ListenableFuture<LibraryResult<ImmutableList<MediaItem>>>, which:
    - Ensures **asynchronous execution**.
    - Returns a LibraryResult containing an immutable list of MediaItem objects.

Implementation Details

1️⃣ Create a Deferred Result

```
None

val deferredResult =
SettableFuture.create<LibraryResult<ImmutableList<MediaItem>>>()
```

- Uses SettableFuture to **defer** the response until data is available asynchronously.

2️⃣ Handle Different Parent IDs

Root Items (ROOT_ID)

```
None

val children = if (LoginStateRepository.isLoggedIn.value) {

    browseTree.getRootItems()
```

```
} else {
    listOf(browseTree.getErrorLoggedOutMediaItem())
}
setDeferredResult(deferredResult, params, children)
```

- If logged in, retrieve the root items.
- If not logged in, return an error item prompting login.

Home Section (HOME_ID)

```
None
serviceScope.launch {
    browseTree.homeResult.collect { result ->
        handleResource(result, onSuccess = {
            val items = browseTree.mapHomeToMediaItems(result?.data)
            setDeferredResult(deferredResult, params, items)
         }, onEmpty = {
            setEmptyDeferredResult(deferredResult, params)
        })
    }
}
```

- Fetch **dynamic content** from API (homeResult).
- Convert it into MediaItem objects.

Library Playlists (LIBRARY_PLAYLISTS)

```
None

serviceScope.launch {

    browseTree.playlistsResponse.collect { response ->

        handleResource(response, onSuccess = {

            val items = browseTree.mapPlaylistsToMediaItems(response?.data)

            setDeferredResult(deferredResult, params, items)

        }, onEmpty = {

            setEmptyDeferredResult(deferredResult, params)

        })

    }}
```

- Fetch **playlists** and return their media items.

Other Requests

```
None

val parentItem = browseTree.getMediaItemByMediaId(parentId)

if (parentItem == null) {

    setEmptyDeferredResult(deferredResult, params)

} else {

    handleSpecificItem(parentItem, parentId, deferredResult, params)

}
```

- If parentId exists, process the specific type (playlist, album, podcast).

Handling Specific Items

```
None

when (parentItem.mediaMetadata.extras?.getString(EXTRA_ITEM_TYPE)) {

    TYPE_PLAYLIST -> listenToPlaylistTracksResponse(parentId, deferredResult,
params)

    TYPE_ALBUM -> listenToAlbumTracksResponse(parentId, deferredResult, params)

    TYPE_PODCAST -> listenToPodcastEpisodesResponse(parentId, deferredResult,
params)

    else -> setEmptyDeferredResult(deferredResult, params)

}
```

- **Playlists (TYPE_PLAYLIST)** → Fetch tracks in the playlist.
- **Albums (TYPE_ALBUM)** → Fetch tracks in the album.
- **Podcasts (TYPE_PODCAST)** → Fetch podcast episodes.
- **Unknown types** → Return an empty result.

Summary

1. **Android Auto requests child items** of a given parentId.
2. The function **determines what type of media is requested**:
    - If **Home** → Fetch dynamic sections from API.
    - If **Playlists** → Fetch playlists from API.
    - If **Library root** → Check login status.
    - If **Specific items** (playlist, album, podcast) → Fetch their contents.
3. The function **defers the result** until the data is ready, ensuring non-blocking execution.

This function is essential for **populating the media browser UI in Android Auto**. Without it, users wouldn't see any media categories or playable content.

Best Practices

1. Use Coroutines for Asynchronous Fetching

- **Why?** Fetching media items often involves network or database operations, which should not block the main thread.
- **How?** Use `serviceScope.launch` to call repository functions asynchronously.

2. Implement Proper Error Handling

- Provide an error media item for cases where fetching fails.
- Ensure that empty results do not crash the session.

3. Use Consistent ID Formats

- Keep a structured format for `parentId` (e.g., `playlist_123`, `album_456`) to make identification easier.

onGetItem()

The `onGetItem()` function is responsible for retrieving a specific media item based on its unique identifier (`mediaId`). This function is crucial for ensuring that Android Auto and other media browsing clients can request and display individual media items correctly.

```
None

override fun onGetItem(

session: MediaLibrarySession,

browser: MediaSession.ControllerInfo,

mediaId: String,

): ListenableFuture<LibraryResult<MediaItem>> {

Log.d(TAG, "onGetItem: unique id: $mediaId")

val params = LibraryParams.Builder().build()

val item = browseTree.getMediaItemByMediaId(mediaId)

val placeholderItem = getPlaceholderMediaItem()

return try {

Futures.immediateFuture(

LibraryResult.ofItem(
```

```
item ?: placeholderItem,
params
)
)
} catch (e: Exception) {
e.printStackTrace()
Futures.immediateFuture(
LibraryResult.ofItem(
placeholderItem,
params
))}}
```

Function Logic Breakdown

Function Signature

```
None
override fun onGetItem(
    session: MediaLibrarySession,
    browser: MediaSession.ControllerInfo,
    mediaId: String,
): ListenableFuture<LibraryResult<MediaItem>>
```

- **session**: The active `MediaLibrarySession`.
- **browser**: The media client requesting the item.
- **mediaId**: The unique identifier of the requested media item.

Logging for Debugging

```
None
```

```
Log.d(TAG, "onGetItem: unique id: $mediaId")
```

- Logs the unique media ID and its original form for debugging and tracking requests.

Preparing Library Parameters

```
None
```

```
val params = LibraryParams.Builder().build()
```

- Creates an instance of `LibraryParams`, which can include additional metadata about the request.

Retrieving the Media Item

```
None
```

```
val item = browseTree.getMediaItemByMediaId(mediaId)

val placeholderItem = getPlaceholderMediaItem()
```

- **`item`**: Attempts to retrieve the requested media item from `browseTree`.
- **`placeholderItem`**: A fallback item used if the requested media item does not exist.

Handling Success and Failure

```
None
```

```
return try {

    Futures.immediateFuture(

        LibraryResult.ofItem(

            item ?: placeholderItem,

            params
```

```
        )
    )
} catch (e: Exception) {
    e.printStackTrace()
    Futures.immediateFuture(
        LibraryResult.ofItem(
            placeholderItem,
            params
        ))
}
```

- If an item is found, it is returned.
- If the item is null, the function falls back to returning `placeholderItem`.
- If an exception occurs during processing, it logs the error and returns `placeholderItem` to ensure a valid response is always provided.

Why This Is Important

1. **Ensures Stability**: Even if an item is missing or an error occurs, a placeholder is always returned, preventing crashes.
2. **Enhances Debugging**: Logging helps track issues with media ID resolution.
3. **Supports Media Browsing**: Essential for Android Auto and other clients to request and display individual media items.

Best Practices

- Always validate `mediaId` before processing the request.
- Ensure `browseTree.getMediaItemByMediaId(mediaId)` correctly maps media IDs to items.
- Implement meaningful placeholder items to improve user experience.

This implementation ensures a seamless integration with Android Auto, allowing users to browse and play media effortlessly.

onSearch() & onGetSearchResult()

These functions handle search functionality within a MediaLibrarySession in Android's media playback system. They allow a media browser (such as a media player app) to perform searches and retrieve results.
This function ensures a smooth user experience by always returning a valid media item, even if the requested one is missing or an error occurs.

onSearch()

The `onSearch` function is a callback method in a `MediaLibrarySession`, triggered when a media browser (such as an app or voice assistant) requests a search operation. This function processes search queries and returns relevant media items.

Key Responsibilities

1. *Logging the Search Query*
    - Logs the received search query for debugging purposes.
2. *Checking User Authentication*
    - Verifies whether the user is logged in using `LoginStateRepository`.
    - If the user is not logged in, returns an error media item.
3. *Performing the Search*
    - Calls `browseTree.searchMediaItems(query)` to retrieve relevant media items.
4. *Notifying the Browser of Search Results*
    - Notifies the media browser (e.g., UI or assistant) that search results have changed using `notifySearchResultChanged`.
5. *Returning a Future Result*
    - Returns a `ListenableFuture<LibraryResult<Void>>`, indicating that the operation has completed.

Generic Implementation Approach

When implementing `onSearch` in a media playback service, follow these steps:

1. Implement the Method in Your `MediaLibrarySession.Callback`

None

```
override fun onSearch(
    session: MediaLibrarySession,
    browser: MediaSession.ControllerInfo,
    query: String,
    params: LibraryParams?,
): ListenableFuture<LibraryResult<Void>> {
    Log.d(TAG, "onSearch query: $query")
    // Check if the user is authenticated
    val isLoggedIn = LoginStateRepository.isLoggedIn.value
    val searchResult = if (!isLoggedIn) {
          mutableListOf(browseTree.getErrorLoggedOutMediaItem()) // Return an error item
    } else {
        browseTree.searchMediaItems(query) // Perform search
    }
    // Notify the client about the search result update
    mediaSession.notifySearchResultChanged(browser, query, searchResult.size, params)
    // Return an immediate future result
    return Futures.immediateFuture(LibraryResult.ofVoid())
}
```

2. Handling Authentication

- Ensure that search results are only accessible to authenticated users if required.
- Provide a fallback response (e.g., an error media item) for unauthenticated users.

3. Implementing the `searchMediaItems` Function

This function should query the media catalog and return a list of media items matching the search criteria.

Example:

```
None
fun searchMediaItems(query: String): List<MediaItem> {

    return mediaItems.filter { it.title.contains(query, ignoreCase = true) }

}
```

4. Notifying Search Result Changes

- Use `notifySearchResultChanged()` to inform the media browser of updated search results.
- This is crucial for updating UI elements in the media browsing client.

5. Returning the Result

- `Futures.immediateFuture(LibraryResult.ofVoid())` is used when the result does not contain actual media data but only notifies the client that the search operation is complete.

onGetSearchResult()

The `onGetSearchResult` function is a callback method in a `MediaLibrarySession`, triggered when a media browser (such as an app or voice assistant) requests paginated search results. This function processes search queries, retrieves the appropriate page of results, and returns a list of media items.

Key Responsibilities

1. *Logging the Search Query*
    - Logs the received search query for debugging purposes.

2. *Checking User Authentication*
    - Verifies whether the user is logged in using `LoginStateRepository`.
    - If the user is not logged in, returns an error media item.
3. *Performing the Search*
    - Calls `browseTree.searchMediaItems(query)` to retrieve relevant media items.
4. *Paginating the Results*
    - Extracts the appropriate subset of search results based on the requested `page` and `pageSize`.
5. *Notifying the Browser of Search Results*
    - Notifies the media browser (e.g., UI or assistant) that search results have changed using `notifySearchResultChanged`.
6. *Returning a Future Result*
    - Returns a `ListenableFuture<LibraryResult<ImmutableList<MediaItem>>>`, containing the search results.

Generic Implementation Approach

When implementing `onGetSearchResult` in a media playback service, follow these steps:

1. Implement the Method in Your `MediaLibrarySession.Callback`

None

```
override fun onGetSearchResult(

    session: MediaLibrarySession,

    browser: MediaSession.ControllerInfo,

    query: String,

    page: Int,

    pageSize: Int,
```

```
    params: LibraryParams?,
): ListenableFuture<LibraryResult<ImmutableList<MediaItem>>> {
    Log.d(TAG, "onGetSearchResult query: $query")
    // Check if the user is authenticated
     val isLoggedIn = LoginStateRepository.isLoggedIn.value
    val searchResult = if (!isLoggedIn) {
          mutableListOf(browseTree.getErrorLoggedOutMediaItem()) // Return an
error item
    } else {
        browseTree.searchMediaItems(query) // Perform search
    }
    // Notify the client about the search result update
    mediaSession.notifySearchResultChanged(browser, query, searchResult.size,
params)
    // Return a paginated result
    return Futures.immediateFuture(LibraryResult.ofItemList(searchResult,
params))
}
```

2. Handling Authentication

- Ensure that search results are only accessible to authenticated users if required.
- Provide a fallback response (e.g., an error media item) for unauthenticated users.

3. Implementing the `searchMediaItems` Function

This function should query the media catalog and return a list of media items matching the search criteria.

Example:

```
None
fun searchMediaItems(query: String): List<MediaItem> {

    return mediaItems.filter { it.title.contains(query, ignoreCase = true) }

}
```

4. Implementing Pagination

- Extract the correct subset of search results using the `page` and `pageSize` parameters.

Example:

```
None
fun paginateResults(results: List<MediaItem>, page: Int, pageSize: Int):
List<MediaItem> {

    val fromIndex = page * pageSize

    if (fromIndex >= results.size) return emptyList()

    val toIndex = minOf(fromIndex + pageSize, results.size)

    return results.subList(fromIndex, toIndex)

}
```

5. Notifying Search Result Changes

- Use `notifySearchResultChanged()` to inform the media browser of updated search results.
- This is crucial for updating UI elements in the media browsing client.

6. Returning the Result

- `Futures.immediateFuture(LibraryResult.ofItemList(searchResult, params))` is used to return a list of media items to the client.

onSetMediaItems()

This function is triggered when a media controller (such as Android Auto) sets a list of media items for playback(When the user clicks on a playable MediaItem). I have implemented it to ensure that the correct queue of media items is established, handling a known **Android Auto** bug where only the clicked item is set for playback instead of the entire list.

Key Responsibilities

1. *Logging the Received Media Items*
    - Logs the size and media IDs of the received media items for debugging purposes.
2. *Handling Android Auto Playback Bug*
    - Android Auto plays only the clicked playable item instead of the full list. This function works around that limitation.
3. *Retrieving the Clicked Item*
    - Retrieves the clicked media item from `browseTree` using its `mediaId`.
4. *Finding the Item Index in the Global Queue*
    - Searches `AndroidAutoStateRepository.globalMediaItems` for the clicked item's index.
5. *Setting Media Playback Parameters*
    - If the clicked item exists in the global media queue, playback parameters are set accordingly.
    - If not, a new media queue is created using the clicked item and mapped media items.
6. *Returning the Media Items with Start Position*
    - Returns a `ListenableFuture<MediaSession.MediaItemsWithStartPosition>`, ensuring proper playback handling.

Generic Implementation Approach

When implementing `onSetMediaItems` in a media playback service, follow these steps:

1. Implement the Method in Your `MediaSession.Callback`

None

```
override fun onSetMediaItems(

    mediaSession: MediaSession,

    controller: MediaSession.ControllerInfo,

    mediaItems: MutableList<MediaItem>,

    startIndex: Int,

    startPositionMs: Long,

): ListenableFuture<MediaSession.MediaItemsWithStartPosition> {

    Log.d("MediaSession", "Initial media items: ${mediaItems.size}")

    mediaItems.forEach { Log.d("MediaSession", "Original MediaId:
${it.mediaId}") }

    val itemClicked = browseTree.getMediaItemByMediaId(mediaItems[0].mediaId)

    Log.d("MediaSession", "itemClicked: ${itemClicked?.mediaId}")

    val itemIndexInQueue =

        AndroidAutoStateRepository.globalMediaItems.value.indexOfFirst {
it.mediaId == itemClicked?.mediaId }

    Log.d("MediaSession", "itemIndexInQueue: $itemIndexInQueue")

    if (itemIndexInQueue != -1) {

        AndroidAutoStateRepository.setMediaPlaybackParams(

            params = MediaPlaybackParams(

                AndroidAutoStateRepository.globalMediaItems.value,

                AndroidAutoStateRepository.globalTracks.value,

                itemIndexInQueue,

                true,

                startPositionMs
```

```
            )
        )
        return Futures.immediateFuture(
            MediaSession.MediaItemsWithStartPosition(
                AndroidAutoStateRepository.globalMediaItems.value,
                itemIndexInQueue,
                startPositionMs
            )
        )
    } else {
        val mapped = mediaItems.map {
            browseTree.getMediaItemByMediaId(it.mediaId) ?: MediaItem.EMPTY
        }.toMutableList()
        val tracks = listOf(exoPlayerMapper.asAudioItem(mapped[0])!!)
        AndroidAutoStateRepository.setMediaPlaybackParams(
            params = MediaPlaybackParams(
                mapped,
                tracks.toMutableList(),
                Constants.FIRST_ITEM_POSITION,
                true,
                startPositionMs
            )
        )
        return Futures.immediateFuture(
```

```
            MediaSession.MediaItemsWithStartPosition(
                mapped,
                Constants.FIRST_ITEM_POSITION,
                startPositionMs
            )
        )
    }
}
```

2. Handling Android Auto Issues

- Android Auto might send only the clicked media item instead of the full playlist.
- The function reconstructs the queue by checking the global media list.

3. Retrieving and Validating Media Items

- Uses `browseTree.getMediaItemByMediaId(mediaItems[0].mediaId)` to find the actual media item.
- Maps `mediaItems` to ensure they are valid and not empty.

4. Updating Playback Parameters

- If the item exists in the queue, it updates playback parameters using `AndroidAutoStateRepository.setMediaPlaybackParams`.
- Otherwise, it creates a new playback session with a mapped list.

5. Returning the Result

- Uses `Futures.immediateFuture(MediaSession.MediaItemsWithStartPosition(...))` to return the updated media queue.

Core classes explained in more detail

BrowseTree - Media Browsing Structure for AudioService

Overview

The `BrowseTree` class is a core component of the `AudioService` in this Android music app. It is responsible for structuring media content in a hierarchical format, enabling seamless browsing of albums, artists, playlists, podcasts, and tracks.

Purpose

The primary role of `BrowseTree` is to:

- Organize MediaItems into a tree structure that can be browsed efficiently.
- Retrieve media data dynamically from repositories and use cases.
- Maintain and manage media browsing state using Kotlin's `StateFlow` for real-time updates.

Key Responsibilities

1. Managing Media Hierarchy

- Maps **media IDs** to their corresponding child items, such as an album ID mapping to its tracks.
- Stores **MediaItems** in an organized way to facilitate efficient lookup.

2. Fetching Media Data

- Retrieves media content from repositories and use cases (e.g., fetching all albums, playlists, podcasts, or tracks for a given entity).
- Updates the browsing tree dynamically as media content changes.

3. State Management

- Uses `MutableStateFlow<Resource<T>>` to track loading states and update UI components.
- Ensures real-time updates when media content is modified.

Dependencies

The `BrowseTree` class depends(those are injected in the constructor via HILT) on multiple repositories and use cases for fetching media data from remote and local db:

```
None

class BrowseTree @Inject constructor(

@ApplicationContext val context: Context,

@CoroutineScopeIO val ioScope: CoroutineScope,

private val homeUseCase: HomeUseCase,

private val getRecentlyPlayedUseCase: GetRecentlyPlayedUseCase,

tracksLocalRepo: TracksLocalRepo,

private val podcastsLocalRepo: PodcastsLocalRepo,

) {

... }
```

Use Cases

Example of use cases to fetch and manage media items from remote:

- `HomeUseCase` – Retrieves home screen data.
- `GetRelatedApiPlaylistUseCase` – Loads related playlists from API.

Local Repositories

Example of repositories to provide locally stored media data:

- `TracksLocalRepo` - Retrieves downloaded single tracks and episodes.
- `PodcastsLocalRepo` - Retrieves downloaded podcasts.

State Management

The `BrowseTree` maintains various state flows to track the loading state and availability of different media sections:

```
None
private val _homeResult =
MutableStateFlow<Resource<HomeResult>?>(null)

val homeResult: StateFlow<Resource<HomeResult>?> =
_homeResult.asStateFlow()
```

Available State Flows:

- `homeResult`: Stores home screen data.
- `recentlyPlayedTracks`: Manages recently played tracks.

```
None
{

// HOME

val homeResult: StateFlow<Resource<HomeResult>?> = _homeResult.asStateFlow()

// RECENTLY PLAYED

private val _recentlyPlayedTracks =
MutableStateFlow<Resource<TracksResponse>?>(null)

val recentlyPlayedTracks: StateFlow<Resource<TracksResponse>?> =

_recentlyPlayedResponse.asStateFlow()

...
```

Load local tracks and podcasts

```
None
val downloadedTracks = tracksLocalRepo.getSingleTracks()
```

```
val downloadedPodcasts = podcastsLocalRepo.readAllPodcasts()
```

Companion Object

```
None
companion object {
   private val mediaIdToChildren = mutableMapOf<String,
MutableList<MediaItem>>()
   private val mediaIdToMediaItem = mutableMapOf<String, MediaItem>()

   fun addMediaItem(mediaItem: MediaItem) {
       // This is to remove duplications
       if (mediaIdToMediaItem.contains(mediaItem.mediaId)) {
           mediaIdToMediaItem.remove(mediaItem.mediaId)
       }

       mediaIdToMediaItem[mediaItem.mediaId] = mediaItem
   }

   fun addChildren(parentId: String, children: MutableList<MediaItem>) {
       // This is to remove duplications
       if (mediaIdToChildren.contains(parentId)) {
           mediaIdToChildren.remove(parentId)
       }
       mediaIdToChildren[parentId] = children
   }
}
```

This companion object serves as an in-memory cache for media items and their hierarchical relationships, ensuring efficient retrieval and preventing duplicates. Here's a breakdown:

1. Usage

- The `companion object` allows these properties and functions to be shared across all instances of the containing class.

- It maintains two maps for quick lookup of media items and their parent-child relationships.

2. Variables

```
None

private val mediaIdToChildren = mutableMapOf<String, MutableList<MediaItem>>()
```

- Stores a mapping between a **parent media ID** (e.g., an album or playlist ID) and its **list of child media items** (e.g., tracks).
- Allows quick retrieval of all child items given a parent ID.

```
None

private val mediaIdToMediaItem = mutableMapOf<String, MediaItem>()
```

- Stores a **mapping of media IDs to individual media items**.
- Ensures that each media item can be retrieved efficiently by its unique `mediaId`.

3. Functions

addMediaItem(mediaItem: MediaItem)

```
None

fun addMediaItem(mediaItem: MediaItem) {
    if (mediaIdToMediaItem.contains(mediaItem.mediaId)) {
        mediaIdToMediaItem.remove(mediaItem.mediaId)
    }
    mediaIdToMediaItem[mediaItem.mediaId] = mediaItem
}
```

- Ensures that **duplicate media items do not exist** in the map.

- If a media item with the same `mediaId` exists, it is removed before adding the new one.
- This prevents stale or outdated media items from remaining in the cache.

addChildren(parentId: String, children: MutableList<MediaItem>)

```
None

fun addChildren(parentId: String, children: MutableList<MediaItem>) {

    if (mediaIdToChildren.contains(parentId)) {

        mediaIdToChildren.remove(parentId)

    }

    mediaIdToChildren[parentId] = children

}
```

- Associates a **parent media ID** with a list of **child media items**.
- Ensures that previous child associations are cleared before adding new ones, preventing duplicate or incorrect data.

Example Queries

- A media ID like `"Albums"` fetches all albums.
- A media ID like `"Album_A"` fetches all tracks in that album.
- A media ID like `"Playlist_X"` retrieves all tracks in that playlist.

Class Functions

The init block

Since BrowseTree is instantiated at service onCreate then the init block is used to call functions that fetches needed data from both remote and local via injected dependencies then mapping them into a tree of MediaItems to draw the required screens on Android Auto's UI.

Example init block

```
None
init {
// Static nodes
   buildAppRootList()
   buildLibraryRoot()
   buildLibraryDownloadsRoot()
// Dynamic nodes
   getHome()
   getRecentlyPlayed()
}
```

Fetching data, mapping and search functions

Function	Description
`getHome()`	Fetches home and highlights data asynchronously and updates responses.
`getRecentlyPlayed()`	Fetches recently played tracks and updates response.
`getPodcastWithEpisodesFromDB(podcastId: Int)`	Retrieves podcast with episodes from local database.
`mapHomeToMediaItems(sections: List<HomeSection?>?)`	Maps home sections to media items.

`mapDownloadedTracksToMediaItems(downloadedTracks: List<TrackEntity>, parentNodeId: String)`	Maps downloaded tracks to media items.
`mapDownloadedPodcastsToMediaItems(downloaded: List<PodcastEntity>, parentNodeId: String)`	Maps downloaded podcasts to media items.
`searchMediaItems(query: String)`	Searches media items based on a query string.

📌These functions use higher-order functions from these util files, MappingUtils, NodeUtils and MediaItemUtils.

MappingUtils.kt

This file provides utility functions to map various domain models into `MediaItem` objects. These `MediaItem` objects are used within an audio playback system, integrated with `androidx.media3.common.MediaItem`.

Class Example Functions

mapHomeToMediaItems()

Converts a list of `HomeSection` objects into `MediaItem` objects by calling `mapHomeSectionContentsToMediaItems` function in MappingUtils.kt for detailed mapping.

None

```
fun mapHomeToMediaItems(

   sections: List<HomeSection?>?,

): MutableList<MediaItem> {

   return mapHomeToMediaItems(sections, context)
```

```
}
```

mapTracksToMediaItems()

Converts a list of `Tracks` objects into playable `MediaItem` objects by calling `createPlayableMediaItem` function in MappingUtils.kt for detailed mapping.

```
None
fun mapTracksToMediaItems(
   tracks: MutableList<Track>?,
   parentNodeId: String,
): MutableList<MediaItem> {
   val mediaItemsList = mutableListOf<MediaItem>()

   tracks?.map { track ->
       createPlayableMediaItem(
           track.id,
           track.name,
           track.subtitle(),
           track.image?.url,
           track.file?.gcpUrl,
           track.type,
           track,
       )
   }?.also {

       mediaItemsList.addAll(it)
       mediaItemsList.forEach { item ->
           addMediaItem(item)
       }
       addChildren(parentNodeId, mediaItemsList)
   }
   return mediaItemsList
}
```

NodeUtils.kt

This file contains utility functions for building and managing the media browsing tree in the audio player. The functions help define browsable roots, static nodes, and hierarchical structures for different media categories, such as home, library, and downloads.

Class Example Functions

`getAppBrowsableRoot`

Creates and returns the main browsable root node for the media browsing tree. The node is added to the provided `mediaIdToMediaItem` map.

```
None
fun getAppBrowsableRoot(mediaIdToMediaItem: MutableMap<String, MediaItem>):
MediaItem {
   val rootNode = MediaItem.Builder()
       .setMediaId(APP_BROWSABLE_ROOT)
       .setMediaMetadata(
           MediaMetadata.Builder()
               .setFolderType(MediaMetadata.FOLDER_TYPE_ALBUMS)
               .setIsPlayable(false)
               .build()
       )
       .build()
   mediaIdToMediaItem[rootNode.mediaId] = rootNode
   return rootNode
}
```

`buildAppRootList`

Builds the primary root-level navigation structure, for example:

- Home
- Recently Played

- Library

Each node is added to the `mediaIdToMediaItem` map and associated with the `APP_BROWSABLE_ROOT` parent.

```
None
fun buildAppRootList(
   context: Context,
   mediaIdToChildren: MutableMap<String, MutableList<MediaItem>>,
   mediaIdToMediaItem: MutableMap<String, MediaItem>,
): List<MediaItem> {
   val rootList = mediaIdToChildren[APP_BROWSABLE_ROOT] ?:
mutableListOf()

   rootList += createStaticNode(
       APP_HOME_ROOT,
       R.string.android_auto_home,
       R.drawable.ic_menu_home_aa,
       context,
   )

   rootList += createStaticNode(
       APP_RECENTLY_PLAYED_ROOT,
       R.string.android_auto_recently_played,
       R.drawable.ic_recently_played,
       context,
   )

   rootList += createStaticNode(
       APP_LIBRARY_ROOT,
       R.string.android_auto_library,
       R.drawable.ic_menu_library_aa,
       context,
   )

   rootList.forEach {
       mediaIdToMediaItem[it.mediaId] = it
   }

   mediaIdToChildren[APP_BROWSABLE_ROOT] = rootList
```

```
    return rootList
}
```

MediaItemUtils.kt

The file contains multiple helper functions that generate `MediaItem` instances based on different use cases. The key components include:

- **Playable Media Items** (for actual audio tracks)
- **Browsable Media Items** (for grouping items like folders or categories)
- **Static Nodes** (non-playable UI elements)
- **Unknown Media Items** (fallbacks for unrecognized items)

Each function sets appropriate **metadata**, such as title, subtitle, artwork, and playback properties.

Class Example Functions

createPlayableMediaItem()

This function generates `MediaItem` instances for actual **audio tracks** that can be played.

What it does:

- Set **mediaId** (unique identifier for the track).
- Set **audio URI** (source URL for playback).
- Set **metadata** (title, subtitle, artist, artwork).
- Mark `setIsPlayable(true)` to indicate that this item can be played.

```
None
fun createPlayableMediaItem(
    id: Int?,
```

```
    title: String?,
    subtitle: String?,
    imageUrl: String?,
    gcpUrl: String?,
    type: String?,
    track: Track,
): MediaItem {
    val extras = Bundle()
    extras.putString(EXTRA_ITEM_TYPE, type)
    extras.putParcelable(EXTRAS_TRACK, track)

    val audioUri = Uri.parse(gcpUrl)
    val metadataBuilder = MediaMetadata.Builder()
        .setTitle(title)
        .setSubtitle(subtitle)
        .setArtist(subtitle)
        .setIsBrowsable(false)
        .setIsPlayable(true)
        .setExtras(extras)
        .setArtworkUri(imageUrl.parseUri())

    return MediaItem.Builder()
        .setMediaId(id.toString().toUniqueId(type!!))
        .setUri(audioUri)
        .setMimeType(MimeTypes.AUDIO_MPEG)
        .setMediaMetadata(metadataBuilder.build())
        .build()
}
```

Important note:

None

```
    val extras = Bundle()

    extras.putString(EXTRA_ITEM_TYPE, type)

    extras.putParcelable(EXTRAS_TRACK, track)
```

This part while creating a playable Media Item in my implementation is crucial since it creates a `Bundle` and stores the domain track's type(track-episode) and the actual object in a two key-value pairs as follows:

1. `extras.putString(EXTRA_ITEM_TYPE, type)`:
    - Stores the `type` of the media item as a `String` in the `Bundle`.
    - `EXTRA_ITEM_TYPE` is a constant (likely a key for identifying the media item's type).
    - The `type` might represent whether the media item is a song, podcast, radio stream, etc.
2. `extras.putParcelable(EXTRAS_TRACK, track)`:
    - Stores the `track` object as a `Parcelable` in the `Bundle`.
    - `EXTRAS_TRACK` is a constant key used to retrieve the track later.
    - `track` is an instance of the `Track` class, which implements `Parcelable`.
    - This allows the media item to carry detailed track information, such as title, artist, duration, and other metadata.

This `extras` bundle is then passed to the `MediaMetadata.Builder` or `MediaItem.Builder`, which helps Android's media framework understand and manage the media content with additional metadata.

2. CreateBrowsableMediaItem

This function generates **folder-like items(parent nodes in the tree such as Podcasts)** that group tracks.

What it does:

- Mark `setIsBrowsable(true)` to indicate this is a folder.
- Assign **folderType** (`FOLDER_TYPE_MIXED`) to classify it.
- Store **metadata** like title, subtitle, and artwork.
- These items cannot be played (`setIsPlayable(false)`) but can be expanded to reveal playable items.

3. `createPlayableMediaItemWithSubgroupHeading`

This function creates groups of related media items together under a title.

```
None
fun createPlayableMediaItemWithSubgroupHeading(
   mediaId: Int?,
```

```
    folderName: String,
    title: String?,
    subtitle: String?,
    imageUrl: String?,
    gcpUrl: String?,
    type: String?,
    track: Track,
): MediaItem {
    val extras = Bundle()
    extras.putString(EXTRA_ITEM_TYPE, type)
    extras.putParcelable(EXTRAS_TRACK, track)

    val audioUri = Uri.parse(gcpUrl)
    val uniqueId = mediaId.toString().toUniqueId(type!!)

    val mediaDescriptionBuilder = MediaDescription.Builder()
    mediaDescriptionBuilder.setMediaId(uniqueId)
    mediaDescriptionBuilder.setTitle(folderName)
    mediaDescriptionBuilder.setIconUri(imageUrl.parseUri())

    extras.putString(
        MediaConstants.DESCRIPTION_EXTRAS_KEY_CONTENT_STYLE_GROUP_TITLE,
        folderName
    )
    mediaDescriptionBuilder.setExtras(extras)

    val metadataBuilder = MediaMetadata.Builder()
        .setTitle(title)
        .setSubtitle(subtitle)
        .setArtist(subtitle)
        .setIsBrowsable(false)
        .setIsPlayable(true)
        .setExtras(extras)
        .setArtworkUri(imageUrl.parseUri())

    return MediaItem.Builder()
        .setMediaId(uniqueId)
        .setUri(audioUri)
        .setMimeType(MimeTypes.AUDIO_MPEG)
        .setMediaMetadata(metadataBuilder.build())
        .build()
}
```

Group media items under a title *Official ref(Group items using title hints)*

To group related media items together, you use a per-item hint. Every media item in a group needs to declare an extras bundle in their MediaDescription that includes a mapping with the key DESCRIPTION_EXTRAS_KEY_CONTENT_STYLE_GROUP_TITLE and an identical string value. Localize this string, which is used as the title of the group.

Loading Artwork in MediaItem

⚠️Official ref -> did not work with me and the working version was extremely heavy and sluggish!

In any create media item util function , artwork is loaded and assigned to a `MediaItem` using the `imageUrl.parseUri()` function. This ensures that a valid image is always available, even if the provided URL is null or invalid.

1. Setting the Icon for the Media Description

```
None
mediaDescriptionBuilder.setIconUri(imageUrl.parseUri())
```

- This sets a small icon for the media item, which is commonly used in media lists, notifications, or compact UI elements.

2. Setting the Artwork for Media Metadata

```
None
.setArtworkUri(imageUrl.parseUri())
```

- This assigns the artwork URI to the `MediaMetadata` object, which is typically used for displaying full-size album artwork in media players.

3. Handling Null or Invalid Image URLs

The `parseUri()` function is responsible for converting the `imageUrl` string into a `Uri`.

```
None
private fun String?.parseUri(): Uri {
    return try {
        if (this == null) return getPlaceholderImageUri() else Uri.parse(this)
    } catch (e: Exception) {
        e.printStackTrace()
          getPlaceholderImageUri()
     }
}
```

- If `imageUrl` is `null` or an invalid string, it falls back to `getPlaceholderImageUri()`, ensuring a valid image is always set.

This approach guarantees that a `MediaItem` always has an associated artwork, preventing UI issues that might arise from missing images.

I read a placeholder image url from a Firebase remote config variable.

Function: `parseUri`

Description:

The `parseUri` function is an extension function for `String?`, primarily used for handling `imageUrl` values. It attempts to convert the string into a `Uri`. If the string is `null` or an exception occurs during parsing, it returns a placeholder image URI instead.

Function Implementation:

```
None
private fun String?.parseUri(): Uri {
    return try {
        if (this == null) return getPlaceholderImageUri() else Uri.parse(this)
    } catch (e: Exception) {
        e.printStackTrace()
```

```
        getPlaceholderImageUri()

    }

}
```

Parameters:

- `this: String?` - A nullable string representing a potential URI, usually an `imageUrl`.

Return Value:

- A `Uri` object parsed from the given string.
- If the string is `null` or an invalid URI, a default placeholder image URI is returned.

Usage Example:

None

```
val imageUrl: String? = "https://example.com/image.jpg"

val imageUri: Uri = imageUrl.parseUri()
```

Error Handling:

- If `this` is `null`, the function directly returns `getPlaceholderImageUri()`.
- If parsing fails (e.g., due to an invalid URL format), an exception is caught, printed to the log, and `getPlaceholderImageUri()` is returned instead.

Purpose:

- Ensures that the application always has a valid `Uri`, even if an image URL is missing or malformed.
- Prevents crashes due to invalid URI parsing.
- Provides a consistent fallback mechanism for image handling.

AndroidAutoStateRepository class

Overview

The `AndroidAutoStateRepository` is a **singleton** responsible for maintaining the global state of media playback, specifically for Android Auto integration. It enables media items to be played in a queue rather than a single-track queue and ensures synchronization between the Android Auto queue and the mobile device.

Purpose

- **Queue Management:** Maintains a global queue of media items and tracks to ensure that playback remains consistent.
- **State Synchronization:** Ensures that when a track is played through Android Auto, the mobile app's queue is updated accordingly.
- **Playback Parameter Management:** Stores and updates `MediaPlaybackParams`, which is collected in the `MainActivity` to manage playback on the device.

Properties

- `_mediaPlaybackParams`: A `MutableStateFlow` that holds the current media playback parameters, including the queue, track index, and playback position.
- `mediaPlaybackParams`: A `StateFlow` providing read-only access to `_mediaPlaybackParams`.
- `_globalMediaItems`: A `MutableStateFlow` that holds the list of media items currently in the queue.
- `globalMediaItems`: A `StateFlow` providing read-only access to `_globalMediaItems`.
- `_globalTracks`: A `MutableStateFlow` storing the list of track objects in the queue.
- `globalTracks`: A `StateFlow` providing read-only access to `_globalTracks`.

Methods

- `setMediaPlaybackParams(params: MediaPlaybackParams)`:
    - Updates `_mediaPlaybackParams` with the provided playback parameters.
    - Logs the updated playback parameters for debugging.

- `setGlobalMediaItems(items: MutableList<MediaItem>)`:
    - Updates `_globalMediaItems` with a new list of media items.
    - Private function to restrict direct access.
- `setGlobalTracks(tracks: MutableList<Track>)`:
    - Updates `_globalTracks` with a new list of tracks.
    - Private function to restrict direct access.
- `setGlobalState(items: MutableList<MediaItem>, tracks: MutableList<Track>)`:
    - Updates both `_globalMediaItems` and `_globalTracks` at the same time to keep them in sync.

Code

None

```
/**
* This is a Singleton to keep global state of the media playback of
Android Auto
* to be able to play media items in a queue instead of one-track queue.
*
* Also to reflect the AA queue onto the mobile phone.
* Whenever AA calls an API that returns a list of tracks, we keep it
* in [_globalMediaItems] & [_globalTracks] so it's always updated with
the last
* opened screen.
*
* Then when the user clicks a track to play, it's played
* inside the global queue here THEN the new queue is updated on the
device
* via [setMediaPlaybackParams] which updates [mediaPlaybackParams]
* that is collected in MainActivity where we use the player viewmodel
* to update phone's queue.
*/
object AndroidAutoStateRepository {
   private val _mediaPlaybackParams = MutableStateFlow(
       MediaPlaybackParams(
           mutableListOf(), mutableListOf(), 0, true, 0L
```

```
        )
    )

    val mediaPlaybackParams: StateFlow<MediaPlaybackParams> get() =
_mediaPlaybackParams

    fun setMediaPlaybackParams(params: MediaPlaybackParams) {
        Log.d("AndroidAutoStateRepository", "setMediaPlaybackParams:
$params")
        _mediaPlaybackParams.value = params
    }

    private val _globalMediaItems:
MutableStateFlow<MutableList<MediaItem>> =
MutableStateFlow(mutableListOf())
    val globalMediaItems: StateFlow<MutableList<MediaItem>> get() =
_globalMediaItems

    private fun setGlobalMediaItems(items: MutableList<MediaItem>) {
        _globalMediaItems.value = items
    }

    private val _globalTracks: MutableStateFlow<MutableList<Track>> =
MutableStateFlow(mutableListOf())
    val globalTracks: StateFlow<MutableList<Track>> get() = _globalTracks

    private fun setGlobalTracks(tracks: MutableList<Track>) {
        _globalTracks.value = tracks
    }

    fun setGlobalState(items: MutableList<MediaItem>, tracks:
MutableList<Track>) {
        setGlobalMediaItems(items)
        setGlobalTracks(tracks)
    }

}
```

Usage

- **Singleton for Media Playback State:** Maintains the global state of media playback for Android Auto.
- **Queue Management:** Allows playing media items in a queue instead of a single-track queue.
- **Sync with Mobile Device:** Ensures that the Android Auto queue is reflected on the mobile phone.
- **Tracks API Handling:** Stores the latest list of tracks from Android Auto API calls in `_globalMediaItems` and `_globalTracks`.
- **Playback Handling:** When a track is selected for playback, it is played within the global queue.
- **Queue Update on Device:** Updates the device queue via `setMediaPlaybackParams`, which modifies `mediaPlaybackParams`.
- **MainActivity Integration:** `mediaPlaybackParams` is collected in MainActivity, where the player ViewModel updates the phone's queue accordingly.

Logging

- The `setMediaPlaybackParams` function logs changes to the playback parameters, which aids in debugging and tracking updates.

MediaPlaybackParams Data Class

Overview

The `MediaPlaybackParams` data class is used to manage media playback state in the application, particularly for handling playback in queues for Android Auto. It stores details about the currently playing media items, track list, playback position, and playback behavior.

Properties

1. mediaItems (MutableList)
    - Holds the list of media items currently queued for playback.
    - Used to manage the media queue for seamless playback.
2. tracks (MutableList)
    - Stores the corresponding track objects related to the media items.

    - Ensures track metadata is available alongside media items.
3. pos (Int)
    - Represents the position of the currently playing track in the queue.
    - Helps in tracking which media item is currently active.
4. playNow (Boolean, default = true)
    - Determines if playback should start immediately upon setting the playback parameters.
    - Default value is `true`, meaning playback starts instantly unless explicitly set otherwise.
5. startPositionMs (Long)
    - Specifies the start position of playback in milliseconds.
    - Useful for resuming playback from a specific timestamp.

Code

```
None
data class MediaPlaybackParams(
   val mediaItems: MutableList<MediaItem>,
   val tracks: MutableList<Track>,
   val pos: Int,
   val playNow: Boolean = true,
   val startPositionMs: Long
)
```

Usage

The `MediaPlaybackParams` class is primarily used for setting and managing playback in `AndroidAutoStateRepository`, ensuring the correct media queue and playback behavior is maintained across different components of the application.